

Modelos econômicos aplicados em Python¹

Matheus L. Carrijo²

LEMC-FEARP/USP & DCM-FFCLRP/USP

22 de janeiro de 2022

¹O autor agradece o apoio financeiro do Programa Unificado de Bolsas (PUB), o suporte institucional da Universidade de São Paulo ao *Laboratório de Economia, Matemática e Computação* (LEMC-FEARP/USP), contexto no qual este trabalho foi desenvolvido, e a excepcional orientação dos professores Jefferson Bertolai e Fernando Barros Jr. durante os encontros do Grupo de Estudos em Economia Computacional do LEMC-FEARP/USP. O autor agradece, ainda, as excelentes e frutíferas discussões com os colegas participantes deste projeto, sem as quais muito do conteúdo deste trabalho não seria possível de ser implementado.

²Faculdade de Filosofia, Ciências e Letras de Ribeirão Preto - Universidade de São Paulo. Departamento de Computação e Matemática. ORCID: 0000-0002-3429-214X. Email: matheuslcarrijo@usp.br (✉ **corresponding author**).

Sumário

1	Introdução	7
2	Oferta e Demanda	9
2.1	O modelo	9
2.2	Implementação computacional	10
3	Crescimento Econômico: Modelo de Solow	13
3.1	O modelo	13
3.2	Implementação computacional	14
4	Renda Nacional: o Modelo IS-LM	19
4.1	O modelo	19
4.2	Implementação computacional	21
5	Desemprego: <i>Lake Model</i>	27
5.1	O modelo	27
5.2	Implementação computacional	28
5.3	Extensões do modelo	33
6	Escolha Intertemporal	37
6.1	O modelo	37
6.2	Implementação computacional	39
7	Escolha sob incerteza	45
7.1	O modelo	46
7.2	Implementação computacional	48
8	Stag Hunt Game (Rousseau)	51
8.1	O modelo	51
8.2	Implementação Computacional	53

9	Oligopólio (competição imperfeita)	59
9.1	O modelo	59
9.2	Implementação Computacional	61
10	Caixa de Edgeworth: uma análise para o equilíbrio geral	67
10.1	O modelo	67
10.2	Implementação Computacional	70
11	Matching (estabilidade de casamentos)	79
11.1	O modelo	79
11.2	Implementação Computacional	80
12	Equilíbrio Geral	91
12.1	O modelo	91
12.2	Implementação Computacional	91
13	Desenho de Mecanismo	93
13.1	O modelo	93
13.2	Implementação Computacional	93
	Referências Bibliográficas	95
	Apêndice A Programas completos	97
A.1	Oferta e Demanda	97
A.2	Crescimento Econômico: Modelo de Solow	99
A.3	Renda Nacional: Modelo IS-LM	102
A.4	Desemprego: <i>Lake Model</i>	108
A.5	Escolha Intertemporal	114
A.6	Escolha sob Incerteza	117
A.7	Stag Hunt Game (Rousseau)	119
A.8	Oligopólio	126
A.9	Caixa de Edgeworth	130
A.10	Matching	136
A.11	Equilíbrio Geral	143
A.12	Desenho de Mecanismo	143

Lista de Figuras

2.1	<i>Equilíbrio entre oferta e demanda.</i>	12
3.1	<i>Investimento, depreciação e estado estacionário.</i>	16
4.1	<i>O equilíbrio simultâneo nos mercados de bens e serviços e monetário.</i>	25
5.1	<i>Exemplo de uma economia em seu nível estacionário de desemprego.</i>	30
5.2	<i>Estática comparativa das taxas de desemprego e emprego.</i>	31
5.3	<i>Frequência dos trabalhadores encontrando emprego no tempo.</i>	35
5.4	<i>Estática comparativa: frequência de trabalhadores encontrando emprego no tempo.</i>	36
6.1	<i>Restrição orçamentária.</i>	40
6.2	<i>Consumo ótimo para um exemplo de economia.</i>	41
6.3	<i>Consumo ótimo para diferentes taxas de juros.</i>	42
7.1	<i>A restrição orçamentária.</i>	46
7.2	<i>Escolha ótima do consumidor para um exemplo de economia.</i>	49
9.1	<i>O equilíbrio de Cournot.</i>	64
9.2	<i>Efeito do aumento no número de firmas nos níveis de produção e preço.</i>	65
10.1	<i>Caixa de Edgeworth.</i>	69
10.2	<i>Curva de contrato.</i>	74
10.3	<i>Caixa de Edgeworth para um exemplo de economia.</i>	75
11.1	<i>Propostas e casais formados por rodada.</i>	88

Lista de Tabelas

8.1	Matriz de <i>payoffs</i> do jogo Dilema dos Prisoneiros.	52
8.2	Matriz geral de <i>payoffs</i> do jogo <i>Stag Hunt</i>	53
8.3	Matriz particular de <i>payoffs</i> do jogo <i>Stag Hunt</i>	58

Capítulo 1

Introdução

Para ser escrito.

Capítulo 2

Oferta e Demanda

Sendo um dos conceitos (se não o conceito) mais usado em economia, a interação entre oferta e demanda pode ser implementada de modo simples na linguagem Python. O modelo pode ser estudado de forma mais básica através de Mankiw (2011). Para uma abordagem mais formal, o estudo da teoria do consumidor e da firma através de Varian (2012) pode fornecer o rigor necessário para melhor entender o que há por detrás das curvas de oferta e demanda e do modelo como um todo.

2.1 O modelo

Basicamente, a um nível agregado, a curva de demanda nos informa a propensão a pagar dos consumidores para determinadas quantidades de um produto. Como elemento fundamental, o preço deste bem é quem vai determinar a quantidade demandada. Em geral, assumimos que *ceteris paribus a quantidade demandada varia na direção oposta do preço*, isto é, quando o preço aumenta, a quantidade demandada do bem cai; quando o preço diminui, a quantidade demandada aumenta. Esta lógica é tão geral que os economistas a chamam de *lei da demanda*.¹

Já no que diz respeito à curva de oferta, pode-se dizer que ela nos informa a quantidade que os produtores estão dispostos a vender para qualquer preço dado. Novamente, o preço desempenha um papel mais que importante aqui. Ao contrário do que ocorre na lei da demanda, a *lei da oferta* nos diz que quanto maior o preço, maior a quantidade que os vendedores estão dispostos a vender; e quanto menor o preço, menor é a quantidade do bem que os comerciantes estão dispostos a vender. Ou seja, *há uma relação diretamente proporcional entre preço e quantidade ofertada*.

Assim, a *interação* entre estas duas curvas permite com que analisemos, num nível macro, o comportamento dos compradores e vendedores de um mercado, estabelecendo, por exemplo,

¹Cabe destacar, no entanto, que para certos tipos de bem, esta lei não vale. Como não estamos interessados em estudar estes desvios, manteremos este conceito como elemento norteador da análise de oferta e demanda.

qual o preço e quantidade que faz com que toda oferta de um bem seja demandada pelos consumidores.

