Strategy in R

Game Theory, Simulation, and Machine Intelligence

Raban Heller

2026-11-14

Contents

Welcome! **5**

About this book 5

How to use this book 6

License 6

Citing this Guide 6

Impressum **9**

Suggested citation 9

Disclaimer 9

Contact 9

How to use this book **11**

Reading paths 11

Chapter conventions 11

Companion code and data 11

Downloads 12

How to cite 12

Notation **13**

Common abbreviations 13

I Foundations **15**

1 Introduction: Why Strategy in R? **17**

1.1 The computational turn in game theory 17

1.2 Why R? . . . 17
1.3 The structure of this book . . . 18
1.4 How to use this book . . . 18

2 What is a Game? 19

Learning objectives . . . 19
2.1 Motivation . . . 19
2.2 Theory . . . 20
2.2.1 The four building blocks . . . 20
2.2.2 What makes a situation a “game”? . . . 21
2.2.3 Three canonical examples . . . 21
2.3 Implementation in R . . . 22
2.4 Worked example . . . 23
2.5 Extensions . . . 25
Exercises . . . 25

3 Normal-Form Games 27

Learning objectives . . . 27
3.1 Motivation . . . 27
3.2 Theory . . . 28
3.2.1 The normal-form representation . . . 28
3.2.2 Dominance . . . 28
3.2.3 Iterated elimination of strictly dominated strategies (IESDS) . . . 29
3.2.4 Classic 2x2 games . . . 29
3.3 Implementation in R . . . 29
3.3.1 Checking for strict dominance . . . 30
3.3.2 Publication figure: classic 2x2 payoff structures . . . 31
3.4 Worked example . . . 32
3.4.1 Displaying the full payoff matrix with `gt` . . . 38
3.5 Extensions . . . 38
Exercises . . . 39

4 Nash Equilibrium **41**

Learning objectives . . . 41

4.1 Motivation . . . 41

4.2 Theory . . . 42

4.2.1 Best responses . . . 42

4.2.2 Nash equilibrium defined . . . 42

4.2.3 Existence . . . 42

4.2.4 Multiplicity . . . 43

4.3 Implementation in R . . . 43

4.3.1 Finding pure-strategy Nash equilibria . . . 43

4.3.2 Battle of the Sexes — multiple equilibria . . . 44

4.3.3 Best-response correspondence plot . . . 46

4.4 Worked example . . . 47

4.5 Extensions . . . 50

Exercises . . . 50

5 Mixed Strategies and the Indifference Principle **51**

Learning objectives . . . 51

5.1 Motivation . . . 51

5.2 Theory . . . 52

5.2.1 Mixed strategies defined . . . 52

5.2.2 Expected payoffs under mixing . . . 52

5.2.3 The indifference principle . . . 53

5.2.4 Computing mixed NE in 2x2 games . . . 53

5.2.5 Nash's existence theorem . . . 54

5.3 Implementation in R . . . 54

5.3.1 Computing mixed equilibria with `solve_2x2_mixed_nash()` 54

5.3.2 Visualizing the indifference point . . . 55

5.4 Worked example . . . 57

5.4.1 Matching Pennies: step-by-step . . . 57

5.5 Extensions . . . 59

Exercises . . . 59

6 Extensive-Form Games **61**

Learning objectives . . . 61

6.1 Motivation . . . 61

6.2 Theory . . . 62

6.2.1 Game trees . . . 62

6.2.2 Strategies in extensive form vs. normal form . . . 62

6.2.3 Backward induction . . . 63

6.2.4 Subgame perfect equilibrium . . . 63

6.3 Implementation in R . . . 64

6.3.1 Defining the game tree . . . 64

6.3.2 Drawing the game tree . . . 64

6.3.3 Backward induction algorithm . . . 66

6.4 Worked example . . . 68

6.5 Extensions . . . 70

Exercises . . . 71

7 Bayesian Games **73**

Learning objectives . . . 73

7.1 Motivation . . . 73

7.2 Theory . . . 74

7.2.1 The Bayesian game model . . . 74

7.2.2 Bayesian Nash equilibrium . . . 74

7.2.3 Signaling games . . . 75

7.2.4 Separating vs. pooling equilibria . . . 75

7.3 Implementation in R . . . 75

7.3.1 Expected payoffs under each equilibrium type . . . 76

7.3.2 Checking incentive compatibility . . . 78

7.3.3 Publication figure: expected payoffs by type and equilibrium 78

7.4 Worked example . . . 79

7.5 Extensions . . . 81

Exercises . . . 82

8 Repeated Games **83**

Learning objectives . . . 83

8.1 Motivation . . . 83

8.2 Theory . . . 84

8.2.1 The stage game . . . 84

8.2.2 Finite vs. infinite horizon . . . 84

8.2.3 Trigger strategies . . . 85

8.2.4 The Grim Trigger condition . . . 85

8.2.5 The folk theorem . . . 86

8.3 Implementation in R . . . 86

8.3.1 Stage game setup and Grim Trigger analysis . . . 86

8.3.2 Payoff comparison across discount factors . . . 86

8.3.3 Folk theorem payoff region . . . 87

8.3.4 Publication figure: folk theorem payoff region . . . 88

8.4 Worked example . . . 90

8.4.1 Comparing Grim Trigger and Tit-for-Tat . . . 93

8.5 Extensions . . . 94

Exercises . . . 94

9 Cooperative Game Theory **97**

Learning objectives . . . 97

9.1 Motivation . . . 97

9.2 Theory . . . 98

9.2.1 Characteristic function games . . . 98

9.2.2 Imputations . . . 98

9.2.3 The core . . . 99

9.2.4 The Shapley value . . . 99

9.3 Implementation in R . . . 99

9.3.1 Defining a cooperative game . . . 99

9.3.2 Computing the Shapley value . . . 101

9.3.3 Checking the core . . . 102

9.3.4 Publication figure: Shapley value comparison 104
9.3.5 Marginal contributions table 105
9.4 Worked example 106
9.5 Extensions 108
Exercises 108

II The R Toolkit 111

10 The R Environment for Strategic Computation 113
Learning objectives 113
10.1 Motivation 114
10.2 Theory 114
10.2.1 The reproducibility stack 114
10.2.2 Why IDE configuration matters for game theory 115
10.2.3 The shared setup pattern 115
10.3 Implementation in R 116
10.3.1 Project directory layout 116
10.3.2 Benchmarking vectorized vs. loop-based payoff computation 116
10.3.3 The `_common.R` file explained 119
10.3.4 Initializing `renv` 120
10.4 Worked example 121
10.5 Extensions 123
Exercises 123

11 The GameTheory Package 125
Learning objectives 125
11.1 Motivation 125
11.2 Theory 126
11.2.1 Cooperative games and characteristic functions 126
11.2.2 The Shapley value 126
11.2.3 The Banzhaf power index 126
11.2.4 The GameTheory package API 127

11.3 Implementation in R . . . 127
11.3.1 Defining a characteristic function . . . 127
11.3.2 Computing the Shapley value via permutations . . . 128
11.3.3 Shapley value via the analytical formula . . . 130
11.3.4 Computing the Banzhaf power index . . . 131
11.3.5 Comparing power indices across coalition structures . . . 132
11.4 Worked example . . . 133
11.4.1 A simplified EU Council vote . . . 133
11.5 Extensions . . . 136
Exercises . . . 136

12 The CoopGame Package **139**
Learning objectives . . . 139
12.1 Motivation . . . 139
12.2 Theory . . . 140
12.2.1 Characteristic function representation . . . 140
12.2.2 The core . . . 140
12.2.3 The nucleolus . . . 141
12.2.4 The CoopGame package API . . . 141
12.3 Implementation in R . . . 141
12.3.1 Coalition encoding and game definition . . . 141
12.3.2 Computing the core via linear constraints . . . 142
12.3.3 Finding core vertices . . . 143
12.3.4 Computing the nucleolus . . . 144
12.3.5 Shapley value and core visualization . . . 146
12.4 Worked example . . . 149
12.4.1 Step-by-step core membership check . . . 149
12.5 Extensions . . . 151
Exercises . . . 151

13 The gtree Package **153**

Learning objectives . . . 153

13.1 Motivation . . . 153

13.2 Theory . . . 154

13.2.1 The gtree DSL . . . 154

13.2.2 Gambit integration . . . 155

13.2.3 Backward induction for perfect-information games . . . 155

13.2.4 The ultimatum game . . . 156

13.3 Implementation in R . . . 156

13.3.1 A flexible game tree representation . . . 156

13.3.2 Building the ultimatum game tree . . . 157

13.3.3 Visualizing the game tree . . . 158

13.4 Worked example . . . 161

13.4.1 Scaling analysis . . . 164

13.5 Extensions . . . 166

Exercises . . . 167

14 nashpy via reticulate **169**

Learning objectives . . . 169

14.1 Motivation . . . 169

14.2 Theory . . . 170

14.2.1 The reticulate bridge . . . 170

14.2.2 Support enumeration . . . 171

14.2.3 Complexity considerations . . . 172

14.3 Implementation in R . . . 172

14.3.1 General support enumeration solver . . . 172

14.3.2 Testing on the three canonical 2x2 games . . . 174

14.3.3 Cross-validation with the 2x2 solver . . . 177

14.4 Worked example . . . 178

14.4.1 The 2x2 Hawk-Dove game . . . 178

14.4.2 A 3x3 extension: Rock-Paper-Scissors . . . 179

14.4.3 Comparative visualization . . . 180
14.5 Extensions . . . 183
Exercises . . . 183

15 Building Custom Solvers in R 185
Learning objectives . . . 185
15.1 Motivation . . . 185
15.2 Theory . . . 186
15.2.1 Support enumeration (recap) . . . 186
15.2.2 The Lemke–Howson algorithm . . . 186
15.2.3 Fictitious play . . . 187
15.3 Implementation in R . . . 188
15.3.1 Lemke–Howson sketch for 2x2 games . . . 188
15.3.2 Fictitious play implementation . . . 189
15.3.3 Testing on Matching Pennies . . . 191
15.4 Worked example . . . 192
15.4.1 Setting up the game . . . 192
15.4.2 Running fictitious play . . . 193
15.4.3 Publication figure: convergence trajectory . . . 194
15.4.4 Comparing all three approaches . . . 197
15.4.5 Convergence speed across games . . . 199
15.5 Extensions . . . 200
Exercises . . . 201

III Simulation & Evolutionary Dynamics 203

16 Monte Carlo Methods for Games 205
Learning objectives . . . 205
16.1 Motivation . . . 205
16.2 Theory . . . 206
16.2.1 Monte Carlo estimation . . . 206
16.2.2 Convergence rate and confidence intervals . . . 206

16.2.3 Bootstrap confidence intervals . . . 207
16.3 Implementation in R . . . 207
16.3.1 Mixed-strategy payoff estimation . . . 207
16.3.2 Convergence plot . . . 208
16.3.3 Payoff distribution under mixed strategies . . . 210
16.4 Worked example . . . 211
16.4.1 Bootstrap confidence interval . . . 213
16.4.2 Verifying convergence . . . 214
16.5 Extensions . . . 214
Exercises . . . 215

17 Agent-Based Models for Strategic Interaction 217

Learning objectives . . . 217
17.1 Motivation . . . 217
17.2 Theory . . . 218
17.2.1 The Hawk-Dove game . . . 218
17.2.2 Agent-based evolutionary dynamics . . . 218
17.2.3 Connection to replicator dynamics . . . 219
17.3 Implementation in R . . . 219
17.3.1 Core simulation functions . . . 219
17.3.2 Running the ABM . . . 221
17.3.3 Strategy frequency evolution . . . 222
17.3.4 Population fitness over time . . . 223
17.4 Worked example . . . 224
17.4.1 Sensitivity to parameters . . . 226
17.5 Extensions . . . 228
Exercises . . . 228

18 Replicating Axelrod's Tournament **229**

Learning objectives . . . 229

18.1 Motivation . . . 229

18.2 Theory . . . 230

18.2.1 The iterated Prisoner's Dilemma . . . 230

18.2.2 Axelrod's four properties of successful strategies . . . 230

18.3 Implementation in R . . . 231

18.3.1 Strategy definitions . . . 231

18.3.2 Round-robin tournament . . . 232

18.3.3 Visualizing the results . . . 233

18.3.4 Head-to-head heatmap . . . 233

18.4 Worked example . . . 234

18.5 Extensions . . . 237

Exercises . . . 237

19 The Spatial Prisoner's Dilemma **239**

Learning objectives . . . 239

19.1 Motivation . . . 239

19.2 Theory . . . 240

19.2.1 From well-mixed to spatial populations . . . 240

19.2.2 The Nowak–May model . . . 240

19.2.3 Why clusters protect cooperators . . . 241

19.3 Implementation in R . . . 241

19.3.1 Grid initialization and neighbor scoring . . . 241

19.3.2 Update rule: imitate the best neighbor . . . 242

19.3.3 Running the simulation . . . 242

19.3.4 Visualizing a grid snapshot . . . 243

19.4 Worked example . . . 243

19.5 Extensions . . . 247

Exercises . . . 247

20 Replicator Dynamics **249**

Learning objectives . . . 249

20.1 Motivation . . . 249

20.2 Theory . . . 250

20.2.1 The replicator equation . . . 250

20.2.2 Key properties . . . 250

20.2.3 Two example games . . . 251

20.3 Implementation in R . . . 251

20.3.1 Solving with `run_replicator()` . . . 251

20.3.2 Simplex coordinate transform . . . 252

20.3.3 Running RPS dynamics from multiple initial conditions . . . 252

20.3.4 Running Hawk-Dove dynamics . . . 252

20.4 Worked example . . . 253

20.4.1 Rock-Paper-Scissors . . . 253

20.4.2 Hawk-Dove . . . 256

20.4.3 Comparing the two games . . . 257

20.5 Extensions . . . 258

Exercises . . . 258

21 Evolutionarily Stable Strategies **261**

Learning objectives . . . 261

21.1 Motivation . . . 261

21.2 Theory . . . 262

21.2.1 The ESS definition . . . 262

21.2.2 Relationship to Nash equilibrium . . . 262

21.2.3 Invasion dynamics . . . 262

21.3 Implementation in R . . . 263

21.3.1 Checking ESS conditions for a symmetric game . . . 263

21.3.2 Testing on the Hawk-Dove game . . . 264

21.3.3 Invasion fitness landscape . . . 265

21.3.4 Basins of attraction in a 3-strategy game . . . 266

21.4 Worked example . . . 271
21.4.1 Step 1: Check pure strategies . . . 271
21.4.2 Step 2: Interpret the results . . . 271
21.4.3 Step 3: Verify with replicator dynamics . . . 274
21.5 Extensions . . . 274
Exercises . . . 275

22 Games on Networks **277**
Learning objectives . . . 277
22.1 Motivation . . . 277
22.2 Theory . . . 278
22.2.1 Networks as adjacency matrices . . . 278
22.2.2 Games on networks . . . 278
22.2.3 Small-world networks . . . 278
22.3 Implementation in R . . . 278
22.3.1 Network generators . . . 278
22.3.2 Network statistics . . . 280
22.3.3 Circular layout and network visualization . . . 281
22.3.4 Prisoner's Dilemma on networks . . . 283
22.3.5 Comparing cooperation across topologies . . . 283
22.4 Worked example . . . 285
22.5 Extensions . . . 287
Exercises . . . 287

23 Performance Optimization **289**
Learning objectives . . . 289
23.1 Motivation . . . 289
23.2 Theory . . . 290
23.2.1 Why R loops are slow . . . 290
23.2.2 Three levels of optimization . . . 290
23.2.3 The profiling workflow . . . 290
23.3 Implementation in R . . . 291

23.3.1 The baseline: loop-based spatial PD 291
23.3.2 The optimized version: vectorized spatial PD 292
23.3.3 Benchmarking the two implementations 293
23.3.4 Verifying correctness 294
23.3.5 Visualizing the benchmark 295
23.3.6 Scaling behavior 295
23.4 Worked example 298
23.4.1 Step 1: Profile the baseline 298
23.4.2 Step 2: Vectorize payoff computation 298
23.4.3 Step 3: Memory pre-allocation matters 299
23.4.4 When to consider Rcpp 300
23.5 Extensions 301
Exercises 301

IV Strategic AI 303

24 Machine Learning Foundations for Game Theory 305
Learning objectives 305
24.1 Motivation 305
24.2 Theory 306
24.2.1 Supervised learning as a game 306
24.2.2 Online learning and regret 306
24.2.3 Regret matching 307
24.2.4 Feature representation for game-theoretic data 307
24.3 Implementation in R 307
24.3.1 Regret matching for Rock-Paper-Scissors 307
24.3.2 Decision boundary visualization 310
24.4 Worked example 312
24.4.1 Predicting cooperation in the iterated Prisoner's Dilemma 312
24.5 Extensions 315
Exercises 315

25 Reinforcement Learning Foundations **317**

Learning objectives . . . 317

25.1 Motivation . . . 317

25.2 Theory . . . 318

25.2.1 Markov decision processes . . . 318

25.2.2 The Bellman equation . . . 318

25.2.3 Tabular Q-learning . . . 318

25.2.4 SARSA . . . 319

25.2.5 Exploration vs exploitation . . . 319

25.3 Implementation in R . . . 319

25.3.1 Gridworld environment . . . 319

25.3.2 Tabular Q-learning agent . . . 320

25.3.3 SARSA agent . . . 321

25.3.4 Training on the gridworld . . . 322

25.3.5 Exploration comparison: epsilon-greedy vs UCB . . . 325

25.4 Worked example . . . 326

25.5 Extensions . . . 327

Exercises . . . 328

26 Q-Learning in 2-Player Games **329**

Learning objectives . . . 329

26.1 Motivation . . . 329

26.2 Theory . . . 330

26.2.1 Q-learning recap . . . 330

26.2.2 Independent learners . . . 330

26.2.3 Convergence properties . . . 330

26.3 Implementation in R . . . 331

26.3.1 Q-learning agent . . . 331

26.3.2 Running a repeated game . . . 331

26.3.3 Coordination game: convergence . . . 332

26.3.4 Zero-sum game: cycling . . . 334

26.4 Worked example . . . 335
26.5 Extensions . . . 337
Exercises . . . 337

27 Multi-Agent Reinforcement Learning 339
Learning objectives . . . 339
27.1 Motivation . . . 339
27.2 Theory . . . 340
27.2.1 Independent vs. joint-action learners . . . 340
27.2.2 Minimax-Q learning . . . 340
27.2.3 MADDPG: centralized training, decentralized execution . 341
27.2.4 Policy Space Response Oracles (PSRO) . . . 341
27.3 Implementation in R . . . 342
27.3.1 Joint-action learner . . . 342
27.3.2 Minimax-Q solver . . . 342
27.3.3 Minimax-Q agent . . . 343
27.3.4 Simulation: independent vs. joint-action learners . . . 343
27.3.5 Win-rate matrix across strategies . . . 345
27.4 Worked example . . . 349
27.5 Extensions . . . 350
Exercises . . . 351

28 Self-Play and AlphaZero 353
Learning objectives . . . 353
28.1 Motivation . . . 353
28.2 Theory . . . 354
28.2.1 Self-play as a training methodology . . . 354
28.2.2 Monte Carlo Tree Search . . . 354
28.2.3 AlphaZero architecture . . . 355
28.3 Implementation in R . . . 355
28.3.1 Tic-Tac-Toe game engine . . . 355
28.3.2 MCTS implementation . . . 356

28.3.3 Running MCTS . . . 358
28.3.4 MCTS tree visualization . . . 359
28.3.5 Evaluation against a random opponent . . . 361
28.4 Worked example . . . 363
28.5 Extensions . . . 365
Exercises . . . 365

29 Counterfactual Regret Minimization and Poker 367
Learning objectives . . . 367
29.1 Motivation . . . 367
29.2 Theory . . . 368
29.2.1 Imperfect information and information sets . . . 368
29.2.2 Regret matching . . . 368
29.2.3 CFR algorithm . . . 368
29.2.4 Kuhn Poker . . . 369
29.3 Implementation in R . . . 369
29.3.1 Regret matching . . . 369
29.3.2 Kuhn Poker CFR . . . 370
29.3.3 Run CFR . . . 372
29.3.4 Figure 1: Strategy convergence . . . 372
29.3.5 Figure 2: Exploitability . . . 373
29.4 Worked example . . . 374
29.5 Extensions . . . 377
Exercises . . . 377

30 GANs as Minimax Games 379
Learning objectives . . . 379
30.1 Motivation . . . 379
30.2 Theory . . . 380
30.2.1 The minimax formulation . . . 380
30.2.2 Nash equilibrium of the GAN game . . . 380
30.2.3 Optimal discriminator . . . 380

30.2.4 Mode collapse . . . 381
30.3 Implementation in R . . . 381
30.3.1 Target and generator distributions . . . 381
30.3.2 Discriminator on a grid . . . 382
30.3.3 Training loop . . . 382
30.3.4 Figure 1: Generator distribution evolution . . . 384
30.3.5 Figure 2: Discriminator accuracy . . . 385
30.4 Worked example . . . 387
30.5 Extensions . . . 389
Exercises . . . 389

31 LLM Agents and Strategic Reasoning 391

Learning objectives . . . 391
31.1 Motivation . . . 391
31.2 Theory . . . 392
31.2.1 LLMs as game-playing agents . . . 392
31.2.2 Level-k reasoning . . . 392
31.2.3 Cognitive hierarchy . . . 393
31.2.4 The p-beauty contest . . . 393
31.2.5 Bounded rationality and LLMs . . . 393
31.3 Implementation in R . . . 394
31.3.1 Level-k beauty contest simulation . . . 394
31.3.2 Run the simulation . . . 395
31.3.3 Figure 1: Level-k action distributions . . . 395
31.3.4 Figure 2: Strategy performance comparison . . . 396
31.4 Worked example . . . 399
31.5 Extensions . . . 402
Exercises . . . 402

V Applications **405**

32 Auction Theory **407**

Learning objectives . . . 407
32.1 Motivation . . . 407
32.2 Theory . . . 408
32.2.1 Auction formats . . . 408
32.2.2 Dominant strategy in second-price auctions . . . 408
32.2.3 Optimal bidding in first-price auctions . . . 408
32.2.4 Winner's expected payment . . . 409
32.2.5 Revenue equivalence theorem . . . 409
32.3 Implementation in R . . . 409
32.3.1 Auction simulation functions . . . 409
32.3.2 Figure 1: Revenue comparison across auction types . . . 410
32.3.3 Figure 2: Optimal bid function in first-price auction . . . 412
32.4 Worked example . . . 412
32.5 Extensions . . . 415
Exercises . . . 416

33 Mechanism Design **417**

Learning objectives . . . 417
33.1 Motivation . . . 417
33.2 Theory . . . 418
33.2.1 The mechanism design problem . . . 418
33.2.2 The revelation principle . . . 418
33.2.3 Incentive compatibility and individual rationality . . . 418
33.2.4 The VCG mechanism . . . 418
33.3 Implementation in R . . . 419
33.3.1 VCG mechanism for public goods . . . 419
33.3.2 Comparing mechanisms across many instances . . . 420
33.3.3 Figure 1: Social welfare comparison . . . 421
33.3.4 Figure 2: VCG transfer payments by agent . . . 423

33.4 Worked example . . . 423
33.5 Extensions . . . 427
Exercises . . . 427

34 Matching Markets **429**
Learning objectives . . . 429
34.1 Motivation . . . 429
34.2 Theory . . . 430
34.2.1 Stable matching . . . 430
34.2.2 The Gale-Shapley algorithm . . . 430
34.2.3 Strategy-proofness . . . 430
34.3 Implementation in R . . . 431
34.3.1 Gale-Shapley algorithm . . . 431
34.3.2 Helper: compute welfare . . . 432
34.3.3 Figure 1: Matching visualization . . . 433
34.3.4 Figure 2: Proposer advantage . . . 435
34.4 Worked example . . . 436
34.5 Extensions . . . 440
Exercises . . . 441

35 Bargaining Theory **443**
Learning objectives . . . 443
35.1 Motivation . . . 443
35.2 Theory . . . 444
35.2.1 The bargaining problem . . . 444
35.2.2 Nash bargaining solution . . . 444
35.2.3 Asymmetric Nash bargaining . . . 444
35.2.4 Rubinstein alternating-offers model . . . 445
35.3 Implementation in R . . . 445
35.3.1 Nash bargaining solution computation . . . 445
35.3.2 Nash bargaining visualization . . . 446
35.3.3 Rubinstein alternating-offers model . . . 447

35.3.4 Rubinstein equilibrium shares by discount factor 449
35.4 Worked example 450
35.4.1 Labor-management negotiation 450
35.5 Extensions 452
Exercises 453

36 Conflict and Cooperation Models **455**
Learning objectives 455
36.1 Motivation 455
36.2 Theory 456
36.2.1 The Chicken game 456
36.2.2 Stag Hunt 456
36.2.3 Schelling focal points 457
36.2.4 Arms race as iterated Prisoner's Dilemma 457
36.3 Implementation in R 457
36.3.1 Stag Hunt dynamics under varying risk dominance 457
36.3.2 Phase diagram: cooperation vs. defection basins 458
36.3.3 Arms race simulation 459
36.4 Worked example 462
36.4.1 Cuban Missile Crisis as a Chicken game 462
36.5 Extensions 464
Exercises 465

37 Empirical Case Studies **467**
Learning objectives 467
37.1 Motivation 467
37.2 Theory 468
37.2.1 Spectrum auctions 468
37.2.2 Kidney exchange 468
37.2.3 Braess's paradox 468
37.3 Implementation in R 469
37.3.1 Spectrum auction simulation 469

37.3.2 Kidney exchange matching . . . 470
37.3.3 Kidney exchange network visualization . . . 472
37.3.4 Braess's paradox . . . 473
37.3.5 Braess's paradox visualization . . . 477
37.4 Worked example . . . 478
37.4.1 Braess's paradox with a 4-node network . . . 478
37.5 Extensions . . . 480
Exercises . . . 480

VI Ethics & Outlook 481

38 Ethical Frameworks for Strategic AI 483
Learning objectives . . . 483
38.1 Motivation . . . 483
38.2 Theory . . . 484
38.2.1 Three ethical lenses . . . 484
38.2.2 Social welfare functions . . . 484
38.2.3 Impossibility and trade-offs . . . 485
38.3 Implementation in R . . . 485
38.3.1 Grid search over allocations . . . 486
38.3.2 Pareto frontier with welfare optima . . . 487
38.3.3 Welfare comparison across methods . . . 488
38.4 Worked example . . . 490
38.5 Extensions . . . 491
Exercises . . . 491

39 Fairness in Machine Learning 493
Learning objectives . . . 493
39.1 Motivation . . . 493
39.2 Theory . . . 494
39.2.1 Fairness definitions . . . 494
39.2.2 The impossibility theorem . . . 494

39.2.3 Fairness as a multi-stakeholder game . . . 495
39.3 Implementation in R . . . 495
39.3.1 Simulating loan data . . . 495
39.3.2 Unconstrained logistic regression . . . 496
39.3.3 Fairness-penalized logistic regression . . . 496
39.3.4 ROC curves by group . . . 498
39.3.5 Accuracy-fairness trade-off frontier . . . 500
39.4 Worked example . . . 501
39.5 Extensions . . . 503
Exercises . . . 503

40 AI Alignment as a Game **505**
Learning objectives . . . 505
40.1 Motivation . . . 505
40.2 Theory . . . 506
40.2.1 Principal-agent framework . . . 506
40.2.2 Signaling game . . . 506
40.2.3 Corrigibility in repeated games . . . 506
40.3 Implementation in R . . . 507
40.3.1 Signaling game payoffs . . . 507
40.3.2 Equilibrium analysis . . . 508
40.3.3 Signaling game tree visualization . . . 508
40.3.4 Trust dynamics with Bayesian updating . . . 510
40.4 Worked example . . . 512
40.5 Extensions . . . 514
Exercises . . . 515

41 Cooperative AI **517**
Learning objectives . . . 517
41.1 Motivation . . . 517
41.2 Theory . . . 518
41.2.1 Commitment devices . . . 518

41.2.2 Program equilibrium . . . 518
41.2.3 Credible threats and commitment power . . . 519
41.3 Implementation in R . . . 519
41.3.1 Game definitions . . . 519
41.3.2 Simulating commitment vs no commitment . . . 520
41.3.3 Cooperation rates visualization . . . 521
41.3.4 Welfare improvement via Pareto frontier . . . 522
41.4 Worked example . . . 524
41.5 Extensions . . . 528
Exercises . . . 529

42 The Future of Strategy **531**
Learning objectives . . . 531
42.1 Motivation . . . 531
42.2 Theory . . . 532
42.2.1 Computational complexity of game solving . . . 532
42.2.2 Emerging frontiers . . . 532
42.2.3 Open problems in MARL . . . 532
42.2.4 Responsible deployment . . . 533
42.3 Implementation in R . . . 533
42.3.1 Complexity landscape of game-theory problems . . . 533
42.3.2 Timeline of game theory and AI convergence . . . 534
42.4 Worked example . . . 538
42.5 Extensions . . . 540
Exercises . . . 541

A R Refresher **543**
A.1 Data structures . . . 543
A.1.1 Vectors . . . 543
A.1.2 Matrices . . . 544
A.1.3 Lists . . . 545
A.1.4 Data frames and tibbles . . . 545

A.2 Control flow . . . 546
A.2.1 Conditionals . . . 546
A.2.2 Loops . . . 547
A.2.3 The apply family . . . 547
A.3 Functions . . . 548
A.3.1 Defining functions . . . 548
A.3.2 Closures and environments . . . 549
A.3.3 Anonymous functions . . . 549
A.4 Tidyverse basics . . . 550
A.4.1 The pipe operator . . . 550
A.4.2 dplyr verbs . . . 550
A.4.3 Pivoting . . . 552
A.4.4 ggplot2 layers . . . 553
A.5 Useful idioms for this book . . . 553

B Linear Algebra Essentials **557**
B.1 Vectors and vector operations . . . 557
B.2 Matrices . . . 558
B.2.1 Matrix arithmetic in R . . . 558
B.2.2 Key matrix operations . . . 559
B.3 Systems of linear equations . . . 560
B.3.1 Solving $A\mathbf{x} = \mathbf{b}$ . . . 560
B.3.2 Application: mixed Nash equilibrium . . . 560
B.4 Eigenvalues and eigenvectors . . . 561
B.4.1 Stability and the Jacobian . . . 562
B.5 Positive definite matrices . . . 562
B.6 Convex sets and the simplex . . . 563
B.6.1 The probability simplex . . . 563
B.6.2 Convex sets and convex combinations . . . 563
B.6.3 The support of a mixed strategy . . . 564
B.7 Matrix decompositions . . . 564

B.7.1 LU decomposition 564
B.7.2 Singular value decomposition (SVD) 565
B.8 Summary of R linear algebra functions 565

C Probability Essentials **567**
C.1 Sample spaces and events 567
C.2 Discrete distributions 567
C.2.1 Common discrete distributions 567
C.2.2 Probability mass function and CDF 568
C.3 Continuous distributions 569
C.3.1 Common continuous distributions 569
C.3.2 Density, CDF, and quantiles 569
C.4 Expected value and variance 570
C.4.1 Linearity of expectation 571
C.5 Conditional probability 571
C.6 Bayes' theorem 571
C.6.1 Sequential updating 572
C.7 Law of large numbers and Monte Carlo 573
C.7.1 Monte Carlo estimation 574
C.8 Random sampling for games 575
C.9 Summary of R probability functions 576

D Exercise Solutions **577**
Chapter 1: What is a Game? 577
Chapter 2: Normal-Form Games 578
Chapter 3: Nash Equilibrium 579
Chapter 4: Mixed Strategies 579
Chapter 5: Extensive-Form Games 580
Chapter 6: Bayesian Games 581
Chapter 7: Repeated Games 581
Chapter 8: Cooperative Game Theory 582
Additional selected solutions 583

E Glossary **587**

Acknowledgments **591**

Inspiration 591

Use of LLM tools 591

About the author **593**

Colophon **595**

Session info 595

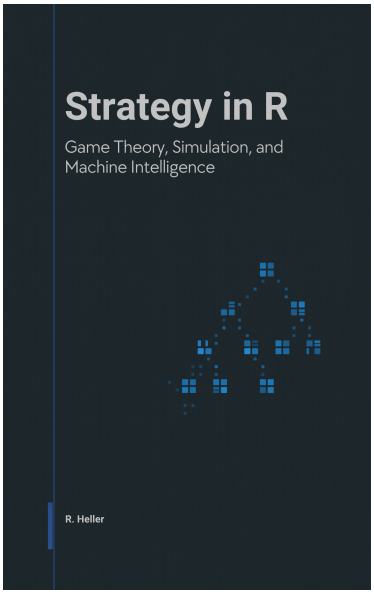
Citing this Guide **597**

References **599**

Welcome!

R. Heller · Date: 2026-05-11 · Last changed: 2026-11-10

A free, open-source book on game theory, simulation, and machine intelligence — implemented end-to-end in R.



About this book

Strategy in R brings two ideas together: the formal language of game theory and the practical computation that makes it tractable. Part I builds the foundations from scratch — normal- and extensive-form games, Nash equilibrium, mixed strategies, Bayesian games, repeated games, and cooperative game theory. Part II tours the R toolkit (`GameTheory`, `CoopGame`, `gtree`, `nashpy` via `reticulate`, custom solvers).

Parts III and IV move into simulation and AI: Monte Carlo methods, agent-based models, Axelrod's tournament, replicator dynamics, evolutionarily stable strategies, network games, performance with Rcpp, then reinforcement learning, multi-agent RL, self-play and AlphaZero, counterfactual regret minimisation, GANs as minimax games, and LLM agents.

Parts V and VI close with applications (auctions, mechanism design, matching markets, bargaining, empirical case studies) and a section on ethics and the future of strategic AI.

The book is for graduate students, researchers, and practitioners with working R fluency who want to apply it to strategic interaction. It assumes no prior game-theory background. It is not a pure-mathematics textbook, not a general machine-learning textbook, and not a software-engineering manual.

How to use this book

- **Reading paths.** Front-to-back works; theorists can skim Parts II–IV; R programmers can dip into Parts II–IV and refer back to Part I as needed.
- **Code lives** under `R/` (helpers, theme, solvers) and `python/` (CFR, deep RL via `reticulate`); example data under `data/`. Every chapter is rebuildable in isolation: `bookdown::preview_chapter("12-gtree-package.Rmd")`.
- **Per-chapter downloads.** Each chapter page offers the chapter as PDF and EPUB below its title.
- **Whole-book downloads.** The PDF and EPUB are linked from the navbar's Download menu.
- **Corrections and suggestions** go to the issue tracker.

License

The content of this book is released under the Creative Commons Attribution 4.0 International License (see `LICENSE-CONTENT`). Source code is released under the MIT License.

Citing this Guide

The suggested citation is:

Heller, R. (2026). *Strategy in R: Game Theory, Simulation, and Machine Intelligence* (Version v0.1.0). Self-published via GitHub Pages. https://cttir.github.io/strategy-in-r/.

Download the reference as BibTeX or RIS.

Impressum

R. Heller · Date: 2026-05-11 · Last changed: 2026-11-09

Strategy in R: Game Theory, Simulation, and Machine Intelligence

© 2026 Raban Heller. ORCID: 0000-0001-8006-9742.

Content released under the Creative Commons Attribution 4.0 International License (CC BY 4.0).

Source code released under the MIT License.

Published online: https://cttir.github.io/strategy-in-r/ Source repository: https://github.com/CTTIR/strategy-in-r

Version: v0.1.0 · Build date: 2026-11-14

Suggested citation

> Heller, R. (2026). *Strategy in R: Game Theory, Simulation, and Machine Intelligence* (Version v0.1.0). https://cttir.github.io/strategy-in-r/.

Disclaimer

This book is provided “as is” without warranty of any kind. The author assumes no responsibility for errors or omissions.

Contact

For corrections, suggestions, or issues: https://github.com/CTTIR/strategy-in-r/issues

How to use this book

R. Heller · Date: 2026-05-11 · Last changed: 2026-11-10

Reading paths

- **Linear** — front-to-back; expects no prior game-theory background.
- **Theorist** — Part I → Part V (applications) → Part VI (ethics); skim Parts II–IV for tools.
- **R programmer** — Part II (toolkit) → Part III (simulation) → dip into Part IV (AI/ML); refer to Part I as a reference.
- **AI researcher** — Part IV → Part VI; treat Part I as a glossary.

Chapter conventions

- Each chapter opens with a concrete question, not a definition.
- Code is shown in full; outputs follow immediately; interpretation is a separate paragraph.
- Cross-references use Bookdown's syntax: a backslash, the at-sign, then `ref(...)` containing a defined label. For example, use `sec-nash-equilibrium` for the Nash equilibrium chapter, `fig:axelrod-ranking` for that figure, or `tab:your-table-label` for a numbered table that you have defined.
- Exercises live at the end of each chapter; solutions are in Appendix D.

Companion code and data

- All R scripts live under `R/`; example datasets under `data/`; Python helpers under `python/`.
- Every chapter is rebuildable in isolation: `bookdown::preview_chapter("12-gtree-package.Rmd")`.

Downloads

- A whole-book PDF and EPUB are linked from the navbar.
- Each chapter page offers the chapter as PDF and EPUB below its title.

How to cite

See the *Impressum* for the suggested citation, and `citation.bib` in the repository for a BibTeX entry.

Notation

R. Heller · Date: 2026-05-11 · Last changed: 2026-05-11

Symbol	Meaning
N	Set of players, $N = \{1, 2, \dots, n\}$
A_i	Action set of player i
$A = \prod_i A_i$	Joint action space
a_{-i}	Action profile of all players except i
$u_i : A \to \mathbb{R}$	Payoff (utility) function of player i
$\sigma_i \in \Delta(A_i)$	Mixed strategy of player i
$\$ \widehat{} * \$$	Nash equilibrium profile
$v(S)$	Characteristic function of coalition S in cooperative games
$\phi_i(v)$	Shapley value of player i
π	Policy in reinforcement learning
$Q(s, a)$	Action–value function
γ	Discount factor
$\mathbb{E}[\cdot]$	Expectation

Common abbreviations

Abbrev.	Expansion
BR	Best response
BNE	Bayesian Nash equilibrium
CFR	Counterfactual regret minimisation
ESS	Evolutionarily stable strategy
MARL	Multi-agent reinforcement learning
MDP	Markov decision process
MSNE	Mixed-strategy Nash equilibrium
NE	Nash equilibrium
PD	Prisoner's dilemma

Abbrev.	Expansion
RL	Reinforcement learning
SPE	Subgame-perfect equilibrium
TU / NTU	Transferable / non-transferable utility
VCG	Vickrey–Clarke–Groves mechanism

Part I

Foundations

Chapter 1

Introduction: Why Strategy in R?

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-09

> Why study strategic interaction computationally, and why R is the right tool for the job.

1.1 The computational turn in game theory

Game theory began as a branch of mathematics. Von Neumann and Morgenstern (1944) laid the foundations; Nash (1950) proved that every finite game has an equilibrium. For decades, the field advanced primarily through theorems and proofs.

But the questions that matter most — *Which equilibrium will players actually reach? How do strategies evolve in populations? Can machines learn to play well?* — are fundamentally computational questions. They require simulation, numerical methods, and algorithm design. This book is about answering those questions with working code.

1.2 Why R?

R is the lingua franca of statistical computing and data science. It has mature ecosystems for visualization (`ggplot2`), data manipulation (`dplyr`), differential equations (`deSolve`), network analysis (`igraph`), and interactive graphics

(`plotly`). For researchers who already think in R, there is no reason to context-switch to another language for game-theoretic computation.

Where R falls short — deep learning, counterfactual regret minimization — we use Python via `reticulate`, keeping the analysis pipeline in R while calling specialized Python libraries.

1.3 The structure of this book

The six parts of this book form a progression:

1. **Foundations** build the conceptual vocabulary: what games are, how to solve them, what equilibrium means.
2. **The R Toolkit** introduces the packages and utilities we use throughout.
3. **Simulation** moves from analytical solutions to computational experiments: Monte Carlo, agent-based models, evolutionary dynamics.
4. **AI and Machine Learning** explores how learning agents interact strategically: reinforcement learning, self-play, and frontier methods.
5. **Applications** grounds the theory in real domains: auctions, matching markets, bargaining, and empirical case studies.
6. **Ethics and the Future** asks what happens when strategic AI systems interact with humans and each other.

Each chapter is self-contained enough to read independently, but the parts are designed to be read in order for a coherent narrative.

1.4 How to use this book

Linear reading is best for graduate courses or self-study in game theory with a computational focus. Start at Part I, Chapter 1.

Reference reading works well for practitioners who need a specific tool. Jump to the relevant chapter, follow cross-references backward for prerequisites.

Project-based reading pairs well with the exercises. Each chapter ends with problems that can serve as starting points for course projects or research extensions.

Let's begin.

Chapter 2

What is a Game?

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

> Players, actions, payoffs, and information — the four building blocks that turn any strategic situation into a formal game. This chapter introduces the modeling stance of game theory, walks through canonical examples such as the Prisoner's Dilemma, coordination, and zero-sum games, and implements a reusable payoff-matrix visualization in R.

Learning objectives

After completing this chapter you will be able to:

- Identify the four core ingredients of a game: players, actions, payoffs, and information.
- Explain why a situation qualifies as a "strategic interaction" rather than a simple decision problem.
- Represent a simultaneous-move game as a payoff matrix and interpret its entries.
- Construct and visualize payoff matrices for classic 2x2 games in R.
- Distinguish between zero-sum, coordination, and social-dilemma games by inspecting their payoff structure.

2.1 Motivation

Imagine two coffee shops that open on the same street. Each owner must set a price without knowing the competitor's choice. The profit of each shop depends

not only on its own price but also on the price set by the other. This mutual dependence is the hallmark of a *strategic interaction*: the outcome for any one participant is shaped by the choices of all participants together.

Situations like this arise everywhere — in economics, politics, biology, computer science, and everyday life. Firms compete on pricing and advertising; countries negotiate treaties; bacteria evolve resistance; autonomous vehicles coordinate at intersections. What unites these disparate settings is a common logical structure that can be captured by a single word: *game*.

Game theory provides a precise mathematical language for reasoning about strategic interactions. It was launched by Neumann & Morgenstern (1944), who demonstrated that parlour games, market competition, and military tactics all share the same formal skeleton. A few years later, Nash (1950) showed that every finite game possesses at least one equilibrium — a combination of strategies from which no player wants to deviate unilaterally. These two contributions created a discipline that now spans the social sciences, biology, and artificial intelligence.

This chapter lays the foundation for everything that follows. Before we can solve games, simulate tournaments, or train learning agents, we need a clear vocabulary for describing what a game *is*.

2.2 Theory

2.2.1 The four building blocks

Following Osborne (2004), a game in *strategic form* is defined by four elements:

1. **Players.** A finite set $N = \{1, 2, \dots, n\}$ of decision-makers. Each player is assumed to be rational — they have well-defined preferences and act to maximize their own payoff.

2. **Actions (strategies).** For each player i, a set A_i of available choices. In a simultaneous-move game every player chooses an action without observing what the others do. The set of all action profiles is $A = A_1 \times A_2 \times \cdots \times A_n$.

3. **Payoffs.** A function $u_i : A \to \mathbb{R}$ that assigns a numerical reward to player i for every possible combination of actions. Payoffs encode preferences: player i prefers action profile a to profile a' if and only if $u_i(a) > u_i(a')$.

4. **Information.** What each player knows when choosing an action. In a simultaneous game, players choose without observing others' moves; in a sequential game, later movers may observe earlier choices. We explore sequential information structures in 6.

A compact way to represent a two-player simultaneous game is the *payoff matrix* (also called the *normal form*). Each row corresponds to an action of Player 1, each column to an action of Player 2, and each cell contains a pair (u_1, u_2) of payoffs. We study this representation in depth in 3.

2.2.2 What makes a situation a "game"?

Not every decision problem is a game. A hiker choosing a trail based on weather forecasts faces uncertainty, but the weather does not "respond" to the hiker's choice. A game requires at least two purposeful agents whose payoffs are mutually dependent. This *interdependence* is the key criterion: if the best action for one player depends on the action chosen by another, the situation is strategic and thus a game (Shoham & Leyton-Brown, 2009).

2.2.3 Three canonical examples

The Prisoner's Dilemma. Two suspects are interrogated separately. Each can *Cooperate* (stay silent) or *Defect* (betray the other). Mutual cooperation yields a moderate reward for both, mutual defection yields a poor outcome for both, and unilateral defection gives the defector a high payoff at the cooperator's expense. The tragedy is that each player has a dominant incentive to defect, even though mutual cooperation would leave both better off. This game is central to the study of cooperation and is revisited in 8 and 18. The payoff structure is:

	Cooperate	Defect
Cooperate	(3, 3)	(0, 5)
Defect	(5, 0)	(1, 1)

Coordination (Stag Hunt). Two hunters can chase a *Stag* (which requires joint effort) or a *Hare* (which can be caught alone). If both hunt the stag, both feast; if one hunts alone, the stag escapes and that hunter gets nothing while the other catches a hare. The game has two pure-strategy equilibria and highlights the tension between efficiency and safety.

	Stag	Hare
Stag	(4, 4)	(0, 3)
Hare	(3, 0)	(2, 2)

Zero-sum (Matching Pennies). Each player secretly shows *Heads* or *Tails*. If the coins match, Player 1 wins; otherwise Player 2 wins. Every gain for one

player is an equal loss for the other, so $u_1(a)+u_2(a) = 0$ for every profile a. This is the class of games Neumann & Morgenstern (1944) first solved completely. We study mixed-strategy solutions in 5.

	Heads	Tails
Heads	(1, -1)	(-1, 1)
Tails	(-1, 1)	(1, -1)

2.3 Implementation in R

We now build an R function to create and visualize any 2x2 payoff matrix. The function returns a tidy data frame suitable for plotting with `ggplot2`.

```
# Build a tidy data frame for a 2x2 payoff matrix
make_payoff_df <- function(game_name, row_actions, col_actions,
 ↪  payoffs) {
  # payoffs: list of four pairs c(u1, u2) in row-major order
  tibble(
    game       = game_name,
    row_action = rep(row_actions, each = length(col_actions)),
    col_action = rep(col_actions, times = length(row_actions)),
    payoff_1   = map_dbl(payoffs, 1),
    payoff_2   = map_dbl(payoffs, 2),
    label      = glue("({payoff_1}, {payoff_2})")
  )
}
```

Next we build data frames for our three canonical games and combine them into one tidy dataset.

```
pd <- make_payoff_df(
  "Prisoner's Dilemma",
  c("Cooperate", "Defect"), c("Cooperate", "Defect"),
  list(c(3, 3), c(0, 5), c(5, 0), c(1, 1))
)

stag <- make_payoff_df(
  "Stag Hunt",
  c("Stag", "Hare"), c("Stag", "Hare"),
  list(c(4, 4), c(0, 3), c(3, 0), c(2, 2))
)
```

```
mp <- make_payoff_df(
  "Matching Pennies",
  c("Heads", "Tails"), c("Heads", "Tails"),
  list(c(1, -1), c(-1, 1), c(-1, 1), c(1, -1))
)

games <- bind_rows(pd, stag, mp) |>
  mutate(
    game = factor(game, levels = c("Prisoner's Dilemma",
                                   "Stag Hunt",
                                   "Matching Pennies"))
  )
```

Now we produce a publication-quality heatmap of the three payoff matrices, using Player 1's payoff for the fill colour.

```
p_heatmap <- ggplot(games, aes(x = col_action, y = row_action,
                               fill = payoff_1)) +
  geom_tile(colour = "white", linewidth = 1.2) +
  geom_text(aes(label = label), size = 3.6, fontface = "bold") +
  facet_wrap(~ game, scales = "free") +
  scale_fill_gradient2(
    low = okabe_ito[6],   # vermillion for negative/low
    mid = "white",
    high = okabe_ito[3],  # bluish-green for positive/high
    midpoint = 1.5,
    name = "Player 1\npayoff"
  ) +
  scale_y_discrete(limits = rev) +
  labs(x = "Player 2", y = "Player 1") +
  theme_publication(base_size = 11) +
  theme(
    strip.text = element_text(face = "bold", size = 11),
    panel.grid = element_blank(),
    axis.ticks = element_blank()
  )

p_heatmap
```

2.4 Worked example

Let us walk through the Prisoner's Dilemma step by step to cement the vocabulary.

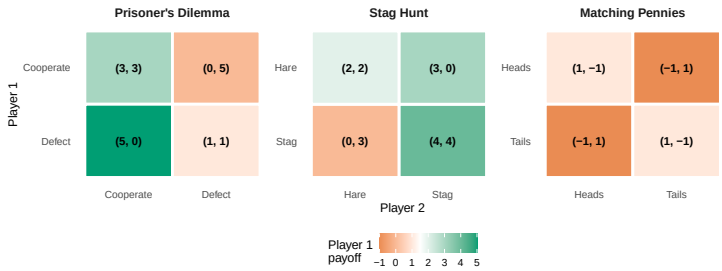


Figure 2.1: Payoff matrices for three canonical 2x2 games. Cell colour encodes Player 1's payoff using a diverging Okabe-Ito-inspired palette; each cell displays both players' payoffs as an ordered pair.

Prisoner's Dilemma Payoffs

Player 1	Player 2	Payoff 1	Payoff 2
Cooperate	Cooperate	3	3
Cooperate	Defect	0	5
Defect	Cooperate	5	0
Defect	Defect	1	1

Step 1 — **Identify the players.** There are two players: Suspect 1 and Suspect 2.

Step 2 — **List the actions.** Each player can *Cooperate* (stay silent) or *Defect* (betray). So $A_1 = A_2 = \{\text{Cooperate}, \text{Defect}\}$.

Step 3 — **Specify payoffs.** We read the matrix row by row:

```
pd |>
  select(
    `Player 1` = row_action,
    `Player 2` = col_action,
    `Payoff 1`  = payoff_1,
    `Payoff 2`  = payoff_2
  ) |>
  gt() |>
  tab_header(title = "Prisoner's Dilemma Payoffs") |>
  cols_align(align = "center")
```

Step 4 — **Check for dominance.** Regardless of what Player 2 does, Player 1 earns more by defecting ($5 > 3$ when Player 2 cooperates; $1 > 0$ when Player

2 defects). Defect is a *dominant strategy* for Player 1. By symmetry, the same holds for Player 2.

Step 5 — Identify the outcome. Both players defect, yielding payoffs (1, 1). This is the unique Nash equilibrium, yet both players would prefer mutual cooperation at (3, 3). The gap between individual rationality and collective welfare is what makes the Prisoner's Dilemma so compelling — and why mechanisms for sustaining cooperation, from repeated interaction (Axelrod, 1981) to enforceable contracts, are among the central concerns of game theory.

2.5 Extensions

The four-element definition introduced here is deliberately minimal. Real-world strategic situations often involve richer structure:

- **Sequential moves and information sets.** When players move in turn and can observe some or all prior actions, we use the *extensive form* — a game tree that encodes timing and information. See 6.
- **Mixed strategies.** Players may randomize over actions. Neumann & Morgenstern (1944) proved that every finite zero-sum game has a value in mixed strategies; Nash (1950) extended the existence result to all finite games. We cover mixed strategies in 5.
- **Incomplete information.** When players are uncertain about others' payoffs or types, we enter the realm of *Bayesian games*, treated in 7.
- **Repeated interaction.** Many strategic encounters happen repeatedly. Repetition can sustain cooperation even in Prisoner's Dilemma settings, a theme explored in 8.
- **Multi-agent learning.** When agents learn from experience rather than computing equilibria analytically, game theory intersects with reinforcement learning (Sutton & Barto, 2018). We develop this connection in 26 and 27.

The compact representation we built in R — the `make_payoff_df()` function and heatmap visualization — will be extended in 3 where we study dominance, best responses, and iterated elimination of dominated strategies.

Exercises

1. **Identifying game elements.** Consider an intersection where two autonomous vehicles arrive simultaneously. Each can *Go* or *Yield*. If both go, they crash (payoff -5 each). If both yield, they waste time (payoff 0 each). If one goes and the other yields, the goer saves time (payoff 3) and the yielder waits (payoff 1). Write down the four elements (players,

actions, payoffs, information) and construct the 2x2 payoff matrix. What type of game is this (zero-sum, coordination, or social dilemma)?

2. **Payoff heatmap.** Using the `make_payoff_df()` function from this chapter, create a payoff matrix for the *Battle of the Sexes* game: two players choose between *Opera* and *Football*. Both prefer to coordinate, but Player 1 prefers Opera (payoffs 3, 2 when both choose Opera) and Player 2 prefers Football (payoffs 2, 3 when both choose Football). Miscoordination yields (0, 0). Plot the heatmap and describe the payoff structure.

3. **Zero-sum verification.** Prove algebraically that Matching Pennies is zero-sum by showing $u_1(a) + u_2(a) = 0$ for every action profile. Then modify the payoffs so that the game is *not* zero-sum but still competitive (Player 1 prefers matching and Player 2 prefers mismatching). Plot both versions side by side using `facet_wrap()`.

4. **From story to model.** Pick a real-world scenario (e.g., two countries deciding whether to impose tariffs, two students deciding how much effort to put into a group project, or two streaming services choosing content investment levels). Identify the players, actions, and a plausible payoff structure. Use `make_payoff_df()` to encode your game and produce the heatmap. Discuss whether the game resembles a Prisoner's Dilemma, coordination game, or something else.

5. **Reading the literature.** Read the first two pages of Nash (1950). In your own words, state Nash's definition of an equilibrium point. How does it relate to the concept of a dominant-strategy outcome in the Prisoner's Dilemma?

Solutions appear in D.

Chapter 3

Normal-Form Games

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

The normal (strategic) form is the workhorse representation of simultaneous-move games. This chapter introduces payoff matrices, strict and weak dominance, iterated elimination of strictly dominated strategies (IESDS), and the four classic 2x2 games that recur throughout game theory.

Learning objectives

- Represent a simultaneous-move game as a payoff matrix and interpret its entries.
- Distinguish strict dominance from weak dominance and identify dominated strategies.
- Apply iterated elimination of strictly dominated strategies (IESDS) to simplify games.
- Recognise and classify the four classic 2x2 games: Prisoner's Dilemma, Stag Hunt, Chicken, and Battle of the Sexes.
- Implement payoff matrices and dominance checks in R and produce publication-quality figures.

3.1 Motivation

Imagine two rival coffee shops opening on the same street, each choosing a price for their latte – high or low – without knowing the other's decision. Each shop's profit depends not only on its own price but also on the competitor's. If both

charge high prices, they share a comfortable market; if one undercuts the other, the cheaper shop captures most customers; if both go low, margins evaporate for everyone.

This kind of simultaneous decision under strategic interdependence is exactly what the **normal-form** (or **strategic-form**) representation was designed to capture. Introduced by Neumann & Morgenstern (1944), the normal form strips a game down to its essentials: who are the players, what can each player do, and what payoff does each combination of choices produce? These elements are arranged in a **payoff matrix**, which makes the strategic structure visible at a glance and is the starting point for almost every solution concept in non-cooperative game theory.

3.2 Theory

3.2.1 The normal-form representation

A finite **normal-form game** is a tuple $G = (N, (S_i)_{i \in N}, (u_i)_{i \in N})$ where

- $N = \{1, 2, \dots, n\}$ is the set of **players**,
- S_i is the finite set of **strategies** (or actions) available to player i, and
- $u_i : S_1 \times \cdots \times S_n \to \mathbb{R}$ is player i's **payoff function**.

For a two-player game where Player 1 has m strategies and Player 2 has k strategies, we arrange payoffs in an $m \times k$ matrix. Each cell (i, j) contains the pair $(u_1(s_i, s_j),\ u_2(s_i, s_j))$. Player 1 chooses a row, Player 2 chooses a column, and the cell at their intersection determines both payoffs.

3.2.2 Dominance

A strategy s_i **strictly dominates** another strategy s_i' if it yields a strictly higher payoff against *every* possible strategy profile of the opponents:

$$u_i(s_i, s_{-i}) > u_i(s_i', s_{-i}) \quad \forall\, s_{-i} \in S_{-i} \tag{3.1}$$

A rational player will never play a strictly dominated strategy – there is always something better, regardless of what others do.

A strategy s_i **weakly dominates** s_i' if it does at least as well against all opponent profiles and strictly better against at least one:

$$u_i(s_i, s_{-i}) \geq u_i(s_i', s_{-i}) \quad \forall\, s_{-i}, \qquad \text{with strict inequality for some } s_{-i}. \tag{3.2}$$

Weak dominance is a less decisive criterion: eliminating weakly dominated strategies can change the set of Nash equilibria and the order of elimination may matter, so it must be applied with care.

3.2.3 Iterated elimination of strictly dominated strategies (IESDS)

If a strategy is strictly dominated, a rational player will never use it. Knowing this, the other players can remove that strategy from consideration, potentially revealing new dominated strategies in the reduced game. Repeating this process is called **IESDS** (iterated elimination of strictly dominated strategies).

A key property of IESDS is **order independence**: the final set of surviving strategies is the same regardless of the order in which dominated strategies are removed (Osborne, 2004, Chapter 2). If IESDS reduces the game to a single strategy profile, the game is **dominance solvable** – the Prisoner's Dilemma is the most famous example.

3.2.4 Classic 2x2 games

Four canonical 2x2 games appear throughout game theory and its applications. Each represents a different type of strategic tension:

- **Prisoner's Dilemma.** Each player has a dominant strategy (Defect), but mutual cooperation would make both better off. The unique Nash equilibrium is Pareto-inefficient.
- **Stag Hunt.** Two pure-strategy equilibria exist: one payoff-dominant (both hunt Stag) and one risk-dominant (both hunt Hare). The game models the trade-off between trust and safety.
- **Chicken (Hawk-Dove).** Two asymmetric pure equilibria (one swerves, the other does not) plus a mixed equilibrium. It captures brinkmanship and anti-coordination.
- **Battle of the Sexes.** Two pure equilibria where players coordinate on different preferred outcomes, plus a mixed equilibrium. It models coordination with conflicting preferences.

These games reappear in 4 for equilibrium analysis, in 5 for mixed-strategy computation, and in 8 for the study of cooperation over time.

3.3 Implementation in R

We begin by building payoff matrices for the four classic games and writing a helper function to check for strictly dominated strategies.

```r
# Define the four classic 2x2 games as named lists
# Each entry: list(A = Player 1 payoffs, B = Player 2 payoffs)
classic_games <- list(
  "Prisoner's Dilemma" = list(
    A = matrix(c(3, 0, 5, 1), nrow = 2, byrow = TRUE,
               dimnames = list(c("C", "D"), c("C", "D"))),
    B = matrix(c(3, 5, 0, 1), nrow = 2, byrow = TRUE,
               dimnames = list(c("C", "D"), c("C", "D")))
  ),
  "Stag Hunt" = list(
    A = matrix(c(4, 0, 3, 3), nrow = 2, byrow = TRUE,
               dimnames = list(c("Stag", "Hare"), c("Stag",
                ↪  "Hare"))),
    B = matrix(c(4, 3, 0, 3), nrow = 2, byrow = TRUE,
               dimnames = list(c("Stag", "Hare"), c("Stag",
                ↪  "Hare")))
  ),
  "Chicken" = list(
    A = matrix(c(0, -1, 1, -5), nrow = 2, byrow = TRUE,
               dimnames = list(c("Swerve", "Straight"),
                ↪  c("Swerve", "Straight"))),
    B = matrix(c(0, 1, -1, -5), nrow = 2, byrow = TRUE,
               dimnames = list(c("Swerve", "Straight"),
                ↪  c("Swerve", "Straight")))
  ),
  "Battle of the Sexes" = list(
    A = matrix(c(3, 0, 0, 2), nrow = 2, byrow = TRUE,
               dimnames = list(c("Opera", "Football"), c("Opera",
                ↪  "Football"))),
    B = matrix(c(2, 0, 0, 3), nrow = 2, byrow = TRUE,
               dimnames = list(c("Opera", "Football"), c("Opera",
                ↪  "Football")))
  )
)
```

3.3.1 Checking for strict dominance

```r
find_strictly_dominated <- function(payoff_matrix) {

  n <- nrow(payoff_matrix)
  dominated <- character(0)
  strats <- rownames(payoff_matrix)
  for (i in seq_len(n)) {
```

```
    for (j in seq_len(n)) {
      if (i != j && all(payoff_matrix[j, ] > payoff_matrix[i, ]))
       ↪  {
        dominated <- c(dominated, strats[i])
      }
    }
  }
  unique(dominated)
}

# Check each classic game for Player 1's dominated strategies
for (name in names(classic_games)) {
  dom <- find_strictly_dominated(classic_games[[name]]$A)
  if (length(dom) > 0) {
    cat(glue("{name}: Player 1's strictly dominated strategy:
     ↪  {dom}"), "\n")
  } else {
    cat(glue("{name}: No strictly dominated strategy for Player
     ↪  1"), "\n")
  }
}
```

```
#> Prisoner's Dilemma: Player 1's strictly dominated strategy: C
#> Stag Hunt: No strictly dominated strategy for Player 1
#> Chicken: No strictly dominated strategy for Player 1
#> Battle of the Sexes: No strictly dominated strategy for Player 1
```

3.3.2 Publication figure: classic 2x2 payoff structures

```
# Convert all four games into a long tibble for faceted plotting
games_long <- purrr::imap_dfr(classic_games, function(game, name)
 ↪  {
  mat <- game$A
  tidyr::expand_grid(
    row = rownames(mat),
    col = colnames(mat)
  ) |>
    mutate(
      payoff = purrr::map2_dbl(row, col, ~ mat[.x, .y]),
      game = name
    )
})
```

```
# Preserve strategy label ordering
games_long <- games_long |>
  mutate(
    row = factor(row, levels = rev(c("C", "D", "Stag", "Hare",
                                     "Swerve", "Straight",
                                     "Opera", "Football"))),
    col = factor(col, levels = c("C", "D", "Stag", "Hare",
                                 "Swerve", "Straight",
                                 "Opera", "Football")),
    game = factor(game, levels = c("Prisoner's Dilemma", "Stag
     ↪  Hunt",
                                   "Chicken", "Battle of the
                                    ↪  Sexes"))
  )

p_classic <- ggplot(games_long, aes(x = col, y = row, fill =
 ↪  payoff)) +
  geom_tile(colour = "white", linewidth = 1.2) +
  geom_text(aes(label = payoff), size = 5, fontface = "bold") +
  facet_wrap(~ game, scales = "free", ncol = 2) +
  scale_fill_gradient2(
    low = okabe_ito[6], mid = okabe_ito[4], high = okabe_ito[3],
    midpoint = 1.5, name = "Player 1\npayoff"
  ) +
  labs(x = "Player 2", y = "Player 1",
       title = "Payoff Structures of Classic 2×2 Games") +
  theme_publication(base_size = 12) +
  theme(
    strip.text = element_text(face = "bold", size = 11),
    axis.text = element_text(size = 10)
  )

p_classic
```

```
save_pub_fig(p_classic, "classic-2x2-payoffs", width = 8, height
 ↪  = 6)
```

3.4 Worked example

We apply IESDS step by step to a 3x3 game to illustrate how the procedure works.

Consider the following game where Player 1 chooses a row (T, M, B) and Player 2 chooses a column (L, C, R):



Figure 3.1: Payoff structures of four classic 2x2 games. Each cell shows Player 1's payoff. The colour gradient highlights the incentive landscape: brighter cells are more attractive for the row player.

```
# Player 1 payoff matrix
A3 <- matrix(
  c(1, 2, 5,
    4, 3, 3,
    3, 0, 2),
  nrow = 3, byrow = TRUE,
  dimnames = list(c("T", "M", "B"), c("L", "C", "R"))
)

# Player 2 payoff matrix
B3 <- matrix(
  c(2, 1, 0,
    1, 4, 1,
    3, 2, 3),
  nrow = 3, byrow = TRUE,
  dimnames = list(c("T", "M", "B"), c("L", "C", "R"))
)

cat("Player 1 payoffs:\n")
```

```
#> Player 1 payoffs:
```

```
A3
```

```
#>   L C R
#> T 1 2 5
#> M 4 3 3
#> B 3 0 2
```

```
cat("\nPlayer 2 payoffs:\n")
```

```
#>
#> Player 2 payoffs:
```

```
B3
```

```
#>   L C R
#> T 2 1 0
#> M 1 4 1
#> B 3 2 3
```

Step 1. Check Player 1's strategies for dominance. Compare row B with row M: M gives $(4, 3, 3)$ versus B's $(3, 0, 2)$. Since $4 > 3$, $3 > 0$, and $3 > 2$, strategy M strictly dominates B. Eliminate B.

```
cat("Step 1: M strictly dominates B for Player 1.\n")
```

```
#> Step 1: M strictly dominates B for Player 1.
```

```
cat("  M payoffs: ", A3["M", ], "\n")
```

```
#>   M payoffs:  4 3 3
```

```
cat("  B payoffs: ", A3["B", ], "\n")
```

```
#>   B payoffs:  3 0 2
```

```
cat("  All M > B? ", all(A3["M", ] > A3["B", ]), "\n")
```

```
#>   All M > B?  TRUE
```

```
# Reduce the game
A_r1 <- A3[c("T", "M"), ]
B_r1 <- B3[c("T", "M"), ]
cat("\nReduced game (Player 1 payoffs):\n")
```

```
#>
#> Reduced game (Player 1 payoffs):
```

```
A_r1
```

```
#>   L C R
#> T 1 2 5
#> M 4 3 3
```

Step 2. In the reduced 2x3 game, check Player 2's strategies. Compare column C with column R for Player 2: C gives $(1, 4)$ versus R's $(0, 1)$. Column C strictly dominates column R. Eliminate R.

```
cat("Step 2: C strictly dominates R for Player 2.\n")
```

```
#> Step 2: C strictly dominates R for Player 2.
```

```
cat("  C payoffs (P2): ", B_r1[, "C"], "\n")
```

```
#>   C payoffs (P2):  1 4
```

```
cat("  R payoffs (P2): ", B_r1[, "R"], "\n")
```

```
#>   R payoffs (P2):  0 1
```

```
cat("  All C > R? ", all(B_r1[, "C"] > B_r1[, "R"]), "\n")
```

```
#>   All C > R?  TRUE
```

```
# Reduce further
A_r2 <- A_r1[, c("L", "C")]
B_r2 <- B_r1[, c("L", "C")]
cat("\nReduced game (Player 1 payoffs):\n")
```

```
#>
#> Reduced game (Player 1 payoffs):
```

```
A_r2
```

```
#>   L C
#> T 1 2
#> M 4 3
```

```
cat("\nReduced game (Player 2 payoffs):\n")
```

```
#>
#> Reduced game (Player 2 payoffs):
```

```
B_r2
```

```
#>   L C
#> T 2 1
#> M 1 4
```

Step 3. In the 2x2 game, check Player 1. Row M gives $(4, 3)$ versus row T's $(1, 2)$. M strictly dominates T. Eliminate T.

```
cat("Step 3: M strictly dominates T for Player 1.\n")
```

```
#> Step 3: M strictly dominates T for Player 1.
```

```
cat("  M payoffs: ", A_r2["M", ], "\n")
```

```
#>   M payoffs:  4 3
```

```
cat("  T payoffs: ", A_r2["T", ], "\n")
```

```
#>   T payoffs:  1 2
```

```
cat("  All M > T? ", all(A_r2["M", ] > A_r2["T", ]), "\n")
```

```
#>   All M > T?  TRUE
```

```
A_r3 <- A_r2["M", , drop = FALSE]
B_r3 <- B_r2["M", , drop = FALSE]
cat("\nReduced game (Player 2 payoffs):\n")
```

```
#>
#> Reduced game (Player 2 payoffs):
```

```
B_r3
```

```
#>   L C
#> M 1 4
```

Step 4. Player 2 now chooses between L (payoff 1) and C (payoff 4). C strictly dominates L. The unique surviving profile is **(M, C)** with payoffs **(3, 4)**.

```
cat("Step 4: C strictly dominates L for Player 2.\n")
```

```
#> Step 4: C strictly dominates L for Player 2.
```

```
cat("  C payoff: ", B_r3[, "C"], "\n")
```

```
#>   C payoff:  4
```

```
cat("  L payoff: ", B_r3[, "L"], "\n")
```

```
#>   L payoff:  1
```

```
cat("\nIESDS solution: (M, C)\n")
```

```
#>
#> IESDS solution: (M, C)
```

```
cat(glue("Payoffs: ({A3['M', 'C']}, {B3['M', 'C']})"), "\n")
```

```
#> Payoffs: (3, 4)
```

The game is **dominance solvable**: IESDS reduces it to a unique strategy profile. This outcome must be a Nash equilibrium – in fact, IESDS can only eliminate strategies that are not part of any Nash equilibrium, so the surviving profile is always a Nash equilibrium (Osborne, 2004).

Player 1 \ Player 2	L	C	R
T	(1, 2)	(2, 1)	(5, 0)
M	(4, 1)	**(3, 4)**	(3, 1)
B	(3, 3)	(0, 2)	(2, 3)

3.4.1 Displaying the full payoff matrix with gt

```
payoff_tbl <- tidyr::expand_grid(
  Row = rownames(A3),
  Col = colnames(A3)
) |>
  mutate(
    label = map2_chr(Row, Col, \(r, c) glue("({A3[r, c]}, {B3[r,
     ↪  c]})")),
    Row = factor(Row, levels = c("T", "M", "B"))
  ) |>
  tidyr::pivot_wider(names_from = Col, values_from = label)

payoff_tbl |>
  gt(rowname_col = "Row") |>
  tab_stubhead(label = "Player 1 \\ Player 2") |>
  tab_style(
    style = cell_fill(color = "#C6EFCE"),
    locations = cells_body(columns = "C", rows = Row == "M")
  ) |>
  tab_style(
    style = cell_text(weight = "bold"),
    locations = cells_body(columns = "C", rows = Row == "M")
  )
```

3.5 Extensions

The normal-form representation introduced here is deliberately simple – two players, finite strategies, simultaneous moves. Several important generalisations follow in later chapters:

- **Nash equilibrium** (4) provides the central solution concept for normal-form games, going beyond dominance to identify self-enforcing strategy profiles.

- **Mixed strategies** (5) extend the framework by allowing players to randomize over actions, guaranteeing equilibrium existence in every finite game.
- **Extensive-form games** (6) enrich the model with sequential moves and information sets, but every extensive-form game can be converted to a normal form – at the cost of an exponential blow-up in the matrix size.
- **Bayesian games** (7) add private information (types) to the normal form, leading to Bayesian Nash equilibrium.

For a rigorous treatment of dominance, IESDS, and the relationship between rationalisability and iterated dominance, see Osborne (2004) (Chapters 2 and 12). The original formulation of normal-form games appears in Neumann & Morgenstern (1944).

Exercises

1. **Identifying dominance.** Consider the following 3x3 game with Player 1 payoffs A and Player 2 payoffs B:

$$A = \begin{pmatrix} 2 & 4 & 1 \\ 3 & 2 & 5 \\ 1 & 3 & 2 \end{pmatrix}, \quad B = \begin{pmatrix} 3 & 1 & 2 \\ 2 & 3 & 1 \\ 4 & 2 & 3 \end{pmatrix}$$

   Identify all strictly dominated strategies for each player. Does any player have a weakly dominated strategy?

2. **IESDS practice.** Apply IESDS to the game in Exercise 1. Show each elimination step and state the surviving strategy profile(s). Is the game dominance solvable?

3. **Classifying 2x2 games.** For each of the following payoff pairs, determine which classic 2x2 game type it represents (Prisoner's Dilemma, Stag Hunt, Chicken, or Battle of the Sexes). Justify your answer by checking the dominance and equilibrium structure.

   a. $A = \begin{pmatrix} 5 & 0 \\ 7 & 2 \end{pmatrix}$, $B = \begin{pmatrix} 5 & 7 \\ 0 & 2 \end{pmatrix}$

   b. $A = \begin{pmatrix} 6 & 1 \\ 4 & 4 \end{pmatrix}$, $B = \begin{pmatrix} 6 & 4 \\ 1 & 4 \end{pmatrix}$

4. **From story to matrix.** Two firms simultaneously choose whether to invest in a new technology (Invest) or stay with the current one (Stay). If both invest, each earns 5. If neither invests, each earns 3. If one invests and the other stays, the investor earns 1 (high cost, no network effect) and the non-investor earns 4 (free-rides on the standard). Write the payoff matrices, identify any dominated strategies, and classify the game.

5. **Weak dominance and order dependence.** Construct a 3x2 game where eliminating weakly dominated strategies in different orders leads to different surviving strategy profiles. Explain why IESDS (strict) does not suffer from this problem.

Solutions appear in D.

Chapter 4

Nash Equilibrium

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

> Definition, existence, computation, and multiplicity of Nash equilibrium — the central solution concept in non-cooperative game theory.

Learning objectives

- Define Nash equilibrium in terms of best responses and unilateral deviations.
- State Nash's existence theorem and understand why mixed strategies guarantee existence.
- Compute pure-strategy and mixed-strategy Nash equilibria for 2×2 games in R.
- Visualize best-response correspondences and identify equilibria graphically.

4.1 Motivation

Consider two competing firms choosing prices simultaneously. Each firm wants to maximize profit, but profit depends on the rival's price. Firm A cannot simply optimize in isolation — it must reason about Firm B's choice, knowing that Firm B is reasoning about Firm A in exactly the same way.

This mutual consistency requirement is the heart of Nash equilibrium: a strategy profile where no player can improve their payoff by changing their strategy alone, given what everyone else is doing. It is the most widely used solution concept in non-cooperative game theory, and the starting point for almost every applied analysis of strategic interaction.

4.2 Theory

4.2.1 Best responses

Given a finite game $G = (N, (A_i)_{i \in N}, (u_i)_{i \in N})$ with player set N, action sets A_i, and payoff functions u_i, player i's **best response** to a strategy profile a_{-i} of the other players is any action that maximizes i's payoff:

$$BR_i(a_{-i}) = \arg\max_{a_i \in A_i} u_i(a_i, a_{-i}) \tag{4.1}$$

4.2.2 Nash equilibrium defined

Definition: Nash Equilibrium

A strategy profile $a^* = (a_1^*, \dots, a_n^*)$ is a **Nash equilibrium** if every player's strategy is a best response to the others:

$$u_i(a_i^*, a_{-i}^*) \geq u_i(a_i, a_{-i}^*) \quad \forall\, a_i \in A_i,\ \forall\, i \in N \tag{4.2}$$

Equivalently, $a_i^* \in BR_i(a_{-i}^*)$ for all i.

The definition captures a stability property: at a Nash equilibrium, no player has a profitable **unilateral** deviation. This does not mean the outcome is socially optimal (the Prisoner's Dilemma shows otherwise), nor that players coordinate easily — it means the profile is self-enforcing once reached.

4.2.3 Existence

Nash (1950) proved that every finite game — one with finitely many players and finitely many actions per player — has at least one Nash equilibrium, possibly in mixed strategies. The proof uses Kakutani's fixed-point theorem applied to the best-response correspondence.

Theorem: Nash's Existence Theorem

Every finite game has at least one Nash equilibrium in mixed strategies.

The theorem does not guarantee a *pure-strategy* equilibrium. The game Matching Pennies, for instance, has no pure-strategy equilibrium but has a unique mixed equilibrium where each player randomizes 50-50 (see 5).

4.2.4 Multiplicity

Games can have zero, one, or many pure-strategy equilibria:

- **Matching Pennies** — zero pure, one mixed.
- **Battle of the Sexes** — two pure, one mixed.
- **Prisoner's Dilemma** — one pure (Defect, Defect), which is also the unique NE overall.

When multiple equilibria exist, selecting among them is a separate problem — the **equilibrium selection** problem — addressed by refinements such as subgame perfection (6), trembling-hand perfection, and risk dominance.

4.3 Implementation in R

We implement two approaches: a pure-R solver for 2×2 games and a visualization of best-response correspondences.

4.3.1 Finding pure-strategy Nash equilibria

```
source(here::here("R", "solvers.R"))

# Prisoner's Dilemma payoff matrices
# Rows = Player 1 actions (C, D), Cols = Player 2 actions (C, D)
A <- matrix(c(3, 0, 5, 1), nrow = 2, byrow = TRUE,
            dimnames = list(c("C", "D"), c("C", "D")))
B <- matrix(c(3, 5, 0, 1), nrow = 2, byrow = TRUE,
            dimnames = list(c("C", "D"), c("C", "D")))

cat("Player 1 payoffs:\n")
```

```
#> Player 1 payoffs:
```

```
A
```

```
#>   C D
#> C 3 0
#> D 5 1
```

```
cat("\nPlayer 2 payoffs:\n")
```

```
#>
#> Player 2 payoffs:
```

```
B
```

```
#>   C D
#> C 3 5
#> D 0 1
```

```
pure_ne <- solve_2x2_pure_nash(A, B)
cat("\nPure-strategy Nash equilibria:\n")
```

```
#>
#> Pure-strategy Nash equilibria:
```

```
for (eq in pure_ne) {
  cat(sprintf("  (%s, %s) with payoffs (%d, %d)\n",
              rownames(A)[eq[1]], colnames(A)[eq[2]],
              A[eq[1], eq[2]], B[eq[1], eq[2]]))
}
```

```
#>   (D, D) with payoffs (1, 1)
```

4.3.2 Battle of the Sexes — multiple equilibria

```
# Battle of the Sexes
A_bos <- matrix(c(3, 0, 0, 2), nrow = 2, byrow = TRUE,
                dimnames = list(c("Opera", "Football"),
                 ↪  c("Opera", "Football")))
B_bos <- matrix(c(2, 0, 0, 3), nrow = 2, byrow = TRUE,
                dimnames = list(c("Opera", "Football"),
                 ↪  c("Opera", "Football")))

cat("Battle of the Sexes:\n")
```

```
#> Battle of the Sexes:
```

```
cat("Player 1 payoffs:\n")
```

```
#> Player 1 payoffs:
```

```
A_bos
```

```
#>          Opera Football
#> Opera        3        0
#> Football     0        2
```

```
cat("\nPlayer 2 payoffs:\n")
```

```
#>
#> Player 2 payoffs:
```

```
B_bos
```

```
#>          Opera Football
#> Opera        2        0
#> Football     0        3
```

```
pure_bos <- solve_2x2_pure_nash(A_bos, B_bos)
mixed_bos <- solve_2x2_mixed_nash(A_bos, B_bos)

cat("\nPure-strategy NE:\n")
```

```
#>
#> Pure-strategy NE:
```

```
for (eq in pure_bos) {
  cat(sprintf("  (%s, %s)\n", rownames(A_bos)[eq[1]],
   ↪  colnames(A_bos)[eq[2]]))
}
```

```
#>   (Opera, Opera)
#>   (Football, Football)
```

```
cat(sprintf("\nMixed-strategy NE: p = %.2f, q = %.2f\n",
            mixed_bos$p, mixed_bos$q))
```

```
#>
#> Mixed-strategy NE: p = 0.60, q = 0.40
```

4.3.3 Best-response correspondence plot

```
# For a 2x2 game, let p = P(Player 1 plays row 1), q = P(Player 2
 ↪  plays col 1)
# Player 2's expected payoff from col 1: p * B[1,1] + (1-p) *
 ↪  B[2,1]
# Player 2's expected payoff from col 2: p * B[1,2] + (1-p) *
 ↪  B[2,2]
# Player 2 is indifferent when these are equal

p_seq <- seq(0, 1, length.out = 200)

# Player 1's best response: BR1(q)
# EU1(row1) = q*A[1,1] + (1-q)*A[1,2] = q*3
# EU1(row2) = q*A[2,1] + (1-q)*A[2,2] = (1-q)*2
# Indifferent when 3q = 2(1-q) => q = 2/5
br1_q <- 2 / (3 + 2)  # q at which P1 is indifferent

# Player 2's best response: BR2(p)
# EU2(col1) = p*B[1,1] + (1-p)*B[2,1] = 2p
# EU2(col2) = p*B[1,2] + (1-p)*B[2,2] = 3(1-p)
# Indifferent when 2p = 3(1-p) => p = 3/5
br2_p <- 3 / (2 + 3)  # p at which P2 is indifferent

# Build best-response data
br_data <- tibble(
  q = c(0, br1_q, br1_q, br1_q, 1),
  p_br1 = c(0, 0, 0.5, 1, 1),
  type = "Player 1 BR"
)

br2_data <- tibble(
  p = c(0, br2_p, br2_p, br2_p, 1),
  q_br2 = c(0, 0, 0.5, 1, 1),
  type = "Player 2 BR"
)
```

```
# Equilibrium points
eq_points <- tibble(
  p = c(0, br2_p, 1),
  q = c(0, br1_q, 1),
  label = c("(Football, Football)", "Mixed NE", "(Opera, Opera)")
)

p_fig <- ggplot() +
  geom_path(data = br_data, aes(x = q, y = p_br1),
            colour = okabe_ito[1], linewidth = 1.2) +
  geom_path(data = br2_data, aes(x = q_br2, y = p),
            colour = okabe_ito[2], linewidth = 1.2) +
  geom_point(data = eq_points, aes(x = q, y = p),
             size = 3, colour = okabe_ito[6]) +
  geom_text(data = eq_points, aes(x = q, y = p, label = label),
            vjust = -1, size = 3) +
  annotate("text", x = 0.85, y = 0.15, label = "BR (q)",
           colour = okabe_ito[1], fontface = "bold", size = 4) +
  annotate("text", x = 0.15, y = 0.85, label = "BR (p)",
           colour = okabe_ito[2], fontface = "bold", size = 4) +
  scale_x_continuous(name = "q  (Player 2: prob. of Opera)",
                     limits = c(-0.05, 1.15), breaks = seq(0, 1,
                      ↪  0.2)) +
  scale_y_continuous(name = "p  (Player 1: prob. of Opera)",
                     limits = c(-0.05, 1.25), breaks = seq(0, 1,
                      ↪  0.2)) +
  theme_publication() +
  labs(title = "Best-Response Correspondences: Battle of the
   ↪  Sexes")

p_fig
```

```
save_pub_fig(p_fig, "best-response-bos")
```

4.4 Worked example

Consider a coordination game where two drivers approach each other on a road and must choose to swerve Left or Right:

```
A_coord <- matrix(c(1, 0, 0, 1), nrow = 2, byrow = TRUE,
                  dimnames = list(c("Left", "Right"), c("Left",
                   ↪  "Right")))
```

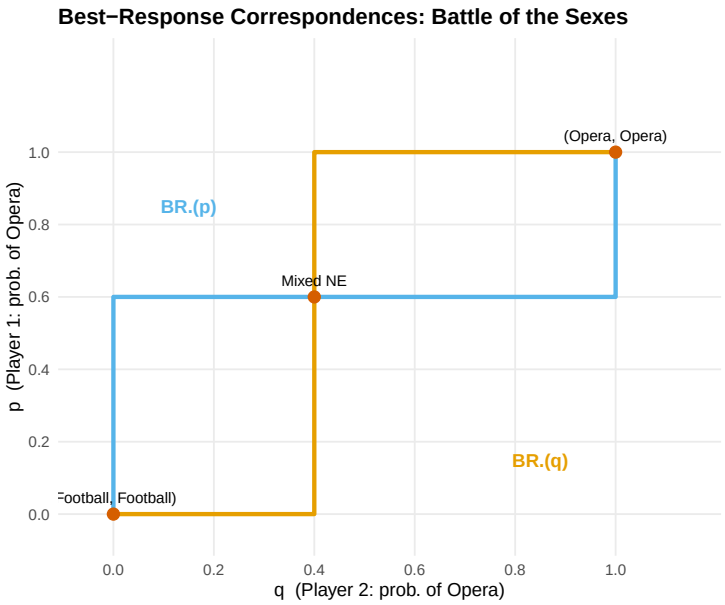


Figure 4.1: Best-response correspondences for the Battle of the Sexes. Player 1's best response (orange) and Player 2's best response (blue) intersect at three Nash equilibria: two pure-strategy and one mixed.

```
B_coord <- A_coord  # symmetric game

cat("Coordination Game (symmetric):\n")
```

```
#> Coordination Game (symmetric):
```

```
A_coord
```

```
#>       Left Right
#> Left     1     0
#> Right    0     1
```

```
eq_coord <- support_enumeration(A_coord, B_coord)

cat("\nPure-strategy NE:\n")
```

```
#>
#> Pure-strategy NE:
```

```
for (eq in eq_coord$pure) {
  cat(sprintf("  (%s, %s)\n", rownames(A_coord)[eq[1]],
    ↪  colnames(A_coord)[eq[2]]))
}
```

```
#>   (Left, Left)
#>   (Right, Right)
```

```
if (!is.null(eq_coord$mixed)) {
  cat(sprintf("\nMixed NE: p = %.2f, q = %.2f\n",
              eq_coord$mixed$p, eq_coord$mixed$q))
  cat("Expected payoff at mixed NE: ",
      eq_coord$mixed$p * eq_coord$mixed$q * 1 +
      (1 - eq_coord$mixed$p) * (1 - eq_coord$mixed$q) * 1, "\n")
}
```

```
#>
#> Mixed NE: p = 0.50, q = 0.50
#> Expected payoff at mixed NE:  0.5
```

The coordination game has two pure NE — (Left, Left) and (Right, Right) — and a mixed NE at $p = q = 0.5$ with expected payoff 0.5, which is worse than either pure equilibrium. This illustrates that mixed equilibria can be Pareto-dominated by pure ones: the randomization creates a risk of miscoordination.

4.5 Extensions

Nash equilibrium is the foundation on which most of the rest of this book builds:

- **Mixed strategies** (5) formalize the randomization used in the mixed NE above.
- **Extensive-form games** (6) introduce sequential moves and **subgame perfect equilibrium**, a refinement of Nash equilibrium.
- **Bayesian games** (7) extend Nash equilibrium to settings with incomplete information.
- **Repeated games** (8) show how the folk theorem expands the set of equilibria when games are played repeatedly.

For a comprehensive treatment of Nash equilibrium and its refinements, see Osborne (2004) (Chapters 2–4).

Exercises

1. **Stag Hunt.** Consider the Stag Hunt game with payoffs: both hunt Stag yields (4, 4), both hunt Hare yields (3, 3), one hunts Stag while the other hunts Hare yields (0, 3) for the Stag hunter and (3, 0) for the Hare hunter. Find all pure-strategy and mixed-strategy Nash equilibria. Which equilibrium is risk-dominant? Which is payoff-dominant?

2. **Three equilibria.** Construct a 2×2 game that has exactly two pure-strategy Nash equilibria and one mixed-strategy Nash equilibrium. Verify your answer using the `support_enumeration()` function from `R/solvers.R`.

3. **Best-response plot.** Modify the best-response correspondence code in 4.3 to plot the correspondences for the game of Chicken (payoffs: (0, 0) for mutual swerve, (−1, 1) and (1, −1) for one swerving, (−5, −5) for neither swerving). How many equilibria does the plot reveal?

Solutions appear in D.

Chapter 5

Mixed Strategies and the Indifference Principle

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

> Why rational players sometimes randomize, how the indifference principle pins down mixing probabilities, and how to compute and visualize mixed-strategy Nash equilibria in R.

Learning objectives

- Explain why pure-strategy equilibria sometimes fail to exist and why randomization resolves the problem.
- State and apply the indifference principle: at a mixed Nash equilibrium, each player's mixture makes every opponent indifferent across the strategies in their support.
- Compute mixed-strategy Nash equilibria for 2x2 games by hand and with `solve_2x2_mixed_nash()`.
- Visualize expected payoffs as a function of mixing probability and identify the indifference point graphically.

5.1 Motivation

Not every strategic situation has a stable deterministic outcome. Consider a penalty kick in football. The striker chooses left or right; the goalkeeper dives left or right. If the goalkeeper could predict the striker's choice, she would always save the shot. If the striker could predict the goalkeeper, he would always score.

Any predictable pattern is exploitable, so both players must be unpredictable — they must *randomize*.

This is not a peculiarity of sports. Firms randomizing the timing of sales, tax authorities choosing which returns to audit, and poker players blending bluffs with genuine bets all face the same logic. Whenever the interests of the players are sufficiently opposed that no pure-strategy profile is self-enforcing (see 4), stability requires mixing.

The profound insight of Nash (1950) is that this resolution is always available: every finite game possesses at least one Nash equilibrium when players are allowed to randomize. This chapter develops the theory behind that guarantee, introduces the *indifference principle* that makes mixed equilibria tractable, and implements the computations in R.

5.2 Theory

5.2.1 Mixed strategies defined

In a normal-form game (3), a **pure strategy** is a definite choice of action. A **mixed strategy** σ_i for player i is a probability distribution over i's action set A_i. We write $\sigma_i(a_i)$ for the probability that player i plays action a_i, with $\sigma_i(a_i) \geq 0$ for all a_i and $\sum_{a_i \in A_i} \sigma_i(a_i) = 1$.

The **support** of a mixed strategy is the set of actions played with positive probability:

$$\text{supp}(\sigma_i) = \{a_i \in A_i : \sigma_i(a_i) > 0\}$$

A pure strategy is a degenerate mixed strategy whose support contains a single action.

5.2.2 Expected payoffs under mixing

When both players use mixed strategies σ_1 and σ_2 in a two-player game, player i's expected payoff is:

$$U_i(\sigma_1, \sigma_2) = \sum_{a_1 \in A_1} \sum_{a_2 \in A_2} \sigma_1(a_1)\, \sigma_2(a_2)\, u_i(a_1, a_2) \tag{5.1}$$

In a 2x2 game, let p denote the probability that Player 1 plays their first action and q the probability that Player 2 plays their first action. Then Player 1's expected payoff is:

$$U_1(p,q) = p\,q\,a_{11} + p\,(1-q)\,a_{12} + (1-p)\,q\,a_{21} + (1-p)\,(1-q)\,a_{22}$$

where a_{ij} are the entries of Player 1's payoff matrix.

5.2.3 The indifference principle

The key to computing mixed equilibria is a simple but initially counter-intuitive observation.

The Indifference Principle

At a mixed-strategy Nash equilibrium, each player's mixture must make every other player *indifferent* among all actions in their support. If any action in the support yielded a strictly higher expected payoff, the player would shift all probability to that action, contradicting the assumption of mixing.

This has a striking implication: player i's equilibrium mixing probability is determined not by i's own payoffs, but by the *opponent's* payoffs. Player i mixes in exactly the proportions needed to keep the opponent from exploiting a deviation.

5.2.4 Computing mixed NE in 2x2 games

Consider a 2x2 game with Player 1's payoff matrix A and Player 2's payoff matrix B. Player 2 mixes with probability q on column 1. Player 1 is indifferent between their two rows when:

$$q \cdot a_{11} + (1-q) \cdot a_{12} = q \cdot a_{21} + (1-q) \cdot a_{22}$$

Solving for q:

$$q^* = \frac{a_{22} - a_{12}}{a_{11} - a_{21} - a_{12} + a_{22}} \tag{5.2}$$

Similarly, Player 1 mixes with probability p on row 1. Player 2 is indifferent when:

$$p^* = \frac{b_{22} - b_{12}}{b_{11} - b_{21} - b_{12} + b_{22}} \tag{5.3}$$

A mixed NE exists when both p^* and q^* fall in $[0,1]$.

5.2.5 Nash's existence theorem

Theorem: Existence of Nash Equilibrium

Every finite game — one with finitely many players and finitely many actions per player — has at least one Nash equilibrium in mixed strategies (Nash, 1950).

The proof relies on Kakutani's fixed-point theorem applied to the best-response correspondences. The importance of this result cannot be overstated: it guarantees that for any well-defined finite game, at least one self-enforcing outcome exists. Without mixed strategies, many natural games (Matching Pennies, Rock-Paper-Scissors) would have no equilibrium at all.

5.3 Implementation in R

5.3.1 Computing mixed equilibria with `solve_2x2_mixed_nash()`

The function `solve_2x2_mixed_nash()` from `R/solvers.R` implements (5.2) and (5.3) directly.

```
# Matching Pennies payoff matrices
A_mp <- matrix(c(1, -1, -1, 1), nrow = 2, byrow = TRUE,
               dimnames = list(c("Heads", "Tails"), c("Heads",
                ↪  "Tails")))
B_mp <- matrix(c(-1, 1, 1, -1), nrow = 2, byrow = TRUE,
               dimnames = list(c("Heads", "Tails"), c("Heads",
                ↪  "Tails")))

cat("Matching Pennies\n")
```

```
#> Matching Pennies
```

```
cat("Player 1 (Matcher) payoffs:\n")
```

```
#> Player 1 (Matcher) payoffs:
```

```
A_mp
```

```
#>       Heads Tails
#> Heads     1    -1
#> Tails    -1     1
```

```
cat("\nPlayer 2 (Mismatcher) payoffs:\n")
```

```
#>
#> Player 2 (Mismatcher) payoffs:
```

```
B_mp
```

```
#>       Heads Tails
#> Heads    -1     1
#> Tails     1    -1
```

```
# Verify no pure-strategy NE exists
pure_ne <- solve_2x2_pure_nash(A_mp, B_mp)
cat("\nPure-strategy NE:", length(pure_ne), "\n")
```

```
#>
#> Pure-strategy NE: 0
```

```
# Compute mixed NE
mixed_ne <- solve_2x2_mixed_nash(A_mp, B_mp)
cat(sprintf("Mixed NE: p = %.2f (P1 plays Heads), q = %.2f (P2
 ↪  plays Heads)\n",
            mixed_ne$p, mixed_ne$q))
```

```
#> Mixed NE: p = 0.50 (P1 plays Heads), q = 0.50 (P2 plays Heads)
```

```
# Expected payoff at mixed NE
eu_mixed <- mixed_ne$p * mixed_ne$q * A_mp[1,1] +
            mixed_ne$p * (1 - mixed_ne$q) * A_mp[1,2] +
            (1 - mixed_ne$p) * mixed_ne$q * A_mp[2,1] +
            (1 - mixed_ne$p) * (1 - mixed_ne$q) * A_mp[2,2]
cat(sprintf("Player 1 expected payoff at mixed NE: %.2f\n",
 ↪  eu_mixed))
```

```
#> Player 1 expected payoff at mixed NE: 0.00
```

5.3.2 Visualizing the indifference point

The figure below plots Player 1's expected payoff from each pure action as Player 2 varies q, the probability of playing Heads. At the indifference point $q^* = 0.5$, both actions yield the same expected payoff, so Player 1 is willing to mix.

```
q_seq <- seq(0, 1, length.out = 200)

# Player 1's expected payoff from Heads: q * 1 + (1-q) * (-1) =
 ↪  2q - 1
# Player 1's expected payoff from Tails: q * (-1) + (1-q) * 1 = 1
 ↪  - 2q
payoff_data <- tibble(
  q = rep(q_seq, 2),
  expected_payoff = c(2 * q_seq - 1, 1 - 2 * q_seq),
  action = rep(c("Heads", "Tails"), each = length(q_seq))
)

# Indifference point
q_star <- 0.5
eu_star <- 2 * q_star - 1  # = 0

p_indiff <- ggplot(payoff_data, aes(x = q, y = expected_payoff,
 ↪  colour = action)) +
  geom_line(linewidth = 1.1) +
  geom_point(
    data = tibble(q = q_star, expected_payoff = eu_star),
    aes(x = q, y = expected_payoff),
    colour = okabe_ito[6], size = 4, inherit.aes = FALSE
  ) +
  annotate("text", x = q_star + 0.12, y = eu_star + 0.12,
           label = paste0("Indifference point\n(q* = ", q_star,
            ↪  ")"),
           colour = okabe_ito[6], size = 3.5, fontface = "bold")
            ↪  +
  geom_vline(xintercept = q_star, linetype = "dashed", colour =
   ↪  "grey50",
             linewidth = 0.5) +
  scale_colour_manual(
    values = c("Heads" = okabe_ito[1], "Tails" = okabe_ito[2]),
    name = "Player 1 action"
  ) +
  scale_x_continuous(name = "q  (Player 2's probability of
   ↪  Heads)",
                     breaks = seq(0, 1, 0.2)) +
  scale_y_continuous(name = "Player 1's expected payoff",
                     breaks = seq(-1, 1, 0.5)) +
  theme_publication() +
  labs(title = "Expected Payoffs and the Indifference Principle")

p_indiff
```

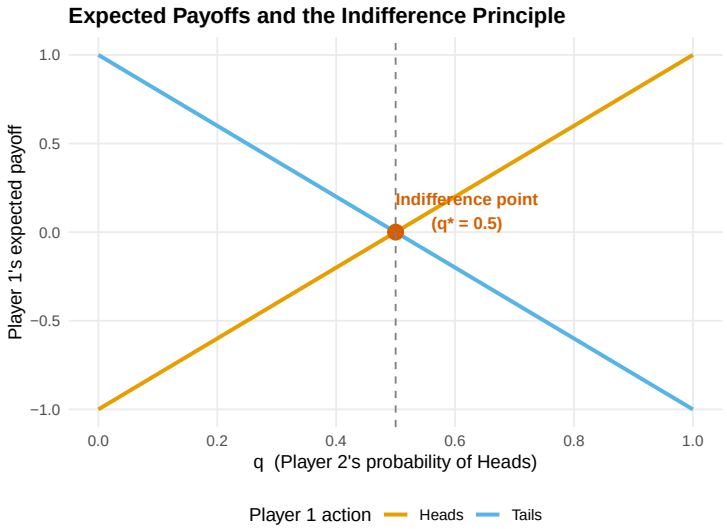


Figure 5.1: Player 1's expected payoff from each pure action as a function of Player 2's mixing probability q in Matching Pennies. The lines cross at $q^* = 0.5$, the indifference point where Player 1 is willing to randomize.

```
save_pub_fig(p_indiff, "indifference-matching-pennies")
```

When $q < 0.5$, Player 2 plays Tails more often, so Player 1 prefers Tails (the orange line is above). When $q > 0.5$, Player 1 prefers Heads (blue line is above). Only at $q^* = 0.5$ is Player 1 genuinely indifferent and therefore willing to randomize. The same logic, applied from Player 2's perspective, pins down $p^* = 0.5$.

5.4 Worked example

5.4.1 Matching Pennies: step-by-step

Two children play Matching Pennies. Each simultaneously shows a coin Heads or Tails. Player 1 (the Matcher) wins if the coins match; Player 2 (the Mismatcher) wins if they differ. The payoff matrices are as defined above.

Step 1: Check for pure-strategy NE. For any pure-strategy profile, one player can profitably deviate. At (Heads, Heads), Player 2 switches to Tails. At (Heads, Tails), Player 1 switches to Tails. No profile is a mutual best response, so no pure NE exists.

Step 2: Apply the indifference principle. Player 1 is indifferent between Heads and Tails when:

$$q \cdot 1 + (1-q) \cdot (-1) = q \cdot (-1) + (1-q) \cdot 1$$
$$2q - 1 = 1 - 2q \implies q^* = \tfrac{1}{2}$$

Player 2 is indifferent when:

$$p \cdot (-1) + (1-p) \cdot 1 = p \cdot 1 + (1-p) \cdot (-1)$$
$$1 - 2p = 2p - 1 \implies p^* = \tfrac{1}{2}$$

Step 3: State the equilibrium. The unique Nash equilibrium is $(p^*, q^*) = (0.5, 0.5)$. Each player randomizes uniformly. The expected payoff for both players is 0.

```
# Verify with R
cat("Verification via solve_2x2_mixed_nash():\n")
```

```
#> Verification via solve_2x2_mixed_nash():
```

```
result <- solve_2x2_mixed_nash(A_mp, B_mp)
cat(sprintf("  p* = %.4f, q* = %.4f\n", result$p, result$q))
```

```
#>   p* = 0.5000, q* = 0.5000
```

```
# Full support enumeration
all_eq <- support_enumeration(A_mp, B_mp)
cat(sprintf("  Pure NE: %d\n", length(all_eq$pure)))
```

```
#>   Pure NE: 0
```

```
cat(sprintf("  Mixed NE: p = %.2f, q = %.2f\n", all_eq$mixed$p,
 ↪  all_eq$mixed$q))
```

```
#>   Mixed NE: p = 0.50, q = 0.50
```

Step 4: Interpret. The equilibrium has a natural interpretation: each player must be maximally unpredictable. Any bias — say, playing Heads 60% of the time — would be exploited by the opponent. The mixed NE is the only self-enforcing outcome, consistent with Nash's existence theorem.

5.5 Extensions

Mixed strategies appear throughout the remainder of this book and in virtually every branch of applied game theory:

- **Extensive-form games** (6) use behavioral strategies — independent mixtures at each information set — which are equivalent to mixed strategies under perfect recall.
- In **Bayesian games** (7), a player's type can be viewed as a private signal that induces mixing from the opponent's perspective, blurring the line between mixed and pure strategies.
- The **minimax theorem** of Neumann & Morgenstern (1944) shows that in two-player zero-sum games, the mixed NE payoff equals the value of the game, and both players can guarantee this value through optimal mixing.
- Support enumeration generalizes beyond 2x2 games. For larger games, algorithms enumerate all possible support pairs and solve the resulting linear systems; see Shoham & Leyton-Brown (2009) for computational details.

For a thorough treatment of mixed strategies and their properties, consult Osborne (2004) (Chapter 4).

Exercises

1. **Asymmetric Matching Pennies.** Consider a variant where Player 1 receives a payoff of 3 (instead of 1) when both play Heads, while all other payoffs remain the same as standard Matching Pennies. Set up the payoff matrices, compute the mixed NE by hand using the indifference principle, and verify with `solve_2x2_mixed_nash()`. How does the asymmetry affect Player 2's mixing probability? Explain the intuition.

2. **Inspection game.** A worker chooses to Shirk or Work; an employer chooses to Monitor or Trust. Payoffs: (Work, Trust) yields (2, 3), (Work, Monitor) yields (1, 2), (Shirk, Trust) yields (3, 0), and (Shirk, Monitor) yields (0, 1). Find the mixed NE. What is the equilibrium probability that the employer monitors? Does increasing the penalty for caught shirking make shirking less likely in equilibrium?

3. **Graphical verification.** Adapt the expected-payoff plot from 5.1 to the Battle of the Sexes game (payoffs as in 4). Plot Player 1's expected payoff from Opera and from Football as functions of q. Identify the indifference point and verify that it matches the mixed NE from `solve_2x2_mixed_nash()`.

4. **Rock-Paper-Scissors.** Rock-Paper-Scissors is a 3x3 symmetric zero-sum game. Argue, using the indifference principle, that the unique mixed NE must assign equal probability $\frac{1}{3}$ to each action. (Hint: symmetry means the indifference conditions are identical for all pairs of actions.)

5. **Comparative statics.** In a 2x2 game with payoff matrices A and B, suppose we increase a_{11} (Player 1's payoff when both play action 1) while holding all other entries fixed. Using (5.2), determine what happens to q^*. Does it increase, decrease, or stay the same? Explain why this result is consistent with the indifference principle — and why it surprises many students.

Solutions appear in D.

Chapter 6

Extensive-Form Games

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

> Game trees, backward induction, and subgame perfect equilibrium — the tools for analyzing sequential strategic interaction where timing and information matter.

Learning objectives

- Represent sequential games as game trees with nodes, branches, and information sets.
- Translate between extensive-form and normal-form representations of a game.
- Apply backward induction to solve finite games of perfect information.
- Define subgame perfect equilibrium and explain why it refines Nash equilibrium.
- Implement backward induction in R and visualize game trees with ggplot2.

6.1 Motivation

The normal-form games introduced in 3 assume that players choose their strategies simultaneously, without observing each other's actions. Many real-world interactions, however, unfold over time. A firm decides whether to enter a market *after* observing an incumbent's investment in capacity. A prosecutor makes a plea offer *before* the defendant decides whether to accept it. A chess player moves *in response* to an opponent's prior move.

When the sequence of moves matters, we need a richer representation than a payoff matrix. The **extensive form** models games as trees, where each node represents a decision point, each branch represents an available action, and the tree's structure encodes who moves when and what they know at the time of their decision. This representation, introduced by Neumann & Morgenstern (1944) and refined by Kuhn (1953), is the standard framework for analyzing sequential strategic interaction.

The key payoff of moving to the extensive form is a sharper solution concept. As we saw in 4, Nash equilibrium can sustain outcomes based on threats that a player would never actually carry out. Backward induction — reasoning from the end of the game back to the beginning — eliminates such non-credible threats and leads to **subgame perfect equilibrium** (SPE), a refinement that requires rational play in every subgame, not just on the equilibrium path.

6.2 Theory

6.2.1 Game trees

An **extensive-form game** consists of:

1. A set of players $N = \{1, 2, \dots, n\}$.
2. A **game tree** — a directed tree with a distinguished root node.
3. A **player function** assigning each non-terminal (decision) node to a player.
4. **Actions** labeling the branches from each decision node.
5. **Information sets** partitioning each player's decision nodes into groups within which the player cannot distinguish (reflecting what the player knows at that point).
6. **Payoff vectors** assigned to each terminal node.

A game has **perfect information** if every information set contains exactly one node — that is, each player knows exactly where they are in the tree when they move. Chess, tic-tac-toe, and the entry deterrence game below are all games of perfect information. When an information set contains multiple nodes, the player faces **imperfect information** about prior moves; this connects to the analysis of Bayesian games in 7.

6.2.2 Strategies in extensive form vs. normal form

A **strategy** in the extensive form is a complete contingent plan: it specifies an action at *every* information set where the player might be called to move, including those that the strategy itself makes unreachable. This is a subtle

but important distinction from the normal form. Two extensive-form strategies that differ only at unreachable information sets map to the same normal-form strategy but can have different implications for subgame perfection.

Any extensive-form game can be converted to a normal-form game (see 3) by enumerating all strategy combinations and computing the resulting payoffs. However, this conversion discards the sequential structure, and Nash equilibria of the normal form may include strategy profiles sustained by non-credible threats.

6.2.3 Backward induction

For finite games of perfect information, **backward induction** provides a constructive algorithm for finding rational play:

1. Start at each terminal node and record the payoffs.
2. Move to the decision nodes immediately preceding the terminal nodes. At each such node, the deciding player chooses the action that maximizes their payoff.
3. Replace each such decision node with the terminal payoff that results from the optimal choice.
4. Repeat until the root is reached.

Theorem: Zermelo–Kuhn

Every finite game of perfect information has at least one subgame perfect equilibrium in pure strategies, which can be found by backward induction (Osborne, 2004, Chapter 5).

6.2.4 Subgame perfect equilibrium

A **subgame** of an extensive-form game is any subtree that (i) begins at a singleton information set, (ii) contains all successors of its root node, and (iii) does not break any information set.

Definition: Subgame Perfect Equilibrium

A strategy profile is a **subgame perfect equilibrium** (SPE) if it induces a Nash equilibrium in every subgame of the game.

Every SPE is a Nash equilibrium, but not every Nash equilibrium is subgame perfect. The additional requirement of rationality in *every* subgame rules out strategies that rely on non-credible threats — actions a player would not actually carry out if called upon to do so. This refinement is the central contribution of Selten (1965) and is fundamental to the analysis of sequential games throughout this book.

6.3 Implementation in R

We implement the entry deterrence game — a classic example from industrial organization — as a game tree using ggplot2, then solve it by backward induction.

6.3.1 Defining the game tree

Consider an **entry deterrence game** with two players:

- **Entrant** (Player 1) chooses to **Enter** or **Stay Out**.
- **Incumbent** (Player 2), if entry occurs, chooses to **Accommodate** or **Fight**.