2.2 Implementação computacional

Começarei importando as bibliotecas necessárias ao programa através do código abaixo. Elas serão considerados o padrão para todos os programas escritos neste documento. Assim, é importante que o leitor mantenha-as em todos os códigos de programas, exceto quando lhe for dito o contrário. Outras bibliotecas serão necessárias em alguns códigos e, quando for o caso, o leitor também será alertado.

```
1 #####
2 # Importando pacotes #
3 #####
4
5 import matplotlib.pyplot as plt #importing graph package
6 plt.figure(figsize=(9, 6), dpi=100) #set default figure size
7 import numpy as np #importing numpy
```

Como o comentário no código acima deixa claro, a biblioteca `matplotlib.pyplot` é suficiente para gerar figuras e plotar gráficos.² O comando da linha 6 configura o tamanho e a qualidade da figura. Por fim, a biblioteca `numpy` será extremamente importante para tudo o que envolver programação numérica dos modelos. Para uma exposição mais detalhada sobre, ver a seção sobre esta biblioteca em Sargent and Stachurski (2021) ou clicando aqui.

Em seguida, deve-se definir as funções que serão usadas na análise do modelo de oferta e demanda. O seguinte código deve ser implementado:

```
1 def demanda(p, a = -2, b = 7):
2     # Setting demand function (slope coefficient must be negative!)
3
4     assert a < 0 and b > 0
5
6     global a2, b2 # estabelece as variáveis como globais, isto é, elas são
7                   # reconhecidas em todo o programa, não só dentro das funções
8     a2, b2 = a, b
9
10    return a*p + b
11
```

²Ver Sargent and Stachurski (2021), mais especificamente a seção que trata sobre a referida biblioteca (clique aqui).

```

12 def oferta(p, c = 4, d = 1):
13     # Setting supply function (slope coefficient must be posive!)
14     # d in (0,b) é suficiente pra equilibrio
15
16     assert c > 0 and d > 0 and b2 > d
17
18     global c2, d2
19     c2, d2 = c, d
20
21     return c*p + d
22
23 def excesso_demanda(p):
24     # Setting excess demand function. It must be clear that equilibrium requires
25     # excesso_demanda(p) = 0, for some especific p
26
27     return demanda(p) - oferta(p)

```

As duas primeiras funções descrevem, como o próprio nome diz, a demanda e a oferta de mercado. O comando `assert` garante que os coeficientes das funções estejam no intervalo correto. A função `excesso_demanda(p)` calcula a diferença entre a demanda e oferta, para um dado preço. Assim, fica claro que o equilíbrio da economia ocorre quando esta diferença é nula.

Depois de definidas as funções, no programa principal devemos fazer a busca pelo preço que torna o excesso de demanda nulo. O programa principal é implementado da seguinte maneira:

```

1 print(f"\nFaremos o cálculo para a função demanda D = {a2}p + {b2}", end = " ")
2 print(f"e oferta S = {c2}p + {d2}\n")
3
4 p_grid = np.linspace(-1000, 1000, 1000)
5 q_grid = np.linspace(-1000, 1000, 1000)
6
7 E_abs = []
8 for i in p_grid:
9     E_abs.append(abs(excesso_demanda(i)))
10
11 indice_min = E_abs.index(min(E_abs))
12 p_equilibrio = p_grid[indice_min]
13 q_equilibrio = oferta(p_equilibrio)
14
15 print("O preço e a quantidade de equilíbrio são", end = "")
16 print(f", respectivamente, {p_equilibrio: .2f} e {q_equilibrio: .2f}")

```

Note que definimos um grid para os preços e quantidades. Depois, fazemos uma busca

nesse grid de tal maneira a encontrar o preço que faz o excesso de demanda ser nulo. Após este processo, descobrimos o preço e quantidade de equilíbrio simplesmente tomando este mesmo preço (que tornou o excesso de demanda nulo) em qualquer uma das funções de demanda e oferta (já que ambas intersectam-se neste mesmo valor!). Informamos, através da função `print`, ao final, o preço e a quantidade de equilíbrio, dado pelo par $(p, q) = (1, 5)$.

Finalmente, depois de concluído a análise numérica para determinar o equilíbrio desta economia, queremos visualizar graficamente o resultado. O seguinte código permite isso:

```
1 q1, q2, q3 = demanda(p_grid), oferta(p_grid), excesso_demanda(p_grid)
2 plt.plot(q1, p_grid, color="blue", linewidth=3.0, ls = "dashed")
3 plt.plot(q2, p_grid, color="green", linewidth=3.0)
4 plt.plot(q3, p_grid, color="red", linewidth=3.0, ls = "dotted")
5 plt.xlim(d2, b2)
6 plt.ylim(-d2/c2, (-b2) / a2)
7 plt.legend([f'Curva de Demanda: D = {a2}p + {b2}',
8             f'Curva de Oferta: S = {c2}p + {d2}',
9             'Curva do excesso de demanda: E = D - S'])
10 plt.ylabel("Preço", fontsize = 15)
11 plt.xlabel("Quantidade", fontsize = 15)
12 plt.show()
```

O código gera o seguinte gráfico da figura (2.1). A figura nos permite analisar o resultado do programa. Embora a implementação seja de uma economia específica, com as funções de demanda e oferta dadas, respectivamente, por $D(p) = -2p + 7$ e $S(p) = 4p + 1$, é importante observar que as funções permitem alterações destes parâmetros. O leitor pode implementar o programa e experimentar outros exemplos de uma economia.

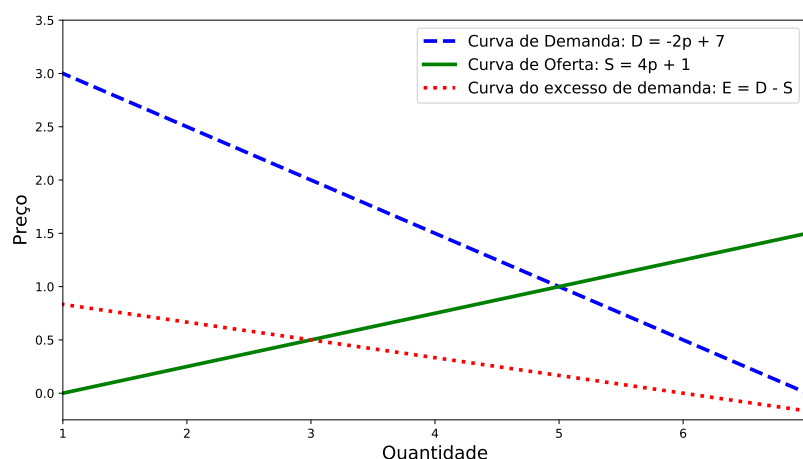


Figura 2.1: Equilíbrio entre oferta e demanda.

Capítulo 3

Crescimento Econômico: Modelo de Solow

Quando os economistas falam de crescimento econômico, é comum ter em mente o modelo de Solow. Em resumo, este modelo tenta explicar como a economia de um país se desenvolve durante um prazo de tempo muito longo. Como veremos, é um modelo possível de programar em Python e com relativa facilidade. Para o estudo mais aprofundado deste modelo, é sugerido o capítulo 8 de Mankiw (2015).

3.1 O modelo

O modelo de crescimento econômico de Solow busca entender como o capital, a força de trabalho e a tecnologia se relacionam de maneira a determinar o acúmulo de riqueza de um país. Para os propósitos do algoritmo aqui implementado, consideraremos apenas o impacto da acumulação de capital no crescimento econômico, assumindo a força de trabalho e a tecnologia como variáveis exógenas.

Partindo do arcabouço teórico construído no modelo clássico de determinação da renda¹, a oferta e a demanda de bens e serviços permitirá a análise da acumulação de capital numa dada economia. A oferta de bens no modelo de Solow pode ser dada pela habitual função de produção $Y = F(K, L)$ ponderada pelo número de trabalhadores L , desde que supomos retornos constantes de escala. Assim, podemos reescrevê-la como

$$y = f(k), \tag{3.1}$$

em que $y = Y/L$ (produto por trabalhador), $k = K/L$ (capital por trabalhador) e $f(k) \equiv F(K/L, 1)$.

¹O modelo clássico de determinação da renda é de suma importância no estudo de modelos macroeconômicos mais sofisticados e muitas vezes é considerado como ponto de partida para o assunto. Assim, o capítulo 3 de Mankiw (2015) pode ajudar em muito na compreensão dos conceitos que serão abordados.

Já o lado da demanda é representado pela tradicional identidade da renda dividida pelo número de trabalhadores desta economia L , isto é, $y = c + i$ (supondo uma economia fechada e sem governo), em que $c = C/L$ e $i = I/L$. Adicionalmente, ao supor que as pessoas poupam uma parcela s da renda e consomem uma parcela $(1 - s)$, ou seja, $c = (1 - s)y$, com uma álgebra simples podemos reescrever a identidade da renda como

$$i = sy = sf(k), \quad (3.2)$$

sendo a última igualdade acima uma implicação de (3.1). Note, por esta equação, a relação que surge entre o estoque de capital existente, k , e a acumulação de novo capital, i .

Como ressalta Mankiw (2015), “Para qualquer estoque específico de capital, a função de produção determina a quantidade de produção gerada pela economia, e a taxa de poupança determina a distribuição dessa produção entre consumo e investimento.”

Por fim, podemos considerar a depreciação do capital como uma função linear da taxa de depreciação, isto é, δk . Assim, segue naturalmente que

$$\Delta k = sf(k) - \delta k, \quad (3.3)$$

isto é, a acumulação de capital ocorre quando o investimento supera a depreciação da economia em um dado período de tempo.

Como $f(k)$ apresenta produtividade marginal do capital decrescente, isto é, a produtividade do capital cresce a taxas decrescentes, e dado que a depreciação é uma função linear de k , ocorre que existe k^* tal que $sf(k^*) = \delta k^*$, ou seja, $\Delta k = 0$, o que configura o equilíbrio estacionário da economia no longo prazo. Esta noção deve ficar mais clara no exemplo numérico dado no algoritmo abaixo e representado graficamente.

3.2 Implementação computacional

Os mesmos pacotes `matplotlib.pyplot` e `numpy`, usados no modelo de oferta e demanda, serão necessários para este programa. As funções do modelo são definidas por

```
1 # Set the Cobb-Douglas production function:
2 def produto_agregado(k_agregado, l_agregado = 1):
3     return np.sqrt(k_agregado*l_agregado)
4
5 # Set the Cobb-Douglas production function per worker:
6 def produto_por_trabalhador(k_agregado, l_agregado = 1):
7     return produto_agregado(k_agregado)/l_agregado
8
```

```

9  # Depreciation is a linear function of k per worker, i.e.,  $k=K/L$ 
10 def depreciacao(k_agregado, l_agregado = 1, taxa_depreciacao = 0.1):
11     return taxa_depreciacao*(k_agregado/l_agregado)
12
13 def investimento(k_agregado, taxa_poupanca = 0.3):
14     return taxa_poupanca*produto_por_trabalhador(k_agregado)
15
16 def consumo(k_agregado, taxa_poupanca = 0.3):
17     return (1 - taxa_poupanca)*produto_por_trabalhador(k_agregado)
18
19 def k_variacao(k_agregado, taxa_poupanca = 0.3):
20     return (taxa_poupanca*produto_por_trabalhador(k_agregado)
21             - depreciacao(k_agregado))

```

O programa principal é dado pelo código abaixo:

```

1  #####
2  #                               Programa principal                               #
3  #####
4
5  taxa_depreciacao, taxa_poupanca, l_agregado, k_t0 = 0.1, 0.3, 1, 4
6
7  print("Faremos uma exemplo em que a taxa de poupança é", end = " ")
8  print(f"{taxa_poupanca}%, a taxa de depreciação é", end = " ")
9  print(f"{taxa_depreciacao}%, e que a economia comece com uma", end = " ")
10 print("relação de 4 capital por trabalhador")
11
12 tolerancia = 0.001
13 norma = 10 # número apenas para entrar no While
14 while norma > tolerancia:
15     k_t1 = (taxa_poupanca*produto_por_trabalhador(k_t0) +
16            k_t0*(1-taxa_depreciacao))
17     norma = abs(k_t1 - k_t0)
18     k_t0 = k_t1

```

Nele, é definido os parâmetros que descrevem um exemplo de uma economia:

$$\left\{ \begin{array}{l} \text{taxa de depreciação: } \delta = 0.1 \\ \text{taxa de poupança: } s = 0.3 \\ \text{número agregado de trabalhadores: } L = 1 \\ \text{número inicial agregado de capital: } K = 4 \end{array} \right.$$

Observe que o que interessa realmente para o modelo é a relação $k \equiv K/L$ e, assim, pouco

importa em si o número agregado de capital e trabalhador. Após a definição dos parâmetros, o programa informa ao usuário o exemplo da economia que será analisada. Em seguida, há uma iteração de modo que, dada a taxa de poupança e a depreciação, o capital (por trabalhador) no tempo $t+1$ é calculado com base no capital (por trabalhador) no tempo anterior t . Este processo segue até que a diferença absoluta entre ambos seja menor que uma tolerância (próxima a zero), que no programa foi definida por `tolerancia = 0.001`. Este é exatamente o estado de equilíbrio da economia.