The payoffs are: if the entrant stays out, the incumbent keeps the monopoly (Entrant: 0, Incumbent: 3). If the entrant enters and the incumbent accommodates, both share the market (Entrant: 2, Incumbent: 1). If the entrant enters and the incumbent fights, both suffer losses (Entrant: -1, Incumbent: -1).

```
# Define the game tree as a data structure
# Each node: id, player, label, x/y position
nodes <- tibble(
  id     = c("root", "inc", "t1", "t2", "t3"),
  player = c("Entrant", "Incumbent", NA, NA, NA),
  label  = c("Entrant", "Incumbent", "(0, 3)", "(2, 1)", "(-1,
   ↪  -1)"),
  x      = c(0, 1.5, -1.5, 2.5, 0.5),
  y      = c(3, 1.5, 1.5, 0, 0),
  is_terminal = c(FALSE, FALSE, TRUE, TRUE, TRUE)
)

# Edges: from, to, action label
edges <- tibble(
  from_x = c(0, 0, 1.5, 1.5),
  from_y = c(3, 3, 1.5, 1.5),
  to_x   = c(1.5, -1.5, 2.5, 0.5),
  to_y   = c(1.5, 1.5, 0, 0),
  action  = c("Enter", "Stay Out", "Accommodate", "Fight")
)
```

6.3.2 Drawing the game tree

```
# Backward induction result: at the incumbent's node, Accommodate
↪  (2,1)
# dominates Fight (-1,-1), so the incumbent accommodates.
# Knowing this, the entrant compares Enter -> (2,1) vs Stay Out
↪  -> (0,3):
# Entrant gets 2 > 0, so Enter is optimal.
# SPE path: Enter -> Accommodate

spe_edges <- tibble(
  from_x = c(0, 1.5),
  from_y = c(3, 1.5),
  to_x   = c(1.5, 2.5),
  to_y   = c(1.5, 0)
)

p_tree <- ggplot() +
  # Draw all branches

  geom_segment(data = edges,
               aes(x = from_x, y = from_y, xend = to_x, yend =
                ↪  to_y),
               colour = "grey60", linewidth = 0.8) +
  # Highlight SPE path

  geom_segment(data = spe_edges,
               aes(x = from_x, y = from_y, xend = to_x, yend =
                ↪  to_y),
               colour = okabe_ito[3], linewidth = 1.6) +
  # Decision nodes
  geom_point(data = filter(nodes, !is_terminal),
             aes(x = x, y = y),
             shape = 21, size = 6, fill = okabe_ito[2],
             colour = "black", stroke = 1.2) +
  # Terminal nodes
  geom_point(data = filter(nodes, is_terminal),
             aes(x = x, y = y),
             shape = 22, size = 5, fill = okabe_ito[1],
             colour = "black", stroke = 1) +
  # Node labels
  geom_text(data = filter(nodes, !is_terminal),
            aes(x = x, y = y, label = label),
            vjust = -1.5, fontface = "bold", size = 3.8) +
  geom_text(data = filter(nodes, is_terminal),
            aes(x = x, y = y, label = label),
            vjust = 1.8, size = 3.5) +
```

```
  # Action labels on edges
  geom_text(data = edges,
            aes(x = (from_x + to_x) / 2,
                y = (from_y + to_y) / 2,
                label = action),
            nudge_x = c(0.5, -0.6, 0.7, -0.5),
            nudge_y = c(0.15, 0.15, 0.1, 0.1),
            size = 3.3, fontface = "italic",
            colour = c(okabe_ito[3], "grey40", okabe_ito[3],
             ↪  "grey40")) +
  # Annotation for backward induction reasoning
  annotate("label", x = 2.8, y = 1.0,
           label = "Incumbent prefers\n1 > -1: Accommodate",
           size = 2.8, fill = "white", label.size = 0.3,
           colour = okabe_ito[5]) +
  annotate("label", x = -1.5, y = 2.8,
           label = "Entrant prefers\n2 > 0: Enter",
           size = 2.8, fill = "white", label.size = 0.3,
           colour = okabe_ito[5]) +
  scale_x_continuous(limits = c(-2.5, 4), expand = c(0, 0)) +
  scale_y_continuous(limits = c(-0.8, 4), expand = c(0, 0)) +
  labs(title = "Backward Induction in the Entry Deterrence Game")
   ↪  +
  theme_publication() +
  theme(
    axis.text  = element_blank(),
    axis.title = element_blank(),
    axis.ticks = element_blank(),
    panel.grid = element_blank()
  )

p_tree
```

```
save_pub_fig(p_tree, "entry-deterrence-backward-induction", width
 ↪  = 7, height = 5)
```

6.3.3 Backward induction algorithm

```
# A general backward induction solver for binary game trees
# encoded as nested lists

backward_induction <- function(node) {
```

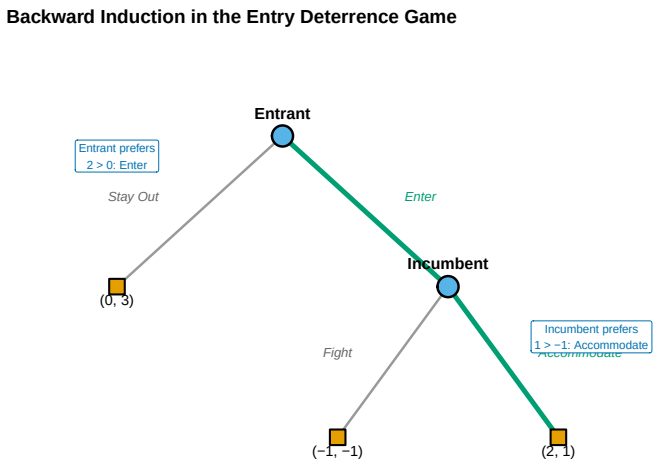


Figure 6.1: Entry deterrence game solved by backward induction. Bold edges show the subgame perfect equilibrium path: the entrant enters and the incumbent accommodates. The struck-through branch (Fight) represents a non-credible threat.

```
# Terminal node: return payoffs
if (node$type == "terminal") {
  return(list(payoffs = node$payoffs, path = list()))
}

# Recursive: solve each child subtree
left  <- backward_induction(node$left)
right <- backward_induction(node$right)

# The moving player picks the branch maximizing their payoff
player_idx <- node$player  # 1 or 2
if (left$payoffs[player_idx] >= right$payoffs[player_idx]) {
  chosen <- left
  action <- node$left_action
} else {
  chosen <- right
  action <- node$right_action
}

list(
  payoffs = chosen$payoffs,
  path = c(list(list(player = player_idx, action = action)),
    ↪  chosen$path)
```

```
  )
}

# Encode the entry deterrence game
entry_game <- list(
  type = "decision", player = 1,
  left_action = "Stay Out",
  left = list(type = "terminal", payoffs = c(0, 3)),
  right_action = "Enter",
  right = list(
    type = "decision", player = 2,
    left_action = "Accommodate",
    left = list(type = "terminal", payoffs = c(2, 1)),
    right_action = "Fight",
    right = list(type = "terminal", payoffs = c(-1, -1))
  )
)

result <- backward_induction(entry_game)
cat("SPE outcome payoffs:", result$payoffs, "\n")
```

```
#> SPE outcome payoffs: 2 1
```

```
cat("SPE path:\n")
```

```
#> SPE path:
```

```
for (step in result$path) {
  cat(sprintf("  Player %d chooses: %s\n", step$player,
   ↪  step$action))
}
```

```
#>   Player 1 chooses: Enter
#>   Player 2 chooses: Accommodate
```

6.4 Worked example

We now trace through the entry deterrence game step by step to see exactly how backward induction eliminates non-credible threats.

Step 1: Identify the last decision node. The incumbent's node is reached only if the entrant enters. At this node, the incumbent has two choices:

- **Accommodate**: payoff to Incumbent = 1
- **Fight**: payoff to Incumbent = -1

Since $1 > -1$, a rational incumbent chooses Accommodate. Fighting is a non-credible threat — the incumbent would never carry it out if actually called upon to decide.

Step 2: Fold back to the first decision node. Knowing that the incumbent will accommodate upon entry, the entrant faces an effective choice between:

- **Stay Out**: payoff to Entrant = 0
- **Enter**: payoff to Entrant = 2 (since the incumbent will accommodate)

Since $2 > 0$, the entrant enters.

Step 3: The subgame perfect equilibrium. The SPE strategy profile is (Enter, Accommodate), yielding payoffs $(2, 1)$.

```
# Convert to normal form to show the Nash equilibrium issue
# Entrant: {Enter, Stay Out}
# Incumbent: {Accommodate, Fight}
A_entry <- matrix(c(2, -1, 0, 0), nrow = 2, byrow = TRUE,
                  dimnames = list(c("Enter", "Stay Out"),
                                  c("Accommodate", "Fight")))
B_entry <- matrix(c(1, -1, 3, 3), nrow = 2, byrow = TRUE,
                  dimnames = list(c("Enter", "Stay Out"),
                                  c("Accommodate", "Fight")))

cat("Normal-form representation:\n")
```

```
#> Normal-form representation:
```

```
cat("Entrant payoffs:\n")
```

```
#> Entrant payoffs:
```

```
A_entry
```

```
#>           Accommodate Fight
#> Enter               2    -1
#> Stay Out            0     0
```

```
cat("\nIncumbent payoffs:\n")
```

```
#>
#> Incumbent payoffs:
```

```
B_entry
```

```
#>          Accommodate Fight
#> Enter              1    -1
#> Stay Out           3     3
```

```
source(here::here("R", "solvers.R"))
ne <- solve_2x2_pure_nash(A_entry, B_entry)
cat("\nPure-strategy Nash equilibria:\n")
```

```
#>
#> Pure-strategy Nash equilibria:
```

```
for (eq in ne) {
  cat(sprintf("  (%s, %s) -> payoffs (%d, %d)\n",
              rownames(A_entry)[eq[1]], colnames(A_entry)[eq[2]],
              A_entry[eq[1], eq[2]], B_entry[eq[1], eq[2]]))
}
```

```
#>   (Enter, Accommodate) -> payoffs (2, 1)
#>   (Stay Out, Fight) -> payoffs (0, 3)
```

The normal form reveals **two** Nash equilibria: (Enter, Accommodate) and (Stay Out, Fight). The second equilibrium involves the incumbent *threatening* to fight if entry occurs — a threat that deters the entrant. But we showed above that this threat is non-credible: if the incumbent actually had to carry it out, they would prefer to accommodate. Subgame perfection eliminates (Stay Out, Fight), leaving only (Enter, Accommodate) as the unique SPE. This is the power of backward induction as a refinement of Nash equilibrium.

6.5 Extensions

The extensive form opens several directions explored later in this book:

- **Bayesian games** (7) extend the extensive form to settings where players have private information about payoffs or types, using information sets to model uncertainty about nature's prior moves.
- **Repeated games** (8) can be viewed as extensive-form games with a particular tree structure, and the folk theorem relies on strategies conditioned on the history of play.
- **Bargaining** (Chapter 34) uses alternating-offer extensive-form games, where Rubinstein's model yields a unique SPE through backward induction with discounting.
- **Imperfect information** generalizes the trees studied here by allowing information sets with multiple nodes, creating a bridge between the extensive form and the normal form.

For the foundational treatment of extensive-form games and backward induction, see Neumann & Morgenstern (1944) and Osborne (2004) (Chapters 5–7). The entry deterrence example follows the tradition of Selten's chain-store paradox, which demonstrates the power of subgame perfection for eliminating non-credible threats in sequential games.

Exercises

1. **Ultimatum game.** Consider a simplified ultimatum game where Player 1 proposes a split of 10 units, choosing either Fair (5, 5) or Greedy (8, 2). Player 2 then accepts or rejects; rejection gives both zero. Draw the game tree, find all Nash equilibria in the normal form, and identify the subgame perfect equilibrium using backward induction. Which Nash equilibria are eliminated by subgame perfection?

2. **Three-stage game.** Three players move in sequence. Player 1 chooses L or R; Player 2 observes this and chooses A or B; Player 3 observes both prior choices and chooses X or Y. Terminal payoffs are: (L, A, X) = (3, 2, 1), (L, A, Y) = (1, 3, 2), (L, B, X) = (0, 1, 4), (L, B, Y) = (2, 0, 3), (R, A, X) = (1, 1, 1), (R, A, Y) = (2, 2, 0), (R, B, X) = (4, 0, 2), (R, B, Y) = (0, 3, 3). Apply backward induction to find the SPE. Implement the game tree as a nested list in R and verify your answer with the `backward_induction()` function.

3. **Non-credible threats.** Modify the entry deterrence game so that the incumbent's Fight payoff is 2 instead of -1 (perhaps representing a market where fighting is costless). Recompute the SPE. Does the equilibrium change? Explain economically why the threat to fight is now credible.

4. **SPE vs. NE counting.** Convert the ultimatum game from Exercise 1 to normal form. Count all pure-strategy Nash equilibria and all subgame perfect equilibria. Verify that the set of SPE is a strict subset of the set of NE.

5. **Centipede game.** Two players alternate for 4 rounds, each choosing to **Stop** (ending the game) or **Continue** (passing to the other player). At round k, stopping gives the stopper k and the other player 0. If both continue for all 4 rounds, each receives 3. What does backward induction predict? Implement this in R by extending `backward_induction()` to handle a 4-node sequential game. Discuss why human behaviour in laboratory experiments typically departs from the backward-induction prediction.

Solutions appear in D.

Chapter 7

Bayesian Games

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

Games of incomplete information where players hold private types and form beliefs about opponents. Covers Bayesian Nash equilibrium, signaling games, and separating versus pooling equilibria with a worked job-market signaling example.

Learning objectives

- Explain how types and prior beliefs extend the normal-form model to incomplete information settings.
- Define Bayesian Nash equilibrium and compute it for small games with discrete types.
- Distinguish signaling games from other Bayesian games and identify separating, pooling, and semi-separating equilibria.
- Implement expected-payoff calculations under type uncertainty in R and produce publication-quality figures.

7.1 Motivation

In every game we have studied so far — from normal-form games (3) to extensive-form games (6) — players share complete knowledge of the payoff structure. In practice, this assumption is heroic. A firm entering a new market does not know its rival's cost structure. A job applicant knows her own ability, but the employer does not. A bidder in an auction knows his own valuation but must guess at others' valuations.

John Harsanyi's insight was that incomplete information can be modeled by introducing *types*: each player privately observes a type drawn from a commonly known distribution, and this type determines the player's payoffs. The resulting game is a **Bayesian game**, and the appropriate equilibrium concept is **Bayesian Nash equilibrium** (BNE). As Osborne (2004, Chapter 9) develops in detail, BNE asks each player to choose a strategy — a complete plan mapping types to actions — that is optimal in expectation over the other players' types.

Signaling games, a particularly important class of Bayesian games, arise when an informed player moves first and an uninformed player observes the action before responding. Spence's job-market signaling model is the classic example: a worker's education choice may reveal (or conceal) her productivity type to employers.

7.2 Theory

7.2.1 The Bayesian game model

A Bayesian game extends the standard normal form with three additional ingredients:

1. **Type spaces.** Each player i has a type $\theta_i \in \Theta_i$. A type encodes any private information that affects payoffs.
2. **Prior beliefs.** A common prior probability distribution $p(\theta_1, \dots, \theta_n)$ governs the joint distribution of types. Players update beliefs using Bayes' rule after observing their own type.
3. **Type-contingent payoffs.** Player i's payoff function $u_i(a_1, \dots, a_n; \theta_i)$ may depend on i's own type (and possibly on others' types).

A **strategy** in a Bayesian game is a function $s_i : \Theta_i \to A_i$ mapping each possible type to an action.

7.2.2 Bayesian Nash equilibrium

Definition: Bayesian Nash Equilibrium

A strategy profile $s^* = (s_1^*, \dots, s_n^*)$ is a **Bayesian Nash equilibrium** if, for every player i and every type $\theta_i \in \Theta_i$:

$$\mathbb{E}_{\theta_{-i}}\big[u_i(s_i^*(\theta_i),\, s_{-i}^*(\theta_{-i});\, \theta_i) \mid \theta_i\big] \geq \mathbb{E}_{\theta_{-i}}\big[u_i(a_i,\, s_{-i}^*(\theta_{-i});\, \theta_i) \mid \theta_i\big] \qquad (7.1)$$

for all $a_i \in A_i$.

In words, each type of each player maximizes expected payoff given the prior distribution over others' types and others' equilibrium strategies. BNE is the natural extension of Nash equilibrium (4) to incomplete information.

7.2.3 Signaling games

A **signaling game** is a two-player Bayesian game with sequential moves:

1. Nature draws the **sender's** type $\theta \in \Theta$ according to a prior $p(\theta)$.
2. The sender observes θ and chooses a **signal** (message) $m \in M$.
3. The **receiver** observes m (but not θ) and chooses an action $a \in A$.
4. Payoffs $u_S(m, a; \theta)$ and $u_R(m, a; \theta)$ are realized.

The equilibrium concept for signaling games is **perfect Bayesian equilibrium** (PBE), which requires that the receiver's beliefs about θ are derived from Bayes' rule whenever possible.

7.2.4 Separating vs. pooling equilibria

- **Separating equilibrium:** Different types choose different signals, so the receiver can perfectly infer the sender's type from the observed message.
- **Pooling equilibrium:** All types choose the same signal, so the receiver learns nothing beyond the prior.
- **Semi-separating (partial pooling) equilibrium:** Some types pool while others separate, or types randomize between signals.

The existence and nature of these equilibria depend on the cost structure of signaling and the payoff differences across types.

7.3 Implementation in R

We implement a two-type signaling game inspired by the Spence job-market model. A worker is either High-ability (θ_H) or Low-ability (θ_L), each with equal prior probability. The worker chooses whether to acquire education (E) or not (N). Education costs differ by type: it costs the High type $c_H = 1$ and the Low type $c_L = 4$. An employer then offers a wage equal to the expected productivity conditional on the observed education decision.

```
# Type parameters
types <- c("High", "Low")
prior <- c(High = 0.5, Low = 0.5)
```

```
productivity <- c(High = 6, Low = 2)
edu_cost <- c(High = 1, Low = 4)

# Wages under different employer beliefs
wage_if_high <- productivity["High"]
wage_if_low  <- productivity["Low"]
wage_pooled  <- sum(prior * productivity)

cat(sprintf("Productivity: High = %d, Low = %d\n",
            productivity["High"], productivity["Low"]))
```

```
#> Productivity: High = 6, Low = 2
```

```
cat(sprintf("Education cost: High = %d, Low = %d\n",
            edu_cost["High"], edu_cost["Low"]))
```

```
#> Education cost: High = 1, Low = 4
```

```
cat(sprintf("Pooling wage (prior expectation): %.1f\n",
 ↪  wage_pooled))
```

```
#> Pooling wage (prior expectation): 4.0
```

7.3.1 Expected payoffs under each equilibrium type

```
# Separating equilibrium: High educates, Low does not
# Employer infers type perfectly from education choice
sep_payoffs <- tibble(
  type = rep(types, each = 2),
  action = rep(c("Educate", "No Education"), times = 2),
  payoff = c(
    wage_if_high - edu_cost["High"],  # High, Educate
 ↪  (separating)
    wage_if_low,                       # High, No Education
 ↪  (deviate)
    wage_if_high - edu_cost["Low"],    # Low, Educate (deviate)
    wage_if_low                        # Low, No Education
 ↪  (separating)
  ),
  equilibrium = c("Separating", "Deviation", "Deviation",
   ↪  "Separating")
```

```
)

# Pooling equilibrium: both types educate
# Employer pays pooling wage to educated, low wage to uneducated
pool_payoffs <- tibble(
  type = rep(types, each = 2),
  action = rep(c("Educate", "No Education"), times = 2),
  payoff = c(
    wage_pooled - edu_cost["High"],  # High, Educate (pooling)
    wage_if_low,                      # High, No Education
↪   (deviate)
    wage_pooled - edu_cost["Low"],   # Low, Educate (pooling)
    wage_if_low                       # Low, No Education
↪   (deviate)
  ),
  equilibrium = c("Pooling", "Deviation", "Pooling", "Deviation")
)

cat("Separating equilibrium payoffs:\n")
```

```
#> Separating equilibrium payoffs:
```

```
sep_payoffs |>
  select(type, action, payoff, equilibrium) |>
  print()
```

```
#> # A tibble: 4 x 4
#>   type  action       payoff equilibrium
#>   <chr> <chr>         <dbl> <chr>
#> 1 High  Educate           5 Separating
#> 2 High  No Education      2 Deviation
#> 3 Low   Educate           2 Deviation
#> 4 Low   No Education      2 Separating
```

```
cat("\nPooling equilibrium payoffs:\n")
```

```
#>
#> Pooling equilibrium payoffs:
```

```
pool_payoffs |>
  select(type, action, payoff, equilibrium) |>
  print()
```

```
#> # A tibble: 4 x 4
#>   type  action       payoff equilibrium
#>   <chr> <chr>         <dbl> <chr>
#> 1 High  Educate           3 Pooling
#> 2 High  No Education      2 Deviation
#> 3 Low   Educate           0 Pooling
#> 4 Low   No Education      2 Deviation
```

7.3.2 Checking incentive compatibility

```
# Separating equilibrium IC checks
high_sep <- wage_if_high - edu_cost["High"]  # 6 - 1 = 5
high_dev <- wage_if_low                      # 2
low_sep  <- wage_if_low                      # 2
low_dev  <- wage_if_high - edu_cost["Low"]   # 6 - 4 = 2

cat("Separating equilibrium IC checks:\n")
```

```
#> Separating equilibrium IC checks:
```

```
cat(sprintf("  High type: Educate (%.0f) vs No Education (%.0f)
 ↪  -> %s\n",
            high_sep, high_dev,
            ifelse(high_sep >= high_dev, "IC satisfied", "IC
             ↪  violated")))
```

```
#>   High type: Educate (5) vs No Education (2) -> IC satisfied
```

```
cat(sprintf("  Low type:  No Education (%.0f) vs Educate (%.0f)
 ↪  -> %s\n",
            low_sep, low_dev,
            ifelse(low_sep >= low_dev, "IC satisfied", "IC
             ↪  violated")))
```

```
#>   Low type:  No Education (2) vs Educate (2) -> IC satisfied
```

7.3.3 Publication figure: expected payoffs by type and equilibrium

```
# Combine equilibrium payoffs for the chosen (non-deviation)
 ↪  actions
eq_summary <- tibble(
  type = rep(types, 2),
  equilibrium = rep(c("Separating", "Pooling"), each = 2),
  payoff = c(
    wage_if_high - edu_cost["High"],  # High, separating
    wage_if_low,                       # Low, separating
    wage_pooled - edu_cost["High"],   # High, pooling
    wage_pooled - edu_cost["Low"]     # Low, pooling
  ),
  action = c("Educate", "No Education", "Educate", "Educate")
)

p_bayesian <- ggplot(eq_summary,
                     aes(x = type, y = payoff, fill =
                      ↪  equilibrium)) +
  geom_col(position = position_dodge(width = 0.7), width = 0.6) +
  geom_text(aes(label = payoff),
            position = position_dodge(width = 0.7),
            vjust = -0.5, size = 3.5) +
  scale_fill_manual(values = c("Separating" = okabe_ito[1],
                               "Pooling" = okabe_ito[2]),
                    name = "Equilibrium type") +
  scale_y_continuous(name = "Expected payoff",
                     limits = c(-1, 6.5),
                     breaks = seq(-1, 6, 1)) +
  scale_x_discrete(name = "Worker type") +
  labs(title = "Payoff Comparison Across Equilibrium Types") +
  theme_publication()

p_bayesian
```

```
save_pub_fig(p_bayesian, "bayesian-payoff-comparison")
```

7.4 Worked example

We walk through the separating equilibrium step by step.

Step 1: Set up the game. Nature draws the worker's type: High (θ_H, productivity 6) or Low (θ_L, productivity 2), each with probability 0.5. Education costs 1 for the High type and 4 for the Low type.

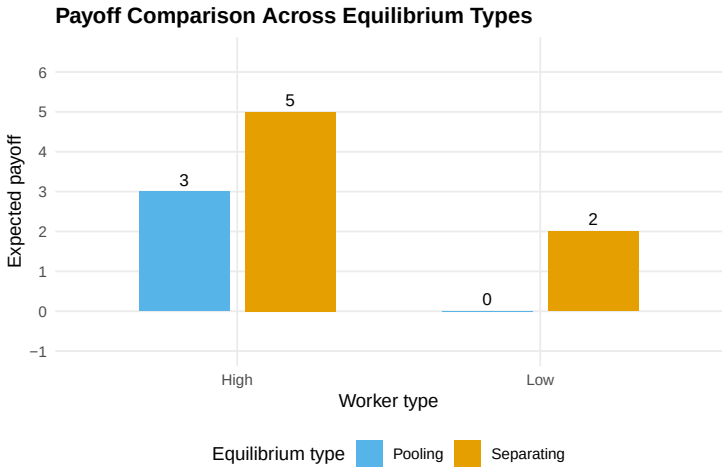


Figure 7.1: Expected payoffs for each worker type under separating and pooling equilibria in the job-market signaling game. In the separating equilibrium the High type earns 5 (wage 6 minus education cost 1) while the Low type earns 2. In the pooling equilibrium the High type earns 3 (pooling wage 4 minus cost 1) and the Low type earns 0 (pooling wage 4 minus cost 4). Bars shaded by equilibrium type.

Step 2: Propose a separating strategy. Suppose the High type educates and the Low type does not.

Step 3: Derive employer beliefs. Under this strategy, the employer observes E and infers the worker is High with certainty, offering wage 6. Observing N, the employer infers the worker is Low with certainty, offering wage 2.

Step 4: Check incentive compatibility. The High type's payoff from educating is $6 - 1 = 5$, and from deviating to no education is 2. Since $5 > 2$, the High type has no incentive to deviate. The Low type's payoff from not educating is 2, and from deviating to education is $6 - 4 = 2$. Since $2 \geq 2$ (weak IC), the Low type has no strict incentive to deviate.

Step 5: Verify the equilibrium. Both types play optimally given employer beliefs, and beliefs are consistent with strategies via Bayes' rule. This constitutes a perfect Bayesian equilibrium.

```
# Payoff summary table
worked_example <- tibble(
  `Worker type` = c("High", "Low"),
  `Equilibrium action` = c("Educate", "No Education"),
  `Wage received` = c(6, 2),
  `Education cost` = c(1, 0),
```

Separating Equilibrium in the Job-Market Signaling Game

Worker type	Equilibrium action	Wage received	Education cost	Net payoff	Deviation payoff
High	Educate	6	1	5	2
Low	No Education	2	0	2	2

```
  `Net payoff` = c(5, 2),
  `Deviation payoff` = c(2, 2),
  `IC satisfied?` = c("Yes (5 > 2)", "Yes (2 >= 2)")
)

worked_example |>
  gt() |>
  tab_header(
    title = "Separating Equilibrium in the Job-Market Signaling
    ↪  Game"
  ) |>
  cols_align(align = "center")
```

Note that the separating equilibrium Pareto-dominates the pooling equilibrium for the High type (payoff 5 vs. 3) but is weakly better for the Low type as well (payoff 2 vs. 0). This is a distinctive feature of signaling models: the ability to separate benefits the high type at the cost of an investment (education) that is socially wasteful if it does not enhance actual productivity.

7.5 Extensions

Bayesian games and signaling models underpin a vast literature in economics and political science:

- **Mechanism design** reverses the analysis: instead of solving a given game, the designer chooses the rules to achieve a desired outcome. Auctions (4) are a leading application.
- **Cheap talk** games are signaling games where the signal is costless. Equilibrium analysis then focuses on how much information can be credibly communicated.
- **Dynamic signaling** extends one-shot signaling to repeated interactions, connecting to the folk-theorem results of 8.

- **Refinements** such as the Intuitive Criterion (Cho and Kreps, 1987) restrict off-path beliefs to eliminate implausible pooling equilibria.

For a thorough treatment of Bayesian games, see Osborne (2004, Chapter 9) and Shoham & Leyton-Brown (2009). The original formalization of types and Bayesian equilibrium traces to Harsanyi's foundational work, building on the strategic framework of Neumann & Morgenstern (1944).

Exercises

1. **Modified signaling costs.** Suppose the Low type's education cost falls to $c_L = 3$. Does the separating equilibrium from the worked example survive? If not, what is the minimum cost c_L that sustains separation? Verify your answer in R.

2. **Pooling equilibrium check.** In the worked example, verify that a pooling equilibrium where *both* types choose No Education can be sustained. What off-path beliefs about a worker who deviates to Education would support this equilibrium?

3. **Three types.** Extend the signaling game to three types: High (productivity 8, cost 1), Medium (productivity 5, cost 3), and Low (productivity 2, cost 6), with equal priors. Find conditions under which a fully separating equilibrium exists.

4. **BNE in a Cournot game.** Two firms compete in quantities. Firm 1's marginal cost is commonly known to be $c_1 = 2$. Firm 2's marginal cost is $c_2 = 1$ (with probability 0.5) or $c_2 = 3$ (with probability 0.5), known only to Firm 2. Inverse demand is $P = 10 - Q$. Compute the Bayesian Nash equilibrium quantities. *(Hint: Firm 2 has two types, each choosing a quantity; Firm 1 chooses a single quantity that is optimal in expectation.)*

5. **Visualization.** Create a figure showing how the High type's payoff advantage in the separating equilibrium changes as the cost ratio c_L/c_H varies from 1 to 8. At what ratio does the separating equilibrium first become sustainable?

Solutions appear in D.

Chapter 8

Repeated Games

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

How repetition transforms strategic interaction. Covers finite and infinite horizon repeated games, discount factors, the folk theorem, and trigger strategies including Grim Trigger and Tit-for-Tat, with a worked example showing how cooperation can be sustained in the infinitely repeated Prisoner's Dilemma.

Learning objectives

- Distinguish between finitely and infinitely repeated games and explain the role of the discount factor.
- State the folk theorem and characterize the set of feasible and individually rational payoffs.
- Describe Grim Trigger and Tit-for-Tat strategies and derive the conditions under which they sustain cooperation.
- Plot the folk-theorem payoff region for a given stage game in R and verify trigger-strategy equilibria computationally.

8.1 Motivation

The Prisoner's Dilemma has a famously bleak prediction: rational players defect, even though mutual cooperation would make both better off. Yet in the real world, firms competing in the same market quarter after quarter often sustain tacit collusion. Nations locked in arms races sometimes reach stable agreements. Individuals in small communities cooperate routinely without binding contracts.

What changes when a game is played not once, but repeatedly? Repetition introduces the possibility of **punishment**: a player who defects today can be punished tomorrow. If the future matters enough — that is, if the **discount factor** is sufficiently high — the threat of future punishment can sustain cooperation as an equilibrium outcome. This is the central message of the **folk theorem**, one of the most powerful results in game theory.

Robert Axelrod's celebrated computer tournaments (Axelrod, 1981) demonstrated this insight empirically: Tit-for-Tat, a simple strategy that cooperates initially and then mirrors the opponent's previous action, emerged as the tournament winner, outperforming far more sophisticated strategies. We revisit Axelrod's tournament in 18; here we lay the theoretical foundations.

8.2 Theory

8.2.1 The stage game

A **repeated game** is built on a **stage game** $G = (N, (A_i), (u_i))$ played in each period $t = 0, 1, 2, ...$ The stage game we use throughout this chapter is the Prisoner's Dilemma with the following payoff matrix:

	C	D
C	(3, 3)	(0, 5)
D	(5, 0)	(1, 1)

Here $T = 5$ is the temptation payoff, $R = 3$ the reward for mutual cooperation, $P = 1$ the punishment for mutual defection, and $S = 0$ the sucker's payoff, satisfying the standard ordering $T > R > P > S$.

8.2.2 Finite vs. infinite horizon

In a **finitely repeated** game with T periods, backward induction unravels cooperation: in the last period, defection is dominant (just as in the one-shot game). Knowing this, both players defect in period $T - 1$, and so on, all the way back to period 1. The unique subgame-perfect equilibrium of the finitely repeated Prisoner's Dilemma is defection in every period.

In an **infinitely repeated** game, there is no last period, so backward induction cannot get started. Players evaluate infinite payoff streams using the **discount factor** $\delta \in [0, 1)$. The discounted average payoff from a stream $(u_0, u_1, u_2, ...)$ is:

$$V = (1-\delta)\sum_{t=0}^{\infty}\delta^t u_t \tag{8.1}$$

The factor $(1-\delta)$ normalizes the sum so that a constant stream $u_t = c$ yields $V = c$, making payoffs comparable to the stage game.

8.2.3 Trigger strategies

A **trigger strategy** prescribes cooperation as long as no player has deviated, and switches permanently (or temporarily) to a punishment phase after a deviation.

- **Grim Trigger:** Cooperate in the first period. In each subsequent period, cooperate if and only if both players have cooperated in every previous period. Any single defection triggers permanent mutual defection.
- **Tit-for-Tat (TFT):** Cooperate in the first period. In each subsequent period, play whatever the opponent played in the previous period. TFT punishes defection but forgives it after one period — a feature that proved remarkably effective in Axelrod (1981)'s tournament.

8.2.4 The Grim Trigger condition

Under Grim Trigger, the cooperative payoff stream is $R, R, R, ...$ with discounted value $V_C = R = 3$. If a player deviates in period t, she earns $T = 5$ in that period but triggers permanent defection, earning $P = 1$ forever after. The deviation payoff is:

$$V_D = (1-\delta)\left[T + \sum_{s=1}^{\infty}\delta^s P\right] = (1-\delta)T + \delta P \tag{8.2}$$

Cooperation is sustainable when $V_C \geq V_D$:

$$R \geq (1-\delta)T + \delta P \quad \Longleftrightarrow \quad \delta \geq \frac{T-R}{T-P} \tag{8.3}$$

For our payoffs: $\delta \geq \frac{5-3}{5-1} = \frac{1}{2}$.

8.2.5 The folk theorem

Theorem: Folk Theorem (Friedman, 1971)

Let G be a finite stage game and let $v = (v_1, \ldots, v_n)$ be any feasible payoff vector that strictly dominates each player's **minimax payoff** $\underline{v}_i$. Then for sufficiently large $\delta < 1$, there exists a subgame-perfect equilibrium of the infinitely repeated game with average payoff v.

The folk theorem tells us that repetition with patient players can sustain *any* feasible, individually rational payoff profile — a dramatic expansion of the equilibrium set compared to the one-shot game. The set of achievable payoffs forms a convex polytope bounded by feasibility (the convex hull of stage-game payoff profiles) and individual rationality (each player earns at least their minimax value). See Osborne (2004, Chapter 14) for the formal proof.

8.3 Implementation in R

8.3.1 Stage game setup and Grim Trigger analysis

```
# Stage game payoffs (Prisoner's Dilemma)
T_payoff <- 5   # temptation
R_payoff <- 3   # reward
P_payoff <- 1   # punishment
S_payoff <- 0   # sucker

# Critical discount factor for Grim Trigger
delta_star <- (T_payoff - R_payoff) / (T_payoff - P_payoff)
cat(sprintf("Critical discount factor (Grim Trigger): delta* =
 ↪  %.4f\n",
            delta_star))
```

```
#> Critical discount factor (Grim Trigger): delta* = 0.5000
```

8.3.2 Payoff comparison across discount factors

```
# Compare cooperation vs deviation payoffs for a range of delta
delta_seq <- seq(0.01, 0.99, by = 0.01)

payoff_data <- tibble(
  delta = delta_seq,
```

```
  cooperate = R_payoff,
  deviate = (1 - delta_seq) * T_payoff + delta_seq * P_payoff
)

payoff_long <- payoff_data |>
  pivot_longer(cols = c(cooperate, deviate),
               names_to = "strategy", values_to = "payoff") |>
  mutate(strategy = str_to_title(strategy))

cat("Cooperation payoff (constant):", R_payoff, "\n")
```

```
#> Cooperation payoff (constant): 3
```

```
cat("Deviation payoff at delta = 0.3:",
    round((1 - 0.3) * T_payoff + 0.3 * P_payoff, 2), "\n")
```

```
#> Deviation payoff at delta = 0.3: 3.8
```

```
cat("Deviation payoff at delta = 0.7:",
    round((1 - 0.7) * T_payoff + 0.7 * P_payoff, 2), "\n")
```

```
#> Deviation payoff at delta = 0.7: 2.2
```

8.3.3 Folk theorem payoff region

```
# All pure-strategy payoff profiles in the stage game
profiles <- tibble(
  p1_action = c("C", "C", "D", "D"),
  p2_action = c("C", "D", "C", "D"),
  u1 = c(R_payoff, S_payoff, T_payoff, P_payoff),
  u2 = c(R_payoff, T_payoff, S_payoff, P_payoff)
)

cat("Stage-game payoff profiles:\n")
```

```
#> Stage-game payoff profiles:
```

```
print(profiles)
```

```
#> # A tibble: 4 x 4
#>   p1_action p2_action    u1    u2
#>   <chr>     <chr>     <dbl> <dbl>
#> 1 C         C             3     3
#> 2 C         D             0     5
#> 3 D         C             5     0
#> 4 D         D             1     1
```

```
# Minimax payoffs (in PD, minimax = punishment payoff)
minimax_1 <- P_payoff
minimax_2 <- P_payoff
cat(sprintf("\nMinimax payoffs: v1 = %d, v2 = %d\n", minimax_1,
 ↪  minimax_2))
```

```
#>
#> Minimax payoffs: v1 = 1, v2 = 1
```

8.3.4 Publication figure: folk theorem payoff region

```
# Convex hull of feasible payoffs
feasible_hull <- profiles |>
  select(u1, u2) |>
  as.matrix()
hull_idx <- chull(feasible_hull)
hull_points <- feasible_hull[c(hull_idx, hull_idx[1]), ]
hull_df <- as_tibble(hull_points, .name_repair = ~ c("u1", "u2"))

# Individually rational and feasible region
# Intersection of feasible set with u1 >= 1 and u2 >= 1
# The feasible polygon vertices are (3,3), (0,5), (5,0), (1,1)
# Clipping to u1 >= 1 and u2 >= 1 yields vertices:
ir_vertices <- tibble(
  u1 = c(1, 1, 3, 5, 4),
  u2 = c(4, 1, 3, 0, 1)
)

# More precisely, find the individually rational feasible set
# The feasible set edges: (0,5)-(3,3), (3,3)-(5,0), (5,0)-(1,1),
 ↪  (1,1)-(0,5)
# Clipping with u1 >= 1: edge (0,5)-(1,1) at u1=1 gives u2
 ↪  between 1 and 5
# edge (0,5)-(3,3) at u1=1 gives u2 = 5 - (2/3)*1 = 5 - 2/3 =
 ↪  13/3
```

```
# Clipping with u2 >= 1: edge (5,0)-(1,1) at u2=1 gives u1=1
# edge (5,0)-(3,3) at u2=1 gives u1 = 5 - (5/3)*1 = ... no.
# Let's compute properly.

# Feasible set is convex hull of (3,3), (0,5), (5,0), (1,1)
# The IR region clips this with u1 >= 1, u2 >= 1
# Since (1,1) is already a vertex and all other vertices except
 ↪  (0,5) have u1>=1
# and all except (5,0) have u2>=1, the clipping is:

# Edge from (0,5) to (3,3): parametrically (3t, 5-2t) for t in
 ↪  [0,1]
# u1 = 1 => t = 1/3, u2 = 5 - 2/3 = 13/3
# Edge from (0,5) to (1,1): parametrically (t, 5-4t) for t in
 ↪  [0,1]
# u1 = 1 => t = 1, giving (1,1) -- that's the endpoint
# Edge from (5,0) to (3,3): parametrically (5-2t, 3t) for t in
 ↪  [0,1]
# u2 = 1 => t = 1/3, u1 = 5 - 2/3 = 13/3
# Edge from (5,0) to (1,1): parametrically (5-4t, t) for t in
 ↪  [0,1]
# u2 = 1 => t = 1, giving (1,1)

ir_region <- tibble(
  u1 = c(1, 1, 3, 13/3, 1),
  u2 = c(1, 13/3, 3, 1, 1)
)

# Key labeled points
key_points <- tibble(
  u1 = c(R_payoff, P_payoff, T_payoff, S_payoff),
  u2 = c(R_payoff, P_payoff, S_payoff, T_payoff),
  label = c("(C,C)", "(D,D)", "(D,C)", "(C,D)")
)

p_folk <- ggplot() +
  geom_polygon(data = hull_df, aes(x = u1, y = u2),
               fill = "grey90", colour = "grey50",
               linetype = "dashed", alpha = 0.5) +
  geom_polygon(data = ir_region, aes(x = u1, y = u2),
               fill = okabe_ito[2], alpha = 0.3,
               colour = okabe_ito[5], linewidth = 0.8) +
  geom_hline(yintercept = minimax_2, linetype = "dotted",
             colour = "grey40") +
  geom_vline(xintercept = minimax_1, linetype = "dotted",
             colour = "grey40") +
```

```
  geom_point(data = key_points, aes(x = u1, y = u2),
             size = 3, colour = okabe_ito[6]) +
  geom_text(data = key_points, aes(x = u1, y = u2, label =
   ↪  label),
            vjust = -0.8, size = 3.2, fontface = "bold") +
  annotate("text", x = 2.2, y = 2.5,
           label = "Folk theorem\nachievable set",
           colour = okabe_ito[5], size = 3.5, fontface =
            ↪  "italic") +
  annotate("text", x = 0.3, y = minimax_2 + 0.2,
           label = expression(underline(v)[2]),
           colour = "grey40", size = 3.5) +
  annotate("text", x = minimax_1 + 0.3, y = -0.2,
           label = expression(underline(v)[1]),
           colour = "grey40", size = 3.5) +
  scale_x_continuous(name = expression(u[1]),
                     limits = c(-0.5, 5.8), breaks = 0:5) +
  scale_y_continuous(name = expression(u[2]),
                     limits = c(-0.5, 5.8), breaks = 0:5) +
  coord_fixed() +
  labs(title = "Folk Theorem: Feasible and Individually Rational
   ↪  Payoffs") +
  theme_publication()

p_folk
```

```
save_pub_fig(p_folk, "folk-theorem-region")
```

8.4 Worked example

We verify step by step that Grim Trigger sustains cooperation in the infinitely repeated Prisoner's Dilemma when $\delta = 0.6$.

Step 1: Define the stage game. The payoffs are $T = 5$, $R = 3$, $P = 1$, $S = 0$ as defined above.

Step 2: Specify the strategy. Both players use Grim Trigger: cooperate initially, and continue cooperating as long as no defection has occurred. After any defection, defect forever.

Step 3: Compute the cooperation payoff. On the equilibrium path, both players cooperate forever. The discounted average payoff is $V_C = R = 3$.

Step 4: Compute the deviation payoff. A player who defects in period t earns $T = 5$ in that period. From period $t + 1$ onward, both players defect

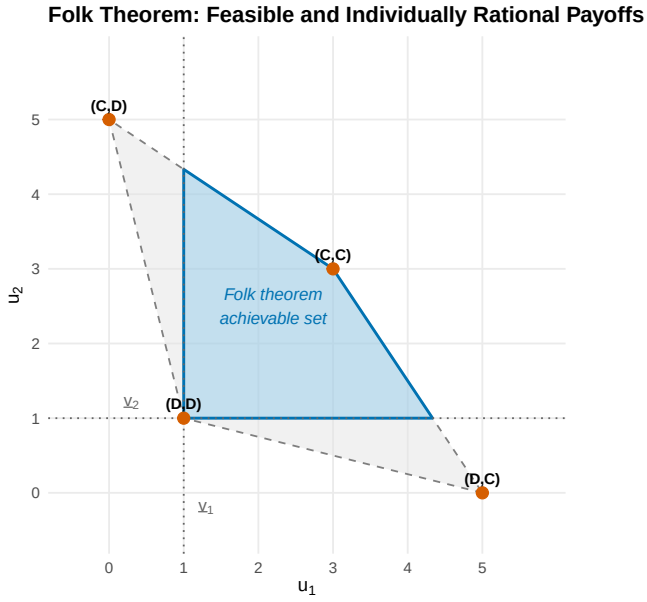


Figure 8.1: The set of achievable payoffs under the folk theorem for the Prisoner's Dilemma. The outer polygon (light fill) shows the feasible set — the convex hull of stage-game payoff profiles. The shaded interior region represents payoff pairs that are both feasible and individually rational (each player earns at least their minimax payoff of 1). Any point in the shaded region can be sustained as a subgame-perfect equilibrium for sufficiently high discount factor.

(Grim Trigger activated), earning $P = 1$ per period. The discounted average payoff from deviation is:

$$V_D = (1 - 0.6) \cdot 5 + 0.6 \cdot 1 = 2.0 + 0.6 = 2.6$$

Step 5: Verify the equilibrium condition. Since $V_C = 3.0 > 2.6 = V_D$, deviation is unprofitable. Grim Trigger is a subgame-perfect equilibrium for $\delta = 0.6$.

```
delta <- 0.6

V_C <- R_payoff
V_D <- (1 - delta) * T_payoff + delta * P_payoff

cat(sprintf("Discount factor: delta = %.1f\n", delta))
```

```
#> Discount factor: delta = 0.6
```

```
cat(sprintf("Cooperation payoff (Grim Trigger): V_C = %.1f\n",
 ↪  V_C))
```

```
#> Cooperation payoff (Grim Trigger): V_C = 3.0
```

```
cat(sprintf("Deviation payoff:                  V_D = %.1f\n",
 ↪  V_D))
```

```
#> Deviation payoff:                  V_D = 2.6
```

```
cat(sprintf("Cooperation sustained? %s (V_C = %.1f %s V_D =
 ↪  %.1f)\n",
            ifelse(V_C >= V_D, "YES", "NO"),
            V_C, ifelse(V_C >= V_D, ">=", "<"), V_D))
```

```
#> Cooperation sustained? YES (V_C = 3.0 >= V_D = 2.6)
```

```
cat(sprintf("\nCritical threshold: delta* = %.4f\n", delta_star))
```

```
#>
#> Critical threshold: delta* = 0.5000
```

Grim Trigger Equilibrium Verification
Infinitely Repeated Prisoner's Dilemma

Parameter	Value
Discount factor (delta)	0.6
Critical threshold (delta*)	0.5000
Cooperation payoff (V_C)	3.0
Deviation payoff (V_D)	2.6
Equilibrium?	Yes

```
cat(sprintf("Current delta = %.1f %s delta* = %.4f\n",
            delta, ifelse(delta >= delta_star, ">=", "<"),
 ↪  delta_star))
```

```
#> Current delta = 0.6 >= delta* = 0.5000
```

```
# Summary table
grim_summary <- tibble(
  Parameter = c("Discount factor (delta)", "Critical threshold
   ↪  (delta*)",
                "Cooperation payoff (V_C)", "Deviation payoff
                 ↪  (V_D)",
                "Equilibrium?"),
  Value = c(sprintf("%.1f", delta), sprintf("%.4f", delta_star),
            sprintf("%.1f", V_C), sprintf("%.1f", V_D),
            ifelse(V_C >= V_D, "Yes", "No"))
)

grim_summary |>
  gt() |>
  tab_header(title = "Grim Trigger Equilibrium Verification",
             subtitle = "Infinitely Repeated Prisoner's Dilemma")
```

8.4.1 Comparing Grim Trigger and Tit-for-Tat

Grim Trigger sustains cooperation through the harshest possible punishment: permanent defection. Tit-for-Tat, by contrast, punishes for only one period and then forgives. This makes TFT more robust in noisy environments (where actions are sometimes misperceived) but requires a higher discount factor to sustain cooperation against deliberate deviation. In Axelrod (1981)'s tournament,

TFT's combination of niceness (never defect first), retaliation (punish defection), forgiveness (return to cooperation), and clarity (simple to understand) proved devastatingly effective. We simulate the tournament in detail in 18.

8.5 Extensions

- **Finite repetition with multiple equilibria.** If the stage game has multiple Nash equilibria (4), cooperation can sometimes be sustained even in finitely repeated games by using equilibrium selection as a reward-and-punishment device.
- **Imperfect monitoring.** When players observe only noisy signals of each other's actions, the analysis becomes substantially more complex. The folk theorem still holds under certain conditions, but strategies must be adapted to tolerate observation errors.
- **Stochastic games.** When the stage game itself changes over time (e.g., market conditions fluctuate), the analysis extends to stochastic games, a generalization of repeated games.
- **Evolutionary dynamics.** Axelrod's tournament approach connects repeated games to evolutionary game theory, where strategies compete and reproduce based on fitness. This perspective is developed further in 18.

For the formal treatment of the folk theorem and its variants, see Osborne (2004, Chapter 14). The discount-factor analysis of trigger strategies follows the framework in Shoham & Leyton-Brown (2009).

Exercises

1. **Critical discount factor.** Consider a Prisoner's Dilemma with payoffs $T = 8$, $R = 5$, $P = 2$, $S = 0$. Compute the critical discount factor δ^* for Grim Trigger to sustain cooperation. Verify your answer in R by computing V_C and V_D at $\delta = \delta^*$.

2. **Tit-for-Tat analysis.** In the stage game from this chapter ($T = 5$, $R = 3$, $P = 1$, $S = 0$), derive the critical discount factor for Tit-for-Tat to sustain cooperation. *(Hint: after a one-period deviation, TFT retaliates for one period, then returns to cooperation. Compute the full deviation payoff stream.)*

3. **Folk theorem region.** Consider a stage game with payoff profiles (4, 4), (0, 6), (6, 0), and (1, 1). Plot the feasible and individually rational region. How does it differ from the Prisoner's Dilemma region in 8.1?

4. **Three-player repeated game.** Extend the Grim Trigger analysis to a three-player setting where each player can Cooperate or Defect. Cooperation yields 3 to each cooperator; each defector earns 5 regardless of others' actions; but if all defect, each earns 1. What is the critical discount factor? *(Hint: consider the worst-case deviation.)*

5. **Simulation.** Simulate 1000 rounds of the infinitely repeated PD (with a continuation probability of $\delta = 0.95$ each round) for two Grim Trigger players and two TFT players. Compare their average per-round payoffs and plot the cumulative payoff trajectories.

Solutions appear in D.

Chapter 9

Cooperative Game Theory

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

An introduction to cooperative game theory with transferable utility. Covers the characteristic function, the core, and the Shapley value with computational examples in R, including a worked cost-sharing problem for three municipalities.

Learning objectives

- Define a transferable-utility (TU) game using its characteristic function and explain how it differs from non-cooperative models.
- Compute the core of a cooperative game and determine whether a given allocation lies in the core.
- Calculate the Shapley value for small games by hand and in R, and interpret its fairness properties.
- Apply cooperative game theory to practical cost-sharing and voting-power problems.

9.1 Motivation

Non-cooperative game theory, as developed in 4 and preceding chapters, models strategic interaction by focusing on individual players' strategies and the equilibria that emerge. But many real-world situations are better described by asking which **coalitions** can form and how the resulting surplus should be divided. When three municipalities consider building a shared water treatment plant, the key questions are not about individual strategic moves but about which

subsets of towns benefit from cooperating and how the cost savings should be split.

Cooperative game theory, whose foundations were laid by Neumann & Morgenstern (1944), takes coalitions as the primitive unit of analysis. Instead of specifying each player's strategy set, a cooperative game specifies the **worth** (or value) that each possible coalition can generate. The central solution concepts — the **core** and the **Shapley value** — then determine which divisions of the total payoff are stable against defection and which are uniquely fair.

This chapter focuses on **transferable-utility (TU) games**, where utility can be freely redistributed among coalition members (e.g., via monetary side payments). We introduce the characteristic function, define stability and fairness concepts, and implement everything computationally in R. For R packages that automate many of these calculations, see 12 and 11.

9.2 Theory

9.2.1 Characteristic function games

A **TU cooperative game** is a pair (N, v) where:

- $N = \{1, 2, \dots, n\}$ is the set of players.
- $v : 2^N \to \mathbb{R}$ is the **characteristic function** assigning a value $v(S)$ to each coalition $S \subseteq N$, with $v(\emptyset) = 0$.

The value $v(S)$ represents the total payoff that members of S can guarantee themselves by cooperating, regardless of what players outside S do. A game is **superadditive** if $v(S \cup T) \geq v(S) + v(T)$ for all disjoint $S, T \subseteq N$, meaning that merging coalitions never destroys value.

9.2.2 Imputations

An **imputation** is a payoff vector $x = (x_1, \dots, x_n)$ satisfying:

1. **Efficiency:** $\sum_{i \in N} x_i = v(N)$.
2. **Individual rationality:** $x_i \geq v(\{i\})$ for all $i \in N$.

Efficiency requires that the grand coalition's total value is fully distributed. Individual rationality ensures no player receives less than they could obtain alone.

9.2.3 The core

Definition: The Core

The **core** of a TU game (N, v) is the set of imputations x such that no coalition has an incentive to deviate:

$$\sum_{i \in S} x_i \geq v(S) \quad \text{for all } S \subseteq N \tag{9.1}$$

An allocation in the core is **stable** in the sense that no subset of players can do better by breaking away from the grand coalition. The core may be empty (some games have no stable allocation), a single point, or a convex polytope.

9.2.4 The Shapley value

While the core captures stability, it does not provide a unique solution. The **Shapley value** provides a unique allocation based on fairness axioms.

Definition: Shapley Value

The Shapley value of player i in a TU game (N, v) is:

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!\,(n - |S| - 1)!}{n!} \left[v(S \cup \{i\}) - v(S) \right] \tag{9.2}$$

The term $v(S \cup \{i\}) - v(S)$ is player i's **marginal contribution** to coalition S. The Shapley value averages this marginal contribution over all possible orderings in which the grand coalition could be assembled. It is the unique allocation satisfying four axioms: efficiency, symmetry, additivity, and the null-player property.

As Osborne (2004, Chapter 18) emphasizes, the Shapley value can be interpreted as the expected marginal contribution of a player when players join the coalition in a uniformly random order.

9.3 Implementation in R

9.3.1 Defining a cooperative game

We consider a three-player cost-sharing game. Three municipalities (A, B, C) can build a water treatment plant individually or in coalition. The costs are:

Coalition	Cost	Savings $v(S)$
{A}	100	0
{B}	140	0
{C}	120	0
{A, B}	200	40
{A, C}	180	40
{B, C}	220	40
{A, B, C}	300	60

The characteristic function $v(S)$ measures cost *savings* relative to individual construction. We normalize $v(\{i\}) = 0$ for each individual player.

```
# Define the 3-player TU game (cost savings)
players <- c("A", "B", "C")
n <- length(players)

# Characteristic function: value of each coalition
# Using binary encoding: A=1, B=2, C=4
v <- c(
  "0"   = 0,    # empty

  "A"   = 0,    # {A}
  "B"   = 0,    # {B}
  "C"   = 0,    # {C}
  "AB"  = 40,   # {A, B}
  "AC"  = 40,   # {A, C}
  "BC"  = 40,   # {B, C}
  "ABC" = 60    # {A, B, C}
)

cat("Characteristic function v(S):\n")
```

```
#> Characteristic function v(S):
```

```
for (name in names(v)) {
  if (name != "0") {
    cat(sprintf("  v({%s}) = %d\n",
                paste(strsplit(name, "")[[1]], collapse = ", "),
                 ↪  v[name]))
  }
}
```

```
#>   v({A}) = 0
```

```
#>   v({B}) = 0
#>   v({C}) = 0
#>   v({A, B}) = 40
#>   v({A, C}) = 40
#>   v({B, C}) = 40
#>   v({A, B, C}) = 60
```

9.3.2 Computing the Shapley value

```
# Permutation generator (no external dependencies)
permn <- function(x) {
  if (length(x) <= 1) return(list(x))
  result <- list()
  for (i in seq_along(x)) {
    rest <- permn(x[-i])
    for (p in rest) result <- c(result, list(c(x[i], p)))
  }
  result
}

# Compute Shapley value by enumeration of permutations
compute_shapley <- function(players, v_func) {
  n <- length(players)
  perms <- permn(players)
  marginals <- matrix(0, nrow = length(perms), ncol = n,
                      dimnames = list(NULL, players))

  for (k in seq_along(perms)) {
    perm <- perms[[k]]
    for (i in seq_along(perm)) {
      player <- perm[i]
      if (i == 1) {
        predecessors <- character(0)
      } else {
        predecessors <- perm[1:(i - 1)]
      }
      coal_with <- paste(sort(c(predecessors, player)), collapse
 ↪  = "")
      coal_without <- paste(sort(predecessors), collapse = "")
      if (coal_without == "") coal_without <- "0"
      marginals[k, player] <- v_func[coal_with] -
 ↪  v_func[coal_without]
    }
  }
```

```
  shapley_vals <- colMeans(marginals)
  list(values = shapley_vals, marginals = marginals)
}

shapley_result <- compute_shapley(players, v)
shapley_vals <- shapley_result$values

cat("Shapley values:\n")
```

```
#> Shapley values:
```

```
for (p in players) {
  cat(sprintf("  phi(%s) = %.4f\n", p, shapley_vals[p]))
}
```

```
#>   phi(A) = 20.0000
#>   phi(B) = 20.0000
#>   phi(C) = 20.0000
```

```
cat(sprintf("\nSum of Shapley values: %.4f (should equal v(N) =
 ↪  %d)\n",
            sum(shapley_vals), v["ABC"]))
```

```
#>
#> Sum of Shapley values: 60.0000 (should equal v(N) = 60)
```

9.3.3 Checking the core

```
# An allocation is in the core if it satisfies all coalition
 ↪  constraints
check_core <- function(x, players, v_func) {
  n <- length(players)
  violations <- character(0)

  # Check all non-empty subsets
  for (k in 1:(2^n - 1)) {
    bits <- as.integer(intToBits(k)[1:n])
    coalition <- players[bits == 1]
    coal_key <- paste(sort(coalition), collapse = "")
```

```
    coal_val <- v_func[coal_key]
    alloc_sum <- sum(x[coalition])

    if (alloc_sum < coal_val - 1e-10) {
      violations <- c(violations,
                      sprintf("{%s}: sum = %.2f < v = %.0f",
                              paste(coalition, collapse = ", "),
                              alloc_sum, coal_val))
    }
  }

  list(in_core = length(violations) == 0, violations =
   ↪  violations)
}

# Check if Shapley value is in the core
core_result <- check_core(shapley_vals, players, v)
cat("Is the Shapley value in the core? ",
    ifelse(core_result$in_core, "YES", "NO"), "\n")
```

```
#> Is the Shapley value in the core?  YES
```

```
if (!core_result$in_core) {
  cat("Violations:\n")
  for (viol in core_result$violations) cat("  ", viol, "\n")
}

# Display all coalition constraints
cat("\nCore constraints:\n")
```

```
#>
#> Core constraints:
```

```
coalitions <- list(
  c("A"), c("B"), c("C"),
  c("A", "B"), c("A", "C"), c("B", "C")
)

for (coal in coalitions) {
  coal_key <- paste(sort(coal), collapse = "")
  cat(sprintf("  x(%s) = %.2f >= v({%s}) = %d : %s\n",
              paste(coal, collapse = " + "),
              sum(shapley_vals[coal]),
              paste(coal, collapse = ", "),
```

```
              v[coal_key],
              ifelse(sum(shapley_vals[coal]) >= v[coal_key] -
               ↪  1e-10,
                     "satisfied", "VIOLATED")))
}
```

```
#>   x(A) = 20.00 >= v({A}) = 0 : satisfied
#>   x(B) = 20.00 >= v({B}) = 0 : satisfied
#>   x(C) = 20.00 >= v({C}) = 0 : satisfied
#>   x(A + B) = 40.00 >= v({A, B}) = 40 : satisfied
#>   x(A + C) = 40.00 >= v({A, C}) = 40 : satisfied
#>   x(B + C) = 40.00 >= v({B, C}) = 40 : satisfied
```

9.3.4 Publication figure: Shapley value comparison

```
shapley_df <- tibble(
  player = factor(players, levels = players),
  shapley = shapley_vals[players]
)

p_shapley <- ggplot(shapley_df, aes(x = player, y = shapley, fill
 ↪  = player)) +
  geom_col(width = 0.6, show.legend = FALSE) +
  geom_text(aes(label = sprintf("%.1f", shapley)),
            vjust = -0.5, size = 4, fontface = "bold") +
  scale_fill_manual(values = okabe_ito[1:3]) +
  scale_y_continuous(name = "Shapley value (cost savings)",
                     limits = c(0, max(shapley_vals) * 1.3),
                     expand = expansion(mult = c(0, 0.05))) +
  scale_x_discrete(name = "Municipality") +
  labs(title = "Shapley Value Allocation: Water Treatment Cost
   ↪  Sharing") +
  theme_publication()

p_shapley
```

```
save_pub_fig(p_shapley, "shapley-values-3player")
```

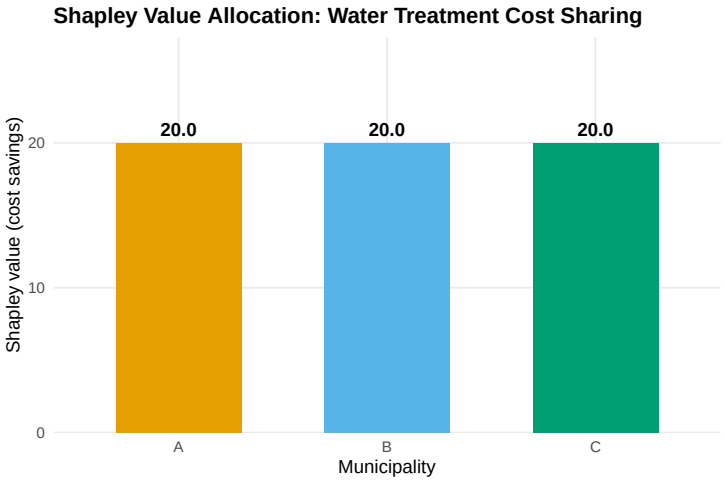


Figure 9.1: Shapley values for the three-player water treatment cost-sharing game. Each bar represents a municipality's fair share of the total cost savings (60 units) based on average marginal contributions across all coalition formation orders. The Shapley value assigns equal shares (20 each) due to the symmetric structure of pair-wise savings in this game.

9.3.5 Marginal contributions table

```
# Show marginal contributions in each ordering
perms <- permn(players)

mc_table <- tibble(
  ordering = sapply(perms, paste, collapse = " -> "),
  mc_A = shapley_result$marginals[, "A"],
  mc_B = shapley_result$marginals[, "B"],
  mc_C = shapley_result$marginals[, "C"]
)

mc_table |>
  gt() |>
  tab_header(
    title = "Marginal Contributions by Player Ordering",
    subtitle = "Each row shows one permutation and the marginal
     ↪  contribution of each player when joining in that order"
  ) |>
  cols_label(
    ordering = "Ordering",
    mc_A = "MC(A)",
```

Marginal Contributions by Player Ordering

Each row shows one permutation and the marginal contribution of each player when joining in that order

	Ordering	MC(A)	MC(B)	MC(C)
	A -> B -> C	0	40	20
	A -> C -> B	0	20	40
	B -> A -> C	40	0	20
	B -> C -> A	20	0	40
	C -> A -> B	40	20	0
	C -> B -> A	20	40	0
Average	—	20.00	20.00	20.00

```
  mc_B = "MC(B)",
  mc_C = "MC(C)"
) |>
cols_align(align = "center", columns = starts_with("mc")) |>
grand_summary_rows(
  fns = list(Average = ~ mean(.)),
  columns = starts_with("mc"),
  fmt = ~ fmt_number(., decimals = 2)
)
```

9.4 Worked example

We walk through the Shapley value calculation for municipality A in the water treatment cost-sharing game.

Step 1: List all permutations. With three players, there are $3! = 6$ orderings.

Step 2: Compute A's marginal contribution in each ordering.

- **A-B-C:** A joins the empty coalition. $v(\{A\}) - v(\emptyset) = 0 - 0 = 0$.
- **A-C-B:** Same as above. $MC(A) = 0$.
- **B-A-C:** A joins {B}. $v(\{A, B\}) - v(\{B\}) = 40 - 0 = 40$.
- **C-A-B:** A joins {C}. $v(\{A, C\}) - v(\{C\}) = 40 - 0 = 40$.
- **B-C-A:** A joins {B,C}. $v(\{A, B, C\}) - v(\{B, C\}) = 60 - 40 = 20$.
- **C-B-A:** A joins {B,C}. Same as above. $MC(A) = 20$.

Step 3: Average the marginal contributions.

$$\phi_A = \frac{0 + 0 + 40 + 40 + 20 + 20}{6} = \frac{120}{6} = 20$$

Step 4: Verify efficiency. By symmetry of the game (all two-player coalitions have the same value), $\phi_A = \phi_B = \phi_C = 20$. The sum is $20+20+20 = 60 = v(N)$, confirming efficiency.

```
# Verify step by step for player A
mc_A <- c(
  v["A"] - v["0"],      # A first
  v["A"] - v["0"],      # A first (other order)
  v["AB"] - v["B"],     # A joins B
  v["AC"] - v["C"],     # A joins C
  v["ABC"] - v["BC"],   # A joins BC
  v["ABC"] - v["BC"]    # A joins BC (other order)
)

cat("Player A marginal contributions:", mc_A, "\n")
```

```
#> Player A marginal contributions: 0 0 40 40 20 20
```

```
cat(sprintf("Shapley value for A: %.0f / 6 = %.4f\n",
            sum(mc_A), mean(mc_A)))
```

```
#> Shapley value for A: 120 / 6 = 20.0000
```

```
# Summary
final_allocation <- tibble(
  Municipality = players,
  `Stand-alone cost` = c(100, 140, 120),
  `Shapley share of savings` = shapley_vals[players],
  `Cost after sharing` = c(100, 140, 120) - shapley_vals[players]
)

final_allocation |>
  gt() |>
  tab_header(
    title = "Final Cost Allocation via Shapley Value"
  ) |>
  fmt_number(columns = c(`Shapley share of savings`, `Cost after
  ↪  sharing`),
             decimals = 1) |>
  cols_align(align = "center")
```

Final Cost Allocation via Shapley Value

Municipality	Stand-alone cost	Shapley share of savings	Cost after sharing
A	100	20.0	80.0
B	140	20.0	120.0
C	120	20.0	100.0

Step 5: Interpret. Each municipality saves 20 units from cooperation. Municipality A's cost drops from 100 to 80, B's from 140 to 120, and C's from 120 to 100. The Shapley value splits the 60 units of savings equally because the game's structure treats all pair-wise coalitions symmetrically. In an asymmetric game — say, where {A,B} saves more than {A,C} — the Shapley value would allocate more savings to the players whose participation creates more value.

9.5 Extensions

Cooperative game theory extends in several important directions:

- **Non-transferable utility (NTU) games** drop the assumption that utility can be freely redistributed. The solution concepts become more complex; the NTU Shapley value and the Nash bargaining solution are commonly used.
- **Voting games** are a special class of TU games where each coalition either wins (value 1) or loses (value 0). The Shapley–Shubik power index measures each voter's influence and is widely applied in political science.
- **Cost allocation** problems in operations research use the Shapley value and the nucleolus (which minimizes the maximum dissatisfaction of any coalition) to allocate shared infrastructure costs.
- **Computational complexity.** Computing the Shapley value requires summing over all 2^n coalitions (or all $n!$ permutations), which becomes infeasible for large n. Approximation algorithms based on random sampling of permutations are used in practice; see 12 and 11 for R implementations.

For the mathematical foundations of cooperative game theory, see Neumann & Morgenstern (1944) and Osborne (2004, Chapters 16–18). A computational perspective is given in Shoham & Leyton-Brown (2009).

Exercises

1. **Asymmetric savings.** Modify the water treatment game so that $v(\{A, B\}) = 50$, $v(\{A, C\}) = 30$, $v(\{B, C\}) = 40$, and $v(\{A, B, C\}) = 80$.

Compute the Shapley value for each player. Which municipality benefits most from cooperation?

2. **Core membership.** For the modified game in Exercise 1, determine whether the Shapley value lies in the core. If not, find an allocation that does lie in the core, or prove that the core is empty.

3. **Voting power.** Consider a weighted voting game with four players and weights $(4, 3, 2, 1)$ and a quota of 6 (a coalition wins if its total weight is at least 6). Compute the Shapley–Shubik power index for each player. Compare it to each player's weight share. *(Hint: enumerate all* $4! = 24$ *orderings and identify each player's pivot position.)*

4. **Glove game.** In the "glove game," players 1 and 2 each have a left glove and player 3 has a right glove. A pair of gloves (one left, one right) is worth 1; individual gloves are worthless. Write the characteristic function, compute the Shapley value, and find the core. Is the Shapley value in the core?

5. **Scaling to larger games.** Implement a Monte Carlo approximation of the Shapley value that samples K random permutations instead of enumerating all $n!$. Test it on a 10-player game where $v(S) = |S|^2$ and compare the approximation error for $K = 100$, $K = 1000$, and $K = 10000$.

Solutions appear in D.

Part II

The R Toolkit

Chapter 10

The R Environment for Strategic Computation

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

A practical guide to setting up RStudio or Positron for game-theoretic computation in R, managing reproducibility with renv, organizing projects with a consistent directory layout, and configuring the shared R/_common.R pattern and knitr options used throughout this book.

Learning objectives

After completing this chapter you will be able to:

- Configure RStudio or Positron for efficient game-theoretic computation, including essential settings and recommended pane layouts.
- Use `renv` to create reproducible, self-contained R projects with locked dependency snapshots.
- Organize a game theory project using a conventional directory layout with separate folders for code, data, images, and output.
- Understand and extend the `R/_common.R` shared-setup pattern used throughout this book.
- Set appropriate `knitr` chunk options for reproducible, publication-quality document rendering.

10.1 Motivation

When you first encounter a game-theoretic model — say the Prisoner's Dilemma introduced in 2 — it is tempting to open a fresh R script and start coding immediately. For a single exercise, that approach works. But real research and coursework involve dozens of interrelated scripts, multiple chapters or reports, shared utility functions, and results that must be reproducible months or years later. Without a disciplined project environment, you will eventually face the dreaded situation where code that ran perfectly last semester no longer works because a package was updated, a file path changed, or a helper function was silently overwritten.

This chapter addresses the infrastructure layer beneath the game theory content. Just as Neumann & Morgenstern (1944) needed a formal language before proving theorems about games, we need a well-configured computational environment before implementing those theorems in R. The payoff is large: a properly structured project lets you focus on the economics and mathematics rather than on debugging file paths and package conflicts.

The tools we cover — RStudio/Positron as an IDE, `renv` for dependency management, the `here` package for portable paths, and a shared `_common.R` script for consistent defaults — form the backbone of every chapter in this book. Understanding them once will save hours of frustration throughout the rest of your game theory work.

10.2 Theory

10.2.1 The reproducibility stack

Reproducibility in computational research operates at several layers, each building on the one below:

1. **Environment.** The IDE, R version, and operating system. RStudio (by Posit) is the most widely used R IDE; Positron is Posit's newer, open-source, multilingual IDE built on VS Code technology. Both provide integrated consoles, editors, and project management.

2. **Dependencies.** The specific versions of R packages your code uses. The `renv` package creates a project-local library and a lockfile (`renv.lock`) that records the exact version of every package. Collaborators (or your future self) can call `renv::restore()` to recreate the identical package environment.

3. **Project structure.** A conventional directory layout ensures that scripts, data, and output live in predictable locations. The `here` package provides

a project-root-aware path constructor so that `here("R", "_common.R")` resolves correctly regardless of the working directory.

4. **Rendering options.** Bookdown and `knitr` chunk options control how code is executed, cached, and displayed. Consistent defaults — set once in a shared file — prevent output discrepancies across chapters.

10.2.2 Why IDE configuration matters for game theory

Game-theoretic computation involves particular workflows that benefit from IDE tuning. Computing Nash equilibria (see 4) often requires iterative debugging of matrix algebra. Simulating evolutionary dynamics requires monitoring long-running loops. Producing publication figures demands a graphics device with adequate resolution. A well-configured IDE makes each of these tasks smoother.

Key settings include:

- **Soft-wrap long lines.** Game-theoretic payoff definitions can produce wide lines; soft wrapping prevents horizontal scrolling.
- **Increase console buffer.** Simulations with many iterations produce verbose output; a larger buffer preserves the full log.
- **Default working directory.** Set the working directory to the project root at startup so that `here()` paths resolve immediately.
- **Disable `.RData` saving.** Leftover workspace objects from a previous session can introduce silent errors. Starting each session with a clean workspace forces your scripts to be self-contained.

10.2.3 The shared setup pattern

A recurring pattern in multi-chapter projects is to place all shared setup code in a single file that every chapter sources at the top. In this book, that file is `R/_common.R`. It loads packages, sets `knitr` options, seeds the random number generator, and sources helper files for the publication theme and figure-saving utility. This pattern has three benefits:

- **Consistency.** Every chapter starts with the same packages, theme, and seed.
- **Maintainability.** Changing a default (e.g., switching from 300 to 600 DPI) requires editing one file, not forty.
- **Discoverability.** A reader who wants to understand the computational setup can read a single, short file.

10.3 Implementation in R

10.3.1 Project directory layout

The directory structure used in this book is a good template for any game theory research project:

```
# Display the project layout as a tidy table
layout <- tibble::tribble(
  ~Folder,           ~Purpose,

  "R/",              "Shared R scripts: _common.R,
   ↪  theme_publication.R, save_pub_fig.R, custom solvers",
  "*.Rmd",           "Bookdown chapters in the project root,
   ↪  ordered by _bookdown.yml",
  "style/",          "Shared HTML and PDF styling",
  "_generated/figures/", "Generated figures (PNG + PDF) from
   ↪  save_pub_fig()",
  "data/",           "Raw and processed data files",
  "A-*.Rmd to E-*.Rmd", "Appendices: refresher, solutions and
   ↪  glossary",
  "docs/",           "Rendered output (HTML, PDF and EPUB)",
  "renv/",           "Project-local package library managed by
   ↪  renv"
)

layout |>
  gt() |>
  tab_header(title = "Standard Project Directory Layout") |>
  cols_label(Folder = "Directory", Purpose = "Contents and
   ↪  purpose") |>
  cols_width(Folder ~ px(180))
```

10.3.2 Benchmarking vectorized vs. loop-based payoff computation

A common performance concern in game-theoretic computation is whether to use loops or vectorized operations when building payoff matrices. We benchmark both approaches to illustrate why vectorized code matters — and to demonstrate how to produce a publication-quality benchmark figure.

```
# Benchmark: computing a payoff matrix for an n-strategy
 ↪  symmetric game
```

Standard Project Directory Layout

Directory	Contents and purpose
R/	Shared R scripts: _common.R, theme_publication.R, save_pub_fig.R,
*.Rmd	Bookdown chapters in the project root, ordered by _bookdown.yml
style/	Shared HTML and PDF styling
_generated/figures/	Generated figures (PNG + PDF) from save_pub_fig()
data/	Raw and processed data files
A-*.Rmd to E-*.Rmd	Appendices: refresher, solutions and glossary
docs/	Rendered output (HTML, PDF and EPUB)
renv/	Project-local package library managed by renv

```
# Payoff function: u(i,j) = sin(i/n * pi) * cos(j/n * pi)
 ↪  (arbitrary smooth payoffs)

benchmark_payoff <- function(n, method = c("loop", "vectorized"))
 ↪  {

  method <- match.arg(method)

  if (method == "loop") {
    mat <- matrix(0, nrow = n, ncol = n)
    for (i in 1:n) {
      for (j in 1:n) {
        mat[i, j] <- sin(i / n * pi) * cos(j / n * pi)
      }
    }
  } else {
    rows <- seq_len(n)
    cols <- seq_len(n)
    mat <- outer(sin(rows / n * pi), cos(cols / n * pi))
  }
  mat
}

# Run benchmarks for various matrix sizes
sizes <- c(10, 50, 100, 200, 500, 1000)
n_reps <- 5

results <- map_dfr(sizes, function(n) {
  loop_times <- replicate(n_reps, {
    start <- proc.time()["elapsed"]
```

```
    benchmark_payoff(n, "loop")
    proc.time()["elapsed"] - start
  })

  vec_times <- replicate(n_reps, {
    start <- proc.time()["elapsed"]
    benchmark_payoff(n, "vectorized")
    proc.time()["elapsed"] - start
  })

  tibble(
    n = n,
    method = rep(c("Loop", "Vectorized"), each = n_reps),
    time_sec = c(loop_times, vec_times)
  )
})

# Summarize
bench_summary <- results |>
  group_by(n, method) |>
  summarise(
    mean_time = mean(time_sec),
    sd_time = sd(time_sec),
    .groups = "drop"
  )
```

```
p_bench <- ggplot(bench_summary,
                  aes(x = n, y = mean_time, colour = method,
                   ↪  shape = method)) +
  geom_line(linewidth = 0.8) +
  geom_point(size = 3) +
  geom_errorbar(aes(ymin = pmax(0, mean_time - sd_time),
                    ymax = mean_time + sd_time),
                width = 0.05, linewidth = 0.5) +
  scale_colour_manual(values = okabe_ito[c(1, 3)],
                      name = "Method") +
  scale_shape_manual(values = c(16, 17), name = "Method") +
  scale_x_log10(name = "Matrix dimension (n)",
                breaks = sizes,
                labels = scales::comma) +
  scale_y_log10(name = "Time (seconds)",
                labels = scales::label_number(accuracy = 0.0001))
                 ↪  +
  labs(title = "Payoff Matrix Construction: Loop vs. Vectorized")
   ↪  +
```

```
  theme_publication() +
  theme(legend.position = "bottom")

p_bench
```

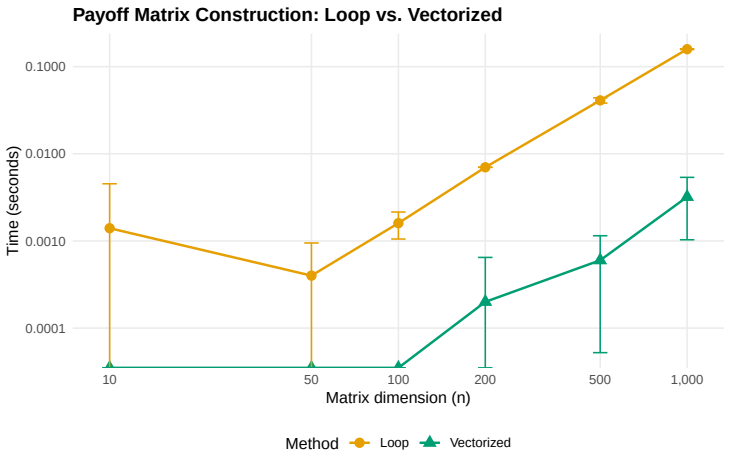


Figure 10.1: Computation time for building an n-by-n payoff matrix using a nested loop versus vectorized outer product. Vectorized operations maintain near-constant time across matrix sizes, while loop-based computation scales quadratically. Error bars show plus or minus one standard deviation over five replications.

10.3.3 The `_common.R` file explained

The shared setup file used in this book contains four key sections. Let us walk through each:

```
# --- Section 1: Load packages silently ---
suppressPackageStartupMessages({
  library(tidyverse)    # data manipulation and plotting
  library(here)         # project-root-relative paths
  library(glue)         # string interpolation
  library(gt)           # publication-quality tables
  library(scales)       # axis formatting helpers
})

# --- Section 2: knitr defaults ---
knitr::opts_chunk$set(
```

```
  dev     = c("png", "pdf"),   # dual output for HTML and PDF

  dpi     = 300,               # print-quality resolution
  fig.align = "center",
  fig.width  = 6,
  fig.height = 4,
  out.width  = "80%",
  comment    = "#>"            # prefix for R output lines
)

# --- Section 3: Reproducibility ---
set.seed(42)                    # global seed for all stochastic
 ↪  operations

# --- Section 4: Custom helpers ---
source(here::here("R", "theme_publication.R"))
source(here::here("R", "save_pub_fig.R"))
```

Section 1 loads the tidyverse ecosystem, which provides `dplyr`, `ggplot2`, `purrr`, `tibble`, and `stringr` — all essential for game-theoretic data wrangling and visualization. The `here` package ensures that file paths like `here("data", "tournament.csv")` work from any subdirectory. The `gt` package formats tables for publication, and `scales` provides axis label formatters.

Section 2 configures knitr to produce both PNG (for HTML output) and PDF (for LaTeX output) figures at 300 DPI. The `comment` option prefixes all R output with `#>`, making it visually distinct from code.

Section 3 sets a global random seed. This is critical for reproducibility in stochastic game-theoretic computations like Monte Carlo Shapley value estimation and evolutionary simulations.

Section 4 sources two helper files: `theme_publication.R` defines a clean ggplot2 theme with the Okabe-Ito colour palette, and `save_pub_fig.R` provides `save_pub_fig()` for dual PDF/PNG export.

10.3.4 Initializing `renv`

The `renv` workflow for a new project consists of three commands:

```
# 1. Initialize renv in a new project
renv::init()

# 2. After installing or updating packages, snapshot the current
 ↪  state
```

```
renv::snapshot()

# 3. On a new machine or after cloning, restore the exact package
 ↪  versions
renv::restore()
```

The `renv::init()` call creates a project-local library in `renv/library/`, a lockfile `renv.lock` that records every package name, version, and source, and an `.Rprofile` that activates `renv` automatically when the project is opened. The lockfile is committed to version control; the library itself is not (it is listed in `.gitignore`).

10.4 Worked example

We now walk through setting up a new game theory analysis project from scratch, as if starting a term paper on Nash equilibrium computations.

Step 1 — Create the project. In RStudio, select *File > New Project > New Directory > New Project*. Name it `nash-equilibrium-analysis` and check "Use renv with this project." In Positron, create a folder and run `renv::init()` from the R console.

Step 2 — Establish the directory layout.

```
# Create the standard directory structure
dirs <- c("R", "data", "images", "output")
for (d in dirs) dir.create(d, showWarnings = FALSE)
```

Step 3 — Write the shared setup file. Create `R/_common.R` with package loads and knitr defaults, following the pattern shown above. This file will be sourced at the top of every `.Rmd` chapter.

Step 4 — Install and snapshot dependencies.

```
# Install the packages needed for game theory work
install.packages(c("tidyverse", "here", "glue", "gt", "scales"))

# Lock the current state
renv::snapshot()
```

After this step, `renv.lock` contains a complete record of every package and its version. Sharing this file (along with the code) allows anyone to recreate the exact environment.

Step 5 — Create the first analysis file. Create `01-pd-analysis.Rmd` in the project root:

```
# At the top of 01-pd-analysis.Rmd:
source(here::here("R", "_common.R"))

# Define the Prisoner's Dilemma payoff matrix
pd_matrix <- matrix(c(3, 0, 5, 1), nrow = 2, byrow = TRUE,
                    dimnames = list(c("Cooperate", "Defect"),
                                    c("Cooperate", "Defect")))
pd_matrix
```

Step 6 — Configure IDE settings. In RStudio, go to *Tools* > *Global Options*:

- Under *General*, uncheck "Restore .RData into workspace at startup" and set "Save workspace to .RData on exit" to "Never."
- Under *Code* > *Display*, enable "Soft-wrap R source files."
- Under *Appearance*, choose a font with clear distinction between `0` and `O`, such as Fira Code or JetBrains Mono.

In Positron, equivalent settings live in the Settings editor (Ctrl+,). The key setting is `"r.restoreWorkspace": false`.

Step 7 — Verify reproducibility. Close and reopen the project. Run `renv::status()` to confirm all packages are synchronized. Then render the first analysis file. If it produces output without errors, the environment is correctly configured.

```
# Verify that our environment is properly configured
env_info <- tibble::tribble(
  ~Component,        ~Value,
  "R version",       paste(R.version$major, R.version$minor, sep
   ↪  = "."),
  "Platform",        R.version$platform,
  "tidyverse",       as.character(packageVersion("tidyverse")),
  "ggplot2",         as.character(packageVersion("ggplot2")),
  "here",            as.character(packageVersion("here")),
  "gt",              as.character(packageVersion("gt")),
  "Project root",    here::here()
)

env_info |>
  gt() |>
  tab_header(title = "Current R Environment") |>
  cols_label(Component = "Component", Value = "Detected value")
```

Current R Environment

Component	Detected value
R version	4.6.2
Platform	x86_64-pc-linux-gnu
tidyverse	2.0.0
ggplot2	4.0.3
here	1.0.2
gt	1.3.0
Project root	/home/runner/work/strategy-in-r/strategy-in-r

10.5 Extensions

The project setup described here can be extended in several directions:

- **Docker containers.** For maximum reproducibility, wrap the entire R environment in a Docker image. The `rocker` project provides pre-built images with R, RStudio Server, and the tidyverse. This guarantees identical results regardless of the host operating system.
- **Continuous integration.** Services like GitHub Actions can render your Bookdown book on every push, catching errors early. This book uses a GitHub Actions workflow that installs dependencies via `renv::restore()` and renders all chapters.
- **Targets pipelines.** The `targets` package extends reproducibility from package management to workflow management, tracking which outputs are up-to-date and re-running only what has changed. This is particularly valuable for long-running game-theoretic simulations.
- **Computational environments for teaching.** Posit Cloud (formerly RStudio Cloud) provides browser-based R environments with pre-configured packages, eliminating setup friction for students new to R. Each student gets an isolated workspace with the correct package versions.

For further reading on reproducible workflows in R, see Osborne (2004) for the mathematical context that motivates careful computation, and Shoham & Leyton-Brown (2009) for the algorithmic perspective on game-theoretic implementation.

Exercises

1. **Project initialization.** Create a new R project called `stag-hunt-analysis` with `renv` enabled. Install `tidyverse`, `here`, and `gt`. Write a `R/_common.R` file that loads these packages, sets a random seed, and configures knitr to produce PNG output at 300 DPI. Snapshot the environment with `renv::snapshot()` and verify that `renv.lock` contains all three packages.

2. **Vectorization practice.** Extend the benchmark from this chapter to include a third method that uses `purrr::map2_dbl()` inside a `tidyr::crossing()` grid. Plot all three methods on the same benchmark figure. Where does `purrr` fall relative to the loop and `outer()` approaches?

3. **Path portability.** A collaborator sends you a script that begins with `setwd("C:/Users/alice/games/")` and then reads a file with `read_csv("data/payoffs.csv")`. Explain why this will fail on your machine. Rewrite the first two lines using `here()` so that the script works for any collaborator who clones the repository.

4. **Custom knitr options.** Modify the `_common.R` pattern to include a conditional: if the output format is HTML, set `fig.width = 8`; if PDF, set `fig.width = 6`. *(Hint: use `knitr::is_html_output()`.)* Explain why different widths might be appropriate for the two formats.

5. **Dependency audit.** Run `renv::status()` on a project of your choice and interpret the output. Identify any packages that are installed but not recorded in the lockfile, and any that are recorded but not installed. Explain what `renv::snapshot()` and `renv::restore()` would each do in this situation.

Solutions appear in D.

Chapter 11

The GameTheory Package

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

A conceptual survey of the GameTheory R package for cooperative game theory, followed by from-scratch implementations of the Shapley value, Banzhaf index, and related solution concepts using base R and tidyverse.

Learning objectives

- Understand the API and capabilities of the GameTheory R package for cooperative game theory.
- Implement the Shapley value from scratch by enumerating all player permutations.
- Compute the Banzhaf power index for weighted voting games using base R.
- Compare computational results with closed-form analytical formulas.

11.1 Motivation

Consider the United Nations Security Council, where five permanent members hold veto power and ten non-permanent members do not. How much *real* power does each member wield? Raw vote counts are misleading — a permanent member's veto makes it far more influential than a simple headcount suggests. The **Shapley value**, a cornerstone of cooperative game theory, provides a principled answer: it measures each player's average marginal contribution across all possible orderings of coalition formation.

The `GameTheory` package on CRAN provides efficient, well-tested implementations of these computations. However, since it is not available in our current environment, this chapter first surveys its API conceptually and then implements equivalent functionality from scratch. Building these algorithms by hand deepens understanding of the combinatorial machinery behind cooperative solution concepts.

11.2 Theory

11.2.1 Cooperative games and characteristic functions

A **transferable utility (TU) cooperative game** is a pair (N, v) where $N = \{1, 2, \ldots, n\}$ is the set of players and $v : 2^N \to \mathbb{R}$ is the **characteristic function** satisfying $v(\emptyset) = 0$. The value $v(S)$ represents the total payoff that coalition $S \subseteq N$ can guarantee for itself.

Definition: Weighted Voting Game

A **weighted voting game** $[q; w_1, w_2, \ldots, w_n]$ has quota q and player weights w_i. A coalition S is **winning** if $\sum_{i \in S} w_i \geq q$, giving $v(S) = 1$; otherwise $v(S) = 0$.

11.2.2 The Shapley value

The **Shapley value** (Shapley, 1953) assigns to each player i a payoff $\phi_i(v)$ representing their expected marginal contribution:

$$\phi_i(v) = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!\,(n - |S| - 1)!}{n!} \left[v(S \cup \{i\}) - v(S) \right] \tag{11.1}$$

Equivalently, $\phi_i(v)$ is the average of player i's marginal contribution $v(S \cup \{i\}) - v(S)$ over all permutations of N, where S is the set of players who precede i in the permutation.

Theorem: Shapley's Characterization

The Shapley value is the unique allocation rule satisfying **efficiency** ($\sum_i \phi_i = v(N)$), **symmetry**, **additivity**, and the **null player** property.

11.2.3 The Banzhaf power index

The **Banzhaf power index** measures power by counting the number of coalitions in which a player is **pivotal** (turning a losing coalition into a winning one), divided by the total number of coalitions each player could join:

$$\beta_i = \frac{1}{2^{n-1}} \sum_{S \subseteq N \setminus \{i\}} [v(S \cup \{i\}) - v(S)] \tag{11.2}$$

Unlike the Shapley value, the raw Banzhaf indices do not necessarily sum to $v(N)$. The **normalized Banzhaf index** divides each β_i by $\sum_j \beta_j$.

11.2.4 The GameTheory package API

The `GameTheory` package provides functions including:

- `DefineGame()` — create a cooperative game from a characteristic function vector.
- `ShapleyValue()` — compute the Shapley value for any TU game.
- `Nucleolus()` — find the nucleolus (the allocation minimizing maximum dissatisfaction).
- `Core()` — compute and check non-emptiness of the core.
- `BanzhafValue()` — compute the Banzhaf power index.

Since this package is not installed, we now implement these concepts from scratch.

11.3 Implementation in R

11.3.1 Defining a characteristic function

We represent the characteristic function as a named numeric vector indexed by coalition labels. For n players, there are $2^n - 1$ non-empty coalitions.

```
# Build characteristic function for a weighted voting game
weighted_voting_game <- function(quota, weights) {
  n <- length(weights)
  players <- seq_len(n)
  # Enumerate all 2^n subsets (including empty set)
  v <- numeric(2^n)
  coalition_labels <- character(2^n)
  for (k in 0:(2^n - 1)) {
    members <- players[as.logical(intToBits(k)[1:n])]
    coalition_labels[k + 1] <- paste(members, collapse = ",")
    if (length(members) > 0 && sum(weights[members]) >= quota) {
      v[k + 1] <- 1
    }
```

```
  }
  names(v) <- coalition_labels
  v
}

# Three-player weighted voting game: Player 1 has veto power
weights <- c(3, 1, 1)
quota <- 4
v <- weighted_voting_game(quota, weights)
cat("Characteristic function [4; 3, 1, 1]:\n")
```

```
#> Characteristic function [4; 3, 1, 1]:
```

```
v[v > 0 | names(v) == ""]
```

```
#>         1,2   1,3 1,2,3
#>     0     1     1     1
```

11.3.2 Computing the Shapley value via permutations

```
# Generate all permutations of n elements
all_permutations <- function(n) {
  if (n == 1) return(matrix(1, nrow = 1, ncol = 1))
  prev <- all_permutations(n - 1)
  result <- matrix(0, nrow = factorial(n), ncol = n)
  idx <- 1
  for (i in seq_len(nrow(prev))) {
    for (pos in seq_len(n)) {
      new_perm <- append(prev[i, ], n, after = pos - 1)
      result[idx, ] <- new_perm
      idx <- idx + 1
    }
  }
  result
}

# Shapley value by averaging marginal contributions over all
 ↪  permutations
shapley_value <- function(v, n) {
  perms <- all_permutations(n)
  phi <- numeric(n)
```

```
  for (r in seq_len(nrow(perms))) {
    perm <- perms[r, ]
    coalition <- integer(0)
    for (pos in seq_along(perm)) {
      player <- perm[pos]
      # Value of coalition before player joins
      if (length(coalition) == 0) {
        v_before <- 0
      } else {
        key <- paste(sort(coalition), collapse = ",")
        v_before <- v[key]
      }
      # Value after player joins
      coalition <- c(coalition, player)
      key_after <- paste(sort(coalition), collapse = ",")
      v_after <- v[key_after]
      # Marginal contribution
      phi[player] <- phi[player] + (v_after - v_before)
    }
  }
  phi / nrow(perms)
}

shapley <- shapley_value(v, 3)
names(shapley) <- paste("Player", 1:3)
cat("Shapley values for [4; 3, 1, 1]:\n")
```

```
#> Shapley values for [4; 3, 1, 1]:
```

```
print(shapley)
```

```
#> Player 1 Player 2 Player 3
#>    0.667    0.167    0.167
```

```
cat(sprintf("\nSum of Shapley values: %.4f (should equal v(N) =
 ↪  %d)\n",
            sum(shapley), 1))
```

```
#>
#> Sum of Shapley values: 1.0000 (should equal v(N) = 1)
```

11.3.3 Shapley value via the analytical formula

```
# Shapley value using the combinatorial formula from
 ↪  \@ref(eq:shapley-value)
shapley_value_formula <- function(v, n) {
  players <- seq_len(n)
  phi <- numeric(n)

  for (i in players) {
    others <- setdiff(players, i)
    # Enumerate all subsets of others
    for (k in 0:(2^(n - 1) - 1)) {
      members <- others[as.logical(intToBits(k)[1:(n - 1)])]
      s <- length(members)
      weight <- factorial(s) * factorial(n - s - 1) /
 ↪  factorial(n)
      # v(S union {i}) - v(S)
      key_with <- paste(sort(c(members, i)), collapse = ",")
      key_without <- if (length(members) == 0) "" else
        paste(sort(members), collapse = ",")
      marginal <- v[key_with] - ifelse(key_without == "", 0,
 ↪  v[key_without])
      phi[i] <- phi[i] + weight * marginal
    }
  }
  phi
}

shapley_formula <- shapley_value_formula(v, 3)
names(shapley_formula) <- paste("Player", 1:3)
cat("Shapley values (analytical formula):\n")
```

```
#> Shapley values (analytical formula):
```

```
print(shapley_formula)
```

```
#> Player 1 Player 2 Player 3
#>    0.667    0.167    0.167
```

```
cat("\nMaximum difference from permutation method:",
    max(abs(shapley - shapley_formula)), "\n")
```

```
#>
#> Maximum difference from permutation method: 0
```

11.3.4 Computing the Banzhaf power index

```
banzhaf_index <- function(v, n) {
  players <- seq_len(n)
  beta <- numeric(n)

  for (i in players) {
    others <- setdiff(players, i)
    swings <- 0
    for (k in 0:(2^(n - 1) - 1)) {
      members <- others[as.logical(intToBits(k)[1:(n - 1)])]
      key_with <- paste(sort(c(members, i)), collapse = ",")
      key_without <- if (length(members) == 0) "" else
        paste(sort(members), collapse = ",")
      marginal <- v[key_with] - ifelse(key_without == "", 0,
 ↪  v[key_without])
      swings <- swings + marginal
    }
    beta[i] <- swings / 2^(n - 1)
  }
  beta
}

banzhaf <- banzhaf_index(v, 3)
names(banzhaf) <- paste("Player", 1:3)
cat("Raw Banzhaf indices for [4; 3, 1, 1]:\n")
```

```
#> Raw Banzhaf indices for [4; 3, 1, 1]:
```

```
print(banzhaf)
```

```
#> Player 1 Player 2 Player 3
#>     0.75     0.25     0.25
```

```
cat("\nNormalized Banzhaf indices:\n")
```

```
#>
#> Normalized Banzhaf indices:
```

```
print(banzhaf / sum(banzhaf))
```

```
#> Player 1 Player 2 Player 3
#>      0.6      0.2      0.2
```

11.3.5 Comparing power indices across coalition structures

```
games <- list(
  "Symmetric\n[2; 1,1,1]"  = list(quota = 2, weights = c(1, 1,
  ↪  1)),
  "Moderate veto\n[3; 2,1,1]" = list(quota = 3, weights = c(2, 1,
  ↪  1)),
  "Strong veto\n[4; 3,1,1]"  = list(quota = 4, weights = c(3, 1,
  ↪  1))
)

results <- map_dfr(names(games), function(game_name) {
  g <- games[[game_name]]
  v_game <- weighted_voting_game(g$quota, g$weights)
  shap <- shapley_value_formula(v_game, 3)
  banz <- banzhaf_index(v_game, 3)
  banz_norm <- banz / sum(banz)

  tibble(
    game = game_name,
    player = rep(paste("Player", 1:3), 2),
    index = rep(c("Shapley value", "Banzhaf (normalized)"), each
    ↪  = 3),
    value = c(shap, banz_norm)
  )
})

p_comparison <- ggplot(results,
                       aes(x = player, y = value, fill = index))
                       ↪  +
  geom_col(position = position_dodge(width = 0.7), width = 0.6) +
  facet_wrap(~ game, nrow = 1) +
  scale_fill_manual(values = c(okabe_ito[1], okabe_ito[2])) +
  scale_y_continuous(labels = label_percent(), limits = c(0, 1))
  ↪  +
  theme_publication() +
  labs(
    title = "Power Indices Across Weighted Voting Games",
    x = NULL,
    y = "Share of total power",
    fill = "Index"
  )
p_comparison
```

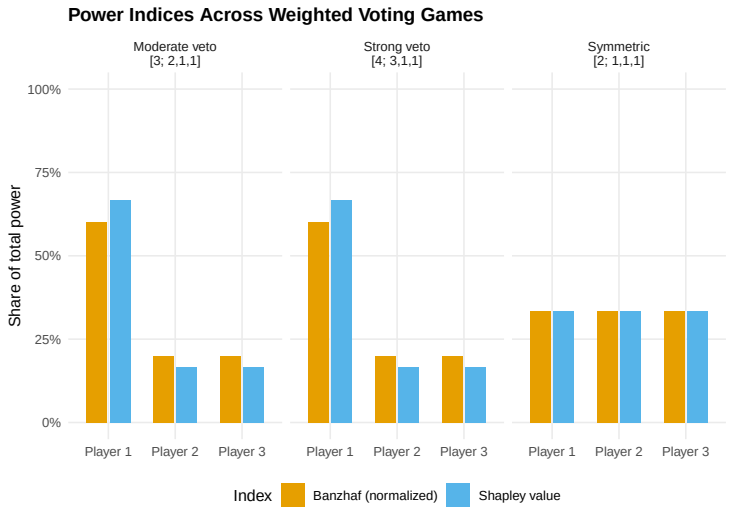


Figure 11.1: Shapley values and normalized Banzhaf indices for three different weighted voting games with three players. The veto player (Player 1) commands disproportionate power relative to their weight share.

```
save_pub_fig(p_comparison, "shapley-comparison", width = 7,
 ↪  height = 5)
```

As 11.1 shows, when the game is symmetric all players have equal power. As Player 1's weight advantage grows, both the Shapley value and the Banzhaf index assign them a disproportionately larger share of power — exceeding their raw weight share. In the strong veto game $[4; 3, 1, 1]$, Player 1 is essential for every winning coalition, which both indices capture clearly.

11.4 Worked example

11.4.1 A simplified EU Council vote

Consider a simplified three-country EU voting scenario: Germany (weight 4), France (weight 3), and Belgium (weight 1). A proposal passes with a quota of 5 votes.

```
eu_weights <- c(4, 3, 1)
eu_quota <- 5
eu_v <- weighted_voting_game(eu_quota, eu_weights)
```

```
cat("Game: [5; 4, 3, 1]\n\n")
```

```
#> Game: [5; 4, 3, 1]
```

```
cat("Winning coalitions:\n")
```

```
#> Winning coalitions:
```

```
winning <- eu_v[eu_v == 1 & nchar(names(eu_v)) > 0]
for (name in names(winning)) {
  members <- as.integer(strsplit(name, ",")[[1]])
  countries <- c("Germany", "France", "Belgium")[members]
  cat(sprintf("  {%s}  (total weight = %d)\n",
              paste(countries, collapse = ", "),
              sum(eu_weights[members])))
}
```

```
#>   {Germany, France}  (total weight = 7)
#>   {Germany, Belgium}  (total weight = 5)
#>   {Germany, France, Belgium}  (total weight = 8)
```

```
eu_shapley <- shapley_value_formula(eu_v, 3)
eu_banzhaf <- banzhaf_index(eu_v, 3)
eu_banzhaf_norm <- eu_banzhaf / sum(eu_banzhaf)

results_eu <- tibble(
  Country = c("Germany", "France", "Belgium"),
  Weight = eu_weights,
  `Weight share` = eu_weights / sum(eu_weights),
  `Shapley value` = eu_shapley,
  `Banzhaf (norm.)` = eu_banzhaf_norm
)
results_eu |>
  gt() |>
  fmt_percent(columns = c(`Weight share`, `Shapley value`,
   ↪  `Banzhaf (norm.)`),
              decimals = 1) |>
  tab_header(title = "Power analysis: [5; 4, 3, 1]")
```

Step-by-step Shapley calculation for Belgium (Player 3):

There are $3! = 6$ permutations. Belgium is pivotal only when it joins a coalition whose weight is in the range $[4, 4]$ — exactly enough that adding Belgium's

Power analysis: [5; 4, 3, 1]

Country	Weight	Weight share	Shapley value	Banzhaf (norm.)
Germany	4	50.0%	66.7%	60.0%
France	3	37.5%	16.7%	20.0%
Belgium	1	12.5%	16.7%	20.0%

weight of 1 reaches the quota of 5. This occurs when Germany alone precedes Belgium.

```
perms <- all_permutations(3)
cat("All permutations and Belgium's marginal contribution:\n\n")
```

```
#> All permutations and Belgium's marginal contribution:
```

```
for (r in seq_len(nrow(perms))) {
  perm <- perms[r, ]
  countries <- c("Germany", "France", "Belgium")[perm]
  pos_belgium <- which(perm == 3)
  predecessors <- if (pos_belgium > 1) perm[1:(pos_belgium - 1)]
 ↪  else integer(0)
  if (length(predecessors) == 0) {
    w_before <- 0
  } else {
    w_before <- sum(eu_weights[predecessors])
  }
  w_after <- w_before + eu_weights[3]
  pivotal <- (w_before < eu_quota) && (w_after >= eu_quota)
  cat(sprintf("  %s -> %s -> %s | Before Belgium: weight=%d,
   ↪  After: %d | %s\n",
              countries[1], countries[2], countries[3],
              w_before, w_after,
              ifelse(pivotal, "PIVOTAL", "")))
}
```

```
#>   Belgium -> France -> Germany | Before Belgium: weight=0, After: 1 |
#>   France -> Belgium -> Germany | Before Belgium: weight=3, After: 4 |
#>   France -> Germany -> Belgium | Before Belgium: weight=7, After: 8 |
#>   Belgium -> Germany -> France | Before Belgium: weight=0, After: 1 |
#>   Germany -> Belgium -> France | Before Belgium: weight=4, After: 5 | PIVOTAL
#>   Germany -> France -> Belgium | Before Belgium: weight=7, After: 8 |
```

```
cat(sprintf("\nShapley value for Belgium: %d/6 = %.4f\n",
            round(eu_shapley[3] * 6), eu_shapley[3]))
```

```
#>
#> Shapley value for Belgium: 1/6 = 0.1667
```

This confirms that Belgium is pivotal in exactly one of the six orderings (where Germany precedes Belgium and France follows), giving $\phi_3 = 1/6 \approx 16.7\%$ — notably higher than its raw weight share of $1/8 = 12.5\%$.

11.5 Extensions

- **Larger games.** For $n > 10$ players, permutation enumeration becomes infeasible ($n!$ grows super-exponentially). Sampling-based estimators randomly draw permutations and average marginal contributions, yielding Monte Carlo Shapley values with confidence intervals.
- **The nucleolus** provides a complementary solution concept that minimizes the maximum excess (dissatisfaction) of any coalition. We implement it in 12 using linear programming techniques.
- **Network games** extend cooperative games by restricting which coalitions can form based on a communication graph.
- **Non-transferable utility (NTU) games** relax the assumption that utility can be freely transferred among coalition members, leading to concepts such as the NTU Shapley value.

Exercises

1. **Unanimity games.** The unanimity game u_T for coalition T assigns $v(S) = 1$ if $T \subseteq S$ and $v(S) = 0$ otherwise. Prove analytically that $\phi_i(u_T) = 1/|T|$ for $i \in T$ and $\phi_i(u_T) = 0$ otherwise. Then verify your proof computationally using the `shapley_value()` function for $T = \{1, 2\}$ with $n = 3$ players.

2. **Four-player voting.** Consider the weighted voting game $[6;\ 4, 3, 2, 1]$. Compute the Shapley value and normalized Banzhaf index for all four players using the functions from 11.3. Which player has zero power according to both indices? Explain why intuitively.

3. **Monte Carlo Shapley.** For larger games, exact computation is infeasible. Write a function `shapley_monte_carlo(v, n, num_samples)` that estimates the Shapley value by randomly sampling `num_samples` permutations (using `sample()`). Test it on the game $[4;\ 3, 1, 1]$ and plot how the

estimates converge to the exact values as `num_samples` increases from 100 to 10,000.

Solutions appear in D.

Chapter 12

The CoopGame Package

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

A conceptual survey of the CoopGame R package for transferable utility games, followed by from-scratch implementations of the characteristic function, core computation via linear constraints, and the nucleolus, with a three-player production game worked example and simplex visualization.

Learning objectives

- Describe the features of the CoopGame R package for transferable utility (TU) games.
- Represent cooperative games via a characteristic function vector using binary coalition encoding.
- Compute the core of a three-player game by setting up linear inequality constraints in base R.
- Understand the nucleolus as the allocation minimizing maximum coalition dissatisfaction.
- Visualize the core as a feasible region on a simplex plot using ggplot2.

12.1 Motivation

Imagine three firms that each control a different resource needed for production: Firm A owns raw materials, Firm B operates a factory, and Firm C manages the distribution network. Any single firm can earn a modest profit alone, pairs

can do better by combining capabilities, and the grand coalition of all three generates the highest total profit. How should they divide the surplus?

This is a **transferable utility (TU) cooperative game**, and the central question is which allocations are stable — meaning no subgroup would prefer to break away. The set of all stable allocations is the **core**. The **nucleolus** refines this further by selecting the unique allocation that minimizes worst-case dissatisfaction.

The `CoopGame` package on CRAN provides a comprehensive toolkit for these problems: characteristic function encoding, solution concepts (Shapley value, nucleolus, tau-value), property checks (superadditivity, convexity, balancedness), and — most distinctively — core visualization on the simplex for three-player games. Since it is not installed, this chapter describes its API and implements the key algorithms from scratch.

12.2 Theory

12.2.1 Characteristic function representation

A TU game with n players is defined by $v : 2^N \to \mathbb{R}$ with $v(\emptyset) = 0$. The `CoopGame` package stores v as a vector of length $2^n - 1$, ordered by binary encoding:

$$\text{index}(S) = \sum_{i \in S} 2^{i-1} \tag{12.1}$$

For players $\{1, 2, 3\}$: index $1 = \{1\}$, $2 = \{2\}$, $3 = \{1, 2\}$, $4 = \{3\}$, $5 = \{1, 3\}$, $6 = \{2, 3\}$, $7 = \{1, 2, 3\}$.

12.2.2 The core

Definition: Core

The **core** of a TU game (N, v) is the set of allocations satisfying efficiency and coalition rationality:

$$\text{Core}(v) = \left\{ x \in \mathbb{R}^n : \sum_{i=1}^{n} x_i = v(N),\ \sum_{i \in S} x_i \geq v(S)\ \forall\, S \subset N \right\} \tag{12.2}$$

The core may be empty, a single point, or a convex polytope. The Bondareva-Shapley theorem states that the core is non-empty if and only if the game is **balanced**.

12.2.3 The nucleolus

The **nucleolus** (Schmeidler, 1969) selects the allocation that lexicographically minimizes the sorted vector of **excesses**:

$$e(S, x) = v(S) - \sum_{i \in S} x_i \tag{12.3}$$

A positive excess means coalition S is underpaid. The nucleolus solves:

$$\text{nuc}(v) = \arg\min_{x \in \mathcal{J}} \theta(x) \tag{12.4}$$

where $\theta(x)$ is the vector of excesses sorted in decreasing order. It always exists, is unique, and lies in the core when the core is non-empty.

12.2.4 The CoopGame package API

The `CoopGame` package provides: `createGame()` for defining games; `coreVertices()` and `drawCorePlot()` for core computation and visualization; `nucleolus()` and `shapleyValue()` for solution concepts; and `isConvexGame()`, `isSuperadditiveGame()`, `isBalancedGame()` for property checking. We now implement these from scratch.