Depois, o código define o gráfico das curvas de depreciação e investimento, ambas em função do capital por trabalhador:

```

1 #####
2 #                               #
3 #####
4
5 k_grid = np.linspace(0, 2*k_t1, 1000)
6
7 plt.plot(k_grid, depreciacao(k_grid), color="blue", linewidth=3.0,
8          ls = "dashed")
9 plt.plot(k_grid, investimento(k_grid), color="green", linewidth=3.0)
10 plt.xlim(0, 2*k_t1) # entre -5 e 5 por exemplo
11 plt.ylim(0, 2*investimento(k_t1))
12 plt.legend([f'Depreciação, {taxa_depreciacao}k',
13            f'Investimento, {taxa_poupanca}f(k)'])
14 plt.ylabel("Investimento e Depreciação", fontsize = 15)
15 plt.xlabel("Capital por trabalhador, k", fontsize = 15)
16 plt.show()

```

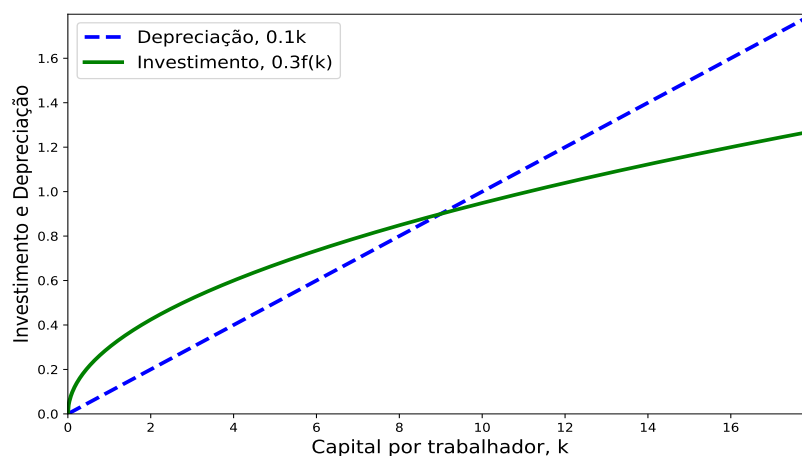


Figura 3.1: *Investimento, depreciação e estado estacionário.*

A visualização da figura (3.1) permite ver o estado de equilíbrio desta economia, descrito por um nível de capital por trabalhador $k \approx 9$. Observe como cada uma das funções se comporta. Este padrão é geral: como dissemos anteriormente, a forma das curvas fará com que em algum momento haja intersecção entre ambas, constituindo um ponto de equilíbrio estacionário entre investimento e depreciação.

Capítulo 4

Renda Nacional: o Modelo IS-LM

No contexto do desenvolvimento dos modelos macroeconômicos, o modelo IS-LM surge como uma proposta de sistematizar a teoria keynesiana desenvolvida após a famosa crise financeira de 1929. Se antes os postulados dos economistas clássicos eram geralmente aceitos como a teoria válida para explicar o funcionamento dos mercados, com a referida crise e a falha dos economistas em entendê-la satisfatoriamente, estes passaram a refletir sobre o que havia de errado nos modelos desenvolvidos até então.

Foram os economistas John Hicks e Alvin Hansen quem condensaram as ideias da teoria keynesiana no chamado modelo IS-LM, estabelecendo uma rigorosa análise macroeconômica com o formalismo matemático necessário para explicar as equações e as variáveis que descrevem a interação econômica dos agentes. Sobre a aplicação em Python, ainda é um modelo relativamente simples de ser implementado, semelhante aos anteriores. Para uma discussão mais detalhada do modelo, é sugerido a leitura de toda a parte *IV* de Mankiw (2015), em especial os capítulos 11 e 12.

4.1 O modelo

Podemos separar a explicação do modelo em duas partes referentes às curvas *investment-saving* (IS) e *liquidity-money* (LM). Em suma, a primeira nos informa as combinações de produto e taxa de juros nas quais o mercado de bens e serviços encontra-se em equilíbrio, enquanto que o segundo caso mostra as mesmas combinações mas referindo-se ao equilíbrio no mercado monetário.

No que se refere à curva IS, o modelo parte da condição de equilíbrio dada por

$$Y = C + I + G. \quad (4.1)$$

Note que estamos considerando uma economia fechada. Podemos estabelecer o consumo

como função da renda disponível (isto é, a parcela da renda depois de deduzidos os impostos), o investimento como função da taxa de juros, e a política fiscal, os gastos do governo e impostos, dada de maneira exógena. Em notação,

$$C = C(Y - T)$$

$$I = I(r)$$

$$G = \bar{G},$$

em que Y é a renda, $T = tY$ são os impostos cobrados por uma alíquota sobre a renda, r a taxa de juros, e \bar{G} os gastos do governo. Mais do que isso, podemos determinar uma forma linear para as funções de consumo e investimento. Levando em conta a teoria keynesiana, podemos escrever que

$$C = C_1(Y - T) + C_0 = C_1(Y - tY) + C_0 = C_1Y(1 - t) + C_0$$

$$I = I_0 + I_1r,$$

sendo C_0, C_1, I_0 constantes reais não negativas e I_1 um número negativo (é natural que a taxa de juros seja inversamente proporcional ao investimento já que determina seu custo).

Substituindo em (4.1) o gasto do governo e as funções acima, com alguma manipulação algébrica podemos ver que

$$Y = \frac{C_0 + I_0 + \bar{G}}{1 - C_1(1 - t)} + \frac{I_1}{1 - C_1(1 - t)}r, \quad (4.2)$$

o que determina uma relação entre o produto e a taxa de juros, isto é, nos dá a curva IS. Perceba que esta relação é negativa, isto é, o produto varia de forma inversamente proporcional à taxa de juros, já que o coeficiente de inclinação é negativo em decorrência de I_1 ser negativo e $C_1(1 - t) < 1$.

No que se refere à curva LM, como não poderia deixar de ser, a oferta e demanda monetária desempenham o papel principal na condução do mercado monetário ao equilíbrio. Consideraremos a oferta determinada pela política monetária adotada pelo governo e, assim, será exógena ao modelo. Como o nível de preços, P , também é fixo (em decorrência da fundamental hipótese dos preços rígidos), teremos uma oferta real de moeda também fixa. Já com relação ao lado da demanda, a hipótese fundamental feita pela teoria keynesiana é que a taxa de juros tem efeito direto sobre a quantidade de moeda que as pessoas optam por manter em suas carteiras e, mais que isso, ela representa o custo de manter moeda em mãos.¹ Assim, é natural que a

¹É a chamada teoria da preferência pela liquidez.

demanda monetária varie de forma inversa à taxa de juros. No entanto, além da taxa de juros, podemos considerar outros determinantes da demanda monetária. Em particular, os motivos transação e precaução nos diz que a demanda monetária é positivamente relacionada com a renda e, também, que as pessoas sempre estarão interessadas em manter uma quantidade fixa de moeda em mãos, independentemente da taxa de juros e da renda nacional.

Feitas estas considerações, podemos escrever as equações que determinam o equilíbrio no mercado monetário:

$$\begin{aligned}\left(\frac{M}{P}\right)^S &= \frac{\bar{M}}{\bar{P}} \\ \left(\frac{M}{P}\right)^D &= m_0 + m_1 Y + m_2 r,\end{aligned}$$

em que $m_0, m_1 > 0$ e $m_2 < 0$.

A condição de equilíbrio, portanto, é dada pela igualdade entre oferta e demanda monetária:

$$\frac{\bar{M}}{\bar{P}} = m_0 + m_1 Y + m_2 r.$$

Com alguma álgebra, podemos escrever que

$$Y = \left(\frac{\bar{M}}{\bar{P}} - m_0\right) \frac{1}{m_1} - \frac{m_2}{m_1} r, \quad (4.3)$$

o que novamente nos dá uma relação entre o produto, Y , e a taxa de juros, r . Como derivamos esta relação do equilíbrio no mercado monetário, temos a representação algébrica da curva LM. No entanto, note que agora esta relação é positiva, uma vez que $m_1 > 0$ e $m_2 < 0$.

Dada as equações (4.2) e (4.3), podemos encontrar a taxa de juros e o produto tais que tanto o mercado de bens e serviços quanto o mercado monetário estejam em equilíbrio. Em outras palavras, podemos encontrar o par ordenado $(r, Y) = (r^*, Y^*)$ de modo que haja a intersecção entre as curvas IS e LM . O código abaixo é justamente a implementação de um algoritmo que, uma vez fornecido o valor das variáveis exógenas, determina a taxa de juros e produto necessários para estabelecer o equilíbrio em ambos os mercados.

4.2 Implementação computacional

Começamos por definir as funções do modelo. Deixaremos as variáveis exógenas serem definidas apenas no programa principal, de forma global (e não pelos parâmetros das funções). O seguinte código implementa as funções:

```

1 #####
2 #                               Definindo funções                               #
3 #####
4
5 def investimento(taxa_juros):
6     """
7     Dada a taxa de juros, calcula o nível de investimento. Deve ser
8     negativamente relacionada com a taxa de juros. Por simplicidade,
9     consideramos o caso linear.
10    """
11
12    if a < 0 and b > 0:
13        return a*taxa_juros + b
14
15 def demanda_monetaria(taxa_juros, produto):
16     """
17     Modelamos a demanda monetária tentando incorporar linearmente os motivos
18     precaução, especulação, e transação, respectivamente pelas variáveis
19     e, c, d.
20    """
21
22    return c*produto + d*taxa_juros + e
23
24 def consumo(produto):
25     """
26     Consideraremos o consumo como uma função linear da renda disponível, de
27     modo que o coeficiente de inclinação é a propensão marginal a consumir e
28     a constante é o consumo autônomo.
29    """
30
31    return (propensao_marginal_a_consumir * (produto - aliquota_imposto)
32            + consumo_autonomo)
33
34 def produto_IS(taxa_juros):
35     """
36     Construímos a curva IS através da equação  $Y = C + I + G$ , em que  $C$  é a
37     função consumo (que depende da renda disponível),  $I$  é o investimento,
38     dependente da taxa de juros, e  $G$  são os gastos do governo. Como o consumo
39     depende do nível de renda, esta equação é manipulada até que tenhamos uma
40     expressão do produto em relação à taxa de juros
41    """
42
43    I = investimento(taxa_juros)
44
45    return ((consumo_autonomo + I + gastos_governo) /

```

```

46         (1 - propensao_marginal_a_consumir
47         * (1 - aliquota_imposto)))
48
49
50 def produto_LM(taxa_juros):
51     """
52     Construímos a curva LM através da equação
53     oferta_monetaria = demanda_monetaria = c*produto + d*taxa_juros + e,
54     em que a oferta é exógena e a demanda calculada pela função. Então, a
55     expressão pode ser manipulada até que tenhamos uma função de Y em função
56     da taxa de juros.
57     """
58
59     return ((oferta_monetaria/nivel_precos - e) / c - taxa_juros * (d / c))
60
61 def excesso_ISLM(taxa_juros):
62     """
63     Função para calcular o equilíbrio fazendo excesso_ISLM = 0, isto é,
64     o par (r*, Y*) ótimo que nos dá a intersecção entre as curvas IS e LM.
65     """
66
67     return produto_IS(taxa_juros) - produto_LM(taxa_juros)

```

As funções `investimento(taxa_juros)` e `consumo(produto)` são partes essenciais da curva IS, conforme a equação (4.1) nos mostra: junto com os gastos do governo (que são exógenos ao modelo), em equilíbrio ela determina o produto. Já a função com o nome `demanda_monetaria(taxa_juros, produto)`, junto com a oferta monetária (exógena), determina a curva LM. Por fim, as funções que implementam analiticamente as equações (4.2) e (4.3), respectivamente, são `produto_IS(taxa_juros)` e `produto_LM(taxa_juros)`. Note ainda que a função `excesso_ISLM(taxa_juros)` faz o mesmo papel que a outra função do modelo de oferta e demanda, definida por `excesso_demanda(p)`, isto é, calcula o descompasso entre as curvas IS-LM. Quando ambas se cruzarem no equilíbrio, o excesso obviamente será nulo.

O programa principal é dado pelo código abaixo. Observe que as primeiras linhas definem as variáveis exógenas ao modelo. Logo após, o usuário é informado do exemplo numérico da economia que o programa está implementando. Levando em consideração a teoria explicada anteriormente, especialmente as expressões (4.2) e (4.3), esta economia será caracterizada pelas equações mostradas abaixo. Note também que o cálculo do equilíbrio é feito numericamente através de uma busca no grid da taxa de juros, procedimento muito semelhante ao do modelo de oferta e demanda estudado anteriormente.

Consumo: $C = 0.3Y + 100$

Investimento: $I = -20r + 5$

Demanda monetária: $(M/P)^D = 10Y - 300r + 50$

Oferta monetária: $(M/P)^S = 10$

Função Consumo: $C = 0.3Y + 100$

Política fiscal: $(G, T) = (50, 0.2Y)$

```
1 #####
2 #                               Programa principal                               #
3 #####
4
5 a, b, propensao_marginal_a_consumir, consumo_autonomo = -20, 5, 0.3, 100
6 gastos_governo, aliquota_imposto, c, d, e = 50, .2, 10, -300, 50
7 nivel_precos, oferta_monetaria = 10, 100
8
9 print("\nFaremos uma exemplo em que a função investimento é", end = " ")
10 print(f"dada por {a}r + {b};", end = " ")
11 print("a função demanda monetária, por", end = " ")
12 print(f"{c}Y + {d}i + {e};", end = " ")
13 print("a função consumo, por", end = " ")
14 print(f"{propensao_marginal_a_consumir}Y + {consumo_autonomo};", end = " ")
15 print(f"e a política fiscal por G = {gastos_governo} e", end = " ")
16 print(f"T = {aliquota_imposto}Y.", end = " ")
17 print("Ainda, a oferta monetária real é dada por M/P =", end = " ")
18 print(f"{oferta_monetaria/nivel_precos}.")
19
20
21 # Calculo do equilibrio
22
23 taxa_juros_grid = np.linspace(0, 1000, 100000)
24
25 Excesso_abs = []
26
27 for i in taxa_juros_grid:
28
29     Excesso_abs.append(abs(excesso_ISLM(i)))
30
31 indice_min = Excesso_abs.index(min(Excesso_abs))
32 juros_equilibrio = taxa_juros_grid[indice_min]
33 produto_equilibrio = produto_IS(juros_equilibrio)
```

```

34
35 print("\nPortanto, a taxa de juros e produto de equilíbrio são,", end = "")
36 print(f" respect., {juros_equilibrio: .2f} e {produto_equilibrio: .2f}")

```

Com os inputs mostrados acima, o código mostra o produto e a taxa de juros que estabelece o equilíbrio simultâneo nos mercados de bens e serviços e monetário, dado pelo par $(r^*, Y^*) = (3.69, 106.84)$.