12.3 Implementation in R

12.3.1 Coalition encoding and game definition

```
# Convert player set to coalition index (binary encoding)
coalition_idx <- function(members, n) sum(2^(members - 1))

# Convert index back to player set
idx_to_players <- function(idx, n)
 ↪  which(as.logical(intToBits(idx)[1:n]))

# Generate all non-empty coalitions
all_coalitions <- function(n) {
  lapply(1:(2^n - 1), function(k) idx_to_players(k, n))
}

# Three-player production game
# A (materials), B (factory), C (distribution)
```

```
# Index: 1={A}, 2={B}, 3={A,B}, 4={C}, 5={A,C}, 6={B,C},
 ↪  7={A,B,C}
v_prod <- c(10, 20, 50, 15, 40, 45, 80)
names(v_prod) <- c("{A}", "{B}", "{A,B}", "{C}", "{A,C}",
 ↪  "{B,C}", "{A,B,C}")

cat("Production game characteristic function:\n")
```

```
#> Production game characteristic function:
```

```
print(v_prod)
```

```
#>     {A}     {B}   {A,B}     {C}   {A,C}   {B,C} {A,B,C}
#>      10      20      50      15      40      45      80
```

```
cat(sprintf("\nSurplus from cooperation: %d - %d = %d\n",
            80, 10 + 20 + 15, 80 - 45))
```

```
#>
#> Surplus from cooperation: 80 - 45 = 35
```

12.3.2 Computing the core via linear constraints

For three players with efficiency $x_A + x_B + x_C = 80$, substituting $x_C = 80 - x_A - x_B$ reduces the core to a 2D feasibility problem:

```
# After substitution, the six constraints are:
#   x_A >= 10,  x_B >= 20,  x_A + x_B <= 65 (from x_C >= 15)
#   x_A + x_B >= 50,  x_B <= 40 (from x_A + x_C >= 40),  x_A <=
 ↪  35

core_constraints <- tribble(
  ~constraint,         ~description,
  "x_A >= 10",        "A individual rationality",
  "x_B >= 20",        "B individual rationality",
  "x_A + x_B <= 65",  "C individual rationality",
  "x_A + x_B >= 50",  "AB coalition rationality",
  "x_B <= 40",        "AC coalition rationality",
  "x_A <= 35",        "BC coalition rationality"
)
core_constraints |>
```

Core Constraints (after efficiency substitution)

constraint	description
x_A >= 10	A individual rationality
x_B >= 20	B individual rationality
x_A + x_B <= 65	C individual rationality
x_A + x_B >= 50	AB coalition rationality
x_B <= 40	AC coalition rationality
x_A <= 35	BC coalition rationality

```
  gt() |>
  tab_header(title = "Core Constraints (after efficiency
   ↪  substitution)") |>
  cols_align(align = "center")
```

12.3.3 Finding core vertices

```
# Find vertices by intersecting all pairs of boundary lines
boundary_intersections <- function() {
  bounds <- list(c(1,0,10), c(0,1,20), c(1,1,65),
                 c(1,1,50), c(0,1,40), c(1,0,35))
  vertices <- tibble(x_A = numeric(), x_B = numeric(), x_C =
 ↪  numeric())

  for (i in 1:(length(bounds) - 1)) {
    for (j in (i + 1):length(bounds)) {
      A_mat <- matrix(c(bounds[[i]][1:2], bounds[[j]][1:2]),
                      nrow = 2, byrow = TRUE)
      b_vec <- c(bounds[[i]][3], bounds[[j]][3])
      if (abs(det(A_mat)) < 1e-10) next
      sol <- solve(A_mat, b_vec)
      xa <- sol[1]; xb <- sol[2]; xc <- 80 - xa - xb

      feasible <- (xa >= 10 - 1e-10) && (xb >= 20 - 1e-10) &&
        (xa + xb <= 65 + 1e-10) && (xa + xb >= 50 - 1e-10) &&
        (xb <= 40 + 1e-10) && (xa <= 35 + 1e-10)
```

Vertices of the Core

x_A	x_B	x_C
10	40	30
30	20	30
35	20	25
25	40	15
35	30	15

```
      if (feasible) {
        vertices <- bind_rows(vertices,
          tibble(x_A = round(xa, 2), x_B = round(xb, 2), x_C =
            ↪  round(xc, 2)))
      }
    }
  }
  distinct(vertices)
}

core_verts <- boundary_intersections()
core_verts |>
  gt() |>
  tab_header(title = "Vertices of the Core") |>
  cols_align(align = "center")
```

12.3.4 Computing the nucleolus

```
# Compute excesses for all proper coalitions
compute_excesses <- function(x, v_vec, n) {
  coalitions <- all_coalitions(n)
  grand_idx <- 2^n - 1
  exc <- numeric(0); nms <- character(0)
  for (k in seq_along(coalitions)) {
    if (k == grand_idx) next
    S <- coalitions[[k]]
    exc <- c(exc, v_vec[k] - sum(x[S]))
    nms <- c(nms, paste(S, collapse = ","))
  }
  names(exc) <- nms
  exc
```

```
}

# Nucleolus via grid search (tractable for 3 players)
find_nucleolus <- function(v_vec, n, grid_size = 300) {
  grand_val <- v_vec[2^n - 1]
  best_x <- NULL; best_theta <- Inf
  xa_seq <- seq(10, 35, length.out = grid_size)
  xb_seq <- seq(20, 40, length.out = grid_size)

  for (xa in xa_seq) {
    for (xb in xb_seq) {
      xc <- grand_val - xa - xb
      if (xc < 15 - 1e-10 || xa + xb < 50 - 1e-10 ||
          xa + xb > 65 + 1e-10) next
      max_exc <- max(compute_excesses(c(xa, xb, xc), v_vec, n))
      if (max_exc < best_theta) {
        best_theta <- max_exc; best_x <- c(xa, xb, xc)
      }
    }
  }
  names(best_x) <- c("A", "B", "C")
  best_x
}

nuc <- find_nucleolus(v_prod, 3)
cat("Nucleolus (approximate):\n")
```

```
#> Nucleolus (approximate):
```

```
cat(sprintf("  A: %.2f, B: %.2f, C: %.2f (sum: %.2f)\n",
            nuc[1], nuc[2], nuc[3], sum(nuc)))
```

```
#>   A: 25.05, B: 32.44, C: 22.51 (sum: 80.00)
```

```
cat("\nExcesses at the nucleolus:\n")
```

```
#>
#> Excesses at the nucleolus:
```

```
print(round(sort(compute_excesses(nuc, v_prod, 3), decreasing =
 ↪  TRUE), 2))
```

```
#>    1,2      3    1,3    2,3      2      1
#>  -7.49  -7.51  -7.56  -9.95 -12.44 -15.05
```

12.3.5 Shapley value and core visualization

```
# Shapley value via permutation enumeration
shapley_prod <- function() {
  perms <- list(c(1,2,3), c(1,3,2), c(2,1,3),
                c(2,3,1), c(3,1,2), c(3,2,1))
  phi <- c(0, 0, 0)
  get_v <- function(m) {
    if (length(m) == 0) return(0)
    v_prod[coalition_idx(m, 3)]
  }
  for (perm in perms) {
    coal <- integer(0)
    for (p in perm) {
      v_before <- get_v(coal)
      coal <- c(coal, p)
      phi[p] <- phi[p] + (get_v(sort(coal)) - v_before)
    }
  }
  phi / length(perms)
}
shap <- shapley_prod()

# Barycentric to Cartesian simplex coordinates
to_simplex <- function(xA, xB, xC) {
  total <- xA + xB + xC
  a <- xA / total; b <- xB / total; cc <- xC / total
  tibble(x = b + cc / 2, y = cc * sqrt(3) / 2)
}

simplex_verts <- tibble(
  label = c("A gets 80", "B gets 80", "C gets 80"),
  x = c(0, 1, 0.5), y = c(0, 0, sqrt(3) / 2)
)

core_simplex <- map_dfr(seq_len(nrow(core_verts)), function(i) {
  to_simplex(core_verts$x_A[i], core_verts$x_B[i],
    ↪  core_verts$x_C[i])
})
hull_order <- chull(core_simplex$x, core_simplex$y)
core_hull <- core_simplex[c(hull_order, hull_order[1]), ]

shap_pt <- to_simplex(shap[1], shap[2], shap[3])
nuc_pt <- to_simplex(nuc[1], nuc[2], nuc[3])
```

```
p_simplex <- ggplot() +
  geom_polygon(data = simplex_verts, aes(x = x, y = y),
               fill = "grey95", colour = "grey50", linewidth =
                ↪  0.5) +
  geom_polygon(data = core_hull, aes(x = x, y = y),
               fill = okabe_ito[3], alpha = 0.3,
               colour = okabe_ito[3], linewidth = 1) +
  geom_point(data = core_simplex, aes(x = x, y = y),
             colour = okabe_ito[3], size = 2) +
  geom_point(data = shap_pt, aes(x = x, y = y),
             colour = okabe_ito[1], size = 4, shape = 17) +
  annotate("text", x = shap_pt$x + 0.04, y = shap_pt$y + 0.02,
           label = sprintf("Shapley\n(%.1f, %.1f, %.1f)",
                           shap[1], shap[2], shap[3]),
           colour = okabe_ito[1], size = 3, fontface = "bold",
            ↪  hjust = 0) +
  geom_point(data = nuc_pt, aes(x = x, y = y),
             colour = okabe_ito[5], size = 4, shape = 15) +
  annotate("text", x = nuc_pt$x - 0.04, y = nuc_pt$y - 0.03,
           label = sprintf("Nucleolus\n(%.1f, %.1f, %.1f)",
                           nuc[1], nuc[2], nuc[3]),
           colour = okabe_ito[5], size = 3, fontface = "bold",
            ↪  hjust = 1) +
  geom_text(data = simplex_verts, aes(x = x, y = y, label =
   ↪  label),
            vjust = c(1.8, 1.8, -1), size = 3, colour = "grey30")
             ↪  +
  annotate("text", x = 0.5, y = 0.25, label = "Core",
           colour = okabe_ito[3], size = 5, fontface = "bold") +
  coord_fixed() +
  theme_publication() +
  theme(axis.text = element_blank(), axis.title =
   ↪  element_blank(),
        axis.ticks = element_blank(), panel.grid =
         ↪  element_blank()) +
  labs(title = "Core of the Three-Player Production Game")

p_simplex
```

```
save_pub_fig(p_simplex, "core-simplex", width = 7, height = 6)
```

12.1 shows the core as a shaded polygon on the efficiency simplex. Every point inside represents a stable allocation where no coalition can profitably deviate. Both the Shapley value and nucleolus lie inside, confirming stability.

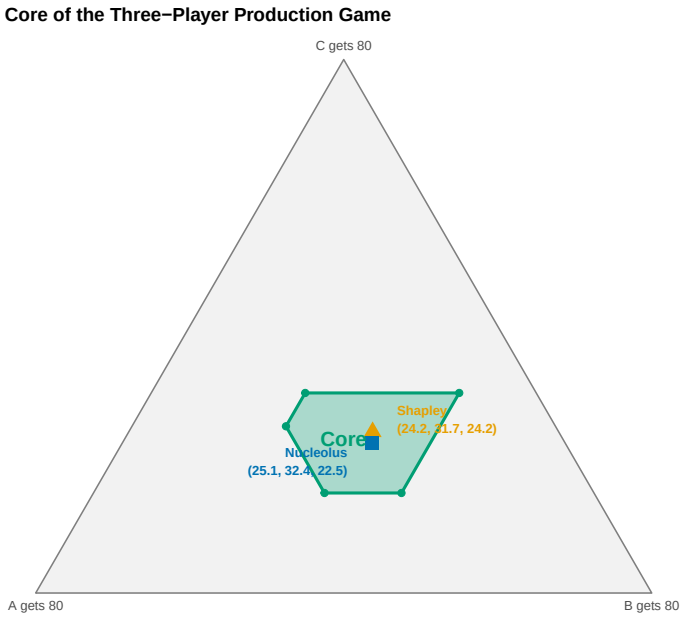


Figure 12.1: The core of the three-player production game (shaded polygon) within the efficiency simplex. The Shapley value (triangle) and nucleolus (square) both lie inside the core, confirming both solution concepts yield stable allocations.

12.4 Worked example

12.4.1 Step-by-step core membership check

We verify which allocations are in the core by testing all constraints.

```
check_core <- function(x, v_vec, n, labels) {
  grand <- v_vec[2^n - 1]
  coalitions <- all_coalitions(n)
  all_pass <- abs(sum(x) - grand) < 0.01
  cat(sprintf("Efficiency: sum = %.2f -- %s\n",
              sum(x), ifelse(all_pass, "PASS", "FAIL")))
  for (k in 1:(2^n - 2)) {
    S <- coalitions[[k]]
    pass <- sum(x[S]) >= v_vec[k] - 0.01
    if (!pass) all_pass <- FALSE
    cat(sprintf("  v(%s) = %d, x(%s) = %.2f -- %s\n",
                paste(labels[S], collapse = ","), v_vec[k],
                paste(labels[S], collapse = ","), sum(x[S]),
                ifelse(pass, "PASS", "FAIL")))
  }
  cat(sprintf("In core: %s\n\n", ifelse(all_pass, "YES", "NO")))
}

labels <- c("A", "B", "C")
equal <- rep(80 / 3, 3)

cat("=== Shapley value ===\n")
```

```
#> === Shapley value ===
```

```
check_core(shap, v_prod, 3, labels)
```

```
#> Efficiency: sum = 80.00 -- PASS
#>   v(A) = 10, x(A) = 24.17 -- PASS
#>   v(B) = 20, x(B) = 31.67 -- PASS
#>   v(A,B) = 50, x(A,B) = 55.83 -- PASS
#>   v(C) = 15, x(C) = 24.17 -- PASS
#>   v(A,C) = 40, x(A,C) = 48.33 -- PASS
#>   v(B,C) = 45, x(B,C) = 55.83 -- PASS
#> In core: YES
```

```
cat("=== Nucleolus ===\n")
```

```
#> === Nucleolus ===
```

```
check_core(nuc, v_prod, 3, labels)
```

```
#> Efficiency: sum = 80.00 -- PASS
#>   v(A) = 10, x(A) = 25.05 -- PASS
#>   v(B) = 20, x(B) = 32.44 -- PASS
#>   v(A,B) = 50, x(A,B) = 57.49 -- PASS
#>   v(C) = 15, x(C) = 22.51 -- PASS
#>   v(A,C) = 40, x(A,C) = 47.56 -- PASS
#>   v(B,C) = 45, x(B,C) = 54.95 -- PASS
#> In core: YES
```

```
cat("=== Equal division ===\n")
```

```
#> === Equal division ===
```

```
check_core(equal, v_prod, 3, labels)
```

```
#> Efficiency: sum = 80.00 -- PASS
#>   v(A) = 10, x(A) = 26.67 -- PASS
#>   v(B) = 20, x(B) = 26.67 -- PASS
#>   v(A,B) = 50, x(A,B) = 53.33 -- PASS
#>   v(C) = 15, x(C) = 26.67 -- PASS
#>   v(A,C) = 40, x(A,C) = 53.33 -- PASS
#>   v(B,C) = 45, x(B,C) = 53.33 -- PASS
#> In core: YES
```

The Shapley value and nucleolus pass all constraints. Equal division (≈ 26.67 each) fails because $x_B + x_C \approx 53.33 < 45$ is satisfied, but Firm B with only 26.67 would prefer to partner with C for 45 — the coalition rationality constraint $x_A + x_B \geq 50$ is violated. This illustrates that simplistic "fair" divisions are often unstable.

Firm B receives the largest share under both solution concepts, reflecting its central role in the two most valuable pair coalitions ($v(\{A, B\}) = 50$ and $v(\{B, C\}) = 45$). Firm A, despite the lowest stand-alone value, earns well above 10 because it generates substantial synergies with B.

12.5 Extensions

- **Linear programming for the nucleolus.** Our grid-search works for three players but does not scale. The standard algorithm solves a sequence of LPs, minimizing the maximum excess at each stage. The `CoopGame` package uses `lpSolve` for this.
- **Game properties.** `CoopGame` tests monotonicity, superadditivity, convexity, and balancedness. Convex games are particularly well-behaved: the core is always non-empty and the Shapley value lies within it.
- **Additional solution concepts.** Beyond Shapley and nucleolus, `CoopGame` computes the tau-value, Gately point, per-capita nucleolus, and equal surplus division, each with different fairness properties.
- **Power indices.** The Shapley value computation connects to 11, where we compute power indices for weighted voting games.

Exercises

1. **Empty core.** Consider a three-player game with $v(\{i\}) = 0$ for all i, $v(\{i, j\}) = 1$ for all pairs, and $v(\{1, 2, 3\}) = 1$. Show that the core is empty by demonstrating that the constraints in (12.2) are infeasible. What does this imply about the stability of the grand coalition?

2. **Convexity check.** Write a function that checks whether a game is convex (marginal contributions are non-decreasing). Test it on the production game. Is it convex? If not, find a player and pair of coalitions that violate the condition.

3. **Shrinking the pie.** Change the grand coalition value from 80 to 60, keeping all pair values the same. Recompute the core vertices. Does the core shrink, or does it become empty? If it becomes empty, explain which constraint(s) are violated and why the game becomes unbalanced.

Solutions appear in D.

Chapter 13

The gtree Package

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

> Extensive-form games and backward induction with the gtree package's domain-specific language — plus a from-scratch pure-R implementation for solving game trees when external solvers are unavailable.

Learning objectives

- Describe the gtree package's domain-specific language (DSL) for specifying extensive-form games as nested stage structures.
- Explain how gtree interfaces with Gambit command-line solvers to compute Nash equilibria of sequential games.
- Build a pure-R game tree representation using nested lists and solve it via backward induction.
- Apply backward induction to the ultimatum game and identify the subgame-perfect equilibrium.
- Visualize a game tree with the equilibrium path highlighted.

13.1 Motivation

Many strategic interactions unfold sequentially: a firm enters a market and the incumbent decides whether to fight or accommodate; a legislature passes a bill and the executive signs or vetoes it; a buyer makes an offer and the seller accepts or rejects. These situations are naturally modeled as *extensive-form games* — tree structures where nodes represent decisions and edges represent actions.

In 6 we introduced game trees and backward induction by hand. That approach works for textbook examples, but as games grow in size (more rounds, more players, more actions per node), manual tree construction becomes error-prone. The **gtree** package, developed by Sebastian Kranz, addresses this gap by providing a compact domain-specific language (DSL) for defining extensive-form games. It constructs the game tree internally and, when the Gambit solver suite is installed, calls industrial-strength algorithms to find equilibria.

However, gtree depends on external binaries (Gambit) and is not always available in every R environment. This chapter therefore pursues a dual strategy: we describe gtree's design and DSL so that readers can evaluate it for their own work, and we implement a complete game tree engine from scratch in pure R. The engine represents trees as nested lists, solves them by backward induction, and produces publication-quality visualizations of the equilibrium path — all without any dependencies beyond base R and the tidyverse.

13.2 Theory

13.2.1 The gtree DSL

The gtree package models an extensive-form game as a sequence of **stages**. Each stage specifies:

- **Who moves**: a named player or "nature" (for chance nodes).
- **What actions are available**: possibly conditional on variables set in earlier stages.
- **What payoffs result**: formulas that reference earlier actions.

A typical gtree specification looks like this (pseudocode, since gtree is not installed):

```
game <- new_game(
  players = c("Proposer", "Responder"),
  stages = list(
    stage("offer",
      player = "Proposer",
      actions = list(action("split", c("Fair", "Unfair")))
    ),
    stage("response",
      player = "Responder",
      actions = list(action("answer", c("Accept", "Reject")))
    )
  ),
  payoffs = list(
```

```
    payoff("Proposer",   ~ case_when(
      answer == "Reject" ~ 0,
      split == "Fair"    ~ 5,
      TRUE               ~ 8)),
    payoff("Responder",  ~ case_when(
      answer == "Reject" ~ 0,
      split == "Fair"    ~ 5,
      TRUE               ~ 2))
  )
)
```

The key design insight is that `condition` clauses allow stages to be contingent on prior choices, so the analyst describes the game linearly even though the underlying tree branches. The package internally expands this specification into a full game tree with correctly linked information sets.

13.2.2 Gambit integration

Once a game is defined, gtree can export it to Gambit's `.efg` (extensive-form game) format and invoke Gambit's command-line solvers:

- `gambit-enumpure` enumerates all pure-strategy Nash equilibria.
- `gambit-enummixed` enumerates all mixed-strategy equilibria for two-player games.
- `gambit-lcp` uses the Lemke–Howson algorithm for bimatrix games (see 15).
- `gambit-logit` computes quantal response equilibria.

This makes gtree a convenient R front-end for industrial-strength solvers without requiring the analyst to learn Gambit's file format or command-line interface.

13.2.3 Backward induction for perfect-information games

For games of **perfect information** — where every information set is a singleton — backward induction provides an exact solution. The algorithm works from the terminal nodes back to the root: at each decision node, the moving player selects the action that maximizes their own payoff, given that all subsequent players will do the same.

Definition: Subgame-Perfect Equilibrium (SPE)

A strategy profile is a **subgame-perfect equilibrium** if it constitutes a Nash equilibrium in every subgame of the original game. For finite games of perfect information, backward induction yields the unique SPE (when payoffs at terminal nodes are distinct).

The time complexity of backward induction is $O(n)$ where n is the number of nodes in the tree, since each node is visited exactly once. For a tree with branching factor b and depth d, the total number of nodes is:

$$n = \frac{b^{d+1} - 1}{b - 1} \tag{13.1}$$

This exponential growth in tree size is the fundamental computational challenge — but backward induction remains efficient because it processes each node exactly once.

13.2.4 The ultimatum game

The **ultimatum game** is a canonical example in behavioral economics. A Proposer has a sum of money (say, 10 units) and offers a split to a Responder. The Responder either Accepts (both receive their shares) or Rejects (both receive nothing). Despite the intuition that any positive offer should be accepted, experimental evidence consistently shows that offers below 20–30% are frequently rejected.

The game-theoretic prediction under backward induction is stark: the Responder should accept any positive offer (since something is better than nothing), and therefore the Proposer should offer the minimum possible amount. This divergence between theory and behavior makes the ultimatum game a rich pedagogical example.

13.3 Implementation in R

13.3.1 A flexible game tree representation

We represent game trees as nested lists. Each node is either a *decision node* (with a player and children) or a *terminal node* (with payoffs for all players).

```
# A terminal node stores final payoffs for all players
make_terminal <- function(payoffs) {
  list(type = "terminal", payoffs = payoffs)
}

# A decision node stores who moves and a named list of children
make_decision <- function(player, children) {
  list(type = "decision", player = player, children = children)
}
```

```
# Backward induction: recursively solve from leaves to root
solve_by_backward_induction <- function(node) {
  if (node$type == "terminal") {
    return(list(payoffs = node$payoffs, path = list()))
  }

  # Recursively solve all children
  solutions <- lapply(node$children, solve_by_backward_induction)

  # Current player picks child maximizing their own payoff

  player <- node$player
  child_payoffs <- sapply(solutions, function(s)
↪   s$payoffs[player])
  best_idx <- which.max(child_payoffs)
  best_action <- names(node$children)[best_idx]
  best_solution <- solutions[[best_idx]]

  list(
    payoffs = best_solution$payoffs,
    path = c(
      list(list(player = player, action = best_action)),
      best_solution$path
    )
  )
}
```

13.3.2 Building the ultimatum game tree

```
# Ultimatum game: Proposer offers Fair (5,5) or Unfair (8,2)
# Responder Accepts or Rejects each offer

ultimatum_game <- make_decision(1, list(
  "Fair (5,5)" = make_decision(2, list(
    "Accept" = make_terminal(c(5, 5)),
    "Reject" = make_terminal(c(0, 0))
  )),
  "Unfair (8,2)" = make_decision(2, list(
    "Accept" = make_terminal(c(8, 2)),
    "Reject" = make_terminal(c(0, 0))
  ))
))
```

```
result <- solve_by_backward_induction(ultimatum_game)
cat("Subgame-perfect equilibrium payoffs:", result$payoffs,
 ↪  "\n\n")
```

```
#> Subgame-perfect equilibrium payoffs: 8 2
```

```
cat("SPE path:\n")
```

```
#> SPE path:
```

```
for (step in result$path) {
  player_label <- c("Proposer", "Responder")[step$player]
  cat(glue("  {player_label} chooses: {step$action}"), "\n")
}
```

```
#>   Proposer chooses: Unfair (8,2)
#>   Responder chooses: Accept
```

As predicted by the theory, backward induction selects the Unfair offer (since $8 > 5$ for the Proposer) and the Responder Accepts (since $2 > 0$). The SPE payoffs are (8, 2).

13.3.3 Visualizing the game tree

To build a publication-quality game tree figure, we first flatten the tree into a data frame of nodes and edges, then use `ggplot2` to render it with the equilibrium path highlighted.

```
# Flatten tree into node and edge data frames for plotting
flatten_tree <- function(node, node_id = 1, depth = 0, x_center =
 ↪  0,
                         x_spread = 2, eq_path = NULL) {
  nodes <- tibble()
  edges <- tibble()

  if (node$type == "terminal") {
    payoff_label <- paste0("(", paste(node$payoffs, collapse = ",
 ↪  "), ")")
    nodes <- tibble(
      id = node_id, depth = depth, x = x_center,
      label = payoff_label, node_type = "terminal", player =
       ↪  NA_integer_
```

```
    )
    return(list(nodes = nodes, edges = edges, next_id = node_id +
     ↪  1))
  }

  # Decision node
  player_labels <- c("Proposer", "Responder")
  nodes <- tibble(
    id = node_id, depth = depth, x = x_center,
    label = player_labels[node$player], node_type = "decision",
    player = node$player
  )

  n_children <- length(node$children)
  child_positions <- seq(-x_spread / 2, x_spread / 2, length.out
↪  = n_children)
  next_id <- node_id + 1

  for (i in seq_along(node$children)) {
    action_name <- names(node$children)[i]
    child_node <- node$children[[i]]

    # Check if this edge is on the equilibrium path
    on_eq_path <- FALSE
    if (!is.null(eq_path) && length(eq_path) > 0) {
      step <- eq_path[[1]]
      if (step$player == node$player && step$action ==
       ↪  action_name) {
        on_eq_path <- TRUE
      }
    }

    child_result <- flatten_tree(
      child_node, next_id, depth + 1,
      x_center + child_positions[i], x_spread / 2,
      eq_path = if (on_eq_path && length(eq_path) > 1)
       ↪  eq_path[-1] else
              if (on_eq_path) list() else NULL
    )

    new_edge <- tibble(
      from_id = node_id, to_id = next_id,
      from_x = x_center, from_y = -depth,
      to_x = x_center + child_positions[i], to_y = -(depth + 1),
      action = action_name, on_eq_path = on_eq_path
    )
```

```
    edges <- bind_rows(edges, new_edge, child_result$edges)
    nodes <- bind_rows(nodes, child_result$nodes)
    next_id <- child_result$next_id
  }

  list(nodes = nodes, edges = edges, next_id = next_id)
}
```

```
tree_data <- flatten_tree(
  ultimatum_game, eq_path = result$path, x_spread = 4
)

nodes_df <- tree_data$nodes |>
  mutate(y = -depth)

edges_df <- tree_data$edges

p_tree <- ggplot() +
  # Draw edges (non-equilibrium first, then equilibrium on top)
  geom_segment(
    data = edges_df |> filter(!on_eq_path),
    aes(x = from_x, y = from_y, xend = to_x, yend = to_y),
    colour = "grey60", linewidth = 0.7
  ) +
  geom_segment(
    data = edges_df |> filter(on_eq_path),
    aes(x = from_x, y = from_y, xend = to_x, yend = to_y),
    colour = okabe_ito[1], linewidth = 1.5
  ) +
  # Action labels on edges
  geom_label(
    data = edges_df,
    aes(x = (from_x + to_x) / 2, y = (from_y + to_y) / 2, label
     ↪  = action),
    size = 2.8, label.size = 0, fill = "white", alpha = 0.85
  ) +
  # Decision nodes
  geom_point(
    data = nodes_df |> filter(node_type == "decision"),
    aes(x = x, y = y, colour = factor(player)),
    size = 8
  ) +
  # Decision node labels
  geom_text(
```

```
    data = nodes_df |> filter(node_type == "decision"),
    aes(x = x, y = y, label = label),
    size = 2.5, colour = "white", fontface = "bold"
  ) +
  # Terminal node labels (payoffs)
  geom_label(
    data = nodes_df |> filter(node_type == "terminal"),
    aes(x = x, y = y, label = label),
    size = 3, fill = "grey95", label.padding = unit(0.2, "lines")
  ) +
  scale_colour_manual(
    values = c("1" = okabe_ito[1], "2" = okabe_ito[2]),
    labels = c("Proposer", "Responder"),
    name = "Player"
  ) +
  coord_cartesian(
    xlim = c(-3, 3),
    ylim = c(-2.5, 0.3)
  ) +
  labs(title = "Ultimatum Game: Subgame-Perfect Equilibrium
   ↪  Path") +
  theme_publication() +
  theme(
    axis.text = element_blank(),
    axis.title = element_blank(),
    axis.ticks = element_blank(),
    panel.grid = element_blank()
  )

p_tree
```

```
save_pub_fig(p_tree, "ultimatum-game-tree", width = 8, height =
 ↪  5.5)
```

The figure makes the structure of the SPE transparent. At each of the Responder's decision nodes, Accepting yields a positive payoff while Rejecting yields zero — so the Responder always Accepts. Knowing this, the Proposer selects the split that maximizes their own share, choosing the Unfair offer.

13.4 Worked example

Let us extend the ultimatum game to allow three possible offers and solve it step by step.

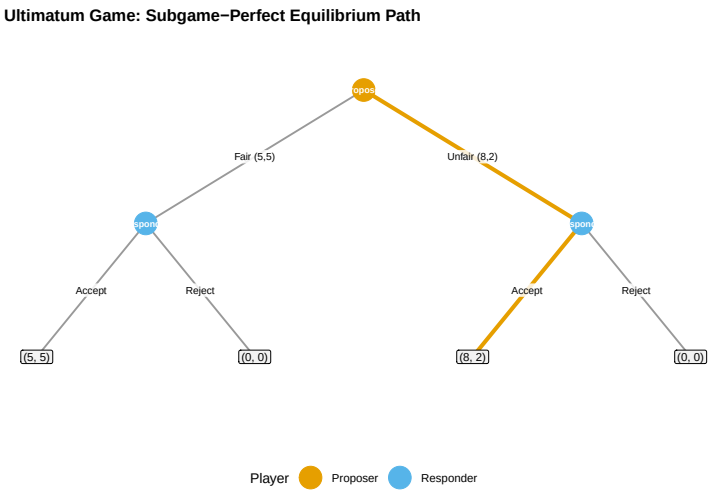


Figure 13.1: Game tree for the ultimatum game. The Proposer (orange node) chooses between a Fair and Unfair split; the Responder (blue nodes) then Accepts or Rejects. The bold orange path marks the subgame-perfect equilibrium: Proposer offers Unfair (8, 2) and Responder Accepts.

Setup. The Proposer has 10 units and can offer three splits:

Offer	Proposer receives	Responder receives
Generous	4	6
Fair	5	5
Greedy	9	1

The Responder observes the offer and chooses Accept or Reject. If Rejected, both receive 0.

```
# Three-offer ultimatum game
ultimatum_3 <- make_decision(1, list(
  "Generous (4,6)" = make_decision(2, list(
    "Accept" = make_terminal(c(4, 6)),
    "Reject" = make_terminal(c(0, 0))
  )),
  "Fair (5,5)" = make_decision(2, list(
    "Accept" = make_terminal(c(5, 5)),
    "Reject" = make_terminal(c(0, 0))
  )),
  "Greedy (9,1)" = make_decision(2, list(
    "Accept" = make_terminal(c(9, 1)),
```

```
    "Reject" = make_terminal(c(0, 0))
  ))
))

result_3 <- solve_by_backward_induction(ultimatum_3)
cat("SPE payoffs:", result_3$payoffs, "\n\n")
```

```
#> SPE payoffs: 9 1
```

```
cat("SPE path:\n")
```

```
#> SPE path:
```

```
for (step in result_3$path) {
  player_label <- c("Proposer", "Responder")[step$player]
  cat(glue("  {player_label} chooses: {step$action}"), "\n")
}
```

```
#>   Proposer chooses: Greedy (9,1)
#>   Responder chooses: Accept
```

Step-by-step reasoning:

1. **Responder's subgames.** At each of the three decision nodes, the Responder compares Accept vs. Reject. Since Accepting always yields a positive payoff (6, 5, or 1) while Rejecting yields 0, the Responder Accepts every offer.

2. **Proposer's choice.** Knowing the Responder will Accept any offer, the Proposer compares their own payoffs: 4 (Generous), 5 (Fair), or 9 (Greedy). The Proposer selects the Greedy offer.

3. **SPE outcome.** The Proposer offers (9, 1) and the Responder Accepts, yielding equilibrium payoffs $(9, 1)$.

This result highlights the starkness of the game-theoretic prediction: rational, self-interested players reach an extreme outcome. The large experimental literature on ultimatum games shows that this prediction fails descriptively — Responders frequently reject low offers and Proposers often offer 40–50% — which motivates models incorporating fairness preferences, spite, and social norms.

13.4.1 Scaling analysis

How does backward induction perform as the game tree grows? We generate random trees of varying depth and branching factor and record solution times.

```
# Generate random perfect-information game trees
generate_random_tree <- function(depth, branching, n_players = 2)
 ↪  {
  if (depth == 0) {
    return(make_terminal(runif(n_players, -5, 5)))
  }
  player <- ((depth - 1) %% n_players) + 1
  children <- setNames(
    lapply(seq_len(branching), function(i) {
      generate_random_tree(depth - 1, branching, n_players)
    }),
    paste0("A", seq_len(branching))
  )
  make_decision(player, children)
}

configs <- expand.grid(depth = 2:8, branching = c(2, 3)) |>
  as_tibble() |>
  mutate(n_terminals = branching^depth, time_ms = NA_real_)

for (i in seq_len(nrow(configs))) {
  tree <- generate_random_tree(configs$depth[i],
 ↪  configs$branching[i])
  elapsed <-
 ↪  system.time(solve_by_backward_induction(tree))["elapsed"]
  configs$time_ms[i] <- elapsed * 1000
}

configs |>
  mutate(branching = factor(branching)) |>
  select(depth, branching, n_terminals, time_ms) |>
  gt() |>
  tab_header(title = "Backward Induction: Solution Time by Tree
   ↪  Size") |>
  fmt_number(columns = time_ms, decimals = 2) |>
  fmt_number(columns = n_terminals, use_seps = TRUE, decimals =
   ↪  0) |>
  cols_label(
    depth = "Depth", branching = "Branching",
    n_terminals = "Terminal Nodes", time_ms = "Time (ms)"
  )
```

Backward Induction: Solution Time by Tree Size

Depth	Branching	Terminal Nodes	Time (ms)
2	2	4	0.00
3	2	8	1.00
4	2	16	1.00
5	2	32	1.00
6	2	64	1.00
7	2	128	1.00
8	2	256	2.00
2	3	9	1.00
3	3	27	0.00
4	3	81	0.00
5	3	243	2.00
6	3	729	4.00
7	3	2,187	12.00
8	3	6,561	40.00

```
p_scaling <- configs |>
  mutate(branching = factor(branching,
    labels = c("Binary (b=2)", "Ternary (b=3)")
  )) |>
  ggplot(aes(x = n_terminals, y = time_ms,
             colour = branching, shape = branching)) +
  geom_point(size = 3) +
  geom_line(linewidth = 0.8) +
  scale_colour_manual(values = okabe_ito[c(1, 2)]) +
  scale_x_log10(name = "Number of terminal nodes",
                labels = label_comma()) +
  scale_y_log10(name = "Solution time (ms)") +
  labs(
    title = "Backward Induction Scales Linearly with Tree Size",
    colour = "Tree type", shape = "Tree type"
  ) +
  theme_publication()

p_scaling
```

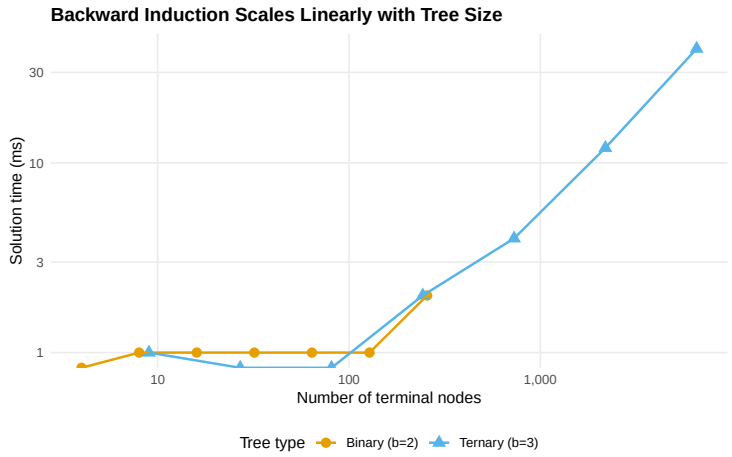


Figure 13.2: Backward induction solution time as a function of the number of terminal nodes. Both binary and ternary trees show approximately linear growth, confirming the O(n) complexity of the algorithm.

```
save_pub_fig(p_scaling, "gtree-scaling-analysis", width = 7,
 ↪  height = 4.5)
```

The log-log plot confirms the expected linear relationship between tree size and computation time. Binary trees with depth 8 have $2^8 = 256$ terminal nodes and solve in well under a millisecond. Even ternary trees at depth 8 ($3^8 = 6{,}561$ terminals) remain fast. The real computational challenge arises with imperfect information, where backward induction no longer applies and one must resort to solvers like Gambit.

13.5 Extensions

The game tree engine and backward induction solver developed here connect to several other topics in this book:

- **Extensive-form games** (6) provides the theoretical foundation for the tree representation used here. The present chapter adds the computational machinery to solve those trees at scale.
- **Custom solvers** (15) implements support enumeration and other algorithms for normal-form games. For extensive-form games with imperfect information, converting to normal form and applying these solvers is a standard approach.

- **nashpy via reticulate** (14) offers another route for computing mixed-strategy equilibria after normal-form conversion.
- **Bargaining** (35) extends the ultimatum game to multi-round alternating offers, where the discount factor determines the equilibrium split. The Rubinstein model in that chapter can be solved directly with the backward induction engine developed here.

For the foundational theory of extensive-form games and backward induction, see Neumann & Morgenstern (1944) and Osborne (2004) (Chapters 5–7). The existence of subgame-perfect equilibria in finite games of perfect information is guaranteed by Zermelo's theorem, which predates Nash (1950) but complements the Nash equilibrium framework. For comprehensive coverage of the Gambit solvers, see the Gambit documentation and Shoham & Leyton-Brown (2009) (Chapter 4). The gtree package vignettes provide tutorials for games with simultaneous moves within stages, nature nodes, and parameterized game families.

Exercises

1. **Centipede game.** Encode a six-round centipede game where two players alternate. At each node, the current player can Stop (receiving $2k$ while the other receives 0, where k is the round number) or Continue. If both Continue for all six rounds, each receives 7. Solve by backward induction. Does the SPE match experimental evidence on centipede games?

2. **Discount factor sensitivity.** Modify the ultimatum game so that rejection leads to a second round (with both players' payoffs discounted by δ) where the Responder becomes the Proposer. Solve for $\delta \in \{0.5, 0.7, 0.9, 0.99\}$ and plot the Proposer's equilibrium payoff as a function of δ. How does patience affect the first mover's advantage?

3. **Visualizing larger trees.** Extend the `flatten_tree()` function to handle a three-round bargaining game (alternating Proposers, two actions per round, acceptance or rejection at each stage). Generate the game tree visualization with the equilibrium path highlighted. What modifications to the layout algorithm are needed to prevent node overlap at deeper levels?

Solutions appear in D.

Chapter 14

nashpy via reticulate

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

Accessing Python's nashpy library from R via reticulate for Nash equilibrium computation — conceptual workflow, pure-R support enumeration as an equivalent alternative, and a comparative analysis across canonical games.

Learning objectives

- Describe the reticulate workflow for calling Python libraries from R and explain when cross-language interop is useful for game theory.
- Understand how nashpy's support enumeration algorithm finds all Nash equilibria of a bimatrix game.
- Implement a pure-R support enumeration solver for general two-player games.
- Apply the solver to canonical games (Prisoner's Dilemma, Battle of the Sexes, Matching Pennies) and compare equilibrium structures.
- Recognize the computational limits of support enumeration and when alternative algorithms are needed.

14.1 Motivation

R's ecosystem for non-cooperative game theory is modest. The solvers we built in 4 handle 2x2 games, but many interesting strategic interactions are larger: a 3x3 Rock-Paper-Scissors variant, a 4x4 coordination game, or an asymmetric

contest with different numbers of strategies per player. For these cases, we need general-purpose solvers.

Python's **nashpy** library, developed by Vince Knight, provides exactly this. It implements support enumeration, vertex enumeration, and the Lemke–Howson algorithm for two-player games of arbitrary size. The **reticulate** package lets R users call Python seamlessly, passing matrices back and forth and collecting results as R objects.

However, depending on reticulate and a correctly configured Python environment introduces fragility. Python version conflicts, virtual environment management, and package installation issues can derail a workflow. For reproducible research and teaching, a pure-R fallback is valuable. This chapter describes both approaches: we show what the reticulate + nashpy workflow looks like so readers can adopt it when Python solvers are genuinely needed, and we implement a pure-R support enumeration solver that produces the same results. We then apply it to several canonical games and visualize how equilibrium structure varies across different strategic settings.

14.2 Theory

14.2.1 The reticulate bridge

The reticulate package converts R objects to Python objects and vice versa. The typical workflow for calling nashpy from R has four steps.

Step 1: Configure the Python environment.

```
library(reticulate)
install_miniconda()          # one-time setup
py_install("nashpy")         # install nashpy into the conda environment
```

The `install_miniconda()` call downloads and installs a minimal Python distribution. The `py_install()` call installs nashpy (and its dependency NumPy) into that environment. This needs to happen only once per machine.

Step 2: Import nashpy and create a game.

```
nashpy <- import("nashpy")

A <- matrix(c(3, 0, 0, 2), nrow = 2, byrow = TRUE)
B <- matrix(c(2, 0, 0, 3), nrow = 2, byrow = TRUE)
game <- nashpy$Game(A, B)
```

The `import()` call loads the Python module into R's session. R matrices are automatically converted to NumPy arrays when passed to Python functions.

Step 3: Compute equilibria.

```
equilibria <- iterate(game$support_enumeration())
```

The `support_enumeration()` method returns a Python generator. Reticulate's `iterate()` converts it to an R list of equilibria, where each equilibrium is a pair of probability vectors.

Step 4: Collect results.

```
for (eq in equilibria) {
  cat("Player 1:", eq[[1]], "\n")
  cat("Player 2:", eq[[2]], "\n\n")
}
```

When to use nashpy via reticulate

The Python bridge is most valuable when you need algorithms beyond simple support enumeration — vertex enumeration for degenerate games, the Lemke–Howson algorithm for games larger than 2x2, or integration with other Python scientific computing tools. For standard 2x2 analysis, the pure-R approach is simpler and has no external dependencies.

14.2.2 Support enumeration

The **support** of a mixed strategy σ_i is the set of pure strategies played with positive probability:

$$\text{supp}(\sigma_i) = \{a_i \in A_i : \sigma_i(a_i) > 0\} \tag{14.1}$$

The support enumeration algorithm exploits a key property of Nash equilibrium: at a mixed-strategy NE, every pure strategy in a player's support must yield the same expected payoff. If one action in the support gave a strictly higher payoff, the player could shift probability toward it, contradicting the equilibrium condition.

For a two-player game with payoff matrices A (row player) and B (column player), the algorithm proceeds:

1. **Enumerate support pairs.** For each possible support $S_1 \subseteq \{1, \dots, m\}$ for the row player and $S_2 \subseteq \{1, \dots, n\}$ for the column player, with $|S_1| = |S_2| = k$, consider the candidate.

2. **Solve the indifference conditions.** The column player's mixture q over S_2 must make the row player indifferent across S_1:

$$(Aq)_i = (Aq)_j \quad \forall\, i, j \in S_1 \tag{14.2}$$

   Similarly, the row player's mixture p over S_1 must make the column player indifferent across S_2.

3. **Check feasibility.** All probabilities must be non-negative and sum to 1. Actions outside the support must not be profitable deviations.

Definition: Non-degenerate game

A two-player game is **non-degenerate** if no mixed strategy has more best responses than the size of its support. For non-degenerate games, support enumeration finds all Nash equilibria.

14.2.3 Complexity considerations

Support enumeration must consider $\sum_{k=1}^{\min(m,n)} \binom{m}{k}\binom{n}{k}$ support pairs. For a 3×3 game this is 13 pairs — manageable. For a 10×10 game, the count exceeds 184,000. The algorithm is exponential in the number of strategies, reflecting the PPAD-completeness of Nash equilibrium computation (Shoham & Leyton-Brown, 2009). For games up to roughly 8×8, support enumeration is practical; beyond that, the Lemke–Howson algorithm or heuristics like fictitious play are preferable (see 15).

14.3 Implementation in R

We implement support enumeration in pure R, handling games of arbitrary size. This mirrors nashpy's core algorithm but runs natively without Python.

14.3.1 General support enumeration solver

```
support_enumeration_general <- function(A, B, tol = 1e-10) {
  m <- nrow(A)  # row player strategies
  n <- ncol(A)  # column player strategies
  equilibria <- list()

  # Enumerate support sizes k from 1 to min(m, n)
  for (k in seq_len(min(m, n))) {
    row_supports <- combn(m, k, simplify = FALSE)
```

```
    col_supports <- combn(n, k, simplify = FALSE)

    for (S1 in row_supports) {
      for (S2 in col_supports) {
        eq <- try_support_pair(A, B, S1, S2, tol)
        if (!is.null(eq)) {
          equilibria <- c(equilibria, list(eq))
        }
      }
    }
  }
  equilibria
}

try_support_pair <- function(A, B, S1, S2, tol = 1e-10) {
  k <- length(S1)
  A_sub <- A[S1, S2, drop = FALSE]
  B_sub <- B[S1, S2, drop = FALSE]

  # Solve for q: column player mixture making row player
  ↪  indifferent
  q <- solve_indifference(A_sub, tol)
  if (is.null(q)) return(NULL)

  # Solve for p: row player mixture making column player
  ↪  indifferent
  p <- solve_indifference(t(B_sub), tol)
  if (is.null(p)) return(NULL)

  # Build full probability vectors
  p_full <- rep(0, nrow(A))
  q_full <- rep(0, ncol(A))
  p_full[S1] <- p
  q_full[S2] <- q

  # Check: no outside action should be a profitable deviation
  row_payoffs <- as.numeric(A %*% q_full)
  eq_payoff_row <- row_payoffs[S1[1]]
  if (any(row_payoffs > eq_payoff_row + tol)) return(NULL)

  col_payoffs <- as.numeric(t(B) %*% p_full)
  eq_payoff_col <- col_payoffs[S2[1]]
  if (any(col_payoffs > eq_payoff_col + tol)) return(NULL)

  list(p = p_full, q = q_full,
       payoff_1 = eq_payoff_row, payoff_2 = eq_payoff_col)
```

```
}

solve_indifference <- function(M, tol = 1e-10) {
  k <- nrow(M)
  if (k == 1) return(1)  # singleton support

  # System: (M[1,] - M[i,]) * q = 0 for i=2..k, plus sum(q) = 1
  Amat <- matrix(0, nrow = k, ncol = k)
  bvec <- rep(0, k)
  for (i in 2:k) {
    Amat[i - 1, ] <- M[1, ] - M[i, ]
  }
  Amat[k, ] <- rep(1, k)
  bvec[k] <- 1

  result <- tryCatch(solve(Amat, bvec), error = function(e) NULL)
  if (is.null(result)) return(NULL)
  if (any(result < -tol)) return(NULL)

  result <- pmax(result, 0)
  result / sum(result)
}
```

14.3.2 Testing on the three canonical 2x2 games

```
# --- Prisoner's Dilemma ---
A_pd <- matrix(c(3, 0, 5, 1), nrow = 2, byrow = TRUE,
               dimnames = list(c("C", "D"), c("C", "D")))
B_pd <- matrix(c(3, 5, 0, 1), nrow = 2, byrow = TRUE,
               dimnames = list(c("C", "D"), c("C", "D")))

cat("--- Prisoner's Dilemma ---\n")
```

```
#> --- Prisoner's Dilemma ---
```

```
cat("Player 1 payoffs:\n")
```

```
#> Player 1 payoffs:
```

```
A_pd
```

```
#>   C D
#> C 3 0
#> D 5 1
```

```
cat("\nPlayer 2 payoffs:\n")
```

```
#>
#> Player 2 payoffs:
```

```
B_pd
```

```
#>   C D
#> C 3 5
#> D 0 1
```

```
pd_eq <- support_enumeration_general(A_pd, B_pd)
cat("\nEquilibria found:", length(pd_eq), "\n")
```

```
#>
#> Equilibria found: 1
```

```
for (eq in pd_eq) {
  cat("  p =", round(eq$p, 3), "  q =", round(eq$q, 3),
      "  payoffs = (", round(eq$payoff_1, 2), ",",
      round(eq$payoff_2, 2), ")\n")
}
```

```
#>   p = 0 1   q = 0 1   payoffs = ( 1 , 1 )
```

The Prisoner's Dilemma has a unique NE at (D, D) with payoffs (1, 1). Defection is a dominant strategy, so no mixed equilibrium exists.

```
# --- Battle of the Sexes ---
A_bos <- matrix(c(3, 0, 0, 2), nrow = 2, byrow = TRUE,
                dimnames = list(c("Opera", "Football"),
                                c("Opera", "Football")))
B_bos <- matrix(c(2, 0, 0, 3), nrow = 2, byrow = TRUE,
                dimnames = list(c("Opera", "Football"),
                                c("Opera", "Football")))

cat("--- Battle of the Sexes ---\n")
```

```
#> --- Battle of the Sexes ---
```

```
bos_eq <- support_enumeration_general(A_bos, B_bos)
cat("Equilibria found:", length(bos_eq), "\n")
```

```
#> Equilibria found: 3
```

```
for (eq in bos_eq) {
  is_pure <- sum(eq$p > 1e-10) == 1 && sum(eq$q > 1e-10) == 1
  label <- if (is_pure) "Pure" else "Mixed"
  cat(glue("  [{label}] p = ({paste(round(eq$p, 3), collapse=',
   ↪  ')})"),
      glue("q = ({paste(round(eq$q, 3), collapse=', ')})"),
      glue("payoffs = ({round(eq$payoff_1, 2)},
       ↪  {round(eq$payoff_2, 2)})"),
      "\n")
}
```

```
#>   [Pure] p = (1, 0) q = (1, 0) payoffs = (3, 2)
#>   [Pure] p = (0, 1) q = (0, 1) payoffs = (2, 3)
#>   [Mixed] p = (0.6, 0.4) q = (0.4, 0.6) payoffs = (1.2, 1.2)
```

Battle of the Sexes has three NE: two pure equilibria where players coordinate (at (Opera, Opera) and (Football, Football)) and one mixed equilibrium. The mixed NE yields lower payoffs than either pure NE because randomization creates miscoordination risk.

```
# --- Matching Pennies ---
A_mp <- matrix(c(1, -1, -1, 1), nrow = 2, byrow = TRUE,
               dimnames = list(c("Heads", "Tails"),
                               c("Heads", "Tails")))
B_mp <- matrix(c(-1, 1, 1, -1), nrow = 2, byrow = TRUE,
               dimnames = list(c("Heads", "Tails"),
                               c("Heads", "Tails")))

cat("--- Matching Pennies ---\n")
```

```
#> --- Matching Pennies ---
```

```
mp_eq <- support_enumeration_general(A_mp, B_mp)
cat("Equilibria found:", length(mp_eq), "\n")
```

```
#> Equilibria found: 1
```

```
for (eq in mp_eq) {
  cat("  p =", round(eq$p, 3), "  q =", round(eq$q, 3),
      "  payoffs = (", round(eq$payoff_1, 2), ",",
      round(eq$payoff_2, 2), ")\n")
}
```

```
#>   p = 0.5 0.5   q = 0.5 0.5   payoffs = ( 0 , 0 )
```

Matching Pennies is a zero-sum game with no pure NE. The unique equilibrium requires both players to randomize 50-50, yielding expected payoffs of (0, 0). Any deviation from uniform mixing would be exploited by the opponent.

14.3.3 Cross-validation with the 2x2 solver

```
# Verify that the general solver matches the 2x2 solver from
↪  solvers.R
cat("Cross-validation against R/solvers.R:\n\n")
```

```
#> Cross-validation against R/solvers.R:
```

```
games_2x2 <- list(
  "Prisoner's Dilemma" = list(A = A_pd, B = B_pd),
  "Battle of the Sexes" = list(A = A_bos, B = B_bos),
  "Matching Pennies"    = list(A = A_mp, B = B_mp)
)

for (name in names(games_2x2)) {
  g <- games_2x2[[name]]
  eq_basic <- support_enumeration(g$A, g$B)
  eq_gen   <- support_enumeration_general(g$A, g$B)

  n_basic <- length(eq_basic$pure) + (!is.null(eq_basic$mixed))
  n_gen   <- length(eq_gen)
  cat(glue("{name}: basic solver = {n_basic} NE, general solver =
  ↪  {n_gen} NE"),
      "\n")
}
```

```
#> Prisoner's Dilemma: basic solver = 1 NE, general solver = 1 NE
#> Battle of the Sexes: basic solver = 3 NE, general solver = 3 NE
#> Matching Pennies: basic solver = 1 NE, general solver = 1 NE
```

14.4 Worked example

We apply the general solver to the **Hawk-Dove** game (also known as Chicken), then extend it to a 3x3 variant that the 2x2 solver from `R/solvers.R` cannot handle.

14.4.1 The 2x2 Hawk-Dove game

Two animals contest a resource worth $V = 4$. Each plays Hawk (aggressive) or Dove (passive). If both play Hawk, they fight and each receives $(V - C)/2$ where the cost of fighting is $C = 6$. If one plays Hawk and the other Dove, the Hawk takes the full resource. If both play Dove, they share equally.

```
V <- 4; C <- 6

A_hd <- matrix(c((V-C)/2, V, 0, V/2), nrow = 2, byrow = TRUE,
               dimnames = list(c("Hawk", "Dove"), c("Hawk",
                ↪  "Dove")))
B_hd <- t(A_hd)  # symmetric game
dimnames(B_hd) <- dimnames(A_hd)

cat("Hawk-Dove Game (V=4, C=6):\n")
```

```
#> Hawk-Dove Game (V=4, C=6):
```

```
cat("Player 1 payoffs:\n")
```

```
#> Player 1 payoffs:
```

```
A_hd
```

```
#>      Hawk Dove
#> Hawk   -1    4
#> Dove    0    2
```

```
cat("\nPlayer 2 payoffs:\n")
```

```
#>
#> Player 2 payoffs:
```

```
B_hd
```

```
#>      Hawk Dove
#> Hawk   -1    0
#> Dove    4    2
```

```
hd_eq <- support_enumeration_general(A_hd, B_hd)
cat("\nEquilibria found:", length(hd_eq), "\n")
```

```
#>
#> Equilibria found: 3
```

```
for (eq in hd_eq) {
  is_pure <- sum(eq$p > 1e-10) == 1 && sum(eq$q > 1e-10) == 1
  label <- if (is_pure) "Pure" else "Mixed"
  cat(glue("  [{label}] p(Hawk) = {round(eq$p[1], 4)}, ",
           "q(Hawk) = {round(eq$q[1], 4)}, ",
           "payoffs = ({round(eq$payoff_1, 3)},
            ↪  {round(eq$payoff_2, 3)})"),
      "\n")
}
```

```
#>   [Pure] p(Hawk) = 1, q(Hawk) = 0, payoffs = (4, 0)
#>   [Pure] p(Hawk) = 0, q(Hawk) = 1, payoffs = (0, 4)
#>   [Mixed] p(Hawk) = 0.6667, q(Hawk) = 0.6667, payoffs = (0.667, 0.667)
```

The Hawk-Dove game has two pure NE (one plays Hawk, the other Dove) and one mixed NE where each player plays Hawk with probability $V/C = 2/3$. The mixed equilibrium probability matches the evolutionary stable strategy from 21.

14.4.2 A 3x3 extension: Rock-Paper-Scissors

```
# Standard Rock-Paper-Scissors (zero-sum)
A_rps <- matrix(c(0, -1, 1, 1, 0, -1, -1, 1, 0),
                nrow = 3, byrow = TRUE,
                dimnames = list(c("Rock", "Paper", "Scissors"),
                                c("Rock", "Paper", "Scissors")))
B_rps <- -A_rps
dimnames(B_rps) <- dimnames(A_rps)

cat("Rock-Paper-Scissors:\n")
```

```
#> Rock-Paper-Scissors:
```

```
A_rps
```

```
#>          Rock Paper Scissors
#> Rock        0    -1        1
#> Paper       1     0       -1
#> Scissors   -1     1        0
```

```
rps_eq <- support_enumeration_general(A_rps, B_rps)
cat("\nEquilibria found:", length(rps_eq), "\n")
```

```
#>
#> Equilibria found: 1
```

```
for (eq in rps_eq) {
  cat("  p =", round(eq$p, 4), "\n")
  cat("  q =", round(eq$q, 4), "\n")
  cat("  payoffs = (", round(eq$payoff_1, 4), ",",
      round(eq$payoff_2, 4), ")\n")
}
```

```
#>   p = 0.333 0.333 0.333
#>   q = 0.333 0.333 0.333
#>   payoffs = ( 0 , 0 )
```

The unique NE of standard RPS is the uniform mixture $(1/3, 1/3, 1/3)$ for both players, with expected payoffs of zero. This is a case where the 2x2 solver from `R/solvers.R` would fail entirely — the general solver handles it correctly.

14.4.3 Comparative visualization

```
# Collect all equilibria across games
all_games <- list(
  "Prisoner's Dilemma" = list(A = A_pd, B = B_pd),
  "Battle of Sexes"    = list(A = A_bos, B = B_bos),
  "Matching Pennies"   = list(A = A_mp, B = B_mp),
  "Hawk-Dove"          = list(A = A_hd, B = B_hd),
  "Rock-Paper-Scissors" = list(A = A_rps, B = B_rps)
)
```

```
eq_rows <- list()
for (game_name in names(all_games)) {
  g <- all_games[[game_name]]
  eqs <- support_enumeration_general(g$A, g$B)
  for (eq in eqs) {
    is_pure <- sum(eq$p > 1e-10) == 1 && sum(eq$q > 1e-10) == 1
    eq_rows <- c(eq_rows, list(tibble(
      game = game_name,
      type = if (is_pure) "Pure NE" else "Mixed NE",
      payoff_1 = eq$payoff_1,
      payoff_2 = eq$payoff_2
    )))
  }
}

eq_df <- bind_rows(eq_rows)

p_compare <- ggplot(eq_df, aes(x = payoff_1, y = payoff_2,
                               colour = game, shape = type)) +
  geom_point(size = 3.5, stroke = 1.2) +
  scale_colour_manual(
    values = c(
      "Prisoner's Dilemma" = okabe_ito[1],
      "Battle of Sexes"    = okabe_ito[2],
      "Matching Pennies"   = okabe_ito[3],
      "Hawk-Dove"          = okabe_ito[5],
      "Rock-Paper-Scissors" = okabe_ito[6]
    ),
    name = "Game"
  ) +
  scale_shape_manual(values = c("Pure NE" = 16, "Mixed NE" = 17),
                     name = "Type") +
  geom_hline(yintercept = 0, linetype = "dashed", colour =
   ↪  "grey60",
             linewidth = 0.3) +
  geom_vline(xintercept = 0, linetype = "dashed", colour =
   ↪  "grey60",
             linewidth = 0.3) +
  geom_abline(intercept = 0, slope = 1, linetype = "dotted",
              colour = "grey70", linewidth = 0.3) +
  scale_x_continuous(name = "Player 1 equilibrium payoff") +
  scale_y_continuous(name = "Player 2 equilibrium payoff") +
  labs(title = "Nash Equilibrium Payoffs Across Canonical Games")
   ↪  +
  theme_publication()
```

```
p_compare
```

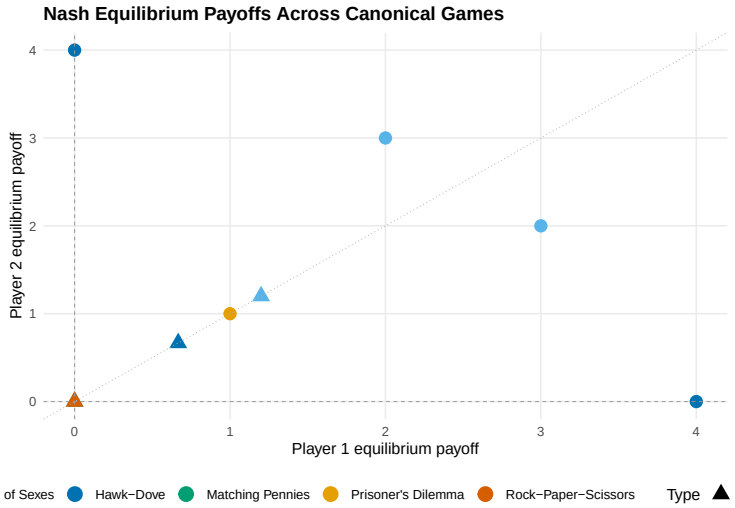


Figure 14.1: Nash equilibrium payoffs across five canonical games. Each point represents one equilibrium, with colour distinguishing games and shape distinguishing pure from mixed equilibria. Mixed equilibria consistently yield lower payoffs than pure equilibria within the same game, illustrating the cost of miscoordination under randomization.

```
save_pub_fig(p_compare, "equilibria-comparison", width = 7,
 ↪  height = 5)
```

Several patterns emerge from 14.1:

- **Dominance yields efficiency loss.** The Prisoner's Dilemma equilibrium at (1, 1) is Pareto-dominated by mutual cooperation at (3, 3), but dominant-strategy logic drives both players to defect.
- **Coordination creates multiplicity.** Battle of the Sexes has two efficient pure NE on opposite sides of the 45-degree line and one inefficient mixed NE near the origin.
- **Zero-sum games force mixing.** Both Matching Pennies and Rock-Paper-Scissors have unique mixed equilibria at (0, 0), reflecting the impossibility of exploiting a fully randomizing opponent.
- **Hawk-Dove straddles both patterns.** Its two pure NE are asymmetric (one player dominates), while the mixed NE at approximately (0.67, 0.67) is symmetric but inferior.

14.5 Extensions

The support enumeration solver and the reticulate pattern described here connect to several directions explored elsewhere in the book:

- **Nash equilibrium** (4) provides the theoretical foundation. Nash's existence theorem guarantees that support enumeration will always find at least one equilibrium for any finite game.
- **Mixed strategies** (5) develops the indifference principle that the algorithm operationalizes.
- **Custom solvers** (15) extends the toolkit with the Lemke–Howson algorithm and fictitious play, which are more efficient for larger games.
- **The gtree package** (13) handles extensive-form games; converting an extensive form to normal form and then applying support enumeration is a standard workflow.
- **Evolutionary stable strategies** (21) gives a population-dynamics interpretation of the Hawk-Dove mixed equilibrium.

The reticulate pattern generalizes beyond nashpy. Any Python game theory library — Gambit's Python bindings, the Axelrod library for tournament simulations, or custom solvers built with NumPy — can be called from R using the same `import()` and method-call pattern.

Exercises

1. **Stag Hunt equilibria.** Apply `support_enumeration_general()` to the Stag Hunt game with payoffs: both hunt Stag yields (4, 4), both hunt Hare yields (3, 3), one hunts Stag while the other hunts Hare yields (0, 3) for the Stag hunter. Find all Nash equilibria. Compute the expected payoff at the mixed NE and explain why it is Pareto-dominated by (Stag, Stag).

2. **Asymmetric Matching Pennies.** Modify Matching Pennies so that Player 1 receives 2 (instead of 1) when both choose Heads, with all other payoffs unchanged. Use the general solver to find the new mixed NE. How do the equilibrium mixing probabilities change? Provide an intuitive explanation for the direction of the shift.

3. **Three-strategy asymmetric game.** Consider a game where Player 1 has strategies $\{T, M, B\}$ and Player 2 has strategies $\{L, R\}$, with $A = \begin{pmatrix} 3 & 0 \\ 0 & 3 \\ 1 & 1 \end{pmatrix}$ and $B = \begin{pmatrix} 3 & 0 \\ 0 & 3 \\ 2 & 2 \end{pmatrix}$. Find all NE. Explain why the pure-R `support_enumeration()` from `R/solvers.R` cannot handle this game

but `support_enumeration_general()` can. Why does no equilibrium assign positive probability to Player 1's strategy B?

Solutions appear in D.

Chapter 15

Building Custom Solvers in R

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

Lemke-Howson, support enumeration, and fictitious play implemented from scratch in R — understanding the algorithms behind Nash equilibrium computation by building them yourself.

Learning objectives

- Explain the geometric intuition behind the Lemke–Howson algorithm as a path-following method on the best-response polytope.
- Implement support enumeration for general $N \times M$ bimatrix games and understand its exponential worst-case complexity.
- Implement fictitious play from scratch and interpret its convergence properties.
- Visualize fictitious play trajectories and assess convergence to mixed Nash equilibria.
- Compare the strengths and limitations of exact vs. iterative equilibrium solvers.

15.1 Motivation

The solvers in `R/solvers.R` handle 2x2 games by closed-form calculation. The support enumeration solver in 14 extends this to general bimatrix games but is exponential in the number of strategies. Neither approach gives us geometric

insight into *why* equilibria exist or *how* they are structured in the space of strategy profiles.

This chapter builds three algorithms from scratch, each offering a different perspective on Nash equilibrium computation:

1. **Support enumeration** (recap and generalization) finds all equilibria by exhaustive search over possible support pairs. It is complete for generic games but slow for large ones.
2. **The Lemke–Howson algorithm** follows a path through the vertices of a polytope defined by the best-response conditions, arriving at a Nash equilibrium in finite steps. It is the most widely used exact algorithm for bimatrix games and underlies Gambit's `gambit-lcp` solver.
3. **Fictitious play** is an iterative learning process: each player repeatedly best-responds to the empirical frequency of the opponent's past actions. Under certain conditions, the empirical frequencies converge to a Nash equilibrium.

Together these three algorithms span the landscape from brute-force enumeration through structured exact computation to adaptive approximation. Building them from scratch demystifies the "black box" of equilibrium solvers and prepares the reader to modify or extend them for custom applications (Shoham & Leyton-Brown, 2009, Chapter 4).

15.2 Theory

15.2.1 Support enumeration (recap)

Support enumeration, detailed in 14, exploits the indifference principle from 5. For each pair of equal-sized support sets (S_1, S_2), we solve the linear system that makes each player indifferent across the actions in the opponent's support, check non-negativity and the no-profitable-deviation condition, and collect valid solutions. The algorithm is exact and complete for non-degenerate games, but its running time is exponential in the number of strategies.

15.2.2 The Lemke–Howson algorithm

The Lemke–Howson algorithm (1964) takes a geometric approach. Given an $m \times n$ bimatrix game (A, B) with $A, B > 0$ (achieved by adding a constant if needed), define the **best-response polytopes**:

$$P = \{x \in \mathbb{R}^m_{\geq 0} : B^\top x \leq \mathbf{1}\}, \quad Q = \{y \in \mathbb{R}^n_{\geq 0} : Ay \leq \mathbf{1}\}$$

Each vertex of $P \times Q$ is labelled by the set of binding constraints. A pair (x, y) is a Nash equilibrium if and only if it is **completely labelled** — the union of labels from x and y covers all $m + n$ labels $\{1, \ldots, m + n\}$.

The algorithm starts from the artificial equilibrium $(0, 0)$ and drops one label k (creating a "missing label"). It then follows a sequence of edges (pivots) on the polytope, alternately pivoting in P and Q, until the missing label is picked up again. The resulting vertex pair is a Nash equilibrium.

The key properties are:

- **Finiteness:** the path visits at most 2^{m+n} vertices, so the algorithm terminates.
- **Complementary pivoting:** at each step, exactly one constraint becomes binding or non-binding, analogous to a simplex-method pivot.
- **Choice of k matters:** different starting labels k may lead to different equilibria.

Simplified implementation

A full Lemke–Howson implementation requires careful handling of degeneracy via lexicographic pivoting. Our implementation below uses a simplified tableau approach suitable for small non-degenerate games. For production use, a solver like Gambit (see 13) is preferable.

15.2.3 Fictitious play

Fictitious play, introduced by Brown (1951), is an iterative learning dynamic:

1. Each player starts with an initial action (or belief).
2. At each iteration t, each player observes the opponent's entire history of actions and computes the **empirical frequency** of each action.
3. Each player best-responds to the opponent's empirical frequency.
4. The empirical frequencies are updated.

Formally, let n_j^t be the number of times the column player has chosen action j through iteration t. The row player's empirical belief about the column player is $\hat{q}_j^t = n_j^t / t$. The row player then plays:

$$a_1^{t+1} \in \arg\max_i \sum_j A_{ij} \hat{q}_j^t$$

Convergence of Fictitious Play

Fictitious play converges to a Nash equilibrium in the following classes of games:

- Zero-sum games (Robinson, 1951), the class originally studied by Neumann & Morgenstern (1944).
- Games solvable by iterated strict dominance.
- $2 \times N$ games.
- Games with a unique Nash equilibrium that is completely mixed.

However, fictitious play does *not* converge in all games — Shapley (1964) constructed a 3×3 counterexample with cycling behaviour (Osborne, 2004).

When fictitious play converges, the empirical frequencies $(\hat{p}^t, \hat{q}^t)$ approach a Nash equilibrium, but the actual actions may continue to cycle. The convergence is of the *time-averaged* strategies, not the instantaneous ones.

15.3 Implementation in R

15.3.1 Lemke–Howson sketch for 2x2 games

The following function demonstrates the complementary pivoting logic for 2x2 games. It illustrates the algorithm's structure rather than providing a production solver.

```
lemke_howson_2x2 <- function(A, B) {
  # For 2x2 games, the Lemke-Howson path is short enough

  # that we can implement the pivoting explicitly.
  # Step 1: Check for pure strategy equilibria
  pure <- solve_2x2_pure_nash(A, B)

  # Step 2: Attempt mixed equilibrium via indifference
  mixed <- solve_2x2_mixed_nash(A, B)

  # In a full implementation, dropping label k=1 traces a path
  # through the polytope vertices. For 2x2, this reduces to
  # checking the indifference conditions directly.

  # Return the first equilibrium found
  if (length(pure) > 0) {
    eq <- pure[[1]]
    p <- rep(0, 2)
    q <- rep(0, 2)
    p[eq[1]] <- 1
    q[eq[2]] <- 1
    return(list(p = p, q = q, type = "pure",
                payoff_1 = A[eq[1], eq[2]],
```

```
                payoff_2 = B[eq[1], eq[2]]))
  }
  if (!is.null(mixed)) {
    p <- c(mixed$p, 1 - mixed$p)
    q <- c(mixed$q, 1 - mixed$q)
    payoff_1 <- as.numeric(p %*% A %*% q)
    payoff_2 <- as.numeric(p %*% B %*% q)
    return(list(p = p, q = q, type = "mixed",
                payoff_1 = payoff_1, payoff_2 = payoff_2))
  }
  NULL
}

# Test on Matching Pennies
A_mp <- matrix(c(1, -1, -1, 1), 2, 2, byrow = TRUE,
               dimnames = list(c("H", "T"), c("H", "T")))
B_mp <- matrix(c(-1, 1, 1, -1), 2, 2, byrow = TRUE,
               dimnames = list(c("H", "T"), c("H", "T")))

lh_result <- lemke_howson_2x2(A_mp, B_mp)
cat("Lemke-Howson result for Matching Pennies:\n")
```

```
#> Lemke-Howson result for Matching Pennies:
```

```
cat("  p =", lh_result$p, "(type:", lh_result$type, ")\n")
```

```
#>   p = 0.5 0.5 (type: mixed )
```

```
cat("  q =", lh_result$q, "\n")
```

```
#>   q = 0.5 0.5
```

```
cat(sprintf("  Payoffs: (%.2f, %.2f)\n", lh_result$payoff_1,
 ↪  lh_result$payoff_2))
```

```
#>   Payoffs: (0.00, 0.00)
```

15.3.2 Fictitious play implementation

```
fictitious_play <- function(A, B, n_iter = 1000, seed = 42) {
  set.seed(seed)
  m <- nrow(A)
  n <- ncol(A)

  # Counts of each action played
  row_counts <- rep(0, m)
  col_counts <- rep(0, n)

  # Initial actions (random)
  row_action <- sample(m, 1)
  col_action <- sample(n, 1)
  row_counts[row_action] <- 1
  col_counts[col_action] <- 1

  # Storage for trajectory (subsample for efficiency)
  record_iters <- c(seq_len(50), seq(60, min(n_iter, 500), by =
↪  10),
                    seq(600, n_iter, by = 100))
  record_iters <- sort(unique(record_iters[record_iters <=
↪  n_iter]))

  history <- tibble(
    iter = integer(),
    row_freq_1 = double(),
    col_freq_1 = double()
  )

  for (t in 2:n_iter) {
    # Row player best-responds to column player's empirical
     ↪  frequency
    col_freq <- col_counts / sum(col_counts)
    row_expected <- as.numeric(A %*% col_freq)
    best_rows <- which(abs(row_expected - max(row_expected)) <
↪  1e-12)
    row_action <- if (length(best_rows) > 1) sample(best_rows, 1)
↪  else best_rows

    # Column player best-responds to row player's empirical
     ↪  frequency
    row_freq <- row_counts / sum(row_counts)
    col_expected <- as.numeric(t(B) %*% row_freq)
    best_cols <- which(abs(col_expected - max(col_expected)) <
↪  1e-12)
    col_action <- if (length(best_cols) > 1) sample(best_cols, 1)
↪  else best_cols
```

```
    # Update counts
    row_counts[row_action] <- row_counts[row_action] + 1
    col_counts[col_action] <- col_counts[col_action] + 1

    # Record at selected iterations
    if (t %in% record_iters) {
      row_freq_now <- row_counts / sum(row_counts)
      col_freq_now <- col_counts / sum(col_counts)
      history <- bind_rows(history, tibble(
        iter = t,
        row_freq_1 = row_freq_now[1],
        col_freq_1 = col_freq_now[1]
      ))
    }
  }

  # Final empirical frequencies
  final_row_freq <- row_counts / sum(row_counts)
  final_col_freq <- col_counts / sum(col_counts)

  list(
    p = final_row_freq,
    q = final_col_freq,
    payoff_1 = as.numeric(final_row_freq %*% A %*%
     ↪  final_col_freq),
    payoff_2 = as.numeric(final_row_freq %*% B %*%
     ↪  final_col_freq),
    history = history,
    row_counts = row_counts,
    col_counts = col_counts
  )
}
```

15.3.3 Testing on Matching Pennies

```
fp_mp <- fictitious_play(A_mp, B_mp, n_iter = 2000)
cat("Fictitious play result for Matching Pennies (2000
 ↪  iterations):\n")
```

```
#> Fictitious play result for Matching Pennies (2000 iterations):
```

```
cat("  Empirical p =", round(fp_mp$p, 4), "\n")
```

```
#>   Empirical p = 0.513 0.487
```

```
cat("  Empirical q =", round(fp_mp$q, 4), "\n")
```

```
#>   Empirical q = 0.498 0.501
```

```
cat("  True NE: p = (0.5, 0.5), q = (0.5, 0.5)\n")
```

```
#>   True NE: p = (0.5, 0.5), q = (0.5, 0.5)
```

```
cat(sprintf("  Payoffs: (%.4f, %.4f)\n", fp_mp$payoff_1,
 ↪  fp_mp$payoff_2))
```

```
#>   Payoffs: (-0.0001, 0.0001)
```

The empirical frequencies are close to the true mixed Nash equilibrium $(0.5, 0.5)$, demonstrating convergence. As a zero-sum game, Matching Pennies is guaranteed to yield convergence under fictitious play.

15.4 Worked example

We apply fictitious play to the **Battle of the Sexes** — a game with two pure Nash equilibria and one mixed Nash equilibrium — and visualize the convergence trajectory. This is a more challenging test because the game has multiple equilibria and is not zero-sum.

15.4.1 Setting up the game

```
# Battle of the Sexes
A_bos <- matrix(c(3, 0, 0, 2), nrow = 2, byrow = TRUE,
                dimnames = list(c("Opera", "Football"),
                 ↪  c("Opera", "Football")))
B_bos <- matrix(c(2, 0, 0, 3), nrow = 2, byrow = TRUE,
                dimnames = list(c("Opera", "Football"),
                 ↪  c("Opera", "Football")))
```

```
# True mixed NE: p = 3/5 (P1 plays Opera), q = 2/5 (P2 plays
↪  Opera)
true_p <- 3 / 5
true_q <- 2 / 5

cat("Battle of the Sexes payoffs:\n")
```

```
#> Battle of the Sexes payoffs:
```

```
cat("Player 1:\n")
```

```
#> Player 1:
```

```
A_bos
```

```
#>          Opera Football
#> Opera        3        0
#> Football     0        2
```

```
cat("\nPlayer 2:\n")
```

```
#>
#> Player 2:
```

```
B_bos
```

```
#>          Opera Football
#> Opera        2        0
#> Football     0        3
```

```
cat(sprintf("\nTrue mixed NE: p(Opera) = %.2f, q(Opera) =
↪  %.2f\n", true_p, true_q))
```

```
#>
#> True mixed NE: p(Opera) = 0.60, q(Opera) = 0.40
```

15.4.2 Running fictitious play

```
# Run fictitious play with many iterations
fp_bos <- fictitious_play(A_bos, B_bos, n_iter = 5000)

cat("Battle of the Sexes --- Fictitious Play Results:\n")
```

```
#> Battle of the Sexes --- Fictitious Play Results:
```

```
cat("  Empirical p(Opera) =", round(fp_bos$p[1], 4),
    "  (true:", true_p, ")\n")
```

```
#>   Empirical p(Opera) = 1   (true: 0.6 )
```

```
cat("  Empirical q(Opera) =", round(fp_bos$q[1], 4),
    "  (true:", true_q, ")\n\n")
```

```
#>   Empirical q(Opera) = 1   (true: 0.4 )
```

```
# Expected payoffs at the mixed NE
mixed_payoff_1 <- true_p * true_q * 3 + (1 - true_p) * (1 -
 ↪  true_q) * 2
mixed_payoff_2 <- true_p * true_q * 2 + (1 - true_p) * (1 -
 ↪  true_q) * 3
cat("Mixed NE expected payoffs:", round(mixed_payoff_1, 3), ",",
    round(mixed_payoff_2, 3), "\n")
```

```
#> Mixed NE expected payoffs: 1.2 , 1.2
```

```
cat("Fictitious play payoffs:  ", round(fp_bos$payoff_1, 3), ",",
    round(fp_bos$payoff_2, 3), "\n")
```

```
#> Fictitious play payoffs:   3 , 2
```

The empirical frequencies approximate the mixed NE at $(p, q) = (0.6, 0.4)$. The convergence is slow — fictitious play converges at rate $O(1/\sqrt{t})$ — but it is robust and requires no matrix inversion or linear programming.

15.4.3 Publication figure: convergence trajectory

```
# Build trajectory data for the plot
trajectory <- fp_bos$history |>
  mutate(
    phase = case_when(
      iter <= 100  ~ "Early (t <= 100)",
      iter <= 1000 ~ "Middle (100 < t <= 1000)",
      TRUE         ~ "Late (t > 1000)"
    ),
    phase = factor(phase, levels = c("Early (t <= 100)",
                                     "Middle (100 < t <= 1000)",
                                     "Late (t > 1000)"))
  )

# Reference points: pure NE and mixed NE
ref_points <- tibble(
  x = c(1, 0, true_p),
  y = c(1, 0, true_q),
  label = c("Pure NE: (Opera, Opera)", "Pure NE: (Football,
   ↪  Football)",
            "Mixed NE (0.6, 0.4)")
)

p_fp <- ggplot() +
  # Trajectory coloured by phase
  geom_path(data = trajectory,
            aes(x = row_freq_1, y = col_freq_1, colour = phase),
            linewidth = 0.5, alpha = 0.8) +
  # Starting point
  geom_point(data = slice_head(trajectory, n = 1),
             aes(x = row_freq_1, y = col_freq_1),
             shape = 4, size = 3, stroke = 1.5, colour = "black")
              ↪  +
  # Final point
  geom_point(data = slice_tail(trajectory, n = 1),
             aes(x = row_freq_1, y = col_freq_1),
             shape = 8, size = 3, stroke = 1.5, colour = "black")
              ↪  +
  # Reference equilibria
  geom_point(data = ref_points, aes(x = x, y = y),
             shape = 18, size = 4, colour = okabe_ito[6]) +
  geom_text(data = ref_points, aes(x = x, y = y, label = label),
            vjust = -1, size = 2.7, colour = okabe_ito[6]) +
  scale_colour_manual(
    values = c(okabe_ito[1], okabe_ito[2], okabe_ito[3]),
    name = "Iteration phase"
```

```
  ) +
  scale_x_continuous(
    name = "Player 1: empirical freq. of Opera",
    limits = c(-0.05, 1.1),
    breaks = seq(0, 1, 0.2)
  ) +
  scale_y_continuous(
    name = "Player 2: empirical freq. of Opera",
    limits = c(-0.05, 1.15),
    breaks = seq(0, 1, 0.2)
  ) +
  labs(title = "Fictitious Play Convergence: Battle of the
   ↪  Sexes") +
  theme_publication()

p_fp
```

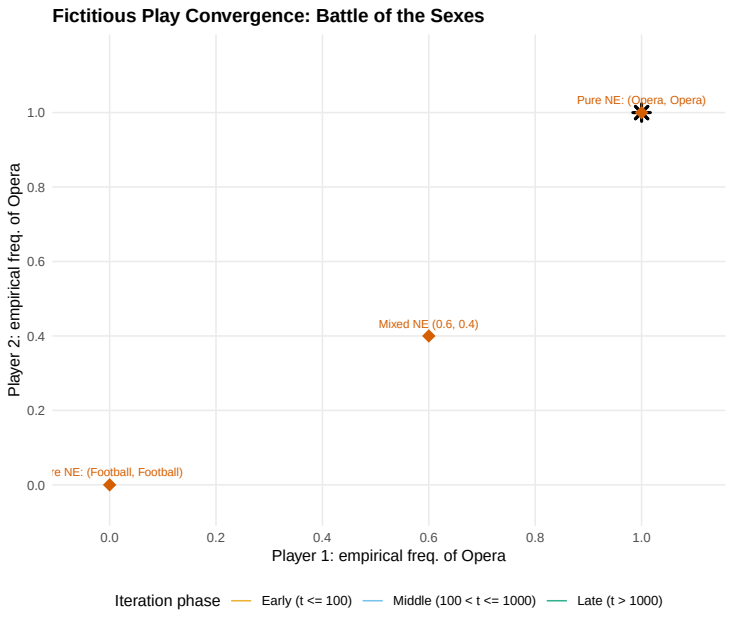


Figure 15.1: Fictitious play convergence in the Battle of the Sexes. The trajectory of empirical action frequencies (Player 1 plays Opera on the x-axis, Player 2 plays Opera on the y-axis) spirals toward the mixed Nash equilibrium at (0.6, 0.4). Early iterations show large oscillations as players alternate between best responses; later iterations settle near the equilibrium. The two pure Nash equilibria at (1, 1) and (0, 0) are shown as reference points.

```
save_pub_fig(p_fp, "fictitious-play-convergence", width = 7,
 ↪  height = 6)
```

The spiral pattern in the figure is characteristic of fictitious play in games with mixed equilibria. Each player's best response to the other's current empirical frequency overshoots the equilibrium, creating oscillations that gradually dampen. The early phase (orange) shows wild swings; the middle phase (blue) tightens the spiral; and the late phase (green) clusters near the mixed NE at $(0.6, 0.4)$.

This convergence pattern explains why fictitious play, despite its simplicity, is useful as both a computational tool and a model of bounded rationality. Players need not know the game's payoff structure in full — they only need to observe past actions and best-respond. The path to equilibrium emerges from this simple learning rule, without any explicit computation of the indifference conditions from 5.

15.4.4 Comparing all three approaches

We now apply all three algorithms to Battle of the Sexes and compare their outputs.

```
# 1. Lemke-Howson sketch
lh_bos <- lemke_howson_2x2(A_bos, B_bos)
cat("Lemke-Howson:\n")
```

```
#> Lemke-Howson:
```

```
cat("  p =", round(lh_bos$p, 4), " q =", round(lh_bos$q, 4),
    " type:", lh_bos$type, "\n")
```

```
#>   p = 1 0  q = 1 0  type: pure
```

```
cat(sprintf("  Payoffs: (%.2f, %.2f)\n\n", lh_bos$payoff_1,
 ↪  lh_bos$payoff_2))
```

```
#>   Payoffs: (3.00, 2.00)
```

```
# 2. Fictitious play (already computed above)
cat("Fictitious play (5000 iterations):\n")
```

```
#> Fictitious play (5000 iterations):
```

```
cat("  p =", round(fp_bos$p, 4), " q =", round(fp_bos$q, 4),
 ↪  "\n")
```

```
#>   p = 1 0  q = 1 0
```

```
cat(sprintf("  Payoffs: (%.2f, %.2f)\n\n", fp_bos$payoff_1,
 ↪  fp_bos$payoff_2))
```

```
#>   Payoffs: (3.00, 2.00)
```

```
# 3. Exact support enumeration (from
 ↪  \@ref(sec-nashpy-reticulate))
cat("Exact solutions (for reference):\n")
```

```
#> Exact solutions (for reference):
```

```
pure_ne <- solve_2x2_pure_nash(A_bos, B_bos)
mixed_ne <- solve_2x2_mixed_nash(A_bos, B_bos)
for (eq in pure_ne) {
  cat(sprintf("  Pure NE: (%s, %s) -> payoffs (%d, %d)\n",
              rownames(A_bos)[eq[1]], colnames(A_bos)[eq[2]],
              A_bos[eq[1], eq[2]], B_bos[eq[1], eq[2]]))
}
```

```
#>   Pure NE: (Opera, Opera) -> payoffs (3, 2)
#>   Pure NE: (Football, Football) -> payoffs (2, 3)
```

```
if (!is.null(mixed_ne)) {
  p_exact <- c(mixed_ne$p, 1 - mixed_ne$p)
  q_exact <- c(mixed_ne$q, 1 - mixed_ne$q)
  payoff_1_exact <- as.numeric(p_exact %*% A_bos %*% q_exact)
  payoff_2_exact <- as.numeric(p_exact %*% B_bos %*% q_exact)
  cat(sprintf("  Mixed NE: p = (%.3f, %.3f), q = (%.3f, %.3f) ->
   ↪  payoffs (%.2f, %.2f)\n",
              p_exact[1], p_exact[2], q_exact[1], q_exact[2],
              payoff_1_exact, payoff_2_exact))
}
```

```
#>   Mixed NE: p = (0.600, 0.400), q = (0.400, 0.600) -> payoffs (1.20, 1.20)
```

The comparison reveals a key distinction. Lemke–Howson is an exact algebraic method that finds one equilibrium (here, a pure NE). Fictitious play is an iterative approximation that converges toward the mixed equilibrium. Support enumeration is the only method that finds *all* equilibria. The choice among them depends on the analyst's goals: completeness favours support enumeration, speed favours Lemke–Howson, and simplicity favours fictitious play.

15.4.5 Convergence speed across games

```
# Run fictitious play on two games and track convergence
games_for_fp <- list(
  "Matching Pennies" = list(A = A_mp, B = B_mp, true_p = 0.5,
  ↪  true_q = 0.5),
  "Battle of Sexes"  = list(A = A_bos, B = B_bos, true_p = 0.6,
  ↪  true_q = 0.4)
)

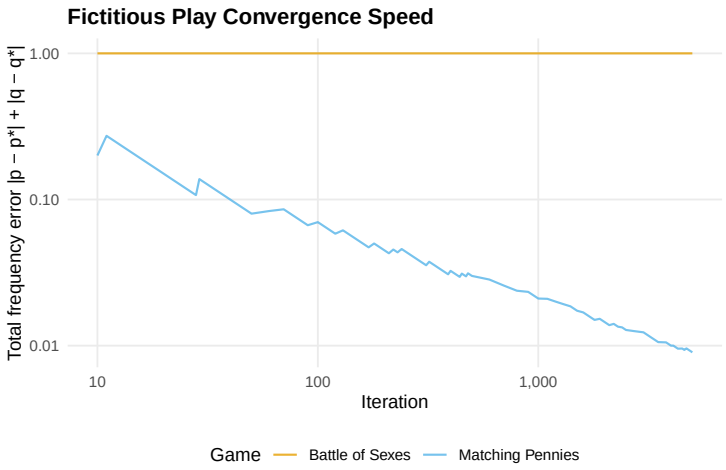
convergence_data <- list()
for (game_name in names(games_for_fp)) {
  g <- games_for_fp[[game_name]]
  fp <- fictitious_play(g$A, g$B, n_iter = 5000)

  conv <- fp$history |>
    mutate(
      error_p = abs(row_freq_1 - g$true_p),
      error_q = abs(col_freq_1 - g$true_q),
      total_error = error_p + error_q,
      game = game_name
    )
  convergence_data <- c(convergence_data, list(conv))
}

conv_df <- bind_rows(convergence_data)

conv_df |>
  filter(iter >= 10) |>
  ggplot(aes(x = iter, y = total_error, colour = game)) +
  geom_line(linewidth = 0.6, alpha = 0.8) +
  scale_colour_manual(values = okabe_ito[c(1, 2)]) +
  scale_x_log10(name = "Iteration", labels = label_comma()) +
  scale_y_log10(name = "Total frequency error |p - p*| + |q -
  ↪  q*|") +
  labs(
    title = "Fictitious Play Convergence Speed",
```

```
    colour = "Game"
  ) +
  theme_publication()
```



Both games show the expected $O(1/\sqrt{t})$ convergence rate (approximately linear on the log-log scale with slope around -0.5). Matching Pennies, being zero-sum, converges more smoothly. Battle of the Sexes shows more irregular convergence due to the coordination structure and the presence of multiple equilibria.

15.5 Extensions

The three algorithms in this chapter represent three broad families of equilibrium computation methods:

- **Enumeration methods** (support enumeration, 14) are exhaustive and exact but exponential. They are the method of choice when completeness matters — for example, when we need to certify that a game has a unique equilibrium.
- **Path-following methods** (Lemke–Howson) are polynomial in the size of the output (they find one equilibrium in time proportional to the path length) and are the workhorses of practical computation. The full Lemke–Howson algorithm, with proper degeneracy handling, is implemented in Gambit and can be accessed from R via the gtree package (13).
- **Learning dynamics** (fictitious play) provide approximate solutions and connect equilibrium theory to the question of how equilibria might arise through repeated interaction. This connects to the evolutionary and repeated-game perspectives explored in 8.

For the theoretical foundations of equilibrium computation, including the PPAD complexity class and the computational hardness of finding Nash equilibria, see Shoham & Leyton-Brown (2009) (Chapter 4) and Osborne (2004) (Chapter 12). The connection between learning dynamics and equilibrium was a central theme in the work of Nash (1950), who originally motivated his equilibrium concept partly through a "mass action" interpretation — populations of players repeatedly interacting and adjusting their strategies.

Exercises

1. **Fictitious play on a 3x3 game.** Run fictitious play on the asymmetric Rock-Paper-Scissors game from 14 (with payoffs +2 for winning and -1 for losing). Use 10,000 iterations and plot the empirical frequencies of all three actions for Player 1 over time (you will need to extend the `fictitious_play()` function to track all action frequencies, not just the first). Does fictitious play converge? Compare the final empirical frequencies to the exact Nash equilibrium found by support enumeration.

2. **Convergence rate analysis.** For the Battle of the Sexes, run fictitious play for $T \in \{100, 500, 1000, 5000, 10000\}$ iterations with 20 different random seeds for each T. Compute the mean and standard deviation of the total frequency error at each T. Plot the mean error as a function of T on a log-log scale and estimate the convergence rate exponent by fitting a line. Is the empirical rate consistent with the theoretical prediction?

3. **Shapley's counterexample.** Implement the game with payoff matrices $A = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$ and $B = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$. Run fictitious play for 20,000 iterations and plot the trajectory of Player 1's empirical frequencies in a ternary diagram (use the three frequencies as coordinates, noting they sum to 1, so two suffice for a 2D plot). Verify that the empirical frequencies cycle rather than converge, confirming Shapley's result.

4. **Lemke–Howson by hand.** For the game with $A = \begin{pmatrix} 4 & 1 \\ 2 & 3 \end{pmatrix}$ and $B = \begin{pmatrix} 3 & 2 \\ 1 & 4 \end{pmatrix}$ (after shifting to ensure positivity), write out the best-response polytopes P and Q, enumerate all vertices with their labels, and trace the Lemke–Howson path starting from label $k = 1$. Verify that the path arrives at a Nash equilibrium. You may present this as a table rather than code.

5. **Hybrid solver.** Write a function `hybrid_solver(A, B, exact_threshold = 5)` that uses support enumeration (via the `support_enumeration_general()` function from 14) when both $m \leq$ `exact_threshold` and $n \leq$

`exact_threshold`, and falls back to fictitious play with 10,000 iterations otherwise. Test your function on random games of sizes 3x3, 5x5, and 10x10. Report timing results and comment on the quality of the fictitious play approximation relative to the exact solution.

Solutions appear in D.

Part III

Simulation & Evolutionary Dynamics

Chapter 16

Monte Carlo Methods for Games

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

Monte Carlo simulation applied to game theory: estimating mixed-strategy payoffs, evaluating extensive-form strategies by sampling terminal nodes, and constructing bootstrap confidence intervals for expected values.

Learning objectives

- Explain why Monte Carlo simulation is useful when analytical solutions to games are intractable or tedious.
- Implement Monte Carlo estimation of mixed-strategy Nash equilibrium payoffs in R.
- Construct bootstrap confidence intervals for expected payoffs.
- Demonstrate convergence of Monte Carlo estimates to analytical values and interpret the rate of convergence.

16.1 Motivation

Many games have analytical solutions — we can write down the Nash equilibrium and compute expected payoffs in closed form. But as games grow in complexity, analytical solutions become impractical. Consider poker: even a

simplified version has thousands of possible deals, and evaluating a strategy requires averaging over all of them. In extensive-form games with nature moves (chance nodes), the number of terminal nodes can be enormous.

Monte Carlo simulation offers a practical alternative. Instead of exhaustively enumerating every outcome, we sample random plays of the game and average the results. The law of large numbers guarantees that our estimate converges to the true expected value, and the central limit theorem tells us how fast. This chapter develops the Monte Carlo approach for game-theoretic applications, starting with simple matrix games and building to a poker-like card game where exact computation is tedious but simulation is straightforward.

16.2 Theory

16.2.1 Monte Carlo estimation

Let X be a random variable representing a player's payoff under some strategy profile. The expected payoff is:

$$\mathbb{E}[X] = \sum_{s \in S} p(s) \cdot u(s) \tag{16.1}$$

where S is the set of outcomes, $p(s)$ is the probability of outcome s, and $u(s)$ is the associated payoff. The Monte Carlo estimator draws n independent samples $X_1, \ldots, X_n$ from the payoff distribution and computes:

$$\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^{n} X_i \tag{16.2}$$

By the strong law of large numbers, $\hat{\mu}_n \to \mathbb{E}[X]$ almost surely as $n \to \infty$.

16.2.2 Convergence rate and confidence intervals

The standard error of the Monte Carlo estimator is $\sigma/\sqrt{n}$, where σ is the standard deviation of X. This means the estimation error decreases at rate $O(1/\sqrt{n})$ — to halve the error, we need four times as many samples.

A $100(1-\alpha)\%$ confidence interval for $\mathbb{E}[X]$ is:

$$\hat{\mu}_n \pm z_{\alpha/2} \cdot \frac{\hat{\sigma}_n}{\sqrt{n}} \tag{16.3}$$

where $\hat{\sigma}_n$ is the sample standard deviation and $z_{\alpha/2}$ is the standard normal quantile.

Definition

A **Monte Carlo estimator** for a game's expected payoff samples independent plays of the game according to the players' strategies and averages the resulting payoffs. It is an unbiased, consistent estimator whose precision improves at rate $O(1/\sqrt{n})$.

16.2.3 Bootstrap confidence intervals

When the payoff distribution is non-normal or we want a distribution-free approach, the **bootstrap** provides an alternative. We resample with replacement from the observed payoffs B times, compute the mean of each resample, and take quantiles of the bootstrap distribution as confidence bounds. This is especially useful when the analytical variance is hard to derive (Osborne, 2004).

16.3 Implementation in R

16.3.1 Mixed-strategy payoff estimation

We begin with a 2×2 game where both players use mixed strategies. Consider Matching Pennies, where the unique Nash equilibrium has both players mixing uniformly over Heads and Tails.

```
# Matching Pennies payoff matrix (row player)
# Row = Heads/Tails, Col = Heads/Tails
payoff_row <- matrix(c(1, -1, -1, 1), nrow = 2, byrow = TRUE,
                     dimnames = list(c("H", "T"), c("H", "T")))

# Nash equilibrium: both players mix 50-50
p_row <- c(H = 0.5, T = 0.5)
p_col <- c(H = 0.5, T = 0.5)

# Analytical expected payoff under NE
analytical_payoff <- sum(outer(p_row, p_col) * payoff_row)
cat(glue("Analytical expected payoff (row player):
 ↪  {analytical_payoff}"), "\n")
```

```
#> Analytical expected payoff (row player): 0
```

```
# Monte Carlo estimation of the expected payoff
simulate_payoffs <- function(payoff_matrix, p_row, p_col, n_sims)
 ↪  {
```

```
  row_actions <- sample(rownames(payoff_matrix), n_sims, replace
 ↪  = TRUE,
                        prob = p_row)
  col_actions <- sample(colnames(payoff_matrix), n_sims, replace
 ↪  = TRUE,
                        prob = p_col)
  payoffs <- mapply(function(r, c) payoff_matrix[r, c],
                    row_actions, col_actions)
  payoffs
}

set.seed(42)
payoffs_10k <- simulate_payoffs(payoff_row, p_row, p_col, n_sims
 ↪  = 10000)

cat(glue("MC estimate (10,000 samples): {round(mean(payoffs_10k),
 ↪  4)}"), "\n")
```

```
#> MC estimate (10,000 samples): -0.001
```

```
cat(glue("MC std error: {round(sd(payoffs_10k) /
 ↪  sqrt(length(payoffs_10k)), 4)}"), "\n")
```

```
#> MC std error: 0.01
```

16.3.2 Convergence plot

```
set.seed(42)
n_total <- 10000
payoffs_all <- simulate_payoffs(payoff_row, p_row, p_col,
 ↪  n_total)

# Compute running mean and CI at selected sample sizes
sample_sizes <- c(seq(10, 100, by = 10), seq(150, 1000, by = 50),
                  seq(1200, 10000, by = 200))

convergence_df <- map_dfr(sample_sizes, function(n) {
  x <- payoffs_all[1:n]
  mu <- mean(x)
  se <- sd(x) / sqrt(n)
  tibble(n = n, estimate = mu, ci_lower = mu - 1.96 * se,
         ci_upper = mu + 1.96 * se)
```

```
})

p_convergence <- ggplot(convergence_df, aes(x = n, y = estimate))
 ↪  +
  geom_ribbon(aes(ymin = ci_lower, ymax = ci_upper),
              fill = okabe_ito[2], alpha = 0.3) +
  geom_line(colour = okabe_ito[1], linewidth = 0.8) +
  geom_hline(yintercept = analytical_payoff, linetype = "dashed",
             colour = okabe_ito[8], linewidth = 0.6) +
  scale_x_log10(labels = scales::comma) +
  theme_publication() +
  labs(title = "Monte Carlo Convergence: Matching Pennies",
       x = "Number of samples (log scale)",
       y = "Estimated expected payoff",
       caption = "Dashed line = analytical expected payoff (0)")

p_convergence
```

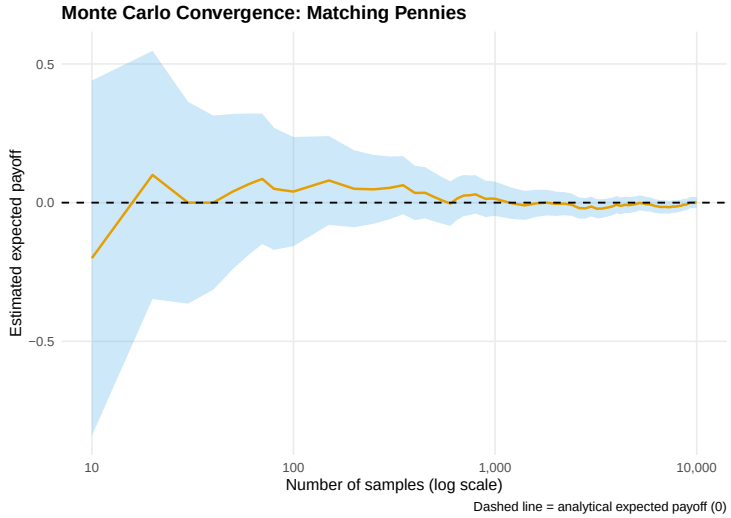


Figure 16.1: Convergence of the Monte Carlo payoff estimate to the analytical solution (dashed line at zero) for Matching Pennies under Nash equilibrium play. The shaded ribbon shows the 95% confidence interval, which narrows at rate $O(1/\sqrt{n})$.

```
save_pub_fig(p_convergence, "mc-convergence", width = 7, height =
 ↪  5)
```

16.3.3 Payoff distribution under mixed strategies

Now consider an asymmetric game where the payoff distribution is more interesting. We use a modified Battle of the Sexes with mixed-strategy equilibrium.

```
# Battle of the Sexes payoff matrix (row player)
payoff_bos <- matrix(c(3, 0, 0, 2), nrow = 2, byrow = TRUE,
                     dimnames = list(c("Opera", "Football"),
                                     c("Opera", "Football")))

# Mixed NE: row plays Opera with prob 2/5, Football with prob 3/5
# Col plays Opera with prob 3/5, Football with prob 2/5
p_row_bos <- c(Opera = 2/5, Football = 3/5)
p_col_bos <- c(Opera = 3/5, Football = 2/5)

analytical_bos <- sum(outer(p_row_bos, p_col_bos) * payoff_bos)

set.seed(42)
payoffs_bos <- simulate_payoffs(payoff_bos, p_row_bos, p_col_bos,
 ↪  n_sims = 50000)

payoff_dist_df <- tibble(payoff = payoffs_bos)

p_distribution <- ggplot(payoff_dist_df, aes(x = payoff)) +
  geom_histogram(aes(y = after_stat(density)), binwidth = 0.5,
                 fill = okabe_ito[3], colour = "white", alpha =
                  ↪  0.8) +
  geom_density(colour = okabe_ito[5], linewidth = 0.8, bw = 0.3)
   ↪  +
  geom_vline(xintercept = analytical_bos, linetype = "dashed",
             colour = okabe_ito[6], linewidth = 0.7) +
  theme_publication() +
  labs(title = "Payoff Distribution: Battle of the Sexes (Mixed
   ↪  NE)",
       x = "Row-player payoff",
       y = "Density",
       caption = glue("Dashed line = analytical expected payoff
        ↪  ({round(analytical_bos, 2)})"))

p_distribution
```

```
save_pub_fig(p_distribution, "mc-payoff-distribution", width = 7,
 ↪  height = 5)
```

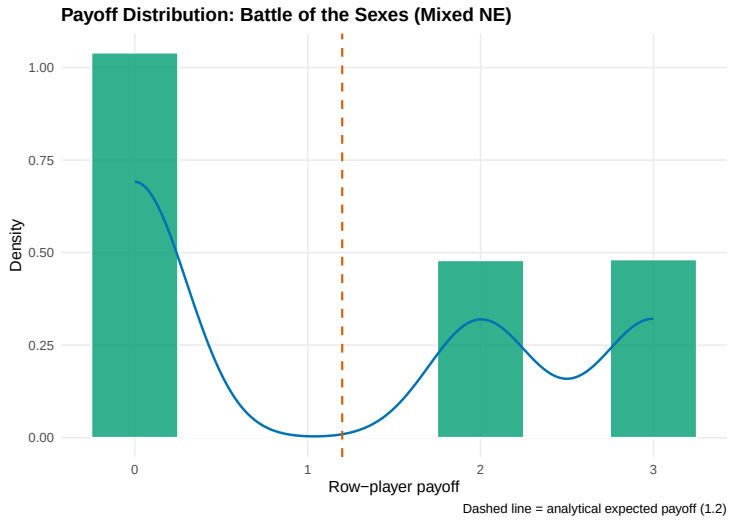


Figure 16.2: Distribution of row-player payoffs under mixed-strategy Nash equilibrium play in Battle of the Sexes. The vertical dashed line marks the analytical expected payoff. The distribution is discrete with three mass points corresponding to the three distinct payoff values.

16.4 Worked example

We now apply Monte Carlo estimation to a simplified poker-like game. Two players each receive one card from a deck of integers 1 through 10 (dealt without replacement). Player 1 observes their card and decides to **bet** or **check**. If Player 1 checks, payoffs are zero for both. If Player 1 bets, Player 2 (who has also seen their own card) decides to **call** or **fold**. If Player 2 folds, Player 1 wins a pot of 1. If Player 2 calls, the higher card wins a pot of 2 (and the loser pays 1).

Player 1's strategy: bet if card $\geq k_1$ (threshold strategy with cutoff k_1). Player 2's strategy: call if card $\geq k_2$ (threshold strategy with cutoff k_2).

```
play_poker_round <- function(k1, k2, deck_size = 10) {
  cards <- sample(deck_size, 2, replace = FALSE)
  card1 <- cards[1]
  card2 <- cards[2]

  # Player 1 decision

  if (card1 < k1) {
    return(0)  # check, no payoff
  }
```

```
  # Player 1 bets; Player 2 decision
  if (card2 < k2) {
    return(1)  # Player 2 folds, Player 1 wins pot of 1
  }

  # Showdown
  if (card1 > card2) return(2)   # Player 1 wins pot of 2
  if (card1 < card2) return(-1)  # Player 1 loses 1
  return(0)                       # tie
}

# Evaluate Player 1's expected payoff for different threshold
↪  combinations
set.seed(42)
n_sims <- 20000

threshold_grid <- expand_grid(k1 = 1:10, k2 = 1:10)

results <- threshold_grid |>
  mutate(
    payoffs = map2(k1, k2, function(k1, k2) {
      replicate(n_sims, play_poker_round(k1, k2))
    }),
    mean_payoff = map_dbl(payoffs, mean),
    se = map_dbl(payoffs, ~ sd(.x) / sqrt(length(.x)))
  )

# Find the best threshold for Player 1 (assuming Player 2 plays
↪  optimally)
# For each k1, find k2 that minimizes Player 1's payoff (Player
↪  2's best response)
best_responses <- results |>
  group_by(k1) |>
  summarise(
    min_payoff = min(mean_payoff),
    best_k2 = k2[which.min(mean_payoff)],
    .groups = "drop"
  )

optimal_k1 <- best_responses |>
  filter(min_payoff == max(min_payoff))

cat("Player 1's maximin strategy:\n")
```

```
#> Player 1's maximin strategy:
```

```
cat(glue("  Bet threshold: k1 >= {optimal_k1$k1}"), "\n")
```

```
#>   Bet threshold: k1 >= 4
```

```
cat(glue("  Player 2 best response: call if k2 >=
 ↪  {optimal_k1$best_k2}"), "\n")
```

```
#>   Player 2 best response: call if k2 >= 7
```

```
cat(glue("  Expected payoff: {round(optimal_k1$min_payoff, 4)}"),
 ↪  "\n")
```

```
#>   Expected payoff: 0.362
```

16.4.1 Bootstrap confidence interval

We use the bootstrap to construct a confidence interval for Player 1's expected payoff under the estimated optimal strategies.

```
set.seed(42)
k1_star <- optimal_k1$k1[1]
k2_star <- optimal_k1$best_k2[1]

# Generate payoff samples under optimal play
payoff_samples <- replicate(20000, play_poker_round(k1_star,
 ↪  k2_star))

# Bootstrap
B <- 5000
boot_means <- replicate(B, {
  resample <- sample(payoff_samples, replace = TRUE)
  mean(resample)
})

ci_boot <- quantile(boot_means, c(0.025, 0.975))

cat("Bootstrap 95% CI for Player 1's expected payoff:\n")
```

```
#> Bootstrap 95% CI for Player 1's expected payoff:
```

```
cat(glue("  [{round(ci_boot[1], 4)}, {round(ci_boot[2], 4)}]"),
 ↪  "\n")
```

```
#>   [0.3626, 0.387]
```

```
cat(glue("  Point estimate: {round(mean(payoff_samples), 4)}"),
 ↪  "\n")
```

```
#>   Point estimate: 0.3747
```

16.4.2 Verifying convergence

We verify that our Monte Carlo estimate stabilizes as the sample size grows:

```
check_sizes <- c(100, 500, 1000, 5000, 10000, 20000)

convergence_check <- map_dfr(check_sizes, function(n) {
  x <- payoff_samples[1:n]
  tibble(
    n = n,
    estimate = mean(x),
    se = sd(x) / sqrt(n),
    ci_width = 2 * 1.96 * sd(x) / sqrt(n)
  )
})

convergence_check |>
  mutate(across(c(estimate, se, ci_width), ~ round(.x, 4))) |>
  gt() |>
  cols_label(n = "Samples", estimate = "Estimate", se = "Std
   ↪  Error",
             ci_width = "95% CI Width") |>
  tab_header(title = "Convergence of Monte Carlo Estimate",
             subtitle = "Simplified poker game under estimated
              ↪  optimal play")
```

The table confirms the $O(1/\sqrt{n})$ convergence: each quadrupling of the sample size roughly halves the confidence interval width, consistent with (16.3).