Após fazer este cálculo, o programa fornece a visualização gráfica das curvas IS e LM, dada pela figura 4.1.

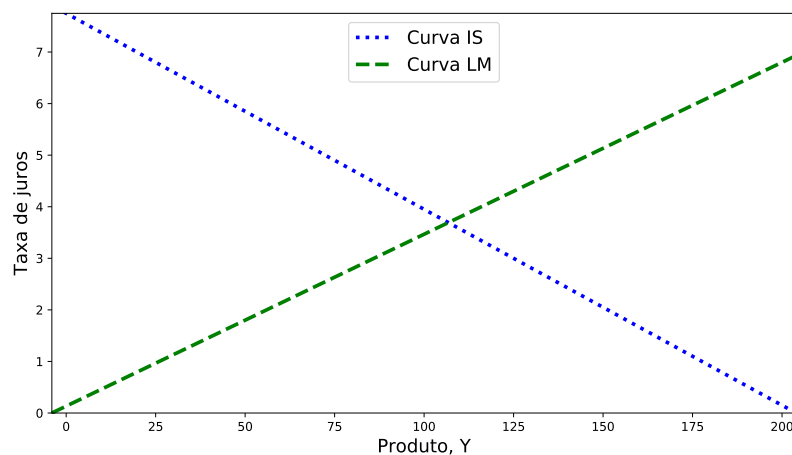


Figura 4.1: O equilíbrio simultâneo nos mercados de bens e serviços e monetário.

O código responsável por gerar este gráfico é mostrado abaixo.

```

1 #####
2 #                               Gráfico                               #
3 #####
4
5 valores_produto = []
6
7 for i in taxa_juros_grid:
8
9     valores_produto.append(abs(produto_IS(i)))
10
11 y = taxa_juros_grid[np.argmin(valores_produto)]
12
13 # O código acima foi para saber a taxa de juros (no grid) que torna a curva IS
14 # nula. Isto será útil para determinar a escala do gráfico no eixo y (vertical)
15

```

```

16 # Gráficos:
17
18 plt.plot(produto_IS(taxa_juros_grid), taxa_juros_grid, color="blue",
19          linewidth=3.0, ls = "dotted")
20 plt.plot(produto_LM(taxa_juros_grid), taxa_juros_grid, color="green",
21          linewidth=3.0, ls = "dashed")
22 plt.ylim(0, y)
23 plt.xlim(min(produto_LM(taxa_juros_grid)),
24          max(produto_IS(taxa_juros_grid)))
25 plt.legend(['Curva IS', 'Curva LM'])
26 # plt.title("Modelo IS-LM", fontsize = 20)
27 plt.ylabel("Taxa de juros", fontsize = 15)
28 plt.xlabel("Produto, Y", fontsize = 15)
29 plt.show()

```

Capítulo 5

Desemprego: *Lake Model*

O desemprego é uma das variáveis macroeconômicas mais importantes que existe. Isto ocorre porque ela afeta diretamente a renda e a vida das pessoas, além de ser, ainda, uma importante fonte de informação para realizar análises de política econômica. Por estes motivos, os economistas costumam se interessar bastante sobretudo pela *taxa de desemprego* de um país, usando modelos que tentam capturar o comportamento desta variável ao longo do tempo. Neste seção, estudaremos o modelo que ficou conhecido como *Lake Model* (modelo dos lagos).

O dinamismo do modelo faz com que o entendimento da parte teórica seja talvez um pouco mais difícil do que os anteriores. No entanto, uma vez entendida a teoria, o algoritmo em Python é relativamente simples de implementar. Para um estudo mais detalhado, os capítulos 7 de Mankiw (2015) e 53 de Sargent and Stachurski (2021) podem ser de grande utilidade.

5.1 O modelo

Consideraremos que no mercado de trabalho há sempre um fluxo de empregados perdendo seus empregos e desempregados conseguindo empregos. Adicionalmente, por simplicidade, a força de trabalho desta economia será considerada fixa (isto é, não há saída nem entrada de novos integrantes na força de trabalho). Assim, denotando por E_t e U_t o número de empregados e desempregados no tempo t , respectivamente, e $N_t = \bar{N}$ a força de trabalho no tempo t , podemos estabelecer que

$$\bar{N} = E_t + U_t$$

Fica claro, então, que alterações no número de empregados deve vir do número de desempregados, e vice-versa. Assim, é natural que as taxas de desemprego e emprego no tempo t sejam dadas por $e_t \equiv E_t/\bar{N}$ e $u_t \equiv U_t/\bar{N}$. Por fim, denote as taxas de obtenção de emprego e perda de emprego por λ e α , respectivamente.

Estamos interessados em estabelecer uma lei de movimento para a taxa de desemprego. Para isto, observe que o número de empregados em $t + 1$ é dado pela parcela de desempregados em t que obtém emprego mais a parcela de empregados em t que continuam com seus empregos. Aplicando o raciocínio semelhante para o número de desempregados, podemos estabelecer a lei de movimento para estas variáveis agregadas, isto é,

$$\begin{aligned}U_{t+1} &= (1 - \lambda)U_t + \alpha E_t \\E_{t+1} &= \lambda U_t + (1 - \alpha)E_t\end{aligned}$$

Matricialmente, se $X_t \equiv \begin{pmatrix} U_t \\ E_t \end{pmatrix}$ e $A \equiv \begin{pmatrix} 1 - \lambda & \alpha \\ \lambda & 1 - \alpha \end{pmatrix}$, então

$$X_{t+1} = AX_t \tag{5.1}$$

nos dá como o número de empregados e desempregados se comporta ao longo do tempo.

Para chegar na lei de movimento das taxas de emprego e desemprego, basta dividirmos ambos os lados de (5.1) pelo número da força de trabalho, \bar{N} , ficando com

$$\begin{pmatrix} U_{t+1}/\bar{N} \\ E_{t+1}/\bar{N} \end{pmatrix} = \begin{pmatrix} 1 - \lambda & \alpha \\ \lambda & 1 - \alpha \end{pmatrix} \begin{pmatrix} U_t/\bar{N} \\ E_t/\bar{N} \end{pmatrix}$$

Mas como $u_t = U_t/\bar{N}$ e $e_t = E_t/\bar{N}$, segue que

$$\begin{pmatrix} u_{t+1} \\ e_{t+1} \end{pmatrix} = \begin{pmatrix} 1 - \lambda & \alpha \\ \lambda & 1 - \alpha \end{pmatrix} \begin{pmatrix} u_t \\ e_t \end{pmatrix},$$

isto é, $x_{t+1} = Ax_t$, em que $x_t \equiv \begin{pmatrix} u_t \\ e_t \end{pmatrix}$.

5.2 Implementação computacional

Ao contrário do que foi feito anteriormente, no presente modelo foi definida uma função que praticamente realiza todos os cálculos numéricos para determinar o equilíbrio, uma vez fornecidos os parâmetros. Além disso, destaca-se também o uso mais elaborado da biblioteca `numpy`, sobretudo na álgebra matricial realizada. O seguinte código implementa a função principal do programa, que calcula o estado estacionário de uma economia:

```
1 def nivel_estacionario(empregados = 250_000, desempregados = 50_000,
2                       Alpha = .027, Lambda = .391):
```

```

3
4     #Definindo as variáveis com o valor dos parâmetros
5
6     forca_trabalho = empregados + desempregados
7
8     A = np.array([[1 - Lambda, Alpha    ],
9                   [Lambda    , 1 - Alpha]])
10    x_inicial = np.array([[desempregados/forca_trabalho],
11                          [empregados/forca_trabalho]])
12
13    tempo = []
14    eixo_taxa_desemprego = [x_inicial[0][0]]
15    eixo_taxa_emprego = [x_inicial[1][0]]
16    t = 0
17    tol = 1e-04
18    dif = tol + 1
19
20    # Calculando o nível estacionário
21
22    while dif > tol:
23
24        x = A @ x_inicial
25        tempo.append(t)
26        eixo_taxa_desemprego.append(x[0][0])
27        eixo_taxa_emprego.append(x[1][0])
28        dif = np.max(np.abs(x - x_inicial))
29        x_inicial = x
30        t += 1
31    tempo.append(t)
32
33    # Informando ao usuário
34
35    print(f"\nPara uma economia com {empregados} empregados,", end = " ")
36    print(f"{desempregados} desempregados, uma taxa de obtenção de", end = " ")
37    print(f"emprego de {100*Lambda: .2f}%, e uma taxa de demissão", end = " ")
38    print(f"de {100*Alpha: .2f}%, a taxa de desemprego no nível", end = " ")
39    print(f"estacionário é dada por {100*x[0][0]: .2f}% \n")
40
41    return x, tempo, eixo_taxa_desemprego, eixo_taxa_emprego

```

Observe que a função tem como padrão valores específicos para os parâmetros e, deste modo, determina um exemplo de uma economia. Com estes valores, as variáveis e as matrizes importantes que vimos na explicação do modelo teórico são definidas, e a lei de movimento pode ser usada para calcular a evolução das taxas de emprego e desemprego ao longo do tempo. Dada uma tolerância, que definimos por `tol = 1e-04`, o processo é iniciado e segue até que

a diferença da taxa de emprego (ou de desemprego) nos tempos t e $t + 1$ seja menor que a tolerância. Neste momento, a economia chega em seu estado estacionário. Por último, o usuário é informado do exemplo implementado e das variáveis resultantes do nível estacionário.

Por padrão, a função que calcula o nível estacionário do desemprego na economia considera os seguintes valores para os parâmetros do modelo:

Taxa de obtenção de emprego: $\lambda = 0.391$

Taxa de demissão: $\alpha = 0.027$

Número de empregados na economia: $E = 250.000$

Número de desempregados na economia: $U = 50.000$

O programa, com estes dados, calcula a taxa de desemprego no nível estacionário, dada por 6.4%, e logo em seguida mostra a evolução, no tempo, das taxas de emprego e desemprego da economia através de um gráfico, como mostrado na Figura 5.1.

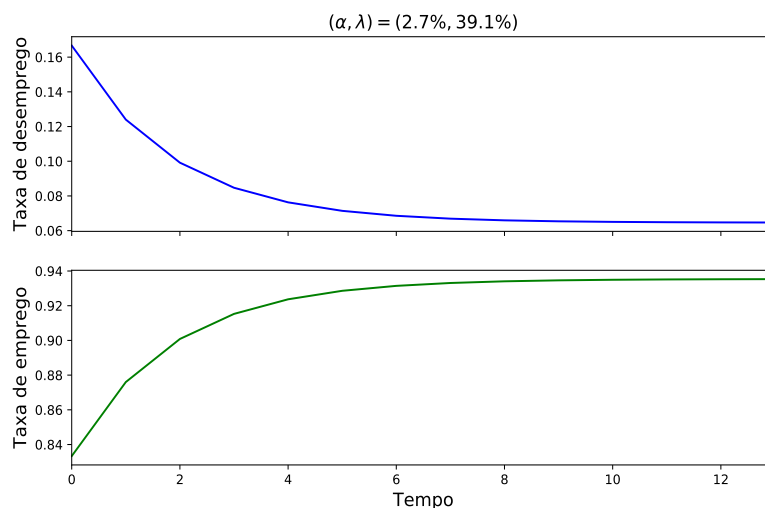


Figura 5.1: Exemplo de uma economia em seu nível estacionário de desemprego.

Poderíamos, ainda, modificar os parâmetros *taxa de obtenção de emprego* e *taxa de demissão* (*perda de emprego*), dados respectivamente por α e λ , e observar como o estado estacionário da economia se comporta ao longo do tempo, realizando assim uma estática comparativa. A Figura 5.2 mostra o comportamento das curvas neste caso.

Observe que as curvas azul e verde mostram como as taxas de emprego e desemprego variam conforme variamos a taxa de obtenção de emprego. Como era de se esperar, aumentos na taxa de obtenção de emprego, λ , faz a taxa de desemprego diminuir e a taxa de emprego, obviamente, aumentar. Já as curvas vermelha e preta mostram mudanças na taxa de emprego

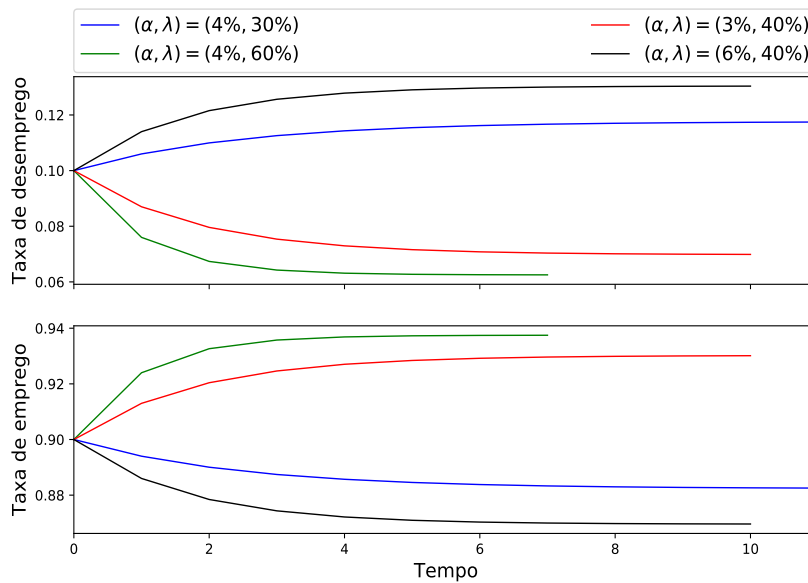


Figura 5.2: Estática comparativa das taxas de desemprego e emprego.

e desemprego ao variarmos a taxa de demissão. Novamente, é esperado que a taxa de desemprego seja maior em seu estado estacionário caso a taxa de demissão, α , for maior, enquanto que a taxa de emprego fica menor.