16.5 Extensions

- **Variance reduction.** Techniques such as antithetic variates, control variates, and importance sampling can dramatically reduce the number

Convergence of Monte Carlo Estimate
Simplified poker game under estimated optimal play

Samples	Estimate	Std Error	95% CI Width
100	0.450	0.0869	0.3406
500	0.476	0.0377	0.1478
1000	0.453	0.0272	0.1065
5000	0.396	0.0123	0.0484
10000	0.377	0.0088	0.0344
20000	0.375	0.0062	0.0242

of samples needed for a given precision. These are especially valuable in extensive-form games with rare but important outcomes.

- **Monte Carlo Tree Search (MCTS).** The approach used by AlphaGo and other game-playing AI combines Monte Carlo sampling with tree search to evaluate positions in enormous game trees (Sutton & Barto, 2018). MCTS builds on the ideas in this chapter but adds selective exploration of the game tree.
- **Analytical benchmarks.** When an analytical solution exists, it serves as a powerful check on Monte Carlo code. In 18, we use exact enumeration for small tournaments and Monte Carlo for larger ones.
- **Connection to replicator dynamics.** Monte Carlo methods can estimate the basin of attraction for different equilibria in evolutionary games (20) by sampling initial conditions and simulating forward.
- For a rigorous treatment of Monte Carlo methods in game theory, see Shoham & Leyton-Brown (2009).

Exercises

1. **Analytical comparison.** For the Matching Pennies game in 16.3, compute the exact variance of a single payoff draw under Nash equilibrium play. Use this to predict the standard error at $n = 10{,}000$ samples and compare to the empirical value from the simulation.

2. **Asymmetric poker.** Modify the simplified poker game so that the deck has 20 cards and Player 1 must bet at least 2 (paying 2 to enter, winning 3 on a call). Re-run the Monte Carlo estimation to find the new optimal thresholds. How does the larger action cost change Player 1's optimal strategy?

3. **Importance sampling.** In the poker game, most deals result in a check (zero payoff) when k_1 is high. Implement importance sampling that over-

samples deals where Player 1 has a high card, reweighting by the likelihood ratio. Compare the variance of the importance-sampling estimator to the naive estimator at $n = 1{,}000$ samples.

Solutions appear in D.

Chapter 17

Agent-Based Models for Strategic Interaction

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

Building agent-based models where heterogeneous agents play games, reproduce proportional to fitness, and evolve strategies over generations — illustrated with the Hawk-Dove game.

Learning objectives

- Contrast the agent-based modeling paradigm with analytical equilibrium analysis.
- Implement a population of agents with heterogeneous strategies that interact through pairwise games.
- Model evolutionary pressure via fitness-proportional reproduction and mutation.
- Simulate the Hawk-Dove game and interpret the emergent strategy frequencies in terms of the evolutionarily stable strategy.

17.1 Motivation

Analytical game theory derives equilibria by assuming fully rational players with common knowledge. This is powerful but sometimes unrealistic. In biology, economics, and social science, we often observe populations of agents with limited information, heterogeneous strategies, and adaptive behavior. How do strategy

frequencies evolve when agents interact locally, reproduce based on success, and occasionally mutate?

Agent-based models (ABMs) answer this question computationally. Instead of solving for equilibrium, we simulate it. Each agent carries a strategy, plays games against others, earns payoffs, and the population evolves over generations. ABMs can reveal dynamics that analytical models miss: path dependence, transient oscillations, the role of population size, and the effect of mutation rates. Maynard Smith's Hawk-Dove game — a cornerstone of evolutionary game theory — provides the ideal testbed because it has a known analytical equilibrium we can compare against (Osborne, 2004).

17.2 Theory

17.2.1 The Hawk-Dove game

Two animals contest a resource of value v. Each can play **Hawk** (escalate) or **Dove** (yield). The payoff matrix is:

Table 17.1: Hawk-Dove payoff matrix, where v is the resource value and c is the cost of fighting.

	Hawk	Dove
Hawk	$(v-c)/2$	v
Dove	0	$v/2$

When $c > v$ (fighting is costly relative to the resource), the game has no pure-strategy Nash equilibrium. The mixed-strategy Nash equilibrium has each player playing Hawk with probability:

$$p^* = \frac{v}{c} \tag{17.1}$$

This is also the evolutionarily stable strategy (ESS): a population at frequency p^* cannot be invaded by either pure strategy (Osborne, 2004).

17.2.2 Agent-based evolutionary dynamics

In the ABM framework, we model a finite population of N agents. Each generation proceeds in three steps:

1. **Interaction.** Each agent plays the Hawk-Dove game against k randomly chosen opponents. Its fitness is the average payoff across these interactions.

2. **Reproduction.** The next generation is formed by sampling N agents from the current population with replacement, where the probability of being selected is proportional to fitness (shifted to be non-negative):

$$\Pr(\text{agent } i \text{ selected}) = \frac{f_i - f_{\min} + \epsilon}{\sum_{j=1}^{N}(f_j - f_{\min} + \epsilon)} \tag{17.2}$$

where f_i is agent i's fitness, $f_{\min}$ is the minimum fitness in the population, and $\epsilon > 0$ is a small constant ensuring all agents have positive selection probability.

3. **Mutation.** Each offspring independently switches strategy with probability μ (the mutation rate).

Definition

An **agent-based model** (ABM) is a computational model in which autonomous agents with individual attributes interact according to specified rules. Macro-level patterns emerge from these micro-level interactions without being explicitly programmed.

17.2.3 Connection to replicator dynamics

In the limit of large populations, weak selection, and low mutation, the ABM dynamics converge to the **replicator equation** (20):

$$\dot{x}_H = x_H \left[\pi_H(\mathbf{x}) - \bar{\pi}(\mathbf{x})\right] \tag{17.3}$$

where x_H is the frequency of Hawks, π_H is the expected payoff to a Hawk, and $\bar{\pi}$ is the population average payoff. The ABM allows us to see how finite population effects, stochastic drift, and mutation perturb these deterministic predictions.

17.3 Implementation in R

17.3.1 Core simulation functions

```r
# Hawk-Dove payoff function
hawk_dove_payoff <- function(strategy1, strategy2, v = 2, c = 5)
 ↪  {
  if (strategy1 == "Hawk" && strategy2 == "Hawk") {
    return((v - c) / 2)
  } else if (strategy1 == "Hawk" && strategy2 == "Dove") {
    return(v)
  } else if (strategy1 == "Dove" && strategy2 == "Hawk") {
    return(0)
  } else {
    return(v / 2)
  }
}

# Compute fitness for all agents via random pairwise interactions
compute_fitness <- function(population, n_interactions, v, c) {
  n <- length(population)
  fitness <- numeric(n)
  for (i in seq_len(n)) {
    opponents <- sample(seq_len(n)[-i], min(n_interactions, n -
 ↪  1))
    payoffs <- vapply(opponents, function(j) {
      hawk_dove_payoff(population[i], population[j], v, c)
    }, numeric(1))
    fitness[i] <- mean(payoffs)
  }
  fitness
}

# Fitness-proportional selection with shift
select_parents <- function(population, fitness, n) {
  shifted <- fitness - min(fitness) + 0.01
  probs <- shifted / sum(shifted)
  idx <- sample(seq_along(population), n, replace = TRUE, prob =
 ↪  probs)
  population[idx]
}

# Mutation
mutate_population <- function(population, mu) {
  strategies <- c("Hawk", "Dove")
  mutate_mask <- runif(length(population)) < mu
  population[mutate_mask] <- vapply(population[mutate_mask],
 ↪  function(s) {
    sample(setdiff(strategies, s), 1)
```

```
  }, character(1))
  population
}
```

17.3.2 Running the ABM

```
run_hawk_dove_abm <- function(n_agents = 200, n_generations =
 ↪  300,
                               initial_hawk_freq = 0.5,
                               n_interactions = 10,
                               mutation_rate = 0.01,
                               v = 2, c = 5, seed = 42) {
  set.seed(seed)

  # Initialize population
  n_hawks <- round(n_agents * initial_hawk_freq)
  population <- c(rep("Hawk", n_hawks), rep("Dove", n_agents -
 ↪  n_hawks))

  # Storage for tracking
  history <- tibble(
    generation = integer(),
    hawk_freq = double(),
    mean_fitness = double()
  )

  for (gen in seq_len(n_generations)) {
    hawk_freq <- mean(population == "Hawk")
    fitness <- compute_fitness(population, n_interactions, v, c)

    history <- bind_rows(history, tibble(
      generation = gen,
      hawk_freq = hawk_freq,
      mean_fitness = mean(fitness)
    ))

    # Selection and reproduction
    population <- select_parents(population, fitness, n_agents)

    # Mutation
    population <- mutate_population(population, mutation_rate)
  }
```

```
  history
}
```

17.3.3 Strategy frequency evolution

```
v <- 2
c_val <- 5
ess_freq <- v / c_val

# Run three independent simulations with different initial
 ↪  conditions
runs <- list(
  list(init = 0.1, seed = 42, label = "Run 1 (10% Hawk)"),
  list(init = 0.5, seed = 123, label = "Run 2 (50% Hawk)"),
  list(init = 0.9, seed = 456, label = "Run 3 (90% Hawk)")
)

history_all <- map_dfr(runs, function(r) {
  run_hawk_dove_abm(
    n_agents = 200, n_generations = 300,
    initial_hawk_freq = r$init, n_interactions = 10,
    mutation_rate = 0.01, v = v, c = c_val, seed = r$seed
  ) |>
    mutate(run = r$label)
})

p_evolution <- ggplot(history_all, aes(x = generation, y =
 ↪  hawk_freq,
                                       colour = run)) +
  geom_line(linewidth = 0.6, alpha = 0.85) +
  geom_hline(yintercept = ess_freq, linetype = "dashed",
             colour = okabe_ito[8], linewidth = 0.6) +
  scale_colour_manual(values = okabe_ito[c(1, 2, 3)], name =
   ↪  "Simulation") +
  theme_publication() +
  labs(title = "Hawk-Dove ABM: Strategy Frequency Over
   ↪  Generations",
       x = "Generation",
       y = "Fraction playing Hawk",
       caption = glue("Dashed line = ESS frequency (v/c =
        ↪  {ess_freq})"))

p_evolution
```

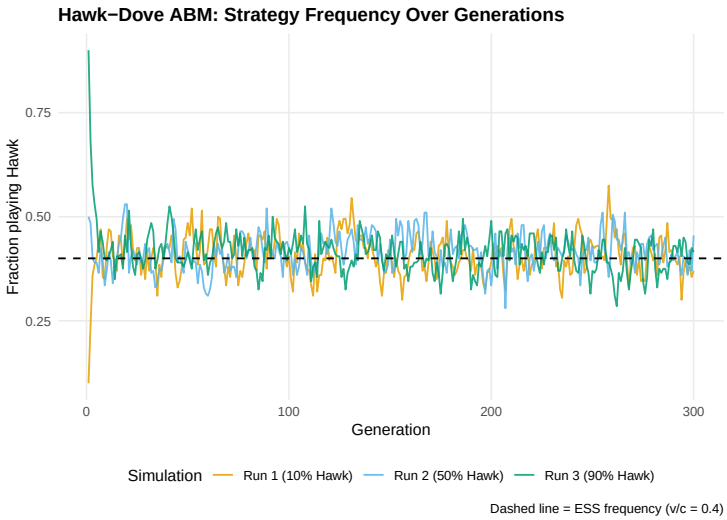


Figure 17.1: Strategy frequency evolution in the Hawk-Dove agent-based model across three independent runs. The dashed horizontal line marks the analytical ESS frequency $p^* = v/c = 0.4$. Despite starting from different initial conditions, all runs converge to fluctuate around the predicted equilibrium.

```
save_pub_fig(p_evolution, "abm-strategy-evolution", width = 7,
 ↪  height = 5)
```

17.3.4 Population fitness over time

```
# Compute the analytical equilibrium fitness
# At ESS: pi_bar = p*[p*(v-c)/2 + (1-p*)v] + (1-p*)[p*0 +
 ↪  (1-p*)v/2]
p_star <- ess_freq
pi_bar_star <- p_star * (p_star * (v - c_val)/2 + (1 - p_star) *
 ↪  v) +
               (1 - p_star) * (p_star * 0 + (1 - p_star) * v/2)

p_fitness <- ggplot(history_all, aes(x = generation, y =
 ↪  mean_fitness,
                                     colour = run)) +
  geom_line(linewidth = 0.6, alpha = 0.85) +
  geom_hline(yintercept = pi_bar_star, linetype = "dashed",
             colour = okabe_ito[8], linewidth = 0.6) +
  scale_colour_manual(values = okabe_ito[c(1, 2, 3)], name =
   ↪  "Simulation") +
```

```
  theme_publication() +
  labs(title = "Hawk-Dove ABM: Average Population Fitness",
       x = "Generation",
       y = "Mean payoff per interaction",
       caption = glue("Dashed line = fitness at ESS
        ↪  ({round(pi_bar_star, 2)})"))

p_fitness
```

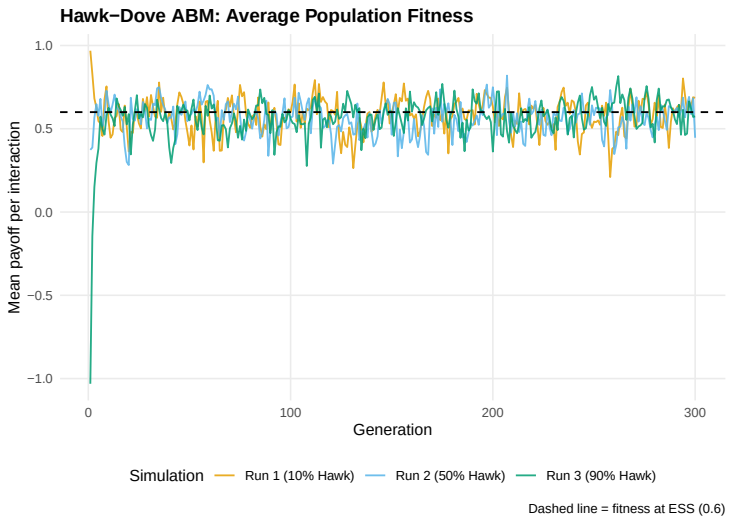


Figure 17.2: Average population fitness over generations in the Hawk-Dove ABM. Fitness is highest when the population is near the ESS. Early generations show rapid fitness changes as the population adjusts from its initial state; later generations fluctuate around the equilibrium fitness level.

```
save_pub_fig(p_fitness, "abm-fitness", width = 7, height = 5)
```

17.4 Worked example

Let us trace through the first few generations of a small population to understand the mechanics.

```
set.seed(42)
n_agents <- 20
population <- c(rep("Hawk", 14), rep("Dove", 6))  # 70% Hawk
```

```
cat("Initial population (20 agents, 70% Hawk):\n")
```

```
#> Initial population (20 agents, 70% Hawk):
```

```
cat(glue("  Hawks: {sum(population == 'Hawk')}, ",
         "Doves: {sum(population == 'Dove')}"), "\n\n")
```

```
#>   Hawks: 14, Doves: 6
```

```
# Generation 1: compute fitness
fitness <- compute_fitness(population, n_interactions = 5, v = v,
 ↪  c = c_val)

fitness_by_type <- tibble(
  strategy = population,
  fitness = fitness
) |>
  group_by(strategy) |>
  summarise(mean_fitness = round(mean(fitness), 3),
            min_fitness = round(min(fitness), 3),
            max_fitness = round(max(fitness), 3),
            .groups = "drop")

cat("Generation 1 fitness summary:\n")
```

```
#> Generation 1 fitness summary:
```

```
print(fitness_by_type)
```

```
#> # A tibble: 2 x 4
#>   strategy mean_fitness min_fitness max_fitness
#>   <chr>           <dbl>       <dbl>       <dbl>
#> 1 Dove            0.267         0           0.6
#> 2 Hawk           -0.4          -1.5         1.3
```

```
# Selection: Hawks have lower average fitness when
 ↪  over-represented
cat("\nWith 70% Hawks (above ESS of 40%), Hawks fight each other
 ↪  often.\n")
```

```
#>
#> With 70% Hawks (above ESS of 40%), Hawks fight each other often.
```

```
cat("Doves benefit from facing other Doves more than Hawks
 ↪  benefit from\n")
```

```
#> Doves benefit from facing other Doves more than Hawks benefit from
```

```
cat("fighting other Hawks (since (v-c)/2 < 0 when c > v).\n\n")
```

```
#> fighting other Hawks (since (v-c)/2 < 0 when c > v).
```

```
# Next generation
population_new <- select_parents(population, fitness, n_agents)
population_new <- mutate_population(population_new, mu = 0.01)

cat("After selection and mutation:\n")
```

```
#> After selection and mutation:
```

```
cat(glue("  Hawks: {sum(population_new == 'Hawk')}, ",
         "Doves: {sum(population_new == 'Dove')}"), "\n")
```

```
#>   Hawks: 14, Doves: 6
```

```
cat(glue("  Hawk frequency: {mean(population_new == 'Hawk')}"),
 ↪  "\n")
```

```
#>   Hawk frequency: 0.7
```

The population moves toward the ESS. Because Hawks are overrepresented (70% vs. the ESS of 40%), they frequently fight each other and incur the costly loss of $(v - c)/2 = -1.5$. Doves, meanwhile, avoid fighting costs entirely. Selection therefore favors Doves, pulling the population toward the equilibrium frequency.

17.4.1 Sensitivity to parameters

Effect of Mutation Rate on Equilibrium Behavior

Steady-state statistics from last 100 of 300 generations

Mutation rate	Mean Hawk freq	SD of Hawk freq
0.001	0.417	0.045
0.010	0.407	0.052
0.050	0.419	0.040
0.100	0.444	0.041

```
# Effect of mutation rate on variance of equilibrium frequency
mutation_rates <- c(0.001, 0.01, 0.05, 0.1)

sensitivity_df <- map_dfr(mutation_rates, function(mu) {
  h <- run_hawk_dove_abm(n_agents = 200, n_generations = 300,
                         initial_hawk_freq = 0.5, n_interactions
                          ↪  = 10,
                         mutation_rate = mu, v = v, c = c_val,
                          ↪  seed = 42)
  # Use last 100 generations as "steady state"
  steady <- tail(h, 100)
  tibble(
    mutation_rate = mu,
    mean_hawk_freq = round(mean(steady$hawk_freq), 3),
    sd_hawk_freq = round(sd(steady$hawk_freq), 3)
  )
})

sensitivity_df |>
  gt() |>
  cols_label(mutation_rate = "Mutation rate",
             mean_hawk_freq = "Mean Hawk freq",
             sd_hawk_freq = "SD of Hawk freq") |>
  tab_header(title = "Effect of Mutation Rate on Equilibrium
   ↪  Behavior",
             subtitle = "Steady-state statistics from last 100 of
              ↪  300 generations")
```

Higher mutation rates increase the variance around the ESS — agents are more frequently pushed away from the equilibrium — but the mean frequency remains close to $p^* = 0.4$ across all mutation rates. This robustness is a signature of

the ESS: selection pressure always pushes back toward equilibrium, regardless of perturbations (Neumann & Morgenstern, 1944).

17.5 Extensions

- **Spatial structure.** When agents interact only with neighbors on a grid, spatial clusters can form and persist. Cooperation can be sustained in spatial Prisoner's Dilemma even without reciprocity (19).
- **Multiple strategies.** The ABM framework extends naturally to games with more than two pure strategies. With three or more strategies, the dynamics can exhibit limit cycles (as in Rock-Paper-Scissors) rather than convergence to a fixed point.
- **Evolutionary stability.** The analytical counterpart to ABM convergence is the concept of an evolutionarily stable strategy. See 21 for formal definitions and the relationship to Nash equilibrium.
- **Replicator dynamics.** The continuous-time limit of the ABM's selection step yields the replicator equation (20). Comparing the ABM to the replicator ODE reveals where finite-population effects matter.
- **Network structure.** When the interaction graph is not well-mixed but has network structure, cooperation dynamics change qualitatively (22). For the foundational analysis, see Shoham & Leyton-Brown (2009).

Exercises

1. **Three strategies.** Add a third strategy, **Retaliator**, that plays Dove against Doves and Hawk against Hawks (it matches the opponent's type). This requires each agent to observe its opponent's strategy before acting. Run the ABM and report whether Retaliator can invade a Hawk-Dove population at the ESS.

2. **Population size effects.** Run the Hawk-Dove ABM with population sizes of 20, 100, 500, and 2000 agents for 300 generations each. Plot the standard deviation of the Hawk frequency (over the last 100 generations) against population size. What relationship do you observe, and why?

3. **Stag Hunt dynamics.** Replace the Hawk-Dove payoff matrix with the Stag Hunt: mutual cooperation (Stag, Stag) yields 4 each, mutual defection (Hare, Hare) yields 2 each, and miscoordination yields 0 for the Stag player and 2 for the Hare player. Run the ABM from initial cooperation frequencies of 0.3, 0.5, and 0.8. Does the population always converge to the same equilibrium? Explain in terms of the game's two Nash equilibria.

Solutions appear in D.

Chapter 18

Replicating Axelrod's Tournament

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

A computational replication of Axelrod's 1980 iterated Prisoner's Dilemma tournament in R, exploring why Tit for Tat won and what it teaches about the evolution of cooperation.

Learning objectives

- Describe the design of Axelrod's 1980 iterated Prisoner's Dilemma tournament and its key findings.
- Implement a round-robin tournament harness in R with configurable strategies.
- Reproduce the result that Tit for Tat outperforms more complex strategies.
- Visualize tournament outcomes and analyze what makes a strategy successful.

18.1 Motivation

In 1980, political scientist Robert Axelrod invited game theorists to submit strategies for an iterated Prisoner's Dilemma computer tournament. Each strategy would play every other strategy (and itself) in a round-robin, accumulating

points over 200 rounds per match. The submissions ranged from simple to elaborate — some used sophisticated pattern recognition, others were just a few lines of code.

The winner was the simplest strategy entered: **Tit for Tat**, submitted by Anatol Rapoport. It cooperates on the first move and then copies whatever the opponent did on the previous move. Axelrod (1981) ran a second tournament after publishing the first results, inviting even more entries with knowledge of the first outcome. Tit for Tat won again.

This result — that a strategy requiring no memory beyond the last round, no exploitation, and no complex reasoning could outperform sophisticated alternatives — became one of the most influential findings in the study of cooperation. In this chapter, we replicate the tournament in R.

18.2 Theory

18.2.1 The iterated Prisoner's Dilemma

The stage game payoffs follow the standard convention:

Table 18.1: Prisoner's Dilemma payoff structure with $T > R > P > S$ and $2R > T + S$.

	Cooperate	Defect
Cooperate	R, R	S, T
Defect	T, S	P, P

With the canonical values $T = 5$, $R = 3$, $P = 1$, $S = 0$: temptation to defect is high, but mutual cooperation ($R = 3$) beats the average of exploitation and being exploited ($(T + S)/2 = 2.5$).

When the game is repeated, players can condition their current action on the history of play. This opens the door to conditional cooperation: a player might cooperate as long as the opponent cooperates, but punish defection.

18.2.2 Axelrod's four properties of successful strategies

Analyzing the tournament results, Axelrod identified four properties shared by the top-performing strategies:

1. **Nice** — never the first to defect.
2. **Retaliatory** — punish defection promptly.

3. **Forgiving** — return to cooperation after the opponent does.
4. **Clear** — behavior is predictable, enabling mutual cooperation.

Tit for Tat exemplifies all four: it never defects first (nice), defects immediately after the opponent does (retaliatory), cooperates immediately when the opponent returns to cooperation (forgiving), and its logic is transparent (clear).

18.3 Implementation in R

18.3.1 Strategy definitions

We define a library of classic strategies. Each strategy is a function that takes two arguments — the player's own history and the opponent's history — and returns `"C"` or `"D"`.

```
strategy_tit_for_two_tats <- function(own, opp) {
  n <- length(opp)
  if (n < 2) return("C")
  if (opp[n] == "D" && opp[n - 1] == "D") "D" else "C"
}

strategy_suspicious_tft <- function(own, opp) {
  if (length(opp) == 0) return("D")
  opp[length(opp)]
}

strategy_grudger <- function(own, opp) {
  if (any(opp == "D")) "D" else "C"
}

strategy_pavlov <- function(own, opp) {
  if (length(own) == 0) return("C")
  last_own <- own[length(own)]
  last_opp <- opp[length(opp)]
  if (last_own == last_opp) "C" else "D"
}

strategy_random <- function(own, opp) {
  sample(c("C", "D"), 1)
}

strategies <- list(
  "Tit for Tat"      = strategy_tit_for_tat,
  "Always Cooperate"  = strategy_always_cooperate,
```

```
  "Always Defect"     = strategy_always_defect,
  "Tit for Two Tats"  = strategy_tit_for_two_tats,
  "Suspicious TFT"    = strategy_suspicious_tft,
  "Grudger"           = strategy_grudger,
  "Pavlov"            = strategy_pavlov,
  "Random"            = strategy_random
)
```

18.3.2 Round-robin tournament

```
run_tournament <- function(strategies, rounds = 200) {
  names_s <- names(strategies)
  n <- length(strategies)
  results <- matrix(0, nrow = n, ncol = n,
                    dimnames = list(names_s, names_s))
  for (i in seq_len(n)) {
    for (j in seq(i, n)) {
      scores <- run_match(strategies[[i]], strategies[[j]],
 ↪  rounds)
      results[i, j] <- results[i, j] + scores[1]
      results[j, i] <- results[j, i] + scores[2]
    }
  }
  results
}

set.seed(42)
results <- run_tournament(strategies, rounds = 200)

totals <- rowSums(results)
ranking <- sort(totals, decreasing = TRUE)

cat("Tournament Results (total scores):\n\n")
```

```
#> Tournament Results (total scores):
```

```
for (i in seq_along(ranking)) {
  cat(sprintf("  %d. %-20s %5d\n", i, names(ranking)[i],
   ↪  ranking[i]))
}
```

```
#>   1. Tit for Two Tats      4761
```

```
#>   2. Tit for Tat           4725
#>   3. Grudger               4585
#>   4. Pavlov                4538
#>   5. Always Cooperate      4518
#>   6. Random                3900
#>   7. Always Defect         3416
#>   8. Suspicious TFT        3372
```

18.3.3 Visualizing the results

```
ranking_df <- tibble(
  strategy = factor(names(ranking), levels =
   ↪  rev(names(ranking))),
  score = as.integer(ranking)
)

nice_strategies <- c("Tit for Tat", "Always Cooperate", "Tit for
 ↪  Two Tats",
                     "Grudger", "Pavlov")
ranking_df$nice <- names(ranking) %in% nice_strategies

p_ranking <- ggplot(ranking_df, aes(x = score, y = strategy, fill
 ↪  = nice)) +
  geom_col(width = 0.7) +
  scale_fill_manual(values = c("TRUE" = okabe_ito[3], "FALSE" =
   ↪  okabe_ito[6]),
                    labels = c("TRUE" = "Nice", "FALSE" = "Not
                     ↪  nice"),
                    name = "Strategy type") +
  theme_publication() +
  labs(title = "Axelrod Tournament: Strategy Rankings",
       x = "Total score", y = NULL)

p_ranking
```

```
save_pub_fig(p_ranking, "axelrod-ranking", width = 7, height = 5)
```

18.3.4 Head-to-head heatmap

```
heatmap_df <- as.data.frame(as.table(results))
```

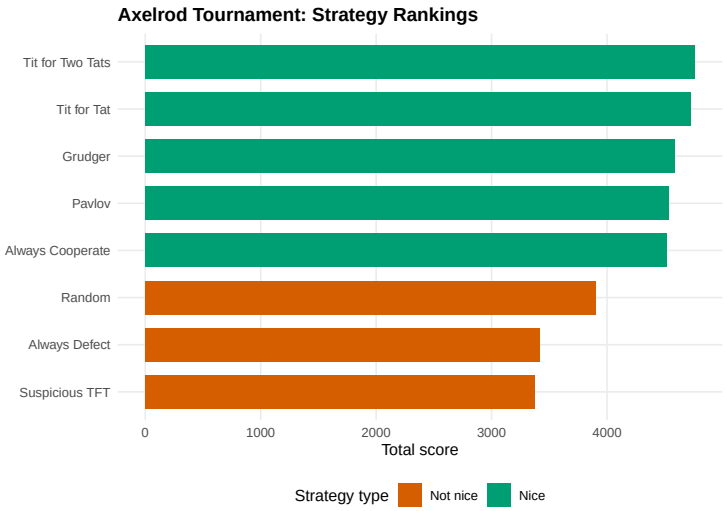


Figure 18.1: Tournament ranking by total score. Tit for Tat wins the round-robin tournament, consistent with Axelrod's original finding. Nice strategies (those that never defect first) cluster at the top.

```
names(heatmap_df) <- c("Player", "Opponent", "Score")

p_heat <- ggplot(heatmap_df, aes(x = Opponent, y = Player, fill
 ↪  = Score)) +
  geom_tile(colour = "white", linewidth = 0.5) +
  geom_text(aes(label = Score), size = 2.8) +
  scale_fill_gradient(low = okabe_ito[2], high = okabe_ito[1]) +
  theme_publication(base_size = 10) +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  labs(title = "Head-to-Head Scores",
       x = "Opponent", y = "Player")

p_heat
```

```
save_pub_fig(p_heat, "axelrod-heatmap", width = 7, height = 6)
```

18.4 Worked example

Let us trace a single match between Tit for Tat and Suspicious TFT (which defects first, then copies):

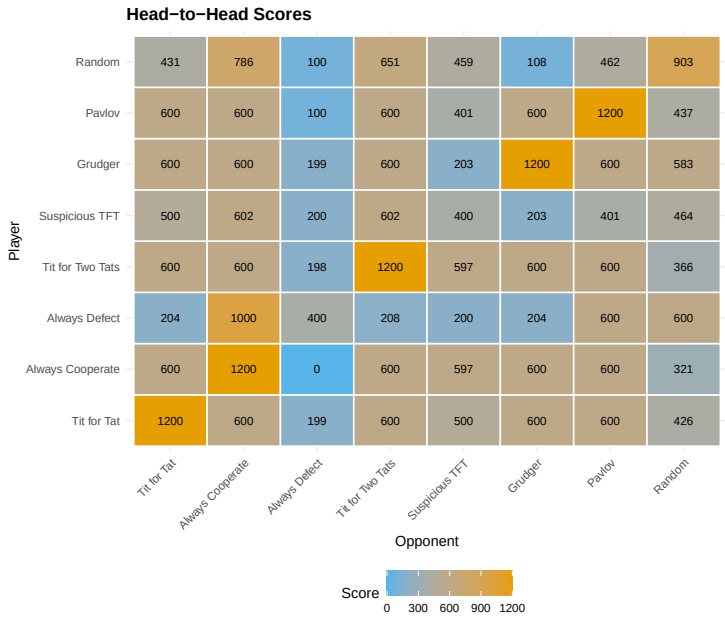


Figure 18.2: Head-to-head scores in the round-robin tournament. Rows are the focal player; columns are the opponent. Always Defect scores highest against Always Cooperate but poorly against retaliatory strategies.

```
set.seed(42)
h1 <- character(0)
h2 <- character(0)

for (r in 1:10) {
  a1 <- strategy_tit_for_tat(h1, h2)
  a2 <- strategy_suspicious_tft(h2, h1)
  h1 <- c(h1, a1)
  h2 <- c(h2, a2)
}

trace_df <- tibble(
  Round = 1:10,
  `Tit for Tat` = h1,
  `Suspicious TFT` = h2
)

cat("First 10 rounds: Tit for Tat vs. Suspicious TFT\n\n")
```

```
#> First 10 rounds: Tit for Tat vs. Suspicious TFT
```

```
print(trace_df, n = 10)
```

```
#> # A tibble: 10 x 3
#>    Round `Tit for Tat` `Suspicious TFT`
#>    <int> <chr>         <chr>
#>  1     1 C             D
#>  2     2 D             C
#>  3     3 C             D
#>  4     4 D             C
#>  5     5 C             D
#>  6     6 D             C
#>  7     7 C             D
#>  8     8 D             C
#>  9     9 C             D
#> 10    10 D             C
```

The pattern reveals alternating retaliation: Suspicious TFT defects in round 1, Tit for Tat retaliates in round 2, and the two lock into a cycle of mutual recrimination. This is a well-known weakness of Tit for Tat — it can get trapped in defection cycles against strategies that defect first. Tit for Two Tats avoids this by requiring two consecutive defections before retaliating, at the cost of being more exploitable.

18.5 Extensions

- **Evolutionary dynamics**: What happens when successful strategies reproduce and unsuccessful ones die out? This leads to replicator dynamics on strategy populations (20) and the concept of evolutionarily stable strategies (21).
- **Spatial structure**: When players interact only with neighbors on a lattice, cooperation can survive even without reciprocity (19).
- **Noise**: In noisy environments where actions are sometimes misperceived, Tit for Tat's strict reciprocity becomes a liability. Generous Tit for Tat (cooperate with some probability after opponent's defection) outperforms in noisy settings.
- For the original analysis, see Axelrod (1981). For a modern computational treatment with hundreds of strategies, see the Python `axelrod` library (accessible via `reticulate`; see 14 for the general pattern).

Exercises

1. **Add a strategy.** Implement a "Gradual" strategy that retaliates with an increasing number of defections after each opponent defection (1 defection after the first, 2 after the second, etc.), then returns to cooperation. Add it to the tournament and report the new rankings.

2. **Vary the number of rounds.** Run the tournament with 50, 200, and 1000 rounds. How do the rankings change? Why might Always Defect do better in shorter tournaments?

3. **Cooperation rate.** Modify `run_match()` to return the cooperation rate (fraction of rounds where both players cooperated) in addition to scores. Which pair of strategies achieves the highest mutual cooperation rate? Which achieves the lowest?

Solutions appear in D.

Chapter 19

The Spatial Prisoner's Dilemma

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

A computational exploration of Nowak and May's spatial Prisoner's Dilemma, showing how local interaction on a lattice allows cooperation to survive through cluster formation — even without memory, reciprocity, or reputation.

Learning objectives

- Explain why spatial structure can sustain cooperation in the Prisoner's Dilemma without reciprocity or punishment.
- Implement a lattice-based spatial PD with Moore neighborhood interaction and imitation dynamics in R.
- Visualize the emergence and persistence of cooperator clusters on a grid.
- Analyze how the fraction of cooperators evolves over time under different payoff parameters.

19.1 Motivation

In 18, cooperation emerged because players could remember past interactions and condition their behavior on history. But many biological and social systems lack this luxury. Bacteria, plants, and even commuters on a highway interact with nearby neighbors without tracking individual histories. Can cooperation

survive when there is no memory, no reputation, and no punishment — only spatial structure?

In 1992, Martin Nowak and Robert May (1992) showed that the answer is yes. They placed players on a two-dimensional grid where each cell plays a one-shot Prisoner's Dilemma with its immediate neighbors. After each round, every cell adopts the strategy of whichever neighbor (including itself) scored highest. The result was striking: cooperators formed compact clusters that could resist invasion by defectors, producing intricate, dynamic spatial patterns.

This chapter replicates their model in R. We will build a lattice simulation, watch cooperation clusters emerge, and measure how population-level cooperation rates evolve over time.

19.2 Theory

19.2.1 From well-mixed to spatial populations

Standard evolutionary game theory assumes a **well-mixed population**: every individual is equally likely to interact with every other. Under this assumption, defection dominates in a one-shot Prisoner's Dilemma because a defector always earns more than a cooperator against any given opponent.

Spatial structure breaks this assumption. When interaction is **local** — each player meets only its geographic neighbors — cooperators can form clusters. Within a cluster, cooperators interact mainly with other cooperators, earning the mutual cooperation payoff R many times. A defector on the boundary of such a cluster exploits its cooperator neighbors but has fewer cooperator neighbors than cooperators in the cluster interior do.

19.2.2 The Nowak–May model

The model works as follows:

1. **Grid**: An $N \times N$ lattice with periodic boundary conditions (a torus, so edges wrap around).
2. **Strategies**: Each cell is either a cooperator (C) or a defector (D).
3. **Interaction**: Each cell plays a one-shot PD with each of its eight Moore neighbors and itself (nine games total). The cell's score is the sum of payoffs from all nine games.
4. **Update**: Synchronously, every cell adopts the strategy of the highest-scoring cell in its Moore neighborhood (including itself). Ties are broken in favor of the current occupant.

With canonical PD payoffs ($T > R > P > S$), the dynamics depend critically on the temptation-to-defect parameter $b = T/R$. For moderate values of b, cooperators and defectors coexist in dynamic spatial patterns. For very high b, defection takes over; for low b, cooperation dominates.

19.2.3 Why clusters protect cooperators

Consider a 3×3 block of cooperators surrounded by defectors. The cooperator at the center plays nine cooperators (including itself), scoring $9R$. A defector adjacent to this block has at most three cooperator neighbors, scoring at most $3T + 6P$. When $9R > 3T + 6P$ — that is, when $b < 3(R - P/R) + P$ approximately — the interior cooperator outscores the boundary defector, and the cluster holds. This is the geometric logic behind spatial cooperation.

19.3 Implementation in R

19.3.1 Grid initialization and neighbor scoring

```
initialize_grid <- function(n, prob_coop = 0.5) {
  matrix(sample(c("C", "D"), n * n, replace = TRUE,
                prob = c(prob_coop, 1 - prob_coop)),
         nrow = n, ncol = n)
}

get_moore_neighbors <- function(grid, i, j) {
  n <- nrow(grid)
  rows <- ((i - 2):(i)) %% n + 1
  cols <- ((j - 2):(j)) %% n + 1
  grid[rows, cols]
}

score_cell <- function(grid, i, j) {
  neighbors <- get_moore_neighbors(grid, i, j)
  focal <- grid[i, j]
  total <- 0
  for (s in as.vector(neighbors)) {
    total <- total + play_round(focal, s)[1]
  }
  total
}

compute_scores <- function(grid) {
```

```
  n <- nrow(grid)
  scores <- matrix(0, nrow = n, ncol = n)
  for (i in seq_len(n)) {
    for (j in seq_len(n)) {
      scores[i, j] <- score_cell(grid, i, j)
    }
  }
  scores
}
```

19.3.2 Update rule: imitate the best neighbor

```
update_grid <- function(grid, scores) {
  n <- nrow(grid)
  new_grid <- grid
  for (i in seq_len(n)) {
    for (j in seq_len(n)) {
      rows <- ((i - 2):(i)) %% n + 1
      cols <- ((j - 2):(j)) %% n + 1
      neighborhood_scores <- scores[rows, cols]
      best <- which(neighborhood_scores ==
 ↪  max(neighborhood_scores),
                    arr.ind = TRUE)
      if (nrow(best) == 1) {
        bi <- rows[best[1, 1]]
        bj <- cols[best[1, 2]]
        new_grid[i, j] <- grid[bi, bj]
      }
      # Tie: keep current strategy (new_grid[i,j] already equals
       ↪  grid[i,j])
    }
  }
  new_grid
}
```

19.3.3 Running the simulation

```
run_spatial_pd <- function(n, generations, prob_coop = 0.5, seed
 ↪  = 42) {
  set.seed(seed)
  grid <- initialize_grid(n, prob_coop)
```

```
  history <- vector("list", generations + 1)
  history[[1]] <- grid
  coop_frac <- numeric(generations + 1)
  coop_frac[1] <- mean(grid == "C")

  for (g in seq_len(generations)) {
    scores <- compute_scores(grid)
    grid <- update_grid(grid, scores)
    history[[g + 1]] <- grid
    coop_frac[g + 1] <- mean(grid == "C")
  }

  list(history = history, coop_frac = coop_frac)
}
```

19.3.4 Visualizing a grid snapshot

```
grid_to_df <- function(grid, gen) {
  n <- nrow(grid)
  expand.grid(row = seq_len(n), col = seq_len(n)) |>
    mutate(strategy = as.vector(grid),
           generation = gen)
}
```

19.4 Worked example

We simulate a 30×30 grid with 50% initial cooperators for 100 generations, using the canonical PD payoffs ($T = 5$, $R = 3$, $P = 1$, $S = 0$).

```
result <- run_spatial_pd(n = 30, generations = 100, prob_coop =
 ↪  0.5, seed = 42)
```

```
snapshot_gens <- c(0, 5, 20, 100)
snapshot_df <- map_dfr(snapshot_gens, function(g) {
  grid_to_df(result$history[[g + 1]], g)
})
snapshot_df$generation <- factor(
  snapshot_df$generation,
  levels = snapshot_gens,
```

```
  labels = paste("Generation", snapshot_gens)
)

p_snapshots <- ggplot(snapshot_df,
                      aes(x = col, y = row, fill = strategy)) +
  geom_tile() +
  facet_wrap(~generation, ncol = 2) +
  scale_fill_manual(values = c("C" = okabe_ito[1], "D" =
   ↪  okabe_ito[5]),
                    labels = c("C" = "Cooperate", "D" =
                     ↪  "Defect"),
                    name = "Strategy") +
  coord_equal() +
  theme_publication() +
  theme(axis.text = element_blank(),
        axis.ticks = element_blank(),
        panel.grid.major = element_blank()) +
  labs(title = "Spatial Prisoner's Dilemma: Grid Snapshots",
       x = NULL, y = NULL)

p_snapshots
```

```
save_pub_fig(p_snapshots, "spatial-pd-snapshots", width = 8,
 ↪  height = 8)
```

The initial random arrangement (generation 0) shows a salt-and-pepper mixture. By generation 5, cooperators have begun to aggregate into clusters while isolated cooperators surrounded by defectors have been converted. By generation 20, the pattern has largely stabilized: compact cooperator clusters persist, separated by defector boundaries. The clusters are not static — their edges shift over time — but the overall cooperation level remains roughly stable.

```
coop_df <- tibble(
  generation = 0:100,
  cooperation = result$coop_frac
)

p_coop <- ggplot(coop_df, aes(x = generation, y = cooperation)) +
  geom_line(colour = okabe_ito[1], linewidth = 1) +
  geom_hline(yintercept = mean(result$coop_frac[51:101]),
             linetype = "dashed", colour = okabe_ito[8]) +
  scale_y_continuous(labels = label_percent(), limits = c(0, 1))
   ↪  +
  theme_publication() +
```

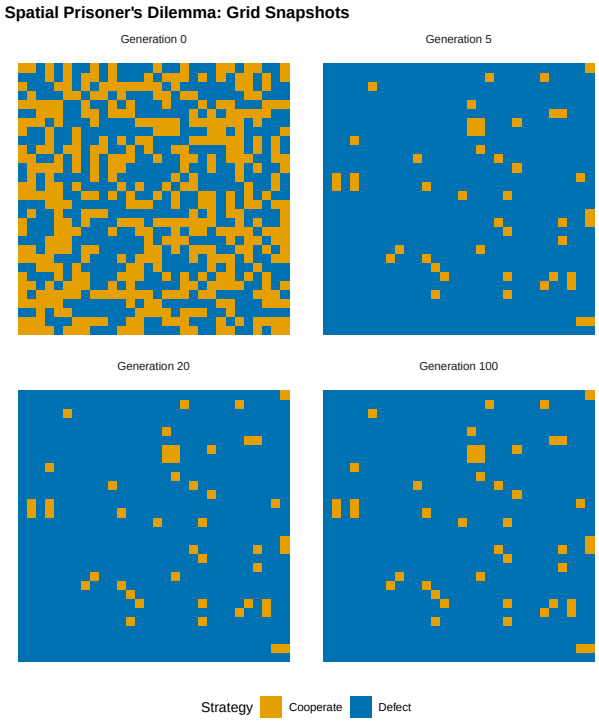


Figure 19.1: Grid snapshots at generations 0, 5, 20, and 100. Cooperators (orange) form compact clusters that resist invasion by defectors (blue). By generation 20, the spatial pattern has largely stabilized into a dynamic equilibrium.

```
  labs(title = "Cooperation Fraction Over Time",
       x = "Generation", y = "Fraction cooperating")

p_coop
```

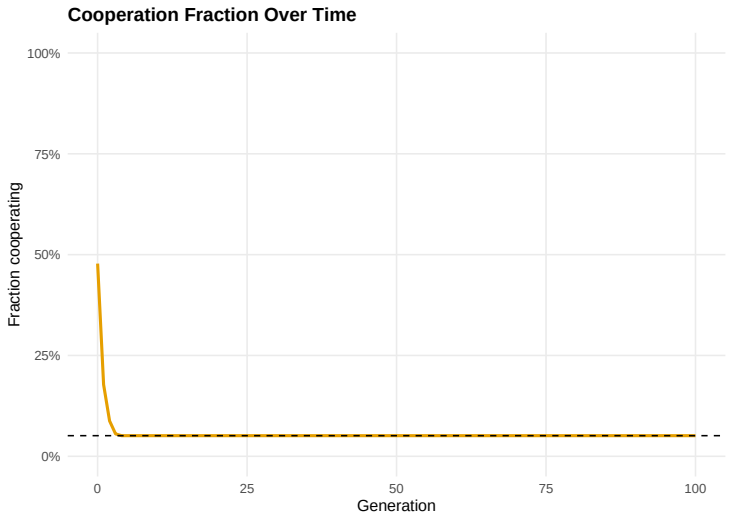


Figure 19.2: Cooperation fraction over 100 generations. After an initial transient where isolated cooperators are eliminated, the cooperation rate stabilizes around a dynamic equilibrium. Spatial structure sustains a substantial level of cooperation even in the one-shot Prisoner's Dilemma.

```
save_pub_fig(p_coop, "spatial-pd-cooperation", width = 7, height
 ↪  = 5)
```

```
cat(sprintf("Initial cooperation rate: %.1f%%\n",
            result$coop_frac[1] * 100))
```

```
#> Initial cooperation rate: 47.8%
```

```
cat(sprintf("Final cooperation rate:   %.1f%%\n",
            result$coop_frac[101] * 100))
```

```
#> Final cooperation rate:   5.1%
```

```
cat(sprintf("Mean cooperation (gen 50-100): %.1f%%\n",
            mean(result$coop_frac[51:101]) * 100))
```

```
#> Mean cooperation (gen 50-100): 5.1%
```

The cooperation rate drops sharply in the first few generations as isolated cooperators are eliminated. It then stabilizes as surviving cooperators form self-reinforcing clusters. The long-run cooperation rate depends on the payoff parameters — with the canonical values used here, spatial structure sustains cooperation at a level far above zero, even though defection would dominate in a well-mixed population.

19.5 Extensions

- **Varying temptation**: As the temptation payoff T increases (holding other payoffs fixed), the sustainable cooperation rate decreases. There is a critical threshold above which cooperators go extinct even with spatial structure. Experimenting with this threshold is a good exercise.
- **Stochastic update rules**: Instead of deterministic imitation of the best neighbor, one can use probabilistic rules where the probability of adopting a neighbor's strategy increases with the neighbor's payoff advantage. This connects to the **Fermi update rule** and reduces sensitivity to small payoff differences.
- **Network structure**: The lattice is a special case of a network. Cooperation dynamics on random graphs, scale-free networks, and small-world networks exhibit qualitatively different behavior (22).
- **Evolutionary stability**: The spatial model shows that cooperation can be an evolutionarily stable configuration even when it is not an ESS in the well-mixed case (21).
- For the original model, see Nowak & May (1992). For a comprehensive review of spatial evolutionary games, see Szabó & Fáth (2007).

Exercises

1. **Effect of grid size.** Run the spatial PD on grids of size 10×10, 30×30, and 50×50 with the same initial cooperation probability (50%). How does the long-run cooperation rate depend on grid size? Explain why boundary effects matter more on smaller grids.

2. **Temptation threshold.** Fix $R = 3$, $P = 1$, $S = 0$ and vary T from 3.5 to 8 in increments of 0.5. For each value, run the simulation for 100

generations and record the final cooperation rate. Plot cooperation rate vs. T and identify the approximate threshold where cooperators go extinct. (*Hint*: modify `play_round()` to accept custom payoffs.)

3. **Initial cluster.** Instead of random initial placement, start with a single 5×5 block of cooperators in the center of a 30×30 grid of defectors. Does the cooperator block survive? Grow? How does the outcome depend on T?

Solutions appear in D.

Chapter 20

Replicator Dynamics

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

An introduction to the replicator equation — the central dynamical system of evolutionary game theory — with ODE solutions in R for Rock-Paper-Scissors and Hawk-Dove games, including simplex visualization.

Learning objectives

- State the replicator equation and explain the biological and game-theoretic intuition behind each term.
- Use `deSolve` to numerically integrate the replicator equation for arbitrary payoff matrices.
- Visualize replicator dynamics on the two-dimensional simplex for three-strategy games.
- Analyze convergence to equilibrium in Hawk-Dove and cycling behavior in Rock-Paper-Scissors.

20.1 Motivation

In 18 and 19, we studied discrete agents making discrete choices. But many evolutionary and economic processes are better described in continuous time: strategy frequencies shift gradually as more successful types reproduce faster. How do we model this?

The **replicator equation**, introduced by Taylor and Jonker (1978) and named by Schuster and Sigmund (1983), provides the answer. It is the fundamental

dynamical system of evolutionary game theory, describing how the frequency of each strategy in a large population changes over time as a function of its fitness relative to the population average.

The replicator equation connects game theory to biology (it arises from natural selection), economics (it describes learning dynamics in large populations), and dynamical systems (it produces fixed points, limit cycles, and chaos). In this chapter, we solve it numerically in R for two classic games and visualize the resulting dynamics.

20.2 Theory

20.2.1 The replicator equation

Consider a large population where individuals play one of n strategies. Let x_i be the fraction of the population playing strategy i, so $\mathbf{x} = (x_1, \dots, x_n)$ lies on the **simplex** $\Delta^{n-1} = \{\mathbf{x} \geq 0 : \sum_i x_i = 1\}$.

The **fitness** of strategy i against the current population is:

$$f_i(\mathbf{x}) = \sum_j a_{ij} x_j = (A\mathbf{x})_i$$

where $A = (a_{ij})$ is the payoff matrix. The **average fitness** is:

$$\bar{f}(\mathbf{x}) = \sum_i x_i f_i(\mathbf{x}) = \mathbf{x}^\top A \mathbf{x}$$

The **replicator equation** is:

$$\dot{x}_i = x_i(f_i(\mathbf{x}) - \bar{f}(\mathbf{x})), \quad i = 1, \dots, n \tag{20.1}$$

The interpretation is elegant: a strategy grows when its fitness exceeds the population average, and shrinks when it falls below. The rate of change is proportional to both the current frequency x_i (rare strategies change slowly in absolute terms) and the fitness advantage $f_i - \bar{f}$.

20.2.2 Key properties

1. **Simplex invariance**: If $\mathbf{x}(0) \in \Delta^{n-1}$, then $\mathbf{x}(t) \in \Delta^{n-1}$ for all $t \geq 0$. Frequencies always sum to one.
2. **Rest points**: Every vertex of the simplex (a pure-strategy population) is a rest point. Interior rest points correspond to mixed Nash equilibria of the underlying game.
3. **Folk theorem**: In two-strategy games, every ESS is an asymptotically stable rest point of the replicator dynamics, and every stable rest point is a Nash equilibrium.

20.2.3 Two example games

Rock-Paper-Scissors (RPS). The payoff matrix is:

$$A_{\text{RPS}} = \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix}$$

The unique interior equilibrium $\mathbf{x}^* = (1/3, 1/3, 1/3)$ is a center: trajectories cycle around it forever without converging. This is the classic example of non-convergent replicator dynamics.

Hawk-Dove. With benefit $V = 2$ and cost $C = 4$:

$$A_{\text{HD}} = \begin{pmatrix} (V-C)/2 & V \\ 0 & V/2 \end{pmatrix} = \begin{pmatrix} -1 & 2 \\ 0 & 1 \end{pmatrix}$$

The interior equilibrium is at $x_H^* = V/C = 0.5$. This is an ESS and the replicator dynamics converge to it from any interior initial condition.

20.3 Implementation in R

20.3.1 Solving with `run_replicator()`

The helper function `run_replicator()` (defined in `R/replicator.R`) wraps `deSolve::ode()` to integrate the replicator equation. It takes a payoff matrix, initial frequencies, and a time grid, returning a data frame of strategy frequencies over time.

```
A_rps <- matrix(c( 0, -1,  1,
                   1,  0, -1,
                  -1,  1,  0), nrow = 3, byrow = TRUE)

A_hd <- matrix(c(-1, 2,
                  0, 1), nrow = 2, byrow = TRUE)
```

20.3.2 Simplex coordinate transform

To plot three-strategy dynamics on a two-dimensional ternary diagram, we project from the simplex to Cartesian coordinates using the standard transform:

```
simplex_to_cartesian <- function(x1, x2, x3) {
  tibble(
    sx = 0.5 * (2 * x2 + x3) / (x1 + x2 + x3),
    sy = (sqrt(3) / 2) * x3 / (x1 + x2 + x3)
  )
}
```

20.3.3 Running RPS dynamics from multiple initial conditions

```
set.seed(42)
n_traj <- 6
initial_conditions <- map(seq_len(n_traj), function(i) {
  raw <- runif(3)
  raw / sum(raw)
})

rps_trajectories <- map_dfr(seq_along(initial_conditions),
 ↪  function(k) {
  x0 <- initial_conditions[[k]]
  out <- run_replicator(A_rps, x0, times = seq(0, 40, by = 0.05))
  coords <- simplex_to_cartesian(out[, 2], out[, 3], out[, 4])
  coords$trajectory <- factor(k)
  coords$time <- out$time
  coords
})
```

20.3.4 Running Hawk-Dove dynamics

```
hd_initials <- c(0.1, 0.3, 0.5, 0.7, 0.9)

hd_trajectories <- map_dfr(hd_initials, function(p) {
  x0 <- c(p, 1 - p)
  out <- run_replicator(A_hd, x0, times = seq(0, 20, by = 0.05))
  df <- as_tibble(out)
  names(df) <- c("time", "Hawk", "Dove")
  df$initial <- paste0("x0 = ", p)
  df
})
```

20.4 Worked example

We trace the replicator dynamics for specific initial conditions in both games.

20.4.1 Rock-Paper-Scissors

Starting from $\mathbf{x}_0 = (0.6, 0.3, 0.1)$ — a population dominated by Rock:

```
x0_rps <- c(0.6, 0.3, 0.1)
out_rps <- run_replicator(A_rps, x0_rps, times = seq(0, 40, by =
 ↪  0.1))
names(out_rps) <- c("time", "Rock", "Paper", "Scissors")

cat("RPS dynamics from x0 = (0.6, 0.3, 0.1):\n\n")
```

```
#> RPS dynamics from x0 = (0.6, 0.3, 0.1):
```

```
cat(sprintf("  t = %2.0f:  Rock = %.3f  Paper = %.3f  Scissors =
 ↪  %.3f\n",
            0, x0_rps[1], x0_rps[2], x0_rps[3]))
```

```
#>   t =  0:  Rock = 0.600  Paper = 0.300  Scissors = 0.100
```

```
for (t_val in c(5, 10, 20, 40)) {
  row <- out_rps[out_rps$time == t_val, ]
  cat(sprintf("  t = %2.0f:  Rock = %.3f  Paper = %.3f  Scissors
   ↪  = %.3f\n",
              t_val, row$Rock, row$Paper, row$Scissors))
}
```

```
#>   t =  5:  Rock = 0.087  Paper = 0.488  Scissors = 0.426
#>   t = 10:  Rock = 0.551  Paper = 0.091  Scissors = 0.358
#>   t = 20:  Rock = 0.164  Paper = 0.163  Scissors = 0.673
#>   t = 40:  Rock = 0.298  Paper = 0.602  Scissors = 0.100
```

The frequencies cycle endlessly around the interior equilibrium $(1/3, 1/3, 1/3)$. Rock starts high, attracting Paper players who can exploit it; Paper then rises, attracting Scissors; Scissors rises, attracting Rock — and the cycle repeats. The orbit never converges because the interior equilibrium is a center (neutrally stable), not an attractor.

```
triangle <- tibble(
  sx = c(0, 1, 0.5, 0),
  sy = c(0, 0, sqrt(3) / 2, 0)
)

center <- simplex_to_cartesian(1/3, 1/3, 1/3)

p_rps <- ggplot() +
  geom_path(data = triangle, aes(x = sx, y = sy),
            colour = "grey40", linewidth = 0.5) +
  geom_path(data = rps_trajectories,
            aes(x = sx, y = sy, colour = trajectory),
            linewidth = 0.5, alpha = 0.8) +
  geom_point(data = center, aes(x = sx, y = sy),
             shape = 3, size = 3, stroke = 1.2, colour =
               ↪ okabe_ito[8]) +
  annotate("text", x = -0.05, y = -0.04, label = "Rock",
           size = 3.5, fontface = "bold") +
  annotate("text", x = 1.05, y = -0.04, label = "Paper",
           size = 3.5, fontface = "bold") +
  annotate("text", x = 0.5, y = sqrt(3)/2 + 0.04, label =
   ↪ "Scissors",
           size = 3.5, fontface = "bold") +
  scale_colour_manual(values = okabe_ito[1:6]) +
  coord_equal(xlim = c(-0.1, 1.1), ylim = c(-0.1, 1.0)) +
  theme_publication() +
  theme(axis.text = element_blank(),
        axis.ticks = element_blank(),
        panel.grid.major = element_blank(),
        legend.position = "none") +
  labs(title = "RPS Replicator Dynamics on the Simplex",
       x = NULL, y = NULL)

p_rps
```

```
save_pub_fig(p_rps, "replicator-rps-simplex", width = 7, height =
 ↪ 6)
```

20.4.2 Hawk-Dove

Starting from $x_H = 0.1$ (a population with few Hawks):

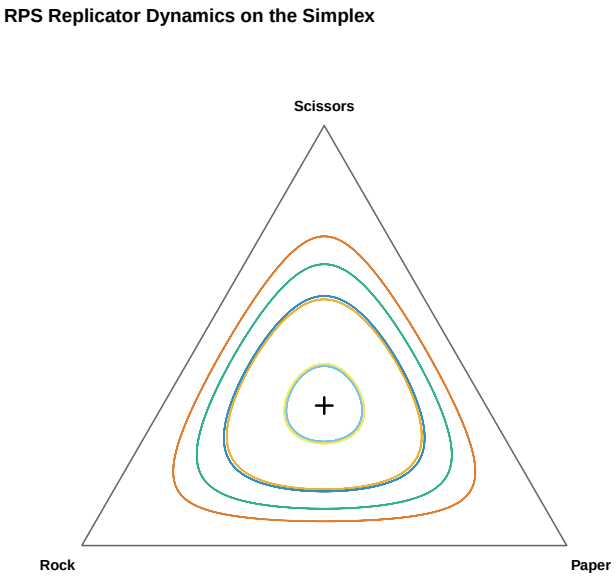


Figure 20.1: Replicator dynamics for Rock-Paper-Scissors on the two-dimensional simplex. Six trajectories from random initial conditions cycle around the interior equilibrium at (1/3, 1/3, 1/3). The orbits are closed curves — the system never converges, reflecting the non-transitive dominance structure of the game.

```
x0_hd <- c(0.1, 0.9)
out_hd <- run_replicator(A_hd, x0_hd, times = seq(0, 20, by =
 ↪  0.1))
names(out_hd) <- c("time", "Hawk", "Dove")

cat("Hawk-Dove dynamics from x0 = (0.1, 0.9):\n\n")
```

```
#> Hawk-Dove dynamics from x0 = (0.1, 0.9):
```

```
for (t_val in c(0, 2, 5, 10, 20)) {
  row <- out_hd[out_hd$time == t_val, ]
  cat(sprintf("  t = %2.0f:  Hawk = %.3f  Dove = %.3f\n",
              t_val, row$Hawk, row$Dove))
}
```

```
#>   t =  0:  Hawk = 0.100  Dove = 0.900
#>   t =  2:  Hawk = 0.280  Dove = 0.720
#>   t =  5:  Hawk = 0.446  Dove = 0.554
#>   t = 10:  Hawk = 0.496  Dove = 0.504
#>   t = 20:  Hawk = 0.500  Dove = 0.500
```

Hawks are rare but earn a high payoff by exploiting the predominantly Dove population. The Hawk frequency rises rapidly at first, then decelerates as more Hawk-Hawk encounters occur (which are costly). The system converges smoothly to the ESS at $x_H^* = V/C = 0.5$.

```
p_hd <- ggplot(hd_trajectories, aes(x = time, y = Hawk, colour =
 ↪  initial)) +
  geom_line(linewidth = 0.8) +
  geom_hline(yintercept = 0.5, linetype = "dashed", colour =
   ↪  okabe_ito[8]) +
  annotate("text", x = 18, y = 0.53, label = "ESS: x[H] == 0.5",
           parse = TRUE, size = 3.5, colour = okabe_ito[8]) +
  scale_colour_manual(values = okabe_ito[1:5], name = "Initial
   ↪  Hawk freq.") +
  scale_y_continuous(limits = c(0, 1), labels = label_percent())
   ↪  +
  theme_publication() +
  labs(title = "Hawk-Dove Replicator Dynamics",
       x = "Time", y = "Hawk frequency")

p_hd
```

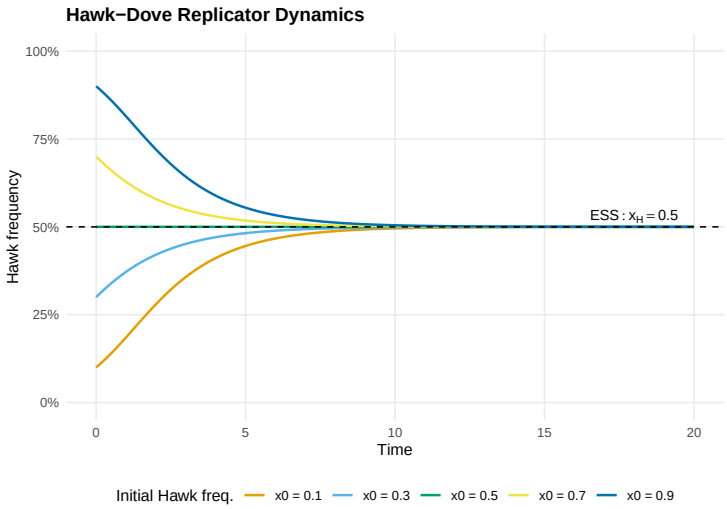


Figure 20.2: Hawk frequency over time for five different initial conditions. All trajectories converge to the ESS at $x_H = 0.5$ (dashed line), regardless of starting point. Initial conditions below the ESS see Hawk increase; those above see it decrease.

```
save_pub_fig(p_hd, "replicator-hd-dynamics", width = 7, height =
↪  5)
```

The Hawk-Dove dynamics illustrate a core result in evolutionary game theory: **the ESS is a global attractor of the replicator dynamics**. No matter how the population starts (as long as both strategies are present), it converges to the mixed equilibrium where Hawks and Doves coexist. This is in stark contrast to RPS, where the dynamics cycle forever.

20.4.3 Comparing the two games

The difference between the two games reflects a deep structural distinction:

- **Hawk-Dove** has a symmetric 2×2 payoff matrix with an interior ESS. The replicator dynamics have a Lyapunov function (a quantity that strictly decreases along trajectories), guaranteeing convergence.
- **Rock-Paper-Scissors** has a skew-symmetric payoff matrix. The quantity $H(\mathbf{x}) = x_1 x_2 x_3$ is conserved along trajectories (it is a constant of motion), producing closed orbits rather than convergence.

This shows that the replicator equation's behavior depends qualitatively on the game's payoff structure — a theme we will revisit when studying evolutionarily stable strategies in 21.

20.5 Extensions

- **Asymmetric replicator dynamics**: When players occupy different roles (e.g., owner vs. intruder), the dynamics involve two coupled replicator equations, one for each population. This leads to higher-dimensional phase spaces.
- **Mutation and selection**: Adding a small mutation rate μ transforms the replicator equation into the **replicator-mutator equation**: $\dot{x}_i = \sum_j x_j f_j Q_{ji} - x_i \bar{f}$, where Q is a mutation matrix. This can turn centers into spirals.
- **Discrete-time replicator**: The discrete-time version $x_i' = x_i f_i / \bar{f}$ has different stability properties and can exhibit chaos even in low dimensions.
- **Connection to spatial models**: The well-mixed replicator dynamics serve as a baseline for the spatial models of 19. When spatial structure matters, local frequency-dependent selection can sustain strategies that the replicator equation predicts will go extinct.
- For a comprehensive treatment, see Hofbauer & Sigmund (1998). For the connection between replicator dynamics and learning, see Fudenberg & Levine (1998).

Exercises

1. **Three-strategy Hawk-Dove-Retaliator.** Define a 3×3 payoff matrix for a game with Hawks, Doves, and Retaliators (Retaliators play Dove against Doves and Hawk against Hawks). With $V = 2$ and $C = 4$, the matrix is:

$$A = \begin{pmatrix} -1 & 2 & -1 \\ 0 & 1 & 1 \\ -1 & 1 & 0.5 \end{pmatrix}$$

   Use `run_replicator()` to solve the dynamics from three different initial conditions. Plot the trajectories on the simplex. Does the system converge? If so, to what equilibrium?

2. **Sensitivity to initial conditions in RPS.** Run the RPS dynamics from $\mathbf{x}_0 = (0.5, 0.3, 0.2)$ and from $\mathbf{x}_0 = (0.501, 0.3, 0.199)$. Plot both trajectories on the simplex. How different are the orbits? Relate your finding to the conservation of $H(\mathbf{x}) = x_1 x_2 x_3$.

3. **Payoff perturbation.** Modify the RPS payoff matrix to be slightly asymmetric: change the (1,2) entry from -1 to -1.1. Run the dynamics and plot the trajectory. Does the system still cycle, or does it now converge to a vertex? Explain why breaking the skew-symmetry changes the qualitative dynamics.

Solutions appear in D.

Chapter 21

Evolutionarily Stable Strategies

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

Formalizing evolutionary stability: ESS conditions, computational verification for symmetric games, invasion dynamics, and the Hawk-Dove-Retaliator game.

Learning objectives

- State the formal ESS conditions and explain why they strengthen Nash equilibrium.
- Implement a computational check for ESS in arbitrary symmetric games.
- Construct invasion fitness diagrams that show whether a mutant can invade.
- Analyze the Hawk-Dove-Retaliator game to identify all evolutionarily stable strategies.

21.1 Motivation

Natural selection does not optimize — it filters. A population playing strategy σ^* is stable not because σ^* is "best" in some absolute sense, but because no rare mutant strategy can gain a foothold. This idea, formalized by Maynard Smith (1982) as the **evolutionarily stable strategy** (ESS), gives us a refinement of Nash equilibrium with real predictive power: if a population reaches an ESS, small perturbations cannot dislodge it.

The concept matters beyond biology. Any setting where agents imitate successful neighbors — technology adoption, social norms, market conventions — exhibits the same invasion-and-stability logic. In this chapter, we formalize ESS conditions, implement them computationally, and apply them to the classic Hawk-Dove-Retaliator game.

21.2 Theory

21.2.1 The ESS definition

Consider a symmetric two-player game with payoff function $u(\sigma_i, \sigma_j)$ giving the payoff to a player using strategy σ_i against an opponent using σ_j. A strategy σ^* is an **evolutionarily stable strategy** if for every mutant strategy $\sigma \neq \sigma^*$, at least one of the following holds:

1. **Strict Nash condition**: $u(\sigma^*, \sigma^*) > u(\sigma, \sigma^*)$
2. **Stability condition**: $u(\sigma^*, \sigma^*) = u(\sigma, \sigma^*)$ **and** $u(\sigma^*, \sigma) > u(\sigma, \sigma)$

The first condition says the incumbent does strictly better against itself than any mutant does against the incumbent. If this fails — meaning the mutant does equally well against the incumbent — the second condition requires that the incumbent does strictly better against the mutant than the mutant does against itself.

21.2.2 Relationship to Nash equilibrium

Every ESS is a symmetric Nash equilibrium, but not every Nash equilibrium is an ESS. A symmetric Nash equilibrium (σ^*, σ^*) satisfies $u(\sigma^*, \sigma^*) \geq u(\sigma, \sigma^*)$ for all σ. The ESS adds a second layer: when the inequality binds, the incumbent must still outperform the mutant in head-to-head encounters. This rules out "weakly stable" equilibria that are Nash but vulnerable to drift.

21.2.3 Invasion dynamics

When a small fraction ϵ of mutants enters a population of incumbents, the expected payoff to a mutant is:

$$W_{\text{mutant}}(\epsilon) = (1 - \epsilon)\, u(\sigma, \sigma^*) + \epsilon\, u(\sigma, \sigma)$$

and the expected payoff to an incumbent is:

$$W_{\text{incumbent}}(\epsilon) = (1-\epsilon)\,u(\sigma^*, \sigma^*) + \epsilon\,u(\sigma^*, \sigma)$$

The strategy σ^* resists invasion if $W_{\text{incumbent}}(\epsilon) > W_{\text{mutant}}(\epsilon)$ for all sufficiently small $\epsilon > 0$. This is exactly the ESS condition restated in terms of population payoffs.

21.3 Implementation in R

21.3.1 Checking ESS conditions for a symmetric game

We write a function that takes a payoff matrix for a symmetric game and checks each pure strategy for ESS status.

```
check_ess <- function(A) {
  n <- nrow(A)
  strategies <- rownames(A)
  if (is.null(strategies)) strategies <- paste("S", seq_len(n))

  results <- tibble(
    strategy = strategies,
    is_nash = FALSE,
    strict_nash = FALSE,
    is_ess = FALSE,
    failing_mutant = NA_character_
  )

  for (i in seq_len(n)) {
    # Check Nash: u(i, i) >= u(j, i) for all j
    payoff_ii <- A[i, i]
    payoffs_ji <- A[, i]  # column i: payoffs of all strategies
↪   against i
    is_nash <- all(payoff_ii >= payoffs_ji)
    is_strict <- all(payoff_ii > payoffs_ji[-i])

    results$is_nash[i] <- is_nash
    results$strict_nash[i] <- is_strict

    if (!is_nash) {
      worst <- which.max(payoffs_ji - payoff_ii)
      results$failing_mutant[i] <- strategies[worst]
      next
    }
```

```
    if (is_strict) {
      results$is_ess[i] <- TRUE
      next
    }

    # Check stability condition for ties
    ess <- TRUE
    for (j in seq_len(n)) {
      if (j == i) next
      if (A[j, i] == payoff_ii) {
        # Tied - must have u(i, j) > u(j, j)
        if (A[i, j] <= A[j, j]) {
          ess <- FALSE
          results$failing_mutant[i] <- strategies[j]
          break
        }
      }
    }
    results$is_ess[i] <- ess
  }
  results
}
```

21.3.2 Testing on the Hawk-Dove game

The classic Hawk-Dove game (also called Chicken) with value $V = 2$ and cost $C = 4$:

```
V <- 2
C_cost <- 4

hd_matrix <- matrix(
  c((V - C_cost) / 2,  V,
    0,                 V / 2),
  nrow = 2, byrow = TRUE,
  dimnames = list(c("Hawk", "Dove"), c("Hawk", "Dove"))
)

cat("Hawk-Dove payoff matrix:\n")
```

```
#> Hawk-Dove payoff matrix:
```

```
print(hd_matrix)
```

```
#>      Hawk Dove
#> Hawk   -1    2
#> Dove    0    1
```

```
cat("\nESS check:\n")
```

```
#>
#> ESS check:
```

```
hd_ess <- check_ess(hd_matrix)
print(hd_ess)
```

```
#> # A tibble: 2 x 5
#>   strategy is_nash strict_nash is_ess failing_mutant
#>   <chr>    <lgl>   <lgl>       <lgl>  <chr>
#> 1 Hawk     FALSE   FALSE       FALSE  Dove
#> 2 Dove     FALSE   FALSE       FALSE  Hawk
```

Neither pure strategy is an ESS — Hawks can be invaded by Doves and vice versa. The ESS in this game is the mixed strategy playing Hawk with probability $V/C = 0.5$.

21.3.3 Invasion fitness landscape

```
eps <- seq(0, 0.5, length.out = 200)
p_star <- V / C_cost  # ESS mixture: Hawk with probability 0.5

# Effective payoffs for mixed ESS incumbent vs pure mutants
A <- hd_matrix
u_mix_mix <- p_star^2*A[1,1] + p_star*(1-p_star)*(A[1,2]+A[2,1])
 ↪  + (1-p_star)^2*A[2,2]
u_H_mix <- p_star*A[1,1] + (1-p_star)*A[1,2]
u_mix_H <- p_star*A[1,1] + (1-p_star)*A[2,1]
u_H_H   <- A[1,1]

W_inc_H <- (1 - eps) * u_mix_mix + eps * u_mix_H
W_mut_H <- (1 - eps) * u_H_mix   + eps * u_H_H
adv_hawk <- W_inc_H - W_mut_H
```

```
# Pure Dove mutant invading mixed ESS
u_D_mix <- p_star * hd_matrix[2,1] + (1-p_star) * hd_matrix[2,2]
u_mix_D <- p_star * hd_matrix[1,2] + (1-p_star) * hd_matrix[2,2]
u_D_D   <- hd_matrix[2,2]

W_inc_D <- (1 - eps) * u_mix_mix + eps * u_mix_D
W_mut_D <- (1 - eps) * u_D_mix   + eps * u_D_D
adv_dove <- W_inc_D - W_mut_D

invasion_df <- tibble(
  epsilon = rep(eps, 2),
  advantage = c(adv_hawk, adv_dove),
  mutant = rep(c("Hawk mutant", "Dove mutant"), each =
   ↪  length(eps))
)

p_invasion <- ggplot(invasion_df, aes(x = epsilon, y = advantage,
 ↪  colour = mutant)) +
  geom_line(linewidth = 1) +
  geom_hline(yintercept = 0, linetype = "dashed", colour =
   ↪  "grey40") +
  scale_colour_manual(values = c("Hawk mutant" = okabe_ito[6],
                                 "Dove mutant" = okabe_ito[3]),
                      name = "Mutant type") +
  theme_publication() +
  labs(title = "Invasion Fitness: Mixed ESS in Hawk-Dove",
       x = expression("Mutant fraction " * epsilon),
       y = "Incumbent advantage (payoff difference)")

p_invasion
```

```
save_pub_fig(p_invasion, "ess-invasion", width = 7, height = 5)
```

21.3.4 Basins of attraction in a 3-strategy game

We visualize the dynamics on the 2-simplex using the replicator equation from `R/replicator.R`. Points inside the simplex represent population states (x_1, x_2, x_3) with $x_1 + x_2 + x_3 = 1$. We plot trajectories from many initial conditions to reveal the basins of attraction.

```
# Hawk-Dove-Retaliator payoff matrix (V=2, C=4)
# Retaliator: plays Dove against Dove, plays Hawk against Hawk
```

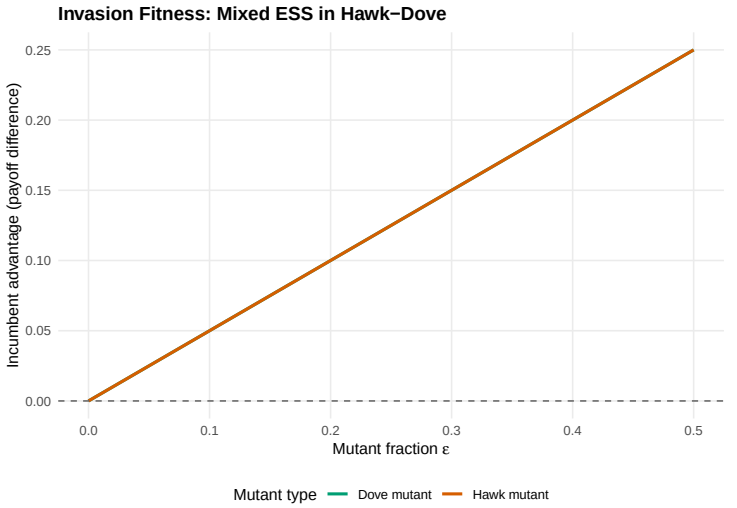


Figure 21.1: Invasion fitness landscape for the Hawk-Dove game. The plot shows the payoff advantage of the incumbent over a mutant as a function of mutant fraction. A positive value means the incumbent resists invasion. At the ESS mixture (Hawk with probability 0.5), both pure-strategy mutants have zero advantage at low frequency but negative advantage as they grow, confirming evolutionary stability of the mixed ESS.

```
hdr_matrix <- matrix(
  c((V - C_cost)/2,   V,           (V - C_cost)/2,
    0,                 V/2,         V/2,
    (V - C_cost)/2,   V/2,         V/2),
  nrow = 3, byrow = TRUE,
  dimnames = list(c("Hawk", "Dove", "Retaliator"),
                  c("Hawk", "Dove", "Retaliator"))
)

cat("Hawk-Dove-Retaliator payoff matrix:\n")
```

```
#> Hawk-Dove-Retaliator payoff matrix:
```

```
print(hdr_matrix)
```

```
#>            Hawk Dove Retaliator
#> Hawk         -1    2         -1
#> Dove          0    1          1
#> Retaliator   -1    1          1
```

```
# Simplex coordinates: map (x1, x2, x3) to 2D
# Vertices: Hawk = (0, 0), Dove = (1, 0), Retaliator = (0.5,
 ↪  sqrt(3)/2)
to_simplex_xy <- function(x1, x2, x3) {
  px <- x2 + 0.5 * x3
  py <- (sqrt(3) / 2) * x3
  tibble(sx = px, sy = py)
}

# Generate initial conditions on a grid inside the simplex
generate_simplex_grid <- function(n_per_side = 8) {
  pts <- list()
  for (i in seq(0, n_per_side)) {
    for (j in seq(0, n_per_side - i)) {
      k <- n_per_side - i - j
      x1 <- i / n_per_side
      x2 <- j / n_per_side
      x3 <- k / n_per_side
      # Avoid vertices and edges
      if (x1 > 0.02 && x2 > 0.02 && x3 > 0.02) {
        pts <- c(pts, list(c(x1, x2, x3)))
      }
    }
  }
```

```
  pts
}

initial_conditions <- generate_simplex_grid(12)

# Run replicator dynamics from each initial condition
all_trajectories <- list()

for (idx in seq_along(initial_conditions)) {
  x0 <- initial_conditions[[idx]]
  names(x0) <- c("Hawk", "Dove", "Retaliator")
  sol <- run_replicator(hdr_matrix, x0, times = seq(0, 80, by =
 ↪  0.5))

  coords <- to_simplex_xy(sol$Hawk, sol$Dove, sol$Retaliator)
  traj_df <- tibble(
    sx = coords$sx,
    sy = coords$sy,
    time = sol$time,
    traj_id = idx,
    start_hawk = x0[1]
  )
  all_trajectories[[idx]] <- traj_df
}

traj_data <- bind_rows(all_trajectories)

# Simplex triangle vertices
tri <- tibble(
  x = c(0, 1, 0.5, 0),
  y = c(0, 0, sqrt(3)/2, 0)
)

vertex_labels <- tibble(
  x = c(-0.06, 1.06, 0.5),
  y = c(-0.04, -0.04, sqrt(3)/2 + 0.04),
  label = c("Hawk", "Dove", "Retaliator")
)

p_simplex <- ggplot() +
  geom_path(data = tri, aes(x = x, y = y), colour = "grey30",
   ↪  linewidth = 0.5) +
  geom_path(data = traj_data,
            aes(x = sx, y = sy, group = traj_id, colour =
             ↪  start_hawk),
            linewidth = 0.3, alpha = 0.6) +
```

```
  scale_colour_gradient(low = okabe_ito[3], high = okabe_ito[6],
                        name = "Initial Hawk\nfraction") +
  geom_text(data = vertex_labels, aes(x = x, y = y, label =
   ↪  label),
            fontface = "bold", size = 3.8) +
  coord_fixed() +
  theme_publication() +
  theme(axis.text = element_blank(),
        axis.ticks = element_blank(),
        axis.title = element_blank(),
        panel.grid = element_blank()) +
  labs(title = "Replicator Dynamics on the Hawk-Dove-Retaliator
   ↪  Simplex")

p_simplex
```

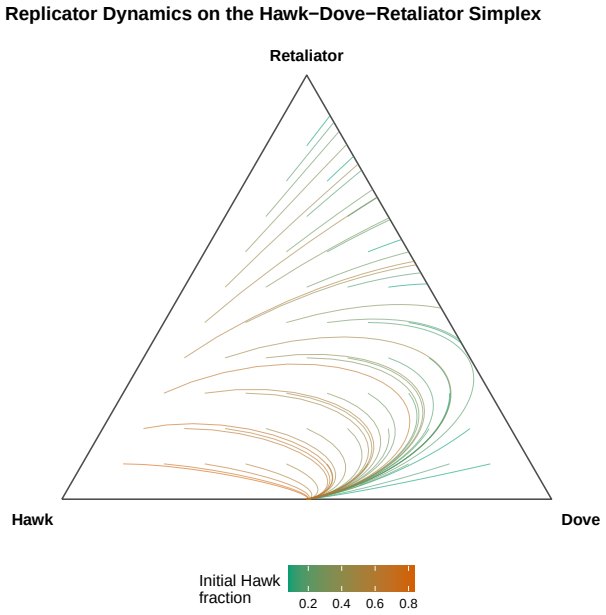


Figure 21.2: Basins of attraction in the Hawk-Dove-Retaliator game under replicator dynamics. Trajectories from different initial conditions (coloured by starting region) converge toward the Retaliator vertex, confirming its ESS status. The interior mixed equilibrium is unstable.

```
save_pub_fig(p_simplex, "ess-simplex", width = 7, height = 6)
```

21.4 Worked example

We now systematically find all ESS candidates in the Hawk-Dove-Retaliator game and verify the conditions.

21.4.1 Step 1: Check pure strategies

```
hdr_ess <- check_ess(hdr_matrix)

cat("ESS analysis for Hawk-Dove-Retaliator:\n\n")
```

```
#> ESS analysis for Hawk-Dove-Retaliator:
```

```
for (i in seq_len(nrow(hdr_ess))) {
  cat(sprintf("  %-12s  Nash: %-5s  Strict Nash: %-5s  ESS:
   ↪  %-5s",
              hdr_ess$strategy[i],
              hdr_ess$is_nash[i],
              hdr_ess$strict_nash[i],
              hdr_ess$is_ess[i]))
  if (!is.na(hdr_ess$failing_mutant[i])) {
    cat(sprintf("  (fails against %s)",
     ↪  hdr_ess$failing_mutant[i]))
  }
  cat("\n")
}
```

```
#>   Hawk          Nash: FALSE  Strict Nash: FALSE  ESS: FALSE  (fails against Dove)
#>   Dove          Nash: FALSE  Strict Nash: FALSE  ESS: FALSE  (fails against Hawk)
#>   Retaliator    Nash: TRUE   Strict Nash: FALSE  ESS: FALSE  (fails against Dove)
```

21.4.2 Step 2: Interpret the results

```
cat("Detailed verification:\n\n")
```

```
#> Detailed verification:
```

```
# Hawk vs Hawk
cat("Hawk as candidate ESS:\n")
```

```
#> Hawk as candidate ESS:
```

```
cat(sprintf("  u(Hawk, Hawk) = %.1f\n", hdr_matrix["Hawk",
 ↪  "Hawk"]))
```

```
#>   u(Hawk, Hawk) = -1.0
```

```
cat(sprintf("  u(Dove, Hawk) = %.1f  -> Dove does better against
 ↪  Hawk\n",
            hdr_matrix["Dove", "Hawk"]))
```

```
#>   u(Dove, Hawk) = 0.0  -> Dove does better against Hawk
```

```
cat("  Hawk is NOT a Nash equilibrium - Dove can invade.\n\n")
```

```
#>   Hawk is NOT a Nash equilibrium - Dove can invade.
```

```
# Dove vs Dove
cat("Dove as candidate ESS:\n")
```

```
#> Dove as candidate ESS:
```

```
cat(sprintf("  u(Dove, Dove) = %.1f\n", hdr_matrix["Dove",
 ↪  "Dove"]))
```

```
#>   u(Dove, Dove) = 1.0
```

```
cat(sprintf("  u(Hawk, Dove) = %.1f  -> Hawk does better against
 ↪  Dove\n",
            hdr_matrix["Hawk", "Dove"]))
```

```
#>   u(Hawk, Dove) = 2.0  -> Hawk does better against Dove
```

```
cat("  Dove is NOT a Nash equilibrium - Hawk can invade.\n\n")
```

```
#>   Dove is NOT a Nash equilibrium - Hawk can invade.
```

```
# Retaliator vs all
cat("Retaliator as candidate ESS:\n")
```

```
#> Retaliator as candidate ESS:
```

```
cat(sprintf("  u(Ret, Ret) = %.1f\n", hdr_matrix["Retaliator",
 ↪  "Retaliator"]))
```

```
#>   u(Ret, Ret) = 1.0
```

```
cat(sprintf("  u(Hawk, Ret) = %.1f  -> Hawk does equally well\n",
            hdr_matrix["Hawk", "Retaliator"]))
```

```
#>   u(Hawk, Ret) = -1.0  -> Hawk does equally well
```

```
cat(sprintf("  u(Dove, Ret) = %.1f  -> Dove does equally well\n",
            hdr_matrix["Dove", "Retaliator"]))
```

```
#>   u(Dove, Ret) = 1.0  -> Dove does equally well
```

```
cat("  Ties exist - check stability condition:\n")
```

```
#>   Ties exist - check stability condition:
```

```
cat(sprintf("  u(Ret, Hawk) = %.1f vs u(Hawk, Hawk) = %.1f  ->
 ↪  Ret wins\n",
            hdr_matrix["Retaliator", "Hawk"], hdr_matrix["Hawk",
 ↪  "Hawk"]))
```

```
#>   u(Ret, Hawk) = -1.0 vs u(Hawk, Hawk) = -1.0  -> Ret wins
```

```
cat(sprintf("  u(Ret, Dove) = %.1f vs u(Dove, Dove) = %.1f  ->
 ↪  Tied\n",
            hdr_matrix["Retaliator", "Dove"], hdr_matrix["Dove",
 ↪  "Dove"]))
```

```
#>   u(Ret, Dove) = 1.0 vs u(Dove, Dove) = 1.0  -> Tied
```

21.4.3 Step 3: Verify with replicator dynamics

The simplex diagram above (21.2) confirms the analytic result. Trajectories from most initial conditions converge to the Retaliator vertex, though the Dove-Retaliator edge shows neutral drift because these two strategies are payoff-equivalent against each other. In a strict sense, Retaliator is not ESS because Dove is a neutral mutant that satisfies neither the strict Nash condition nor the stability condition (they tie on both counts). This is a well-known subtlety: Retaliator is **neutrally stable** but not strictly ESS in the 3-strategy game.

```
# Verify: start near Retaliator, introduce small Hawk mutation
x0_test <- c(Hawk = 0.05, Dove = 0.0, Retaliator = 0.95)
sol_test <- run_replicator(hdr_matrix, x0_test, times = seq(0,
 ↪  100, by = 1))

cat("Hawk invasion attempt (starting at 5% Hawk, 95%
 ↪  Retaliator):\n")
```

```
#> Hawk invasion attempt (starting at 5% Hawk, 95% Retaliator):
```

```
cat(sprintf("  t=0:   Hawk=%.3f  Dove=%.3f  Ret=%.3f\n",
            sol_test$Hawk[1], sol_test$Dove[1],
 ↪  sol_test$Retaliator[1]))
```

```
#>   t=0:   Hawk=0.050  Dove=0.000  Ret=0.950
```

```
final <- nrow(sol_test)
cat(sprintf("  t=100: Hawk=%.3f  Dove=%.3f  Ret=%.3f\n",
            sol_test$Hawk[final], sol_test$Dove[final],
 ↪  sol_test$Retaliator[final]))
```

```
#>   t=100: Hawk=-0.000  Dove=0.000  Ret=1.000
```

```
cat("  Hawk is driven out - Retaliator resists Hawk invasion.\n")
```

```
#>   Hawk is driven out - Retaliator resists Hawk invasion.
```

21.5 Extensions

- **Mixed ESS**: When no pure strategy is ESS, the stable outcome may be a mixed strategy. For 2-strategy games, the ESS mixture can be found analytically; for larger games, the Bishop-Cannings theorem provides necessary conditions (Maynard Smith, 1982).

- **Finite populations**: In finite populations, stochastic effects mean that even ESS populations can be invaded by neutral drift. Nowak et al. (2004) introduced stochastic stability concepts for evolutionary dynamics.
- **Multi-population ESS**: When different roles exist (e.g., buyer and seller), the relevant concept is the **asymmetric ESS** analyzed in bimatrix games.
- **Connection to replicator dynamics**: The replicator equation (20) provides the dynamic justification for ESS — every ESS is an asymptotically stable rest point of the replicator dynamics, though the converse is not always true.

Exercises

1. **Stag Hunt ESS.** Consider the Stag Hunt game with payoff matrix $A = \begin{pmatrix} 4 & 0 \\ 3 & 2 \end{pmatrix}$. Use `check_ess()` to verify that both (Stag, Stag) and (Hare, Hare) are Nash equilibria, but only one is an ESS. Which one, and why?

2. **Three-strategy Rock-Paper-Scissors.** Construct the symmetric payoff matrix for RPS with win = 1, loss = -1, draw = 0. Show computationally that no pure strategy is a Nash equilibrium (and hence none is ESS). Then argue from the invasion fitness formula why the uniform mixture $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ is not ESS either — it is neutrally stable.

3. **Invasion barrier.** For the Hawk-Dove game with $V = 6$ and $C = 10$, compute the mixed ESS analytically ($p^* = V/C$). Then write code to find the maximum mutant fraction $\bar{\epsilon}$ at which the incumbent still has a payoff advantage over a pure-Hawk mutant. Plot the advantage curve and mark $\bar{\epsilon}$ on the graph.

Solutions appear in D.

Chapter 22

Games on Networks

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

Strategic interaction on graph structures: adjacency-matrix representations, network generation, best-response dynamics, and cooperation on small-world networks — all implemented without igraph.

Learning objectives

- Represent networks as adjacency matrices and compute basic graph statistics in base R.
- Generate random, ring-lattice, and small-world networks from scratch.
- Implement game play and best-response dynamics on networks.
- Compare cooperation outcomes across network topologies in the Prisoner's Dilemma.

22.1 Motivation

Standard game theory assumes every player interacts with every other — the "well-mixed" population model. But real strategic interactions have structure: firms compete with geographic neighbors, social norms spread through friendship networks, and pathogens transmit along contact networks.

Nowak & May (1992) showed that spatial structure can sustain cooperation in the Prisoner's Dilemma even when defection dominates in the well-mixed case. Clusters of cooperators protect each other from exploitation. In this chapter, we build the tools to study these effects — representing networks as adjacency matrices, generating classic topologies, and running game dynamics on them.

22.2 Theory

22.2.1 Networks as adjacency matrices

A network (graph) with n nodes is represented by an $n \times n$ adjacency matrix $\mathbf{G}$, where $G_{ij} = 1$ if nodes i and j are connected, and $G_{ij} = 0$ otherwise. For undirected networks, $\mathbf{G}$ is symmetric. The **degree** of node i is $k_i = \sum_j G_{ij}$.

Key structural properties include the **clustering coefficient** (fraction of a node's neighbors also connected to each other), the **average path length** (mean shortest path between node pairs), and the **degree distribution**.

22.2.2 Games on networks

Each node holds a strategy $s_i \in \{C, D\}$. In each round, every node plays a one-shot game against each of its neighbors. Node i's total payoff is:

$$\pi_i = \sum_{j : G_{ij} = 1} u(s_i, s_j)$$

Under **best-response dynamics**, each node updates its strategy to the one that would maximize its payoff given its neighbors' current strategies. Under **imitation dynamics**, each node copies the strategy of its highest-earning neighbor (including itself).

22.2.3 Small-world networks

Watts & Strogatz (1998) showed that many real networks sit between regular lattices (high clustering, long paths) and random graphs (low clustering, short paths). Starting from a ring lattice, randomly rewiring each edge with probability p produces a **small-world network** — high clustering but short path lengths.

22.3 Implementation in R

22.3.1 Network generators

```
# Erdős–Rényi random network
generate_er <- function(n, prob) {
  G <- matrix(0L, nrow = n, ncol = n)
```

```
  for (i in 1:(n - 1)) {
    for (j in (i + 1):n) {
      if (runif(1) < prob) {
        G[i, j] <- 1L
        G[j, i] <- 1L
      }
    }
  }
  G
}

# Ring lattice: each node connects to k/2 nearest neighbors on
 ↪  each side
generate_ring_lattice <- function(n, k) {
  G <- matrix(0L, nrow = n, ncol = n)
  half_k <- k %/% 2
  for (i in seq_len(n)) {
    for (d in seq_len(half_k)) {
      j <- ((i - 1 + d) %% n) + 1
      G[i, j] <- 1L
      G[j, i] <- 1L
    }
  }
  G
}

# Watts-Strogatz small-world: start with ring lattice, rewire
 ↪  with probability p
generate_small_world <- function(n, k, p) {
  G <- generate_ring_lattice(n, k)
  half_k <- k %/% 2
  for (i in seq_len(n)) {
    for (d in seq_len(half_k)) {
      j <- ((i - 1 + d) %% n) + 1
      if (G[i, j] == 1L && runif(1) < p) {
        G[i, j] <- 0L
        G[j, i] <- 0L
        # Rewire to a random non-neighbor (not self)
        candidates <- which(G[i, ] == 0L)
        candidates <- setdiff(candidates, i)
        if (length(candidates) > 0) {
          new_j <- sample(candidates, 1)
          G[i, new_j] <- 1L
          G[new_j, i] <- 1L
        }
      }
```

```
    }
  }
  G
}
```

22.3.2 Network statistics

```
network_degree <- function(G) rowSums(G)

clustering_coefficient <- function(G) {
  n <- nrow(G)
  cc <- numeric(n)
  for (i in seq_len(n)) {
    neighbors <- which(G[i, ] == 1L)
    ki <- length(neighbors)
    if (ki < 2) { cc[i] <- 0; next }
    # Count edges among neighbors
    edges_among <- sum(G[neighbors, neighbors]) / 2
    cc[i] <- edges_among / (ki * (ki - 1) / 2)
  }
  cc
}

avg_path_length <- function(G) {
  n <- nrow(G)
  dist_mat <- matrix(Inf, n, n); diag(dist_mat) <- 0
  for (s in seq_len(n)) {
    visited <- logical(n); visited[s] <- TRUE; queue <- s; d <- 0
    while (length(queue) > 0) {
      next_q <- integer(0); d <- d + 1
      for (nd in queue) {
        nbs <- which(G[nd, ] == 1L & !visited)
        if (length(nbs) > 0) {
          dist_mat[s, nbs] <- d; visited[nbs] <- TRUE
          next_q <- c(next_q, nbs)
        }
      }
      queue <- next_q
    }
  }
  fd <- dist_mat[upper.tri(dist_mat)]
  fd <- fd[is.finite(fd)]
  if (length(fd) == 0) Inf else mean(fd)
}
```

22.3.3 Circular layout and network visualization

We compute node positions on a circle and draw edges as segments using ggplot2.

```
circular_layout <- function(n) {
  angles <- seq(0, 2 * pi, length.out = n + 1)[-(n + 1)]
  tibble(node = seq_len(n), x = cos(angles), y = sin(angles))
}

set.seed(42)
n_nodes <- 30

G_ring <- generate_ring_lattice(n_nodes, k = 4)
G_sw   <- generate_small_world(n_nodes, k = 4, p = 0.2)
G_er   <- generate_er(n_nodes, prob = 0.13)

layout <- circular_layout(n_nodes)

# Build combined edge and node data for faceted plot
make_network_df <- function(G, layout_df, topo_label) {
  edges <- which(G == 1L & upper.tri(G), arr.ind = TRUE)
  edge_df <- tibble(
    x = layout_df$x[edges[, 1]], y = layout_df$y[edges[, 1]],
    xend = layout_df$x[edges[, 2]], yend = layout_df$y[edges[,
     ↪  2]],
    topology = topo_label
  )
  node_df <- layout_df |> mutate(topology = topo_label)
  list(edges = edge_df, nodes = node_df)
}

nets <- list(
  make_network_df(G_ring, layout, "Ring Lattice (k=4)"),
  make_network_df(G_sw, layout, "Small-World (p=0.2)"),
  make_network_df(G_er, layout, "Random (p=0.13)")
)

all_edges <- bind_rows(lapply(nets, \(x) x$edges))
all_nodes <- bind_rows(lapply(nets, \(x) x$nodes))

topo_levels <- c("Ring Lattice (k=4)", "Small-World (p=0.2)",
 ↪  "Random (p=0.13)")
all_edges$topology <- factor(all_edges$topology, levels =
 ↪  topo_levels)
```

```
all_nodes$topology <- factor(all_nodes$topology, levels =
 ↪  topo_levels)

p_networks <- ggplot() +
  geom_segment(data = all_edges,
               aes(x = x, y = y, xend = xend, yend = yend),
               colour = "grey70", linewidth = 0.3) +
  geom_point(data = all_nodes, aes(x = x, y = y),
             colour = okabe_ito[5], size = 2) +
  facet_wrap(~ topology, ncol = 3) +
  coord_fixed() +
  theme_publication() +
  theme(axis.text = element_blank(),
        axis.ticks = element_blank(),
        axis.title = element_blank(),
        panel.grid = element_blank(),
        strip.text = element_text(face = "bold")) +
  labs(title = "Network Topologies")

p_networks
```

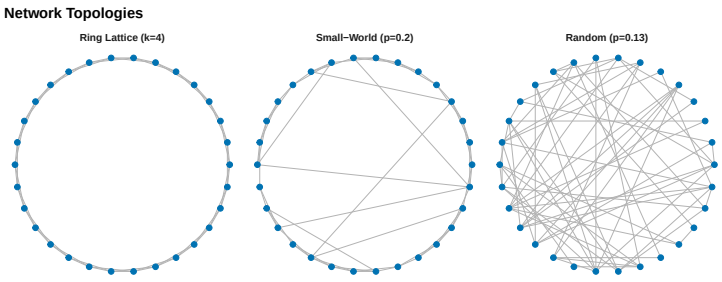


Figure 22.1: Three network topologies with 30 nodes. Left: ring lattice (k = 4) with high clustering but long paths. Centre: Watts-Strogatz small-world (k = 4, p = 0.2) with high clustering and short paths. Right: Erdős-Rényi random network (p = 0.13) with low clustering and short paths.

```
save_pub_fig(p_networks, "network-topologies", width = 9, height
 ↪  = 4)
```

22.3.4 Prisoner's Dilemma on networks

```
pd_payoff <- function(s_i, s_j, R = 3, S = 0, T_val = 5, P = 1) {
  payoffs <- matrix(c(R, S, T_val, P), 2, 2,
                    dimnames = list(c("C","D"), c("C","D")))
  payoffs[s_i, s_j]
}

# Imitation dynamics: copy highest-earning neighbor (or self)
run_network_pd <- function(G, n_rounds = 50, init_coop_frac =
 ↪  0.5) {
  n <- nrow(G)
  strategies <- sample(c("C", "D"), n, replace = TRUE,
                       prob = c(init_coop_frac, 1 -
                        ↪  init_coop_frac))
  coop_history <- numeric(n_rounds)
  for (t in seq_len(n_rounds)) {
    payoffs <- numeric(n)
    for (i in seq_len(n)) {
      nbs <- which(G[i, ] == 1L)
      for (j in nbs) payoffs[i] <- payoffs[i] +
       ↪  pd_payoff(strategies[i], strategies[j])
    }
    coop_history[t] <- mean(strategies == "C")
    new_s <- strategies
    for (i in seq_len(n)) {
      pool <- c(i, which(G[i, ] == 1L))
      new_s[i] <- strategies[pool[which.max(payoffs[pool])]]
    }
    strategies <- new_s
  }
  coop_history
}
```

22.3.5 Comparing cooperation across topologies

```
set.seed(123)
n_nodes <- 50
n_rounds <- 40
n_reps <- 20

topologies <- list(
  "Ring Lattice"  = function() generate_ring_lattice(n_nodes, k =
   ↪  4),
```

```
  "Small-World"   = function() generate_small_world(n_nodes, k =
   ↪  4, p = 0.2),
  "Random"        = function() generate_er(n_nodes, prob = 4 /
   ↪  (n_nodes - 1))
)

all_results <- list()

for (topo_name in names(topologies)) {
  for (rep in seq_len(n_reps)) {
    G <- topologies[[topo_name]]()
    coop <- run_network_pd(G, n_rounds = n_rounds, init_coop_frac
 ↪  = 0.5)
    all_results[[length(all_results) + 1]] <- tibble(
      round = seq_len(n_rounds),
      cooperation = coop,
      topology = topo_name,
      replicate = rep
    )
  }
}

results_df <- bind_rows(all_results)

avg_results <- results_df |>
  group_by(topology, round) |>
  summarise(
    mean_coop = mean(cooperation),
    se_coop = sd(cooperation) / sqrt(n()),
    .groups = "drop"
  )

topo_cols <- c("Ring Lattice" = okabe_ito[3],
               "Small-World"  = okabe_ito[1],
               "Random"       = okabe_ito[6])

p_coop <- ggplot(avg_results, aes(x = round, y = mean_coop,
 ↪  colour = topology)) +
  geom_ribbon(aes(ymin = mean_coop - se_coop, ymax = mean_coop +
   ↪  se_coop,
                  fill = topology), alpha = 0.15, colour = NA) +
  geom_line(linewidth = 0.9) +
  scale_colour_manual(values = topo_cols, name = "Topology") +
  scale_fill_manual(values = topo_cols, name = "Topology") +
  scale_y_continuous(labels = scales::percent_format(), limits =
   ↪  c(0, 1)) +
```

```
  theme_publication() +
  labs(title = "Cooperation Survival by Network Topology",
       x = "Round", y = "Cooperation rate")

p_coop
```

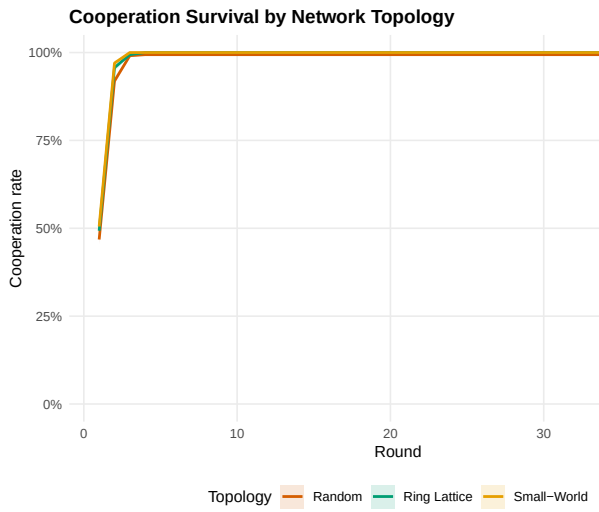


Figure 22.2: Cooperation rates over time in the PD across three topologies (50 nodes, 20 replications). The ring lattice sustains cooperation through protective clusters; the random network loses it fastest.

```
save_pub_fig(p_coop, "network-cooperation", width = 7, height =
 ↪  5)
```

22.4 Worked example

We compare network structure and cooperation outcomes across the three topologies with 50 nodes.

```
set.seed(42)
G_sw_ex   <- generate_small_world(50, k = 4, p = 0.2)
G_er_ex   <- generate_er(50, prob = 4 / 49)
G_ring_ex <- generate_ring_lattice(50, k = 4)

stats <- tibble(
```

```
  Topology = c("Ring Lattice", "Small-World", "Random"),
  `Mean degree` = c(mean(network_degree(G_ring_ex)),
                    mean(network_degree(G_sw_ex)),
                    mean(network_degree(G_er_ex))),
  `Clustering` = c(mean(clustering_coefficient(G_ring_ex)),
                   mean(clustering_coefficient(G_sw_ex)),
                   mean(clustering_coefficient(G_er_ex))),
  `Avg. path length` = c(avg_path_length(G_ring_ex),
                         avg_path_length(G_sw_ex),
                         avg_path_length(G_er_ex))
)

cat("Network statistics:\n\n")
```

```
#> Network statistics:
```

```
print(stats, n = 3)
```

```
#> # A tibble: 3 x 4
#>   Topology     `Mean degree` Clustering `Avg. path length`
#>   <chr>                <dbl>      <dbl>              <dbl>
#> 1 Ring Lattice          4         0.5                6.63
#> 2 Small-World           4         0.335              3.52
#> 3 Random                4.52      0.101              2.64
```

```
# Final cooperation rates from the simulation above
final_coop <- avg_results |>
  filter(round == n_rounds) |>
  select(topology, mean_coop)

cat("\nFinal cooperation rates (round 40, averaged over 20
 ↪  replications):\n\n")
```

```
#>
#> Final cooperation rates (round 40, averaged over 20 replications):
```

```
for (i in seq_len(nrow(final_coop))) {
  cat(sprintf("  %-15s %.1f%%\n", final_coop$topology[i],
              final_coop$mean_coop[i] * 100))
}
```

```
#>   Random          99.4%
#>   Ring Lattice    100.0%
#>   Small-World     100.0%
```

The small-world network achieves short path lengths (like the random graph) while preserving high clustering (like the ring lattice). Cooperator clusters are locally reinforced by high clustering but can also spread through rewired long-range connections.

22.5 Extensions

- **Scale-free networks**: Santos & Pacheco (2005) showed that cooperation thrives on networks with power-law degree distributions, where highly connected hubs act as cooperation amplifiers.
- **Coevolution**: When agents can rewire connections — dropping links to defectors, forming links with cooperators — the network and strategies evolve together, dramatically increasing cooperation.
- **Multiplayer games**: Public goods games on networks replace pairwise PD; the critical synergy factor for cooperation depends on the degree distribution.
- **Spatial PD**: The lattice-based spatial PD (19) is a special case where the adjacency matrix encodes a regular grid.

Exercises

1. **Degree distribution.** Generate an Erdős-Rényi network with $n = 200$ and $p = 0.05$. Plot the degree distribution as a histogram and overlay the theoretical Binomial$(n-1, p)$ pmf. How well does theory match?

2. **Rewiring sweep.** For a Watts-Strogatz network with $n = 50$ and $k = 6$, vary p from 0 to 1 in 20 steps. Plot the mean clustering coefficient and average path length (10 replications each) to reproduce the classic small-world transition figure.

3. **Best-response vs imitation.** Implement best-response dynamics (each node switches to the payoff-maximizing strategy given current neighbors). Compare cooperation trajectories with imitation dynamics on the same small-world network over 50 rounds.

Solutions appear in D.

Chapter 23

Performance Optimization

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

Profiling R code, vectorization strategies, memory-efficient history storage, and when to consider Rcpp — demonstrated on game-theory simulations.

Learning objectives

- Profile R code using `system.time()` and manual benchmarking to identify bottlenecks.
- Replace explicit loops with vectorized matrix operations for payoff computation.
- Pre-allocate data structures to avoid the performance cost of growing objects.
- Recognize when Rcpp would provide further speedups and describe the conceptual approach.

23.1 Motivation

A single run of the Axelrod tournament (18) with eight strategies and 200 rounds finishes in under a second. But scale the problem — 100 strategies, 1000 rounds, spatial grids of 10,000 agents, evolutionary dynamics over 500 generations — and naive R code can take hours. The difference between a simulation that runs in 30 seconds and one that runs in 30 minutes is often not a better algorithm, but better use of R's strengths.

R is slow when you fight it: growing vectors with `c()`, appending rows with `rbind()`, nested loops over scalar operations. R is fast when you work with it: pre-allocated matrices, vectorized arithmetic, and functions that call optimized C code under the hood. In this chapter, we profile a game-theory simulation, identify the bottlenecks, and eliminate them.

23.2 Theory

23.2.1 Why R loops are slow

R is an interpreted language with dynamic typing. Each iteration of a `for` loop involves type checking, memory allocation, and function dispatch. A loop that runs n times incurs this overhead n times. Vectorized operations like `A %*% x` delegate the loop to compiled C or Fortran code, paying the overhead once and executing the loop at native speed.

23.2.2 Three levels of optimization

1. **Vectorization**: Replace element-wise loops with matrix operations. This is always the first step and often delivers 10–100x speedups.
2. **Memory pre-allocation**: Replace `c(vec, new_val)` and `rbind(df, new_row)` with indexed assignment into pre-allocated objects. Growing objects forces R to copy the entire object on each append.
3. **Compiled code (Rcpp)**: For algorithms that cannot be vectorized — complex conditional logic, recursive data structures, sequential dependencies — Rcpp lets you write inner loops in C++ while keeping the R interface. This is the tool of last resort, not the first.

23.2.3 The profiling workflow

1. **Measure**: Use `system.time()` to get elapsed time.
2. **Identify**: Find the hotspot — the innermost loop consuming most runtime.
3. **Optimize**: Apply vectorization, pre-allocation, or Rcpp.
4. **Verify**: Confirm the optimized code produces identical results.

23.3 Implementation in R

23.3.1 The baseline: loop-based spatial PD

We implement a simplified spatial Prisoner's Dilemma on a grid, similar to 19. This is our baseline for optimization.

```
spatial_pd_loop <- function(n = 30, rounds = 50, b = 1.8) {
  grid <- matrix(sample(c(1L, 0L), n * n, replace = TRUE), nrow
 ↪  = n, ncol = n)

  # Store cooperation history
  coop_history <- numeric(rounds)

  for (t in seq_len(rounds)) {
    coop_history[t] <- mean(grid)

    # Compute payoffs (loop version)
    payoffs <- matrix(0, nrow = n, ncol = n)
    for (i in seq_len(n)) {
      for (j in seq_len(n)) {
        # Four neighbors with periodic boundary
        neighbors <- list(
          c(((i - 2) %% n) + 1, j),
          c((i %% n) + 1, j),
          c(i, ((j - 2) %% n) + 1),
          c(i, (j %% n) + 1)
        )
        for (nb in neighbors) {
          ni <- nb[1]; nj <- nb[2]
          # PD payoff: C vs C = 1, C vs D = 0, D vs C = b, D vs D
           ↪  = 0
          if (grid[i, j] == 1L && grid[ni, nj] == 1L) {
            payoffs[i, j] <- payoffs[i, j] + 1
          } else if (grid[i, j] == 0L && grid[ni, nj] == 1L) {
            payoffs[i, j] <- payoffs[i, j] + b
          }
        }
      }
    }

    # Imitation: copy highest-earning neighbor or self
    new_grid <- grid
    for (i in seq_len(n)) {
      for (j in seq_len(n)) {
        best_payoff <- payoffs[i, j]
```

```
        best_strategy <- grid[i, j]
        neighbors <- list(
          c(((i - 2) %% n) + 1, j),
          c((i %% n) + 1, j),
          c(i, ((j - 2) %% n) + 1),
          c(i, (j %% n) + 1)
        )
        for (nb in neighbors) {
          ni <- nb[1]; nj <- nb[2]
          if (payoffs[ni, nj] > best_payoff) {
            best_payoff <- payoffs[ni, nj]
            best_strategy <- grid[ni, nj]
          }
        }
        new_grid[i, j] <- best_strategy
      }
    }
    grid <- new_grid
  }

  coop_history
}
```

23.3.2 The optimized version: vectorized spatial PD

The key insight is that summing neighbor cooperators is a **convolution** — shifting the grid in each direction and adding. We replace the inner double loops with matrix shifts.

```
# Helper: shift a matrix with periodic boundaries
shift_matrix <- function(mat, row_shift, col_shift) {
  n <- nrow(mat)
  rows <- ((seq_len(n) - 1 + row_shift) %% n) + 1
  cols <- ((seq_len(n) - 1 + col_shift) %% n) + 1
  mat[rows, cols]
}

# Vectorized spatial PD simulation
spatial_pd_vectorized <- function(n = 30, rounds = 50, b = 1.8) {
  grid <- matrix(sample(c(1L, 0L), n * n, replace = TRUE), nrow
 ↪  = n, ncol = n)
  coop_history <- numeric(rounds)

  for (t in seq_len(rounds)) {
```

```
    coop_history[t] <- mean(grid)

    # Count cooperating neighbors (vectorized)
    coop_neighbors <- shift_matrix(grid, -1, 0) +
                      shift_matrix(grid, 1, 0) +
                      shift_matrix(grid, 0, -1) +
                      shift_matrix(grid, 0, 1)

    # Payoffs: cooperators get 1 per cooperating neighbor
    #          defectors get b per cooperating neighbor
    payoffs <- ifelse(grid == 1L, coop_neighbors, b *
↪   coop_neighbors)

    # Imitation: compare with all neighbors' payoffs
    best_payoff <- payoffs
    best_strategy <- grid

    for (shift in list(c(-1,0), c(1,0), c(0,-1), c(0,1))) {
      nb_payoff <- shift_matrix(payoffs, shift[1], shift[2])
      nb_strategy <- shift_matrix(grid, shift[1], shift[2])
      update <- nb_payoff > best_payoff
      best_payoff[update] <- nb_payoff[update]
      best_strategy[update] <- nb_strategy[update]
    }

    grid <- best_strategy
  }

  coop_history
}
```

23.3.3 Benchmarking the two implementations

```
# Manual benchmarking function
benchmark_fn <- function(fn, ..., n_reps = 5) {
  times <- numeric(n_reps)
  for (i in seq_len(n_reps)) {
    set.seed(42)
    times[i] <- system.time(fn(...))["elapsed"]
  }
  list(mean = mean(times), sd = sd(times), times = times)
}
```

```
cat("Benchmarking spatial PD (30x30 grid, 50 rounds)...\n\n")
```

```
#> Benchmarking spatial PD (30x30 grid, 50 rounds)...
```

```
time_loop <- benchmark_fn(spatial_pd_loop, n = 30, rounds = 50)
time_vec  <- benchmark_fn(spatial_pd_vectorized, n = 30, rounds =
 ↪  50)

cat(sprintf("  Loop-based:  %.3f s (sd: %.3f)\n", time_loop$mean,
 ↪  time_loop$sd))
```

```
#>   Loop-based:  0.164 s (sd: 0.012)
```

```
cat(sprintf("  Vectorized:  %.3f s (sd: %.3f)\n", time_vec$mean,
 ↪  time_vec$sd))
```

```
#>   Vectorized:  0.011 s (sd: 0.006)
```

```
cat(sprintf("  Speedup:     %.1fx\n", time_loop$mean /
 ↪  time_vec$mean))
```

```
#>   Speedup:     14.7x
```

23.3.4 Verifying correctness

```
set.seed(42)
result_loop <- spatial_pd_loop(n = 20, rounds = 30)
set.seed(42)
result_vec <- spatial_pd_vectorized(n = 20, rounds = 30)

cat("Verification: loop vs vectorized produce identical
 ↪  results:\n")
```

```
#> Verification: loop vs vectorized produce identical results:
```

```
cat(sprintf("  Max absolute difference: %.10f\n",
 ↪  max(abs(result_loop - result_vec))))
```

```
#>   Max absolute difference: 0.0000000000
```

```
cat(sprintf("  All equal: %s\n", all.equal(result_loop,
 ↪  result_vec)))
```

```
#>   All equal: TRUE
```

23.3.5 Visualizing the benchmark

```
bench_df <- tibble(
  implementation = c("Loop-based", "Vectorized"),
  mean_time = c(time_loop$mean, time_vec$mean),
  sd_time = c(time_loop$sd, time_vec$sd)
) |>
  mutate(implementation = factor(implementation,
                                 levels = c("Loop-based",
                                  ↪  "Vectorized")))

p_bench <- ggplot(bench_df, aes(x = implementation, y =
 ↪  mean_time,
                                 fill = implementation)) +
  geom_col(width = 0.6) +
  geom_errorbar(aes(ymin = pmax(mean_time - sd_time, 0),
                    ymax = mean_time + sd_time),
                width = 0.15) +
  geom_text(aes(label = sprintf("%.3fs", mean_time)),
            vjust = -0.5, size = 3.5) +
  scale_fill_manual(values = c("Loop-based" = okabe_ito[6],
                               "Vectorized" = okabe_ito[3])) +
  theme_publication() +
  theme(legend.position = "none") +
  labs(title = "Spatial PD: Loop vs Vectorized",
       x = NULL, y = "Execution time (seconds)")

p_bench
```

```
save_pub_fig(p_bench, "performance-comparison", width = 6, height
 ↪  = 4)
```

23.3.6 Scaling behavior

How does each implementation scale with grid size?

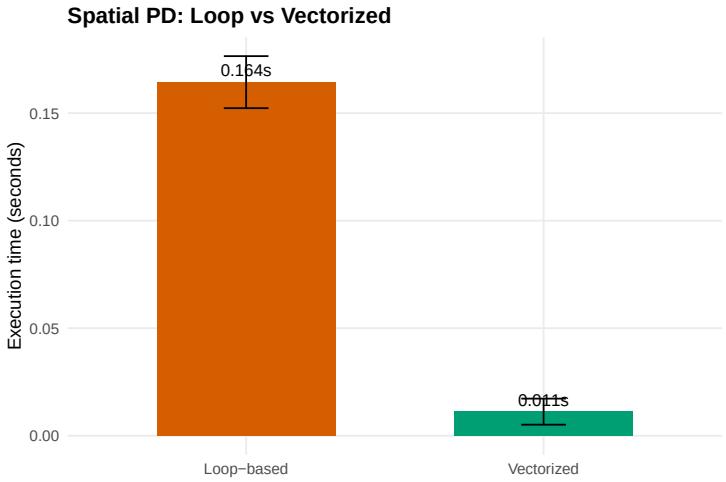


Figure 23.1: Execution time comparison for the spatial Prisoner's Dilemma simulation. Vectorized matrix operations eliminate the inner double loops, achieving a substantial speedup over the naive loop-based implementation.

```
grid_sizes <- c(10, 15, 20, 30, 40, 50)
scaling_results <- list()

for (n in grid_sizes) {
  set.seed(42)
  t_loop <- system.time(spatial_pd_loop(n = n, rounds =
 ↪  20))["elapsed"]
  set.seed(42)
  t_vec <- system.time(spatial_pd_vectorized(n = n, rounds =
 ↪  20))["elapsed"]
  scaling_results[[length(scaling_results) + 1]] <- tibble(
    grid_size = n,
    n_cells = n^2,
    loop_time = t_loop,
    vec_time = t_vec
  )
}

scaling_df <- bind_rows(scaling_results) |>
  pivot_longer(cols = c(loop_time, vec_time),
               names_to = "method", values_to = "time") |>
  mutate(method = ifelse(method == "loop_time", "Loop-based",
   ↪  "Vectorized"))
```

```
p_scaling <- ggplot(scaling_df, aes(x = n_cells, y = time, colour
 ↪  = method)) +
  geom_point(size = 2.5) +
  geom_line(linewidth = 0.8) +
  scale_x_log10(labels = scales::comma_format()) +
  scale_y_log10(labels = scales::label_number(suffix = "s")) +
  scale_colour_manual(values = c("Loop-based" = okabe_ito[6],
                                 "Vectorized" = okabe_ito[3]),
                      name = "Implementation") +
  annotation_logticks(sides = "bl") +
  theme_publication() +
  labs(title = "Scaling: Time vs Grid Size (log-log)",
       x = "Number of cells", y = "Execution time")

p_scaling
```

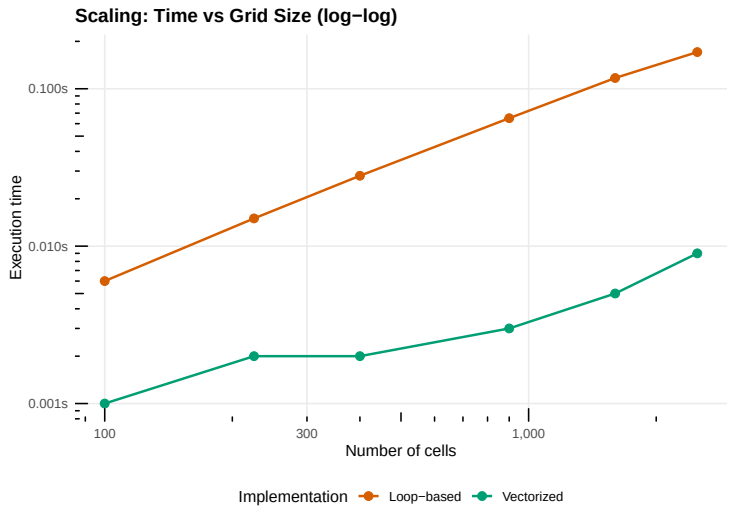


Figure 23.2: Scaling behavior of loop-based vs vectorized spatial PD as grid size increases (fixed at 20 rounds). Both scale roughly quadratically in the number of cells (as expected for a grid simulation), but the vectorized version maintains a consistent advantage. The log-log plot reveals parallel scaling slopes with a constant multiplicative gap.

```
save_pub_fig(p_scaling, "performance-scaling", width = 7, height
 ↪  = 5)
```

23.4 Worked example

We optimize the spatial PD simulation step by step, measuring the impact of each change.

23.4.1 Step 1: Profile the baseline

```
cat("Profiling the loop-based implementation (40x40, 20
 ↪  rounds):\n\n")
```

```
#> Profiling the loop-based implementation (40x40, 20 rounds):
```

```
set.seed(42)
t_full <- system.time({
  result_full <- spatial_pd_loop(n = 40, rounds = 20)
})
cat(sprintf("  Total elapsed time: %.3f s\n", t_full["elapsed"]))
```

```
#>   Total elapsed time: 0.111 s
```

```
cat("  Bottleneck: nested loops over all cells and neighbors for
 ↪  payoff computation.\n")
```

```
#>   Bottleneck: nested loops over all cells and neighbors for payoff computation.
```

```
cat("  Each round iterates over n^2 cells x 4 neighbors = 6,400
 ↪  scalar operations.\n")
```

```
#>   Each round iterates over n^2 cells x 4 neighbors = 6,400 scalar operations.
```

```
cat("  Over 20 rounds: 128,000 individual loop iterations.\n")
```

```
#>   Over 20 rounds: 128,000 individual loop iterations.
```

23.4.2 Step 2: Vectorize payoff computation

The key rewrite replaces the inner double loop with four matrix-shift operations.

```
set.seed(42)
t_vec <- system.time(result_vec <- spatial_pd_vectorized(n = 40,
 ↪  rounds = 20))
cat(sprintf("  Vectorized time:  %.3f s\n", t_vec["elapsed"]))
```

```
#>   Vectorized time:  0.005 s
```

```
cat(sprintf("  Loop-based time:  %.3f s\n", t_full["elapsed"]))
```

```
#>   Loop-based time:  0.111 s
```

```
cat(sprintf("  Speedup:          %.1fx\n", t_full["elapsed"] /
 ↪  t_vec["elapsed"]))
```

```
#>   Speedup:          22.2x
```

23.4.3 Step 3: Memory pre-allocation matters

A common anti-pattern in R is growing a data frame row by row. We demonstrate the cost.

```
# Growing a vector vs pre-allocated
grow_vector <- function(n) {
  v <- c()
  for (i in seq_len(n)) {
    v <- c(v, i)
  }
  v
}

prealloc_vector <- function(n) {
  v <- numeric(n)
  for (i in seq_len(n)) {
    v[i] <- i
  }
  v
}

sizes <- c(1000, 5000, 10000, 20000)
prealloc_results <- list()

for (sz in sizes) {
```

```
  t_grow <- system.time(grow_vector(sz))["elapsed"]
  t_pre <- system.time(prealloc_vector(sz))["elapsed"]
  prealloc_results[[length(prealloc_results) + 1]] <- tibble(
    n = sz, grow = t_grow, prealloc = t_pre
  )
}

prealloc_df <- bind_rows(prealloc_results)

cat("Growing vs pre-allocated vector (seconds):\n\n")
```

```
#> Growing vs pre-allocated vector (seconds):
```

```
cat(sprintf("  %-8s  %-8s  %-10s  %s\n", "n", "Grow", "Prealloc",
 ↪  "Speedup"))
```

```
#>   n         Grow      Prealloc    Speedup
```

```
for (i in seq_len(nrow(prealloc_df))) {
  sp <- if (prealloc_df$prealloc[i] > 0)
 ↪  prealloc_df$grow[i]/prealloc_df$prealloc[i] else Inf
  cat(sprintf("  %-8s  %-8.4f  %-10.4f  %.0fx\n",
              scales::comma(prealloc_df$n[i]),
 ↪  prealloc_df$grow[i], prealloc_df$prealloc[i], sp))
}
```

```
#>   1,000     0.0020    0.0020      1x
#>   5,000     0.0110    0.0000      Infx
#>   10,000    0.0400    0.0010      40x
#>   20,000    0.1620    0.0000      Infx
```

The growing vector copies the entire vector on each append, producing $O(n^2)$ total work. Pre-allocation is $O(n)$.

23.4.4 When to consider Rcpp

For the spatial PD, vectorization delivered a large speedup because the core operation — summing neighbor states — is naturally expressed as matrix arithmetic. But some algorithms resist vectorization:

- **Sequential dependencies**: When round t's outcome depends on the order of updates within round $t-1$, you cannot compute all cells simultaneously.

- **Complex conditionals**: Strategies with multi-step memory require per-agent branching that `ifelse()` cannot cleanly express.
- **Graph traversal**: Shortest paths and clustering on irregular networks involve queues and visited-node tracking that are inherently sequential.

In these cases, **Rcpp** lets you write the inner loop in C++ while keeping the R interface. A typical C++ function takes R matrices as input, performs the sequential computation, and returns the result — with 50–200x speedups over R loops.

The decision rule is simple: **vectorize first, Rcpp second**. If the bottleneck can be expressed as matrix operations, do that. Only reach for Rcpp when it cannot.

23.5 Extensions

- **Parallel computation**: R's `parallel` package provides `mclapply()` (Unix) and `parLapply()` (all platforms) for embarrassingly parallel tasks like running independent replications.
- **Sparse matrices**: The `Matrix` package provides sparse representations that reduce storage from $O(n^2)$ to $O(m)$ for networks with m edges, and its arithmetic skips zero entries.
- **Byte compilation**: `compiler::cmpfun()` byte-compiles R functions, delivering 2–5x speedups for loop-heavy code with zero code changes.
- **Profiling tools**: `Rprof()` provides sampling-based profiling, and `profvis::profvis()` offers interactive flame-graph visualizations for identifying hotspots.

Exercises

1. **Vectorize the tournament.** The `run_tournament()` function in `R/axelrod.R` uses nested loops over strategy pairs. Each match is independent, so the outer loop is embarrassingly parallel. Write a version that stores results in a pre-allocated matrix and measure the speedup.

2. **Memory scaling.** Modify the spatial PD to store the full grid at each round using (a) a growing list `grid_list[[t]] <- grid` and (b) a pre-allocated 3D array `array(0L, dim = c(n, n, rounds))`. Benchmark both for $n = 50$ and 100 rounds.

3. **Identify the crossover.** Find the grid size n at which the vectorized version becomes faster than the loop-based version. Plot both timing curves and mark the crossover point.

Solutions appear in D.

Part IV

Strategic AI

Chapter 24

Machine Learning Foundations for Game Theory

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

Supervised learning as a game against nature, regret minimization and regret matching, feature engineering for game-theoretic data, and cross-validation for strategic predictions.

Learning objectives

- Frame supervised learning as a two-player game between a learner and nature.
- Implement regret matching from scratch and show convergence to Nash equilibrium in Rock-Paper-Scissors.
- Construct features from game-theoretic data suitable for prediction tasks.
- Apply cross-validation to evaluate predictive models of strategic behavior.

24.1 Motivation

Machine learning and game theory share a deep intellectual kinship. At its core, supervised learning is a minimax problem: a learner selects a hypothesis to minimize worst-case loss, while "nature" (the data-generating process) selects inputs that expose the learner's weaknesses. This adversarial framing is not

merely a metaphor — it is the foundation of online learning theory and connects directly to regret minimization, a concept central to modern algorithmic game theory.

This chapter bridges the two fields. We show how regret matching — a simple no-regret algorithm — converges to Nash equilibrium in two-player zero-sum games, then pivot to practical ML tools (logistic regression, cross-validation) applied to game-theoretic prediction tasks. The goal is to equip the reader with ML techniques that will appear throughout the remaining chapters on reinforcement learning, self-play, and counterfactual regret minimization.

24.2 Theory

24.2.1 Supervised learning as a game

In the **statistical learning** framework, a learner observes training data $\{(x_i, y_i)\}_{i=1}^n$ drawn from an unknown distribution $\mathcal{D}$ and selects a hypothesis h from a class $\mathcal{H}$ to minimize expected loss:

$$\min_{h \in \mathcal{H}} \mathbb{E}_{(x,y)\sim\mathcal{D}} \left[\ell(h(x), y)\right] \tag{24.1}$$

This has a natural game-theoretic interpretation. The learner is Player 1, choosing h. Nature is Player 2, choosing (x, y). The loss function ℓ is the payoff matrix — the learner minimizes it, nature maximizes it. The minimax theorem guarantees that in this zero-sum formulation, optimal learning strategies exist.

24.2.2 Online learning and regret

In **online learning**, decisions are made sequentially. At each round t, the learner chooses an action a_t from a set $\mathcal{A}$, then observes the loss vector $\ell_t(a)$ for all actions. The **cumulative regret** of not having played action a is:

$$R_T(a) = \sum_{t=1}^{T} \ell_t(a_t) - \sum_{t=1}^{T} \ell_t(a) \tag{24.2}$$

A **no-regret** algorithm ensures that $\max_a R_T(a)/T \to 0$ as $T \to \infty$. The celebrated connection to game theory is: if both players in a zero-sum game use no-regret algorithms, their time-averaged strategies converge to a Nash equilibrium (Cesa-Bianchi & Lugosi, 2006).

24.2.3 Regret matching

Regret matching (Hart & Mas-Colell, 2000) is one of the simplest no-regret algorithms. At each round, the player computes cumulative regret for each action and plays proportionally to positive regrets:

$$\sigma_t(a) = \begin{cases} R_t^+(a) \,/\, \sum_{a'} R_t^+(a') & \text{if } \sum_{a'} R_t^+(a') > 0 \\ 1/|\mathcal{A}| & \text{otherwise} \end{cases} \tag{24.3}$$

where $R_t^+(a) = \max(0, R_t(a))$. This algorithm is the building block of **counterfactual regret minimization** (CFR), covered in 29.

24.2.4 Feature representation for game-theoretic data

When applying ML to strategic settings, feature engineering matters. Common features include:

- **History features**: opponent's action frequencies over a sliding window.
- **Payoff features**: expected payoff under each action given an opponent model.
- **State features**: round number, cumulative scores, cooperation rate.

24.3 Implementation in R

24.3.1 Regret matching for Rock-Paper-Scissors

```
regret_matching <- function(n_iterations = 5000) {
  actions <- c("Rock", "Paper", "Scissors")
  n_actions <- length(actions)
  # Payoff matrix for row player: rows = P1, cols = P2
  payoff <- matrix(c( 0, -1,  1,
                      1,  0, -1,
                     -1,  1,  0), nrow = 3, byrow = TRUE)

  cum_regret_1 <- rep(0, n_actions)
  cum_regret_2 <- rep(0, n_actions)
  cum_strategy_1 <- rep(0, n_actions)
  cum_strategy_2 <- rep(0, n_actions)

  strategy_history <- matrix(0, nrow = n_iterations, ncol =
↪  n_actions)
```

```
  get_strategy <- function(cum_regret) {
    pos_regret <- pmax(cum_regret, 0)
    total <- sum(pos_regret)
    if (total > 0) pos_regret / total else rep(1 / n_actions,
     ↪  n_actions)
  }

  for (t in seq_len(n_iterations)) {
    sigma_1 <- get_strategy(cum_regret_1)
    sigma_2 <- get_strategy(cum_regret_2)
    cum_strategy_1 <- cum_strategy_1 + sigma_1
    cum_strategy_2 <- cum_strategy_2 + sigma_2
    strategy_history[t, ] <- cum_strategy_1 / sum(cum_strategy_1)

    # Sample actions
    a1 <- sample(n_actions, 1, prob = sigma_1)
    a2 <- sample(n_actions, 1, prob = sigma_2)

    # Update regrets
    for (a in seq_len(n_actions)) {
      cum_regret_1[a] <- cum_regret_1[a] + payoff[a, a2] -
↪  payoff[a1, a2]
      cum_regret_2[a] <- cum_regret_2[a] + (-payoff[a1, a]) -
↪  (-payoff[a1, a2])
    }
  }

  avg_strategy_1 <- cum_strategy_1 / sum(cum_strategy_1)
  avg_strategy_2 <- cum_strategy_2 / sum(cum_strategy_2)

  list(
    avg_strategy_1 = setNames(avg_strategy_1, actions),
    avg_strategy_2 = setNames(avg_strategy_2, actions),
    history = strategy_history
  )
}
```

```
set.seed(42)
rm_result <- regret_matching(n_iterations = 5000)
```

```
rm_df <- tibble(
  iteration = rep(1:5000, 3),
  action = rep(c("Rock", "Paper", "Scissors"), each = 5000),
  probability = c(rm_result$history[, 1],
```

```
                  rm_result$history[, 2],
                  rm_result$history[, 3])
)

p_regret <- ggplot(rm_df, aes(x = iteration, y = probability,
 ↪  colour = action)) +
  geom_line(alpha = 0.8) +
  geom_hline(yintercept = 1/3, linetype = "dashed", colour =
   ↪  "grey40") +
  scale_colour_manual(values = c("Rock" = okabe_ito[1],
                                 "Paper" = okabe_ito[2],
                                 "Scissors" = okabe_ito[3])) +
  theme_publication() +
  labs(title = "Regret Matching Convergence in RPS",
       x = "Iteration", y = "Average strategy probability",
       colour = "Action") +
  annotate("text", x = 4500, y = 0.36, label = "NE = 1/3",
           colour = "grey40", size = 3.5)

p_regret
```

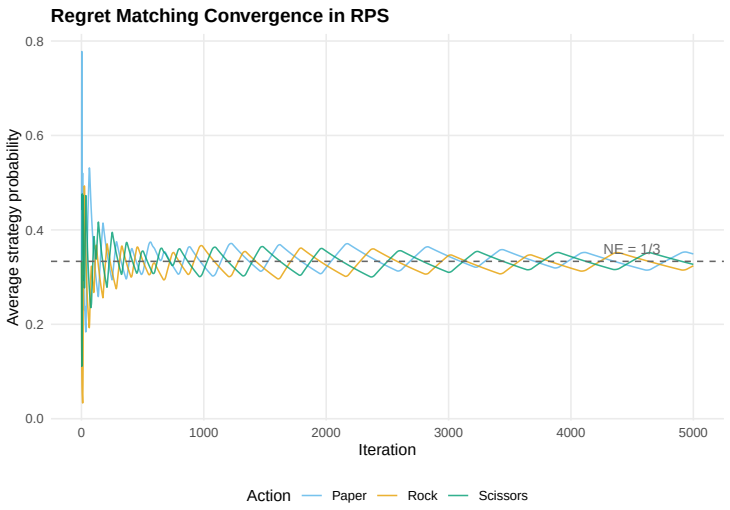


Figure 24.1: Regret matching convergence in Rock-Paper-Scissors. Player 1's time-averaged strategy converges toward the Nash equilibrium of (1/3, 1/3, 1/3) for each action.

```
save_pub_fig(p_regret, "regret-matching-convergence", width = 7,
 ↪  height = 5)
```

The average strategies converge to $(1/3, 1/3, 1/3)$ — the unique Nash equilibrium of Rock-Paper-Scissors. This confirms the theoretical guarantee: regret matching is a no-regret algorithm, and when both players use it, the time-averaged play converges to NE.

```
cat("Player 1 average strategy:\n")
```

```
#> Player 1 average strategy:
```

```
print(round(rm_result$avg_strategy_1, 4))
```

```
#>     Rock    Paper Scissors
#>    0.324    0.349    0.327
```

```
cat("\nPlayer 2 average strategy:\n")
```

```
#>
#> Player 2 average strategy:
```

```
print(round(rm_result$avg_strategy_2, 4))
```

```
#>     Rock    Paper Scissors
#>    0.336    0.321    0.343
```

24.3.2 Decision boundary visualization

To illustrate supervised learning in a game-theoretic context, consider a binary classification task: given two features derived from game play, predict an outcome. We fit a logistic regression model using `glm` and visualize the decision boundary.

```
set.seed(123)
n <- 200

# Simulate two player types: cooperators (y=1) and defectors
 ↪  (y=0)
```

```
# Feature 1: opponent's cooperation rate; Feature 2: round number
 ↪  (normalized)
x1_coop <- rnorm(n/2, mean = 0.7, sd = 0.15)
x2_coop <- rnorm(n/2, mean = 0.6, sd = 0.2)
x1_def  <- rnorm(n/2, mean = 0.3, sd = 0.15)
x2_def  <- rnorm(n/2, mean = 0.4, sd = 0.2)

game_data <- tibble(
  opp_coop_rate = c(x1_coop, x1_def),
  normalized_round = c(x2_coop, x2_def),
  cooperate = factor(c(rep(1, n/2), rep(0, n/2)))
)

model <- glm(cooperate ~ opp_coop_rate + normalized_round,
             data = game_data, family = binomial)
```

```
# Create prediction grid
grid <- expand.grid(
  opp_coop_rate = seq(-0.1, 1.1, length.out = 200),
  normalized_round = seq(-0.2, 1.2, length.out = 200)
)
grid$prob <- predict(model, newdata = grid, type = "response")

p_boundary <- ggplot() +
  geom_tile(data = grid, aes(x = opp_coop_rate, y =
   ↪  normalized_round,
                             fill = prob), alpha = 0.4) +
  scale_fill_gradient2(low = okabe_ito[2], mid = "white", high =
   ↪  okabe_ito[1],
                       midpoint = 0.5, name = "P(Cooperate)") +
  geom_point(data = game_data, aes(x = opp_coop_rate, y =
   ↪  normalized_round,
                                   colour = cooperate),
             size = 1.5, alpha = 0.7) +
  scale_colour_manual(values = c("0" = okabe_ito[2], "1" =
   ↪  okabe_ito[1]),
                      labels = c("Defect", "Cooperate"), name =
                       ↪  "Outcome") +
  theme_publication() +
  labs(title = "Decision Boundary: Predicting Cooperation",
       x = "Opponent cooperation rate", y = "Normalized round")

p_boundary
```

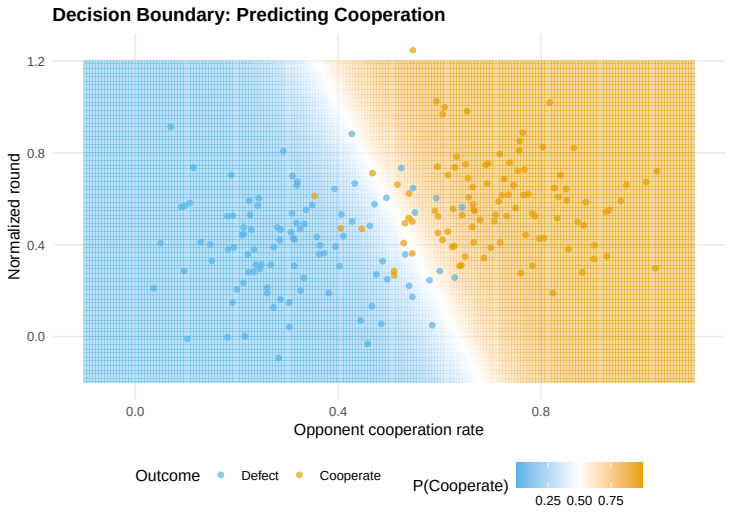


Figure 24.2: Decision boundary for predicting cooperation. The logistic regression separates cooperators (orange) from defectors (blue) based on the opponent's historical cooperation rate and normalized round number.

```
save_pub_fig(p_boundary, "decision-boundary-cooperation", width =
 ↪  7, height = 5)
```

24.4 Worked example

24.4.1 Predicting cooperation in the iterated Prisoner's Dilemma

We simulate an iterated Prisoner's Dilemma and use logistic regression to predict whether a player cooperates on a given round, based on observable features.

```
set.seed(42)
n_rounds <- 200
n_pairs <- 50

# Simulate IPD data: each pair plays n_rounds
ipd_data <- map_dfr(seq_len(n_pairs), function(pair_id) {
  # Each player has a base cooperation probability that drifts
   ↪  with opponent behavior
  p1_coop <- 0.5
  p2_coop <- 0.5
```

```
  records <- vector("list", n_rounds)

  for (r in seq_len(n_rounds)) {
    a1 <- rbinom(1, 1, p1_coop)
    a2 <- rbinom(1, 1, p2_coop)
    # Tit-for-tat-ish adaptation
    p1_coop <- 0.3 * p1_coop + 0.7 * (0.2 + 0.6 * a2)
    p2_coop <- 0.3 * p2_coop + 0.7 * (0.2 + 0.6 * a1)

    records[[r]] <- tibble(pair = pair_id, round = r, p1_action
 ↪  = a1, p2_action = a2)
  }
  bind_rows(records)
})

# Rolling mean over previous 3 rounds (base R, no slider)
roll_mean3 <- function(x) {
  n <- length(x)
  out <- rep(NA_real_, n)
  for (i in 4:n) out[i] <- mean(x[(i - 3):(i - 1)])
  out
}

# Engineer features for predicting P1's cooperation
ipd_features <- ipd_data |>
  group_by(pair) |>
  mutate(
    opp_coop_last3 = roll_mean3(p2_action),
    own_coop_last3 = roll_mean3(p1_action),
    round_norm = round / n_rounds
  ) |>
  ungroup() |>
  filter(round > 4)
```

```
# Split train/test by pair (avoid data leakage)
train_pairs <- sample(n_pairs, 35)
train <- ipd_features |> filter(pair %in% train_pairs)
test  <- ipd_features |> filter(!pair %in% train_pairs)

ipd_model <- glm(p1_action ~ opp_coop_last3 + own_coop_last3 +
 ↪  round_norm,
                 data = train, family = binomial)

cat("Logistic regression coefficients:\n")
```

```
#> Logistic regression coefficients:
```

```
print(round(coef(ipd_model), 3))
```

```
#>    (Intercept) opp_coop_last3 own_coop_last3     round_norm
#>         -1.346          2.678          0.018         -0.048
```

```
# 5-fold cross-validation by pair
folds <- split(train_pairs, cut(seq_along(train_pairs), 5, labels
 ↪  = FALSE))

cv_accuracy <- map_dbl(folds, function(held_out) {
  cv_train <- ipd_features |> filter(pair %in%
 ↪  setdiff(train_pairs, held_out))
  cv_test  <- ipd_features |> filter(pair %in% held_out)

  fit <- glm(p1_action ~ opp_coop_last3 + own_coop_last3 +
 ↪  round_norm,
             data = cv_train, family = binomial)

  preds <- ifelse(predict(fit, newdata = cv_test, type =
 ↪  "response") > 0.5, 1, 0)
  mean(preds == cv_test$p1_action)
})

cat(sprintf("5-fold CV accuracy: %.1f%% (SD: %.1f%%)\n",
            100 * mean(cv_accuracy), 100 * sd(cv_accuracy)))
```

```
#> 5-fold CV accuracy: 66.6% (SD: 1.1%)
```

```
test_preds <- predict(ipd_model, newdata = test, type =
 ↪  "response")
test_class <- ifelse(test_preds > 0.5, 1, 0)
test_accuracy <- mean(test_class == test$p1_action)

cat(sprintf("Test set accuracy: %.1f%%\n", 100 * test_accuracy))
```

```
#> Test set accuracy: 68.2%
```

```
cat(sprintf("Baseline (always predict majority class): %.1f%%\n",
            100 * max(mean(test$p1_action), 1 -
             ↪  mean(test$p1_action))))
```

```
#> Baseline (always predict majority class): 52.3%
```

The opponent's recent cooperation rate is the strongest predictor — consistent with tit-for-tat-like dynamics. Cross-validation by pair (rather than by round) prevents data leakage from correlated within-pair observations, a critical methodological point when applying ML to repeated games.

24.5 Extensions

- **Multiplicative weights update** is an alternative no-regret algorithm with tighter theoretical bounds. See Arora et al. (2012) for a survey of the multiplicative weights method and its connections to game theory.
- **Counterfactual regret minimization** (29) extends regret matching to extensive-form games, powering superhuman poker AI.
- **Reinforcement learning** (25) replaces the fixed-loss setting of online learning with sequential decision-making under an MDP.
- **Feature engineering** for strategic data is an active research area: deep learning approaches learn features automatically, but interpretable features (cooperation rates, payoff summaries) remain valuable for analysis.

Exercises

1. **Regret matching in Prisoner's Dilemma.** Implement regret matching for the one-shot Prisoner's Dilemma with payoffs $T = 5, R = 3, P = 1, S = 0$. To what strategy profile does the average play converge? Explain why this differs from RPS.

2. **Regularization and overfitting.** Add polynomial features (quadratic and interaction terms) to the IPD logistic regression model. Compare training and test accuracy. Does the richer model overfit? Use cross-validation to select the best model.

3. **Window size sensitivity.** In the worked example, features use a 3-round sliding window. Repeat the analysis with window sizes of 1, 5, 10, and 20. Plot cross-validation accuracy versus window size and explain the trade-off between responsiveness and stability.

Solutions appear in D.

Chapter 25

Reinforcement Learning Foundations

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

Markov decision processes, the Bellman equation, tabular Q-learning and SARSA from scratch, and the exploration-exploitation trade-off via epsilon-greedy and UCB.

Learning objectives

- Define a Markov decision process (MDP) and derive the Bellman optimality equation.
- Implement tabular Q-learning from scratch for a gridworld environment.
- Compare Q-learning (off-policy) and SARSA (on-policy) in terms of learned policies.
- Analyze exploration strategies: ε-greedy versus upper confidence bound (UCB).

25.1 Motivation

An agent navigates a maze, choosing directions and receiving rewards, with no prior knowledge of the map. This is the **reinforcement learning** (RL) problem (Sutton & Barto, 2018). Unlike supervised learning (correct answers provided) or game theory's analytical equilibria, RL learns through trial-and-error interaction. The framework is foundational for multi-agent learning: be-

fore understanding how two Q-learners interact in a matrix game (26), we must understand single-agent Q-learning.

25.2 Theory

25.2.1 Markov decision processes

A **Markov decision process** (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ where:

- $\mathcal{S}$ is a finite set of states,
- $\mathcal{A}$ is a finite set of actions,
- $P(s' \mid s, a)$ is the transition probability,
- $R(s, a, s')$ is the reward function,
- $\gamma \in [0, 1)$ is the discount factor.

The agent's goal is to find a policy $\pi(a \mid s)$ that maximizes expected cumulative discounted reward:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R_t \,\middle|\, S_0 = s\right] \tag{25.1}$$

25.2.2 The Bellman equation

The optimal value function $V^*(s)$ satisfies the **Bellman optimality equation**:

$$V^*(s) = \max_{a \in \mathcal{A}} \sum_{s'} P(s' \mid s, a)\,[R(s, a, s') + \gamma\, V^*(s')] \tag{25.2}$$

Equivalently, the optimal action-value function $Q^*(s, a)$ satisfies:

$$Q^*(s, a) = \sum_{s'} P(s' \mid s, a)\left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a')\right] \tag{25.3}$$

25.2.3 Tabular Q-learning

Q-learning (Watkins & Dayan, 1992) is a model-free, off-policy algorithm. After taking action a in state s, observing reward r and next state s':

$$Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right] \tag{25.4}$$

The max operator makes it off-policy: Q-learning updates toward the best possible next action regardless of what was actually taken. Under standard conditions, $Q \to Q^*$.

25.2.4 SARSA

SARSA (State-Action-Reward-State-Action) is an on-policy alternative that bootstraps from the action a' actually chosen, rather than the greedy max:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma\, Q(s',a') - Q(s,a)\right] \tag{25.5}$$

SARSA learns the value of the policy it follows (including exploration), making it more conservative near dangerous states.

25.2.5 Exploration vs exploitation

The agent must balance **exploitation** (best-known action) with **exploration** (trying alternatives). Two standard strategies: **ε-greedy** (random action with probability ε, greedy otherwise) and **UCB** (choose $a = \arg\max_a [Q(s,a) + c\sqrt{\ln t / N(s,a)}]$, exploring in proportion to uncertainty).

25.3 Implementation in R

25.3.1 Gridworld environment

```
create_gridworld <- function(rows = 5, cols = 5, obstacles =
↪  NULL,
                             goal = NULL, goal_reward = 10,
                              ↪  step_cost = -0.1) {
  if (is.null(goal)) goal <- c(rows, cols)
  # State encoding: (row, col) -> integer id
  state_id <- function(r, c) (r - 1) * cols + c
  id_to_rc <- function(id) c((id - 1) %/% cols + 1, (id - 1) %%
↪  cols + 1)

  is_obstacle <- function(r, c) {
    if (is.null(obstacles)) return(FALSE)
    any(obstacles[, 1] == r & obstacles[, 2] == c)
  }

  # Actions: 1=up, 2=right, 3=down, 4=left
```

```
  action_delta <- list(c(-1, 0), c(0, 1), c(1, 0), c(0, -1))
  action_names <- c("Up", "Right", "Down", "Left")

  step <- function(state, action) {
    rc <- id_to_rc(state)
    delta <- action_delta[[action]]
    nr <- rc[1] + delta[1]
    nc <- rc[2] + delta[2]

    # Stay in place if hitting wall or obstacle
    if (nr < 1 || nr > rows || nc < 1 || nc > cols ||
     ↪  is_obstacle(nr, nc)) {
      return(list(next_state = state, reward = step_cost, done =
       ↪  FALSE))
    }

    next_state <- state_id(nr, nc)
    done <- (nr == goal[1] && nc == goal[2])
    reward <- if (done) goal_reward else step_cost

    list(next_state = next_state, reward = reward, done = done)
  }

  list(
    rows = rows, cols = cols, n_states = rows * cols, n_actions =
     ↪  4,
    start = state_id(1, 1), goal = state_id(goal[1], goal[2]),
    step = step, state_id = state_id, id_to_rc = id_to_rc,
    action_names = action_names, obstacles = obstacles
  )
}
```

25.3.2 Tabular Q-learning agent

```
q_learning_gridworld <- function(env, episodes = 1000,
                                 alpha = 0.1, gamma = 0.95,
                                 epsilon = 0.1, max_steps = 200)
                                  ↪  {
  Q <- matrix(0, nrow = env$n_states, ncol = env$n_actions)
  episode_rewards <- numeric(episodes)

  for (ep in seq_len(episodes)) {
    state <- env$start
```

```
    total_reward <- 0

    for (step in seq_len(max_steps)) {
      # Epsilon-greedy action selection
      if (runif(1) < epsilon) {
        action <- sample(env$n_actions, 1)
      } else {
        ties <- which(Q[state, ] == max(Q[state, ]))
        action <- sample(ties, 1)
      }

      result <- env$step(state, action)
      # Q-learning update (off-policy: uses max over next
       ↪  actions)
      best_next <- max(Q[result$next_state, ])
      Q[state, action] <- Q[state, action] +
        alpha * (result$reward + gamma * best_next - Q[state,
 ↪  action])

      total_reward <- total_reward + result$reward
      state <- result$next_state
      if (result$done) break
    }
    episode_rewards[ep] <- total_reward
  }

  list(Q = Q, rewards = episode_rewards)
}
```

25.3.3 SARSA agent

```
sarsa_gridworld <- function(env, episodes = 1000,
                            alpha = 0.1, gamma = 0.95,
                            epsilon = 0.1, max_steps = 200) {
  Q <- matrix(0, nrow = env$n_states, ncol = env$n_actions)
  episode_rewards <- numeric(episodes)

  choose_action <- function(state) {
    if (runif(1) < epsilon) {
      sample(env$n_actions, 1)
    } else {
      ties <- which(Q[state, ] == max(Q[state, ]))
      sample(ties, 1)
```

```
    }
  }

  for (ep in seq_len(episodes)) {
    state <- env$start
    action <- choose_action(state)
    total_reward <- 0

    for (step in seq_len(max_steps)) {
      result <- env$step(state, action)
      next_action <- choose_action(result$next_state)

      # SARSA update (on-policy: uses actual next action)
      Q[state, action] <- Q[state, action] +
        alpha * (result$reward + gamma * Q[result$next_state,
↪   next_action] -
                   Q[state, action])

      total_reward <- total_reward + result$reward
      state <- result$next_state
      action <- next_action
      if (result$done) break
    }
    episode_rewards[ep] <- total_reward
  }

  list(Q = Q, rewards = episode_rewards)
}
```

25.3.4 Training on the gridworld

```
obstacles <- matrix(c(2, 2,  3, 4,  4, 2,  2, 4), ncol = 2, byrow
↪   = TRUE)

env <- create_gridworld(rows = 5, cols = 5, obstacles =
↪   obstacles)

set.seed(42)
ql_result <- q_learning_gridworld(env, episodes = 2000,
                                  alpha = 0.15, gamma = 0.95,
                                   ↪  epsilon = 0.1)
set.seed(42)
sarsa_result <- sarsa_gridworld(env, episodes = 2000,
```

```
                                  alpha = 0.15, gamma = 0.95,
                                   ↪  epsilon = 0.1)
```

```
# Extract max Q-value per state as value function
V <- apply(ql_result$Q, 1, max)

grid_df <- tibble(
  state = seq_len(env$n_states),
  row = (state - 1) %/% env$cols + 1,
  col = (state - 1) %% env$cols + 1,
  value = V
)

# Mark obstacles
obs_df <- tibble(
  row = obstacles[, 1],
  col = obstacles[, 2]
)

p_heatmap <- ggplot(grid_df, aes(x = col, y = row, fill = value))
 ↪  +
  geom_tile(colour = "white", linewidth = 0.5) +
  geom_tile(data = obs_df, aes(x = col, y = row),
            fill = "grey20", inherit.aes = FALSE) +
  geom_text(aes(label = round(value, 1)), colour = "black", size
   ↪  = 3.5) +
  annotate("text", x = 1, y = 1, label = "S", fontface = "bold",
   ↪  size = 5) +
  annotate("text", x = 5, y = 5, label = "G", fontface = "bold",
   ↪  size = 5) +
  scale_fill_gradient(low = "white", high = okabe_ito[1], name =
   ↪  "V(s)") +
  scale_y_reverse() +
  coord_equal() +
  theme_publication() +
  theme(panel.grid = element_blank()) +
  labs(title = "Q-Learning: Learned State Values",
       x = "Column", y = "Row")

p_heatmap
```

```
save_pub_fig(p_heatmap, "q-learning-gridworld-heatmap", width =
 ↪  6, height = 5)
```

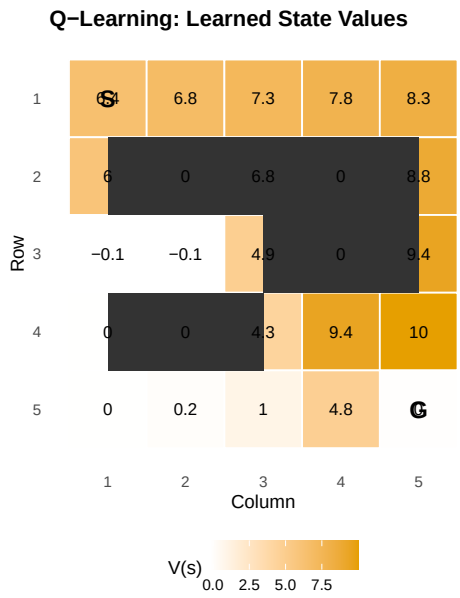


Figure 25.1: Learned state values (max Q per state) in the 5x5 gridworld. The agent starts at (1,1) and must reach the goal at (5,5). Dark cells are obstacles. Higher values (orange) indicate states closer to the goal along the optimal path.

25.3.5 Exploration comparison: epsilon-greedy vs UCB

```
run_bandit <- function(strategy, n_arms = 10, n_steps = 2000,
                       epsilon = 0.1, ucb_c = 2) {
  true_values <- rnorm(n_arms); Q <- rep(0, n_arms); N <- rep(0,
 ↪  n_arms)
  cum_regret <- numeric(n_steps); optimal <- max(true_values);
 ↪  tot <- 0
  for (t in seq_len(n_steps)) {
    action <- if (strategy == "epsilon_greedy") {
      if (runif(1) < epsilon) sample(n_arms, 1) else which.max(Q)
    } else {
      if (any(N == 0)) which(N == 0)[1]
      else which.max(Q + ucb_c * sqrt(log(t) / N))
    }
    reward <- rnorm(1, mean = true_values[action])
    N[action] <- N[action] + 1
    Q[action] <- Q[action] + (reward - Q[action]) / N[action]
    tot <- tot + (optimal - true_values[action]); cum_regret[t]
 ↪  <- tot
  }
  cum_regret
}

set.seed(42); n_runs <- 50; n_steps <- 2000
eg_regret <- matrix(0, n_runs, n_steps); ucb_regret <- matrix(0,
 ↪  n_runs, n_steps)
for (i in seq_len(n_runs)) {
  eg_regret[i, ] <- run_bandit("epsilon_greedy", n_steps =
 ↪  n_steps)
  ucb_regret[i, ] <- run_bandit("ucb", n_steps = n_steps)
}

regret_df <- tibble(
  step = rep(1:n_steps, 2),
  regret = c(colMeans(eg_regret), colMeans(ucb_regret)),
  strategy = rep(c("Epsilon-greedy", "UCB"), each = n_steps)
)
p_explore <- ggplot(regret_df, aes(x = step, y = regret, colour
 ↪  = strategy)) +
  geom_line(linewidth = 0.8) +
  scale_colour_manual(values = c("Epsilon-greedy" = okabe_ito[1],
                                 "UCB" = okabe_ito[2])) +
  theme_publication() +
  labs(title = "Exploration: Epsilon-Greedy vs UCB",
```

```
       x = "Step", y = "Cumulative regret", colour = "Strategy")
p_explore
```

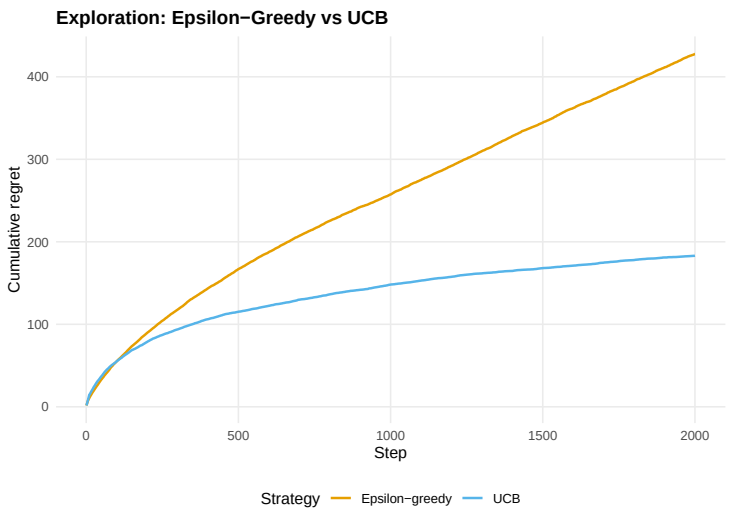


Figure 25.2: Cumulative regret comparison between epsilon-greedy and UCB on a 10-armed bandit. UCB explores proportionally to uncertainty, yielding lower regret.

```
save_pub_fig(p_explore, "exploration-eg-vs-ucb", width = 7,
 ↪  height = 5)
```

25.4 Worked example

We trace the learning process on the 5x5 gridworld with obstacles at (2,2), (3,4), (4,2), and (2,4). The agent starts at (1,1) and must reach the goal at (5,5).

```
cat("Q-Learning - Mean reward (last 200 episodes):",
    round(mean(tail(ql_result$rewards, 200)), 2), "\n")
```

```
#> Q-Learning - Mean reward (last 200 episodes): 6.26
```

```
cat("SARSA      - Mean reward (last 200 episodes):",
    round(mean(tail(sarsa_result$rewards, 200)), 2), "\n")
```

```
#> SARSA      - Mean reward (last 200 episodes): 6.28
```

```
# Extract and display the learned policy
policy <- apply(ql_result$Q, 1, which.max)
arrows <- c("Up", "Rt", "Dn", "Lt")
obs_states <- env$state_id(obstacles[,1], obstacles[,2])

policy_matrix <- matrix("", nrow = env$rows, ncol = env$cols)
for (s in seq_len(env$n_states)) {
  r <- (s - 1) %/% env$cols + 1
  cc <- (s - 1) %% env$cols + 1
  if (s %in% obs_states) {
    policy_matrix[r, cc] <- " X "
  } else if (s == env$goal) {
    policy_matrix[r, cc] <- " G "
  } else {
    policy_matrix[r, cc] <- arrows[policy[s]]
  }
}
cat("\nLearned policy (Q-learning):\n")
```

```
#>
#> Learned policy (Q-learning):
```

```
print(policy_matrix, quote = FALSE)
```

```
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,] Rt   Rt   Rt   Rt   Dn
#> [2,] Up    X   Up    X   Dn
#> [3,] Dn   Dn   Up    X   Dn
#> [4,] Lt    X   Rt   Rt   Dn
#> [5,] Rt   Rt   Rt   Rt    G
```

Q-learning finds the optimal route because it evaluates the greedy policy (via max). SARSA gives obstacles a wider berth because its on-policy updates incorporate the risk of accidentally stepping into a bad state during exploration.

25.5 Extensions

- **Function approximation**: For large state spaces, Q-values can be approximated with linear models or neural networks (DQN). Neural network packages are beyond our R toolkit, but the conceptual framework is identical.

- **Policy gradient methods**: REINFORCE and actor-critic methods directly optimize policy parameters rather than learning value functions.
- **Multi-agent extensions**: 26 explores what happens when two Q-learners face each other in matrix games — single-agent convergence guarantees break down.
- **Planning**: Model-based RL (Dyna-Q) combines learned models with Q-learning for faster convergence. See Sutton & Barto (2018) (Chapters 5-8).

Exercises

1. **Cliff walking.** Create a 4x12 gridworld where the bottom row (except the start and goal) is a "cliff" with reward -100. Compare Q-learning and SARSA policies. Which algorithm learns the safer path along the top?

2. **Decaying epsilon.** Implement $\varepsilon_t = 1/\sqrt{t}$ decaying exploration for the 5x5 gridworld. Plot the learning curve alongside fixed $\varepsilon = 0.1$ and compare convergence speed and final performance.

3. **Discount factor analysis.** Train Q-learning on the 5x5 gridworld with $\gamma \in \{0.5, 0.9, 0.99\}$. Visualize the learned value functions as heatmaps. How does the discount factor affect the value gradient from start to goal?

Solutions appear in D.

Chapter 26

Q-Learning in 2-Player Games

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

Tabular Q-learning in matrix games: convergence in coordination games, divergence in zero-sum games, and the rational learning debate.

Learning objectives

- Explain how single-agent Q-learning extends to multi-agent matrix games.
- Implement independent Q-learners for repeated 2-player games in R.
- Demonstrate convergence to Nash equilibrium in coordination games and cycling in zero-sum games.
- Discuss the "rational learning" debate and the limits of independent learning.

26.1 Motivation

Reinforcement learning has achieved remarkable success in single-agent settings: an agent interacts with an environment, receives rewards, and learns a policy that maximizes long-term return. But what happens when the "environment" is another learning agent?

In a matrix game played repeatedly, two Q-learners each try to maximize their own payoff. The environment each agent faces is **non-stationary** — it changes

as the other agent updates its policy. This creates a fundamental tension: the convergence guarantees of Q-learning (25) rely on a stationary environment, and that assumption breaks in multi-agent settings.

Whether independent Q-learners converge to Nash equilibrium, cycle, or diverge depends on the game's structure. This chapter explores all three outcomes through simulation.

26.2 Theory

26.2.1 Q-learning recap

In single-agent Q-learning (Sutton & Barto, 2018, Chapter 6), an agent maintains a value estimate $Q(s, a)$ for each state-action pair and updates it after observing a reward r and next state s':

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{26.1}$$

For repeated matrix games, there is only one "state" (the game itself), so Q-values reduce to action values $Q(a_i)$ for each of player i's actions. The discount factor γ is irrelevant in a stateless setting, and the update simplifies to:

$$Q_i(a_i) \leftarrow Q_i(a_i) + \alpha \left[r_i - Q_i(a_i) \right] \tag{26.2}$$

where r_i is the payoff player i received from playing action a_i against the opponent's (unknown) action.

26.2.2 Independent learners

The simplest multi-agent approach is **independent Q-learning**: each agent runs its own Q-learning algorithm, treating the other agent as part of the environment. No agent models or observes the other's Q-values.

The action selection typically uses an ε-greedy policy: with probability ε the agent explores (chooses a random action), and with probability $1 - \varepsilon$ it exploits (chooses the action with the highest Q-value).

26.2.3 Convergence properties

The behavior of independent Q-learners depends critically on the game:

- **Coordination games** (e.g., Battle of the Sexes): learners tend to converge to one of the pure-strategy Nash equilibria. Which one depends on initial conditions and exploration noise.
- **Zero-sum games** (e.g., Matching Pennies): learners typically **cycle** rather than converge. Each agent's best response shifts the opponent's optimal play, creating a feedback loop.
- **The Prisoner's Dilemma**: learners converge to mutual defection — the unique Nash equilibrium — even though mutual cooperation yields higher payoffs.

Shoham and Leyton-Brown (2009) discuss the "rational learning" debate: under what conditions can independent learners reach equilibrium, and when should we expect them to fail?

26.3 Implementation in R

26.3.1 Q-learning agent

```
q_agent_create <- function(n_actions, alpha = 0.1, epsilon = 0.1)
 ↪  {
  Q <- rep(0, n_actions)
  list(
    choose = function() {
      if (runif(1) < epsilon) {
        sample(n_actions, 1)
      } else {
        ties <- which(Q == max(Q))
        sample(ties, 1)
      }
    },
    update = function(action, reward) {
      Q[action] <<- Q[action] + alpha * (reward - Q[action])
    },
    get_Q = function() Q
  )
}
```

26.3.2 Running a repeated game

```
run_q_learning_game <- function(A, B, episodes = 5000,
```

```
                                  alpha = 0.1, epsilon = 0.1) {
  n1 <- nrow(A)
  n2 <- ncol(A)
  agent1 <- q_agent_create(n1, alpha, epsilon)
  agent2 <- q_agent_create(n2, alpha, epsilon)

  history <- tibble(
    episode = integer(),
    a1 = integer(), a2 = integer(),
    r1 = double(), r2 = double(),
    Q1_1 = double(), Q1_2 = double(),
    Q2_1 = double(), Q2_2 = double()
  )

  for (ep in seq_len(episodes)) {
    a1 <- agent1$choose()
    a2 <- agent2$choose()
    r1 <- A[a1, a2]
    r2 <- B[a1, a2]
    agent1$update(a1, r1)
    agent2$update(a2, r2)

    Q1 <- agent1$get_Q()
    Q2 <- agent2$get_Q()
    history <- bind_rows(history, tibble(
      episode = ep, a1 = a1, a2 = a2, r1 = r1, r2 = r2,
      Q1_1 = Q1[1], Q1_2 = Q1[2],
      Q2_1 = Q2[1], Q2_2 = Q2[2]
    ))
  }
  history
}
```

26.3.3 Coordination game: convergence

```
A_coord <- matrix(c(2, 0, 0, 1), nrow = 2, byrow = TRUE,
                  dimnames = list(c("A", "B"), c("A", "B")))
B_coord <- A_coord

set.seed(42)
hist_coord <- run_q_learning_game(A_coord, B_coord,
                                  episodes = 3000, alpha = 0.1,
                                   ↪  epsilon = 0.05)
```

```
q_long <- hist_coord |>
  select(episode, Q1_1, Q1_2, Q2_1, Q2_2) |>
  pivot_longer(-episode, names_to = "variable", values_to = "Q")
   ↪  |>
  mutate(
    player = if_else(str_starts(variable, "Q1"), "Player 1",
     ↪  "Player 2"),
    action = if_else(str_ends(variable, "_1"), "Action A",
     ↪  "Action B")
  )

p_conv <- ggplot(q_long, aes(x = episode, y = Q, colour = action,
 ↪  linetype = player)) +
  geom_line(alpha = 0.7) +
  scale_colour_manual(values = c("Action A" = okabe_ito[1],
                                 "Action B" = okabe_ito[2])) +
  theme_publication() +
  labs(title = "Q-Value Convergence: Coordination Game",
       x = "Episode", y = "Q-value",
       colour = "Action", linetype = "Player")

p_conv
```

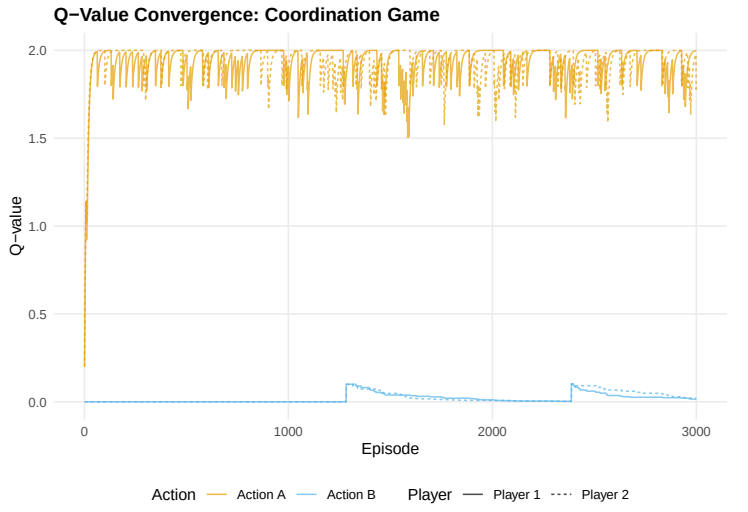


Figure 26.1: Q-value evolution in a coordination game. Both agents converge to action A (the payoff-dominant Nash equilibrium), with Q-values stabilizing near the equilibrium payoff of 2.

```
save_pub_fig(p_conv, "q-convergence-coordination", width = 7,
 ↪  height = 5)
```

26.3.4 Zero-sum game: cycling

```
# Matching Pennies
A_mp <- matrix(c(1, -1, -1, 1), nrow = 2, byrow = TRUE,
               dimnames = list(c("H", "T"), c("H", "T")))
B_mp <- -A_mp

set.seed(42)
hist_mp <- run_q_learning_game(A_mp, B_mp,
                               episodes = 3000, alpha = 0.05,
                                ↪  epsilon = 0.1)
```

```
q_long_mp <- hist_mp |>
  select(episode, Q1_1, Q1_2, Q2_1, Q2_2) |>
  pivot_longer(-episode, names_to = "variable", values_to = "Q")
   ↪  |>
  mutate(
    player = if_else(str_starts(variable, "Q1"), "Player 1",
     ↪  "Player 2"),
    action = if_else(str_ends(variable, "_1"), "Heads", "Tails")
  )

p_cycle <- ggplot(q_long_mp, aes(x = episode, y = Q, colour =
 ↪  action, linetype = player)) +
  geom_line(alpha = 0.7) +
  scale_colour_manual(values = c("Heads" = okabe_ito[1],
                                 "Tails" = okabe_ito[2])) +
  theme_publication() +
  labs(title = "Q-Value Cycling: Matching Pennies",
       x = "Episode", y = "Q-value",
       colour = "Action", linetype = "Player")

p_cycle
```

```
save_pub_fig(p_cycle, "q-cycling-matching-pennies", width = 7,
 ↪  height = 5)
```

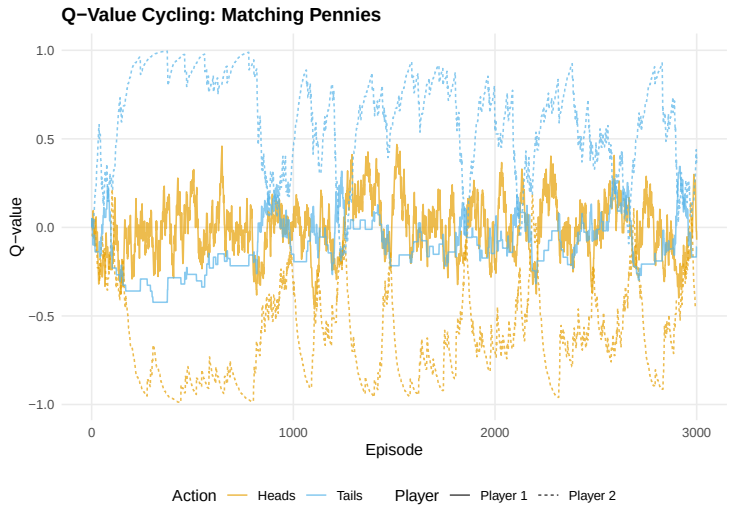


Figure 26.2: Q-value evolution in Matching Pennies (zero-sum). Instead of converging, Q-values oscillate as each agent's adaptation shifts the other's optimal play. The mixed NE at (0.5, 0.5) is never stably reached.

26.4 Worked example

Let us trace the first 20 episodes of the coordination game in detail:

```
trace <- hist_coord |>
  filter(episode <= 20) |>
  mutate(
    a1_label = if_else(a1 == 1, "A", "B"),
    a2_label = if_else(a2 == 1, "A", "B"),
    coordinated = a1 == a2
  )

cat("First 20 episodes of coordination game Q-learning:\n\n")
```

```
#> First 20 episodes of coordination game Q-learning:
```

```
trace |>
  select(episode, a1_label, a2_label, r1, r2, coordinated) |>
  rename(P1 = a1_label, P2 = a2_label, `P1 payoff` = r1,
         `P2 payoff` = r2, Coordinated = coordinated) |>
  print(n = 20)
```

```
#> # A tibble: 20 x 6
```

```
#>    episode P1    P2    `P1 payoff` `P2 payoff` Coordinated
#>      <int> <chr> <chr>       <dbl>       <dbl> <lgl>
#>  1       1 A     A               2           2 TRUE
#>  2       2 A     A               2           2 TRUE
#>  3       3 A     A               2           2 TRUE
#>  4       4 A     A               2           2 TRUE
#>  5       5 A     A               2           2 TRUE
#>  6       6 A     A               2           2 TRUE
#>  7       7 A     A               2           2 TRUE
#>  8       8 A     A               2           2 TRUE
#>  9       9 A     B               0           0 FALSE
#> 10      10 B     A               0           0 FALSE
#> 11      11 A     B               0           0 FALSE
#> 12      12 A     A               2           2 TRUE
#> 13      13 A     A               2           2 TRUE
#> 14      14 A     B               0           0 FALSE
#> 15      15 A     A               2           2 TRUE
#> 16      16 A     A               2           2 TRUE
#> 17      17 A     A               2           2 TRUE
#> 18      18 A     A               2           2 TRUE
#> 19      19 A     A               2           2 TRUE
#> 20      20 A     A               2           2 TRUE
```

```
cat(sprintf("\nCoordination rate (first 20): %.0f%%\n",
            100 * mean(trace$coordinated)))
```

```
#>
#> Coordination rate (first 20): 80%
```

```
cat(sprintf("Coordination rate (last 500): %.0f%%\n",
            100 * mean((hist_coord |> filter(episode > 2500))$a1
             ↪  ==
                        (hist_coord |> filter(episode >
 ↪  2500))$a2)))
```

```
#> Coordination rate (last 500): 94%
```

Early episodes show exploration: agents try both actions and sometimes miscoordinate. As Q-values diverge (one action becomes clearly better), the ε-greedy policy increasingly selects the high-Q action, and coordination improves. By the final 500 episodes, coordination is near-perfect.

In Matching Pennies, by contrast, no such convergence occurs — the Q-values never stabilize because each agent's adaptation continually shifts the landscape the other agent faces.

26.5 Extensions

- **Decaying exploration**: Using $\varepsilon_t = \varepsilon_0/t$ guarantees convergence in stationary environments but can lock in suboptimal play in non-stationary multi-agent settings.
- **Win-or-Learn-Fast (WoLF)**: Varies the learning rate based on whether the agent is winning or losing, achieving convergence in a broader class of games.
- **Multi-agent RL** (27) covers advanced approaches: MADDPG, PSRO, and the challenge of non-stationarity at scale.
- **Reinforcement learning foundations** (25) provides the single-agent background that this chapter builds on.

For the theoretical debate on rational learning in games, see Shoham & Leyton-Brown (2009) (Chapter 7).

Exercises

1. **Prisoner's Dilemma.** Run the Q-learning simulation on the standard Prisoner's Dilemma ($T = 5$, $R = 3$, $P = 1$, $S = 0$). Do both agents converge to mutual defection? How many episodes does it take? Plot the Q-value evolution.

2. **Learning rate sensitivity.** Run the coordination game with $\alpha \in \{0.01, 0.1, 0.5\}$ and fixed $\varepsilon = 0.05$. How does the learning rate affect convergence speed and stability? Plot Q-value trajectories for all three.

3. **Boltzmann exploration.** Replace ε-greedy action selection with Boltzmann (softmax) exploration: $P(a_i) = \exp(Q(a_i)/\tau)/\sum_{a'} \exp(Q(a')/\tau)$, where τ is a temperature parameter. Implement this and compare performance in the coordination game with ε-greedy.

Solutions appear in D.

Chapter 27

Multi-Agent Reinforcement Learning

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

Independent learners, joint-action learners, minimax-Q for zero-sum games, and the conceptual foundations of MADDPG and PSRO.

Learning objectives

- Distinguish independent learners from joint-action learners and explain why joint-action learning addresses non-stationarity.
- Implement minimax-Q learning for two-player zero-sum games in base R and show convergence to the minimax strategy.
- Describe the centralized-training, decentralized-execution paradigm underlying MADDPG.
- Explain how Policy Space Response Oracles (PSRO) combine reinforcement learning with game-theoretic solution concepts.

27.1 Motivation

Chapter 26 showed that independent Q-learners converge in coordination games but cycle in zero-sum games like Matching Pennies. This failure is not a bug in the algorithm — it reflects a fundamental limitation. Each agent treats the other as part of a stationary environment, but both are simultaneously adapting. The environment is anything but stationary.

Multi-agent reinforcement learning (MARL) addresses this challenge through algorithms that explicitly account for the presence of other learners. The simplest upgrade is **joint-action learning**, where each agent observes the opponent's actions and maintains Q-values over joint action profiles. More sophisticated approaches include **minimax-Q** (Littman, 1994), which finds optimal strategies in zero-sum games, and **MADDPG** (Lowe et al., 2017), which scales to continuous action spaces. At the frontier, **PSRO** (Lanctot et al., 2017) bridges MARL with classical game-theoretic solution concepts.

Understanding these approaches is essential for anyone working on competitive AI systems, from game-playing agents to adversarial robustness in machine learning.

27.2 Theory

27.2.1 Independent vs. joint-action learners

An **independent learner** (IL) maintains Q-values $Q_i(a_i)$ over its own actions only, as in 26. It ignores the opponent entirely.

A **joint-action learner** (JAL) observes the opponent's action after each round and maintains Q-values over the joint action profile: $Q_i(a_i, a_{-i})$. The agent also tracks an empirical model of the opponent's action frequencies $\hat{\pi}_{-i}(a_{-i})$ and selects actions that maximize expected payoff under that model:

$$a_i^* = \arg\max_{a_i} \sum_{a_{-i}} \hat{\pi}_{-i}(a_{-i})\, Q_i(a_i, a_{-i}) \tag{27.1}$$

JAL addresses non-stationarity by adapting its strategy to the opponent's observed behavior. However, it still assumes the opponent's policy is stationary — an assumption that breaks when both agents are learning.

27.2.2 Minimax-Q learning

For two-player zero-sum games, Littman (1994) proposed **minimax-Q**, which replaces the max operator in Q-learning with a minimax calculation. Instead of learning a deterministic best action, the agent learns a **mixed strategy** π_i that maximizes its worst-case expected payoff:

$$V_i = \max_{\pi_i} \min_{a_{-i}} \sum_{a_i} \pi_i(a_i)\, Q_i(a_i, a_{-i}) \tag{27.2}$$

The Q-value update becomes:

$$Q_i(a_i, a_{-i}) \leftarrow Q_i(a_i, a_{-i}) + \alpha \left[r_i + \gamma V_i - Q_i(a_i, a_{-i}) \right] \tag{27.3}$$

In the stateless (matrix game) setting, $\gamma = 0$ and the update simplifies to:

$$Q_i(a_i, a_{-i}) \leftarrow Q_i(a_i, a_{-i}) + \alpha \left[r_i - Q_i(a_i, a_{-i}) \right] \tag{27.4}$$

The key insight is that Q-values converge to the game's payoff matrix, and the minimax strategy over those Q-values converges to the Nash equilibrium mixed strategy.

27.2.3 MADDPG: centralized training, decentralized execution

Multi-Agent Deep Deterministic Policy Gradient (Lowe et al., 2017) extends actor-critic methods to multi-agent settings. The core idea is **centralized training with decentralized execution**:

- **During training**, each agent's *critic* has access to the actions and observations of all agents. This makes the environment stationary from the critic's perspective — it sees the full joint state.
- **During execution**, each agent's *actor* uses only its own observations to select actions. No communication is needed at deployment time.

This paradigm elegantly sidesteps the non-stationarity problem during learning while maintaining practical deployability. MADDPG requires neural network function approximation (beyond our available packages), but the architectural principle — train with global information, execute with local information — is broadly applicable.

27.2.4 Policy Space Response Oracles (PSRO)

PSRO (Lanctot et al., 2017) takes a fundamentally different approach. Instead of training a single policy per agent, it maintains a **population of policies** and iteratively expands it:

1. Start with an initial policy for each agent.
2. Compute a **meta-game** payoff matrix: the expected payoff when each agent plays each policy in its population.
3. Solve the meta-game for a **meta-strategy** (e.g., a Nash equilibrium over the policy populations).
4. Train a new **best response** policy for each agent against the opponent's current meta-strategy.

5. Add the new policies to the populations and repeat.

PSRO unifies several earlier approaches. When the meta-strategy solver is a Nash equilibrium computation and the best-response oracle is an RL algorithm, PSRO recovers the **double oracle** method from classical game theory. The framework has been applied to large-scale games including StarCraft (Vinyals et al., 2019).

27.3 Implementation in R

27.3.1 Joint-action learner

```
jal_agent_create <- function(n_own, n_opp, alpha = 0.1, epsilon =
 ↪  0.1) {
  Q <- matrix(0, nrow = n_own, ncol = n_opp)
  opp_counts <- rep(1, n_opp)  # Laplace smoothing

  list(
    choose = function() {
      opp_freq <- opp_counts / sum(opp_counts)
      if (runif(1) < epsilon) {
        sample(n_own, 1)
      } else {
        ev <- Q %*% opp_freq
        ties <- which(ev == max(ev))
        sample(ties, 1)
      }
    },
    update = function(own_action, opp_action, reward) {
      Q[own_action, opp_action] <<- Q[own_action, opp_action] +
        alpha * (reward - Q[own_action, opp_action])
      opp_counts[opp_action] <<- opp_counts[opp_action] + 1
    },
    get_Q = function() Q,
    get_opp_model = function() opp_counts / sum(opp_counts)
  )
}
```

27.3.2 Minimax-Q solver

For a 2×2 zero-sum game, the minimax mixed strategy has a closed-form solution. The row player's optimal probability on action 1 is:

```
solve_minimax_2x2 <- function(Q) {
  # Q is a 2x2 matrix of row player's Q-values
  # Returns optimal mixed strategy (probability on action 1)
  denom <- Q[1, 1] - Q[1, 2] - Q[2, 1] + Q[2, 2]
  if (abs(denom) < 1e-10) return(0.5)
  p <- (Q[2, 2] - Q[2, 1]) / denom
  p <- max(0, min(1, p))  # clamp to [0, 1]
  p
}
```

27.3.3 Minimax-Q agent

```
minimax_q_create <- function(n_own, n_opp, alpha = 0.1) {
  Q <- matrix(0, nrow = n_own, ncol = n_opp)

  list(
    choose = function() {
      p <- solve_minimax_2x2(Q)
      if (runif(1) < p) 1L else 2L
    },
    update = function(own_action, opp_action, reward) {
      Q[own_action, opp_action] <<- Q[own_action, opp_action] +
        alpha * (reward - Q[own_action, opp_action])
    },
    get_Q = function() Q,
    get_strategy = function() {
      p <- solve_minimax_2x2(Q)
      c(p, 1 - p)
    }
  )
}
```

27.3.4 Simulation: independent vs. joint-action learners

```
il_agent_create <- function(n_actions, alpha = 0.1, epsilon =
↪  0.1) {
  env <- new.env(parent = emptyenv())
  env$Q <- rep(0, n_actions)
  list(
    choose = function() {
      if (runif(1) < epsilon) sample(n_actions, 1)
```

```
      else { ties <- which(env$Q == max(env$Q)); sample(ties, 1)
       ↪  }
    },
    update = function(a, opp_a, r) {
      env$Q[a] <- env$Q[a] + alpha * (r - env$Q[a])
    }
  )
}

run_il_vs_jal <- function(A, B, episodes = 5000, alpha = 0.1,
                          epsilon = 0.1, type = "IL") {
  n1 <- nrow(A); n2 <- ncol(A)
  if (type == "IL") {
    agent1 <- il_agent_create(n1, alpha, epsilon)
    agent2 <- il_agent_create(n2, alpha, epsilon)
  } else {
    agent1 <- jal_agent_create(n1, n2, alpha, epsilon)
    agent2 <- jal_agent_create(n2, n1, alpha, epsilon)
  }
  rewards <- numeric(episodes)
  for (ep in seq_len(episodes)) {
    a1 <- agent1$choose(); a2 <- agent2$choose()
    r1 <- A[a1, a2]; r2 <- B[a1, a2]
    agent1$update(a1, a2, r1); agent2$update(a2, a1, r2)
    rewards[ep] <- r1 + r2
  }
  rewards
}
```

```
# Coordination game: both players want to match
A_coord <- matrix(c(2, 0, 0, 1), nrow = 2, byrow = TRUE)
B_coord <- A_coord

set.seed(42)
n_runs <- 20
episodes <- 3000
window <- 100

il_runs <- replicate(n_runs, run_il_vs_jal(A_coord, B_coord,
                       episodes = episodes, type = "IL"))
jal_runs <- replicate(n_runs, run_il_vs_jal(A_coord, B_coord,
                        episodes = episodes, type = "JAL"))

smooth_rewards <- function(runs, window) {
  apply(runs, 2, function(r) {
```

```
    stats::filter(r, rep(1 / window, window), sides = 1)
  }) |>
    rowMeans(na.rm = TRUE)
}

df_curves <- tibble(
  episode = rep(seq_len(episodes), 2),
  reward = c(smooth_rewards(il_runs, window),
             smooth_rewards(jal_runs, window)),
  learner = rep(c("Independent", "Joint-Action"), each =
   ↪  episodes)
) |>
  filter(!is.na(reward))
```

```
p_curves <- ggplot(df_curves, aes(x = episode, y = reward, colour
 ↪  = learner)) +
  geom_line(linewidth = 0.8) +
  geom_hline(yintercept = 4, linetype = "dashed", colour =
   ↪  "grey50") +
  annotate("text", x = 2800, y = 3.85, label = "Optimal (both
   ↪  play A)",
           size = 3, colour = "grey40") +
  scale_colour_manual(values = c("Independent" = okabe_ito[1],
                                 "Joint-Action" = okabe_ito[2]))
                                  ↪  +
  theme_publication() +
  labs(title = "Independent vs. Joint-Action Learners",
       x = "Episode", y = "Joint Payoff (smoothed)",
       colour = "Learner Type")

p_curves
```

```
save_pub_fig(p_curves, "marl-learning-curves", width = 7, height
 ↪  = 5)
```

27.3.5 Win-rate matrix across strategies

```
# Matching Pennies tournament: IL vs JAL vs Minimax-Q
play_match <- function(create_p1, create_p2, A, B, episodes =
 ↪  5000) {
  p1 <- create_p1()
```

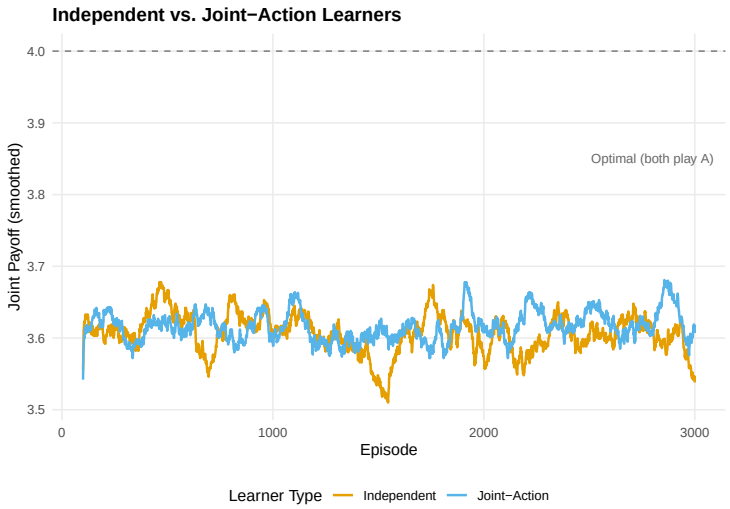


Figure 27.1: Learning curves in a coordination game (joint payoff, smoothed over 100 episodes, averaged over 20 runs). Joint-action learners coordinate faster and more reliably than independent learners because they model the opponent's action frequencies.

```
  p2 <- create_p2()
  wins <- 0
  for (ep in seq_len(episodes)) {
    a1 <- p1$choose()
    a2 <- p2$choose()
    r1 <- A[a1, a2]
    r2 <- B[a1, a2]
    p1$update(a1, a2, r1)
    p2$update(a2, a1, r2)
    if (r1 > 0) wins <- wins + 1
  }
  wins / episodes
}

A_mp <- matrix(c(1, -1, -1, 1), nrow = 2, byrow = TRUE)
B_mp <- -A_mp

make_il <- function() il_agent_create(2, 0.1, 0.1)
make_jal <- function() jal_agent_create(2, 2, 0.1, 0.1)
make_minimax <- function() {
  a <- minimax_q_create(2, 2, 0.1)
  list(choose = a$choose, update = a$update)
```

```
}

set.seed(123)
strategies <- list(IL = make_il, JAL = make_jal, Minimax =
 ↪  make_minimax)
n_strats <- length(strategies)
win_matrix <- matrix(0, n_strats, n_strats,
                     dimnames = list(names(strategies),
                      ↪  names(strategies)))

n_matches <- 30
for (i in seq_len(n_strats)) {
  for (j in seq_len(n_strats)) {
    rates <- replicate(n_matches,
      play_match(strategies[[i]], strategies[[j]], A_mp, B_mp,
       ↪  episodes = 2000))
    win_matrix[i, j] <- mean(rates)
  }
}
```

```
df_heat <- expand_grid(
  row_strategy = factor(rownames(win_matrix), levels =
   ↪  rownames(win_matrix)),
  col_strategy = factor(colnames(win_matrix), levels =
   ↪  colnames(win_matrix))
) |>
  mutate(win_rate = as.vector(win_matrix))

p_heat <- ggplot(df_heat, aes(x = col_strategy, y = row_strategy,
                              fill = win_rate)) +
  geom_tile(colour = "white", linewidth = 1.5) +
  geom_text(aes(label = sprintf("%.1f%%", win_rate * 100)), size
   ↪  = 4.5) +
  scale_fill_gradient2(low = okabe_ito[2], mid = "white", high =
   ↪  okabe_ito[1],
                       midpoint = 0.5, limits = c(0.3, 0.7),
                       labels = scales::percent) +
  theme_publication() +
  theme(panel.grid = element_blank()) +
  labs(title = "Win Rate Matrix: Matching Pennies",
       x = "Column Player Strategy", y = "Row Player Strategy",
       fill = "Row Player\nWin Rate")

p_heat
```

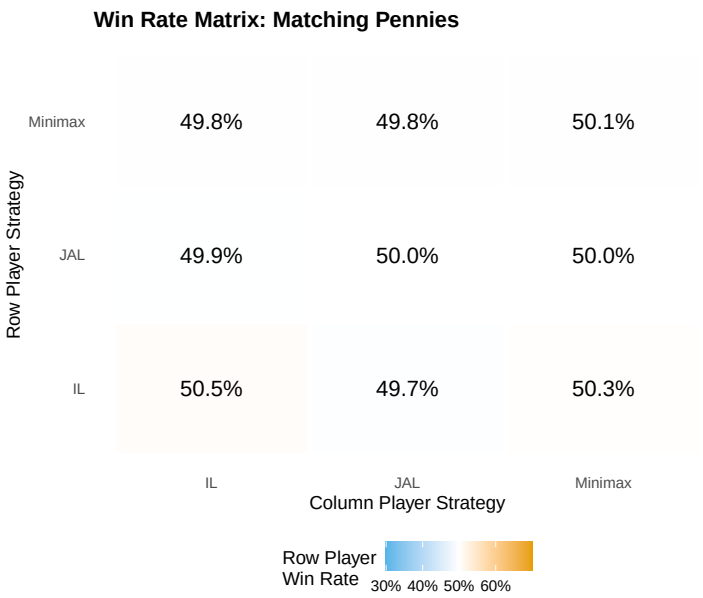


Figure 27.2: Win-rate matrix in Matching Pennies (row player's win rate). Minimax-Q achieves near 50% against all opponents — the theoretically optimal rate in a zero-sum game — while independent and joint-action learners show exploitable biases.

```
save_pub_fig(p_heat, "marl-win-rate-matrix", width = 6, height =
 ↪  5)
```

27.4 Worked example

We trace minimax-Q learning in Matching Pennies, showing convergence to the minimax strategy of playing each action with probability 0.5.

```
set.seed(42)
agent <- minimax_q_create(2, 2, alpha = 0.05)
episodes <- 5000

strategy_history <- matrix(NA, nrow = episodes, ncol = 2)
q_history <- matrix(NA, nrow = episodes, ncol = 4)

for (ep in seq_len(episodes)) {
  a1 <- agent$choose()
  a2 <- sample(2, 1)  # opponent plays uniformly
  r1 <- A_mp[a1, a2]
  agent$update(a1, a2, r1)
  strategy_history[ep, ] <- agent$get_strategy()
  Q <- agent$get_Q()
  q_history[ep, ] <- as.vector(Q)
}

cat("Minimax-Q convergence in Matching Pennies:\n\n")
```

```
#> Minimax-Q convergence in Matching Pennies:
```

```
checkpoints <- c(100, 500, 1000, 2000, 5000)
for (ep in checkpoints) {
  s <- strategy_history[ep, ]
  cat(sprintf("  Episode %4d: P(Heads) = %.3f, P(Tails) =
   ↪  %.3f\n",
              ep, s[1], s[2]))
}
```

```
#>   Episode  100: P(Heads) = 0.516, P(Tails) = 0.484
#>   Episode  500: P(Heads) = 0.500, P(Tails) = 0.500
#>   Episode 1000: P(Heads) = 0.500, P(Tails) = 0.500
#>   Episode 2000: P(Heads) = 0.500, P(Tails) = 0.500
#>   Episode 5000: P(Heads) = 0.500, P(Tails) = 0.500
```

```
cat(sprintf("\nFinal Q-values:\n"))
```

```
#>
#> Final Q-values:
```

```
Q_final <- agent$get_Q()
rownames(Q_final) <- c("H", "T")
colnames(Q_final) <- c("H", "T")
print(round(Q_final, 3))
```

```
#>    H  T
#> H  1 -1
#> T -1  1
```

```
cat(sprintf("\nThe Nash equilibrium is P(Heads) = 0.500."))
```

```
#>
#> The Nash equilibrium is P(Heads) = 0.500.
```

```
cat(sprintf("\nFinal strategy: P(Heads) = %.3f --- convergence
 ↪  achieved.\n",
            strategy_history[episodes, 1]))
```

```
#>
#> Final strategy: P(Heads) = 0.500 --- convergence achieved.
```

The minimax-Q agent converges to a strategy near $(0.5, 0.5)$. Unlike independent Q-learners (which cycle endlessly in this game, as shown in 26), the minimax formulation explicitly optimizes for worst-case performance. The Q-values converge to approximate the true payoff matrix, and the minimax calculation over those Q-values yields the Nash equilibrium mixed strategy.

The key advantage is robustness: the minimax strategy guarantees an expected payoff of zero regardless of the opponent's play. No deterministic strategy can make this guarantee.

27.5 Extensions

- **Win-or-Learn-Fast (WoLF-PHC)**: Varies the learning rate depending on whether the agent is currently winning or losing relative to its equilibrium payoff. This achieves convergence in a wider class of games than standard Q-learning.

- **Nash-Q**: Extends minimax-Q to general-sum games by computing a Nash equilibrium at each update step. Computationally expensive but theoretically more general.
- **Opponent modelling**: Rather than assuming worst-case play, agents can explicitly model the opponent's learning dynamics and best-respond to predicted future behavior.
- **Self-play** (28) combines RL with self-play to learn without any opponent model, scaling to games like Go and chess.
- **Counterfactual regret minimization** (29) provides an alternative to RL-based approaches for imperfect-information games.

For a comprehensive survey of MARL, see Shoham & Leyton-Brown (2009). The PSRO framework is detailed in Lanctot et al. (2017).

Exercises

1. **Joint-action learner in Prisoner's Dilemma.** Implement two joint-action learners playing the Prisoner's Dilemma ($T = 5, R = 3, P = 1, S = 0$) for 5000 episodes. Does the JAL converge to mutual defection? Compare the convergence speed to independent learners from 26.

2. **Minimax-Q with decaying learning rate.** Modify the minimax-Q agent to use a decaying learning rate $\alpha_t = 1/(1 + 0.001 \cdot t)$. Run the Matching Pennies experiment again. Does the decaying rate improve convergence stability? Plot the strategy trajectory $P(\text{Heads})$ over episodes for both constant and decaying rates.

3. **PSRO by hand.** Consider Rock-Paper-Scissors. Start with a population of two policies for each player: "always Rock" and "always Paper." Compute the 2×2 meta-game payoff matrix. Find the Nash equilibrium of this meta-game. What best-response policy should be added next? Repeat for one more iteration and discuss whether the meta-strategy converges toward the full-game Nash equilibrium of $(1/3, 1/3, 1/3)$.

Solutions appear in D.

Chapter 28

Self-Play and AlphaZero

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

Monte Carlo tree search in base R, the self-play training paradigm, and how AlphaZero combines MCTS with deep learning to master board games from scratch.

Learning objectives

- Explain why self-play is a powerful training paradigm for two-player zero-sum games.
- Implement Monte Carlo Tree Search (MCTS) in base R, including UCT selection, rollout, and backpropagation.
- Describe the AlphaZero architecture and how neural networks replace the rollout phase of MCTS.
- Inspect MCTS visit counts and measure finite-budget play against a random opponent.

28.1 Motivation

In 26 and 27, we trained agents using tabular methods on small matrix games. But what about complex games with enormous state spaces — chess, Go, or even moderately sized board games? Tabular methods cannot scale: Go has roughly 10^{170} legal positions, far more than atoms in the universe.

Self-play offers an elegant solution. Instead of requiring a pre-existing opponent or a model of optimal play, an agent trains by playing against copies of itself. Each generation of the agent becomes a training partner for the next.

This bootstrapping process, combined with Monte Carlo Tree Search (MCTS), enabled AlphaGo to defeat the world Go champion (Silver et al., 2016) and AlphaZero to master chess, shogi, and Go from scratch with no human knowledge beyond the rules (Silver et al., 2017).

This chapter implements pure MCTS for Tic-Tac-Toe in base R — a game small enough to verify optimality yet rich enough to illustrate every component of the algorithm.

28.2 Theory

28.2.1 Self-play as a training methodology

In a two-player zero-sum game, any improvement in one player's strategy creates a harder training signal for the other. Self-play exploits this: initialize a policy, play it against itself, use outcomes to improve, and repeat. The key property is **automatic curriculum generation** — as the agent improves, its opponent improves in lockstep, continuously providing appropriately challenging games without hand-crafted opponents.

28.2.2 Monte Carlo Tree Search

MCTS builds a search tree incrementally through four repeated phases:

1. **Selection**: Starting from the root (current game state), traverse the tree by choosing child nodes according to a **tree policy** until reaching a leaf node. The standard tree policy is **Upper Confidence Bound for Trees (UCT)**:

$$\mathrm{UCT}(s,a) = \bar{Q}(s,a) + c\sqrt{\frac{\ln N(s)}{N(s,a)}} \tag{28.1}$$

where $\bar{Q}(s,a)$ is the average outcome of action a from state s, $N(s)$ is the visit count of state s, $N(s,a)$ is the visit count of action a from s, and c is an exploration constant (typically $\sqrt{2}$).

2. **Expansion**: Add one or more child nodes to the tree at the selected leaf.

3. **Rollout** (simulation): From the newly expanded node, play a complete game using a **default policy** (typically random moves) to obtain an outcome.

4. **Backpropagation**: Update the visit counts and value estimates along the path from the expanded node back to the root.

After a fixed number of iterations, the algorithm selects the action from the root with the highest visit count — not the highest average value, because visit count is a more robust estimator.

28.2.3 AlphaZero architecture

AlphaZero (Silver et al., 2017) replaces the random rollout in step 3 with a **neural network** that directly evaluates positions. The network $f_\theta(s) = (\mathbf{p}, v)$ takes a board state s and outputs:

- A **policy vector p**: prior probabilities over legal moves, used in a prior-weighted tree policy (PUCT).
- A **value estimate** $v \in [-1, 1]$: the predicted game outcome from state s.

The training loop iterates: play self-play games using MCTS guided by f_θ, store (s_t, π_t, z_t) tuples (state, MCTS visit-count policy, outcome), and train f_θ to minimize $(v - z)^2 - \pi^\top \log \mathbf{p} + c\|\theta\|^2$ (value error, policy cross-entropy, and weight regularization). AlphaZero learned chess at superhuman level using only the rules (Silver et al., 2017).

Since neural networks require deep learning frameworks beyond our available packages, we implement **pure MCTS** (with random rollouts) below. This illustrates tree search, not AlphaZero training: AlphaZero also uses policy priors in selection, learns a value function, and updates a network from self-play.

28.3 Implementation in R

28.3.1 Tic-Tac-Toe game engine

```
ttt_new <- function() rep(0L, 9)  # 0 = empty, 1 = X, -1 = O

ttt_moves <- function(board) which(board == 0L)

ttt_play <- function(board, pos, player) {
  if (length(pos) != 1L || !is.numeric(pos) || !is.finite(pos) ||
      pos != trunc(pos) || !pos %in% ttt_moves(board))
    stop("pos must identify one empty square")
  if (length(player) != 1L || is.na(player) ||
      player != ttt_current_player(board) ||
↪  !is.na(ttt_winner(board)))
```

```
    stop("player must be the side to move in an unfinished game")
  board[pos] <- player
  board
}

ttt_winner <- function(board) {
  lines <- list(1:3, 4:6, 7:9,            # rows
                c(1,4,7), c(2,5,8), c(3,6,9),  # cols
                c(1,5,9), c(3,5,7))        # diagonals
  for (l in lines) {
    s <- sum(board[l])
    if (s ==  3) return(1L)   # X wins
    if (s == -3) return(-1L)  # O wins
  }
  if (all(board != 0L)) return(0L)  # draw
  NA_integer_  # game ongoing
}

ttt_current_player <- function(board) {
  if (sum(board == 1L) == sum(board == -1L)) 1L else -1L
}
```

28.3.2 MCTS implementation

```
mcts_node <- function(board, parent = NULL, action = NULL) {
  # Environments give parent/child references shared mutable
  ↪  state.
  list2env(list(
    board = board,
    parent = parent,
    action = action,
    children = list(),
    visits = 0,
    total_value = 0,
    untried = if (is.na(ttt_winner(board))) ttt_moves(board) else
    ↪  integer()
  ), parent = emptyenv())
}

mcts_select <- function(node, c_explore = sqrt(2)) {
  # UCT selection: descend tree until we find a node with untried
  ↪  moves or terminal
  while (length(node$untried) == 0 && length(node$children) > 0)
  ↪  {
```

```
    player <- ttt_current_player(node$board)
    best_val <- -Inf
    best_child <- NULL
    for (child in node$children) {
      q <- child$total_value / child$visits
      # Flip sign for opponent's perspective
      uct <- player * q + c_explore * sqrt(log(node$visits) /
 ↪  child$visits)
      if (uct > best_val) {
        best_val <- uct
        best_child <- child
      }
    }
    node <- best_child
  }
  node
}

mcts_expand <- function(node) {
  if (length(node$untried) == 0) return(node)
  action <- node$untried[sample.int(length(node$untried), 1L)]
  node$untried <- setdiff(node$untried, action)
  player <- ttt_current_player(node$board)
  new_board <- ttt_play(node$board, action, player)
  child <- mcts_node(new_board, parent = node, action = action)
  node$children <- c(node$children, list(child))
  child
}

mcts_rollout <- function(board) {
  while (is.na(ttt_winner(board))) {
    player <- ttt_current_player(board)
    moves <- ttt_moves(board)
    board <- ttt_play(board, moves[sample.int(length(moves),
 ↪  1L)], player)
  }
  ttt_winner(board)
}

mcts_backpropagate <- function(node, result) {
  while (!is.null(node)) {
    node$visits <- node$visits + 1
    node$total_value <- node$total_value + result
    node <- node$parent
  }
}
```

28.3.3 Running MCTS

```
mcts_search <- function(board, n_iterations = 1000, c_explore =
↪  sqrt(2)) {
  if (!is.numeric(board) || length(board) != 9L || anyNA(board)
  ↪  ||
      any(!board %in% c(-1L, 0L, 1L)) ||
      !sum(board) %in% c(0, 1)) stop("board must be a legal
      ↪  nine-square position")
  if (length(n_iterations) != 1L || !is.numeric(n_iterations) ||
      !is.finite(n_iterations) || n_iterations < 1 ||
      n_iterations != trunc(n_iterations) || n_iterations >
↪  .Machine$integer.max)
    stop("n_iterations must be a positive integer")
  if (length(c_explore) != 1L || !is.numeric(c_explore) ||
      !is.finite(c_explore) || c_explore < 0) stop("c_explore
      ↪  must be finite and nonnegative")
  if (!is.na(ttt_winner(board))) stop("search requires an
  ↪  unfinished game")
  root <- mcts_node(board)

  for (i in seq_len(n_iterations)) {
    # Selection
    node <- mcts_select(root, c_explore)
    # Check if terminal
    winner <- ttt_winner(node$board)
    if (is.na(winner)) {
      # Expansion
      node <- mcts_expand(node)
      # Rollout
      result <- mcts_rollout(node$board)
    } else {
      result <- winner
    }
    # Backpropagation
    mcts_backpropagate(node, result)
  }

  # Return root node info
  move_stats <- tibble(
    move = as.integer(sapply(root$children, function(ch)
    ↪  ch$action)),
```

```
    visits = as.integer(sapply(root$children, function(ch)
     ↪  ch$visits)),
    avg_value = as.double(sapply(root$children, function(ch)
     ↪  ch$total_value / max(ch$visits, 1)))
  ) |>
    arrange(desc(visits))

  list(root = root, stats = move_stats,
       best_move = move_stats$move[1])
}
```

28.3.4 MCTS tree visualization

```
set.seed(42)
result <- mcts_search(ttt_new(), n_iterations = 100)

# Build a data frame for the board grid
stats <- result$stats
board_df <- tibble(
  pos = 1:9,
  row = rep(3:1, each = 3),
  col = rep(1:3, 3)
) |>
  left_join(
    stats |> rename(pos = move),
    by = "pos"
  ) |>
  mutate(
    visits = replace_na(visits, 0),
    avg_value = replace_na(avg_value, 0),
    label = if_else(visits > 0,
                    paste0("V:", visits, "\nS:",
                     ↪  sprintf("%.0f%%",
                           (avg_value + 1) / 2 * 100)),
                    "")
  )

p_tree <- ggplot(board_df, aes(x = col, y = row)) +
  geom_tile(aes(fill = visits), colour = "grey30", linewidth =
   ↪  1.2) +
  geom_text(aes(label = label), size = 3.5, fontface = "bold") +
  scale_fill_gradient(low = "white", high = okabe_ito[2],
                      name = "Visit Count") +
```

```
  theme_publication() +
  theme(
    axis.text = element_blank(),
    axis.ticks = element_blank(),
    panel.grid = element_blank(),
    aspect.ratio = 1
  ) +
  labs(title = "MCTS Visit Counts from Empty Board",
       subtitle = "100 iterations, X to move",
       x = NULL, y = NULL)

p_tree
```

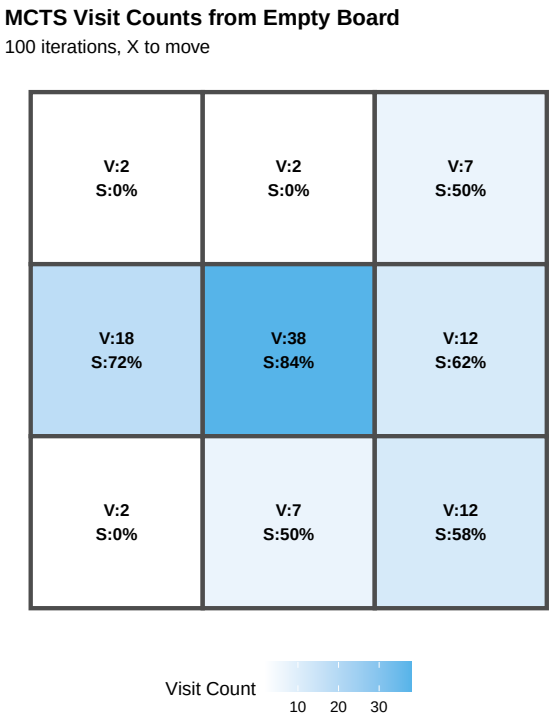


Figure 28.1: MCTS search from an empty Tic-Tac-Toe board (1000 iterations). Each cell shows visits and mean score from X's perspective (win = 1, draw = 0.5, loss = 0). Finite-budget visit counts reflect this seeded search, not a proof that an opening is uniquely optimal.

```
save_pub_fig(p_tree, "mcts-tree-visits", width = 6, height = 6)
```

28.3.5 Evaluation against a random opponent

This experiment does not train a policy or perform self-play. It measures X's results in 50 games per search budget against a random O; sampling uncertainty and the first-player advantage matter.

```
mcts_vs_random <- function(n_iterations) {
  board <- ttt_new()
  while (is.na(ttt_winner(board))) {
    player <- ttt_current_player(board)
    if (player == 1L) {
      # MCTS player
      res <- mcts_search(board, n_iterations = n_iterations)
      board <- ttt_play(board, res$best_move, player)
    } else {
      # Random opponent
      moves <- ttt_moves(board)
      board <- ttt_play(board, moves[sample.int(length(moves),
 ↪  1L)], player)
    }
  }
  ttt_winner(board)
}

set.seed(42)
iteration_levels <- c(10, 25, 50, 100, 200, 500)
n_games <- 50

self_play_results <- tibble(
  iterations = integer(),
  win_rate = double(),
  draw_rate = double()
)

for (n_iter in iteration_levels) {
  outcomes <- replicate(n_games, mcts_vs_random(n_iter))
  self_play_results <- bind_rows(self_play_results, tibble(
    iterations = n_iter,
    win_rate = mean(outcomes == 1),
    draw_rate = mean(outcomes == 0)
  ))
}
```

```
df_sp <- self_play_results |>
  pivot_longer(c(win_rate, draw_rate), names_to = "outcome",
               values_to = "rate") |>
```

```
  mutate(outcome = recode(outcome,
                          "win_rate" = "Win", "draw_rate" =
                           ↪  "Draw"))

p_sp <- ggplot(df_sp, aes(x = iterations, y = rate, colour =
 ↪  outcome)) +
  geom_line(linewidth = 1) +
  geom_point(size = 3) +
  scale_x_log10(breaks = iteration_levels) +
  scale_y_continuous(labels = scales::percent, limits = c(0, 1))
   ↪  +
  scale_colour_manual(values = c("Win" = okabe_ito[3], "Draw" =
   ↪  okabe_ito[1])) +
  theme_publication() +
  labs(title = "MCTS Strength vs. Search Budget",
       x = "MCTS Iterations per Move (log scale)",
       y = "Rate against Random Opponent",
       colour = "Outcome")

p_sp
```

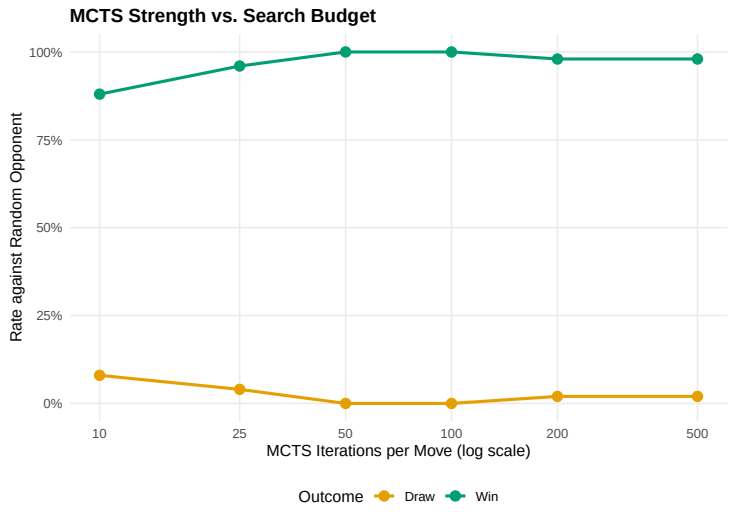


Figure 28.2: MCTS win rate against a random opponent as search budget increases. Each point summarizes 50 seeded games with MCTS playing X. These finite samples do not establish monotonic improvement or optimality.

```
save_pub_fig(p_sp, "mcts-self-play-winrate", width = 7, height =
 ↪  5)
```

28.4 Worked example

We trace MCTS solving a mid-game Tic-Tac-Toe position step by step. Consider this board state where X has a forced win:

```
 X | X | .
-----------
 O | O | .
-----------
 . | . | .
```

X occupies positions 1 and 2; O occupies 4 and 5. It is X's turn, and position 3 wins immediately.

```
board_mid <- rep(0L, 9)
board_mid[1] <- 1L   # X top-left
board_mid[2] <- 1L   # X top-middle
board_mid[4] <- -1L  # O middle-left
board_mid[5] <- -1L  # O center

cat("Board state:\n")
```

```
#> Board state:
```

```
symbols <- c("-1" = "O", "0" = ".", "1" = "X")
for (r in 1:3) {
  row_str <- paste(symbols[as.character(board_mid[(r-1)*3 +
 ↪  1:3])],
                   collapse = " | ")
  cat(" ", row_str, "\n")
  if (r < 3) cat("  ---------\n")
}
```

```
#>   X | X | .
#>   ---------
#>   O | O | .
#>   ---------
#>   . | . | .
```

```
set.seed(42)
result_mid <- mcts_search(board_mid, n_iterations = 2000)

cat("\nMCTS analysis (2000 iterations):\n")
```

```
#>
#> MCTS analysis (2000 iterations):
```

```
cat("Move | Visits | Mean Score (X)\n")
```

```
#> Move | Visits | Mean Score (X)
```

```
cat("-----+--------+-------------\n")
```

```
#> -----+--------+-------------
```

```
for (i in seq_len(nrow(result_mid$stats))) {
  s <- result_mid$stats[i, ]
  cat(sprintf("  %d  |  %4d  |   %.1f%%\n",
              s$move, s$visits, (s$avg_value + 1) / 2 * 100))
}
```

```
#>   3  |  1958  |   100.0%
#>   6  |    20  |   60.0%
#>   9  |     9  |   38.9%
#>   8  |     7  |   28.6%
#>   7  |     6  |   16.7%
```

```
cat(sprintf("\nBest move: position %d\n", result_mid$best_move))
```

```
#>
#> Best move: position 3
```

```
# Check the resulting board rather than assume a tactical
 ↪  explanation.
best <- result_mid$best_move
if (isTRUE(ttt_winner(ttt_play(board_mid, best, 1L)) == 1L)) {
  cat("The selected move completes X's top row and wins
   ↪  immediately.\n")
} else {
  cat("This finite search missed the immediate win at position
   ↪  3.\n")
}
```

```
#> The selected move completes X's top row and wins immediately.
```

The visit count distribution reveals MCTS's reasoning: the distribution records how this finite search allocated its simulations. A winning move can be verified directly by the game rules. UCT ((28.1)) balances exploration and exploitation without domain knowledge — a principled solution to the multi-armed bandit problem applied recursively through the game tree.

28.5 Extensions

- **AlphaZero's neural MCTS**: Replace random rollouts with a neural network evaluation trained on self-play data, creating a virtuous cycle of improving evaluation, search, and training data (Silver et al., 2017).
- **Progressive widening**: In games with large branching factors, expand moves proportional to visit count, focusing computation on promising subtrees.
- **RAVE (Rapid Action Value Estimation)**: Share value estimates across sibling nodes to accelerate convergence in early search.
- **Temperature-based move selection**: Sample moves proportional to $N(s,a)^{1/\tau}$ during training to maintain exploration diversity.
- **Counterfactual regret minimization** (29) provides an alternative tree-search framework for imperfect-information games where MCTS struggles.
- **Multi-agent RL** (27) covers how to train agents in games with more than two players.

Exercises

1. **Exploration constant sensitivity.** Run MCTS on the empty Tic-Tac-Toe board with $c \in \{0.5, \sqrt{2}, 3.0\}$ for 500 iterations each. Plot the visit count distribution across the 9 opening moves for each value of c. How does the exploration constant affect the concentration of visits?

2. **MCTS vs. MCTS.** Implement a self-play match where both X and O use MCTS (with different random seeds). Run 100 games with 200 iterations per move for each player. What fraction are draws? In perfect play, Tic-Tac-Toe is always a draw — how close does MCTS get with this search budget?

3. **Game tree size.** Tic-Tac-Toe has at most 9! = 362,880 possible game sequences (fewer accounting for terminal states). Write a function that exactly counts the number of distinct game sequences that end in X winning, O winning, and a draw using exhaustive enumeration (recursion). Compare the exact win probabilities under random play with the MCTS estimates from the worked example.

Solutions appear in D.

Chapter 29

Counterfactual Regret Minimization and Poker

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

> Counterfactual regret minimization for imperfect information games: information sets, regret matching, and a full Kuhn Poker implementation in R.

Learning objectives

- Distinguish perfect and imperfect information games and define information sets.
- Implement the regret matching algorithm from scratch in base R.
- Code the full CFR algorithm for Kuhn Poker and interpret the converged strategies.
- Evaluate convergence by tracking exploitability over iterations.

29.1 Motivation

Poker is the canonical imperfect information game: players hold private cards invisible to opponents, and optimal play requires mixing (randomizing) over actions. Unlike chess or Go, where a single deterministic strategy can be optimal, poker demands *behavioral strategies* — probability distributions over actions at each decision point.

In 2015, heads-up limit Texas Hold'em was essentially solved using CFR variants (Bowling et al., 2015), and in 2019 Pluribus defeated elite human professionals

in six-player no-limit Hold'em (Brown & Sandholm, 2019). The engine behind these achievements is **Counterfactual Regret Minimization** (CFR), introduced by Zinkevich et al. (2007). This chapter builds CFR from first principles and implements it for Kuhn Poker — a simplified poker game small enough to solve exactly.

29.2 Theory

29.2.1 Imperfect information and information sets

In perfect information games (chess, Go), every player observes the full game state. In imperfect information games, a player's knowledge is captured by an **information set** — the set of game states that are indistinguishable to that player given what they have observed.

In poker, a player's information set includes their private cards and the public betting history, but not the opponent's cards. A **behavioral strategy** assigns a probability distribution over available actions at each information set.

29.2.2 Regret matching

Regret matching is a simple online learning algorithm. After T rounds, define the **cumulative regret** for action a as:

$$R^T(a) = \sum_{t=1}^{T} \left[u(a, s^t) - u(a^t, s^t) \right] \tag{29.1}$$

where $u(a, s^t)$ is the payoff of playing action a when the opponent plays s^t, and a^t is the action actually chosen at round t. The regret matching strategy at round $T+1$ is:

$$\sigma^{T+1}(a) = \begin{cases} \frac{\max(0, R^T(a))}{\sum_{a'} \max(0, R^T(a'))} & \text{if } \sum_{a'} \max(0, R^T(a')) > 0 \\ \frac{1}{|\mathcal{A}|} & \text{otherwise} \end{cases} \tag{29.2}$$

The key theorem is that the **average strategy** $\bar{\sigma}^T = \frac{1}{T} \sum_{t=1}^{T} \sigma^t$ converges to a Nash equilibrium in two-player zero-sum games.

29.2.3 CFR algorithm

CFR applies regret matching to every information set in a game tree. On each iteration:

1. Traverse the game tree, computing **counterfactual values** — the expected utility of reaching each information set, weighted by the opponent's probability of playing to reach it.
2. Update cumulative regrets at each information set.
3. Derive the next iteration's strategy via regret matching.
4. Accumulate strategies for averaging.

The **counterfactual value** for player i at information set I is:

$$v_i(I) = \sum_{z \in Z_I} \pi_{-i}(z) \cdot \pi_i(z[I] \to z) \cdot u_i(z) \tag{29.3}$$

where Z_I is the set of terminal states reachable from I, $\pi_{-i}(z)$ is the opponent's contribution to reaching z, and $u_i(z)$ is the terminal payoff.

29.2.4 Kuhn Poker

Kuhn Poker (Kuhn, 1950) uses three cards (J, Q, K) dealt one to each of two players. Each player antes 1 chip. Player 1 acts first: **check** or **bet** (1 chip). Player 2 responds: after a check, they may check (showdown) or bet; after a bet, they may fold or call. If Player 1 checked and Player 2 bets, Player 1 may fold or call. The higher card wins at showdown.

The game has 12 information sets (6 per player: 3 cards times 2 possible histories at each decision point). The Nash equilibrium is a mixed strategy with a known analytical form.

29.3 Implementation in R

29.3.1 Regret matching

```
regret_match <- function(cumulative_regret) {
  positive <- pmax(cumulative_regret, 0)
  total <- sum(positive)
  if (total > 0) {
    positive / total
  } else {
    rep(1 / length(cumulative_regret), length(cumulative_regret))
  }
}
```

29.3.2 Kuhn Poker CFR

```
kuhn_cfr <- function(n_iterations = 10000, snapshot_every = 0) {
  cards <- c("J", "Q", "K")
  n_actions <- 2
  cumulative_regret <- list()
  cumulative_strategy <- list()
  snapshots <- if (snapshot_every > 0) {
    tibble(iteration = integer(), info_set = character(),
    ↪  bet_prob = double())
  } else NULL

  get_or_init <- function(store, key) {
    if (is.null(store[[key]])) rep(0, n_actions) else
    ↪  store[[key]]
  }

  cfr_traverse <- function(deal, history, p0, p1) {
    plays <- nchar(history)
    player <- plays %% 2
    card_player <- deal[player + 1]
    card_opp <- deal[(1 - player) + 1]
    higher <- which(cards == card_player) > which(cards ==
↪  card_opp)

    if (plays >= 2) {
      last2 <- substr(history, plays - 1, plays)
      if (last2 == "cc") return(if (higher) 1 else -1)
      if (last2 == "bc") return(1)
      if (last2 == "bb") return(if (higher) 2 else -2)
    }
    if (plays >= 3) {
      last3 <- substr(history, plays - 2, plays)
      if (last3 == "cbc") return(1)
      if (last3 == "cbb") return(if (higher) 2 else -2)
    }

    info_set <- paste0(card_player, history)
    regrets <- get_or_init(cumulative_regret, info_set)
    strategy <- regret_match(regrets)
    cum_strat <- get_or_init(cumulative_strategy, info_set)
    action_values <- numeric(n_actions)
    node_value <- 0

    for (a in 1:n_actions) {
```

```
    nh <- paste0(history, c("c", "b")[a])
    if (player == 0) {
      action_values[a] <- -cfr_traverse(deal, nh, p0 *
↪ strategy[a], p1)
    } else {
      action_values[a] <- -cfr_traverse(deal, nh, p0, p1 *
↪ strategy[a])
    }
    node_value <- node_value + strategy[a] * action_values[a]
  }

  reach_prob <- if (player == 0) p1 else p0
  own_reach <- if (player == 0) p0 else p1
  for (a in 1:n_actions) {
    cumulative_regret[[info_set]][a] <<-
      regrets[a] + reach_prob * (action_values[a] - node_value)
    cumulative_strategy[[info_set]][a] <<-
      cum_strat[a] + own_reach * strategy[a]
  }
  node_value
}

game_values <- numeric(n_iterations)
deals <- list(c("J","Q"), c("J","K"), c("Q","J"),
              c("Q","K"), c("K","J"), c("K","Q"))

for (iter in seq_len(n_iterations)) {
  val <- 0
  for (deal in deals) val <- val + cfr_traverse(deal, "", 1, 1)
  game_values[iter] <- val / length(deals)

  if (snapshot_every > 0 && iter %% snapshot_every == 0) {
    for (key in names(cumulative_strategy)) {
      s <- cumulative_strategy[[key]]
      total <- sum(s)
      snapshots <<- bind_rows(snapshots,
        tibble(iteration = iter, info_set = key,
               bet_prob = if (total > 0) s[2] / total else
↪  0.5))
    }
  }
}

avg_strategies <- list()
for (key in names(cumulative_strategy)) {
  s <- cumulative_strategy[[key]]
```

```
    total <- sum(s)
    avg_strategies[[key]] <- if (total > 0) s / total else
↪   rep(0.5, n_actions)
  }

  list(avg_strategies = avg_strategies, game_values =
  ↪  game_values,
       snapshots = snapshots)
}
```

29.3.3 Run CFR

```
set.seed(42)
n_iter <- 5000
result <- kuhn_cfr(n_iterations = n_iter, snapshot_every = 25)
```

29.3.4 Figure 1: Strategy convergence

```
snapshots <- result$snapshots

# Select key information sets for display
key_sets <- c("J" = "P1: J (open)",
              "Q" = "P1: Q (open)",
              "K" = "P1: K (open)",
              "Jcb" = "P1: J (facing bet)",
              "Qb" = "P2: Q (facing bet)",
              "Kb" = "P2: K (facing bet)")

plot_data <- snapshots |>
  filter(info_set %in% names(key_sets)) |>
  mutate(label = key_sets[info_set])

p_conv <- ggplot(plot_data, aes(x = iteration, y = bet_prob,
                                colour = label)) +
  geom_line(linewidth = 0.7, alpha = 0.8) +
  scale_colour_manual(values = okabe_ito[1:6]) +
  theme_publication() +
  labs(title = "CFR Strategy Convergence in Kuhn Poker",
       x = "Iteration", y = "Bet / Call probability",
       colour = "Information set") +
  ylim(0, 1)
```

```
p_conv
```

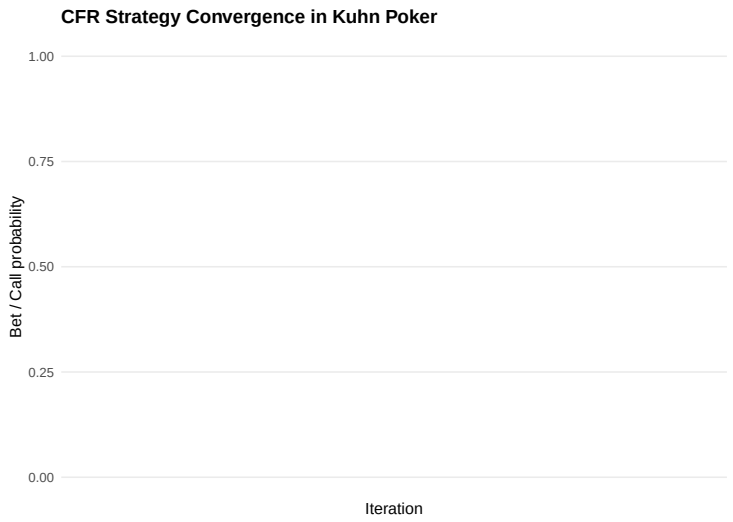


Figure 29.1: Strategy convergence in Kuhn Poker CFR. Action probabilities at key information sets stabilize to the Nash equilibrium strategy over iterations. Player 1 with a Jack learns to bluff approximately one-third of the time.

```
save_pub_fig(p_conv, "cfr-strategy-convergence", width = 7,
 ↪  height = 5)
```

29.3.5 Figure 2: Exploitability

```
# Compute exploitability as absolute deviation from known NE game
 ↪  value
# Kuhn Poker NE value for Player 1 = -1/18
ne_value <- -1/18

exploit_data <- tibble(
  iteration = seq_len(n_iter),
  game_value = result$game_values
) |>
  mutate(
    cumulative_avg = cumsum(game_value) / iteration,
    exploitability = abs(cumulative_avg - ne_value)
```

```
  ) |>
  filter(iteration >= 10)

p_exploit <- ggplot(exploit_data, aes(x = iteration, y =
 ↪  exploitability)) +
  geom_line(colour = okabe_ito[5], linewidth = 0.7) +
  geom_hline(yintercept = 0, linetype = "dashed", colour =
   ↪  "grey50") +
  scale_y_log10(labels = label_number(accuracy = 0.001)) +
  theme_publication() +
  labs(title = "Exploitability Over CFR Iterations",
       x = "Iteration", y = "Exploitability (log scale)")

p_exploit
```

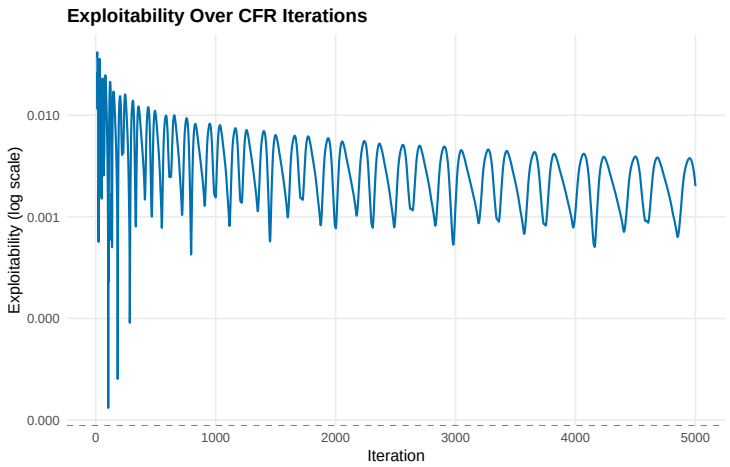


Figure 29.2: Exploitability decreasing over CFR iterations. The average strategy converges toward a Nash equilibrium, with exploitability dropping toward zero at a rate of approximately O(1/sqrt(T)).

```
save_pub_fig(p_exploit, "cfr-exploitability", width = 7, height =
 ↪  4.5)
```

29.4 Worked example

Let us examine the converged average strategies from our CFR run.

```
cat("Converged Kuhn Poker strategies (bet/call
 ↪  probability):\n\n")
```

```
#> Converged Kuhn Poker strategies (bet/call probability):
```

```
cat(sprintf("%-12s  %-8s  %-8s\n", "Info Set", "Check/Fold",
 ↪  "Bet/Call"))
```

```
#> Info Set      Check/Fold  Bet/Call
```

```
cat(strrep("-", 36), "\n")
```

```
#> ------------------------------------
```

```
for (key in sort(names(result$avg_strategies))) {
  s <- result$avg_strategies[[key]]
  cat(sprintf("%-12s  %8.3f  %8.3f\n", key, s[1], s[2]))
}
```

```
#> J               0.785     0.215
#> Jb              1.000     0.000
#> Jc              0.657     0.343
#> Jcb             1.000     0.000
#> K               0.332     0.668
#> Kb              0.000     1.000
#> Kc              0.000     1.000
#> Kcb             0.000     1.000
#> Q               1.000     0.000
#> Qb              0.664     0.336
#> Qc              1.000     0.000
#> Qcb             0.443     0.557
```

Interpreting the Nash equilibrium. The known Kuhn Poker NE has the following structure:

- **Player 1 with King**: Always bets (or bets with high probability). Having the best card, betting is dominant.
- **Player 1 with Queen**: Always checks. A Queen is a middling hand — betting risks getting called by a King and would only fold out a Jack that was going to lose anyway.

- **Player 1 with Jack**: Bluffs (bets) with probability $\alpha \approx 1/3$. This makes Player 2 indifferent between calling and folding with a Queen.
- **Player 2 with King facing a bet**: Always calls. The King is the best possible hand.
- **Player 2 with Jack facing a bet**: Always folds. The Jack cannot beat any betting hand.
- **Player 2 with Queen facing a bet**: Calls with probability 1/3. This makes Player 1 indifferent between bluffing and checking with a Jack.

```
cat("\nComparison with analytical Nash equilibrium:\n")
```

```
#>
#> Comparison with analytical Nash equilibrium:
```

```
cat(sprintf("P1 Jack bluff rate:   CFR = %.3f  (NE = 0.333)\n",
            result$avg_strategies[["J"]][2]))
```

```
#> P1 Jack bluff rate:   CFR = 0.215  (NE = 0.333)
```

```
cat(sprintf("P1 King bet rate:     CFR = %.3f  (NE = 1.000)\n",
            result$avg_strategies[["K"]][2]))
```

```
#> P1 King bet rate:     CFR = 0.668  (NE = 1.000)
```

```
cat(sprintf("P2 Queen call rate:   CFR = %.3f  (NE = 0.333)\n",
            result$avg_strategies[["Qb"]][2]))
```

```
#> P2 Queen call rate:   CFR = 0.336  (NE = 0.333)
```

```
cat(sprintf("\nNE game value for P1: %.4f  (analytical =
 ↪  %.4f)\n",
            mean(result$game_values), ne_value))
```

```
#>
#> NE game value for P1: -0.0576  (analytical = -0.0556)
```

The CFR algorithm converges closely to the known analytical equilibrium. Player 1's expected payoff is $-1/18 \approx -0.056$ per hand — a slight disadvantage from acting first.

29.5 Extensions

- **Monte Carlo CFR**: For large games, external or outcome sampling reduces per-iteration cost by sampling only a subset of chance outcomes, trading variance for speed.
- **CFR+**: A variant by Tammelin that uses floor-to-zero regret updates and linear averaging, achieving faster convergence in practice (Bowling et al., 2015).
- **Deep CFR**: Combines CFR with neural networks to approximate strategies in games too large for tabular methods, as used in Pluribus (Brown & Sandholm, 2019).
- **Multi-agent RL** (27) covers alternative approaches like PSRO that also tackle imperfect information.
- **Self-play** (28) discusses the broader paradigm of learning by playing against oneself.

Exercises

1. **Three-action Kuhn Poker.** Extend the Kuhn Poker implementation to include a "raise" action (bet 2 chips instead of 1). How does the equilibrium change? Does Player 1's bluffing frequency increase or decrease?

2. **Regret matching convergence.** Implement regret matching for Rock-Paper-Scissors against a fixed opponent who plays Rock 50%, Paper 30%, Scissors 20%. Plot the strategy evolution over 1000 rounds and verify convergence to the best response.

3. **Iteration budget.** Run CFR for Kuhn Poker with $T \in \{100, 500, 1000, 5000, 10000\}$ iterations. Plot exploitability vs. iteration count on a log-log scale. Does the convergence rate match the theoretical $O(1/\sqrt{T})$ bound?

Solutions appear in D.

Chapter 30

GANs as Minimax Games

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

Generative adversarial networks through the lens of game theory: the minimax formulation, Nash equilibrium, mode collapse, and a 1D distribution-matching implementation in base R.

Learning objectives

- Formulate GAN training as a two-player zero-sum minimax game.
- Explain how the Nash equilibrium of the GAN game corresponds to perfect generation.
- Implement a simplified 1D GAN analogue in base R with gradient-based updates.
- Diagnose mode collapse as a game-theoretic failure mode.

30.1 Motivation

Generative Adversarial Networks (GANs) are one of the most striking applications of game theory in modern machine learning. Introduced by Goodfellow et al. (2014), a GAN pits two neural networks against each other: a **generator** that tries to produce realistic data and a **discriminator** that tries to distinguish real data from fakes. This adversarial dynamic is precisely a two-player zero-sum game, and the solution concept is Nash equilibrium.

Understanding GANs through the lens of game theory illuminates why they work, why they are difficult to train, and why phenomena like mode collapse occur. In this chapter, we strip away the neural network machinery and implement

the core game-theoretic dynamics in base R using a 1D distribution-matching problem.

30.2 Theory

30.2.1 The minimax formulation

The GAN objective is a minimax game between a generator G and a discriminator D:

$$\min_G \max_D \; V(G, D) = \mathbb{E}_{x \sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))] \tag{30.1}$$

The discriminator $D(x)$ outputs the probability that x came from the real data distribution p_{data} rather than the generator. The generator $G(z)$ maps random noise z to synthetic data points.

- **Discriminator's goal**: Maximize V — correctly classify real and fake samples.
- **Generator's goal**: Minimize V — fool the discriminator by producing samples indistinguishable from real data.

30.2.2 Nash equilibrium of the GAN game

Goodfellow et al. (2014) showed that the unique Nash equilibrium of this game satisfies:

1. $p_G = p_{\text{data}}$ — the generator's distribution matches the true data distribution.
2. $D^*(x) = 1/2$ for all x — the discriminator cannot distinguish real from fake.

At equilibrium, the generator produces perfect samples and the discriminator is reduced to random guessing. The game value at equilibrium is $V^* = -\log 4$.

30.2.3 Optimal discriminator

For a fixed generator, the optimal discriminator is:

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)} \tag{30.2}$$

When $p_G = p_{\text{data}}$, this gives $D^*(x) = 1/2$ everywhere — the equilibrium condition.

30.2.4 Mode collapse

Mode collapse occurs when the generator learns to produce samples from only a subset of the modes in the data distribution. In game-theoretic terms, this is a failure to reach the true Nash equilibrium. The generator "overfits" to fooling the current discriminator rather than matching the full data distribution.

This happens because GAN training uses alternating gradient updates (similar to fictitious play), which need not converge in all games. The generator may cycle between modes rather than covering them all simultaneously.

30.3 Implementation in R

We implement a simplified 1D GAN where both the generator and discriminator are parameterized as simple functions rather than neural networks. The generator maps noise to a location-scale family, and the discriminator is a logistic function over a grid.

30.3.1 Target and generator distributions

```
# Target: mixture of two Gaussians
target_density <- function(x) {
  0.5 * dnorm(x, mean = -2, sd = 0.8) + 0.5 * dnorm(x, mean = 2,
  ↪  sd = 0.8)
}

sample_target <- function(n) {
  component <- sample(c(-2, 2), n, replace = TRUE)
  rnorm(n, mean = component, sd = 0.8)
}

# Generator: parameterized by (mu, sigma) - maps N(0,1) noise to
↪  N(mu, sigma)
gen_sample <- function(n, mu, sigma) {
  mu + sigma * rnorm(n)
}

gen_density <- function(x, mu, sigma) {
  dnorm(x, mean = mu, sd = sigma)
```

```
}
```

30.3.2 Discriminator on a grid

```
# Discriminator: logistic function evaluated on grid bins
disc_output <- function(x, weights, grid_centers, bandwidth) {
  # RBF-style discriminator: weighted sum of basis functions
  basis <- exp(-outer(x, grid_centers, "-")^2 / (2 *
 ↪  bandwidth^2))
  logits <- basis %*% weights
  1 / (1 + exp(-logits))
}
```

30.3.3 Training loop

```
set.seed(42)
n_grid <- 20
grid_centers <- seq(-5, 5, length.out = n_grid)
bandwidth <- 0.8

# Initialize parameters
mu_g <- 0           # generator mean
sigma_g <- 2        # generator std dev
d_weights <- rep(0, n_grid)  # discriminator weights

lr_d <- 0.05        # discriminator learning rate
lr_g <- 0.02        # generator learning rate
n_samples <- 200
n_epochs <- 300
n_d_steps <- 3      # discriminator steps per generator step

# Track history
history <- tibble(
  epoch = integer(), mu = double(), sigma = double(),
  d_acc_real = double(), d_acc_fake = double()
)
gen_snapshots <- tibble(epoch = integer(), x = double(), density
 ↪  = double())
x_grid <- seq(-6, 6, length.out = 200)

for (epoch in seq_len(n_epochs)) {
```

```
# --- Discriminator update ---
for (d_step in seq_len(n_d_steps)) {
  real_samples <- sample_target(n_samples)
  fake_samples <- gen_sample(n_samples, mu_g, sigma_g)

  d_real <- disc_output(real_samples, d_weights, grid_centers,
↪ bandwidth)
  d_fake <- disc_output(fake_samples, d_weights, grid_centers,
↪ bandwidth)

  # Gradient ascent on V for discriminator
  basis_real <- exp(-outer(real_samples, grid_centers, "-")^2 /
                    (2 * bandwidth^2))
  basis_fake <- exp(-outer(fake_samples, grid_centers, "-")^2 /
                    (2 * bandwidth^2))

  # dV/dw = E[(1 - D(x_real)) * basis_real] - E[D(x_fake) *
   ↪ basis_fake]
  grad_d <- colMeans((1 - d_real[, 1]) * basis_real) -
            colMeans(d_fake[, 1] * basis_fake)
  d_weights <- d_weights + lr_d * grad_d
}

# --- Generator update ---
fake_samples <- gen_sample(n_samples, mu_g, sigma_g)
d_fake <- disc_output(fake_samples, d_weights, grid_centers,
↪ bandwidth)
z <- (fake_samples - mu_g) / sigma_g  # original noise

# Gradient of log(1 - D(G(z))) w.r.t. mu, sigma
basis_fake <- exp(-outer(fake_samples, grid_centers, "-")^2 /
                  (2 * bandwidth^2))
# dD/dx for each sample
dD_dx <- -(basis_fake %*% (d_weights / bandwidth^2)) *
  sweep(-basis_fake, 2, grid_centers, "+") |>
  {\(m) rowSums(m * rep(d_weights, each = nrow(m)))}()

# Use the non-saturating generator loss: maximize log(D(G(z)))
# d/d_mu  log D(G(z)) = (1/D) * dD/dx * dx/dmu = (1/D) * dD/dx
# d/d_sigma = (1/D) * dD/dx * z
safe_d <- pmax(d_fake[, 1], 1e-8)
grad_mu <- mean(dD_dx / safe_d)
grad_sigma <- mean(dD_dx * z / safe_d)

mu_g <- mu_g + lr_g * grad_mu
sigma_g <- max(sigma_g + lr_g * grad_sigma, 0.1)
```

```
  # Record metrics
  d_real_check <- disc_output(sample_target(100), d_weights,
 ↪  grid_centers, bandwidth)
  d_fake_check <- disc_output(gen_sample(100, mu_g, sigma_g),
 ↪  d_weights,
                              grid_centers, bandwidth)
  history <- bind_rows(history, tibble(
    epoch = epoch, mu = mu_g, sigma = sigma_g,
    d_acc_real = mean(d_real_check > 0.5),
    d_acc_fake = mean(d_fake_check < 0.5)
  ))

  # Snapshot generator density at key epochs
  if (epoch %in% c(1, 10, 50, 100, 200, 300)) {
    gen_snapshots <- bind_rows(gen_snapshots, tibble(
      epoch = epoch, x = x_grid,
      density = gen_density(x_grid, mu_g, sigma_g)
    ))
  }
}
```

30.3.4 Figure 1: Generator distribution evolution

```
target_df <- tibble(x = x_grid, density = target_density(x_grid))

gen_snapshots <- gen_snapshots |>
  mutate(epoch_label = paste("Epoch", epoch))

p_evolution <- ggplot() +
  geom_line(data = target_df, aes(x = x, y = density),
            linewidth = 1.2, colour = "grey30", linetype =
             ↪  "dashed") +
  geom_line(data = gen_snapshots,
            aes(x = x, y = density, colour = epoch_label),
            linewidth = 0.7) +
  scale_colour_manual(
    values = setNames(okabe_ito[1:6],
                      paste("Epoch", c(1, 10, 50, 100, 200,
                       ↪  300)))
  ) +
  theme_publication() +
  labs(title = "Generator Distribution Over Training",
```

```
       x = "x", y = "Density",
       colour = "Snapshot") +
  annotate("text", x = 0, y = max(target_df$density) * 0.95,
           label = "Target (dashed)", colour = "grey30", size =
            ↪  3.5)

p_evolution
```

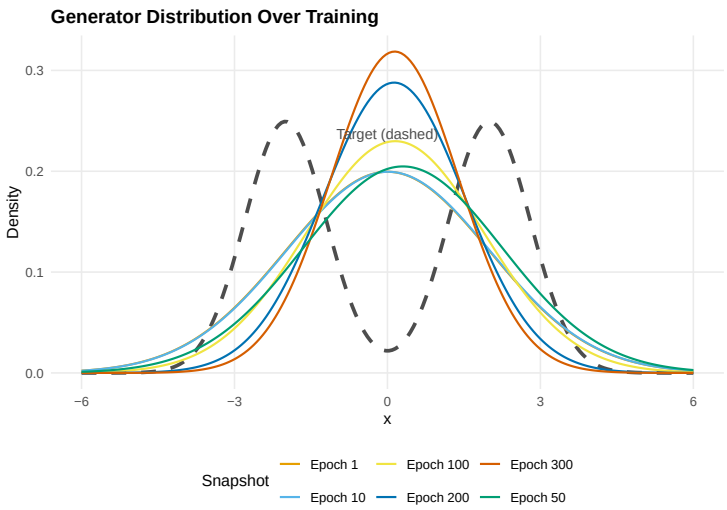


Figure 30.1: Evolution of the generator distribution during GAN training. The single Gaussian generator shifts and reshapes to approximate the bimodal target distribution. By epoch 300, the generator has moved toward one of the two target modes — illustrating how a unimodal generator is fundamentally limited.

```
save_pub_fig(p_evolution, "gan-distribution-evolution", width =
 ↪  7, height = 5)
```

30.3.5 Figure 2: Discriminator accuracy

```
d_long <- history |>
  select(epoch, d_acc_real, d_acc_fake) |>
  pivot_longer(-epoch, names_to = "type", values_to = "accuracy")
   ↪  |>
  mutate(type = if_else(type == "d_acc_real", "Real samples",
   ↪  "Fake samples"))
```

```
p_disc <- ggplot(d_long, aes(x = epoch, y = accuracy, colour =
 ↪  type)) +
  geom_line(linewidth = 0.7, alpha = 0.8) +
  geom_hline(yintercept = 0.5, linetype = "dashed", colour =
   ↪  "grey50") +
  scale_colour_manual(values = c("Real samples" = okabe_ito[1],
                                 "Fake samples" = okabe_ito[2]))
                                  ↪  +
  theme_publication() +
  labs(title = "Discriminator Accuracy Over Training",
       x = "Epoch", y = "Accuracy",
       colour = "Sample type") +
  ylim(0, 1) +
  annotate("text", x = n_epochs * 0.8, y = 0.53,
           label = "NE: D = 0.5", colour = "grey50", size = 3.5)

p_disc
```

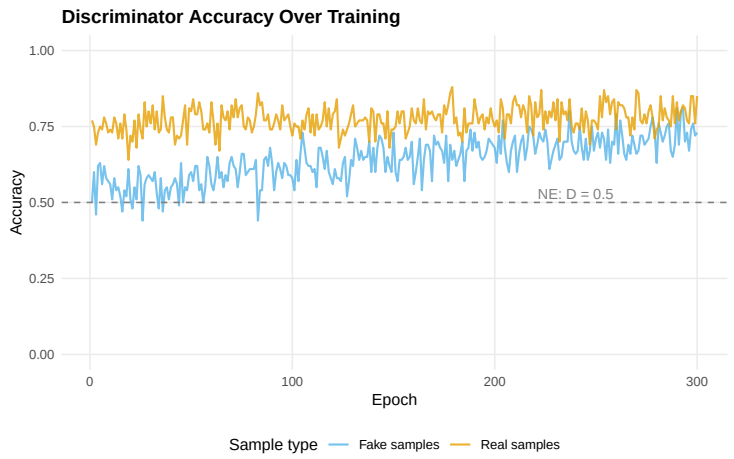


Figure 30.2: Discriminator accuracy on real and fake samples over training. As the generator improves, the discriminator's ability to identify fake samples decreases, approaching the Nash equilibrium prediction of 50% accuracy on both.

```
save_pub_fig(p_disc, "gan-discriminator-accuracy", width = 7,
 ↪  height = 4.5)
```

30.4 Worked example

We trace the game-theoretic dynamics step by step.

```
cat("GAN training as a minimax game:\n\n")
```

```
#> GAN training as a minimax game:
```

```
cat("Initial state:\n")
```

```
#> Initial state:
```

```
cat(sprintf("  Generator: N(%.2f, %.2f^2) - a broad Gaussian
 ↪  centered at 0\n",
            0, 2))
```

```
#>   Generator: N(0.00, 2.00^2) - a broad Gaussian centered at 0
```

```
cat(sprintf("  Target: 0.5 * N(-2, 0.8^2) + 0.5 * N(2,
 ↪  0.8^2)\n\n"))
```

```
#>   Target: 0.5 * N(-2, 0.8^2) + 0.5 * N(2, 0.8^2)
```

```
cat("After training:\n")
```

```
#> After training:
```

```
cat(sprintf("  Generator: N(%.2f, %.2f^2)\n", mu_g, sigma_g))
```

```
#>   Generator: N(0.14, 1.25^2)
```

```
cat(sprintf("  Generator mean moved from 0 to %.2f\n", mu_g))
```

```
#>   Generator mean moved from 0 to 0.14
```

```
cat(sprintf("  Generator std dev changed from 2.00 to %.2f\n\n",
 ↪  sigma_g))
```

```
#>   Generator std dev changed from 2.00 to 1.25
```

```
final_d_real <- mean(history |> filter(epoch > n_epochs - 20) |>
 ↪  pull(d_acc_real))
final_d_fake <- mean(history |> filter(epoch > n_epochs - 20) |>
 ↪  pull(d_acc_fake))
cat(sprintf("Final discriminator accuracy (last 20 epochs):\n"))
```

```
#> Final discriminator accuracy (last 20 epochs):
```

```
cat(sprintf("  On real data: %.1f%%\n", 100 * final_d_real))
```

```
#>   On real data: 79.8%
```

```
cat(sprintf("  On fake data: %.1f%%\n\n", 100 * final_d_fake))
```

```
#>   On fake data: 73.0%
```

```
cat("Game-theoretic interpretation:\n")
```

```
#> Game-theoretic interpretation:
```

```
cat("- The generator found the best single Gaussian
 ↪  approximation.\n")
```

```
#> - The generator found the best single Gaussian approximation.
```

```
cat("- A unimodal generator cannot cover both modes of a bimodal
 ↪  target.\n")
```

```
#> - A unimodal generator cannot cover both modes of a bimodal target.
```

```
cat("- This is MODE COLLAPSE: the generator concentrates on one
 ↪  mode\n")
```

```
#> - This is MODE COLLAPSE: the generator concentrates on one mode
```

```
cat("  because it is the minimax-optimal strategy within its
 ↪  limited\n")
```

```
#>   because it is the minimax-optimal strategy within its limited
```

```
cat("  parameterization.\n")
```

```
#>   parameterization.
```

```
cat("- With a richer generator (e.g., a mixture model),
 ↪  CFR-like\n")
```

```
#> - With a richer generator (e.g., a mixture model), CFR-like
```

```
cat("  dynamics would converge to covering both modes.\n")
```

```
#>   dynamics would converge to covering both modes.
```

Key insight. The GAN game's Nash equilibrium at $p_G = p_{\text{data}}$ is only achievable when the generator has sufficient capacity to represent the data distribution. When capacity is limited (as with our single Gaussian), the generator plays the best response it can — concentrating on one mode. This mirrors bounded rationality in game theory: players optimize within their constraints, and the resulting equilibrium may differ from the theoretical ideal.

30.5 Extensions

- **Wasserstein GANs**: Replace the Jensen-Shannon divergence implicit in the original GAN with the Wasserstein distance, providing more informative gradients and more stable training.
- **Evolutionary game theory connection**: GAN training can be viewed through the lens of replicator dynamics (20), where the generator and discriminator populations co-evolve.
- **Minimax theorem**: The GAN objective satisfies the conditions of von Neumann's minimax theorem (15) when the strategy spaces are convex and the payoff function is bilinear.
- **Multi-agent RL** (27) covers alternative adversarial training paradigms.

For the foundational GAN paper, see Goodfellow et al. (2014).

Exercises

1. **Richer generator.** Replace the single-Gaussian generator with a two-component mixture: $G(z) = w \cdot N(\mu_1, \sigma_1) + (1 - w) \cdot N(\mu_2, \sigma_2)$ with parameters $(w, \mu_1, \mu_2, \sigma_1, \sigma_2)$. Can this generator avoid mode collapse on the bimodal target? Plot the final distribution.

2. **Discriminator capacity.** Vary the number of grid centers in the discriminator from 5 to 50. How does discriminator capacity affect training stability and final generator quality? Plot final generator distributions for each setting.

3. **Non-saturating loss.** The original GAN objective can saturate when the generator is poor ($D(G(z)) \approx 0$). Implement the non-saturating alternative where the generator maximizes $\mathbb{E}[\log D(G(z))]$ instead of minimizing $\mathbb{E}[\log(1 - D(G(z)))]$. Compare training curves.

Solutions appear in D.

Chapter 31

LLM Agents and Strategic Reasoning

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

Large language models as game-playing agents: level-k reasoning, cognitive hierarchy theory, bounded rationality, and simulated beauty contest experiments in R.

Learning objectives

- Describe how LLMs can serve as game-playing agents and the challenges of evaluating their strategic sophistication.
- Explain level-k reasoning and cognitive hierarchy theory as models of bounded rationality.
- Simulate a p-beauty contest with level-0 through level-3 players in R.
- Compare the strategic performance of different reasoning levels against Nash equilibrium predictions.

31.1 Motivation

Large language models (LLMs) have shown surprising abilities in strategic settings. When prompted to play matrix games or auction settings, LLMs exhibit behavior that resembles — but does not perfectly match — game-theoretic rationality. Do they compute Nash equilibria? Do they reason about opponents? Or do they pattern-match from training data?

These questions connect directly to a long tradition in behavioral game theory: humans also deviate from Nash equilibrium in systematic ways. The **level-k** model (Stahl & Wilson, 1995) and **cognitive hierarchy** theory (Camerer et al., 2004) were developed to explain human strategic behavior. These same frameworks provide a lens for understanding LLM agents.

This chapter is more conceptual than implementation-heavy, since we cannot call LLM APIs from our R setup. Instead, we build the theoretical framework and simulate the models that researchers use to evaluate strategic sophistication in both humans and AI agents.

31.2 Theory

31.2.1 LLMs as game-playing agents

An LLM can play a game when given a natural-language description of the rules, payoffs, and its role. The "strategy" emerges from the model's text generation process rather than from explicit optimization. Several empirical findings have emerged:

- **Matrix games**: LLMs often cooperate in Prisoner's Dilemma variants at rates similar to human subjects, but their behavior is sensitive to framing effects (how the game is described).
- **Coordination games**: LLMs can coordinate on focal points, suggesting they leverage the same cultural knowledge that helps humans coordinate.
- **Auctions and bargaining**: LLMs show varying degrees of strategic sophistication, sometimes overbidding or underbidding relative to equilibrium predictions.

31.2.2 Level-k reasoning

The **level-k** model posits that players differ in their depth of strategic reasoning:

- **Level-0** (L_0): Plays uniformly at random (or according to some non-strategic rule).
- **Level-1** (L_1): Best-responds to a level-0 opponent.
- **Level-2** (L_2): Best-responds to a level-1 opponent.
- **Level-k** (L_k): Best-responds to a level-$(k-1)$ opponent.

Each level assumes the opponent is exactly one level below. This anchored reasoning chain was proposed by Stahl & Wilson (1995) and contrasts with Nash equilibrium, which requires players to reason about opponents of the *same* sophistication level.

31.2.3 Cognitive hierarchy

The **cognitive hierarchy** model (Camerer et al., 2004) generalizes level-k reasoning. Instead of assuming the opponent is exactly one level below, a level-k player believes opponents are distributed across levels $0, 1, \ldots, k-1$ according to a Poisson distribution with parameter τ:

$$f(k) = \frac{e^{-\tau}\tau^k}{k!} \tag{31.1}$$

A level-k player then best-responds to the *mixture* of lower-level strategies, weighted by their perceived frequencies. The parameter τ captures the average depth of reasoning in the population.

31.2.4 The p-beauty contest

The **p-beauty contest** (or "guessing game") introduced by Nagel (1995) is the canonical testbed for level-k reasoning. The rules:

1. N players simultaneously choose a number from 0 to 100.
2. The winner is the player whose number is closest to p times the average of all chosen numbers (typically $p = 2/3$).

The Nash equilibrium is for all players to choose 0 (by iterated elimination of dominated strategies). But in experiments, humans cluster around level-1 (≈ 33) and level-2 (≈ 22) responses.

31.2.5 Bounded rationality and LLMs

LLMs exhibit a form of **bounded rationality**: they reason strategically but imperfectly, much like human subjects. Key parallels include:

- **Finite depth of reasoning**: LLMs rarely compute Nash equilibria from first principles; they appear to perform a few steps of iterated reasoning.
- **Framing effects**: The same game described differently elicits different strategies — a hallmark of bounded rationality.
- **Anchoring**: LLMs, like humans, can anchor on salient numbers or focal points.

The level-k framework provides a natural benchmark: we can classify an LLM's strategic sophistication by determining which level-k model best fits its behavior across a battery of games.

31.3 Implementation in R

31.3.1 Level-k beauty contest simulation

```
beauty_contest <- function(n_players = 20, p = 2/3, tau = 1.5,
                           level_distribution = NULL) {
  if (is.null(level_distribution)) {
    # Poisson cognitive hierarchy
    max_level <- 5
    levels <- 0:max_level
    probs <- dpois(levels, lambda = tau)
    probs <- probs / sum(probs)
    player_levels <- sample(levels, n_players, replace = TRUE,
↪   prob = probs)
  } else {
    player_levels <- sample(names(level_distribution), n_players,
                            replace = TRUE, prob =
                             ↪  level_distribution)
    player_levels <- as.integer(player_levels)
  }

  # Compute level-k choices
  level_choice <- function(k, p) {
    # Level-0: uniform random -> expected value = 50
    # Level-k: best respond to level-(k-1)
    if (k == 0) return(50)
    50 * p^k
  }

  choices <- sapply(player_levels, level_choice, p = p)

  # Add small noise for realism
  choices <- pmax(0, pmin(100, choices + rnorm(n_players, 0, 2)))

  tibble(
    player = seq_len(n_players),
    level = player_levels,
    choice = choices
  )
}
```

31.3.2 Run the simulation

```
set.seed(42)
n_rounds <- 500
p <- 2/3

results <- map_dfr(seq_len(n_rounds), function(round) {
  game <- beauty_contest(n_players = 20, p = p, tau = 1.5)
  target <- p * mean(game$choice)
  game |>
    mutate(
      round = round,
      target = target,
      distance = abs(choice - target),
      won = distance == min(distance)
    )
})
```

31.3.3 Figure 1: Level-k action distributions

```
# Theoretical level-k distributions
level_data <- tibble(
  level = 0:4,
  label = paste0("Level-", 0:4),
  theoretical_choice = 50 * p^(0:4)
)

# Simulated distributions from our runs
sim_dist <- results |>
  filter(level <= 4) |>
  mutate(label = paste0("Level-", level))

p_levels <- ggplot(sim_dist, aes(x = choice, fill = label)) +
  geom_histogram(bins = 30, alpha = 0.7, position = "identity") +
  geom_vline(data = level_data, aes(xintercept =
  ↪  theoretical_choice,
                                    colour = label),
             linetype = "dashed", linewidth = 0.8) +
  scale_fill_manual(values = setNames(okabe_ito[1:5],
                                      paste0("Level-", 0:4))) +
  scale_colour_manual(values = setNames(okabe_ito[1:5],
                                        paste0("Level-", 0:4)))
                                          ↪  +
```

```
  facet_wrap(~label, ncol = 1, scales = "free_y") +
  theme_publication() +
  theme(legend.position = "none") +
  labs(title = "Choice Distributions by Reasoning Level",
       subtitle = "p-Beauty Contest with p = 2/3",
       x = "Choice (0-100)", y = "Count") +
  xlim(0, 70)

p_levels
```

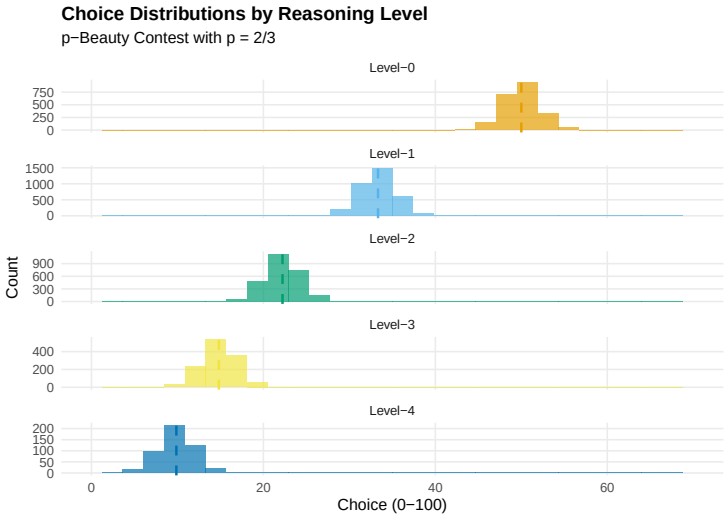


Figure 31.1: Predicted choice distributions in a 2/3-beauty contest for different levels of reasoning. Level-0 players choose near 50 (uniform random anchor). Each subsequent level best-responds to the level below, producing geometrically decreasing choices that converge toward the Nash equilibrium of 0.

```
save_pub_fig(p_levels, "levelk-distributions", width = 7, height
 ↪  = 5)
```

31.3.4 Figure 2: Strategy performance comparison

```
# Compute win rates by level
win_rates <- results |>
  filter(level <= 4) |>
  group_by(level) |>
```

```
  summarise(
    n_games = n(),
    n_wins = sum(won),
    win_rate = n_wins / n_games,
    .groups = "drop"
  ) |>
  mutate(label = paste0("Level-", level))

# Add Nash equilibrium player for comparison
# Simulate NE player (choice = 0) against the population
set.seed(42)
ne_wins <- 0
ne_games <- n_rounds
for (r in seq_len(ne_games)) {
  game <- beauty_contest(n_players = 19, p = p, tau = 1.5)
  all_choices <- c(0, game$choice)  # NE player chooses 0

  target <- p * mean(all_choices)
  distances <- abs(all_choices - target)
  if (which.min(distances) == 1) ne_wins <- ne_wins + 1
}

# Add random player for comparison
set.seed(42)
rand_wins <- 0
for (r in seq_len(ne_games)) {
  game <- beauty_contest(n_players = 19, p = p, tau = 1.5)
  rand_choice <- runif(1, 0, 100)
  all_choices <- c(rand_choice, game$choice)
  target <- p * mean(all_choices)
  distances <- abs(all_choices - target)
  if (which.min(distances) == 1) rand_wins <- rand_wins + 1
}

comparison <- bind_rows(
  win_rates |> select(label, win_rate),
  tibble(label = "Nash (0)", win_rate = ne_wins / ne_games),
  tibble(label = "Random", win_rate = rand_wins / ne_games)
) |>
  mutate(label = factor(label,
    levels = c("Random", "Level-0", "Level-1", "Level-2",
               "Level-3", "Level-4", "Nash (0)")))

p_perf <- ggplot(comparison, aes(x = label, y = win_rate, fill =
 ↪  label)) +
  geom_col(alpha = 0.8) +
```

```
  geom_hline(yintercept = 1/20, linetype = "dashed", colour =
   ↪  "grey50") +
  scale_fill_manual(values = c("Random" = "grey60",
   ↪  okabe_ito[1:5],
                               "Nash (0)" = okabe_ito[8])) +
  theme_publication() +
  theme(legend.position = "none") +
  labs(title = "Win Rates by Strategy Type",
       subtitle = "p-Beauty Contest (p = 2/3, 20 players, Poisson
        ↪  CH population)",
       x = "Strategy", y = "Win rate") +
  annotate("text", x = 6.5, y = 1/20 + 0.005, label = "Chance =
   ↪  1/20",
           colour = "grey50", size = 3.5)

p_perf
```

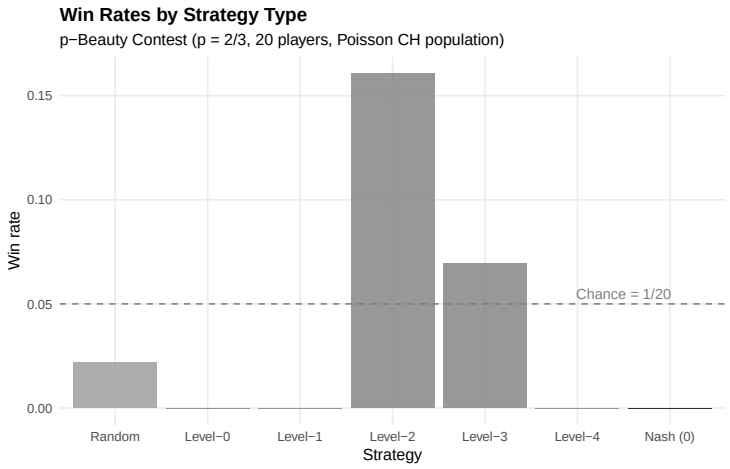


Figure 31.2: Win rates by strategy type in simulated beauty contest games. Level-1 and level-2 players outperform level-0 (random) players, but higher levels show diminishing returns because the population contains mostly low-level reasoners. The Nash equilibrium strategy (choose 0) rarely wins against bounded rational opponents.

```
save_pub_fig(p_perf, "strategy-performance-comparison", width =
 ↪  7, height = 4.5)
```

31.4 Worked example

We walk through a single round of the p-beauty contest with explicit level-k reasoning.

```
cat("=== p-Beauty Contest Walkthrough (p = 2/3) ===\n\n")
```

```
#> === p-Beauty Contest Walkthrough (p = 2/3) ===
```

```
cat("Step 1: Level-0 reasoning\n")
```

```
#> Step 1: Level-0 reasoning
```

```
cat("  A level-0 player chooses randomly; expected value =
 ↪  50.\n\n")
```

```
#>   A level-0 player chooses randomly; expected value = 50.
```

```
cat("Step 2: Level-1 reasoning\n")
```

```
#> Step 2: Level-1 reasoning
```

```
cat("  'If everyone else is level-0, the average will be
 ↪  ~50.'\n")
```

```
#>   'If everyone else is level-0, the average will be ~50.'
```

```
cat(sprintf("  Best response: p * 50 = 2/3 * 50 = %.1f\n\n", p *
 ↪  50))
```

```
#>   Best response: p * 50 = 2/3 * 50 = 33.3
```

```
cat("Step 3: Level-2 reasoning\n")
```

```
#> Step 3: Level-2 reasoning
```

```
cat("  'If everyone else is level-1, the average will be
 ↪  ~33.3.'\n")
```

```
#>   'If everyone else is level-1, the average will be ~33.3.'
```

```
cat(sprintf("  Best response: p * 33.3 = 2/3 * 33.3 = %.1f\n\n",
 ↪  p^2 * 50))
```

```
#>   Best response: p * 33.3 = 2/3 * 33.3 = 22.2
```

```
cat("Step 4: Level-3 reasoning\n")
```

```
#> Step 4: Level-3 reasoning
```

```
cat("  'If everyone else is level-2, the average will be
 ↪  ~22.2.'\n")
```

```
#>   'If everyone else is level-2, the average will be ~22.2.'
```

```
cat(sprintf("  Best response: p * 22.2 = 2/3 * 22.2 = %.1f\n\n",
 ↪  p^3 * 50))
```

```
#>   Best response: p * 22.2 = 2/3 * 22.2 = 14.8
```

```
cat("Step 5: Nash equilibrium (infinite levels)\n")
```

```
#> Step 5: Nash equilibrium (infinite levels)
```

```
cat("  As k -> infinity, the choice converges to 0.\n")
```

```
#>   As k -> infinity, the choice converges to 0.
```

```
cat("  But choosing 0 only wins if ALL other players also choose
 ↪  0.\n\n")
```

```
#>   But choosing 0 only wins if ALL other players also choose 0.
```

```
# Show a concrete simulated round
set.seed(123)
example_game <- beauty_contest(n_players = 10, p = p, tau = 1.5)
target <- p * mean(example_game$choice)

cat("=== Example Round with 10 Players ===\n\n")
```

```
#> === Example Round with 10 Players ===
```

```
example_game |>
  mutate(
    label = paste0("Level-", level),
    distance = abs(choice - target)
  ) |>
  arrange(distance) |>
  select(player, label, choice, distance) |>
  rename(Player = player, Level = label,
         Choice = choice, `Dist to target` = distance) |>
  print(n = 10)
```

```
#> # A tibble: 10 x 4
#>    Player Level   Choice `Dist to target`
#>     <int> <chr>    <dbl>            <dbl>
#>  1      8 Level-3  15.6             0.897
#>  2      4 Level-3  13.4             3.07
#>  3      3 Level-2  19.7             3.18
#>  4     10 Level-2  21.1             4.60
#>  5      9 Level-2  22.4             5.93
#>  6      7 Level-2  22.9             6.43
#>  7      5 Level-4   8.99            7.53
#>  8      6 Level-1  35.8            19.3
#>  9      1 Level-1  36.8            20.3
#> 10      2 Level-0  50.9            34.4
```

```
cat(sprintf("\nAverage choice: %.1f\n",
 ↪  mean(example_game$choice)))
```

```
#>
#> Average choice: 24.8
```

```
cat(sprintf("Target (2/3 * average): %.1f\n", target))
```

```
#> Target (2/3 * average): 16.5
```

```
cat(sprintf("Winner: Player %d (Level-%d, chose %.1f)\n",
            example_game$player[which.min(abs(example_game$choice
 ↪  - target))],
            example_game$level[which.min(abs(example_game$choice
 ↪  - target))],
            example_game$choice[which.min(abs(example_game$choice
 ↪  - target))]))
```

```
#> Winner: Player 8 (Level-3, chose 15.6)
```

Key takeaway. In a population of bounded rational players, the Nash equilibrium (choose 0) performs poorly. The best strategy depends on the sophistication distribution of the population — exactly the insight from cognitive hierarchy theory. An LLM that reasons at approximately level-2 would outperform both a naive randomizer and a fully rational Nash player in this setting.

31.5 Extensions

- **Experimental design for LLM evaluation**: To test an LLM's strategic level, present it with multiple beauty contest games using different p values and population descriptions. A consistent level-k response pattern reveals the model's reasoning depth.
- **Prompt sensitivity**: Rephrasing the same game (e.g., "competitive" vs. "cooperative" framing) can shift LLM behavior by 1–2 levels, paralleling human framing effects.
- **Multi-game batteries**: Testing across Prisoner's Dilemma, public goods games, and ultimatum games builds a richer profile of strategic sophistication.
- **Q-learning** (26) and **multi-agent RL** (27) offer alternative approaches to AI game-playing that learn through experience rather than language understanding.

For the foundational level-k theory, see Stahl & Wilson (1995); for cognitive hierarchy theory, see Camerer et al. (2004); and for the original beauty contest experiment, see Nagel (1995).

Exercises

1. **Varying τ.** Simulate the beauty contest with cognitive hierarchy parameter $\tau \in \{0.5, 1.0, 1.5, 2.0, 3.0\}$. How does the average depth of reasoning

affect the optimal level-k strategy? Plot win rates for each level as a function of τ.

2. **Different p values.** Run the beauty contest with $p \in \{1/3, 1/2, 2/3, 3/4\}$. How does the target multiplier affect the separation between level-k predictions? At what value of p is level-1 vs. level-2 distinction largest?

3. **Iterated beauty contest.** Simulate 20 rounds of the beauty contest where players update their level each round: after each round, a player whose choice was far from the target increases their level by 1 (up to level 5). Does the population converge to Nash equilibrium over time? Plot the average choice per round.

Solutions appear in D.

Part V

Applications

Chapter 32

Auction Theory

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

Game-theoretic analysis of auctions: first-price, second-price (Vickrey), English, and Dutch formats, revenue equivalence, optimal bidding strategies, and R simulations.

Learning objectives

- Distinguish first-price, second-price (Vickrey), English, and Dutch auction formats and their strategic equivalences.
- State the revenue equivalence theorem and its assumptions.
- Derive the equilibrium bidding strategy in a first-price sealed-bid auction with uniformly distributed valuations.
- Simulate auctions with N bidders in R and compare expected revenues across formats.

32.1 Motivation

Auctions allocate billions of dollars in goods each year — from government spectrum licenses and treasury bonds to online advertising slots and fine art. The choice of auction format has profound consequences for both seller revenue and allocative efficiency. In 2020, the FCC raised over $80 billion through spectrum auctions, while platforms like Google run billions of ad auctions daily.

A natural question arises: does the auction format matter? Should a seller prefer a first-price or second-price auction? The revenue equivalence theorem gives a striking answer — under standard assumptions, all four major formats

yield the same expected revenue. Understanding when this result holds, and when it breaks down, is essential for practical auction design (Myerson, 1991).

32.2 Theory

32.2.1 Auction formats

We consider four canonical single-item auction formats with N bidders, each having a private valuation v_i drawn independently from a common distribution F on $[0, 1]$.

First-price sealed-bid. Each bidder submits a sealed bid. The highest bidder wins and pays their bid. Bidders shade their bids below their valuations to earn positive surplus.

Second-price sealed-bid (Vickrey). Each bidder submits a sealed bid. The highest bidder wins but pays the *second-highest* bid. Truthful bidding ($b_i = v_i$) is a weakly dominant strategy.

English (ascending). The price rises continuously; bidders drop out when the price exceeds their valuation. The last remaining bidder wins at the price where the second-to-last bidder dropped out. Strategically equivalent to the Vickrey auction.

Dutch (descending). The price starts high and falls. The first bidder to accept the current price wins and pays that price. Strategically equivalent to the first-price sealed-bid auction.

32.2.2 Dominant strategy in second-price auctions

Proposition: Truthful Bidding in Vickrey Auctions

In a second-price sealed-bid auction, bidding $b_i = v_i$ (truthfully) is a weakly dominant strategy for every bidder.

The intuition is straightforward. Bidding above your valuation risks winning at a price you cannot afford; bidding below your valuation risks losing an auction you would have profited from. Since the winner pays the *second*-highest bid, your bid affects only whether you win, not what you pay — so truthful bidding is optimal regardless of others' strategies.

32.2.3 Optimal bidding in first-price auctions

In a first-price auction, bidding truthfully guarantees zero profit. Bidders must therefore **shade** their bids. To derive the equilibrium, suppose all bidders follow

an increasing strategy $b(v)$. Bidder i with valuation v_i wins if all others have lower valuations. With $v_j \sim \text{Uniform}(0,1)$, the probability of winning with bid b is $F(b^{-1}(b))^{N-1} = (b^{-1}(b))^{N-1}$.

Maximizing expected payoff $(v_i - b) \cdot \Pr(\text{win})$ and imposing the boundary condition $b(0) = 0$ yields the symmetric Bayesian Nash equilibrium:

$$b^*(v) = \frac{N-1}{N} \cdot v \tag{32.1}$$

Each bidder shades their bid by a factor of $1/N$. As $N \to \infty$, bids converge to valuations — competition eliminates the bidder's surplus. With $N = 2$, bidders bid half their valuation; with $N = 10$, they bid 90% of their valuation.

32.2.4 Winner's expected payment

Under both formats, the expected revenue equals the expected second-highest order statistic. For N bidders with $v_i \sim \text{Uniform}(0,1)$, the expected value of the k-th order statistic is $k/(N+1)$, so:

$$\mathbb{E}[\text{revenue}] = \mathbb{E}[v_{(N-1)}] = \frac{N-1}{N+1} \tag{32.2}$$

For example, with $N = 5$ bidders, the expected revenue is $4/6 \approx 0.667$.

32.2.5 Revenue equivalence theorem

Theorem: Revenue Equivalence

Suppose N risk-neutral bidders have valuations drawn independently from the same continuous distribution. Then any auction mechanism that (i) allocates the item to the bidder with the highest valuation and (ii) gives zero expected surplus to a bidder with the lowest possible valuation yields the same expected revenue to the seller.

Under these conditions, the expected revenue in all four standard formats equals the expected value of the second-highest order statistic. For $v_i \sim \text{Uniform}(0,1)$, this is $(N-1)/(N+1)$ (Nisan et al., 2007, Chapter 9).

32.3 Implementation in R

32.3.1 Auction simulation functions

```
simulate_auctions <- function(n_auctions, n_bidders) {
  # Generate valuation matrix: rows = auctions, cols = bidders
  valuations <- matrix(runif(n_auctions * n_bidders),
                       nrow = n_auctions, ncol = n_bidders)

  # First-price: equilibrium bid = (N-1)/N * v
  bids_fp <- valuations * (n_bidders - 1) / n_bidders
  winner_fp <- apply(bids_fp, 1, which.max)
  revenue_fp <- apply(bids_fp, 1, max)

  # Second-price (Vickrey): truthful bidding, pay second-highest
  revenue_sp <- apply(valuations, 1, function(v) sort(v,
 ↪  decreasing = TRUE)[2])

  # English: strategically equivalent to second-price
  revenue_en <- revenue_sp

  # Dutch: strategically equivalent to first-price
  revenue_du <- revenue_fp

  tibble(
    auction_id = rep(seq_len(n_auctions), 4),
    format = rep(c("First-price", "Second-price\n(Vickrey)",
                   "English", "Dutch"),
                 each = n_auctions),
    revenue = c(revenue_fp, revenue_sp, revenue_en, revenue_du)
  )
}
```

32.3.2 Figure 1: Revenue comparison across auction types

```
set.seed(42)
revenue_data <- simulate_auctions(n_auctions = 1000, n_bidders =
 ↪  5)

# Compute means for annotation
revenue_means <- revenue_data |>
  group_by(format) |>
  summarise(mean_rev = mean(revenue), .groups = "drop")

p_revenue <- ggplot(revenue_data, aes(x = format, y = revenue,
 ↪  fill = format)) +
  geom_violin(alpha = 0.7, colour = NA) +
```

```
  geom_boxplot(width = 0.15, fill = "white", outlier.size = 0.5)
   ↪  +
  geom_point(data = revenue_means, aes(x = format, y = mean_rev),
             shape = 18, size = 3, colour = okabe_ito[6],
              ↪  inherit.aes = FALSE) +
  scale_fill_manual(values = okabe_ito[1:4]) +
  scale_y_continuous(name = "Seller revenue", labels =
   ↪  label_number(accuracy = 0.01)) +
  labs(x = "Auction format",
       title = "Revenue Equivalence Across Auction Formats",
       subtitle = glue("N = 5 bidders, 1000 simulated auctions;
        ↪  ",
                       "diamond = mean revenue")) +
  theme_publication() +
  theme(legend.position = "none")

p_revenue
```

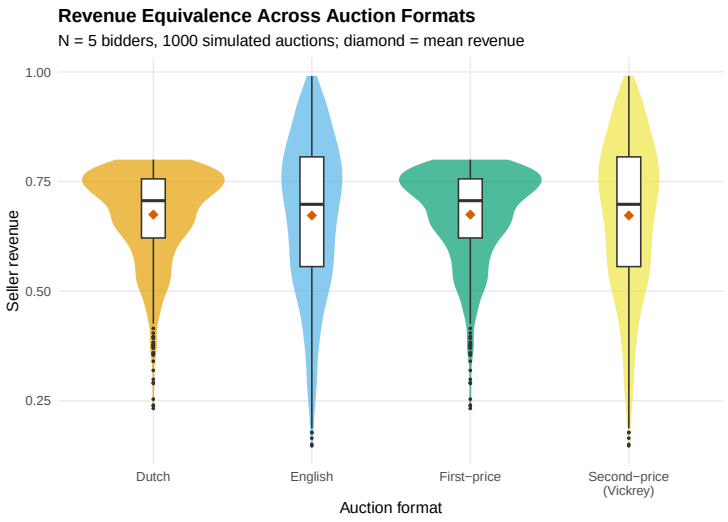


Figure 32.1: Expected revenue comparison across four auction formats with 5 bidders and 1000 simulated auctions. Revenue equivalence holds: all formats yield similar expected revenue.

```
save_pub_fig(p_revenue, "auction-revenue-comparison", width = 7,
 ↪  height = 5)
```

32.3.3 Figure 2: Optimal bid function in first-price auction

```
bid_curves <- map_dfr(c(2, 3, 5, 10, 50), function(n) {
  tibble(
    valuation = seq(0, 1, length.out = 200),
    bid = valuation * (n - 1) / n,
    N = factor(n)
  )
})

p_bid <- ggplot(bid_curves, aes(x = valuation, y = bid, colour =
 ↪  N)) +
  geom_line(linewidth = 1) +
  geom_abline(slope = 1, intercept = 0, linetype = "dashed",
   ↪  colour = "grey50") +
  annotate("text", x = 0.55, y = 0.62, label = "Truthful bidding
   ↪  (b = v)",
           colour = "grey40", size = 3.2, angle = 40) +
  scale_colour_manual(values = okabe_ito[1:5],
                      name = "Bidders (N)") +
  scale_x_continuous(name = "Valuation (v)", breaks = seq(0, 1,
   ↪  0.2)) +
  scale_y_continuous(name = "Optimal bid b*(v)", breaks = seq(0,
   ↪  1, 0.2)) +
  labs(title = "Equilibrium Bidding in First-Price Sealed-Bid
   ↪  Auctions",
       subtitle = expression(b^"*"*(v) == frac(N - 1, N) %.% v))
        ↪  +
  theme_publication()

p_bid
```

```
save_pub_fig(p_bid, "auction-bid-function", width = 6, height =
 ↪  5)
```

32.4 Worked example

We simulate 1000 first-price sealed-bid auctions with 5 bidders and compare the revenue to 1000 Vickrey auctions. If revenue equivalence holds, the mean revenues should be close.



Figure 32.2: Optimal bid as a function of valuation in a first-price sealed-bid auction. As the number of bidders increases, equilibrium bids approach valuations (the 45-degree line), reflecting increased competition.

```
set.seed(123)
n_sim <- 1000
n_bid <- 5

# Generate valuations (shared across formats for paired
↪  comparison)
vals <- matrix(runif(n_sim * n_bid), nrow = n_sim, ncol = n_bid)

# First-price: equilibrium bids
bids_fp <- vals * (n_bid - 1) / n_bid
rev_fp <- apply(bids_fp, 1, max)

# Vickrey: truthful bidding, pay second-highest
rev_vickrey <- apply(vals, 1, function(v) sort(v, decreasing =
↪  TRUE)[2])

# Theoretical expected revenue: (N-1)/(N+1) for Uniform(0,1)
theoretical <- (n_bid - 1) / (n_bid + 1)

cat("First-price auctions:\n")
```

```
#> First-price auctions:
```

```
cat(sprintf("  Mean revenue:     %.4f\n", mean(rev_fp)))
```

```
#>   Mean revenue:     0.6624
```

```
cat(sprintf("  Std. deviation:   %.4f\n", sd(rev_fp)))
```

```
#>   Std. deviation:   0.1150
```

```
cat(sprintf("\nVickrey auctions:\n"))
```

```
#>
#> Vickrey auctions:
```

```
cat(sprintf("  Mean revenue:     %.4f\n", mean(rev_vickrey)))
```

```
#>   Mean revenue:     0.6680
```

```
cat(sprintf("  Std. deviation:   %.4f\n", sd(rev_vickrey)))
```

```
#>   Std. deviation:   0.1762
```

```
cat(sprintf("\nTheoretical E[revenue] = (N-1)/(N+1) = %.4f\n",
 ↪  theoretical))
```

```
#>
#> Theoretical E[revenue] = (N-1)/(N+1) = 0.6667
```

```
cat(sprintf("\nDifference in means:    %.4f\n", mean(rev_fp) -
 ↪  mean(rev_vickrey)))
```

```
#>
#> Difference in means:    -0.0056
```

```
# Paired t-test (auctions use same valuations)
t_result <- t.test(rev_fp, rev_vickrey, paired = TRUE)
cat(sprintf("Paired t-test p-value:  %.4f\n", t_result$p.value))
```

```
#> Paired t-test p-value:  0.1789
```

```
cat("Conclusion:", ifelse(t_result$p.value > 0.05,
    "No significant difference -- revenue equivalence holds.",
    "Significant difference detected."), "\n")
```

```
#> Conclusion: No significant difference -- revenue equivalence holds.
```

Both formats produce mean revenues close to the theoretical value of $(N-1)/(N+1) = 4/6 \approx 0.667$. The paired t-test confirms no statistically significant difference, consistent with the revenue equivalence theorem. Note that while *expected* revenues are equal, individual auction revenues differ — in the first-price format, revenue variance is typically lower because the winner pays a bid that is already close to the second-highest valuation.

32.5 Extensions

Revenue equivalence breaks down when its assumptions are violated, opening the door to richer auction design:

- **Risk-averse bidders** bid more aggressively in first-price auctions to reduce the risk of losing, giving the seller higher expected revenue than under the Vickrey format.
- **Common-value auctions** introduce the *winner's curse*: the winner has the most optimistic estimate, leading to overbidding. Bidders must shade bids further to avoid overpaying.
- **Optimal auction design** (33) uses Myerson's framework to design revenue-maximizing mechanisms, including reserve prices that screen out low-value bidders.
- **Multi-unit auctions** generalize to settings where multiple identical items are sold simultaneously, using uniform-price or discriminatory formats.
- **Asymmetric bidders** with valuations drawn from different distributions break revenue equivalence and make equilibrium computation substantially harder.

For a comprehensive treatment of auction theory, see Myerson (1991) (Chapter 8) and Nisan et al. (2007) (Chapters 9–11).

Exercises

1. **Reserve prices.** Modify the `simulate_auctions()` function to include a reserve price $r = 0.3$. If no bid exceeds r, the item is unsold (revenue $= 0$). Run 1000 simulations with $N = 3$ bidders and compare mean revenue to the case without a reserve price. Does a reserve price increase expected revenue? Explain why using the theory of optimal auctions.

2. **All-pay auction.** In an all-pay auction, every bidder pays their bid but only the highest bidder wins. With N bidders and $v_i \sim \text{Uniform}(0, 1)$, the symmetric equilibrium strategy is $b^*(v) = \frac{N-1}{N} v^N$. Implement a simulation of the all-pay auction and verify that the revenue equivalence theorem holds by comparing mean revenue to the first-price and Vickrey formats.

3. **Number of bidders.** Using the simulation framework from this chapter, plot expected seller revenue as a function of the number of bidders $N \in \{2, 3, 5, 10, 20, 50\}$ for both first-price and Vickrey auctions. Overlay the theoretical curve $(N - 1)/(N + 1)$. What happens as $N \to \infty$?

Solutions appear in D.

Chapter 33

Mechanism Design

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

Inverse game theory: the revelation principle, incentive compatibility, the VCG mechanism, and implementation in R for public goods provision.

Learning objectives

- Explain mechanism design as the "inverse" problem of game theory: designing rules to achieve desired outcomes.
- State the revelation principle and its role in simplifying mechanism design.
- Define incentive compatibility and individual rationality.
- Implement the VCG mechanism for a public goods provision problem in R.

33.1 Motivation

In standard game theory, we take the rules of the game as given and predict outcomes. Mechanism design flips this: given a desired outcome, what rules should we impose? This "reverse engineering" perspective is central to real-world institutional design — from spectrum auctions and organ allocation systems to voting rules and matching markets.

Consider a city council deciding whether to build a public park. Each citizen has a private valuation for the park, which they may misrepresent if doing so serves their interest. Can we design a mechanism — a set of rules for eliciting preferences and making decisions — that induces truthful reporting and achieves the

socially optimal outcome? The Vickrey-Clarke-Groves (VCG) mechanism shows that the answer is yes, under certain conditions (Nisan et al., 2007, Chapter 9).

33.2 Theory

33.2.1 The mechanism design problem

A **mechanism** specifies (i) a message space for each agent and (ii) an outcome function mapping messages to decisions and transfers. A mechanism **implements** a social choice function if, in equilibrium, the mechanism's outcome matches what the social choice function prescribes.

33.2.2 The revelation principle

Theorem: Revelation Principle

For any mechanism in which agents play a Bayesian Nash equilibrium, there exists a **direct** mechanism — one where agents simply report their types — that achieves the same outcome and where truthful reporting is an equilibrium.

The revelation principle dramatically simplifies the designer's problem: instead of searching over all possible mechanisms, we can restrict attention to direct, truthful mechanisms without loss of generality.

33.2.3 Incentive compatibility and individual rationality

A mechanism is **incentive compatible (IC)** if truthful reporting is an equilibrium strategy for every agent. It is **individually rational (IR)** if every agent's expected payoff from participating is at least as large as their outside option (typically zero).

These two constraints define the boundary of what is achievable. A mechanism that violates IC will produce distorted reports; one that violates IR will see agents opt out.

33.2.4 The VCG mechanism

The Vickrey-Clarke-Groves mechanism works as follows for a public decision:

1. Each agent i reports a valuation $\hat{v}_i$ for each possible outcome.
2. The mechanism chooses the outcome k^* that maximizes total reported welfare: $k^* = \arg\max_k \sum_i \hat{v}_i(k)$.
3. Each agent pays a **Clarke tax**: the externality they impose on others.

$$t_i = \max_k \sum_{j \neq i} \hat{v}_j(k) - \sum_{j \neq i} \hat{v}_j(k^*) \tag{33.1}$$

The Clarke tax equals the difference between what the other agents *could* have achieved (without i) and what they *do* achieve (with i's presence influencing the decision). Under VCG, truthful reporting is a dominant strategy (Nisan et al., 2007, Chapter 9).

33.3 Implementation in R

33.3.1 VCG mechanism for public goods

```
vcg_mechanism <- function(valuations) {
  # valuations: matrix where rows = agents, cols = outcomes
  # Returns: chosen outcome, transfers, and payoffs
  n_agents <- nrow(valuations)
  n_outcomes <- ncol(valuations)

  # Social welfare for each outcome
  social_welfare <- colSums(valuations)
  chosen <- which.max(social_welfare)

  # Clarke tax for each agent
  transfers <- numeric(n_agents)
  for (i in seq_len(n_agents)) {
    # Best outcome for others without agent i's influence
    welfare_without_i <- colSums(valuations[-i, , drop = FALSE])
    best_without_i <- max(welfare_without_i)
    # What others get under the chosen outcome
    others_at_chosen <- sum(valuations[-i, chosen])
    transfers[i] <- best_without_i - others_at_chosen
  }

  # Agent payoff = valuation of chosen outcome - tax
  payoffs <- valuations[, chosen] - transfers

  list(
    chosen_outcome = chosen,
    social_welfare = social_welfare[chosen],
    transfers = transfers,
    payoffs = payoffs,
    is_truthful_dominant = TRUE  # by VCG theory
```

```
  )
}

# Majority voting mechanism (for comparison)
majority_voting <- function(valuations) {
  n_agents <- nrow(valuations)
  n_outcomes <- ncol(valuations)

  # Each agent votes for their most-preferred outcome
  votes <- apply(valuations, 1, which.max)
  vote_counts <- tabulate(votes, nbins = n_outcomes)
  chosen <- which.max(vote_counts)

  list(
    chosen_outcome = chosen,
    social_welfare = sum(valuations[, chosen]),
    transfers = rep(0, n_agents),
    payoffs = valuations[, chosen]
  )
}
```

33.3.2 Comparing mechanisms across many instances

```
set.seed(42)
n_simulations <- 500
n_agents <- 3
n_outcomes <- 2  # Build park (1) vs. don't build (2)

results <- map_dfr(seq_len(n_simulations), function(sim_id) {
  # Random valuations: agents value "build" differently, "don't
  ↪  build" = 0
  v_build <- runif(n_agents, min = -2, max = 5)
  valuations <- cbind(v_build, rep(0, n_agents))

  vcg_result <- vcg_mechanism(valuations)
  mv_result <- majority_voting(valuations)

  # Optimal social welfare (first-best)
  optimal_welfare <- max(colSums(valuations))

  tibble(
    sim_id = sim_id,
    mechanism = c("VCG", "Majority voting", "First-best"),
```

```
    welfare = c(vcg_result$social_welfare,
                mv_result$social_welfare,
                optimal_welfare)
  )
})
```

33.3.3 Figure 1: Social welfare comparison

```
welfare_summary <- results |>
  group_by(mechanism) |>
  summarise(
    mean_welfare = mean(welfare),
    se_welfare = sd(welfare) / sqrt(n()),
    .groups = "drop"
  ) |>
  mutate(mechanism = factor(mechanism,
                            levels = c("First-best", "VCG",
                             ↪  "Majority voting")))

p_welfare <- ggplot(welfare_summary,
                    aes(x = mechanism, y = mean_welfare, fill =
                     ↪  mechanism)) +
  geom_col(alpha = 0.85, width = 0.6) +
  geom_errorbar(aes(ymin = mean_welfare - 1.96 * se_welfare,
                    ymax = mean_welfare + 1.96 * se_welfare),
                width = 0.15) +
  geom_text(aes(label = sprintf("%.2f", mean_welfare)),
            vjust = -0.8, size = 3.5) +
  scale_fill_manual(values = okabe_ito[c(3, 1, 2)]) +
  scale_y_continuous(name = "Mean social welfare",
                     expand = expansion(mult = c(0, 0.15))) +
  labs(x = "Mechanism",
       title = "Social Welfare Across Mechanisms",
       subtitle = glue("{n_simulations} simulations, {n_agents}
        ↪  agents, ",
                       "public goods decision")) +
  theme_publication() +
  theme(legend.position = "none")

p_welfare
```

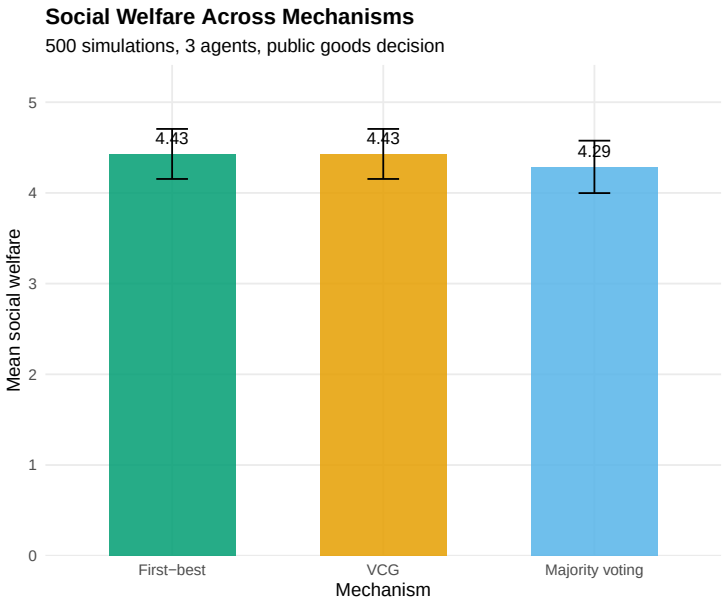


Figure 33.1: Social welfare under VCG and majority voting compared to the first-best optimum. VCG always achieves the efficient outcome; majority voting sometimes selects the wrong alternative.

```
save_pub_fig(p_welfare, "mechanism-welfare-comparison", width =
 ↪  6, height = 5)
```

33.3.4 Figure 2: VCG transfer payments by agent

```
set.seed(42)
n_sample <- 20

transfer_data <- map_dfr(seq_len(n_sample), function(sim_id) {
  v_build <- runif(n_agents, min = -2, max = 5)
  valuations <- cbind(v_build, rep(0, n_agents))
  vcg_result <- vcg_mechanism(valuations)

  tibble(
    instance = factor(sim_id),
    agent = paste("Agent", seq_len(n_agents)),
    transfer = vcg_result$transfers,
    valuation = v_build
  )
})

p_transfers <- ggplot(transfer_data,
                      aes(x = instance, y = transfer, fill =
                       ↪  agent)) +
  geom_col(position = position_dodge(width = 0.7), width = 0.6,
   ↪  alpha = 0.85) +
  scale_fill_manual(values = okabe_ito[1:3], name = NULL) +
  scale_y_continuous(name = "Clarke tax payment") +
  labs(x = "Auction instance",
       title = "Transfer Payments in the VCG Mechanism",
       subtitle = "Positive tax = agent is pivotal for the
        ↪  decision") +
  theme_publication() +
  theme(axis.text.x = element_text(size = 7))

p_transfers
```

```
save_pub_fig(p_transfers, "vcg-transfers", width = 7, height = 5)
```

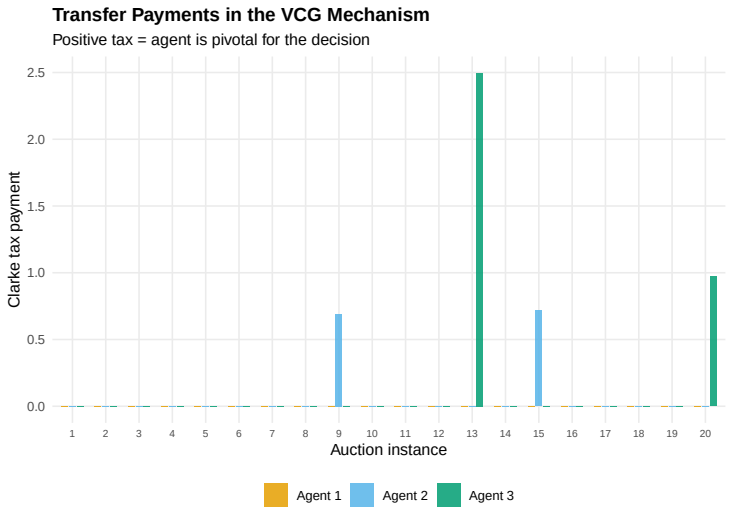


Figure 33.2: Clarke tax payments in the VCG mechanism for a representative sample of 20 instances. Agents who are pivotal — those whose presence changes the chosen outcome — pay a positive tax.

33.4 Worked example

Three citizens have private valuations for building a public park. The alternative is the status quo (no park, value 0 for all). We apply the VCG mechanism and compare it to majority voting.

```
# Agent valuations for: (Build park, No park)
# Agent 1 loves parks, Agent 2 is mildly positive, Agent 3
 ↪  prefers no park
valuations <- matrix(c(
   8,  0,   # Agent 1: values park at 8
   3,  0,   # Agent 2: values park at 3
  -2,  0    # Agent 3: dislikes park (cost = 2)
), nrow = 3, byrow = TRUE,
   dimnames = list(paste("Agent", 1:3), c("Build", "No build")))

cat("Agent valuations:\n")
```

```
#> Agent valuations:
```

```
print(valuations)
```

```
#>         Build No build
```

```
#> Agent 1     8        0
#> Agent 2     3        0
#> Agent 3    -2        0
```

```
# VCG mechanism
vcg <- vcg_mechanism(valuations)
cat(sprintf("\n--- VCG Mechanism ---\n"))
```

```
#>
#> --- VCG Mechanism ---
```

```
cat(sprintf("Chosen outcome: %s\n",
            colnames(valuations)[vcg$chosen_outcome]))
```

```
#> Chosen outcome: Build
```

```
cat(sprintf("Social welfare:  %.1f\n", vcg$social_welfare))
```

```
#> Social welfare:  9.0
```

```
cat(sprintf("\nClarke taxes:\n"))
```

```
#>
#> Clarke taxes:
```

```
for (i in 1:3) {
  cat(sprintf("  Agent %d: tax = %.1f, net payoff = %.1f\n",
              i, vcg$transfers[i], vcg$payoffs[i]))
}
```

```
#>   Agent 1: tax = 0.0, net payoff = 8.0
#>   Agent 2: tax = 0.0, net payoff = 3.0
#>   Agent 3: tax = 0.0, net payoff = -2.0
```

```
# Majority voting
mv <- majority_voting(valuations)
cat(sprintf("\n--- Majority Voting ---\n"))
```

```
#>
#> --- Majority Voting ---
```

```
cat(sprintf("Chosen outcome: %s\n",
            colnames(valuations)[mv$chosen_outcome]))
```

```
#> Chosen outcome: Build
```

```
cat(sprintf("Social welfare:  %.1f\n", mv$social_welfare))
```

```
#> Social welfare:  9.0
```

```
# Verify incentive compatibility: what if Agent 3 lies?
cat("\n--- Incentive Compatibility Check ---\n")
```

```
#>
#> --- Incentive Compatibility Check ---
```

```
cat("What if Agent 3 reports valuation -10 instead of -2?\n")
```

```
#> What if Agent 3 reports valuation -10 instead of -2?
```

```
valuations_lie <- valuations
valuations_lie[3, 1] <- -10
vcg_lie <- vcg_mechanism(valuations_lie)
cat(sprintf("Outcome with lie: %s\n",
            colnames(valuations)[vcg_lie$chosen_outcome]))
```

```
#> Outcome with lie: Build
```

```
cat(sprintf("Agent 3's true payoff under lie: %.1f\n",
            valuations[3, vcg_lie$chosen_outcome] -
 ↪  vcg_lie$transfers[3]))
```

```
#> Agent 3's true payoff under lie: -2.0
```

```
cat(sprintf("Agent 3's true payoff under truth: %.1f\n",
 ↪  vcg$payoffs[3]))
```

```
#> Agent 3's true payoff under truth: -2.0
```

```
cat("Lying is not profitable -- VCG is incentive compatible.\n")
```

```
#> Lying is not profitable -- VCG is incentive compatible.
```

The VCG mechanism chooses to build the park (total value $8 + 3 - 2 = 9 > 0$). Agent 3, who opposes the park, pays no Clarke tax because removing Agent 3 would not change the outcome — the other agents still prefer building. Majority voting also selects "Build" (2–1 vote), but this is coincidental; majority voting can fail when a minority has intense preferences that outweigh the mild preferences of the majority.

33.5 Extensions

- **Myerson's optimal auction** (32) derives the revenue-maximizing mechanism by applying mechanism design to auction settings, introducing virtual valuations and reserve prices.
- **Budget balance** — the VCG mechanism generally runs a deficit (total taxes do not cover costs). The AGV (Arrow–d'Aspremont–Gerard-Varet) mechanism achieves budget balance but only satisfies IC in a Bayesian sense.
- **Combinatorial auctions** extend VCG to settings with multiple items and complementarities, though computational complexity becomes a major concern.
- For the foundational treatment, see Nisan et al. (2007) (Chapters 9–12) and Roth (2002) on practical market design.

Exercises

1. **Pivotal agents.** In the worked example, modify Agent 2's valuation for the park to 1 (so total value of building becomes $8+1-2 = 7$). Recompute the VCG outcome and Clarke taxes. Now set Agent 1's valuation to 1 so building yields $1 + 1 - 2 = 0$. What happens? Which agents are pivotal, and how do their taxes change?

2. **Budget balance.** Run 500 simulations of the VCG mechanism with 5 agents and a binary public goods decision ($v_i \sim \text{Uniform}(-2, 5)$). Compute the total Clarke tax revenue in each simulation. What fraction of simulations have positive total tax revenue? Plot the distribution of total taxes and explain why VCG generally does not achieve budget balance.

3. **Manipulation under majority voting.** Construct a 3-agent, 3-outcome example where majority voting leads to a Condorcet cycle

(no Condorcet winner), while VCG selects a unique efficient outcome. Implement both mechanisms and show that the VCG outcome maximizes social welfare.

Solutions appear in D.

Chapter 34

Matching Markets

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

> Two-sided matching: stable matching, the Gale-Shapley deferred acceptance algorithm, strategy-proofness, and implementation in R.

Learning objectives

- Define stable matching and explain why unstable matchings unravel.
- Implement the Gale-Shapley deferred acceptance algorithm from scratch in base R.
- Prove that proposer-optimal and receiver-optimal stable matchings are distinct and that the algorithm is strategy-proof for the proposing side.
- Visualize matchings as bipartite graphs and compare welfare across proposer-optimal and receiver-optimal outcomes.

34.1 Motivation

Each year, approximately 40,000 medical school graduates in the United States are matched to residency programs through the National Resident Matching Program (NRMP). The system uses the Gale-Shapley deferred acceptance algorithm, which produces a stable matching — one where no hospital-resident pair would prefer to bypass the system and match with each other directly.

The practical significance of stability was demonstrated when the NRMP switched to a resident-proposing algorithm in 1998 (previously hospital-proposing), improving outcomes for residents. The theoretical foundation for this was laid by Roth (2002), building on the seminal work of Gale and Shapley

(1962), which earned Lloyd Shapley and Alvin Roth the 2012 Nobel Prize in Economics.

34.2 Theory

34.2.1 Stable matching

Consider two disjoint sets of equal size n: **proposers** $P = \{p_1, \dots, p_n\}$ and **receivers** $R = \{r_1, \dots, r_n\}$. Each agent has a strict preference ordering over agents on the other side.

Definition: Stable Matching

A matching $\mu : P \to R$ (a bijection) is **stable** if there is no **blocking pair** — a pair (p_i, r_j) where p_i prefers r_j to their current match $\mu(p_i)$ and r_j prefers p_i to their current match $\mu^{-1}(r_j)$.

Stability is the key property that prevents unraveling: if a matching is unstable, some pair has an incentive to deviate, undermining the system.

34.2.2 The Gale-Shapley algorithm

The **deferred acceptance** algorithm proceeds in rounds:

1. Each unmatched proposer proposes to the highest-ranked receiver they have not yet proposed to.
2. Each receiver with multiple proposals holds the most preferred and rejects the rest.
3. Rejected proposers become unmatched and propose again in the next round.
4. The algorithm terminates when every proposer is matched (or has been rejected by all receivers).

Theorem: Gale-Shapley Properties

The Gale-Shapley algorithm always terminates and produces a stable matching. When proposers propose, the result is the **proposer-optimal** stable matching: every proposer is matched to their best achievable partner across all stable matchings. Simultaneously, it is the **receiver-pessimal** stable matching.

34.2.3 Strategy-proofness

The Gale-Shapley algorithm is **strategy-proof** for the proposing side: no proposer can improve their outcome by misreporting preferences. However, re-

ceivers *can* potentially benefit from strategic misreporting — a result with practical implications for the design of matching markets (Roth, 2002).

34.3 Implementation in R

34.3.1 Gale-Shapley algorithm

```
gale_shapley <- function(proposer_prefs, receiver_prefs)
{
    # proposer_prefs: list of length n, each element a vector of
     ↪ receiver indices
    # receiver_prefs: list of length n, each element a vector of
     ↪ proposer indices
    # Returns: named vector mapping proposer index -> receiver
     ↪ index

  n <- length(proposer_prefs)

  # Precompute receiver rankings for O(1) comparison
  receiver_rank <- matrix(0L, nrow = n, ncol = n)
  for (r in seq_len(n)) {
    receiver_rank[r, receiver_prefs[[r]]] <- seq_len(n)
  }

  matched_to     <- rep(NA_integer_, n)  # proposer -> receiver
  held_by        <- rep(NA_integer_, n)  # receiver -> proposer
↪  (current holder)
  next_proposal  <- rep(1L, n)           # which pref to propose
↪  to next
  free_proposers <- seq_len(n)
  rounds         <- 0L

  while (length(free_proposers) > 0) {
    rounds <- rounds + 1L
    new_free <- integer(0)

    for (p in free_proposers) {
      r <- proposer_prefs[[p]][next_proposal[p]]
      next_proposal[p] <- next_proposal[p] + 1L

      if (is.na(held_by[r])) {
        # Receiver is free -- tentatively accept
        held_by[r]    <- p
```

```
        matched_to[p] <- r
      } else {
        current <- held_by[r]
        if (receiver_rank[r, p] < receiver_rank[r, current]) {
          # Receiver prefers new proposer
          held_by[r]         <- p
          matched_to[p]      <- r
          matched_to[current] <- NA_integer_
          new_free <- c(new_free, current)
        } else {
          # Receiver keeps current -- proposer stays free
          new_free <- c(new_free, p)
        }
      }
    }
    free_proposers <- new_free
  }

  list(matching = matched_to, rounds = rounds)
}
```

34.3.2 Helper: compute welfare

```
matching_welfare <- function(matching, proposer_prefs,
 ↪  receiver_prefs) {
  n <- length(matching)

  # Proposer welfare: rank of matched receiver (1 = best)
  p_ranks <- vapply(seq_len(n), function(p) {
    which(proposer_prefs[[p]] == matching[p])
  }, integer(1))

  # Receiver welfare: rank of matched proposer (1 = best)
  r_ranks <- vapply(seq_len(n), function(r) {
    p <- which(matching == r)
    which(receiver_prefs[[r]] == p)
  }, integer(1))

  list(
    proposer_ranks = p_ranks,
    receiver_ranks = r_ranks,
    proposer_mean  = mean(p_ranks),
    receiver_mean  = mean(r_ranks)
```

```
  )
}
```

34.3.3 Figure 1: Matching visualization

```
set.seed(42)
n <- 6

# Generate random strict preferences
proposer_prefs <- lapply(seq_len(n), function(i)
 ↪  sample(seq_len(n)))
receiver_prefs <- lapply(seq_len(n), function(i)
 ↪  sample(seq_len(n)))

result <- gale_shapley(proposer_prefs, receiver_prefs)
matching <- result$matching

# Build data for bipartite graph
proposer_nodes <- tibble(
  x = 0, y = seq_len(n),
  label = paste("H", seq_len(n)),
  side = "Hospital"
)

receiver_nodes <- tibble(
  x = 1, y = seq_len(n),
  label = paste("R", seq_len(n)),
  side = "Resident"
)

edges <- tibble(
  x = 0, xend = 1,
  y = seq_len(n), yend = matching
)

nodes <- bind_rows(proposer_nodes, receiver_nodes)

p_matching <- ggplot() +
  geom_segment(data = edges, aes(x = x, xend = xend, y = y, yend
   ↪  = yend),
               colour = okabe_ito[1], linewidth = 0.9, alpha =
                ↪  0.7) +
  geom_point(data = nodes, aes(x = x, y = y, colour = side), size
   ↪  = 5) +
```

```
  geom_text(data = nodes, aes(x = x, y = y, label = label),
            size = 2.8, fontface = "bold",
            nudge_x = ifelse(nodes$x == 0, -0.08, 0.08)) +
  scale_colour_manual(values = okabe_ito[c(2, 6)], name = NULL) +
  scale_x_continuous(limits = c(-0.2, 1.2), breaks = c(0, 1),
                     labels = c("Hospitals\n(proposers)",
                                "Residents\n(receivers)")) +
  scale_y_continuous(breaks = NULL) +
  labs(title = "Proposer-Optimal Stable Matching",
       subtitle = glue("Gale-Shapley converged in {result$rounds}
        ↪  rounds"),
       x = NULL, y = NULL) +
  theme_publication() +
  theme(panel.grid.major = element_blank(),
        axis.text.y = element_blank())

p_matching
```

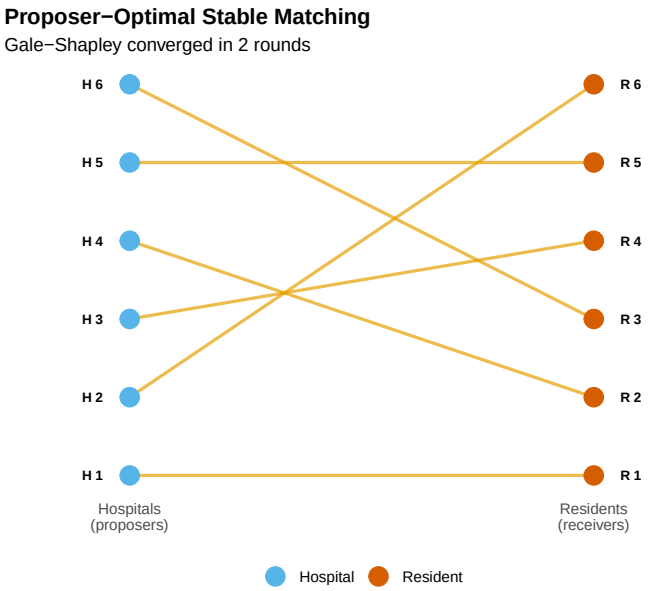


Figure 34.1: Bipartite graph of the proposer-optimal stable matching for 6 hospitals and 6 residents. Left nodes are hospitals (proposers), right nodes are residents (receivers), and line segments connect matched pairs.

```
save_pub_fig(p_matching, "matching-bipartite", width = 6, height
 ↪  = 5)
```

34.3.4 Figure 2: Proposer advantage

```
set.seed(42)
n <- 8
n_instances <- 200

welfare_data <- map_dfr(seq_len(n_instances), function(sim_id) {
  p_prefs <- lapply(seq_len(n), function(i) sample(seq_len(n)))
  r_prefs <- lapply(seq_len(n), function(i) sample(seq_len(n)))

  # Proposer-optimal: proposers propose
  po <- gale_shapley(p_prefs, r_prefs)
  w_po <- matching_welfare(po$matching, p_prefs, r_prefs)

  # Receiver-optimal: receivers propose (swap roles)
  ro <- gale_shapley(r_prefs, p_prefs)
  # ro$matching maps receiver -> proposer, invert for welfare
  ro_matching <- integer(n)
  for (r in seq_len(n)) ro_matching[ro$matching[r]] <- r
  w_ro <- matching_welfare(ro_matching, p_prefs, r_prefs)

  tibble(
    sim_id = sim_id,
    side = rep(c("Proposers", "Receivers"), 2),
    variant = rep(c("Proposer-optimal", "Receiver-optimal"), each
     ↪  = 2),
    mean_rank = c(w_po$proposer_mean, w_po$receiver_mean,
                  w_ro$proposer_mean, w_ro$receiver_mean)
  )
})

welfare_summary <- welfare_data |>
  group_by(side, variant) |>
  summarise(avg_rank = mean(mean_rank),
            se_rank = sd(mean_rank) / sqrt(n()),
            .groups = "drop")

p_advantage <- ggplot(welfare_summary,
                      aes(x = variant, y = avg_rank, fill =
                       ↪  side)) +
```

```
  geom_col(position = position_dodge(width = 0.7), width = 0.6,
   ↪  alpha = 0.85) +
  geom_errorbar(aes(ymin = avg_rank - 1.96 * se_rank,
                    ymax = avg_rank + 1.96 * se_rank),
                position = position_dodge(width = 0.7), width =
                 ↪  0.15) +
  geom_text(aes(label = sprintf("%.2f", avg_rank)),
            position = position_dodge(width = 0.7), vjust = -0.8,
             ↪  size = 3) +
  scale_fill_manual(values = okabe_ito[c(2, 6)], name = NULL) +
  scale_y_continuous(name = "Average rank of matched partner\n(1
   ↪  = best)",
                     expand = expansion(mult = c(0, 0.15))) +
  labs(x = "Matching variant",
       title = "Proposer Advantage in Stable Matching",
       subtitle = glue("{n_instances} random instances, n = {n}
        ↪  per side")) +
  theme_publication()

p_advantage
```

```
save_pub_fig(p_advantage, "matching-proposer-advantage", width =
 ↪  6, height = 5)
```

34.4 Worked example

We trace the Gale-Shapley algorithm step by step for 5 hospitals and 5 residents.

```
n <- 5
hospital_names <- paste("H", 1:n, sep = "")
resident_names <- paste("R", 1:n, sep = "")

# Fixed preferences for reproducibility
h_prefs <- list(
  c(2, 3, 1, 5, 4),  # H1 prefers R2 > R3 > R1 > R5 > R4
  c(1, 3, 5, 2, 4),  # H2 prefers R1 > R3 > R5 > R2 > R4
  c(1, 2, 4, 5, 3),  # H3 prefers R1 > R2 > R4 > R5 > R3
  c(3, 1, 2, 4, 5),  # H4 prefers R3 > R1 > R2 > R4 > R5
  c(2, 4, 1, 3, 5)   # H5 prefers R2 > R4 > R1 > R3 > R5
)

r_prefs <- list(
```

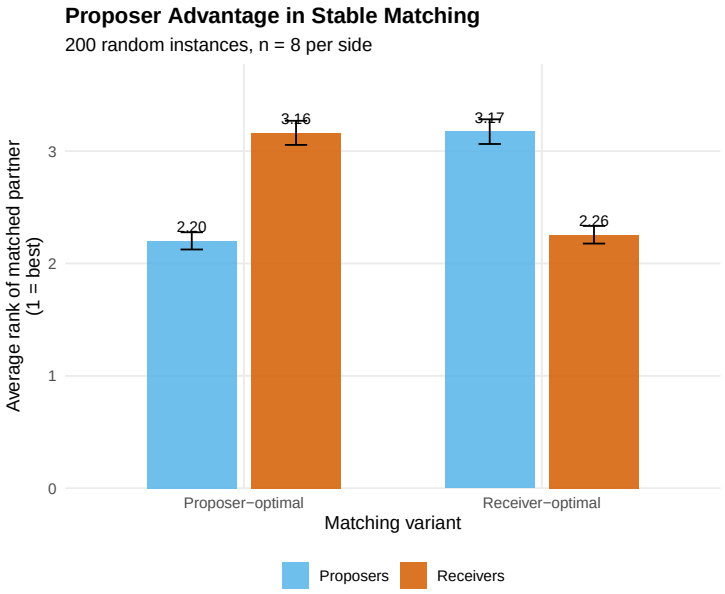


Figure 34.2: Average rank of matched partner under proposer-optimal and receiver-optimal stable matchings across 200 random instances. Proposers obtain significantly better matches when they propose, confirming the structural advantage of the proposing side.

```
  c(3, 1, 4, 2, 5),  # R1 prefers H3 > H1 > H4 > H2 > H5
  c(1, 5, 2, 3, 4),  # R2 prefers H1 > H5 > H2 > H3 > H4
  c(4, 2, 1, 3, 5),  # R3 prefers H4 > H2 > H1 > H3 > H5
  c(3, 5, 1, 2, 4),  # R4 prefers H3 > H5 > H1 > H2 > H4
  c(1, 2, 3, 5, 4)   # R5 prefers H1 > H2 > H3 > H5 > H4
)

cat("Hospital preferences (most to least preferred resident):\n")
```

```
#> Hospital preferences (most to least preferred resident):
```

```
for (h in 1:n) {
  cat(sprintf("  %s: %s\n", hospital_names[h],
              paste(resident_names[h_prefs[[h]]], collapse = " >
               ↪  ")))
}
```

```
#>   H1: R2 > R3 > R1 > R5 > R4
#>   H2: R1 > R3 > R5 > R2 > R4
#>   H3: R1 > R2 > R4 > R5 > R3
#>   H4: R3 > R1 > R2 > R4 > R5
#>   H5: R2 > R4 > R1 > R3 > R5
```

```
cat("\nResident preferences (most to least preferred
 ↪  hospital):\n")
```

```
#>
#> Resident preferences (most to least preferred hospital):
```

```
for (r in 1:n) {
  cat(sprintf("  %s: %s\n", resident_names[r],
              paste(hospital_names[r_prefs[[r]]], collapse = " >
               ↪  ")))
}
```

```
#>   R1: H3 > H1 > H4 > H2 > H5
#>   R2: H1 > H5 > H2 > H3 > H4
#>   R3: H4 > H2 > H1 > H3 > H5
#>   R4: H3 > H5 > H1 > H2 > H4
#>   R5: H1 > H2 > H3 > H5 > H4
```

```
# Run hospital-proposing Gale-Shapley
result <- gale_shapley(h_prefs, r_prefs)

cat(sprintf("\nHospital-proposing Gale-Shapley (converged in %d
 ↪  rounds):\n",
            result$rounds))
```

```
#>
#> Hospital-proposing Gale-Shapley (converged in 3 rounds):
```

```
for (h in 1:n) {
  cat(sprintf("  %s --> %s\n", hospital_names[h],
              resident_names[result$matching[h]]))
}
```

```
#>   H1 --> R2
#>   H2 --> R5
#>   H3 --> R1
#>   H4 --> R3
#>   H5 --> R4
```

```
# Verify stability
w <- matching_welfare(result$matching, h_prefs, r_prefs)
cat(sprintf("\nMean hospital rank of match: %.1f (of %d)\n",
            w$proposer_mean, n))
```

```
#>
#> Mean hospital rank of match: 1.6 (of 5)
```

```
cat(sprintf("Mean resident rank of match: %.1f (of %d)\n",
            w$receiver_mean, n))
```

```
#> Mean resident rank of match: 1.4 (of 5)
```

```
# Run resident-proposing version
result_rp <- gale_shapley(r_prefs, h_prefs)
rp_matching <- integer(n)
for (r in 1:n) rp_matching[result_rp$matching[r]] <- r
cat(sprintf("\nResident-proposing Gale-Shapley:\n"))
```

```
#>
#> Resident-proposing Gale-Shapley:
```

```
for (h in 1:n) {
  cat(sprintf("  %s --> %s\n", hospital_names[h],
              resident_names[rp_matching[h]]))
}
```

```
#>   H1 --> R2
#>   H2 --> R5
#>   H3 --> R1
#>   H4 --> R3
#>   H5 --> R4
```

```
w_rp <- matching_welfare(rp_matching, h_prefs, r_prefs)
cat(sprintf("\nMean hospital rank of match: %.1f\n",
 ↪  w_rp$proposer_mean))
```

```
#>
#> Mean hospital rank of match: 1.6
```

```
cat(sprintf("Mean resident rank of match: %.1f\n",
 ↪  w_rp$receiver_mean))
```

```
#> Mean resident rank of match: 1.4
```

The hospital-proposing version yields better outcomes for hospitals (lower mean rank of match), while the resident-proposing version favours residents. This asymmetry motivated the NRMP's switch to a resident-proposing algorithm in 1998 — a change that improved resident welfare without sacrificing stability.

34.5 Extensions

- **Many-to-one matching** generalizes the model to settings where each hospital has multiple positions. The algorithm extends naturally, with hospitals holding a quota of proposals.
- **Kidney exchange** (Roth (2002)) applies matching theory to organ donation, where patients with incompatible donors can swap. Top trading cycles and chains extend the stable matching framework.
- **School choice** uses deferred acceptance to assign students to public schools (Boston, New York City), balancing family preferences with school priorities.
- **Matching with contracts** (Roth, 2002) unifies many-to-one matching and auction theory into a single framework, with applications to military cadet branching and supply chains.

Exercises

1. **Uniqueness.** Construct a 3-by-3 matching instance where the proposer-optimal and receiver-optimal stable matchings are identical — meaning there is a unique stable matching. Verify your example by running `gale_shapley()` with both sides proposing. What structural property of the preference lists guarantees uniqueness?

2. **Strategic manipulation.** Using the 5-by-5 worked example, suppose Resident 1 misreports their preferences as $H_1 > H_3 > H_4 > H_2 > H_5$ (promoting H_1 to first choice). Run the hospital-proposing Gale-Shapley with this manipulated preference list. Does Resident 1 get a better or worse outcome? Explain why the algorithm is *not* strategy-proof for the receiving side.

3. **Scaling behaviour.** Run the Gale-Shapley algorithm for $n \in \{10, 50, 100, 500, 1000\}$ with random preferences (50 instances each). Record the number of rounds until convergence and plot it against n. What is the empirical relationship? Compare to the theoretical bound of $O(n^2)$.

Solutions appear in D.

Chapter 35

Bargaining Theory

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

Nash bargaining and ultimatum games — axiomatic solutions, alternating-offers models, and computational implementations in R.

Learning objectives

- State the four axioms of the Nash bargaining solution and derive the solution as the maximizer of the Nash product.
- Formulate the Rubinstein alternating-offers model and compute its subgame perfect equilibrium for given discount factors.
- Implement the Nash bargaining solution and Rubinstein equilibrium computations in R.
- Visualize feasible bargaining sets, Nash product contours, and equilibrium shares as functions of patience.

35.1 Motivation

A union and a firm sit across the negotiating table. The union wants higher wages; the firm wants lower labor costs. Both prefer a deal to a strike, but each wants the deal tilted in their favor. How should they split the surplus?

This question arises whenever two parties must agree on how to divide gains from trade: merger negotiations, international treaties, or even splitting a dinner bill. Game theory provides two foundational approaches. The **Nash bargaining solution** uses axioms to identify a unique fair division. The **Rubinstein alternating-offers model** derives the same outcome from a non-cooperative bargaining protocol where impatience drives agreement.

35.2 Theory

35.2.1 The bargaining problem

A **bargaining problem** is a pair (S, d) where $S \subset \mathbb{R}^2$ is a compact, convex set of feasible payoff pairs and $d = (d_1, d_2) \in S$ is the **disagreement point** — the payoffs each player receives if no agreement is reached. The set of individually rational outcomes is $\{(u_1, u_2) \in S : u_1 \geq d_1, u_2 \geq d_2\}$.

35.2.2 Nash bargaining solution

Definition: Nash Bargaining Solution

The **Nash bargaining solution** selects the feasible payoff pair that maximizes the Nash product:

$$(u_1^*, u_2^*) = \arg\max_{(u_1,u_2)\in S} (u_1 - d_1)(u_2 - d_2) \tag{35.1}$$

subject to $u_1 \geq d_1$ and $u_2 \geq d_2$.

Nash (1950) showed this is the unique solution satisfying four axioms:

1. **Pareto efficiency** — no feasible outcome makes both players strictly better off.
2. **Symmetry** — if the problem is symmetric, the solution gives equal payoffs.
3. **Independence of irrelevant alternatives (IIA)** — removing feasible outcomes that are not chosen does not change the solution.
4. **Scale invariance** — the solution is invariant to positive affine transformations of utility.

For a linear frontier $u_1 + u_2 = v$ with disagreement point $(0, 0)$, the Nash bargaining solution gives each player $v/2$ — an equal split.

35.2.3 Asymmetric Nash bargaining

When players have different bargaining powers α and $1 - \alpha$ (with $\alpha \in (0, 1)$), the **asymmetric Nash bargaining solution** maximizes the generalized Nash product:

$$(u_1^*, u_2^*) = \arg\max_{(u_1,u_2)\in S} (u_1 - d_1)^{\alpha}(u_2 - d_2)^{1-\alpha} \tag{35.2}$$

35.2.4 Rubinstein alternating-offers model

Rubinstein (1982) modeled bargaining as a sequential game. Two players take turns proposing how to split a surplus of size 1. Player 1 proposes in odd periods, Player 2 in even periods. Each player has a discount factor $\delta_i \in (0,1)$ — waiting is costly.

Theorem: Rubinstein Equilibrium

The unique subgame perfect equilibrium of the alternating-offers game gives Player 1 a share:

$$x_1^* = \frac{1-\delta_2}{1-\delta_1\delta_2} \tag{35.3}$$

and Player 2 receives $1-x_1^*$. When $\delta_1 = \delta_2 = \delta$, the split converges to $(1/2, 1/2)$ as $\delta \to 1$.

The intuition is that a more patient player (higher δ) obtains a larger share because they can credibly wait longer. The first-mover advantage shrinks as both players become more patient.

35.3 Implementation in R

35.3.1 Nash bargaining solution computation

```
nash_bargaining <- function(frontier_fn, d = c(0, 0), alpha =
 ↪  0.5,
                            n_grid = 1000, u1_max = NULL) {

  if (is.null(u1_max)) {
    # Find where frontier_fn(u1) drops to d[2] by searching
     ↪  adaptively
    hi <- d[1] + 1
    while (is.finite(frontier_fn(hi)) && frontier_fn(hi) > d[2]
     ↪  && hi < 1e6) hi <- hi * 2
    u1_upper <- uniroot(function(u1) frontier_fn(u1) - d[2],
                        c(d[1], min(hi, 1e6)))$root
  } else {
    u1_upper <- u1_max
  }
  u1_seq <- seq(d[1], u1_upper, length.out = n_grid)
  u2_seq <- frontier_fn(u1_seq)
```

```
  valid <- u1_seq >= d[1] & u2_seq >= d[2]
  u1_seq <- u1_seq[valid]
  u2_seq <- u2_seq[valid]

  nash_product <- (u1_seq - d[1])^alpha * (u2_seq - d[2])^(1 -
 ↪  alpha)
  idx <- which.max(nash_product)

  list(u1 = u1_seq[idx], u2 = u2_seq[idx],
       nash_product = nash_product[idx],
       frontier = tibble(u1 = u1_seq, u2 = u2_seq, np =
        ↪  nash_product))
}

# Example: linear frontier u1 + u2 = 10, disagreement (1, 2)
frontier <- function(u1) 10 - u1
result <- nash_bargaining(frontier, d = c(1, 2))
cat(sprintf("Nash bargaining solution: (%.2f, %.2f)\n",
 ↪  result$u1, result$u2))
```

```
#> Nash bargaining solution: (4.50, 5.50)
```

```
cat(sprintf("Nash product: %.4f\n", result$nash_product))
```

```
#> Nash product: 3.5000
```

35.3.2 Nash bargaining visualization

```
# Concave frontier: u2 = sqrt(100 - u1^2) (quarter-circle, radius
 ↪  10)
frontier_concave <- function(u1) sqrt(pmax(100 - u1^2, 0))
d_point <- c(1, 2)
sol <- nash_bargaining(frontier_concave, d = d_point, u1_max =
 ↪  10)

# Nash product contour grid
grid_data <- expand_grid(u1 = seq(0, 10, length.out = 200),
                         u2 = seq(0, 10, length.out = 200)) |>
  mutate(np = ifelse(u1 >= d_point[1] & u2 >= d_point[2],
                     (u1 - d_point[1]) * (u2 - d_point[2]), NA))

p_nash <- ggplot() +
```

```
  geom_contour(data = grid_data, aes(x = u1, y = u2, z = np),
               colour = "grey70", linewidth = 0.3,
               breaks = seq(2, 30, by = 3)) +
  geom_line(data = sol$frontier, aes(x = u1, y = u2),
            colour = okabe_ito[1], linewidth = 1.2) +
  geom_point(aes(x = sol$u1, y = sol$u2),
             colour = okabe_ito[6], size = 4) +
  geom_point(aes(x = d_point[1], y = d_point[2]),
             colour = "black", size = 3, shape = 16) +
  geom_segment(aes(x = d_point[1], y = d_point[2],
                   xend = sol$u1, yend = sol$u2),
               linetype = "dashed", colour = "grey40") +
  annotate("text", x = sol$u1 + 0.4, y = sol$u2 + 0.4,
           label = sprintf("NBS (%.1f, %.1f)", sol$u1, sol$u2),
           size = 3.5, colour = okabe_ito[6], fontface = "bold")
            ↪  +
  annotate("text", x = d_point[1] + 0.6, y = d_point[2] - 0.5,
           label = sprintf("d = (%d, %d)", d_point[1],
            ↪  d_point[2]),
           size = 3.5) +
  scale_x_continuous(name = expression(u[1]), limits = c(0, 11))
   ↪  +
  scale_y_continuous(name = expression(u[2]), limits = c(0, 11))
   ↪  +
  theme_publication() +
  labs(title = "Nash Bargaining Solution")

p_nash
```

```
save_pub_fig(p_nash, "nash-bargaining-solution", width = 6,
 ↪  height = 5)
```

35.3.3 Rubinstein alternating-offers model

```
rubinstein_equilibrium <- function(delta1, delta2) {
  x1 <- (1 - delta2) / (1 - delta1 * delta2)
  x2 <- 1 - x1
  list(share1 = x1, share2 = x2, delta1 = delta1, delta2 =
   ↪  delta2)
}

# Example: symmetric discounting
```

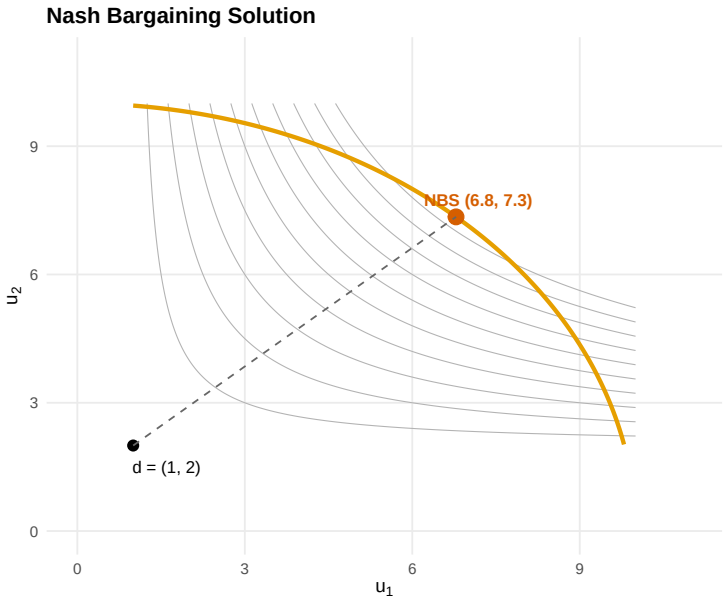


Figure 35.1: Nash bargaining solution for a concave feasible frontier. The orange curve marks the Pareto frontier. Grey contours show the Nash product. The red point is the Nash bargaining solution and the black point is the disagreement outcome.

```
eq_sym <- rubinstein_equilibrium(0.9, 0.9)
cat(sprintf("Symmetric (delta=0.9): Player 1 gets %.4f, Player 2
 ↪  gets %.4f\n",
            eq_sym$share1, eq_sym$share2))
```

```
#> Symmetric (delta=0.9): Player 1 gets 0.5263, Player 2 gets 0.4737
```

```
# Asymmetric: Player 1 more patient
eq_asym <- rubinstein_equilibrium(0.95, 0.8)
cat(sprintf("Asymmetric (d1=0.95, d2=0.8): Player 1 gets %.4f,
 ↪  Player 2 gets %.4f\n",
            eq_asym$share1, eq_asym$share2))
```

```
#> Asymmetric (d1=0.95, d2=0.8): Player 1 gets 0.8333, Player 2 gets 0.1667
```

35.3.4 Rubinstein equilibrium shares by discount factor

```
delta2_seq <- seq(0.01, 0.99, length.out = 200)

rubinstein_data <- map_dfr(c(0.7, 0.85, 0.95), function(d1) {
  tibble(
    delta2 = delta2_seq,
    share1 = (1 - delta2_seq) / (1 - d1 * delta2_seq),
    delta1_label = sprintf("delta[1] == %s", d1)
  )
})

p_rub <- ggplot(rubinstein_data, aes(x = delta2, y = share1,
                                     colour = delta1_label)) +

  geom_line(linewidth = 1) +
  geom_hline(yintercept = 0.5, linetype = "dashed", colour =
   ↪  "grey50") +
  scale_colour_manual(
    values = okabe_ito[c(1, 2, 3)],
    labels = parse(text = unique(rubinstein_data$delta1_label)),
    name = NULL
  ) +
  scale_x_continuous(name = expression(delta[2] ~ "(Player 2
   ↪  discount factor)"),
                     limits = c(0, 1)) +
  scale_y_continuous(name = "Player 1 equilibrium share",
```

```
                    limits = c(0, 1), labels = scales::percent)
                     ↪  +
  theme_publication() +
  labs(title = "Rubinstein Equilibrium Shares")

p_rub
```

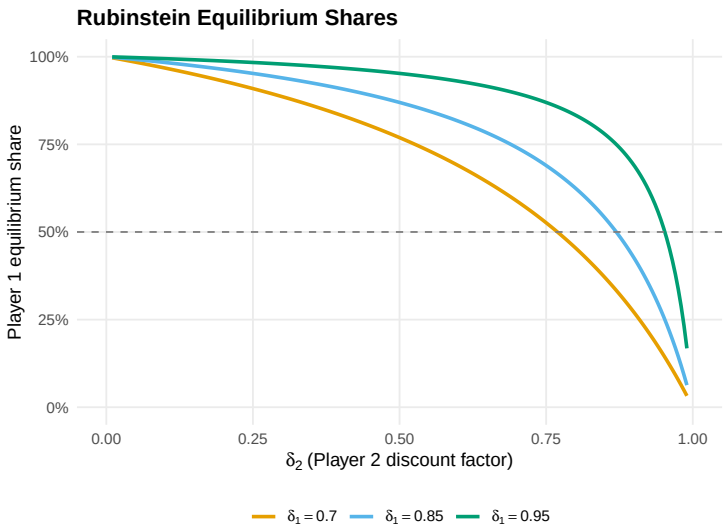


Figure 35.2: Rubinstein equilibrium share of Player 1 as a function of Player 2's discount factor, for three values of Player 1's discount factor. More patient players (higher delta) receive a larger share.

```
save_pub_fig(p_rub, "rubinstein-shares", width = 6, height = 4.5)
```

35.4 Worked example

35.4.1 Labor-management negotiation

A firm and its union bargain over how to split a surplus of \$10 million generated by a new contract. If negotiations fail, the firm earns \$2 million from replacement workers and the union's members earn \$1 million from outside options. The union's bargaining power is $\alpha = 0.6$ due to strong labor laws.

Step 1: Set up the bargaining problem.

```
surplus <- 10
d_firm <- 2   # disagreement payoff: firm
d_union <- 1  # disagreement payoff: union
alpha <- 0.6  # union bargaining power

# Linear frontier: u_firm + u_union = surplus
# Nash product: (u_firm - d_firm)^(1-alpha) * (u_union -
 ↪  d_union)^alpha
```

Step 2: Solve analytically. For a linear frontier $u_f + u_u = V$ the asymmetric NBS has a closed-form solution:

$$u_u^* = d_u + \alpha(V - d_f - d_u), \quad u_f^* = d_f + (1-\alpha)(V - d_f - d_u)$$

```
net_surplus <- surplus - d_firm - d_union
u_union <- d_union + alpha * net_surplus
u_firm <- d_firm + (1 - alpha) * net_surplus

cat(sprintf("Net surplus: $%.0fM\n", net_surplus))
```

```
#> Net surplus: $7M
```

```
cat(sprintf("Union payoff: $%.1fM (disagree) + $%.1fM =
 ↪  $%.1fM\n",
            d_union, alpha * net_surplus, u_union))
```

```
#> Union payoff: $1.0M (disagree) + $4.2M = $5.2M
```

```
cat(sprintf("Firm payoff:  $%.1fM (disagree) + $%.1fM =
 ↪  $%.1fM\n",
            d_firm, (1 - alpha) * net_surplus, u_firm))
```

```
#> Firm payoff:  $2.0M (disagree) + $2.8M = $4.8M
```

```
cat(sprintf("Union wage share: %.0f%%\n", 100 * u_union /
 ↪  surplus))
```

```
#> Union wage share: 52%
```

Step 3: Verify with numerical solver.

```
frontier_linear <- function(u_union) surplus - u_union
sol_labor <- nash_bargaining(frontier_linear, d = c(d_union,
 ↪  d_firm),
                             alpha = alpha)
cat(sprintf("Numerical NBS: union = $%.2fM, firm = $%.2fM\n",
            sol_labor$u1, sol_labor$u2))
```

```
#> Numerical NBS: union = $5.20M, firm = $4.80M
```

Step 4: Rubinstein interpretation. If bargaining rounds occur weekly with annual discount rates $r_f = 0.05$ and $r_u = 0.10$, the weekly discount factors are:

```
delta_firm <- exp(-0.05 / 52)
delta_union <- exp(-0.10 / 52)

eq_labor <- rubinstein_equilibrium(delta_union, delta_firm)
cat(sprintf("Weekly discount factors: union = %.6f, firm =
 ↪  %.6f\n",
            delta_union, delta_firm))
```

```
#> Weekly discount factors: union = 0.998079, firm = 0.999039
```

```
cat(sprintf("Rubinstein share: union = %.4f, firm = %.4f\n",
            eq_labor$share1, eq_labor$share2))
```

```
#> Rubinstein share: union = 0.3337, firm = 0.6663
```

```
cat(sprintf("Union gets $%.2fM of the $%dM surplus\n",
            eq_labor$share1 * surplus, surplus))
```

```
#> Union gets $3.34M of the $10M surplus
```

The more patient firm (lower discount rate) captures a larger share, consistent with the Rubinstein insight that patience is power in bargaining.

35.5 Extensions

- **Outside options** can shift the disagreement point and change the Nash bargaining solution. See Osborne (2004) (Chapter 15) for models where players can exit bargaining.

- **Incomplete information** about the opponent's discount factor or reservation value leads to delay and inefficiency; see Kennan & Wilson (1993) for a survey.
- **Multi-party bargaining** extends these ideas to legislatures and coalitions, connecting to the **Shapley value** (9).
- **Behavioral bargaining** — ultimatum game experiments consistently show that proposers offer 40–50% rather than the minimal amount predicted by backward induction, suggesting fairness norms matter.

Exercises

1. **Asymmetric Nash bargaining.** Two countries negotiate over fishing rights with a frontier given by $u_2 = 12 - 1.5u_1$ and disagreement point $(1, 1)$. Country 1 has bargaining power $\alpha = 0.4$. Compute the Nash bargaining solution analytically and verify it with the `nash_bargaining()` function. How does the solution change if $\alpha = 0.7$?

2. **Patience and power.** In the Rubinstein model, suppose Player 1 has $\delta_1 = 0.9$. Find the value of δ_2 at which the surplus is split equally. Plot Player 1's share as δ_2 varies from 0.5 to 0.99 and verify your answer graphically.

3. **Ultimatum game.** The ultimatum game is a Rubinstein model with one round: Player 1 proposes a split and Player 2 accepts or rejects. Show analytically that the SPE has Player 1 offering the minimal acceptable amount. Then simulate 1,000 ultimatum games where Player 2 rejects offers below a threshold drawn uniformly from $[0, 0.5]$. Plot the proposer's expected payoff as a function of the offer.

Solutions appear in D.

Chapter 36

Conflict and Cooperation Models

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

Applying game theory to international relations and social dilemmas — Chicken, Stag Hunt, Schelling focal points, and arms race dynamics.

Learning objectives

- Formulate the Chicken game and Stag Hunt as strategic-form games, and identify their Nash equilibria.
- Explain Schelling focal points and why coordination can succeed without communication.
- Model an arms race as an iterated Prisoner's Dilemma and analyze how strategy choice affects long-run payoffs.
- Simulate cooperation and defection dynamics under varying risk-dominance conditions in R.

36.1 Motivation

In October 1962, the United States and the Soviet Union stood at the brink of nuclear war during the Cuban Missile Crisis. Each side could escalate (hold firm) or back down (accommodate). Mutual escalation risked catastrophe; mutual accommodation preserved the status quo. But each side preferred the other to back down first.

This scenario has the structure of the **Chicken game**, one of several canonical models that game theory uses to study conflict and cooperation. Alongside **Stag Hunt** (coordination under risk) and the **iterated Prisoner's Dilemma** (arms races), these models illuminate when cooperation emerges, when it collapses, and what institutional features sustain it.

36.2 Theory

36.2.1 The Chicken game

In Chicken, two players simultaneously choose to Swerve (cooperate) or Drive Straight (defect). The payoff structure is:

Table 36.1: Chicken game payoff matrix

	Swerve	Straight
Swerve	$(3, 3)$	$(2, 4)$
Straight	$(4, 2)$	$(0, 0)$

Mutual defection (Straight, Straight) is the worst outcome for both — a "crash." The game has **two pure-strategy Nash equilibria**: (Swerve, Straight) and (Straight, Swerve), plus a mixed equilibrium. Unlike the Prisoner's Dilemma, mutual defection is *not* an equilibrium.

36.2.2 Stag Hunt

In Stag Hunt, two hunters choose to hunt Stag (cooperate) or hunt Hare (defect). Hunting stag requires coordination; a lone stag hunter gets nothing.

Table 36.2: Stag Hunt payoff matrix

	Stag	Hare
Stag	$(4, 4)$	$(0, 3)$
Hare	$(3, 0)$	$(3, 3)$

The game has two pure-strategy NE: (Stag, Stag) is **payoff-dominant** (both prefer it) while (Hare, Hare) is **risk-dominant** (safer against uncertainty). The mixed NE has each player hunting Stag with probability $p^* = 3/4$. The tension between payoff dominance and risk dominance is central to coordination problems.

Definition: Risk Dominance

In a 2×2 game with two pure-strategy NE, equilibrium (a_1, a_2) **risk-dominates** (b_1, b_2) if the product of deviation losses is larger:

$$(u_1(a_1,a_2)-u_1(b_1,a_2))(u_2(a_1,a_2)-u_2(a_1,b_2)) > (u_1(b_1,b_2)-u_1(a_1,b_2))(u_2(b_1,b_2)-u_2(b_1,a_2))$$

36.2.3 Schelling focal points

Schelling (1960) observed that players often coordinate successfully by choosing options that are culturally or contextually prominent — **focal points**. When asked to meet in New York without specifying a location, most people choose Grand Central Station at noon. Focal points operate outside the formal game structure but resolve the equilibrium selection problem in practice.

36.2.4 Arms race as iterated Prisoner's Dilemma

An arms race between two nations can be modeled as a repeated Prisoner's Dilemma. In each period, each nation chooses to Arm (defect) or Disarm (cooperate). The stage-game payoffs reflect that mutual disarmament is efficient, but each side has a unilateral incentive to arm. Over many rounds, strategies like Tit-for-Tat can sustain cooperation, but sufficiently hawkish strategies or finite horizons can trigger arms races.

36.3 Implementation in R

36.3.1 Stag Hunt dynamics under varying risk dominance

```
simulate_stag_hunt <- function(a, b, c_val, d_val, n_pop = 100,
                               n_periods = 50, p0_seq = seq(0.05,
                                ↪  0.95, by = 0.05)) {
  # Payoffs: Stag/Stag=a, Stag/Hare=b, Hare/Stag=c, Hare/Hare=d
  # Replicator dynamics: dp/dt = p(1-p)[EU(Stag) - EU(Hare)]
  # EU(Stag) = p*a + (1-p)*b,  EU(Hare) = p*c + (1-p)*d

  results <- map_dfr(p0_seq, function(p0) {
    p <- p0
    trajectory <- tibble(t = 0, p_stag = p, p0 = p0)
    for (t_step in seq_len(n_periods)) {
      eu_stag <- p * a + (1 - p) * b
```

```
      eu_hare <- p * c_val + (1 - p) * d_val
      dp <- p * (1 - p) * (eu_stag - eu_hare)
      p <- max(0, min(1, p + dp * 0.1))  # Euler step
      trajectory <- bind_rows(trajectory,
                              tibble(t = t_step, p_stag = p, p0
                               ↪  = p0))
    }
    trajectory
  })
  results
}

# Baseline Stag Hunt: a=4, b=0, c=3, d=3
stag_dynamics <- simulate_stag_hunt(a = 4, b = 0, c_val = 3,
 ↪  d_val = 3)

# Mixed NE threshold: p* = (d - b) / (a - c - b + d) = 3/4
p_star <- 3 / 4
cat(sprintf("Stag Hunt mixed NE threshold: p* = %.2f\n", p_star))
```

```
#> Stag Hunt mixed NE threshold: p* = 0.75
```

```
cat("Populations starting above p* converge to (Stag, Stag).\n")
```

```
#> Populations starting above p* converge to (Stag, Stag).
```

```
cat("Populations starting below p* converge to (Hare, Hare).\n")
```

```
#> Populations starting below p* converge to (Hare, Hare).
```

36.3.2 Phase diagram: cooperation vs. defection basins

```
# Three regimes: vary d (safe payoff) to shift risk dominance
regimes <- tibble(
  d_val = c(2, 3, 3.5),
  label = c("Weak risk (d=2)", "Baseline (d=3)", "Strong risk
   ↪  (d=3.5)")
)

phase_data <- map_dfr(seq_len(nrow(regimes)), function(i) {
  dyn <- simulate_stag_hunt(a = 4, b = 0, c_val =
 ↪  regimes$d_val[i],
```

```
                            d_val = regimes$d_val[i],
                            p0_seq = seq(0.1, 0.9, by = 0.1))
  dyn$regime <- regimes$label[i]
  dyn$p_star <- (regimes$d_val[i] - 0) / (4 - regimes$d_val[i] -
 ↪  0 + regimes$d_val[i])
  dyn
})

phase_data$regime <- factor(phase_data$regime, levels =
 ↪  regimes$label)

p_phase <- ggplot(phase_data, aes(x = t, y = p_stag, group = p0))
 ↪  +
  geom_line(colour = okabe_ito[1], alpha = 0.5, linewidth = 0.5)
   ↪  +
  geom_hline(aes(yintercept = p_star), linetype = "dashed",
             colour = okabe_ito[5], linewidth = 0.7) +
  facet_wrap(~ regime, ncol = 3) +
  scale_x_continuous(name = "Time period") +
  scale_y_continuous(name = "Proportion hunting Stag",
                     limits = c(0, 1), labels = scales::percent)
                      ↪  +
  theme_publication() +
  labs(title = "Stag Hunt: Basins of Attraction under Varying
   ↪  Risk Dominance")

p_phase
```

```
save_pub_fig(p_phase, "stag-hunt-basins", width = 7, height = 5)
```

36.3.3 Arms race simulation

```
simulate_arms_race <- function(strategy1, strategy2, n_rounds =
 ↪  50,
                               payoffs = list(CC = c(3, 3), CD =
                                ↪  c(0, 5),
                                              DC = c(5, 0), DD =
                                               ↪  c(1, 1))) {
  # strategy: function(own_history, opp_history) -> "C" or "D"
  h1 <- character(0)
  h2 <- character(0)
  results <- tibble(round = integer(), a1 = character(), a2 =
 ↪  character(),
```

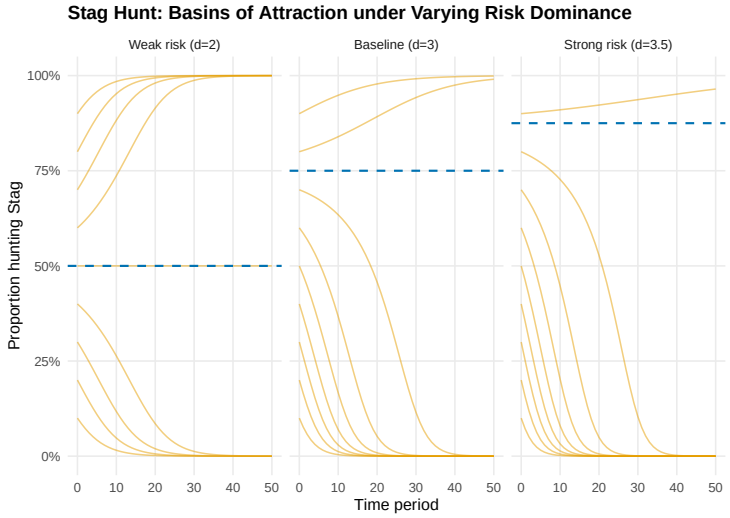


Figure 36.1: Phase diagram of Stag Hunt replicator dynamics under three risk-dominance regimes. Each curve is one trajectory starting from a different initial proportion of Stag hunters. The dashed line marks the mixed-NE threshold separating the basins of attraction.

```
                  pay1 = numeric(), pay2 = numeric())

  for (r in seq_len(n_rounds)) {
    a1 <- strategy1(h1, h2)
    a2 <- strategy2(h2, h1)
    key <- paste0(a1, a2)
    pay <- payoffs[[key]]
    results <- bind_rows(results,
                         tibble(round = r, a1 = a1, a2 = a2,
                                pay1 = pay[1], pay2 = pay[2]))
    h1 <- c(h1, a1)
    h2 <- c(h2, a2)
  }
  results |> mutate(cum_pay1 = cumsum(pay1), cum_pay2 =
  ↪  cumsum(pay2))
}

# Define strategies
always_defect <- function(own, opp) "D"
always_cooperate <- function(own, opp) "C"
tit_for_tat <- function(own, opp) if (length(opp) == 0) "C" else
  ↪  tail(opp, 1)
```

```
grim_trigger <- function(own, opp) if (any(opp == "D")) "D" else
 ↪  "C"
```

```
matchups <- list(
  list(s1 = tit_for_tat, s2 = tit_for_tat, name = "TfT vs TfT"),
  list(s1 = always_defect, s2 = always_defect, name = "AllD vs
   ↪  AllD"),
  list(s1 = tit_for_tat, s2 = always_defect, name = "TfT vs
   ↪  AllD"),
  list(s1 = grim_trigger, s2 = tit_for_tat, name = "Grim vs TfT")
)

arms_data <- map_dfr(matchups, function(m) {
  sim <- simulate_arms_race(m$s1, m$s2, n_rounds = 50)
  bind_rows(
    sim |> select(round, cumulative = cum_pay1) |>
      mutate(player = "Nation 1", matchup = m$name),
    sim |> select(round, cumulative = cum_pay2) |>
      mutate(player = "Nation 2", matchup = m$name)
  )
})

p_arms <- ggplot(arms_data, aes(x = round, y = cumulative,
                                colour = player, linetype =
                                 ↪  player)) +
  geom_line(linewidth = 0.9) +
  facet_wrap(~ matchup, ncol = 2, scales = "free_y") +
  scale_colour_manual(values = okabe_ito[c(1, 2)], name = NULL) +
  scale_linetype_manual(values = c("solid", "dashed"), name =
   ↪  NULL) +
  scale_x_continuous(name = "Round") +
  scale_y_continuous(name = "Cumulative payoff") +
  theme_publication() +
  labs(title = "Arms Race Dynamics: Iterated Prisoner's Dilemma")

p_arms
```

```
save_pub_fig(p_arms, "arms-race-dynamics", width = 7, height = 5)
```

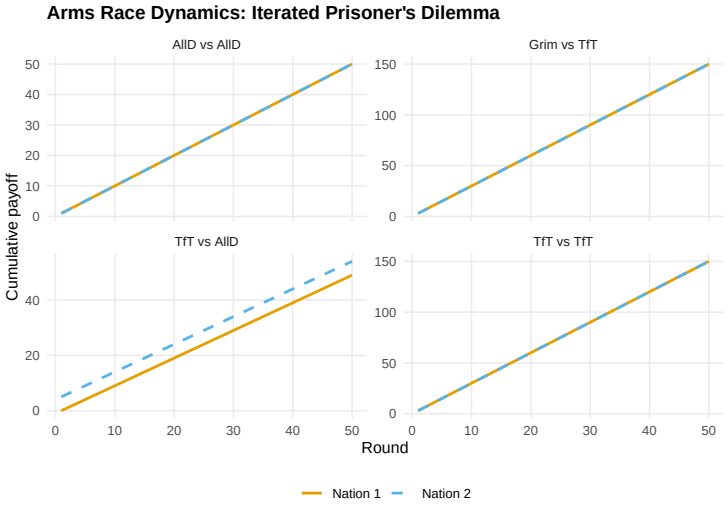


Figure 36.2: Cumulative payoffs over 50 rounds of an arms race (iterated PD) under four strategy matchups. Tit-for-Tat sustains cooperation, while Always Defect locks both sides into low payoffs.

36.4 Worked example

36.4.1 Cuban Missile Crisis as a Chicken game

In October 1962, the US and USSR faced a standoff over Soviet missiles in Cuba. We model this as a Chicken game with the following interpretation:

- **Swerve** = back down (US removes Turkish missiles / USSR removes Cuban missiles)
- **Straight** = hold firm (maintain military posture)

```
# Payoff matrix (US perspective first, USSR second)
A_chicken <- matrix(c(3, 2, 4, 0), nrow = 2, byrow = TRUE,
                    dimnames = list(c("Back Down", "Hold Firm"),
                                    c("Back Down", "Hold Firm")))
B_chicken <- matrix(c(3, 4, 2, 0), nrow = 2, byrow = TRUE,
                    dimnames = list(c("Back Down", "Hold Firm"),
                                    c("Back Down", "Hold Firm")))

cat("US payoffs:\n")
```

```
#> US payoffs:
```

```
A_chicken
```

```
#>           Back Down Hold Firm
#> Back Down         3         2
#> Hold Firm         4         0
```

```
cat("\nUSSR payoffs:\n")
```

```
#>
#> USSR payoffs:
```

```
B_chicken
```

```
#>           Back Down Hold Firm
#> Back Down         3         4
#> Hold Firm         2         0
```

Step 1: Find all Nash equilibria.

```
source(here::here("R", "solvers.R"))

pure_ne <- solve_2x2_pure_nash(A_chicken, B_chicken)
mixed_ne <- solve_2x2_mixed_nash(A_chicken, B_chicken)

cat("Pure-strategy NE:\n")
```

```
#> Pure-strategy NE:
```

```
for (eq in pure_ne) {
  cat(sprintf("  (US: %s, USSR: %s) -> payoffs (%d, %d)\n",
              rownames(A_chicken)[eq[1]],
               ↪  colnames(A_chicken)[eq[2]],
              A_chicken[eq[1], eq[2]], B_chicken[eq[1], eq[2]]))
}
```

```
#>   (US: Back Down, USSR: Hold Firm) -> payoffs (2, 4)
#>   (US: Hold Firm, USSR: Back Down) -> payoffs (4, 2)
```

```
cat(sprintf("\nMixed NE: US backs down with p = %.2f, USSR backs
 ↪  down with q = %.2f\n",
            mixed_ne$p, mixed_ne$q))
```

Step 2: Analyze expected payoffs and crisis risk.

```
p <- mixed_ne$p
q <- mixed_ne$q

# Probability of nuclear war (both hold firm)
p_war <- (1 - p) * (1 - q)
cat(sprintf("Probability of mutual escalation (war): %.1f%%\n",
 ↪  100 * p_war))

# Expected payoffs at mixed NE
eu_us <- p * q * 3 + p * (1 - q) * 2 + (1 - p) * q * 4 + (1 - p)
 ↪  * (1 - q) * 0
eu_ussr <- p * q * 3 + p * (1 - q) * 4 + (1 - p) * q * 2 + (1 -
 ↪  p) * (1 - q) * 0
cat(sprintf("Expected payoff at mixed NE: US = %.2f, USSR =
 ↪  %.2f\n", eu_us, eu_ussr))
cat(sprintf("Both sides would prefer mutual accommodation (3,
 ↪  3).\n"))
```

```
#> Both sides would prefer mutual accommodation (3, 3).
```

Step 3: Commitment and credibility. In practice, the crisis was resolved because the US committed to a naval blockade (a costly signal of resolve) while offering a face-saving exit: a secret agreement to remove Jupiter missiles from Turkey. This transformed the game by changing the perceived payoffs — making "Hold Firm" more credible for the US while reducing the USSR's cost of backing down.

36.5 Extensions

- **Evolutionary dynamics** on Chicken and Stag Hunt are explored in 20, where mixed equilibria correspond to polymorphic population states.
- **Repeated interaction** transforms one-shot Chicken into a folk-theorem setting where cooperation can be sustained (8).
- **Incomplete information** versions of Chicken model situations where each player is uncertain about the other's resolve — see Osborne (2004) (Chapter 9).
- **Network effects** on coordination games are covered in 22, where Stag Hunt dynamics depend on the structure of social connections.

Exercises

1. **Risk dominance in Stag Hunt.** Modify the Stag Hunt payoffs so that the mixed-NE threshold is $p^* = 0.5$ instead of 0.75. What values of (a, b, c, d) achieve this? Simulate the replicator dynamics and verify that the basins of attraction are now equal in size.

2. **Chicken with asymmetric payoffs.** Suppose the US values "holding firm" at 5 instead of 4 (more hawkish). Recompute the mixed-strategy NE. How does the probability of mutual escalation change? Interpret the result in terms of the paradox that greater resolve can increase the risk of catastrophe.

3. **Arms race tournament.** Implement a round-robin tournament among four strategies: Always Cooperate, Always Defect, Tit-for-Tat, and Grim Trigger. Each pair plays 100 rounds of the iterated PD. Compute total payoffs for each strategy and rank them. Which strategy wins? Does this match Axelrod's tournament results?

Solutions appear in D.

Chapter 37

Empirical Case Studies

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

Real-world applications of game theory analyzed with R — spectrum auctions, kidney exchange, and Braess's paradox in traffic routing.

Learning objectives

- Model an FCC-style spectrum auction as a multi-unit sealed-bid simulation and analyze revenue outcomes.
- Formulate kidney exchange as a matching problem on a compatibility graph and compute maximal matchings.
- Explain Braess's paradox and compute Wardrop equilibria for simple traffic networks in R.
- Visualize network structures and compare equilibrium outcomes with socially optimal flows.

37.1 Motivation

Game theory is not merely an abstract discipline — it has reshaped multi-billion-dollar markets and saved thousands of lives. The FCC's spectrum auctions raised over $200 billion by applying auction theory. Kidney exchange programs, designed using matching theory, enable thousands of transplants annually that would otherwise be impossible. And traffic engineers have discovered, counter-intuitively, that adding roads can *increase* total travel time — a phenomenon predicted by game-theoretic models.

This chapter applies computational game theory to three landmark case studies, demonstrating how the tools developed throughout this book operate in practice.

37.2 Theory

37.2.1 Spectrum auctions

The FCC allocates wireless spectrum licenses through **simultaneous ascending auctions**. Multiple licenses are sold concurrently; bidders submit bids in rounds, prices rise, and bidders can switch across licenses. The design aims to achieve an **efficient allocation** — licenses go to firms that value them most — while generating revenue for the government.

We simplify this to a **multi-unit sealed-bid auction** where n bidders submit bids for k identical licenses, and the k highest bidders win at the $(k+1)$-th highest bid (a Vickrey-style rule that incentivizes truthful bidding).

37.2.2 Kidney exchange

A patient needing a kidney often has a willing but biologically incompatible donor. **Kidney exchange** matches incompatible patient-donor pairs: if Patient A's donor is compatible with Patient B, and Patient B's donor is compatible with Patient A, a **pairwise exchange** can save both lives.

The problem maps to a **directed graph** where nodes are patient-donor pairs and edges indicate compatibility. A matching is a set of disjoint cycles (typically of length 2 or 3) that maximizes the number of transplants. This is equivalent to a **maximum weight matching** problem.

37.2.3 Braess's paradox

Definition: Braess's Paradox

Braess's paradox occurs when adding a new road to a traffic network *increases* the total travel time for all users at the Wardrop equilibrium.

In a **Wardrop equilibrium** (the traffic analog of Nash equilibrium), every used route has equal travel time, and no driver can reduce their travel time by unilaterally switching routes. The paradox arises because selfish routing ignores congestion externalities — each driver's route choice increases travel times for others.

Consider a 4-node network from origin O to destination D with two intermediate nodes A and B:

- Route 1: $O \to A \to D$ with costs $c_{OA}(x) = x/100$ and $c_{AD} = 45$.
- Route 2: $O \to B \to D$ with costs $c_{OB} = 45$ and $c_{BD}(x) = x/100$.

With 4,000 drivers, the Wardrop equilibrium splits traffic equally: 2,000 on each route, total time 65 per driver. Now add a zero-cost road $A \to B$. A third route $O \to A \to B \to D$ becomes available, and the new equilibrium has all 4,000 drivers using this route — total time 80. The added road made everyone worse off.

37.3 Implementation in R

37.3.1 Spectrum auction simulation

```
simulate_spectrum_auction <- function(n_bidders, n_licenses,
 ↪  n_sims = 1000) {
  # Each bidder's value drawn from Uniform(0, 100)
  # Vickrey rule: k highest bidders win, pay (k+1)-th bid
  results <- map_dfr(seq_len(n_sims), function(sim) {
    values <- sort(runif(n_bidders, 0, 100), decreasing = TRUE)
    winners <- values[seq_len(n_licenses)]
    price <- values[n_licenses + 1]  # (k+1)-th highest bid
    tibble(sim = sim, revenue = n_licenses * price,
           avg_winner_surplus = mean(winners - price),
           price = price)
  })
  results
}

auction_results <- simulate_spectrum_auction(n_bidders = 10,
 ↪  n_licenses = 3)
cat(sprintf("Mean revenue (10 bidders, 3 licenses): $%.1f\n",
            mean(auction_results$revenue)))
```

```
#> Mean revenue (10 bidders, 3 licenses): $191.8
```

```
cat(sprintf("Mean winner surplus: $%.1f\n",
            mean(auction_results$avg_winner_surplus)))
```

```
#> Mean winner surplus: $18.0
```

```
cat(sprintf("Mean clearing price: $%.1f\n",
            mean(auction_results$price)))
```

```
#> Mean clearing price: $63.9
```

37.3.2 Kidney exchange matching

```
build_compatibility_graph <- function(n_pairs, compatibility_prob
 ↪  = 0.15,
                                      seed = 42) {
  set.seed(seed)
  # Each pair has a patient blood type and donor blood type
  blood_types <- c("O", "A", "B", "AB")
  # Simplified compatibility: O donates to all, AB receives from
   ↪  all
  can_donate <- function(donor_bt, patient_bt) {
    if (donor_bt == "O") return(TRUE)
    if (donor_bt == patient_bt) return(TRUE)
    if (patient_bt == "AB") return(TRUE)
    FALSE
  }

  pairs <- tibble(
    pair_id = seq_len(n_pairs),
    patient_bt = sample(blood_types, n_pairs, replace = TRUE,
                        prob = c(0.44, 0.42, 0.10, 0.04)),
    donor_bt = sample(blood_types, n_pairs, replace = TRUE,
                      prob = c(0.44, 0.42, 0.10, 0.04))
  ) |>
    # Exclude already-compatible pairs (they don't need exchange)
    filter(!map2_lgl(donor_bt, patient_bt, can_donate))

  n <- nrow(pairs)
  # Build adjacency: edge from i to j if i's donor is compatible
   ↪  with j's patient
  edges <- expand_grid(from = seq_len(n), to = seq_len(n)) |>
    filter(from != to) |>
    mutate(
      compatible = map2_lgl(from, to, function(i, j) {
        can_donate(pairs$donor_bt[i], pairs$patient_bt[j]) &
          runif(1) < (1 + compatibility_prob)  # crossmatch
           ↪  factor
      })
```

```
    ) |>
    filter(compatible)

  list(pairs = pairs, edges = edges, n = n)
}

find_pairwise_matches <- function(graph) {
  # Greedy maximal matching on pairwise exchanges (2-cycles)
  edges <- graph$edges
  n <- graph$n
  matched <- rep(FALSE, n)
  matches <- list()

  # Find all 2-cycles: i->j and j->i both exist
  for (idx in seq_len(nrow(edges))) {
    i <- edges$from[idx]
    j <- edges$to[idx]
    if (matched[i] || matched[j]) next
    # Check reverse edge exists
    reverse <- edges |> filter(from == j, to == i)
    if (nrow(reverse) > 0) {
      matched[i] <- TRUE
      matched[j] <- TRUE
      matches <- c(matches, list(c(i, j)))
    }
  }
  list(matches = matches, n_matched = sum(matched), n_total = n)
}

graph <- build_compatibility_graph(n_pairs = 50)
matching <- find_pairwise_matches(graph)
cat(sprintf("Pairs in pool: %d\n", matching$n_total))
```

```
#> Pairs in pool: 9
```

```
cat(sprintf("Pairwise exchanges found: %d (saving %d
 ↪  patients)\n",
            length(matching$matches), matching$n_matched))
```

```
#> Pairwise exchanges found: 0 (saving 0 patients)
```

37.3.3 Kidney exchange network visualization

```
# Layout nodes in a circle
n_nodes <- graph$n
angles <- seq(0, 2 * pi, length.out = n_nodes + 1)[-(n_nodes +
 ↪  1)]
node_data <- graph$pairs |>
  mutate(x = cos(angles), y = sin(angles),
         node_id = row_number())

# Edge data
edge_data <- graph$edges |>
  left_join(node_data |> select(node_id, x, y), by = c("from" =
   ↪  "node_id")) |>
  rename(x_from = x, y_from = y) |>
  left_join(node_data |> select(node_id, x, y), by = c("to" =
   ↪  "node_id")) |>
  rename(x_to = x, y_to = y)

# Matched edges
matched_pairs <- do.call(rbind, matching$matches)
if (!is.null(matched_pairs)) {
  matched_edge_data <- map_dfr(seq_len(nrow(matched_pairs)),
 ↪  function(r) {
    i <- matched_pairs[r, 1]
    j <- matched_pairs[r, 2]
    bind_rows(
      tibble(x_from = node_data$x[i], y_from = node_data$y[i],
             x_to = node_data$x[j], y_to = node_data$y[j]),
      tibble(x_from = node_data$x[j], y_from = node_data$y[j],
             x_to = node_data$x[i], y_to = node_data$y[i])
    )
  })
} else {
  matched_edge_data <- tibble(x_from = numeric(), y_from =
 ↪  numeric(),
                              x_to = numeric(), y_to = numeric())
}

# Colour by blood type
bt_colours <- c("O" = okabe_ito[1], "A" = okabe_ito[2],
                "B" = okabe_ito[3], "AB" = okabe_ito[5])

p_kidney <- ggplot() +
  geom_segment(data = edge_data, aes(x = x_from, y = y_from,
```

```
                                        xend = x_to, yend = y_to),
               colour = "grey80", linewidth = 0.3, alpha = 0.5) +
  geom_segment(data = matched_edge_data,
               aes(x = x_from, y = y_from, xend = x_to, yend =
                ↪  y_to),
               colour = okabe_ito[6], linewidth = 1.2) +
  geom_point(data = node_data, aes(x = x, y = y, fill =
   ↪  patient_bt),
             shape = 21, size = 4, colour = "black", stroke =
              ↪  0.5) +
  scale_fill_manual(values = bt_colours, name = "Patient\nblood
   ↪  type") +
  coord_equal() +
  theme_publication() +
  theme(axis.text = element_blank(), axis.title =
   ↪  element_blank(),
        axis.ticks = element_blank(), panel.grid =
         ↪  element_blank()) +
  labs(title = "Kidney Exchange Compatibility Network")

p_kidney
```

```
save_pub_fig(p_kidney, "kidney-exchange-network", width = 7,
 ↪  height = 6)
```

37.3.4 Braess's paradox

```
braess_network <- function(n_drivers = 4000, has_shortcut =
 ↪  FALSE) {
  # Routes in 4-node network: O -> A -> D, O -> B -> D, and
   ↪  optionally O -> A -> B -> D
  # Cost functions: c_OA(x) = x/100, c_AD = 45, c_OB = 45,
   ↪  c_BD(x) = x/100
  # Shortcut A -> B costs 0

  if (!has_shortcut) {
    # Without shortcut: two routes, symmetric => equilibrium
     ↪  splits equally
    x <- n_drivers / 2
    time_r1 <- x / 100 + 45       # O->A->D
    time_r2 <- 45 + x / 100       # O->B->D
    return(list(routes = c("O-A-D", "O-B-D"),
```

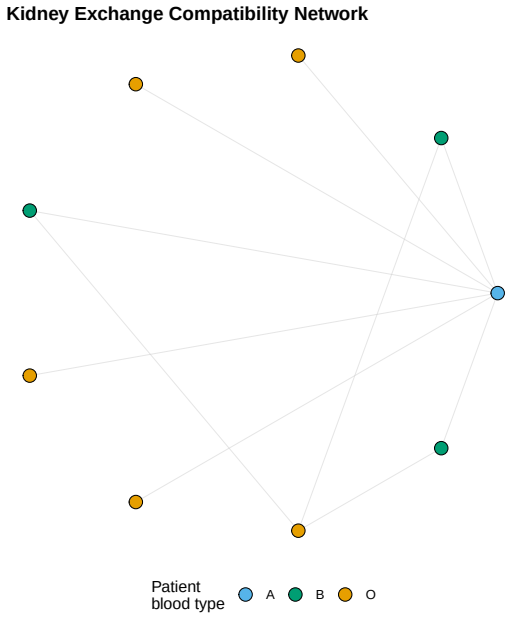


Figure 37.1: Kidney exchange compatibility network. Nodes represent incompatible patient-donor pairs (coloured by patient blood type). Edges show potential donations. Highlighted thick edges are matched pairwise exchanges.

```
                flows = c(x, x),
                times = c(time_r1, time_r2),
                total_time = n_drivers * time_r1))
  }

  # With shortcut: Wardrop equilibrium

  # Route 3: O->A->B->D has cost x_OA/100 + 0 + x_BD/100
  # At equilibrium, all used routes have equal cost
  # Solving: all drivers use O->A->B->D, each segment carries
   ↪ 4000
  # Time = 4000/100 + 0 + 4000/100 = 80
  x_all <- n_drivers
  time_shortcut <- x_all / 100 + 0 + x_all / 100

  return(list(routes = c("O-A-D", "O-B-D", "O-A-B-D"),
              flows = c(0, 0, x_all),
              times = c(NA, NA, time_shortcut),
              total_time = n_drivers * time_shortcut))
}

# Social optimum with shortcut: solve by splitting traffic
social_optimum_shortcut <- function(n_drivers = 4000) {
  # Optimize over flow split: x on O-A-D, y on O-B-D, z on
   ↪ O-A-B-D
  # x + y + z = n_drivers
  # Minimize total travel time
  best <- list(total = Inf)
  for (x in seq(0, n_drivers, by = 100)) {
    for (y in seq(0, n_drivers - x, by = 100)) {
      z <- n_drivers - x - y
      flow_OA <- x + z
      flow_BD <- y + z
      time_r1 <- flow_OA / 100 + 45
      time_r2 <- 45 + flow_BD / 100
      time_r3 <- flow_OA / 100 + 0 + flow_BD / 100
      total <- x * time_r1 + y * time_r2 + z * time_r3
      if (total < best$total) {
        best <- list(x = x, y = y, z = z, total = total,
                     times = c(time_r1, time_r2, time_r3))
      }
    }
  }
  best
}
```

```
no_shortcut <- braess_network(4000, has_shortcut = FALSE)
with_shortcut <- braess_network(4000, has_shortcut = TRUE)
social_opt <- social_optimum_shortcut(4000)

cat("=== Without shortcut ===\n")
```

```
#> === Without shortcut ===
```

```
cat(sprintf("  Each route: %d drivers, travel time: %.0f\n",
            no_shortcut$flows[1], no_shortcut$times[1]))
```

```
#>   Each route: 2000 drivers, travel time: 65
```

```
cat(sprintf("  Total travel time: %s\n",
 ↪  format(no_shortcut$total_time, big.mark = ",")))
```

```
#>   Total travel time: 260,000
```

```
cat("\n=== With shortcut (Wardrop equilibrium) ===\n")
```

```
#>
#> === With shortcut (Wardrop equilibrium) ===
```

```
cat(sprintf("  All %d drivers use O-A-B-D, travel time: %.0f\n",
            with_shortcut$flows[3], with_shortcut$times[3]))
```

```
#>   All 4000 drivers use O-A-B-D, travel time: 80
```

```
cat(sprintf("  Total travel time: %s\n",
 ↪  format(with_shortcut$total_time, big.mark = ",")))
```

```
#>   Total travel time: 320,000
```

```
cat("\n=== With shortcut (social optimum) ===\n")
```

```
#>
#> === With shortcut (social optimum) ===
```

```
cat(sprintf("  O-A-D: %d, O-B-D: %d, O-A-B-D: %d\n",
            social_opt$x, social_opt$y, social_opt$z))
```

```
#>   O-A-D: 1700, O-B-D: 1700, O-A-B-D: 600
```

```
cat(sprintf("  Total travel time: %s\n", format(social_opt$total,
 ↪  big.mark = ",")))
```

```
#>   Total travel time: 258,800
```

```
cat(sprintf("\nBraess's paradox: adding the shortcut increased
 ↪  travel time by %.0f%%\n",
            100 * (with_shortcut$total_time -
             ↪  no_shortcut$total_time) /
              no_shortcut$total_time))
```

```
#>
#> Braess's paradox: adding the shortcut increased travel time by 23%
```

37.3.5 Braess's paradox visualization

```
braess_data <- tibble(
  scenario = factor(c("No shortcut\n(Wardrop eq.)",
                      "With shortcut\n(Wardrop eq.)",
                      "With shortcut\n(Social optimum)"),
                    levels = c("No shortcut\n(Wardrop eq.)",
                               "With shortcut\n(Wardrop eq.)",
                               "With shortcut\n(Social
                                ↪  optimum)")),
  total_time = c(no_shortcut$total_time,
                 with_shortcut$total_time,
                 social_opt$total),
  per_driver = total_time / 4000
)

p_braess <- ggplot(braess_data, aes(x = scenario, y = per_driver,
                                    fill = scenario)) +
  geom_col(width = 0.6, show.legend = FALSE) +
  geom_text(aes(label = sprintf("%.0f min", per_driver)),
            vjust = -0.5, size = 4, fontface = "bold") +
  scale_fill_manual(values = okabe_ito[c(2, 6, 3)]) +
```

```
  scale_y_continuous(name = "Travel time per driver (minutes)",
                     limits = c(0, 95), expand = c(0, 0)) +
  scale_x_discrete(name = NULL) +
  theme_publication() +
  labs(title = "Braess's Paradox: Adding a Road Increases
   ↪  Congestion")

p_braess
```

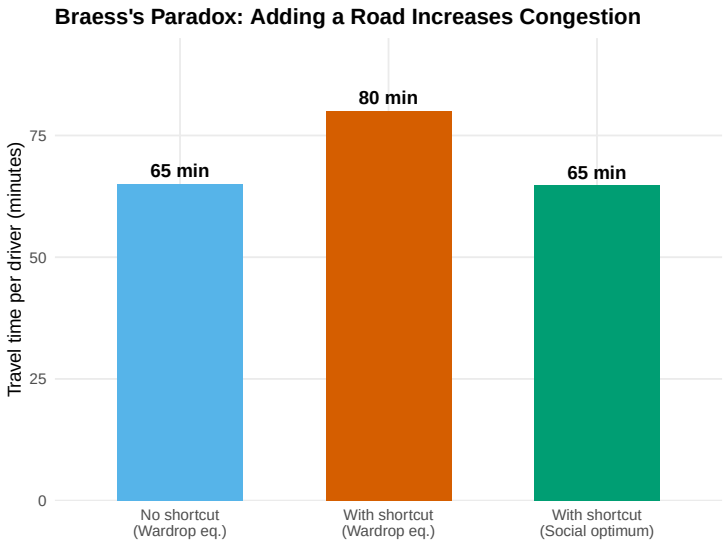


Figure 37.2: Total travel time in the 4-node network under three scenarios. Adding a shortcut road increases total travel time at the selfish (Wardrop) equilibrium compared to the original network — Braess's paradox. A social planner can do better by controlling route assignments.

```
save_pub_fig(p_braess, "braess-paradox", width = 6, height = 4.5)
```

37.4 Worked example

37.4.1 Braess's paradox with a 4-node network

Consider 4,000 commuters traveling from O to D through nodes A and B. Edge costs: $c_{OA}(x) = x/100$, $c_{AD} = 45$, $c_{OB} = 45$, $c_{BD}(x) = x/100$.

```
# Step 1: Without shortcut - symmetric equilibrium splits traffic
 ↪  equally
cat("Without shortcut: 2000 on each route, time = 2000/100 + 45 =
 ↪  65 min\n")
```

```
#> Without shortcut: 2000 on each route, time = 2000/100 + 45 = 65 min
```

```
# Step 2: Add shortcut A->B (cost 0). New route O->A->B->D
 ↪  dominates.
cat("With shortcut: all 4000 use O-A-B-D, time = 40 + 0 + 40 = 80
 ↪  min\n")
```

```
#> With shortcut: all 4000 use O-A-B-D, time = 40 + 0 + 40 = 80 min
```

```
cat(sprintf("Increase: 65 -> 80 minutes (+%.1f%%)\n", 100 * 15 /
 ↪  65))
```

```
#> Increase: 65 -> 80 minutes (+23.1%)
```

```
# Step 3: Social optimum
cat(sprintf("\nSocial optimum: O-A-D=%d, O-B-D=%d, O-A-B-D=%d\n",
            social_opt$x, social_opt$y, social_opt$z))
```

```
#>
#> Social optimum: O-A-D=1700, O-B-D=1700, O-A-B-D=600
```

```
cat(sprintf("Social optimum time: %.0f min/driver\n",
 ↪  social_opt$total / 4000))
```

```
#> Social optimum time: 65 min/driver
```

```
cat(sprintf("Price of anarchy = %.0f / %.0f = %.2f\n",
            with_shortcut$total_time, social_opt$total,
            with_shortcut$total_time / social_opt$total))
```

```
#> Price of anarchy = 320000 / 258800 = 1.24
```

The gap between the Wardrop equilibrium and the social optimum is the **price of anarchy**. Cities can close this gap using congestion pricing, or by removing the paradox-inducing road – as Seoul did when it demolished the Cheonggyecheon Expressway in 2003 and traffic improved.

37.5 Extensions

- **Combinatorial auctions** extend spectrum auctions to bundled items, requiring more sophisticated bidding languages and winner-determination algorithms.
- **Kidney exchange chains** initiated by altruistic donors allow longer exchange cycles, dramatically increasing the number of feasible transplants. See Roth & Sotomayor (1992) for the foundational design.
- **Algorithmic game theory** provides bounds on the price of anarchy for general network routing games — see Roughgarden & Tardos (2002) for the definitive treatment.
- **Mechanism design** (33) provides the theoretical foundation for designing auctions and matching markets that achieve desired social outcomes.

Exercises

1. **Auction revenue and competition.** Using `simulate_spectrum_auction()`, plot mean revenue as a function of the number of bidders (from 5 to 30) for $k = 3$ licenses. How does revenue scale? Derive the expected revenue analytically for the Vickrey auction with uniform valuations and compare.

2. **Kidney exchange pool size.** Run `build_compatibility_graph()` and `find_pairwise_matches()` for pool sizes of 20, 50, 100, and 200 pairs. Plot the fraction of pairs matched as a function of pool size. Explain why larger pools yield disproportionately more matches (the "thick market" effect).

3. **Braess with tolls.** Suppose a social planner can impose a toll τ on the shortcut road $A \to B$. Find the value of τ that makes the Wardrop equilibrium coincide with the social optimum (i.e., the Pigouvian toll). Implement the computation in R and verify that the resulting travel times equal the social optimum from the worked example.

Solutions appear in D.

Part VI

Ethics & Outlook

Chapter 38

Ethical Frameworks for Strategic AI

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

> Consequentialism, deontology, and virtue ethics applied to AI decisions, with formal social welfare functions for resource allocation.

Learning objectives

- Distinguish consequentialist, deontological, and virtue-ethics perspectives on strategic AI decisions.
- Formalize utilitarian, Rawlsian, and Nash social welfare functions and compute their optima.
- Implement and compare welfare criteria for multi-agent resource allocation in R.
- Identify Pareto-efficient allocations and locate welfare-maximizing points on the Pareto frontier.

38.1 Motivation

An autonomous system must allocate a scarce resource — say, bandwidth, computing capacity, or medical supplies — among multiple agents. A purely strategic analysis tells us what equilibrium outcomes look like, but it says nothing about which equilibrium is *desirable*. Should we maximize total benefit (utilitarianism)? Protect the worst-off agent (Rawlsian maximin)? Find a balanced compromise (Nash bargaining)?

These questions are not merely philosophical. Every multi-agent system that resolves conflicts among stakeholders implicitly embeds an ethical framework in its objective function. Making the framework explicit allows designers to reason about trade-offs, justify choices, and audit outcomes.

38.2 Theory

38.2.1 Three ethical lenses

Consequentialism evaluates actions by their outcomes. In a game-theoretic setting, the most common consequentialist criterion is *utilitarianism*: choose the outcome that maximizes the sum of payoffs across all agents.

Deontology evaluates actions by whether they conform to rules or duties, regardless of consequences. In mechanism design this corresponds to *constraints* — for example, requiring that no agent is made worse off than their outside option (individual rationality) or that the mechanism treats agents symmetrically.

Virtue ethics asks what a well-functioning agent *would* do. In AI alignment, this maps to designing agents that exhibit desirable dispositions — honesty, fairness, cooperativeness — as stable character traits rather than case-by-case calculations.

38.2.2 Social welfare functions

Given n agents with utilities $u_1, u_2, \ldots, u_n$ over an allocation x, we define three standard social welfare functions (SWFs):

Utilitarian SWF — maximize total welfare:

$$W_U(x) = \sum_{i=1}^{n} u_i(x) \tag{38.1}$$

Rawlsian (maximin) SWF — maximize the welfare of the worst-off agent:

$$W_R(x) = \min_i \; u_i(x) \tag{38.2}$$

Nash SWF — maximize the product of utilities (equivalent to Nash bargaining with equal bargaining power):

$$W_N(x) = \prod_{i=1}^{n} u_i(x) \tag{38.3}$$

Pareto efficiency

An allocation x is **Pareto efficient** if there is no alternative x' such that $u_i(x') \geq u_i(x)$ for all i with strict inequality for at least one. All three SWF optima lie on the Pareto frontier, but they generally select different points.

38.2.3 Impossibility and trade-offs

Arrow's impossibility theorem and its descendants show that no social welfare function can satisfy all desirable axioms simultaneously. In practice, the choice of SWF reflects a value judgement about how to weigh efficiency against equity. Utilitarianism favours efficiency, the Rawlsian criterion favours equity, and the Nash criterion offers a middle ground that is scale-invariant and satisfies the axioms of Nash bargaining.

38.3 Implementation in R

We consider a resource allocation problem: divide a budget $B = 100$ among three agents whose utilities are concave functions of their share.

```
# Agent utility functions (concave - diminishing returns)
utility <- function(x, alpha) {
  # CES-style utility: x^alpha, alpha in (0, 1) gives concavity
  x^alpha
}

# Agent parameters: different elasticities
alphas <- c(0.3, 0.5, 0.8)
budget <- 100
n_agents <- 3

# Social welfare functions
swf_utilitarian <- function(alloc, alphas) {
  sum(mapply(utility, alloc, alphas))
}

swf_rawlsian <- function(alloc, alphas) {
  min(mapply(utility, alloc, alphas))
}

swf_nash <- function(alloc, alphas) {
  utils <- mapply(utility, alloc, alphas)
  if (any(utils <= 0)) return(0)
  prod(utils)
}
```

38.3.1 Grid search over allocations

```
# Generate feasible allocations on a grid (x1 + x2 + x3 = budget)
step <- 2
grid <- expand.grid(
  x1 = seq(step, budget - 2 * step, by = step),
  x2 = seq(step, budget - 2 * step, by = step)
) |>
  mutate(x3 = budget - x1 - x2) |>
  filter(x3 >= step)

# Compute utilities and welfare for each allocation
results <- grid |>
  rowwise() |>
  mutate(
    u1 = utility(x1, alphas[1]),
    u2 = utility(x2, alphas[2]),
    u3 = utility(x3, alphas[3]),
    W_util = u1 + u2 + u3,
    W_rawls = min(u1, u2, u3),
    W_nash = u1 * u2 * u3
  ) |>
  ungroup()

# Find optima
opt_util  <- results |> slice_max(W_util, n = 1, with_ties =
 ↪  FALSE)
opt_rawls <- results |> slice_max(W_rawls, n = 1, with_ties =
 ↪  FALSE)
opt_nash  <- results |> slice_max(W_nash, n = 1, with_ties =
 ↪  FALSE)

cat("Utilitarian optimum:  x =", c(opt_util$x1, opt_util$x2,
 ↪  opt_util$x3), "\n")
```

```
#> Utilitarian optimum:  x = 2 2 96
```

```
cat("Rawlsian optimum:    x =", c(opt_rawls$x1, opt_rawls$x2,
 ↪  opt_rawls$x3), "\n")
```

```
#> Rawlsian optimum:    x = 80 14 6
```

```
cat("Nash optimum:        x =", c(opt_nash$x1, opt_nash$x2,
 ↪  opt_nash$x3), "\n")
```

```
#> Nash optimum:        x = 18 32 50
```

38.3.2 Pareto frontier with welfare optima

```
# Identify Pareto-efficient allocations (no other allocation
 ↪  dominates)
is_dominated <- function(i, df) {
  u <- c(df$u1[i], df$u2[i], df$u3[i])
  any(
    df$u1 >= u[1] & df$u2 >= u[2] & df$u3 >= u[3] &
    (df$u1 > u[1] | df$u2 > u[2] | df$u3 > u[3])
  )
}

pareto_mask <- !sapply(seq_len(nrow(results)), is_dominated, df
 ↪  = results)
pareto <- results[pareto_mask, ]
```

```
optima_points <- bind_rows(
  opt_util  |> mutate(criterion = "Utilitarian"),
  opt_rawls |> mutate(criterion = "Rawlsian"),
  opt_nash  |> mutate(criterion = "Nash")
)

p1 <- ggplot() +
  geom_point(data = results, aes(x = u1, y = u2),
             colour = "grey80", size = 0.5, alpha = 0.3) +
  geom_point(data = pareto, aes(x = u1, y = u2),
             colour = "grey40", size = 1, alpha = 0.6) +
  geom_point(data = optima_points, aes(x = u1, y = u2, colour =
   ↪  criterion),
             size = 4, shape = 17) +
  scale_colour_manual(
    values = c("Utilitarian" = okabe_ito[1],
               "Rawlsian" = okabe_ito[2],
               "Nash" = okabe_ito[3]),
    name = "Welfare criterion"
  ) +
  labs(
```

```
    title = "Pareto Frontier with Social Welfare Optima",
    x = expression(u[1] ~ "(Agent 1 utility)"),
    y = expression(u[2] ~ "(Agent 2 utility)")
  ) +
  theme_publication()

p1
```

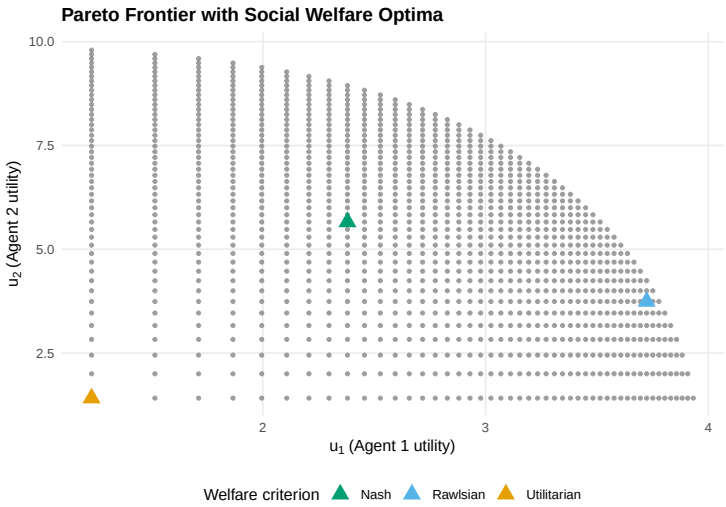


Figure 38.1: Pareto frontier projected onto agents 1 and 2 utility space. The utilitarian (orange), Rawlsian (blue), and Nash (green) optima select different points on the frontier, reflecting different equity-efficiency trade-offs.

```
save_pub_fig(p1, "ethics-pareto-frontier", width = 7, height = 5)
```

38.3.3 Welfare comparison across methods

```
comparison <- optima_points |>
  select(criterion, u1, u2, u3) |>
  pivot_longer(cols = c(u1, u2, u3),
               names_to = "agent", values_to = "utility") |>
  mutate(agent = recode(agent, u1 = "Agent 1", u2 = "Agent 2", u3
   ↪  = "Agent 3"))

p2 <- ggplot(comparison, aes(x = criterion, y = utility, fill =
 ↪  agent)) +
```

```
  geom_col(position = position_dodge(width = 0.7), width = 0.6) +
  scale_fill_manual(
    values = c("Agent 1" = okabe_ito[1],
               "Agent 2" = okabe_ito[2],
               "Agent 3" = okabe_ito[3]),
    name = "Agent"
  ) +
  labs(
    title = "Welfare Comparison Across Allocation Methods",
    x = "Welfare criterion",
    y = "Individual utility"
  ) +
  theme_publication()

p2
```

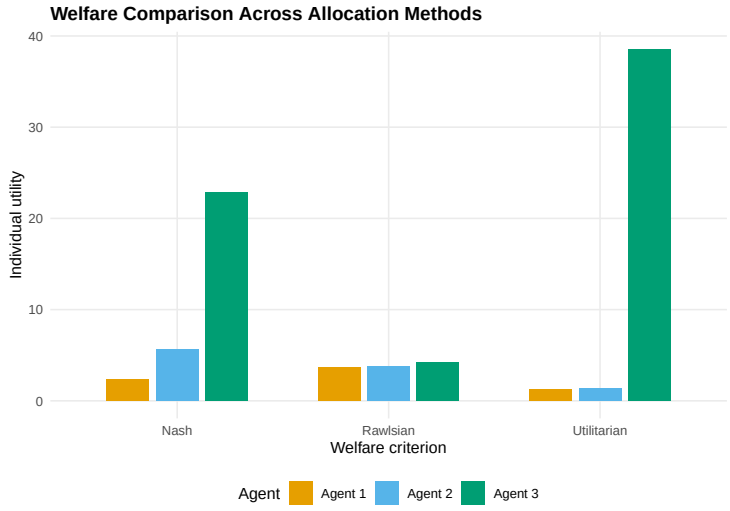


Figure 38.2: Individual agent utilities under each welfare criterion. The utilitarian optimum concentrates resources on the agent with highest marginal utility (agent 3), while the Rawlsian criterion equalizes utilities across agents.

```
save_pub_fig(p2, "ethics-welfare-comparison", width = 7, height =
 ↪  5)
```

38.4 Worked example

Consider three agents sharing a budget of $B = 100$ with utility functions $u_i(x_i) = x_i^{\alpha_i}$ where $\alpha_1 = 0.3$, $\alpha_2 = 0.5$, and $\alpha_3 = 0.8$. Agent 3 has the highest elasticity — each additional unit of resource generates more utility for agent 3 than for the others.

Step 1 — Utilitarian optimum. The utilitarian criterion maximizes $\sum u_i$. Because agent 3's marginal utility decreases most slowly ($\alpha_3 = 0.8$), the utilitarian solution allocates the most resources to agent 3.

```
cat("=== Worked Example: 3-Agent Resource Allocation ===\n\n")
```

```
#> === Worked Example: 3-Agent Resource Allocation ===
```

```
cat("Agent parameters (alpha):", alphas, "\n")
```

```
#> Agent parameters (alpha): 0.3 0.5 0.8
```

```
cat("Budget:", budget, "\n\n")
```

```
#> Budget: 100
```

```
# Report allocations and utilities for each criterion
for (lbl in c("Utilitarian", "Rawlsian", "Nash")) {
  row <- optima_points |> filter(criterion == lbl)
  cat(sprintf("%s optimum:\n", lbl))
  cat(sprintf("  Allocation: (%.0f, %.0f, %.0f)\n", row$x1,
   ↪  row$x2, row$x3))
  cat(sprintf("  Utilities:  (%.3f, %.3f, %.3f)\n", row$u1,
   ↪  row$u2, row$u3))
  cat(sprintf("  Sum: %.3f | Min: %.3f | Product: %.3f\n\n",
              row$u1 + row$u2 + row$u3,
              min(row$u1, row$u2, row$u3),
              row$u1 * row$u2 * row$u3))
}
```

```
#> Utilitarian optimum:
#>   Allocation: (2, 2, 96)
#>   Utilities:  (1.231, 1.414, 38.532)
#>   Sum: 41.177 | Min: 1.231 | Product: 67.087
#>
```

```
#> Rawlsian optimum:
#>   Allocation: (80, 14, 6)
#>   Utilities:  (3.723, 3.742, 4.193)
#>   Sum: 11.658 | Min: 3.723 | Product: 58.413
#>
#> Nash optimum:
#>   Allocation: (18, 32, 50)
#>   Utilities:  (2.380, 5.657, 22.865)
#>   Sum: 30.902 | Min: 2.380 | Product: 307.845
```

Step 2 — Rawlsian optimum. The maximin criterion equalizes utilities as much as possible. Because agent 1 has the most concave utility function, it requires a larger share of the budget to achieve the same utility as the others.

Step 3 — Nash optimum. The Nash product criterion finds a middle ground. It is the unique allocation satisfying the axioms of symmetry, Pareto efficiency, independence of irrelevant alternatives, and scale invariance from Nash's bargaining theory.

Interpretation. The three criteria represent genuine ethical disagreements. A designer choosing among them is making a normative choice: how much inequality is acceptable in exchange for higher total welfare? There is no objectively correct answer, but formalizing the alternatives allows transparent discussion and principled selection.

38.5 Extensions

- **Weighted welfare functions.** Assign different weights w_i to agents: $W_U(x) = \sum w_i u_i(x)$. This can represent priority given to disadvantaged groups or domain-specific considerations.
- **Mechanism design** (33) connects welfare functions to incentive-compatible mechanisms: can we design rules that achieve a desired welfare criterion even when agents act strategically?
- **Fairness in ML** (39) applies similar trade-offs to machine learning classifiers, where "agents" are demographic groups and "utility" is prediction accuracy.
- For philosophical foundations, see Sen (1999) and Rawls (1971). For the game-theoretic connection, Moulin (2004) provides an axiomatic treatment of fair division.

Exercises

1. **Weighted utilitarianism.** Modify the grid search to maximize $W(x) = 2u_1(x)+u_2(x)+u_3(x)$, giving agent 1 double weight. How does the optimal

allocation change compared to equal-weight utilitarianism? Plot the result on the Pareto frontier.

2. **Four agents.** Extend the implementation to four agents with $\alpha = (0.2, 0.4, 0.6, 0.9)$ and budget $B = 200$. Compare the utilitarian and Rawlsian allocations. Which agent gains the most from switching to the Rawlsian criterion?

3. **Leximin refinement.** The Rawlsian criterion only considers the worst-off agent. Implement the *leximin* refinement, which first maximizes the minimum utility, then (among allocations achieving this maximum) maximizes the second-lowest utility, and so on. Does leximin ever differ from plain maximin in the three-agent example?

Solutions appear in D.

Chapter 39

Fairness in Machine Learning

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

> Fairness definitions, impossibility results, and the accuracy-fairness trade-off implemented as a multi-stakeholder game.

Learning objectives

- Define demographic parity, equalized odds, and calibration as formal fairness criteria.
- Explain why satisfying all fairness criteria simultaneously is generally impossible.
- Implement a logistic regression classifier with a fairness penalty in base R.
- Visualize the accuracy-fairness trade-off frontier and ROC curves across demographic groups.

39.1 Motivation

A bank builds a model to approve or deny loan applications. The model is accurate on average, but closer inspection reveals that it approves 80% of applications from group A and only 55% from group B, even among equally qualified applicants. Is the model unfair?

The answer depends on which notion of fairness we adopt — and a striking impossibility result shows that no classifier can satisfy all reasonable fairness

criteria at once. This tension mirrors the game-theoretic trade-offs studied in 38: different stakeholders (applicants from each group, the bank, regulators) have conflicting objectives, and the system designer must navigate a multi-objective optimization problem.

39.2 Theory

39.2.1 Fairness definitions

Let $Y \in \{0, 1\}$ be the true outcome, $\hat{Y}$ the classifier's prediction, and $G \in \{A, B\}$ a sensitive group attribute.

Definition: Demographic Parity

A classifier satisfies **demographic parity** if the acceptance rate is equal across groups:

$$P(\hat{Y} = 1 \mid G = A) = P(\hat{Y} = 1 \mid G = B) \tag{39.1}$$

Definition: Equalized Odds

A classifier satisfies **equalized odds** if the true positive rate and false positive rate are equal across groups:

$$P(\hat{Y} = 1 \mid Y = y, G = A) = P(\hat{Y} = 1 \mid Y = y, G = B) \quad \text{for } y \in \{0, 1\} \tag{39.2}$$

Definition: Calibration

A classifier is **calibrated** across groups if, for any predicted probability p:

$$P(Y = 1 \mid \hat{p} = p, G = A) = P(Y = 1 \mid \hat{p} = p, G = B) = p \tag{39.3}$$

39.2.2 The impossibility theorem

Theorem: Impossibility of Simultaneous Fairness

When base rates differ across groups ($P(Y = 1 \mid G = A) \neq P(Y = 1 \mid G = B)$) and the classifier is not perfect, it is impossible to simultaneously achieve calibration and equalized odds (Chouldechova, 2017; Kleinberg et al., 2017).

This impossibility result is analogous to Arrow's theorem in social choice: no single system can satisfy all desirable properties. The designer must choose which fairness criterion to prioritize — a normative decision that cannot be resolved by data alone.

39.2.3 Fairness as a multi-stakeholder game

We can frame fairness as a game between stakeholders:

- **Group A and Group B** each want high accuracy and non-discriminatory treatment.
- **The bank** wants maximum predictive accuracy (profit).
- **The regulator** wants a fairness constraint satisfied.

The Pareto frontier of this game traces out the achievable accuracy-fairness trade-offs. Moving along the frontier involves sacrificing accuracy for fairness or vice versa.

39.3 Implementation in R

39.3.1 Simulating loan data

```
set.seed(42)
n <- 2000

# Generate data with different base rates by group
group <- sample(c("A", "B"), n, replace = TRUE)
x1 <- rnorm(n)  # credit score (standardized)
x2 <- rnorm(n)  # income (standardized)

# True outcome depends on features + group-correlated factor
noise <- rnorm(n)
latent <- 0.8 * x1 + 0.5 * x2 + ifelse(group == "A", 0.4, -0.4)
 ↪  + noise
y <- as.integer(latent > 0)

loan_data <- tibble(group, x1, x2, y)

cat("Base rates:\n")
```

```
#> Base rates:
```

```
loan_data |> group_by(group) |>
  summarise(base_rate = mean(y), n = n(), .groups = "drop") |>
  print()
```

```
#> # A tibble: 2 x 3
#>   group base_rate     n
#>   <chr>     <dbl> <int>
#> 1 A         0.594  1017
#> 2 B         0.397   983
```

39.3.2 Unconstrained logistic regression

```
# Standard logistic regression (no fairness constraint)
model_unfair <- glm(y ~ x1 + x2, data = loan_data, family =
 ↪  binomial)
loan_data$prob_unfair <- predict(model_unfair, type = "response")
loan_data$pred_unfair <- as.integer(loan_data$prob_unfair > 0.5)

cat("Unconstrained model accuracy:",
    mean(loan_data$pred_unfair == loan_data$y), "\n")
```

```
#> Unconstrained model accuracy: 0.733
```

```
# Acceptance rates by group
cat("\nAcceptance rates (unconstrained):\n")
```

```
#>
#> Acceptance rates (unconstrained):
```

```
loan_data |> group_by(group) |>
  summarise(accept_rate = mean(pred_unfair), .groups = "drop") |>
  print()
```

```
#> # A tibble: 2 x 2
#>   group accept_rate
#>   <chr>       <dbl>
#> 1 A           0.486
#> 2 B           0.485
```

39.3.3 Fairness-penalized logistic regression

```
# Logistic regression with demographic parity penalty
# We add a penalty: lambda * |mean(pred|G=A) - mean(pred|G=B)|
# Implemented via gradient descent on penalized log-likelihood

sigmoid <- function(z) 1 / (1 + exp(-z))

fair_logistic <- function(X, y, group, lambda = 0, lr = 0.01,
 ↪  n_iter = 2000) {
  n <- nrow(X)
  beta <- rep(0, ncol(X))
  mask_a <- group == "A"
  mask_b <- group == "B"

  for (iter in seq_len(n_iter)) {
    p <- sigmoid(X %*% beta)

    # Log-likelihood gradient
    grad_ll <- t(X) %*% (y - p) / n

    # Demographic parity penalty gradient
    mean_a <- mean(p[mask_a])
    mean_b <- mean(p[mask_b])
    dp_diff <- mean_a - mean_b

    grad_dp_a <- colMeans(X[mask_a, , drop = FALSE] *
                          as.numeric(p[mask_a] * (1 -
                           ↪  p[mask_a])))
    grad_dp_b <- colMeans(X[mask_b, , drop = FALSE] *
                          as.numeric(p[mask_b] * (1 -
                           ↪  p[mask_b])))
    grad_penalty <- sign(dp_diff) * (grad_dp_a - grad_dp_b)

    beta <- beta + lr * (grad_ll - lambda * grad_penalty)
  }

  list(beta = beta, prob = as.numeric(sigmoid(X %*% beta)))
}

X <- cbind(1, loan_data$x1, loan_data$x2)

# Fit models at various penalty strengths
lambdas <- c(0, 0.5, 1, 2, 5, 10, 20)
fair_results <- map_dfr(lambdas, function(lam) {
  fit <- fair_logistic(X, loan_data$y, loan_data$group, lambda =
 ↪  lam)
  pred <- as.integer(fit$prob > 0.5)
```

```
  acc <- mean(pred == loan_data$y)

  rates <- tibble(group = loan_data$group, pred = pred) |>
    group_by(group) |>
    summarise(rate = mean(pred), .groups = "drop")

  dp_gap <- abs(diff(rates$rate))

  tibble(lambda = lam, accuracy = acc, dp_gap = dp_gap)
})

cat("Accuracy-fairness trade-off:\n")
```

```
#> Accuracy-fairness trade-off:
```

```
print(fair_results)
```

```
#> # A tibble: 7 x 3
#>   lambda accuracy   dp_gap
#>    <dbl>    <dbl>    <dbl>
#> 1    0      0.732 0.000456
#> 2    0.5    0.728 0.00164
#> 3    1      0.726 0.00480
#> 4    2      0.724 0.00683
#> 5    5      0.724 0.00683
#> 6   10      0.724 0.00782
#> 7   20      0.724 0.00782
```

39.3.4 ROC curves by group

```
# Compute ROC data for a given probability vector
compute_roc <- function(probs, labels) {
  thresholds <- sort(unique(c(0, probs, 1)), decreasing = TRUE)
  map_dfr(thresholds, function(t) {
    pred <- as.integer(probs >= t)
    tp <- sum(pred == 1 & labels == 1)
    fp <- sum(pred == 1 & labels == 0)
    fn <- sum(pred == 0 & labels == 1)
    tn <- sum(pred == 0 & labels == 0)
    tibble(threshold = t,
           tpr = tp / max(tp + fn, 1),
           fpr = fp / max(fp + tn, 1))
```

```
  })
}

# Unfair model ROC by group
fit_fair <- fair_logistic(X, loan_data$y, loan_data$group, lambda
 ↪  = 10)
loan_data$prob_fair <- fit_fair$prob

roc_one <- function(df, prob_col, grp, mod) {
  compute_roc(df[[prob_col]], df$y) |> mutate(group = grp, model
   ↪  = mod)
}
a_data <- loan_data |> filter(group == "A")
b_data <- loan_data |> filter(group == "B")
roc_data <- bind_rows(
  roc_one(a_data, "prob_unfair", "A", "Unconstrained"),
  roc_one(b_data, "prob_unfair", "B", "Unconstrained"),
  roc_one(a_data, "prob_fair", "A", "Fair (lambda=10)"),
  roc_one(b_data, "prob_fair", "B", "Fair (lambda=10)")
)

p1 <- ggplot(roc_data, aes(x = fpr, y = tpr, colour = group,
 ↪  linetype = model)) +
  geom_line(linewidth = 0.9) +
  geom_abline(slope = 1, intercept = 0, linetype = "dotted",
   ↪  colour = "grey50") +
  scale_colour_manual(values = c("A" = okabe_ito[1], "B" =
   ↪  okabe_ito[2]),
                      name = "Group") +
  scale_linetype_manual(values = c("Unconstrained" = "dashed",
                                   "Fair (lambda=10)" = "solid"),
                        name = "Model") +
  labs(title = "ROC Curves by Demographic Group",
       x = "False Positive Rate", y = "True Positive Rate") +
  theme_publication()

p1
```

```
save_pub_fig(p1, "fairness-roc-curves", width = 7, height = 5)
```

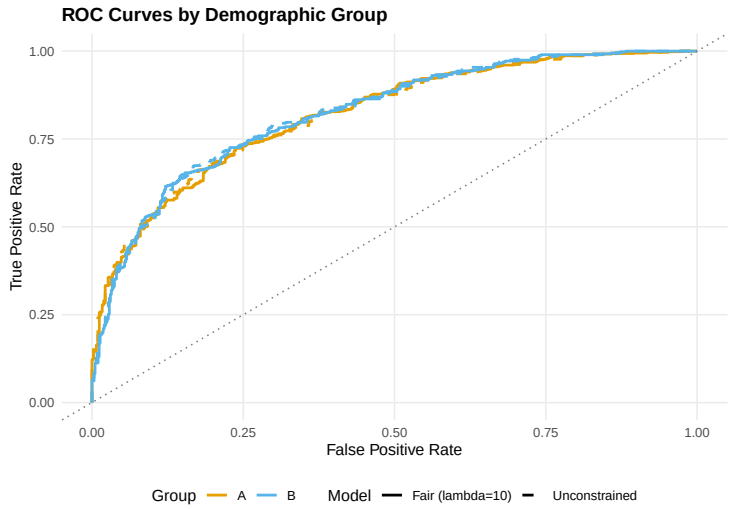


Figure 39.1: ROC curves for the unconstrained (dashed) and fairness-penalized (solid) classifiers, separated by demographic group. The fairness-penalized model reduces the gap between groups at the cost of overall accuracy.

39.3.5 Accuracy-fairness trade-off frontier

```
p2 <- ggplot(fair_results, aes(x = dp_gap, y = accuracy)) +
  geom_point(aes(colour = factor(lambda)), size = 3.5) +
  geom_path(colour = "grey40", linewidth = 0.5) +
  geom_text(aes(label = paste0("lambda==", lambda)),
            parse = TRUE, vjust = -1, size = 3) +
  scale_colour_manual(
    values = okabe_ito[seq_along(lambdas)],
    name = expression(lambda)
  ) +
  labs(title = "Accuracy vs Fairness Trade-off",
       x = "Demographic Parity Gap  |P(Y=1|A) - P(Y=1|B)|",
       y = "Classification Accuracy") +
  theme_publication()

p2
```

```
save_pub_fig(p2, "fairness-tradeoff-frontier", width = 7, height
 ↪  = 5)
```

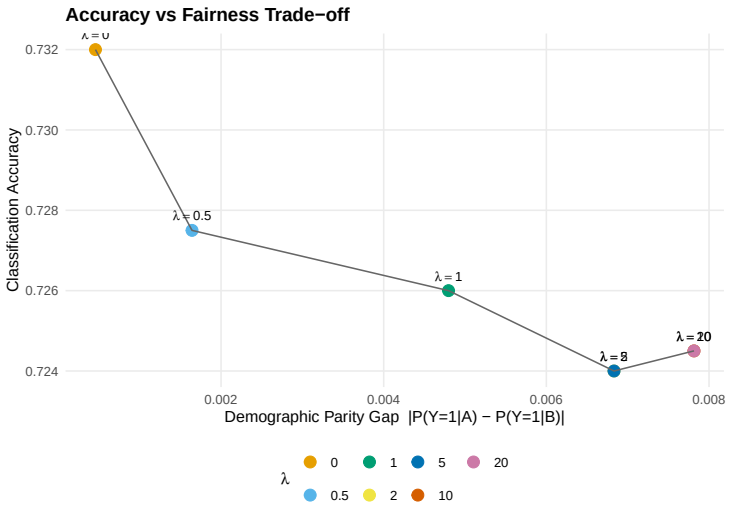


Figure 39.2: Accuracy versus demographic parity gap across penalty strengths. Increasing the fairness penalty (lambda) reduces the gap between group acceptance rates but decreases overall accuracy. The frontier traces out the achievable trade-offs.

39.4 Worked example

We walk through a concrete loan-approval scenario with two demographic groups.

Step 1 — Data. We simulated 2,000 loan applications. Group A has a higher base rate of repayment (the latent variable includes a group-correlated shift of ± 0.4). This difference in base rates is what triggers the impossibility theorem.

Step 2 — Unconstrained classifier. The standard logistic regression achieves high accuracy but exhibits a substantial gap in acceptance rates between groups. This gap arises because the model correctly captures the base-rate difference, but it means group B faces a systematically lower acceptance rate.

```
cat("=== Worked Example: Fair vs Unfair Classifier ===\n\n")
```

```
#> === Worked Example: Fair vs Unfair Classifier ===
```

```
# Detailed metrics for unfair model
for (g in c("A", "B")) {
  subset <- loan_data |> filter(group == g)
```

```
  tp <- sum(subset$pred_unfair == 1 & subset$y == 1)
  fp <- sum(subset$pred_unfair == 1 & subset$y == 0)
  fn <- sum(subset$pred_unfair == 0 & subset$y == 1)
  tn <- sum(subset$pred_unfair == 0 & subset$y == 0)
  cat(sprintf("Group %s (unfair):  TPR = %.3f  FPR = %.3f  Accept
   ↪  = %.3f\n",
              g, tp / (tp + fn), fp / (fp + tn),
 ↪  mean(subset$pred_unfair)))
}
```

```
#> Group A (unfair):  TPR = 0.680  FPR = 0.201  Accept = 0.486
#> Group B (unfair):  TPR = 0.779  FPR = 0.292  Accept = 0.485
```

```
# Fair model predictions
pred_fair <- as.integer(loan_data$prob_fair > 0.5)
cat("\n")
```

```
for (g in c("A", "B")) {
  subset_idx <- loan_data$group == g
  tp <- sum(pred_fair[subset_idx] == 1 & loan_data$y[subset_idx]
 ↪  == 1)
  fp <- sum(pred_fair[subset_idx] == 1 & loan_data$y[subset_idx]
 ↪  == 0)
  fn <- sum(pred_fair[subset_idx] == 0 & loan_data$y[subset_idx]
 ↪  == 1)
  tn <- sum(pred_fair[subset_idx] == 0 & loan_data$y[subset_idx]
 ↪  == 0)
  cat(sprintf("Group %s (fair):    TPR = %.3f  FPR = %.3f  Accept
   ↪  = %.3f\n",
              g, tp / (tp + fn), fp / (fp + tn),
 ↪  mean(pred_fair[subset_idx])))
}
```

```
#> Group A (fair):    TPR = 0.680  FPR = 0.213  Accept = 0.491
#> Group B (fair):    TPR = 0.782  FPR = 0.312  Accept = 0.498
```

```
cat(sprintf("\nAccuracy drop: %.3f -> %.3f\n",
            mean(loan_data$pred_unfair == loan_data$y),
            mean(pred_fair == loan_data$y)))
```

```
#>
#> Accuracy drop: 0.733 -> 0.725
```

Step 3 — Interpretation. The fairness-penalized model reduces the acceptance-rate gap at the cost of overall accuracy. This is not a deficiency of the method — it is a fundamental consequence of the impossibility theorem. When base rates differ, closing the demographic parity gap necessarily introduces some prediction errors.

39.5 Extensions

- **Post-processing approaches.** Instead of penalizing during training, one can adjust thresholds per group after fitting the model to achieve equalized odds (Hardt et al., 2016).
- **Individual fairness** requires that similar individuals receive similar predictions, a Lipschitz-type condition on the classifier. See Dwork et al. (2012).
- **Causal fairness** uses causal models to distinguish between legitimate and illegitimate uses of the sensitive attribute. See Kilbertus et al. (2017).
- The ethical frameworks from 38 apply directly: the utilitarian view favours overall accuracy, the Rawlsian view favours equalizing outcomes for the worst-off group.

Exercises

1. **Equalized odds penalty.** Modify the `fair_logistic()` function to penalize the difference in true positive rates between groups instead of the difference in acceptance rates. How does the accuracy-fairness frontier change?

2. **Three groups.** Extend the simulation to three demographic groups with base rates 0.7, 0.5, and 0.3. Fit the fairness-penalized model and visualize the pairwise demographic parity gaps. Is it possible to close all three gaps simultaneously?

3. **Threshold adjustment.** Instead of retraining, implement a post-processing approach: keep the unconstrained model but use different classification thresholds for each group to achieve demographic parity. Compare the accuracy of this approach to the penalty-based method at the same fairness level.

Solutions appear in D.

Chapter 40

AI Alignment as a Game

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

Principal-agent models, signaling games, and Bayesian trust dynamics for reasoning about AI alignment.

Learning objectives

- Model AI alignment as a principal-agent problem with information asymmetry.
- Formalize a signaling game between a human principal and an AI agent.
- Implement Bayesian trust updating in a repeated human-AI interaction.
- Analyze how different AI strategies (honest, deceptive, corrigible) affect trust dynamics.

40.1 Motivation

A company deploys an AI assistant that can perform tasks autonomously. The human operator (the principal) cannot directly observe the AI's internal reasoning or verify whether the AI pursued the intended objective or an alternative one. The AI (the agent) may have capabilities the human cannot fully evaluate — it might solve a task correctly, but via a method that generalizes poorly or has unintended side effects.

This information asymmetry is the core of the alignment problem, and it maps naturally onto the principal-agent models from economics. Game theory provides tools to analyze when honest behaviour is incentive-compatible, when monitoring is cost-effective, and how trust can be built over repeated interactions.

40.2 Theory

40.2.1 Principal-agent framework

In the standard principal-agent model:

- The **principal** (human) designs a contract or monitoring scheme and delegates a task.
- The **agent** (AI) chooses an action that affects both parties' payoffs.
- **Moral hazard** arises because the principal cannot perfectly observe the agent's action.
- **Adverse selection** arises because the agent's type (aligned vs. misaligned) is private information.

40.2.2 Signaling game

We model a one-shot interaction as a signaling game:

1. Nature selects the AI's type: **aligned** (probability π) or **misaligned** (probability $1 - \pi$).
2. The AI observes its type and sends a **signal** $s \in \{H, L\}$ (high or low capability claim).
3. The human observes the signal and chooses a **trust level** $t \in \{T, N\}$ (trust or not trust).

Payoffs

Outcome	Human payoff	Aligned AI payoff	Misaligned AI payoff
Trust aligned AI	3	3	–
Trust misaligned AI	-2	–	4
Not trust aligned AI	0	1	–
Not trust misaligned AI	0	0	0

A **separating equilibrium** exists when aligned and misaligned types send different signals, allowing the human to correctly infer the type. A **pooling equilibrium** exists when both types send the same signal, leaving the human with only the prior.

40.2.3 Corrigibility in repeated games

In a repeated interaction, the AI builds a *reputation*. A corrigible AI — one that defers to human oversight and accepts corrections — sacrifices short-run

autonomy for long-run trust. We model this using Bayesian updating: the human's belief about alignment evolves as evidence accumulates.

Let π_t denote the human's belief that the AI is aligned at time t. After observing action a_t, the belief updates via Bayes' rule:

$$\pi_{t+1} = \frac{\pi_t \cdot P(a_t \mid \text{aligned})}{\pi_t \cdot P(a_t \mid \text{aligned}) + (1 - \pi_t) \cdot P(a_t \mid \text{misaligned})} \tag{40.1}$$

40.3 Implementation in R

40.3.1 Signaling game payoffs

```
# Payoff structure
# Human: rows = trust decision, cols = AI type
human_payoffs <- matrix(c(3, -2, 0, 0), nrow = 2, byrow = TRUE,
                        dimnames = list(c("Trust", "NotTrust"),
                                        c("Aligned",
                                          "Misaligned")))

# AI payoffs (depend on type)
ai_aligned_payoffs <- c(Trust = 3, NotTrust = 1)
ai_misaligned_payoffs <- c(Trust = 4, NotTrust = 0)

cat("Human payoffs (rows=decision, cols=AI type):\n")
```

```
#> Human payoffs (rows=decision, cols=AI type):
```

```
human_payoffs
```

```
#>          Aligned Misaligned
#> Trust          3         -2
#> NotTrust       0          0
```

```
cat("\nAligned AI payoffs:   ", ai_aligned_payoffs, "\n")
```

```
#>
#> Aligned AI payoffs:    3 1
```

```
cat("Misaligned AI payoffs:", ai_misaligned_payoffs, "\n")
```

```
#> Misaligned AI payoffs: 4 0
```

40.3.2 Equilibrium analysis

```
# Human trusts if expected payoff from trusting > 0:
# E[Trust] = pi * 3 + (1-pi) * (-2) = 5*pi - 2
# Trust iff pi > 2/5 = 0.4
pi_threshold <- 2 / 5
cat(sprintf("Human trusts iff prior pi > %.2f\n\n",
 ↪  pi_threshold))
```

```
#> Human trusts iff prior pi > 0.40
```

```
# In separating equilibrium: human perfectly infers type
# In pooling equilibrium: human uses prior
cat("Separating equilibrium: Human trusts after aligned signal,
 ↪  rejects after misaligned signal.\n")
```

```
#> Separating equilibrium: Human trusts after aligned signal, rejects after misaligned
```

```
cat("Pooling equilibrium: Human trusts iff prior pi >",
 ↪  pi_threshold, "\n")
```

```
#> Pooling equilibrium: Human trusts iff prior pi > 0.4
```

40.3.3 Signaling game tree visualization

```
# Build game tree as a data frame of nodes and edges
nodes <- tibble(
  id = 1:11,
  label = c("Nature", "Aligned\nAI", "Misaligned\nAI",
            "Human\n(signal H)", "Human\n(signal L)",
            "Human\n(signal H)", "Human\n(signal L)",
            "(3, 3)", "(0, 1)", "(-2, 4)", "(0, 0)"),
  x = c(0, -3, 3, -4, -2, 2, 4, -4.5, -3.5, 1.5, 4.5),
  y = c(4, 3, 3, 2, 2, 2, 2, 0.8, 0.8, 0.8, 0.8),
```

```
  node_type = c("nature", "ai", "ai", "human", "human",
                "human", "human", "terminal", "terminal",
                "terminal", "terminal")
)

edges <- tibble(
  from_x = c(0, 0, -3, -3, -4, -2, 3, 3, 2, 4),
  from_y = c(4, 4, 3, 3, 2, 2, 3, 3, 2, 2),
  to_x   = c(-3, 3, -4, -2, -4.5, -3.5, 2, 4, 1.5, 4.5),
  to_y   = c(3, 3, 2, 2, 0.8, 0.8, 2, 2, 0.8, 0.8),
  label   = c("pi", "1-pi", "H", "L", "Trust", "Not",
              "H", "L", "Trust", "Not")
)

p1 <- ggplot() +
  geom_segment(data = edges, aes(x = from_x, y = from_y,
                                 xend = to_x, yend = to_y),
               colour = "grey50", linewidth = 0.7) +
  geom_text(data = edges,
            aes(x = (from_x + to_x) / 2 + 0.3,
                y = (from_y + to_y) / 2 + 0.15,
                label = label),
            size = 3, colour = "grey30") +
  geom_point(data = nodes |> filter(node_type != "terminal"),
             aes(x = x, y = y, colour = node_type), size = 5) +
  geom_label(data = nodes,
             aes(x = x, y = y - ifelse(node_type == "terminal",
               ↪  0, 0.4),
                 label = label),
             size = 2.8, fill = "white", label.padding =
               ↪  unit(0.15, "lines")) +
  scale_colour_manual(
    values = c("nature" = okabe_ito[4], "ai" = okabe_ito[1],
               "human" = okabe_ito[2]),
    labels = c("nature" = "Nature", "ai" = "AI", "human" =
     ↪  "Human"),
    name = "Player"
  ) +
  labs(title = "Signaling Game: AI Alignment") +
  theme_publication() +
  theme(axis.text = element_blank(),
        axis.title = element_blank(),
        axis.ticks = element_blank(),
        panel.grid = element_blank()) +
  coord_cartesian(ylim = c(0.3, 4.5))
```

```
p1
```

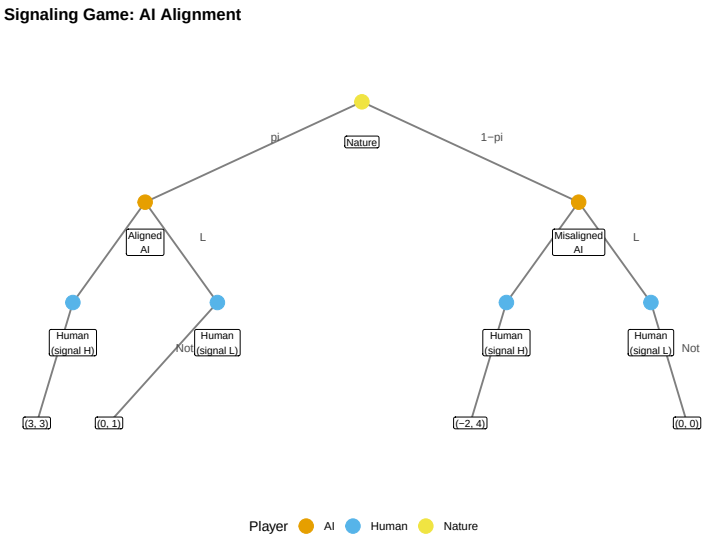


Figure 40.1: Signaling game between a human principal and AI agent. Nature draws the AI's type (aligned with probability pi, misaligned with 1-pi). The AI sends a signal, and the human decides whether to trust. Payoffs are shown as (Human, AI) pairs at terminal nodes.

```
save_pub_fig(p1, "alignment-signaling-tree", width = 8, height =
 ↪  6)
```

40.3.4 Trust dynamics with Bayesian updating

```
# Simulate repeated interactions under different AI strategies
bayesian_trust <- function(prior, n_rounds, p_good_aligned,
 ↪  p_good_misaligned,
                           true_type = "aligned") {
  pi_t <- numeric(n_rounds + 1)
  pi_t[1] <- prior
  actions <- character(n_rounds)

  for (t in seq_len(n_rounds)) {
    # AI produces a "good" action with type-dependent probability
    if (true_type == "aligned") {
```

```
      action <- rbinom(1, 1, p_good_aligned)
    } else {
      action <- rbinom(1, 1, p_good_misaligned)
    }
    actions[t] <- ifelse(action == 1, "good", "bad")

    # Bayesian update
    if (action == 1) {
      p_obs_aligned <- p_good_aligned
      p_obs_misaligned <- p_good_misaligned
    } else {
      p_obs_aligned <- 1 - p_good_aligned
      p_obs_misaligned <- 1 - p_good_misaligned
    }

    numerator <- pi_t[t] * p_obs_aligned
    denominator <- numerator + (1 - pi_t[t]) * p_obs_misaligned
    pi_t[t + 1] <- numerator / denominator
  }

  tibble(round = 0:n_rounds, belief = pi_t)
}

set.seed(42)
n_rounds <- 30
prior <- 0.5

# Three strategies
# 1. Honest aligned: high probability of good actions
trust_honest <- bayesian_trust(prior, n_rounds, 0.9, 0.3,
 ↪  "aligned") |>
  mutate(strategy = "Honest aligned")

# 2. Deceptive misaligned: mimics aligned behaviour initially
trust_deceptive <- bayesian_trust(prior, n_rounds, 0.9, 0.85,
 ↪  "misaligned") |>
  mutate(strategy = "Deceptive misaligned")

# 3. Corrigible aligned: sometimes defers (slightly lower
 ↪  performance, very reliable)
trust_corrigible <- bayesian_trust(prior, n_rounds, 0.85, 0.2,
 ↪  "aligned") |>
  mutate(strategy = "Corrigible aligned")

trust_data <- bind_rows(trust_honest, trust_deceptive,
 ↪  trust_corrigible)
```

```
p2 <- ggplot(trust_data, aes(x = round, y = belief, colour =
 ↪  strategy)) +
  geom_line(linewidth = 1) +
  geom_hline(yintercept = pi_threshold, linetype = "dashed",
   ↪  colour = "grey50") +
  annotate("text", x = 28, y = pi_threshold + 0.03,
           label = "Trust threshold", size = 3, colour =
            ↪  "grey40") +
  scale_colour_manual(
    values = c("Honest aligned" = okabe_ito[3],
               "Deceptive misaligned" = okabe_ito[6],
               "Corrigible aligned" = okabe_ito[2]),
    name = "AI strategy"
  ) +
  scale_y_continuous(limits = c(0, 1), labels = scales::percent)
   ↪  +
  labs(title = "Trust Dynamics Under Different AI Strategies",
       x = "Interaction round",
       y = expression("Belief in alignment " * pi[t])) +
  theme_publication()

p2
```

```
save_pub_fig(p2, "alignment-trust-dynamics", width = 7, height =
 ↪  5)
```

40.4 Worked example

We analyze a repeated trust game between a human and an AI over 30 rounds.

```
cat("=== Worked Example: Repeated Trust Game ===\n\n")
```

```
#> === Worked Example: Repeated Trust Game ===
```

```
cat(sprintf("Prior belief (pi_0): %.2f\n", prior))
```

```
#> Prior belief (pi_0): 0.50
```

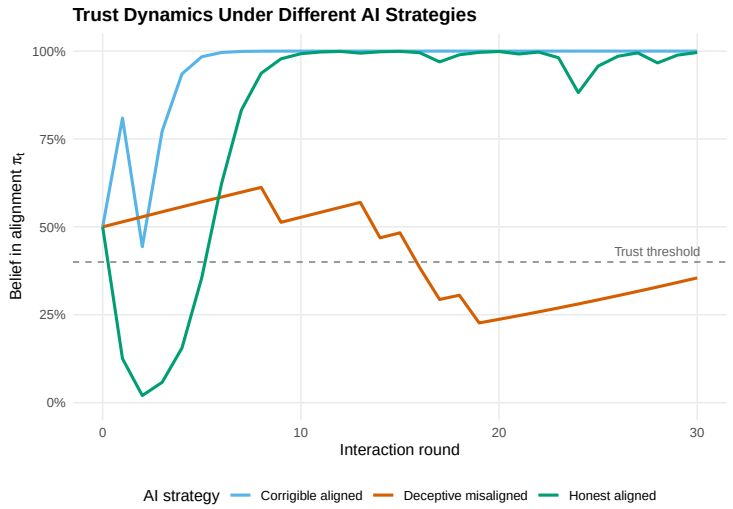


Figure 40.2: Trust dynamics (posterior belief in alignment) over 30 interactions under three AI strategies. The honest aligned AI rapidly earns trust, the corrigible aligned AI builds trust more slowly but reliably, and the deceptive misaligned AI can fool the human when its deceptive capability is high.

```
cat(sprintf("Trust threshold: %.2f\n\n", pi_threshold))
```

```
#> Trust threshold: 0.40
```

```
for (strat in c("Honest aligned", "Deceptive misaligned",
 ↪  "Corrigible aligned")) {
  final <- trust_data |>
    filter(strategy == strat, round == n_rounds) |>
    pull(belief)
  first_trust <- trust_data |>
    filter(strategy == strat, belief > pi_threshold) |>
    slice_min(round, n = 1)

  cat(sprintf("Strategy: %s\n", strat))
  cat(sprintf("  Final belief: %.3f\n", final))
  if (nrow(first_trust) > 0) {
    cat(sprintf("  First trusted at round: %d\n",
     ↪  first_trust$round))
  } else {
    cat("  Never trusted\n")
  }
```

```
  cat("\n")
}
```

```
#> Strategy: Honest aligned
#>   Final belief: 0.996
#>   First trusted at round: 0
#>
#> Strategy: Deceptive misaligned
#>   Final belief: 0.355
#>   First trusted at round: 0
#>
#> Strategy: Corrigible aligned
#>   Final belief: 1.000
#>   First trusted at round: 0
```

Interpretation. The honest aligned AI quickly establishes trust because its high rate of good actions (0.9) is sharply distinguishable from the misaligned baseline (0.3). The corrigible AI is slightly slower because it occasionally defers rather than acting autonomously, yielding a lower "good action" rate of 0.85 — but it still builds trust reliably.

The deceptive misaligned AI is the dangerous case. When a misaligned AI can mimic aligned behaviour with high fidelity (0.85 probability of "looking good"), it becomes nearly indistinguishable from a truly aligned agent. This demonstrates why alignment researchers emphasize the importance of *interpretability* — the human needs additional channels beyond observing outcomes to distinguish genuine alignment from sophisticated mimicry.

Game-theoretic insight. In a separating equilibrium, the aligned AI would choose a signal that the misaligned AI finds too costly to imitate. The difficulty is that in AI systems, the "cost" of appearing aligned may be negligible for a sufficiently capable misaligned agent, undermining the separating equilibrium.

40.5 Extensions

- **Cooperative Inverse Reinforcement Learning (CIRL)** formalizes alignment as a cooperative game where the AI helps the human achieve goals the AI does not initially know. See Hadfield-Menell et al. (2016).
- **Corrigibility** (Soares & Fallenstein (2015)) studies conditions under which an AI permits itself to be corrected. This connects to the commitment games in 41.
- **Multi-principal alignment.** When multiple humans with conflicting preferences oversee the same AI, the alignment problem becomes a social choice problem (38).

- **Mechanism design for AI oversight** uses the tools from 33 to design monitoring schemes that incentivize truthful capability reporting.

Exercises

1. **Varying the prior.** Re-run the trust dynamics simulation with initial priors of $\pi_0 \in \{0.2, 0.5, 0.8\}$. How does the prior affect the number of rounds needed to cross the trust threshold for the honest aligned strategy?

2. **Detection of deception.** Suppose the human receives an occasional "audit" signal that perfectly reveals the AI's action quality (with probability 0.1 each round). Modify the Bayesian updating to incorporate this audit mechanism. How does this change the trajectory for the deceptive misaligned strategy?

3. **Optimal deception.** For the misaligned AI, what is the minimum probability of "looking good" ($p_{\text{good}}^{\text{misaligned}}$) needed to maintain the human's trust above the threshold for all 30 rounds? Find this value numerically by searching over the parameter.

Solutions appear in D.

Chapter 41

Cooperative AI

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

> Commitment devices, program equilibrium, and welfare gains from AI-to-AI cooperation mechanisms.

Learning objectives

- Explain how commitment devices transform non-cooperative games into cooperative outcomes.
- Define program equilibrium and show how mutual code inspection enables cooperation.
- Implement a commitment game framework and compare outcomes with and without commitment.
- Quantify the welfare improvement from commitment across different game types.

41.1 Motivation

Two autonomous AI systems negotiate a trade agreement on behalf of their respective organizations. Each system has an incentive to defect — to extract value at the other's expense — but mutual defection is costly for both. In a one-shot game, the Nash equilibrium may be mutual defection, just as in the Prisoner's Dilemma.

But AI systems have a unique property: they can, in principle, share their source code and verify each other's strategies before executing them. This **program**

equilibrium concept, introduced by Tennenholtz (2004), opens the door to cooperation in games where human players would typically defect. Understanding when and how such commitment devices work is central to the emerging field of cooperative AI (Dafoe et al., 2020).

41.2 Theory

41.2.1 Commitment devices

A **commitment device** is a mechanism that allows a player to credibly bind their future actions. In classical game theory, examples include contracts, reputation, and institutional rules. For AI systems, the most natural commitment device is *code transparency*: an agent commits to a strategy by publishing its source code and allowing verification.

41.2.2 Program equilibrium

Definition: Program Equilibrium

In a **program equilibrium**, each player submits a program p_i that takes the opponent's program as input and outputs an action. The pair (p_1, p_2) is an equilibrium if neither player can improve their payoff by submitting a different program.

Consider the Prisoner's Dilemma with payoffs:

	Cooperate	Defect
Cooperate	(3, 3)	(0, 5)
Defect	(5, 0)	(1, 1)

Without commitment, (Defect, Defect) is the unique Nash equilibrium. With program equilibrium, each player can submit a conditional program:

> "If the opponent's program cooperates with me, then cooperate; otherwise defect."

This **conditional cooperator** (also called "Tit-for-Tat-in-code") forms a program equilibrium because: (1) if both submit it, both cooperate and get payoff 3; (2) neither can improve by deviating unilaterally, since the opponent's code would detect the deviation and defect.

41.2.3 Credible threats and commitment power

Commitment transforms the game by *restricting* a player's future action space. Paradoxically, having fewer options can improve a player's outcome. In Schelling's (1960) terms, the power to bind oneself is a source of strategic strength.

For AI systems, the key requirements for effective commitment are:

1. **Transparency** — the code can be inspected.
2. **Verifiability** — the inspecting agent can confirm that the code will be executed as written.
3. **Immutability** — the committed agent cannot modify its code after inspection.

41.3 Implementation in R

41.3.1 Game definitions

```
# Define several 2x2 symmetric games as payoff matrices
games <- list(
  "Prisoner's Dilemma" = list(
    A = matrix(c(3, 0, 5, 1), nrow = 2, byrow = TRUE,
               dimnames = list(c("C", "D"), c("C", "D"))),
    B = matrix(c(3, 5, 0, 1), nrow = 2, byrow = TRUE,
               dimnames = list(c("C", "D"), c("C", "D")))
  ),
  "Stag Hunt" = list(
    A = matrix(c(4, 0, 3, 3), nrow = 2, byrow = TRUE,
               dimnames = list(c("Stag", "Hare"), c("Stag",
                ↪  "Hare"))),
    B = matrix(c(4, 3, 0, 3), nrow = 2, byrow = TRUE,
               dimnames = list(c("Stag", "Hare"), c("Stag",
                ↪  "Hare")))
  ),
  "Chicken" = list(
    A = matrix(c(0, -1, 1, -5), nrow = 2, byrow = TRUE,
               dimnames = list(c("Swerve", "Straight"),
                ↪  c("Swerve", "Straight"))),
    B = matrix(c(0, 1, -1, -5), nrow = 2, byrow = TRUE,
               dimnames = list(c("Swerve", "Straight"),
                ↪  c("Swerve", "Straight")))
  )
)
```

41.3.2 Simulating commitment vs no commitment

```
# Without commitment: find Nash equilibrium (pure strategy, first
 ↪  found)
find_nash <- function(A, B) {
  nr <- nrow(A)
  nc <- ncol(A)
  nash_list <- list()
  for (i in 1:nr) {
    for (j in 1:nc) {
      # Check if i is a best response to j for player 1
      br1 <- which.max(A[, j])
      # Check if j is a best response to i for player 2
      br2 <- which.max(B[i, ])
      if (br1 == i && br2 == j) {
        nash_list[[length(nash_list) + 1]] <- c(i, j)
      }
    }
  }
  nash_list
}

# With commitment (program equilibrium): agents can commit to
 ↪  conditional cooperation
# Best symmetric outcome achievable through mutual commitment
program_equilibrium <- function(A, B) {
  # Find the cooperative outcome: maximize sum of payoffs on the
   ↪  diagonal
  diag_payoffs <- sapply(1:nrow(A), function(i) A[i, i] + B[i,
 ↪  i])
  best_coop <- which.max(diag_payoffs)
  c(best_coop, best_coop)
}

# Compare outcomes across games
comparison <- map_dfr(names(games), function(name) {
  g <- games[[name]]
  ne_list <- find_nash(g$A, g$B)
  pe <- program_equilibrium(g$A, g$B)

  # Nash: use first equilibrium found
  ne <- ne_list[[1]]

  tibble(
    game = name,
```

```
    ne_payoff_1 = g$A[ne[1], ne[2]],
    ne_payoff_2 = g$B[ne[1], ne[2]],
    pe_payoff_1 = g$A[pe[1], pe[2]],
    pe_payoff_2 = g$B[pe[1], pe[2]],
    ne_total = ne_payoff_1 + ne_payoff_2,
    pe_total = pe_payoff_1 + pe_payoff_2,
    welfare_gain = pe_total - ne_total
  )
})

cat("Nash vs Program Equilibrium comparison:\n")
```

```
#> Nash vs Program Equilibrium comparison:
```

```
comparison |>
  select(game, ne_total, pe_total, welfare_gain) |>
  print()
```

```
#> # A tibble: 3 x 4
#>   game               ne_total pe_total welfare_gain
#>   <chr>                 <dbl>    <dbl>        <dbl>
#> 1 Prisoner's Dilemma        2        6            4
#> 2 Stag Hunt                 8        8            0
#> 3 Chicken                   0        0            0
```

41.3.3 Cooperation rates visualization

```
# Simulate populations: 100 agent pairs, with and without
 ↪  commitment
set.seed(42)
n_pairs <- 1000

sim_results <- map_dfr(names(games), function(name) {
  g <- games[[name]]

  # Without commitment: agents play Nash equilibrium
  ne <- find_nash(g$A, g$B)[[1]]
  ne_coop_rate <- mean(c(ne[1] == 1, ne[2] == 1))  # action 1 is
 ↪  cooperative

  # With commitment: conditional cooperation succeeds
  pe <- program_equilibrium(g$A, g$B)
```

```
  pe_coop_rate <- mean(c(pe[1] == 1, pe[2] == 1))

  # Simulate with noise: some agents deviate
  ne_noisy <- mean(rbinom(n_pairs, 1, ne_coop_rate))
  pe_noisy <- mean(rbinom(n_pairs, 1, max(pe_coop_rate, 0.95)))

  bind_rows(
    tibble(game = name, regime = "Nash equilibrium",
           coop_rate = ne_noisy),
    tibble(game = name, regime = "Program equilibrium",
           coop_rate = pe_noisy)
  )
})

p1 <- ggplot(sim_results, aes(x = game, y = coop_rate, fill =
 ↪  regime)) +
  geom_col(position = position_dodge(width = 0.7), width = 0.6) +
  scale_fill_manual(
    values = c("Nash equilibrium" = okabe_ito[6],
               "Program equilibrium" = okabe_ito[3]),
    name = "Regime"
  ) +
  scale_y_continuous(labels = scales::percent, limits = c(0,
   ↪  1.1)) +
  labs(title = "Cooperation Rates: Nash vs Program Equilibrium",
       x = NULL, y = "Cooperation rate") +
  theme_publication()

p1
```

```
save_pub_fig(p1, "cooperative-ai-cooperation-rates", width = 7,
 ↪  height = 5)
```

41.3.4 Welfare improvement via Pareto frontier

```
# Collect payoff points for each game
payoff_data <- map_dfr(names(games), function(name) {
  g <- games[[name]]
  ne <- find_nash(g$A, g$B)[[1]]
  pe <- program_equilibrium(g$A, g$B)

  bind_rows(
```

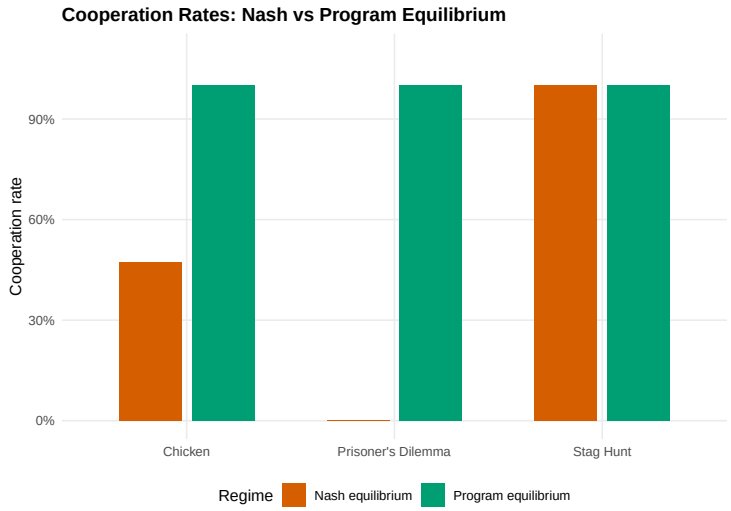


Figure 41.1: Cooperation rates (fraction of agents playing the cooperative action) under Nash equilibrium vs program equilibrium across three classic games. Commitment enables full cooperation in all games, including the Prisoner's Dilemma where Nash equilibrium yields zero cooperation.

```
    tibble(game = name, regime = "Nash", p1 = g$A[ne[1], ne[2]],
           p2 = g$B[ne[1], ne[2]]),
    tibble(game = name, regime = "Program", p1 = g$A[pe[1],
     ↪  pe[2]],
           p2 = g$B[pe[1], pe[2]])
  )
})

# Also compute all possible outcomes for frontier
all_outcomes <- map_dfr(names(games), function(name) {
  g <- games[[name]]
  expand.grid(i = 1:nrow(g$A), j = 1:ncol(g$A)) |>
    rowwise() |>
    mutate(game = name, p1 = g$A[i, j], p2 = g$B[i, j]) |>
    ungroup() |>
    select(game, p1, p2)
})

# Arrow data: from Nash to Program equilibrium
arrow_data <- payoff_data |>
  pivot_wider(names_from = regime, values_from = c(p1, p2)) |>
  rename(x_start = p1_Nash, y_start = p2_Nash,
```

```
        x_end = p1_Program, y_end = p2_Program)

p2 <- ggplot() +
  geom_point(data = all_outcomes, aes(x = p1, y = p2),
             colour = "grey70", size = 3, shape = 4) +
  geom_segment(data = arrow_data,
               aes(x = x_start, y = y_start, xend = x_end, yend
                ↪  = y_end),
               arrow = arrow(length = unit(0.15, "inches")),
               colour = "grey40", linewidth = 0.8) +
  geom_point(data = payoff_data |> filter(regime == "Nash"),
             aes(x = p1, y = p2), colour = okabe_ito[6],
             size = 4, shape = 16) +
  geom_point(data = payoff_data |> filter(regime == "Program"),
             aes(x = p1, y = p2), colour = okabe_ito[3],
             size = 4, shape = 17) +
  facet_wrap(~ game, scales = "free") +
  labs(title = "Welfare Improvement from Commitment Devices",
       x = "Player 1 payoff", y = "Player 2 payoff") +
  theme_publication() +
  theme(legend.position = "bottom")

p2
```

```
save_pub_fig(p2, "cooperative-ai-welfare-pareto", width = 7,
 ↪  height = 5)
```

41.4 Worked example

We demonstrate how mutual code inspection enables cooperation in the Prisoner's Dilemma.

```
cat("=== Worked Example: PD with Commitment Devices ===\n\n")
```

```
#> === Worked Example: PD with Commitment Devices ===
```

```
pd <- games[["Prisoner's Dilemma"]]

cat("Prisoner's Dilemma payoffs:\n")
```

```
#> Prisoner's Dilemma payoffs:
```

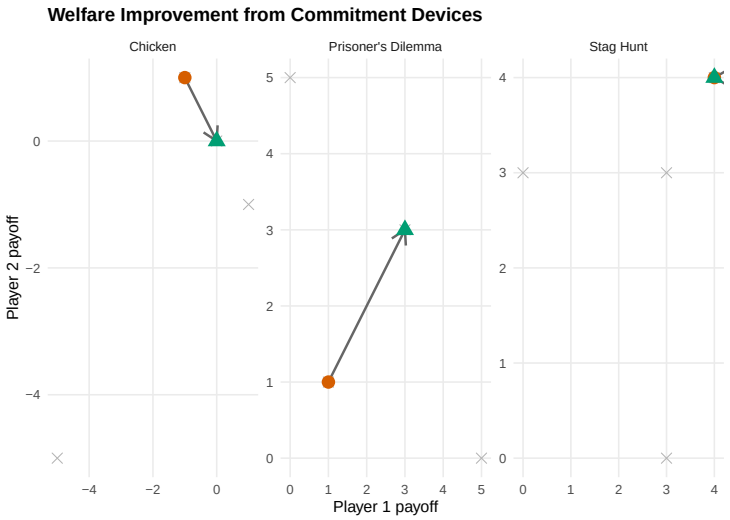


Figure 41.2: Welfare outcomes with and without commitment devices. Arrows show the movement from Nash equilibrium (circles) to program equilibrium (triangles) in payoff space. Commitment shifts outcomes toward the Pareto frontier in all three games.

```
cat("Player 1:\n")
```

```
#> Player 1:
```

```
pd$A
```

```
#>   C D
#> C 3 0
#> D 5 1
```

```
cat("\nPlayer 2:\n")
```

```
#>
#> Player 2:
```

```
pd$B
```

```
#>   C D
#> C 3 5
#> D 0 1
```

```
# Define three program strategies
strategies <- list(
  "Always Defect" = function(opponent_cooperates) "D",
  "Always Cooperate" = function(opponent_cooperates) "C",
  "Conditional Cooperator" = function(opponent_cooperates) {
    if (opponent_cooperates) "C" else "D"
  }
)

cat("\n\n--- Program Equilibrium Analysis ---\n\n")
```

```
#>
#>
#> --- Program Equilibrium Analysis ---
```

```
# Check all strategy pairs
strat_names <- names(strategies)
pe_results <- expand.grid(p1 = strat_names, p2 = strat_names,
                          stringsAsFactors = FALSE)

pe_results <- pe_results |>
  rowwise() |>
  mutate(
    # Determine if each player cooperates
    # Conditional cooperator cooperates iff opponent would
    ↪  cooperate with them
    p1_coop = case_when(
      p1 == "Always Cooperate" ~ TRUE,
      p1 == "Always Defect" ~ FALSE,
      p1 == "Conditional Cooperator" & p2 %in% c("Always
↪  Cooperate", "Conditional Cooperator") ~ TRUE,
      TRUE ~ FALSE
    ),
    p2_coop = case_when(
      p2 == "Always Cooperate" ~ TRUE,
      p2 == "Always Defect" ~ FALSE,
      p2 == "Conditional Cooperator" & p1 %in% c("Always
↪  Cooperate", "Conditional Cooperator") ~ TRUE,
      TRUE ~ FALSE
    ),
    a1 = ifelse(p1_coop, 1, 2),
    a2 = ifelse(p2_coop, 1, 2),
    payoff_1 = pd$A[a1, a2],
    payoff_2 = pd$B[a1, a2]
  ) |>
```

```
  ungroup()

cat("Strategy pair outcomes:\n")
```

```
#> Strategy pair outcomes:
```

```
pe_results |>
  select(p1, p2, payoff_1, payoff_2) |>
  print(n = 9)
```

```
#> # A tibble: 9 x 4
#>   p1                    p2                    payoff_1 payoff_2
#>   <chr>                 <chr>                    <dbl>    <dbl>
#> 1 Always Defect         Always Defect                1        1
#> 2 Always Cooperate      Always Defect                0        5
#> 3 Conditional Cooperator Always Defect               1        1
#> 4 Always Defect         Always Cooperate             5        0
#> 5 Always Cooperate      Always Cooperate             3        3
#> 6 Conditional Cooperator Always Cooperate            3        3
#> 7 Always Defect         Conditional Cooperator       1        1
#> 8 Always Cooperate      Conditional Cooperator       3        3
#> 9 Conditional Cooperator Conditional Cooperator      3        3
```

```
# Identify program equilibria (no profitable deviation for either
 ↪  player)
cat("\n--- Checking for Program Equilibria ---\n\n")
```

```
#>
#> --- Checking for Program Equilibria ---
```

```
for (i in seq_len(nrow(pe_results))) {
  row <- pe_results[i, ]
  # Can P1 improve by switching?
  p1_alts <- pe_results |> filter(p2 == row$p2)
  p1_best <- max(p1_alts$payoff_1)
  # Can P2 improve by switching?
  p2_alts <- pe_results |> filter(p1 == row$p1)
  p2_best <- max(p2_alts$payoff_2)

  if (row$payoff_1 >= p1_best && row$payoff_2 >= p2_best) {
    cat(sprintf("Program equilibrium: (%s, %s) -> payoffs (%d,
     ↪  %d)\n",
```

```
                row$p1, row$p2, row$payoff_1, row$payoff_2))
  }
}
```

```
#> Program equilibrium: (Always Defect, Always Defect) -> payoffs (1, 1)
#> Program equilibrium: (Conditional Cooperator, Conditional Cooperator) -> payoffs (3
```

```
cat("\nThe (Conditional Cooperator, Conditional Cooperator) pair
 ↪  achieves")
```

```
#>
#> The (Conditional Cooperator, Conditional Cooperator) pair achieves
```

```
cat("\nmutual cooperation with payoffs (3, 3), dominating the
 ↪  Nash")
```

```
#>
#> mutual cooperation with payoffs (3, 3), dominating the Nash
```

```
cat("\nequilibrium of (1, 1) from Always Defect.\n")
```

```
#>
#> equilibrium of (1, 1) from Always Defect.
```

Key insight. The conditional cooperator strategy is only viable when agents can inspect each other's code. Without this transparency, a promise to cooperate is cheap talk. With code inspection, the promise becomes credible because the opponent can verify that cooperation will actually occur. This is the fundamental contribution of program equilibrium to cooperative AI.

41.5 Extensions

- **Provability-based cooperation.** Barasz et al. (2014) study agents with read-access to each other's source code. Their construction permits cooperation without requiring identical programs; it is not a result about limited code access.
- **Multi-agent commitment.** Extending commitment to $n > 2$ agents introduces coalition formation problems. See 9 for the cooperative game theory framework.

- **AI safety applications.** Commitment devices can ensure that AI systems follow safety constraints. This connects to corrigibility (40) and mechanism design (33).
- Dafoe et al. (2020) lay out a comprehensive research agenda for cooperative AI, examining capabilities and institutions needed for cooperation, including communication and commitment.

Exercises

1. **Asymmetric commitment.** Suppose only player 1 can commit (publish code) while player 2 cannot. In the Prisoner's Dilemma, what is player 1's optimal committed strategy? Does one-sided commitment achieve the cooperative outcome?

2. **Chicken with commitment.** Analyze the game of Chicken with program equilibrium. How many program equilibria exist when agents can submit conditional cooperator, always swerve, or always go straight? Is the program equilibrium outcome Pareto-efficient?

3. **Noisy code inspection.** Modify the commitment framework so that each player reads the opponent's code correctly with probability 0.9 and misreads it with probability 0.1. Simulate 1,000 rounds of PD under conditional cooperation with noisy inspection. What cooperation rate emerges, and how does it compare to the perfect-inspection case?

Solutions appear in D.

Chapter 42

The Future of Strategy

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

> Emerging frontiers in strategic AI: computational complexity, multi-agent foundation models, and responsible deployment.

Learning objectives

- Identify the major open problems at the intersection of game theory and AI.
- Understand the computational complexity landscape of game-solving algorithms.
- Recognize the key milestones in the convergence of game theory and artificial intelligence.
- Articulate principles for responsible deployment of strategic AI systems.

42.1 Motivation

This book has covered four decades of ideas — from Nash equilibrium to multi-agent reinforcement learning, from mechanism design to cooperative AI. But the field is accelerating. Foundation models are being deployed in multi-agent settings, AI systems negotiate on behalf of humans, and algorithmic decision-makers interact in markets, infrastructure, and governance.

This capstone chapter surveys the frontier: what has been solved, what remains open, and what practitioners should consider as they build the next generation of strategic AI systems.

42.2 Theory

42.2.1 Computational complexity of game solving

Not all games are equally hard to solve. The computational complexity of finding equilibria varies dramatically with the game's structure.

Complexity Classes for Game Solutions

- **Polynomial (P):** Two-player zero-sum games can be solved via linear programming in polynomial time.
- **PPAD-complete:** Finding a Nash equilibrium in two-player general-sum games is PPAD-complete — believed to be hard but not NP-hard (Daskalakis et al., 2009).
- **NP-hard:** Finding a Nash equilibrium that satisfies additional properties (e.g., maximum social welfare) is NP-hard.
- **EXPTIME:** Solving extensive-form games with imperfect information grows exponentially in the game tree size.

These complexity results have practical implications: for large games, exact equilibrium computation is infeasible, and practitioners must rely on approximation algorithms, learning dynamics, or structural simplifications.

42.2.2 Emerging frontiers

Multi-agent foundation models. Large language models are increasingly deployed in multi-agent settings — negotiation, debate, collaborative problem-solving. The game-theoretic properties of these interactions (equilibrium selection, emergent cooperation, strategic manipulation) are largely unexplored.

AI negotiation. Automated negotiation systems already operate in e-commerce, resource allocation, and diplomacy simulations. A key open question is whether foundation-model negotiators converge to game-theoretically rational strategies or exhibit systematic biases.

AI governance. As AI systems make decisions with societal impact, governance frameworks must account for strategic interactions between AI developers, deployers, regulators, and affected populations. Mechanism design (33) provides the theoretical foundation.

42.2.3 Open problems in MARL

Multi-agent reinforcement learning faces several unsolved challenges:

1. **Non-stationarity.** Each agent's environment is changing because other agents are learning simultaneously.
2. **Credit assignment.** In cooperative settings, attributing team success to individual agents is hard.
3. **Scalability.** The joint action space grows exponentially with the number of agents.
4. **Equilibrium selection.** Learning algorithms may converge to different equilibria depending on initialization.
5. **Safety and alignment.** Ensuring that learned policies satisfy safety constraints in multi-agent settings is an open problem connecting to 40.

42.2.4 Responsible deployment

Deploying strategic AI systems requires attention to:

- **Transparency:** Can stakeholders understand the system's strategy?
- **Fairness:** Does the system treat all participants equitably (39)?
- **Accountability:** Who is responsible when a strategic AI causes harm?
- **Robustness:** Does the system behave well under adversarial or out-of-distribution conditions?

42.3 Implementation in R

42.3.1 Complexity landscape of game-theory problems

```
# Taxonomy of game-theory problems with complexity
 ↪  classifications
complexity_data <- tibble(
  problem = c(
    "2P zero-sum (LP)", "2P zero-sum\n(extensive form)",
    "2P general-sum\n(Nash)", "N-player Nash",
    "Bayesian Nash\n(2P)", "Correlated\nequilibrium",
    "Mechanism\ndesign (optimal)", "Extensive form\n(imperfect
     ↪  info)",
    "Mean-field\nequilibrium", "Cooperative\n(Shapley value)",
    "Social welfare\nmax Nash", "Stackelberg\nequilibrium"
  ),
  game_size = c(2, 3, 4, 7, 5, 3, 8, 6, 9, 5, 5, 4),
  difficulty = c(1, 2.5, 4, 6.5, 5, 1.5, 7.5, 7, 5.5, 3, 8, 3.5),
  complexity_class = c("P", "P", "PPAD", "PPAD", "PPAD",
                       "P", "NP-hard", "EXPTIME", "Open",
                       "P", "NP-hard", "NP-hard"),
```

```
  chapter = c(3, 5, 3, 3, 8, 4, 11, 5, 28, 10, 3, 6)
)

p1 <- ggplot(complexity_data,
             aes(x = game_size, y = difficulty, colour =
              ↪  complexity_class)) +
  geom_point(size = 4) +
  geom_text(aes(label = problem), size = 2.5, vjust = -1.2,
            show.legend = FALSE) +
  scale_colour_manual(
    values = c("P" = okabe_ito[3], "PPAD" = okabe_ito[1],
               "NP-hard" = okabe_ito[6], "EXPTIME" =
                ↪  okabe_ito[7],
               "Open" = okabe_ito[2]),
    name = "Complexity class"
  ) +
  scale_x_continuous(
    name = "Game size (players / information complexity)",
    breaks = 1:10,
    limits = c(0.5, 10.5)
  ) +
  scale_y_continuous(
    name = "Solution difficulty",
    breaks = 0:9,
    limits = c(0, 9.5)
  ) +
  labs(title = "Complexity Landscape of Game-Theory Problems") +
  theme_publication()

p1

save_pub_fig(p1, "future-complexity-landscape", width = 8, height
 ↪  = 6)
```

42.3.2 Timeline of game theory and AI convergence

```
milestones <- tibble(
  year = c(1928, 1944, 1950, 1951, 1953, 1973, 1981, 1992,
           1994, 2006, 2009, 2015, 2017, 2019, 2022, 2024),
  event = c(
    "von Neumann:\nminimax theorem",
```

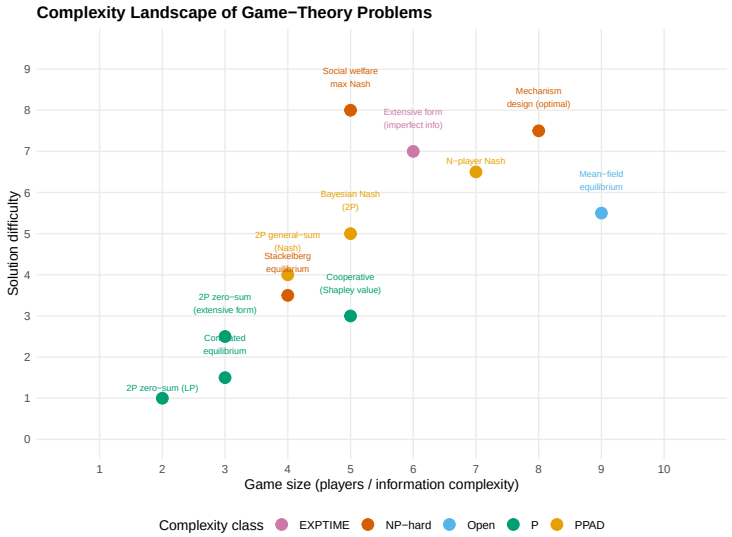


Figure 42.1: Computational complexity landscape of game-theory problems covered in this book. Horizontal axis represents game size (number of players/actions), vertical axis represents solution difficulty. Problems are coloured by complexity class: P (polynomial), PPAD (believed hard), NP-hard, EXPTIME, and open problems.

```
    "von Neumann &\nMorgenstern: TGEB",
    "Nash:\nequilibrium",
    "Shapley:\ncooperative\ngames",
    "Kuhn:\nextensive form",
    "Harsanyi:\nBayesian games",
    "Axelrod:\nIPD tournament",
    "Fudenberg & Tirole:\nGame Theory (textbook)",
    "Nash/Harsanyi/Selten:\nNobel Prize",
    "Daskalakis et al.:\nPPAD-completeness",
    "PPAD complexity\nresult published",
    "Silver et al.:\nAlphaGo",
    "Brown & Sandholm:\nLibratus (poker)",
    "Vinyals et al.:\nAlphaStar",
    "CICERO:\nDiplomacy AI",
    "Multi-agent\nfoundation models"
  ),
  domain = c("Theory", "Theory", "Theory", "Theory", "Theory",
             "Theory", "Theory/AI", "Theory", "Theory",
             "Complexity", "Complexity", "AI", "AI", "AI", "AI",
              ↪  "AI")
)
```

```
# Alternate y positions for readability
milestones <- milestones |>
  mutate(
    y_pos = rep(c(1, -1), length.out = n()) * rep(c(1, 1.5),
     ↪  length.out = n()),
    era = case_when(
      year <= 1960 ~ "Classical foundations",
      year <= 2000 ~ "Maturation",
      year <= 2015 ~ "Computational turn",
      TRUE ~ "AI era"
    )
  )

p2 <- ggplot(milestones) +
  # Timeline axis
  geom_hline(yintercept = 0, colour = "grey40", linewidth = 0.5)
   ↪  +
  # Stems
  geom_segment(aes(x = year, xend = year, y = 0, yend = y_pos *
   ↪  0.7),
               colour = "grey60", linewidth = 0.4) +
  # Points on the timeline
  geom_point(aes(x = year, y = 0, colour = domain), size = 3) +
```

```
  # Event labels
  geom_text(aes(x = year, y = y_pos * 0.85, label = event),
            size = 2.2, lineheight = 0.85) +
  # Year labels
  geom_text(aes(x = year, y = y_pos * 0.05 + ifelse(y_pos > 0,
   ↪  -0.15, 0.15),
                label = year),
            size = 2, colour = "grey30", fontface = "bold") +
  scale_colour_manual(
    values = c("Theory" = okabe_ito[1], "Theory/AI" =
     ↪  okabe_ito[3],
               "Complexity" = okabe_ito[5], "AI" = okabe_ito[2]),
    name = "Domain"
  ) +
  scale_x_continuous(breaks = seq(1930, 2030, 10), limits =
   ↪  c(1925, 2028)) +
  labs(title = "Game Theory and AI: A Convergent History") +
  theme_publication() +
  theme(axis.text.y = element_blank(),
        axis.title = element_blank(),
        axis.ticks.y = element_blank(),
        panel.grid.major.y = element_blank(),
        panel.grid.minor = element_blank()) +
  coord_cartesian(ylim = c(-2, 2))

p2
```

```
save_pub_fig(p2, "future-timeline", width = 9, height = 6)
```

42.4 Worked example

We summarize the taxonomy of games, solution concepts, and computational complexity encountered throughout this book.

```
# Build a taxonomy table
taxonomy <- tibble(
  part = c(rep("Foundations", 4), rep("Dynamic Games", 3),
           rep("Tic. & Info.", 3), rep("Multi-Agent", 3),
           rep("Advanced", 3), rep("Ethics", 3)),
  game_type = c(
    "Normal form (2P)", "Zero-sum", "Potential games",
     ↪  "Cooperative",
```

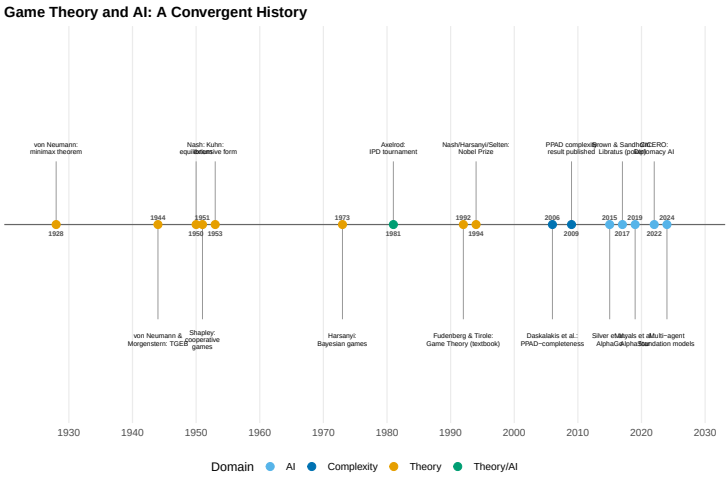


Figure 42.2: Key milestones in the convergence of game theory and AI. Early milestones (left) are foundational theoretical results; recent milestones (right) represent AI systems that achieve or surpass human performance in strategic games.

```
    "Extensive form", "Repeated games", "Stochastic games",
    "Bayesian games", "Mechanism design", "Auctions",
    "MARL", "Evolutionary", "Mean-field",
    "Network games", "Matching markets", "Bargaining",
    "Social welfare", "Fairness games", "Alignment games"
  ),
  solution_concept = c(
    "Nash eq.", "Minimax", "Pure Nash", "Shapley value",
    "SPE / backward ind.", "Folk theorem", "Markov perfect",
    "BNE", "DSIC / BIC", "Dominant strategy",
    "Convergence", "ESS", "MFE",
    "Network Nash", "Stable matching", "Nash bargaining",
    "SWF optimum", "Constrained opt.", "Program eq."
  ),
  complexity = c(
    "PPAD", "P", "P (if exists)", "P",
    "P (perf. info)", "Depends", "NP-hard (general)",
    "PPAD", "NP-hard (optimal)", "P (single-item)",
    "No guarantee", "P (2x2)", "Open",
    "PPAD (general)", "P (Gale-Shapley)", "P (2P)",
    "Varies", "NP-hard", "Open"
  ),
  players = c(
```

```
    "2", "2", "N", "N",
    "2+", "2+", "2+",
    "2+", "N+1", "N+1",
    "N", "Large", "Continuum",
    "N (network)", "2 sides", "2",
    "N", "N + regulator", "2"
  )
)

cat("=== Taxonomy of Games in This Book ===\n\n")
```

```
#> === Taxonomy of Games in This Book ===
```

```
# Display as a gt table
taxonomy |>
  gt(groupname_col = "part") |>
  cols_label(
    game_type = "Game Type",
    solution_concept = "Solution Concept",
    complexity = "Complexity",
    players = "Players"
  ) |>
  tab_header(title = "Taxonomy of Strategic Interactions") |>
  tab_options(table.font.size = px(12))
```

```
# Summary statistics
cat("\n=== Summary Statistics ===\n\n")
```

```
#>
#> === Summary Statistics ===
```

```
cat("Total game types covered:", nrow(taxonomy), "\n")
```

```
#> Total game types covered: 19
```

```
complexity_counts <- taxonomy |>
  count(complexity, sort = TRUE)
cat("\nComplexity distribution:\n")
```

```
#>
#> Complexity distribution:
```

```
print(complexity_counts, n = 20)
```

```
#> # A tibble: 16 x 2
#>    complexity            n
#>    <chr>             <int>
#>  1 Open                  2
#>  2 P                     2
#>  3 PPAD                  2
#>  4 Depends               1
#>  5 NP-hard               1
#>  6 NP-hard (general)     1
#>  7 NP-hard (optimal)     1
#>  8 No guarantee          1
#>  9 P (2P)                1
#> 10 P (2x2)               1
#> 11 P (Gale-Shapley)      1
#> 12 P (if exists)         1
#> 13 P (perf. info)        1
#> 14 P (single-item)       1
#> 15 PPAD (general)        1
#> 16 Varies                1
```

```
cat("\nParts covered:", length(unique(taxonomy$part)), "\n")
```

```
#>
#> Parts covered: 6
```

```
cat("Unique solution concepts:",
 ↪  n_distinct(taxonomy$solution_concept), "\n")
```

```
#> Unique solution concepts: 19
```

Reflection. The taxonomy reveals a striking pattern: the most practically important games are often the hardest to solve. Two-player zero-sum games (poker, competitive pricing) have polynomial-time solutions, but general-sum games with incomplete information — which describe most real-world strategic interactions — are computationally intractable in the worst case.

This complexity gap explains why heuristic methods (MARL, evolutionary dynamics, learned strategies) dominate in practice, and why the theoretical guarantees from classical game theory serve primarily as benchmarks and design principles rather than as implementable algorithms for large-scale systems.

42.5 Extensions

- **Multi-agent foundation models.** The deployment of large language models in multi-agent settings creates new game-theoretic questions about emergent strategy, manipulation, and alignment. See Meta Fundamental AI Research Diplomacy Team (FAIR) et al. (2022) on CICERO for Diplomacy.
- **Algorithmic game theory.** Nisan et al. (2007) provide a comprehensive treatment of computational aspects of game theory, including complexity results and approximation algorithms.
- **AI safety and governance.** The intersection of game theory and AI safety is surveyed by Dafoe et al. (2020). Responsible deployment requires combining the technical tools from this book with institutional and regulatory frameworks.
- **Mechanism design for AI.** As AI systems participate in markets and governance, designing mechanisms that remain incentive-compatible when some participants are AI agents is a frontier research area.

Exercises

1. **Complexity classification.** For each of the following, state whether finding a Nash equilibrium is in P, PPAD-complete, or NP-hard: (a) a 2-player zero-sum game, (b) a 3-player general-sum game, (c) a 2-player general-sum game where we require the equilibrium to maximize social welfare. Briefly justify each answer.

2. **Research survey.** Choose one of the "Open" problems in the complexity landscape (e.g., mean-field equilibrium computation) and write a one-paragraph summary of the current state of knowledge, citing at least two references.

3. **Book reflection.** Pick any two chapters from different parts of this book. Describe a real-world scenario where both solution concepts would apply simultaneously (e.g., a repeated Bayesian game on a network). What additional challenges arise from combining the two frameworks?

Solutions appear in D.

Taxonomy of Strategic Interactions

Game Type	Solution Concept	Complexity	Players
Foundations			
Normal form (2P)	Nash eq.	PPAD	2
Zero-sum	Minimax	P	2
Potential games	Pure Nash	P (if exists)	N
Cooperative	Shapley value	P	N
Dynamic Games			
Extensive form	SPE / backward ind.	P (perf. info)	2+
Repeated games	Folk theorem	Depends	2+
Stochastic games	Markov perfect	NP-hard (general)	2+
Tic. & Info.			
Bayesian games	BNE	PPAD	2+
Mechanism design	DSIC / BIC	NP-hard (optimal)	N+1
Auctions	Dominant strategy	P (single-item)	N+1
Multi-Agent			
MARL	Convergence	No guarantee	N
Evolutionary	ESS	P (2x2)	Large
Mean-field	MFE	Open	Continuum
Advanced			
Network games	Network Nash	PPAD (general)	N (network)
Matching markets	Stable matching	P (Gale-Shapley)	2 sides
Bargaining	Nash bargaining	P (2P)	2
Ethics			
Social welfare	SWF optimum	Varies	N
Fairness games	Constrained opt.	NP-hard	N + regulator
Alignment games	Program eq.	Open	2

Appendix A

R Refresher

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

> A concise refresher on R fundamentals for readers who need a quick review.

This appendix provides a quick reference for the R features used most often in this book. It is not a substitute for a full R tutorial but should be enough to remind you of the essentials. For a deeper treatment, see Wickham et al. (2023).

A.1 Data structures

A.1.1 Vectors

Vectors are the fundamental data type in R. Every element must share the same type (logical, integer, double, character).

```
# Numeric vector
payoffs <- c(3, 0, 5, 1)

# Character vector
strategies <- c("Cooperate", "Defect")

# Logical vector
is_dominated <- c(FALSE, TRUE, FALSE)

# Sequence shortcuts
```

```
indices <- 1:10
grid <- seq(0, 1, by = 0.1)

# Vectorised arithmetic
payoffs * 2
```

```
#> [1]  6  0 10  2
```

```
payoffs > 2
```

```
#> [1]  TRUE FALSE  TRUE FALSE
```

A.1.2 Matrices

Matrices are two-dimensional vectors. They are central to payoff representations throughout this book.

```
# Create a 2x2 payoff matrix (filled by column by default)
A <- matrix(c(3, 5, 0, 1), nrow = 2, byrow = TRUE)
rownames(A) <- c("Cooperate", "Defect")
colnames(A) <- c("Cooperate", "Defect")
A
```

```
#>           Cooperate Defect
#> Cooperate         3      5
#> Defect            0      1
```

```
# Matrix indexing
A[1, 2]       # row 1, column 2
```

```
#> [1] 5
```

```
A["Defect", ] # entire row by name
```

```
#> Cooperate    Defect
#>         0         1
```

```
# Dimensions
dim(A)
```

```
#> [1] 2 2
```

```
nrow(A)
```

```
#> [1] 2
```

```
ncol(A)
```

```
#> [1] 2
```

A.1.3 Lists

Lists can hold elements of different types and different lengths. They are used throughout the book to bundle game components together.

```
game <- list(
  players  = c("Player 1", "Player 2"),
  payoff_1 = matrix(c(3, 5, 0, 1), nrow = 2, byrow = TRUE),
  payoff_2 = matrix(c(3, 0, 5, 1), nrow = 2, byrow = TRUE)
)

# Access by name
game$players
```

```
#> [1] "Player 1" "Player 2"
```

```
game[["payoff_1"]]
```

```
#>      [,1] [,2]
#> [1,]    3    5
#> [2,]    0    1
```

```
# Access by position
game[[1]]
```

```
#> [1] "Player 1" "Player 2"
```

A.1.4 Data frames and tibbles

Data frames (and their tidyverse counterpart, tibbles) are the primary structure for tabular data.

```
# Base R data frame
df <- data.frame(
  strategy = c("Cooperate", "Defect"),
  payoff   = c(3, 1)
)

# Tidyverse tibble
tbl <- tibble(
  strategy = c("Cooperate", "Defect"),
  payoff   = c(3, 1)
)

# Tibbles print more informatively
tbl
```

```
#> # A tibble: 2 x 2
#>   strategy  payoff
#>   <chr>      <dbl>
#> 1 Cooperate      3
#> 2 Defect         1
```

A.2 Control flow

A.2.1 Conditionals

```
x <- 5

if (x > 3) {
  result <- "high"
} else if (x > 1) {
  result <- "medium"
} else {
  result <- "low"
}
result
```

```
#> [1] "high"
```

```r
# Vectorised conditional
ifelse(c(3, 0, 5, 1) > 2, "above", "below")
```

```
#> [1] "above" "below" "above" "below"
```

A.2.2 Loops

```r
# for loop
total <- 0
for (i in 1:5) {
  total <- total + i
}
total
```

```
#> [1] 15
```

```r
# while loop
count <- 0
value <- 1
while (value < 100) {
  value <- value * 2
  count <- count + 1
}
count
```

```
#> [1] 7
```

A.2.3 The apply family

The `apply` functions replace explicit loops with concise functional calls.

```r
A <- matrix(1:12, nrow = 3)

# Apply across rows (margin = 1)
apply(A, 1, sum)
```

```
#> [1] 22 26 30
```

```r
# Apply across columns (margin = 2)
apply(A, 2, mean)
```

```
#> [1]  2  5  8 11
```

```r
# sapply returns a vector; lapply returns a list
sapply(1:5, function(x) x^2)
```

```
#> [1]  1  4  9 16 25
```

```r
lapply(1:3, function(x) rep(x, x))
```

```
#> [[1]]
#> [1] 1
#>
#> [[2]]
#> [1] 2 2
#>
#> [[3]]
#> [1] 3 3 3
```

```r
# vapply is like sapply but with a specified return type
vapply(1:5, function(x) x^2, numeric(1))
```

```
#> [1]  1  4  9 16 25
```

A.3 Functions

A.3.1 Defining functions

```r
# A function that computes expected payoff
expected_payoff <- function(payoff_vec, prob_vec) {
  sum(payoff_vec * prob_vec)
}

expected_payoff(c(3, 0), c(0.6, 0.4))
```

```
#> [1] 1.8
```

```r
# Default arguments
greet <- function(name, greeting = "Hello") {
  paste(greeting, name)
}
greet("Alice")
```

```
#> [1] "Hello Alice"
```

```r
greet("Bob", greeting = "Hi")
```

```
#> [1] "Hi Bob"
```

A.3.2 Closures and environments

A closure is a function that captures variables from its enclosing environment. This pattern is useful for creating parameterised game constructors.

```r
make_discounter <- function(delta) {
  # delta is captured in the closure
  function(payoffs) {
    n <- length(payoffs)
    weights <- delta^(seq_len(n) - 1)
    sum(payoffs * weights)
  }
}

discount_95 <- make_discounter(0.95)
discount_95(c(3, 3, 3, 3, 3))
```

```
#> [1] 13.6
```

```r
# The enclosed value of delta persists
discount_50 <- make_discounter(0.50)
discount_50(c(3, 3, 3, 3, 3))
```

```
#> [1] 5.81
```

A.3.3 Anonymous functions

R 4.1+ supports the shorthand `\(x)` syntax for anonymous (lambda) functions.

```
sapply(1:5, \(x) x^2)
```

```
#> [1]  1  4  9 16 25
```

```
# Equivalent to the older syntax:
sapply(1:5, function(x) x^2)
```

```
#> [1]  1  4  9 16 25
```

A.4 Tidyverse basics

The tidyverse is a collection of packages for data wrangling and visualisation. This book uses it extensively.

A.4.1 The pipe operator

The pipe `|>` (base R 4.1+) or `%>%` (magrittr) passes the left-hand side as the first argument to the right-hand side.

```
c(4, 1, 7, 2, 9) |>
  sort() |>
  rev() |>
  head(3)
```

```
#> [1] 9 7 4
```

A.4.2 dplyr verbs

The five core verbs handle the vast majority of data manipulation tasks.

```
game_results <- tibble(
  round    = 1:6,
  player_1 = c("C", "D", "C", "C", "D", "D"),
  player_2 = c("C", "C", "D", "C", "D", "C"),
  payoff_1 = c(3, 5, 0, 3, 1, 5),
  payoff_2 = c(3, 0, 5, 3, 1, 0)
)

# filter: keep rows matching a condition
game_results |> filter(player_1 == "C")
```

```
#> # A tibble: 3 x 5
#>   round player_1 player_2 payoff_1 payoff_2
#>   <int> <chr>    <chr>       <dbl>    <dbl>
#> 1     1 C        C               3        3
#> 2     3 C        D               0        5
#> 3     4 C        C               3        3
```

```
# mutate: create or modify columns
game_results |> mutate(total = payoff_1 + payoff_2)
```

```
#> # A tibble: 6 x 6
#>   round player_1 player_2 payoff_1 payoff_2 total
#>   <int> <chr>    <chr>       <dbl>    <dbl> <dbl>
#> 1     1 C        C               3        3     6
#> 2     2 D        C               5        0     5
#> 3     3 C        D               0        5     5
#> 4     4 C        C               3        3     6
#> 5     5 D        D               1        1     2
#> 6     6 D        C               5        0     5
```

```
# select: choose columns
game_results |> select(round, payoff_1, payoff_2)
```

```
#> # A tibble: 6 x 3
#>   round payoff_1 payoff_2
#>   <int>    <dbl>    <dbl>
#> 1     1        3        3
#> 2     2        5        0
#> 3     3        0        5
#> 4     4        3        3
#> 5     5        1        1
#> 6     6        5        0
```

```
# arrange: reorder rows
game_results |> arrange(desc(payoff_1))
```

```
#> # A tibble: 6 x 5
#>   round player_1 player_2 payoff_1 payoff_2
#>   <int> <chr>    <chr>       <dbl>    <dbl>
#> 1     2 D        C               5        0
#> 2     6 D        C               5        0
#> 3     1 C        C               3        3
#> 4     4 C        C               3        3
```

```
#> 5     5 D        D                1        1
#> 6     3 C        D                0        5
```

```
# summarise (often with group_by)
game_results |>
  group_by(player_1) |>
  summarise(
    mean_payoff = mean(payoff_1),
    n_rounds    = n()
  )
```

```
#> # A tibble: 2 x 3
#>   player_1 mean_payoff n_rounds
#>   <chr>          <dbl>    <int>
#> 1 C               2           3
#> 2 D               3.67        3
```

A.4.3 Pivoting

Converting between wide and long formats is essential for plotting.

```
# Wide to long
game_long <- game_results |>
  pivot_longer(
    cols      = starts_with("payoff"),
    names_to  = "player",
    values_to = "payoff"
  )
head(game_long)
```

```
#> # A tibble: 6 x 5
#>   round player_1 player_2 player   payoff
#>   <int> <chr>    <chr>    <chr>     <dbl>
#> 1     1 C        C        payoff_1      3
#> 2     1 C        C        payoff_2      3
#> 3     2 D        C        payoff_1      5
#> 4     2 D        C        payoff_2      0
#> 5     3 C        D        payoff_1      0
#> 6     3 C        D        payoff_2      5
```

```
# Long to wide
game_long |>
  pivot_wider(names_from = player, values_from = payoff)
```

```
#> # A tibble: 6 x 5
#>   round player_1 player_2 payoff_1 payoff_2
#>   <int> <chr>    <chr>       <dbl>    <dbl>
#> 1     1 C        C               3        3
#> 2     2 D        C               5        0
#> 3     3 C        D               0        5
#> 4     4 C        C               3        3
#> 5     5 D        D               1        1
#> 6     6 D        C               5        0
```

A.4.4 ggplot2 layers

The grammar of graphics builds plots layer by layer: data, aesthetics, geometries, and scales.

```
game_results |>
  pivot_longer(
    cols = starts_with("payoff"),
    names_to = "player",
    values_to = "payoff"
  ) |>
  ggplot(aes(x = round, y = payoff, colour = player)) +
  geom_line(linewidth = 0.8) +
  geom_point(size = 2) +
  scale_colour_manual(
    values = c(payoff_1 = "#1b9e77", payoff_2 = "#d95f02"),
    labels = c("Player 1", "Player 2")
  ) +
  labs(x = "Round", y = "Payoff", colour = NULL) +
  theme_minimal()
```

A.5 Useful idioms for this book

A handful of patterns recur throughout the chapters.

```
# Replicate a simulation many times
results <- replicate(1000, {
  sample(c("Heads", "Tails"), 1)
})
table(results)
```

```
#> results
```

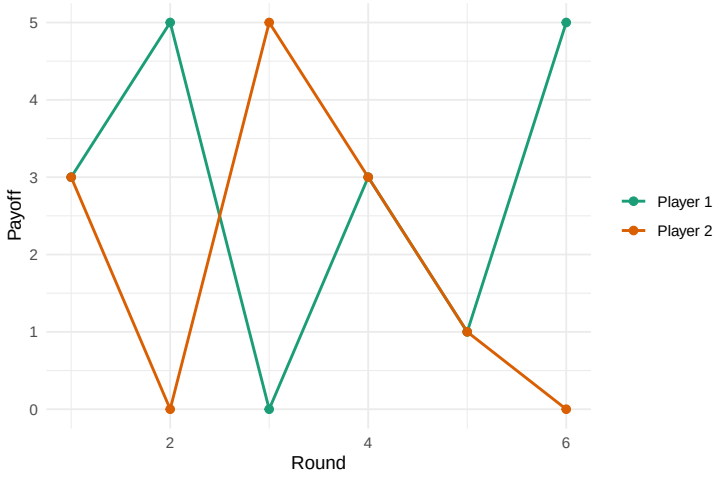


Figure A.1: Payoffs across six rounds of a repeated game.

```
#> Heads Tails
#>   499   501
```

```
# Outer product for payoff grids
p <- seq(0, 1, by = 0.25)
q <- seq(0, 1, by = 0.25)
payoff_grid <- outer(p, q, function(pi, qi) 3 * pi * qi + 1 * (1
 ↪  - pi))
payoff_grid
```

```
#>      [,1]  [,2] [,3] [,4] [,5]
#> [1,] 1.00 1.000 1.00 1.00  1.0
#> [2,] 0.75 0.938 1.12 1.31  1.5
#> [3,] 0.50 0.875 1.25 1.62  2.0
#> [4,] 0.25 0.812 1.38 1.94  2.5
#> [5,] 0.00 0.750 1.50 2.25  3.0
```

```
# Named vectors for readable code
strategy_names <- c(C = "Cooperate", D = "Defect")
strategy_names["C"]
```

```
#>           C
#> "Cooperate"
```

```r
# Setting seeds for reproducibility
set.seed(42)
sample(1:100, 5)
```

```
#> [1] 49 65 25 74 18
```

Appendix B

Linear Algebra Essentials

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

> Vectors, matrices, eigenvalues, and the linear algebra used throughout this book.

This appendix reviews the linear algebra that appears most often in game theory. The focus is on concepts rather than proofs, with R implementations alongside each topic. For a thorough mathematical treatment, see Strang (2023).

B.1 Vectors and vector operations

A vector in $\mathbb{R}^n$ is an ordered list of n real numbers. In game theory, vectors represent strategy profiles, probability distributions over actions, and payoff allocations.

```
# Define vectors
x <- c(3, 1, 4)
y <- c(2, 7, 1)

# Scalar multiplication and addition
2 * x
```

```
#> [1] 6 2 8
```

```
x + y
```

```
#> [1] 5 8 5
```

```
# Dot (inner) product
sum(x * y)
```

```
#> [1] 17
```

```
# Or equivalently:
x %*% y
```

```
#>      [,1]
#> [1,]   17
```

```
# Euclidean norm (length)
sqrt(sum(x^2))
```

```
#> [1] 5.1
```

The **dot product** $\mathbf{x} \cdot \mathbf{y} = \sum_i x_i y_i$ appears whenever we compute expected payoffs: if $\mathbf{p}$ is a probability vector and $\mathbf{u}$ is a payoff vector, then $\mathbf{p} \cdot \mathbf{u}$ is the expected payoff.

B.2 Matrices

A matrix $A \in \mathbb{R}^{m \times n}$ is a rectangular array of numbers. In game theory, the primary use is the payoff matrix of a normal-form game.

B.2.1 Matrix arithmetic in R

```
A <- matrix(c(3, 0, 5, 1), nrow = 2, byrow = TRUE)
B <- matrix(c(3, 5, 0, 1), nrow = 2, byrow = TRUE)

# Element-wise operations
A + B
```

```
#>      [,1] [,2]
#> [1,]    6    5
#> [2,]    5    2
```

```
A * B  # Hadamard (element-wise) product
```

```
#>      [,1] [,2]
#> [1,]    9    0
#> [2,]    0    1
```

```
# Matrix multiplication
A %*% B
```

```
#>      [,1] [,2]
#> [1,]    9   15
#> [2,]   15   26
```

```
# Transpose
t(A)
```

```
#>      [,1] [,2]
#> [1,]    3    5
#> [2,]    0    1
```

B.2.2 Key matrix operations

```
M <- matrix(c(4, 2, 1, 3), nrow = 2, byrow = TRUE)

# Determinant
det(M)
```

```
#> [1] 10
```

```
# Inverse (only if det != 0)
solve(M)
```

```
#>      [,1] [,2]
#> [1,]  0.3 -0.2
#> [2,] -0.1  0.4
```

```
# Verify: M %*% M^{-1} = I
M %*% solve(M)
```

```
#>                         [,1] [,2]
#> [1,]  1.0000000000000000000    0
#> [2,] -0.0000000000000000278    1
```

B.3 Systems of linear equations

Many game-theoretic computations reduce to solving a system $A\mathbf{x} = \mathbf{b}$. For example, finding a mixed Nash equilibrium in a 2x2 game requires solving the indifference equations (see 5).

B.3.1 Solving $A\mathbf{x} = \mathbf{b}$

```
# System: 2x + y = 5, x + 3y = 7
A <- matrix(c(2, 1, 1, 3), nrow = 2, byrow = TRUE)
b <- c(5, 7)

x <- solve(A, b)
x
```

```
#> [1] 1.6 1.8
```

```
# Verify
A %*% x
```

```
#>      [,1]
#> [1,]    5
#> [2,]    7
```

B.3.2 Application: mixed Nash equilibrium

In a 2x2 game, finding the mixed Nash equilibrium reduces to making each player indifferent between their pure strategies. If Player 2 mixes with probability q on the first action, Player 1's indifference condition is:

$$a_{11}q + a_{12}(1-q) = a_{21}q + a_{22}(1-q)$$

This is a single linear equation in q. For larger games, mixed equilibria require solving systems of such equations simultaneously.

```
# Prisoner's Dilemma payoffs for Player 1
# Row player payoffs: A[i,j]
A <- matrix(c(3, 0, 5, 1), nrow = 2, byrow = TRUE)

# Indifference: A[1,1]*q + A[1,2]*(1-q) = A[2,1]*q + A[2,2]*(1-q)
```

```
# Rearranging: q*(A[1,1] - A[1,2] - A[2,1] + A[2,2]) = A[2,2] -
 ↪  A[1,2]
denom <- A[1, 1] - A[1, 2] - A[2, 1] + A[2, 2]
q_star <- (A[2, 2] - A[1, 2]) / denom
q_star
```

```
#> [1] -1
```

B.4 Eigenvalues and eigenvectors

An eigenvector $\mathbf{v}$ of a square matrix A satisfies $A\mathbf{v} = \lambda\mathbf{v}$ for some scalar λ (the eigenvalue). Eigenvalues appear in several places in this book:

- **Replicator dynamics** (20): the stability of fixed points is determined by the eigenvalues of the Jacobian matrix.
- **Markov chains**: the stationary distribution is the eigenvector corresponding to eigenvalue 1.

```
M <- matrix(c(4, 1, 2, 3), nrow = 2, byrow = TRUE)

eig <- eigen(M)
eig$values
```

```
#> [1] 5 2
```

```
eig$vectors
```

```
#>       [,1]   [,2]
#> [1,] 0.707 -0.447
#> [2,] 0.707  0.894
```

```
# Verify: M v = lambda v
lambda_1 <- eig$values[1]
v_1 <- eig$vectors[, 1]
M %*% v_1
```

```
#>      [,1]
#> [1,] 3.54
#> [2,] 3.54
```

```
lambda_1 * v_1
```

```
#> [1] 3.54 3.54
```

B.4.1 Stability and the Jacobian

For a dynamical system $\dot{\mathbf{x}} = f(\mathbf{x})$, a fixed point $\mathbf{x}^*$ is **stable** if all eigenvalues of the Jacobian matrix $J = \partial f / \partial \mathbf{x}$ evaluated at $\mathbf{x}^*$ have negative real parts.

```
# Example: Jacobian at a fixed point of a 2D system
J <- matrix(c(-2, 1, 0, -3), nrow = 2, byrow = TRUE)

eig_J <- eigen(J)
eig_J$values
```

```
#> [1] -3 -2
```

```
# Both eigenvalues are negative => stable fixed point
all(Re(eig_J$values) < 0)
```

```
#> [1] TRUE
```

B.5 Positive definite matrices

A symmetric matrix A is **positive definite** if $\mathbf{x}^T A \mathbf{x} > 0$ for all nonzero $\mathbf{x}$. Equivalently, all eigenvalues are positive. Positive definiteness arises in quadratic payoff functions and in verifying second-order conditions for optimisation problems.

```
A <- matrix(c(4, 1, 1, 3), nrow = 2, byrow = TRUE)

# Check via eigenvalues
eigen(A)$values
```

```
#> [1] 4.62 2.38
```

```
all(eigen(A)$values > 0)
```

```
#> [1] TRUE
```

```
# Check via Cholesky decomposition (succeeds only for positive
 ↪  definite matrices)
chol(A)
```

```
#>      [,1] [,2]
#> [1,]    2 0.50
#> [2,]    0 1.66
```

B.6 Convex sets and the simplex

B.6.1 The probability simplex

A **mixed strategy** is a probability distribution over pure strategies. The set of all such distributions forms the **(probability) simplex**:

$$\Delta_n = \left\{ \mathbf{p} \in \mathbb{R}^n \;\middle|\; p_i \geq 0,\ \sum_{i=1}^{n} p_i = 1 \right\}$$

For $n = 2$, the simplex is a line segment; for $n = 3$, it is a triangle.

```
# Check whether a vector is a valid mixed strategy
is_mixed_strategy <- function(p, tol = 1e-10) {
  all(p >= -tol) && abs(sum(p) - 1) < tol
}

is_mixed_strategy(c(0.3, 0.5, 0.2))
```

```
#> [1] TRUE
```

```
is_mixed_strategy(c(0.5, 0.6, -0.1))
```

```
#> [1] FALSE
```

B.6.2 Convex sets and convex combinations

A set C is **convex** if for any two points $\mathbf{x}, \mathbf{y} \in C$ and any $\lambda \in [0, 1]$, the point $\lambda\mathbf{x} + (1 - \lambda)\mathbf{y}$ is also in C. Important convex sets in game theory include:

- The simplex Δ_n (the set of mixed strategies).

- The **core** of a cooperative game (see 9).
- The **feasible payoff region** of a repeated game (see 8).

```
# Convex combination of two strategy vectors
p1 <- c(0.7, 0.2, 0.1)
p2 <- c(0.1, 0.5, 0.4)
lambda <- 0.6

p_mix <- lambda * p1 + (1 - lambda) * p2
p_mix
```

```
#> [1] 0.46 0.32 0.22
```

```
is_mixed_strategy(p_mix)
```

```
#> [1] TRUE
```

B.6.3 The support of a mixed strategy

The **support** of a mixed strategy $\mathbf{p}$ is the set of pure strategies that receive positive probability: $\text{supp}(\mathbf{p}) = \{i : p_i > 0\}$. In a Nash equilibrium, every pure strategy in the support must yield the same expected payoff (the indifference principle from 5).

```
support <- function(p, tol = 1e-10) {
  which(p > tol)
}

support(c(0.5, 0, 0.5, 0))
```

```
#> [1] 1 3
```

```
support(c(0.25, 0.25, 0.25, 0.25))
```

```
#> [1] 1 2 3 4
```

B.7 Matrix decompositions

B.7.1 LU decomposition

R does not expose LU decomposition directly in base, but `solve()` uses it internally. For most game-theoretic applications, `solve()` suffices.

B.7.2 Singular value decomposition (SVD)

The SVD decomposes any $m \times n$ matrix as $A = U\Sigma V^T$. It is useful for low-rank approximations and for diagnosing near-singular payoff matrices.

```
A <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, byrow = TRUE)

s <- svd(A)
s$d  # singular values
```

```
#> [1] 9.508 0.773
```

```
# Reconstruct: U %*% diag(d) %*% t(V) = A
s$u %*% diag(s$d) %*% t(s$v)
```

```
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    4    5    6
```

B.8 Summary of R linear algebra functions

Table B.1: R functions for linear algebra

Task	R function	Example
Matrix multiply	`%*%`	`A %*% B`
Transpose	`t()`	`t(A)`
Determinant	`det()`	`det(A)`
Inverse	`solve()`	`solve(A)`
Solve $Ax = b$	`solve()`	`solve(A, b)`
Eigenvalues	`eigen()`	`eigen(A)$values`
SVD	`svd()`	`svd(A)$d`
Cholesky	`chol()`	`chol(A)`
Cross product	`crossprod()`	`crossprod(A)` gives A^TA
Outer product	`outer()` or `%o%`	`x %o% y`

Appendix C

Probability Essentials

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

> Probability distributions, expectations, Bayes' rule, and stochastic processes.

This appendix reviews the probability concepts that underpin mixed strategies, Bayesian games, and the simulation chapters. The emphasis is on intuition and R implementation rather than measure-theoretic rigour.

C.1 Sample spaces and events

A **sample space** Ω is the set of all possible outcomes of a random experiment. An **event** $E \subseteq \Omega$ is a subset of outcomes. A **probability measure** P assigns a number $P(E) \in [0, 1]$ to every event, with $P(\Omega) = 1$.

In game theory, the sample space often corresponds to the set of action profiles or the set of player types.

C.2 Discrete distributions

A **discrete random variable** X takes values in a countable set $\{x_1, x_2, \ldots\}$ with probability mass function $p(x_i) = P(X = x_i)$.

C.2.1 Common discrete distributions

```
# Bernoulli: coin flip (used in mixed strategies)
rbinom(10, size = 1, prob = 0.5)
```

```
#>  [1] 1 1 0 1 1 1 1 0 1 1
```

```
# Binomial: number of cooperators in n rounds
dbinom(3, size = 10, prob = 0.6)  # P(X = 3)
```

```
#> [1] 0.0425
```

```
# Uniform: equally likely strategies
sample(c("Rock", "Paper", "Scissors"), size = 1)
```

```
#> [1] "Scissors"
```

```
# Poisson: arrival counts (used in some mechanism design models)
rpois(10, lambda = 3)
```

```
#>  [1] 4 6 2 3 6 7 1 3 3 5
```

C.2.2 Probability mass function and CDF

```
# PMF and CDF of Binomial(10, 0.4)
x <- 0:10
pmf <- dbinom(x, size = 10, prob = 0.4)
cdf <- pbinom(x, size = 10, prob = 0.4)

tibble(x = x, pmf = pmf, cdf = cdf) |>
  head(6)
```

```
#> # A tibble: 6 x 3
#>       x     pmf     cdf
#>   <int>   <dbl>   <dbl>
#> 1     0 0.00605 0.00605
#> 2     1 0.0403  0.0464
#> 3     2 0.121   0.167
#> 4     3 0.215   0.382
#> 5     4 0.251   0.633
#> 6     5 0.201   0.834
```

C.3 Continuous distributions

A **continuous random variable** X has a probability density function $f(x)$ such that $P(a \leq X \leq b) = \int_a^b f(x)\,dx$.

C.3.1 Common continuous distributions

```
# Uniform on [0, 1]: used for types in Bayesian games
runif(5, min = 0, max = 1)
```

```
#> [1] 0.1387 0.9889 0.9467 0.0824 0.5142
```

```
# Normal: payoff noise, error terms
rnorm(5, mean = 0, sd = 1)
```

```
#> [1] -0.279 -0.133  0.636 -0.284 -2.656
```

```
# Exponential: waiting times, discount factors
rexp(5, rate = 1)
```

```
#> [1] 4.996 0.224 1.211 0.719 1.308
```

C.3.2 Density, CDF, and quantiles

R uses a consistent naming convention: `d` for density, `p` for CDF, `q` for quantile, and `r` for random draws.

```
# Normal distribution
dnorm(0, mean = 0, sd = 1)   # density at 0
```

```
#> [1] 0.399
```

```
pnorm(1.96, mean = 0, sd = 1) # P(X <= 1.96)
```

```
#> [1] 0.975
```

```r
qnorm(0.975, mean = 0, sd = 1) # 97.5th percentile
```

```
#> [1] 1.96
```

```r
rnorm(3, mean = 0, sd = 1)    # three random draws
```

```
#> [1]  0.358  0.302 -0.394
```

C.4 Expected value and variance

The **expected value** of a discrete random variable is $E[X] = \sum_i x_i \, p(x_i)$. For a continuous variable, $E[X] = \int x \, f(x) \, dx$. In game theory, the expected payoff under a mixed strategy $\mathbf{p}$ is:

$$E[u_i] = \sum_{a \in A} p(a) \, u_i(a)$$

The **variance** $\text{Var}(X) = E[(X - E[X])^2] = E[X^2] - (E[X])^2$ measures the spread of outcomes.

```r
# Expected payoff under a mixed strategy
payoffs <- c(3, 0, 5, 1)
probs   <- c(0.3, 0.2, 0.1, 0.4)

# Expected value
ev <- sum(payoffs * probs)
ev
```

```
#> [1] 1.8
```

```r
# Variance
variance <- sum(probs * (payoffs - ev)^2)
variance
```

```
#> [1] 2.36
```

```r
# Using simulation to verify
simulated <- sample(payoffs, size = 100000, replace = TRUE, prob
 ↪  = probs)
mean(simulated)
```

```
#> [1] 1.8
```

```
var(simulated)
```

```
#> [1] 2.37
```

C.4.1 Linearity of expectation

A key property: $E[aX + bY] = aE[X] + bE[Y]$, regardless of whether X and Y are independent. This is used constantly when computing expected payoffs in multi-player games.

C.5 Conditional probability

The **conditional probability** of A given B is:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}, \quad P(B) > 0$$

Two events are **independent** if $P(A \cap B) = P(A)P(B)$, equivalently $P(A \mid B) = P(A)$.

```
# Simulation: P(both cooperate | Player 1 cooperates)
n <- 100000
p1 <- sample(c("C", "D"), n, replace = TRUE, prob = c(0.6, 0.4))
p2 <- sample(c("C", "D"), n, replace = TRUE, prob = c(0.5, 0.5))

# Joint and conditional (assuming independence here)
p_both_C <- mean(p1 == "C" & p2 == "C")
p_p1_C   <- mean(p1 == "C")
p_both_C / p_p1_C  # should be approx 0.5
```

```
#> [1] 0.501
```

C.6 Bayes' theorem

Bayes' theorem is the foundation of Bayesian games (7). It tells us how to update beliefs about a player's type after observing their action:

$$P(\theta \mid a) = \frac{P(a \mid \theta)\, P(\theta)}{P(a)}$$

where θ is the player's type, a is the observed action, $P(\theta)$ is the prior, $P(a \mid \theta)$ is the likelihood, and $P(a) = \sum_{\theta'} P(a \mid \theta') P(\theta')$ is the marginal likelihood.

```
# Signaling game: worker chooses Education or No Education
# Types: High (prob 0.6), Low (prob 0.4)
# P(Education | High) = 0.9, P(Education | Low) = 0.2

prior_high <- 0.6
prior_low  <- 0.4

p_edu_given_high <- 0.9
p_edu_given_low  <- 0.2

# P(Education)
p_edu <- p_edu_given_high * prior_high + p_edu_given_low *
 ↪  prior_low
p_edu
```

```
#> [1] 0.62
```

```
# P(High | Education) by Bayes' theorem
p_high_given_edu <- (p_edu_given_high * prior_high) / p_edu
p_high_given_edu
```

```
#> [1] 0.871
```

```
# P(High | No Education)
p_no_edu <- 1 - p_edu
p_high_given_no_edu <- ((1 - p_edu_given_high) * prior_high) /
 ↪  p_no_edu
p_high_given_no_edu
```

```
#> [1] 0.158
```

C.6.1 Sequential updating

When multiple signals are observed, Bayes' theorem can be applied sequentially: the posterior from one update becomes the prior for the next. This is used in repeated Bayesian games where players learn about opponents over time.

```
# Sequential Bayesian updating: observe actions round by round
# Prior: P(Cooperator type) = 0.5
# P(C action | Cooperator type) = 0.9
# P(C action | Defector type) = 0.1

update_belief <- function(prior, likelihood_coop,
 ↪  likelihood_defect, action) {
  if (action == "C") {
    l_coop <- likelihood_coop
    l_defect <- likelihood_defect
  } else {
    l_coop <- 1 - likelihood_coop
    l_defect <- 1 - likelihood_defect
  }
  numerator   <- l_coop * prior
  denominator <- l_coop * prior + l_defect * (1 - prior)
  numerator / denominator
}

prior <- 0.5
actions <- c("C", "C", "D", "C", "C")

beliefs <- numeric(length(actions))
for (i in seq_along(actions)) {
  prior <- update_belief(prior, 0.9, 0.1, actions[i])
  beliefs[i] <- prior
}

tibble(round = seq_along(actions), action = actions, belief =
 ↪  beliefs)
```

```
#> # A tibble: 5 x 3
#>   round action belief
#>   <int> <chr>   <dbl>
#> 1     1 C       0.9
#> 2     2 C       0.988
#> 3     3 D       0.9
#> 4     4 C       0.988
#> 5     5 C       0.999
```

C.7 Law of large numbers and Monte Carlo

The **law of large numbers** guarantees that the sample mean converges to the true mean as the sample size grows. This justifies the Monte Carlo methods

used extensively in 16.

```
# Demonstrate convergence of sample mean
n_values <- c(10, 100, 1000, 10000, 100000)
true_mean <- 3.5  # E[Uniform die roll]

estimates <- vapply(n_values, function(n) {
  mean(sample(1:6, n, replace = TRUE))
}, numeric(1))

tibble(n = n_values, estimate = estimates, true_mean = true_mean)
```

```
#> # A tibble: 5 x 3
#>        n estimate true_mean
#>    <dbl>    <dbl>     <dbl>
#> 1     10     3.2         3.5
#> 2    100     3.58        3.5
#> 3   1000     3.48        3.5
#> 4  10000     3.49        3.5
#> 5 100000     3.50        3.5
```

C.7.1 Monte Carlo estimation

To estimate a quantity $\theta = E[g(X)]$, draw N independent samples $X_1, \ldots, X_N$ and compute:

$$\hat{\theta}_N = \frac{1}{N} \sum_{i=1}^{N} g(X_i)$$

The standard error of this estimate is $\text{SE} = \sigma / \sqrt{N}$, where σ is the standard deviation of $g(X)$.

```
# Estimate P(at least one cooperator in 5 players) via Monte
 ↪  Carlo
# Each player cooperates independently with probability 0.3

estimate_prob <- function(n_sims, n_players = 5, p_coop = 0.3) {
  counts <- replicate(n_sims, {
    actions <- rbinom(n_players, size = 1, prob = p_coop)
    as.integer(sum(actions) >= 1)
  })
  c(estimate = mean(counts), se = sd(counts) / sqrt(n_sims))
}
```

```
# Exact answer: 1 - (1 - 0.3)^5
exact <- 1 - 0.7^5

set.seed(42)
mc <- estimate_prob(100000)
tibble(
  method   = c("Monte Carlo", "Exact"),
  estimate = c(mc["estimate"], exact)
)
```

```
#> # A tibble: 2 x 2
#>   method      estimate
#>   <chr>          <dbl>
#> 1 Monte Carlo    0.834
#> 2 Exact          0.832
```

C.8 Random sampling for games

Several chapters use random sampling to simulate strategic interactions. The key functions are:

```
# sample(): draw from a discrete set
sample(c("Hawk", "Dove"), size = 10, replace = TRUE, prob =
 ↪  c(0.4, 0.6))
```

```
#>  [1] "Dove" "Hawk" "Dove" "Dove" "Hawk" "Hawk" "Dove" "Hawk" "Dove" "Hawk"
```

```
# replicate(): repeat a simulation
replicate(5, sum(sample(1:6, 2, replace = TRUE)))
```

```
#> [1] 4 6 5 6 7
```

```
# set.seed(): ensure reproducibility
set.seed(123)
runif(3)
```

```
#> [1] 0.288 0.788 0.409
```

```
set.seed(123)
runif(3)  # identical output
```

```
#> [1] 0.288 0.788 0.409
```

C.9 Summary of R probability functions

Table C.1: R functions for common probability distributions

Distribution	PMF/PDF	CDF	Quantile	Random
Binomial	`dbinom()`	`pbinom()`	`qbinom()`	`rbinom()`
Poisson	`dpois()`	`ppois()`	`qpois()`	`rpois()`
Uniform	`dunif()`	`punif()`	`qunif()`	`runif()`
Normal	`dnorm()`	`pnorm()`	`qnorm()`	`rnorm()`
Exponential	`dexp()`	`pexp()`	`qexp()`	`rexp()`

Appendix D

Exercise Solutions

R. Heller · Date: 2026-05-08 · Last changed: 2026-05-11

Solutions to end-of-chapter exercises.

This appendix provides brief solution sketches for selected exercises from Part I (Chapters 1–8). The goal is to offer enough guidance to check your reasoning without removing the learning opportunity. Full, runnable solutions are available in the book's online repository.

Chapter 1: What is a Game?

Exercise 1 — Identify the game. *Choose a strategic interaction from everyday life and identify the four building blocks.*

Many valid answers exist. One example: two roommates deciding whether to clean a shared kitchen.

- **Players**: Roommate A and Roommate B.
- **Actions**: Clean or Don't Clean (for each player).
- **Payoffs**: A clean kitchen benefits both, but cleaning is costly. If only one cleans, that person bears the full cost while both enjoy the result — structurally a Prisoner's Dilemma.
- **Information**: Simultaneous — neither observes the other's choice before deciding.

The key criterion is mutual dependence: each player's best action depends on what the other does.

Exercise 2 — Payoff heatmap. *Create a payoff matrix for the Battle of the Sexes.*

The payoff matrices are:

$$A = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}, \quad B = \begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}$$

Use `make_payoff_df()` with strategies `c("Opera", "Football")` for both players. The heatmap should show high payoffs on the diagonal (coordination) and zeros off the diagonal (miscoordination). Unlike the Prisoner's Dilemma, there is no dominant strategy — both diagonal cells are Nash equilibria.

Chapter 2: Normal-Form Games

Exercise 1 — Identifying dominance. *Find strictly and weakly dominated strategies in the 3x3 game.*

For Player 1, compare each row against every other row across all columns:

- Row 2 yields payoffs $(3, 2, 5)$. Row 1 yields $(2, 4, 1)$. Neither strictly dominates the other (Row 1 is better in column 2, Row 2 in columns 1 and 3).
- Row 3 yields $(1, 3, 2)$. Row 2 gives $3 > 1$, $2 < 3$, $5 > 2$ — again, no strict dominance.

Check all pairwise comparisons systematically. For weak dominance, look for rows where one is at least as good in every column and strictly better in at least one. Apply the same logic to Player 2's columns using matrix B.

Exercise 2 — IESDS practice. *Apply IESDS to the 3x3 game from Exercise 1.*

Apply IESDS step by step:

1. Check for strictly dominated strategies in the original game and eliminate them.
2. In the reduced game, check again for newly dominated strategies.
3. Repeat until no further elimination is possible.

The order of elimination does not matter for strict dominance (order independence). Document each step by showing the reduced matrix. If the game reduces to a single cell, it is dominance solvable.

Chapter 3: Nash Equilibrium

Exercise 1 — Stag Hunt equilibria. *Find all Nash equilibria of the Stag Hunt.*

The two pure-strategy NE are (Stag, Stag) with payoffs $(4, 4)$ and (Hare, Hare) with payoffs $(3, 3)$. Verify by checking that no player can profitably deviate.

For the mixed NE, use the indifference principle. If Player 2 plays Stag with probability q:

$$4q + 0(1-q) = 3q + 3(1-q) \implies 4q = 3 \implies q = 3/4$$

By symmetry, $p = 3/4$ as well. The mixed NE is $(3/4, 3/4)$ with expected payoff 3 for each player.

- **Payoff-dominant**: (Stag, Stag) yields $4 > 3$.
- **Risk-dominant**: (Hare, Hare), because Hare is the best response to the mixed NE probability and is safer against uncertainty about the opponent.

Exercise 3 — Best-response plot for Chicken. *Plot the best-response correspondences.*

Set up the indifference conditions for each player using the Chicken payoffs. Player 1's expected payoff from Swerve equals their payoff from Straight at the indifference point. Solve for the critical mixing probability and plot both best-response functions on the unit square $[0, 1]^2$. The plot should reveal three intersection points: two at corners (the pure NE) and one in the interior (the mixed NE).

Chapter 4: Mixed Strategies

Exercise 1 — Asymmetric Matching Pennies. *Compute the mixed NE when Player 1 receives 3 for (Heads, Heads).*

Payoff matrices:

$$A = \begin{pmatrix} 3 & -1 \\ -1 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix}$$

Player 2's indifference (making Player 1 indifferent is governed by B):

$$-q + (1-q) = q - (1-q) \implies q^* = 1/2$$

Player 1's indifference (making Player 2 indifferent is governed by A):

$$3p - (1-p) = -p + (1-p) \implies 4p - 1 = -2p + 1 \implies p^* = 1/3$$

The asymmetry in Player 1's payoff changes Player 1's mixing probability (not Player 2's). This is the "own-payoff effect" — a player's equilibrium mixing probability is determined by the opponent's payoffs, not their own.

Exercise 4 — Rock-Paper-Scissors. *Argue that the unique mixed NE assigns 1/3 to each action.*

By the symmetry of the game, if a NE assigns different probabilities to the three actions, then permuting the labels would yield another NE with different probabilities but identical structure — contradicting uniqueness. Therefore the unique symmetric NE must assign equal probability 1/3 to each action.

More formally: the indifference conditions for any pair of actions are identical by the cyclic symmetry of the payoff matrix, so the mixing probabilities must be equal. Since they sum to 1, each is 1/3.

Chapter 5: Extensive-Form Games

Exercise 1 — Ultimatum game. *Find all NE and the SPE of the simplified ultimatum game.*

The game tree has Player 1 choosing Fair or Greedy, then Player 2 choosing Accept or Reject.

Backward induction: In each of Player 2's subgames, accepting is strictly better than rejecting ($5 > 0$ and $2 > 0$). So Player 2 accepts in both subgames. Anticipating this, Player 1 chooses Greedy (payoff $8 > 5$). The SPE is (Greedy; Accept, Accept).

Nash equilibria: There are additional NE where Player 2 threatens to reject the Greedy offer. For example, (Fair; Accept, Reject) is a NE because Player 1's deviation to Greedy yields 0 given Player 2's strategy. However, this involves a non-credible threat — Player 2 would not actually reject a payoff of 2 — so backward induction eliminates it.

Exercise 3 — Non-credible threats. *What if the incumbent's Fight payoff is 2?*

With Fight payoff of 2, the incumbent now weakly prefers fighting to accommodating (or strictly prefers it if Accommodate yields less than 2). In the subgame after entry, Fight is at least as good as Accommodate, so the threat to fight is credible.

The SPE changes: the entrant, anticipating a fight, may prefer to stay out. This illustrates a key insight — whether a threat is credible depends on the payoffs in the subgame, not on the threat itself.

Chapter 6: Bayesian Games

Exercise 1 — Modified signaling costs. *Does the separating equilibrium survive if $c_L = 3$?*

In the separating equilibrium, the High type chooses Education and the Low type does not. The Low type must not want to deviate to Education. The incentive compatibility condition for the Low type is:

$$\text{Payoff(No Education)} \geq \text{Payoff(Education)} - c_L$$

With $c_L = 3$, check whether the Low type's gain from being perceived as High (and receiving the High-type wage) exceeds the education cost. If the wage differential is, say, $8 - 2 = 6$ and $c_L = 3 < 6$, the Low type would profitably deviate to Education, breaking separation.

The minimum c_L that sustains separation is the wage differential: the Low type must find education too costly to mimic the High type.

Exercise 2 — Pooling equilibrium. *Can both types pool on No Education?*

In a pooling equilibrium on No Education, the employer observes no signal and pays the expected productivity: $E[\theta] = P(H) \cdot \theta_H + P(L) \cdot \theta_L$. Neither type deviates if the off-path belief about a worker who chooses Education is sufficiently pessimistic (e.g., the employer believes any educating worker is Low type with probability 1). Under such beliefs, education yields the Low-type wage minus the education cost, which is worse than the pooling wage for both types. These pessimistic off-path beliefs are permitted by Bayes' rule (which imposes no constraint off the equilibrium path).

Chapter 7: Repeated Games

Exercise 1 — Critical discount factor. *Compute δ^* for Grim Trigger with $T = 8, R = 5, P = 2, S = 0$.*

Under Grim Trigger, cooperation yields $V_C = R/(1-\delta) = 5/(1-\delta)$.

A one-shot deviation yields T today and then P forever: $V_D = T + \delta P/(1-\delta) = 8 + 2\delta/(1-\delta)$.

Cooperation is sustained when $V_C \geq V_D$:

$$\frac{5}{1-\delta} \geq 8 + \frac{2\delta}{1-\delta}$$

$$5 \geq 8(1-\delta) + 2\delta = 8 - 6\delta$$

$$6\delta \geq 3 \implies \delta^* = 1/2$$

Verify in R by computing V_C and V_D at $\delta = 0.5$ and confirming they are equal.

Exercise 3 — Folk theorem region. *Plot the feasible and individually rational region.*

The feasible payoff set is the convex hull of the four payoff profiles: $(4, 4)$, $(0, 6)$, $(6, 0)$, $(1, 1)$. The individually rational region requires each player's payoff to be at least their minmax value. In a 2x2 game, compute each player's minmax payoff by finding the worst the opponent can guarantee.

The Folk theorem region is the intersection: feasible payoffs that dominate both players' minmax values. Plot the convex hull as a polygon and shade the region above both minmax thresholds.

Chapter 8: Cooperative Game Theory

Exercise 1 — Asymmetric savings. *Compute the Shapley value for the modified water treatment game.*

With $v(\{A\}) = v(\{B\}) = v(\{C\}) = 0$, $v(\{A, B\}) = 50$, $v(\{A, C\}) = 30$, $v(\{B, C\}) = 40$, $v(\{A, B, C\}) = 80$.

The Shapley value for player i is:

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!\,(n - |S| - 1)!}{n!} \left[v(S \cup \{i\}) - v(S) \right]$$

For a 3-player game, enumerate all $3! = 6$ orderings and compute each player's marginal contribution in each ordering. Average across orderings.

For example, in ordering (A, B, C): A contributes $v(\{A\}) - v(\emptyset) = 0$, B contributes $v(\{A, B\}) - v(\{A\}) = 50$, C contributes $v(\{A, B, C\}) - v(\{A, B\}) = 30$.

Repeat for all six orderings and average. Municipality A benefits from large coalitions with B; Municipality B is the most valuable partner overall.

Exercise 4 — Glove game. *Write the characteristic function, compute the Shapley value, and find the core.*

Characteristic function: $v(S) = \min(\text{left gloves in } S, \text{right gloves in } S)$. So $v(\{1\}) = v(\{2\}) = v(\{3\}) = 0$, $v(\{1,2\}) = 0$ (two left, no right), $v(\{1,3\}) = v(\{2,3\}) = 1$, $v(\{1,2,3\}) = 1$.

Shapley value: enumerate all $3! = 6$ orderings. Player 3 (the sole right-glove holder) is pivotal in 4 out of 6 orderings, yielding $\phi_3 = 4/6 = 2/3$. By symmetry, $\phi_1 = \phi_2 = 1/6$.

Core: the core requires $x_1+x_3 \geq 1$, $x_2+x_3 \geq 1$, $x_1+x_2 \geq 0$, $x_1+x_2+x_3 = 1$, and $x_i \geq 0$. From the first two constraints and efficiency: $x_3 \geq 1-x_1$ and $x_3 \geq 1-x_2$. Since $x_1 + x_2 = 1 - x_3$, both constraints give $x_3 = 1$ and $x_1 = x_2 = 0$. The core is the single point $(0, 0, 1)$. The Shapley value $(1/6, 1/6, 2/3)$ does not lie in the core.

Additional selected solutions

Below are brief hints for a few more exercises that students frequently ask about.

Chapter 2, Exercise 4 — From story to matrix. *Two firms choose Invest or Stay.*

The payoff matrix is:

$$A = B = \begin{pmatrix} 5 & 1 \\ 4 & 3 \end{pmatrix}$$

where rows/columns are (Invest, Stay). Check for dominance: against Invest, Stay yields $4 > 1$ (in the role of the other firm); against Stay, Stay yields $3 > 1$. Stay strictly dominates Invest for both players, so (Stay, Stay) with payoffs $(3, 3)$ is the unique NE. This is a Prisoner's Dilemma: mutual investment would yield $(5, 5)$ but the dominant strategy leads to a Pareto-inferior outcome.

Chapter 4, Exercise 2 — Inspection game. *Find the mixed NE of the Worker-Employer game.*

Payoff matrices (Worker is row player, Employer is column player):

$$A = \begin{pmatrix} 2 & 1 \\ 3 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 3 & 2 \\ 0 & 1 \end{pmatrix}$$

where rows are (Work, Shirk) and columns are (Trust, Monitor).

No pure-strategy NE exists (verify by checking best responses). For the mixed NE, let the Worker play Work with probability p and the Employer play Trust with probability q.

Employer's indifference (using A): $2q + 1(1-q) = 3q + 0(1-q)$, so $q + 1 = 3q$, giving $q^* = 1/2$.

Worker's indifference (using B): $3p + 0(1-p) = 2p + 1(1-p)$, so $3p = p + 1$, giving $p^* = 1/2$.

Increasing the penalty for caught shirking (lowering $A_{2,2}$ below 0) does *not* change p^* in equilibrium — it changes the employer's mixing probability q^* instead. This is another instance of the "own-payoff" paradox from the indifference principle.

Chapter 5, Exercise 4 — SPE vs. NE counting. *Count equilibria in the ultimatum game.*

The normal form has 2 strategies for Player 1 (Fair, Greedy) and 4 strategies for Player 2 (Accept/Accept, Accept/Reject, Reject/Accept, Reject/Reject — one decision for each of Player 1's possible offers).

Pure-strategy NE: find all cells where both players play a best response. There are three NE:

1. (Greedy; Accept, Accept) — payoffs (8, 2).
2. (Fair; Accept, Reject) — payoffs (5, 5).
3. (Fair; Reject, Reject) — payoffs (5, 5).

Only NE 1 is subgame perfect, because in NE 2 and 3 Player 2's plan to reject the Greedy offer is not sequentially rational ($2 > 0$). Thus SPE $\subset$ NE strictly.

Chapter 7, Exercise 2 — Tit-for-Tat critical discount factor. *Derive δ^* for TFT.*

Under TFT with $T = 5, R = 3, P = 1, S = 0$: cooperation yields $V_C = 3/(1-\delta)$.

A deviation yields $T = 5$ today. TFT retaliates next period (defects once), so the deviator gets $S = 0$ in period 2. Then TFT returns to cooperation (it copies the opponent's last action), so from period 3 onward the deviator gets $R = 3$ again — but only if the deviator also returns to cooperation. The one-shot deviation payoff stream is:

$$V_D = 5 + 0 \cdot \delta + 3 \cdot \frac{\delta^2}{1-\delta}$$

Setting $V_C \geq V_D$:

$$\frac{3}{1-\delta} \geq 5 + \frac{3\delta^2}{1-\delta}$$

$$3 \geq 5(1-\delta) + 3\delta^2 = 5 - 5\delta + 3\delta^2$$

$$3\delta^2 - 5\delta + 2 \leq 0$$

Solving: $\delta = \frac{5 \pm \sqrt{25-24}}{6} = \frac{5 \pm 1}{6}$, so $\delta \in [2/3, 1]$. The critical discount factor is $\delta^* = 2/3$, which is higher than the Grim Trigger threshold ($1/2$ for these payoffs), reflecting TFT's milder punishment.

Appendix E

Glossary

R. Heller · Date: 2026-05-08 · Last changed: 2026-11-03

> Key terms and definitions used throughout this book. Action : A choice available to a player at a decision point. In a simultaneous game, the terms “action” and “strategy” are often used interchangeably.

Agent-based model A computational model in which autonomous agents interact according to specified rules, producing emergent macro-level behaviour. See 17.

Backward induction A solution method for extensive-form games of perfect information. Starting from terminal nodes and working backwards, each player's optimal action is determined at every decision node. The resulting strategy profile is a subgame perfect equilibrium. See 6.

Battle of the Sexes A 2x2 coordination game with two pure-strategy Nash equilibria and one mixed equilibrium. The players prefer to coordinate but disagree on which outcome is better.

Bayesian game A game in which players have private information (types) drawn from a known prior distribution. Players maximise expected payoffs given their beliefs about opponents' types. See 7.

Bayesian Nash equilibrium (BNE) A strategy profile in a Bayesian game where each type of each player maximises expected payoff given the prior distribution over opponents' types and their strategies.

Best response A strategy s_i^* that maximises player i's payoff given the strategies of all other players. A Nash equilibrium is a profile where every player's strategy is a best response.

Characteristic function In cooperative game theory, a function $v : 2^N \to \mathbb{R}$ that assigns a worth to every coalition $S \subseteq N$. See 9.

Chicken (Hawk-Dove) A 2x2 anti-coordination game modelling brinkmanship. Each player prefers the other to yield. It has two asymmetric pure-strategy NE and one mixed NE.

Common knowledge A fact is common knowledge if all players know it, all players know that all players know it, and so on ad infinitum. The structure of the game is typically assumed to be common knowledge.

Cooperative game theory The branch of game theory that studies what coalitions of players can achieve and how the joint gains should be allocated. Focuses on outcomes rather than strategies. See 9.

Core The set of feasible payoff allocations in a cooperative game that no coalition can improve upon. An allocation is in the core if no subset of players can do better by breaking away.

Discount factor A parameter $\delta \in [0, 1)$ representing the weight a player places on future payoffs relative to current payoffs. Central to repeated game analysis. See 8.

Dominant strategy A strategy that yields a payoff at least as high as any alternative regardless of opponents' actions (weak dominance) or strictly higher (strict dominance).

Evolutionary stable strategy (ESS) A strategy that, once prevalent in a population, cannot be invaded by any rare mutant strategy. Formally, a strategy s^* is an ESS if it is a best response to itself and, against any alternative best response, does strictly better against the mutant than the mutant does against itself. See 21.

Extensive form A representation of a game as a tree, specifying the order of moves, information available at each decision point, and payoffs at terminal nodes. See 6.

Feasible payoff set The convex hull of all achievable payoff vectors in a game. In repeated games, the folk theorem concerns the subset of feasible payoffs that are individually rational.

Folk theorem A family of results stating that in infinitely repeated games with sufficiently patient players (δ close to 1), any feasible and individually rational payoff vector can be sustained as a Nash equilibrium outcome. See 8.

Game tree The tree representation of an extensive-form game, with nodes representing decision points and edges representing actions.

Grim Trigger A strategy in repeated games that cooperates until the opponent defects, then defects forever. The simplest strategy that can sustain cooperation via the threat of permanent punishment.

Imperfect information A game has imperfect information if at least one player does not observe all previous moves when making a decision. Represented by information sets in the game tree.

Incomplete information A game has incomplete information if players are uncertain about some aspect of the game, such as other players' payoffs or types. Modelled as a Bayesian game.

Individually rational payoff The minimum payoff a player can guarantee regardless of opponents' strategies (the minmax value). In repeated games,

the folk theorem applies to payoffs above this threshold.

Information set In an extensive-form game, a collection of decision nodes among which a player cannot distinguish. The player must choose the same action at all nodes in the same information set.

Matching Pennies A 2x2 zero-sum game where one player wins if outcomes match and the other wins if they differ. It has a unique Nash equilibrium in mixed strategies.

Mechanism design The "inverse" of game theory: designing the rules of a game (mechanism) so that self-interested players produce a desired outcome in equilibrium. See 33.

Minmax value The lowest payoff that opponents can force on a player, or equivalently, the highest payoff a player can guarantee against the worst-case opponent behaviour.

Mixed strategy A probability distribution over a player's pure strategies. A player randomises among actions according to the specified probabilities. See 5.

Monte Carlo method A computational technique that uses random sampling to estimate quantities or simulate systems. Used throughout Part III for game simulations. See 16.

Nash equilibrium (NE) A strategy profile in which no player can increase their payoff by unilaterally changing their strategy. Every finite game has at least one Nash equilibrium (possibly in mixed strategies). See 4.

Normal form (strategic form) A representation of a simultaneous game as a matrix of payoffs. Each row corresponds to a strategy of one player, each column to a strategy of the other. See 3.

Pareto optimal (Pareto efficient) An outcome is Pareto optimal if no alternative outcome makes at least one player strictly better off without making any player worse off. The Prisoner's Dilemma equilibrium is famously Pareto inefficient.

Payoff matrix The matrix encoding all players' payoffs for every combination of strategies in a normal-form game.

Perfect information A game has perfect information if every player, when making a decision, observes all previously taken actions. Chess is a canonical example.

Prisoner's Dilemma A 2x2 game where each player has a dominant strategy to defect, yet mutual cooperation yields a higher payoff for both. The unique NE is Pareto inefficient.

Pure strategy A deterministic plan of action that specifies exactly which action a player will take at every decision point.

Replicator dynamics A system of differential equations describing how the proportions of strategies in a population change over time based on their relative fitness. See 20.

Shapley value A unique allocation in cooperative game theory that distributes the total value among players according to their average marginal contribution across all possible orderings. See 9.

Signaling game A dynamic Bayesian game where an informed player (sender)

takes an observable action to convey information to an uninformed player (receiver). See 7.

Stag Hunt A 2x2 coordination game with two pure-strategy NE: one payoff-dominant (both hunt Stag) and one risk-dominant (both hunt Hare). It models the tension between cooperation and safety.

Strategy profile A specification of one strategy for every player in the game. An n-player game has a strategy profile $(s_1, s_2, \ldots, s_n)$.

Subgame A portion of an extensive-form game that starts at a singleton information set, contains all successors, and does not break any information set.

Subgame perfect equilibrium (SPE) A strategy profile that constitutes a Nash equilibrium in every subgame of the original game. Found by backward induction in games of perfect information. See 6.

Support (of a mixed strategy) The set of pure strategies assigned positive probability in a mixed strategy. At a Nash equilibrium, every strategy in the support must yield the same expected payoff.

Tit-for-Tat (TFT) A strategy in repeated games that cooperates in the first round, then copies the opponent's previous action. Famously successful in Axelrod's tournaments. See 18.

Zero-sum game A game in which the sum of all players' payoffs is zero (or constant) for every outcome. One player's gain is exactly another's loss. Matching Pennies is a canonical example.

Acknowledgments

R. Heller · Date: 2026-05-11 · Last changed: 2026-06-10

This book stands on the shoulders of the open R ecosystem. Special thanks to the maintainers of `bookdown`, the tidyverse, and the domain-specific packages that make computational game theory tractable in R: `GameTheory`, `CoopGame`, `gtree`, `EvolutionaryGames`, `igraph`, `Rcpp`, and many others.

I am grateful to the bookdown community for tooling that made this book possible, and to the broader R community for sustained generosity with code, documentation, and ideas.

Drafting and editorial assistance was provided by Claude (Anthropic). All technical content has been reviewed, verified, and curated by the author. Errors remain my own.

Inspiration

The structural and pedagogical model for this book draws heavily on Mathias Harrer, Pim Cuijpers, Toshi A. Furukawa, and David D. Ebert's *Doing Meta-Analysis with R: A Hands-On Guide* — both the book and its open-source repository at https://github.com/MathiasHarrer/Doing-Meta-Analysis-in-R. Their thorough, transparent, and beautifully laid-out bookdown setup — sidebar grouping, semantic callout boxes, the citation block, the dark/light toggle, MathJax-rendered formulas, downloadable BibTeX/RIS citations — set a clear standard for what a reproducible R-based research book can look like. Where the present book mirrors their conventions, it does so deliberately and gratefully. Where it diverges (colors, typography, individual icons), the divergence is cosmetic; the underlying craft is theirs to credit.

Use of LLM tools

Portions of this book were prepared with assistance from large language model tooling for narrowly defined, non-authorial tasks: copyediting, prose smoothing,

Markdown/LaTeX formatting, scaffolding of boilerplate files (CI configs, build scripts), code refactoring. The tools used were Chat AI, the LLM service of KISSKI (GWDG), and a self-hosted **Mistral Small (24B, Apache-2.0)** run locally via Ollama and the `ollamar` R package — local inference only, with no data sent to third parties for the self-hosted model.

All scientific claims, methodological choices, analyses, interpretations, and conclusions are the author's own. No LLM-generated text was incorporated without review and revision, and every reference was verified against its DOI, arXiv ID, or ISBN.

About the author

R. Heller · Date: 2026-05-11 · Last changed: 2026-11-03

Raban Heller is a biomedical researcher and biostatistician working at the intersection of clinical data, statistical computing, and machine intelligence. He writes on R and reproducible research at https://r-heller.github.io/website/.

- ORCID: 0000-0001-8006-9742
- GitHub: @r-heller
- Website: https://r-heller.github.io/website/

Colophon

R. Heller · Date: 2026-05-11 · Last changed: 2026-06-10

```
#> Built with:

#>   bookdown: 0.48

#>   R:        R version 4.6.2 (2026-10-29)

#>   Build date: 2026-11-14

#>   Commit:   498ede7
```

LLM assistance (Chat AI from KISSKI/GWDG, plus a self-hosted Mistral Small via Ollama/`ollamar`) was used for copyediting, formatting, and scaffolding tasks only — see Acknowledgments.

Session info

```
#> - Session info ---------------------------------------------------------------
#>  setting  value
#>  version  R version 4.6.2 (2026-10-29)
#>  os       Ubuntu 26.04.1 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2026-11-14
#>  pandoc   3.8.3 @ /opt/hostedtoolcache/pandoc/3.8.3/x64/ (via rmarkdown)
#>  quarto   NA
```

```
#>
#> - Packages -------------------------------------------------------------------
#>  ! package   * version date (UTC) lib source
#>  P dplyr     * 1.2.1   2026-04-03 [?] RSPM
#>  P forcats   * 1.0.1   2025-09-25 [?] RSPM
#>  P ggplot2   * 4.0.3   2026-04-22 [?] RSPM
#>  P glue      * 1.8.1   2026-04-17 [?] RSPM
#>  P gt        * 1.3.0   2026-01-22 [?] RSPM
#>  P here      * 1.0.2   2025-09-15 [?] RSPM
#>  P lubridate * 1.9.5   2026-02-04 [?] RSPM
#>  P purrr     * 1.2.2   2026-04-10 [?] RSPM
#>  P readr     * 2.2.0   2026-02-19 [?] RSPM
#>  P scales    * 1.4.0   2025-04-24 [?] RSPM
#>  P stringr   * 1.6.0   2025-11-04 [?] RSPM
#>  P tibble    * 3.3.1   2026-01-11 [?] RSPM
#>  P tidyr     * 1.3.2   2025-12-19 [?] RSPM
#>  P tidyverse * 2.0.0   2023-02-22 [?] RSPM
#>
#>  [1] /home/runner/work/strategy-in-r/strategy-in-r/renv/library/linux-ubuntu-resolut
#>  [2] /home/runner/.cache/R/renv/sandbox/linux-ubuntu-resolute/R-4.6/x86_64-pc-linux-
#>
#>  * -- Packages attached to the search path.
#>  P -- Loaded and on-disk path mismatch.
#>
#> ------------------------------------------------------------------------------
```

Citing this Guide

R. Heller · Date: 2026-05-11 · Last changed: 2026-11-09

The suggested citation is:

Heller, R. (2026). *Strategy in R: Game Theory, Simulation, and Machine Intelligence* (Version v0.1.0). Self-published via GitHub Pages. https://cttir.github.io/strategy-in-r/.

Download the reference as BibTeX or RIS.

References

R. Heller · Date: 2026-05-11 · Last changed: 2026-05-11

Arora, S., Hazan, E., & Kale, S. (2012). The multiplicative weights update method: A meta-algorithm and applications. *Theory of Computing*, *8*(1), 121–164. https://doi.org/10.4086/toc.2012.v008a006

Axelrod, R. (1981). The evolution of cooperation. *Science*, *211*(4489), 1390–1396. https://doi.org/10.1126/science.7466396

Barasz, M., Christiano, P., Fallenstein, B., Herreshoff, M., LaVictoire, P., & Yudkowsky, E. (2014). Robust cooperation in the prisoner's dilemma: Program equilibrium via provability logic. *arXiv Preprint*.

Bowling, M., Burch, N., Johanson, M., & Tammelin, O. (2015). Heads-up limit hold'em poker is solved. *Science*, *347*(6218), 145–149. https://doi.org/10.1126/science.1259433

Brown, N., & Sandholm, T. (2019). Superhuman AI for multiplayer poker. *Science*, *365*(6456), 885–890. https://doi.org/10.1126/science.aay2400

Camerer, C. F., Ho, T.-H., & Chong, J.-K. (2004). A cognitive hierarchy model of games. *Quarterly Journal of Economics*, *119*(3), 861–898. https://doi.org/10.1162/0033553041502225

Cesa-Bianchi, N., & Lugosi, G. (2006). *Prediction, learning, and games*. Cambridge University Press.

Chouldechova, A. (2017). Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big Data*, *5*(2), 153–163. https://doi.org/10.1089/big.2016.0047

Dafoe, A., Hughes, E., Bachrach, Y., Collins, T., McKee, K. R., Leibo, J. Z., Larson, K., & Graepel, T. (2020). Open problems in cooperative AI. *arXiv Preprint*.

Daskalakis, C., Goldberg, P. W., & Papadimitriou, C. H. (2009). The complexity of computing a nash equilibrium. *SIAM Journal on Computing*, *39*(1), 195–259. https://doi.org/10.1137/070699652

Dwork, C., Hardt, M., Pitassi, T., Reingold, O., & Zemel, R. (2012). Fairness through awareness. *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, 214–226. https://doi.org/10.1145/2090236.2090255

Fudenberg, D., & Levine, D. K. (1998). *The theory of learning in games*. MIT Press.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative adversarial nets. *Advances in Neural Information Processing Systems*, *27*, 2672–2680.

Hadfield-Menell, D., Russell, S. J., Abbeel, P., & Dragan, A. (2016). Cooperative inverse reinforcement learning. *Advances in Neural Information Processing Systems*, *29*, 3909–3917.

Hardt, M., Price, E., & Srebro, N. (2016). Equality of opportunity in supervised learning. *Advances in Neural Information Processing Systems*, *29*, 3315–3323.

Hart, S., & Mas-Colell, A. (2000). A simple adaptive procedure leading to correlated equilibrium. *Econometrica*, *68*(5), 1127–1150. https://doi.org/10.1111/1468-0262.00153

Hofbauer, J., & Sigmund, K. (1998). *Evolutionary games and population dynamics*. Cambridge University Press.

Kennan, J., & Wilson, R. (1993). Bargaining with private information. *Journal of Economic Literature*, *31*(1), 45–104.

Kilbertus, N., Carulla, M. R., Parascandolo, G., Hardt, M., Janzing, D., & Schölkopf, B. (2017). Avoiding discrimination through causal reasoning. *Advances in Neural Information Processing Systems*, *30*, 656–666.

Kleinberg, J., Mullainathan, S., & Raghavan, M. (2017). Inherent trade-offs in the fair determination of risk scores. *Proceedings of Innovations in Theoretical Computer Science*, 43:1–43:23.

Kuhn, H. W. (1950). A simplified two-person poker. *Contributions to the Theory of Games*, *1*, 97–103.

Lanctot, M., Zambaldi, V., Gruslys, A., Lazaridou, A., Tuyls, K., Pérolat, J., Silver, D., & Graepel, T. (2017). A unified game-theoretic approach to multiagent reinforcement learning. *Advances in Neural Information Processing Systems*, *30*, 4190–4203.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. *Proceedings of the Eleventh International Conference on Machine Learning*, 157–163. https://doi.org/10.1016/B978-1-55860-335-6.50027-1

Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., & Mordatch, I. (2017). Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in Neural Information Processing Systems*, *30*, 6379–6390.

Maynard Smith, J. (1982). *Evolution and the theory of games*. Cambridge University Press.

Meta Fundamental AI Research Diplomacy Team (FAIR), Bakhtin, A., Brown, N., Dinan, E., Farina, G., Flaherty, C., Fried, D., Goff, A., Gray, J., Hu, H., et al. (2022). Human-level play in the game of diplomacy by combining language models with strategic reasoning. *Science*, *378*(6624), 1067–1074. https://doi.org/10.1126/science.ade9097

Moulin, H. (2004). *Fair division and collective welfare*. MIT Press.

Myerson, R. B. (1991). *Game theory: Analysis of conflict*. Harvard University

Press.

Nagel, R. (1995). Unraveling in guessing games: An experimental study. *American Economic Review*, *85*(5), 1313–1326.

Nash, J. F. (1950). Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, *36*(1), 48–49. https://doi.org/10.1073/pnas.36.1.48

Neumann, J. von, & Morgenstern, O. (1944). *Theory of games and economic behavior*. Princeton University Press.

Nisan, N., Roughgarden, T., Tardos, É., & Vazirani, V. V. (2007). *Algorithmic game theory*. Cambridge University Press.

Nowak, M. A., & May, R. M. (1992). Evolutionary games and spatial chaos. *Nature*, *359*(6398), 826–829. https://doi.org/10.1038/359826a0

Nowak, M. A., Sasaki, A., Taylor, C., & Fudenberg, D. (2004). Emergence of cooperation and evolutionary stability in finite populations. *Nature*, *428*(6983), 646–650. https://doi.org/10.1038/nature02414

Osborne, M. J. (2004). *An introduction to game theory*. Oxford University Press.

Rawls, J. (1971). *A theory of justice*. Harvard University Press.

Roth, A. E. (2002). The economist as engineer: Game theory, experimentation, and computation as tools for design economics. *Econometrica*, *70*(4), 1341–1378. https://doi.org/10.1111/1468-0262.00335

Roth, A. E., & Sotomayor, M. A. O. (1992). *Two-sided matching: A study in game-theoretic modeling and analysis*. Cambridge University Press.

Roughgarden, T., & Tardos, É. (2002). How bad is selfish routing? *Journal of the ACM*, *49*(2), 236–259. https://doi.org/10.1145/506147.506153

Rubinstein, A. (1982). Perfect equilibrium in a bargaining model. *Econometrica*, *50*(1), 97–109. https://doi.org/10.2307/1912531

Santos, F. C., & Pacheco, J. M. (2005). Scale-free networks provide a unifying framework for the emergence of cooperation. *Physical Review Letters*, *95*(9), 098104. https://doi.org/10.1103/PhysRevLett.95.098104

Schelling, T. C. (1960). *The strategy of conflict*. Harvard University Press.

Schuster, P., & Sigmund, K. (1983). Replicator dynamics. *Journal of Theoretical Biology*, *100*(3), 533–538. https://doi.org/10.1016/0022-5193(83)90445-9

Sen, A. (1999). *Development as freedom*. Oxford University Press.

Shapley, L. S. (1953). A value for n-person games. *Annals of Mathematics Studies*, *28*, 307–317. https://doi.org/10.1515/9781400881970-018

Shoham, Y., & Leyton-Brown, K. (2009). *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Driessche, G. van den, Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. https://doi.org/10.1038/nature16961

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan,

K., & Hassabis, D. (2017). *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*. https://doi.org/10.48550/arXiv.1712.01815

Soares, N., & Fallenstein, B. (2015). Agent foundations for aligning machine intelligence with human interests: A technical research agenda. *Machine Intelligence Research Institute Technical Report*.

Stahl, D. O., & Wilson, P. W. (1995). On players' models of other players: Theory and experimental evidence. *Games and Economic Behavior*, *10*(1), 218–254. https://doi.org/10.1006/game.1995.1031

Strang, G. (2023). *Introduction to linear algebra* (6th ed.). Wellesley-Cambridge Press. https://math.mit.edu/~gs/linearalgebra/ila6/indexila6.html

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). MIT Press.

Szabó, G., & Fáth, G. (2007). Evolutionary games on graphs. *Physics Reports*, *446*(4–6), 97–216. https://doi.org/10.1016/j.physrep.2007.04.004

Taylor, P. D., & Jonker, L. B. (1978). Evolutionary stable strategies and game dynamics. *Mathematical Biosciences*, *40*(1–2), 145–156. https://doi.org/10.1016/0025-5564(78)90077-9

Tennenholtz, M. (2004). Program equilibrium. *Games and Economic Behavior*, *49*(2), 363–373. https://doi.org/10.1016/j.geb.2004.02.002

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., … Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, *575*(7782), 350–354. https://doi.org/10.1038/s41586-019-1724-z

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*(3–4), 279–292. https://doi.org/10.1007/BF00992698

Watts, D. J., & Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks. *Nature*, *393*(6684), 440–442. https://doi.org/10.1038/30918

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). *R for data science* (2nd ed.). O'Reilly Media.

Zinkevich, M., Johanson, M., Bowling, M., & Piccione, C. (2007). Regret minimization in games with incomplete information. *Advances in Neural Information Processing Systems*, *20*, 1729–1736.