O código que implementa o exemplo numérico, os gráficos, e a estática comparativa é o seguinte:

```

1 #####
2 # Uma economia com: #
3 # - empregados = 250_000 #
4 # - desempregados = 50_000 #
5 # - Alpha = .027 #
6 # - Lambda = .391 #
7 #####
8
9 exemplo_padrao = nivel_estacionario()
10
11 #####
12 # Gráfico #
13 #####
14
15 fig, ax = plt.subplots(nrows = 2, ncols = 1, sharex = True)
16
17 # plot desemprego:
18
19 ax[0].plot(exemplo_padrao[1], exemplo_padrao[2], color = "blue", lw = 1.5)
20 ax[0].set_xlim(0, exemplo_padrao[1][-1])

```



```

21 ax[0].set_ylabel("Taxa de desemprego")
22 #ax[0].legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower center',
23 #             ncol=2, mode="expand", borderaxespad=0.)
24 ax[0].set_title(r'$(\alpha, \lambda) = (2.7\%, 39.1\%)$', fontsize = 10)
25
26 # plot emprego:
27
28 ax[1].plot(exemplo_padrao[1], exemplo_padrao[3], color = "green", lw = 1.5)
29 ax[1].set_xlabel("tempo")
30 ax[1].set_ylabel("Taxa de emprego")
31
32 #####
33 #             Variando parâmetros (estática comparativa)             #
34 #####
35
36 # Variando Lambda
37
38 estatica1 = nivel_estacionario(450_000, 50_000, .04, .3)
39 estatica2 = nivel_estacionario(450_000, 50_000, .04, .6)
40
41 # Variando Alpha
42
43 estatica3 = nivel_estacionario(450_000, 50_000, .03, .4)
44 estatica4 = nivel_estacionario(450_000, 50_000, .06, .4)
45
46 #####
47 #                               Gráfico                               #
48 #####
49
50 fig, ax = plt.subplots(nrows = 2, ncols = 1, sharex = True)
51
52 t_max = max(estatica1[1][-1], estatica2[1][-1], estatica3[1][-1],
53             estatica4[1][-1]) # limite eixo x
54
55 # plot desemprego:
56
57 ax[0].plot(estatica1[1], estatica1[2], color = "blue", lw = 1,
58            label = r'$(\alpha, \lambda) = (4\%, 30\%)$')
59 ax[0].plot(estatica2[1], estatica2[2], color = "green", lw = 1,
60            label = r'$(\alpha, \lambda) = (4\%, 60\%)$')
61 ax[0].plot(estatica3[1], estatica3[2], color = "red", lw = 1,
62            label = r'$(\alpha, \lambda) = (3\%, 40\%)$')
63 ax[0].plot(estatica4[1], estatica4[2], color = "black", lw = 1,
64            label = r'$(\alpha, \lambda) = (6\%, 40\%)$')
65 ax[0].set_xlim(0, t_max)
66 ax[0].set_ylabel("Taxa de desemprego")

```

```

67 ax[0].legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower center',
68               ncol=2, mode="expand", borderaxespad=0.)
69
70 # plot emprego:
71
72 ax[1].plot(estatica1[1], estatica1[3], color = "blue", lw = 1)
73 ax[1].plot(estatica2[1], estatica2[3], color = "green", lw = 1)
74 ax[1].plot(estatica3[1], estatica3[3], color = "red", lw = 1)
75 ax[1].plot(estatica4[1], estatica4[3], color = "black", lw = 1)
76 ax[1].set_xlim(0, t_max)
77 ax[1].set_xlabel("tempo")
78 ax[1].set_ylabel("Taxa de emprego")
79 plt.show()

```

5.3 Extensões do modelo

De forma complementar à nossa análise acima, poderíamos usar este modelo para acompanhar a trajetória de alguns trabalhadores desempregados até o tempo em que eles encontrem emprego. Para isto, a complementação do programa acima seria implementada adicionando algumas linhas de código. Em primeiro lugar, será necessário adicionar uma função dada por:

```

1 def individual_path(Lambda = 0.3, Alpha = .027, desempregado = True):
2     """
3
4     Parameters
5     -----
6     Lambda : float, optional
7         Taxa de obtenção de emprego. The default is 0.3.
8     Alpha : float, optional
9         Taxa de demissão. The default is .027.
10    desempregado : bool, optional
11        informa se os indivíduos estão desempregados. The default is True.
12
13    Returns
14    -----
15    tempo : list
16        lista em que a i-ésima entrada corresponde ao tempo que o i-esimo
17        trabalhador demorou pra encontrar emprego (caso esteja desempregado)
18        ou que demorou pra ficar desempregado (caso esteja empregado).
19
20    """
21

```

```

22     if desempregado:
23
24         tempo = []
25
26         j = 1
27         while j <= 1000: # amostra de 1000 individuos
28
29             encontrou_emprego = False
30             t = 0
31             while not encontrou_emprego:
32                 # enquanto a pessoa j não encontrar emprego, ela continua
33                 # procurando
34
35                 r = np.random.rand()
36
37                 if r <= Lambda: # desempregado encontra emprego
38
39                     encontrou_emprego = True
40
41                     t += 1
42
43                 tempo.append(t)
44                 j += 1
45
46     else:
47
48         tempo = []
49
50         j = 1
51         while j <= 1000: # amostra de 1000 individuos
52
53             ficou_desempregado = False
54             t = 0
55             while not ficou_desempregado: # fica no emprego até ser demitido
56
57                 r = np.random.rand()
58
59                 if r <= Alpha: # empregado ficou desempregado
60
61                     ficou_desempregado = True
62                     t += 1
63
64                 t += 1
65
66             tempo.append(t)
67             j += 1

```

68

69

```
return tempo
```

Depois de definida a função que retornará uma lista informando ou o tempo em que cada trabalhador desempregado encontrou emprego ou o tempo em que cada trabalhador empregado perdeu o emprego, podemos exibir um histograma informando a frequência de trabalhadores (numa amostra de mil, como pode ser observado no código) que encontra emprego ou desemprego (novamente, a depender de seu estado inicial) no tempo $t, \forall t \in \{1, 2, \dots\}$.

Por questões de simplicidade, estudaremos apenas o caso em que os trabalhadores começam desempregados, isto é, focaremos no tempo em que estes trabalhadores levam para encontrar um emprego e depois observaremos esta distribuição gerando um histograma. Assim, depois de calculado o vetor *tempo* na função acima, o histograma é feito através do seguinte código:

```
1 exemplo_padrao2 = individual_path()
2
3 aux = max(exemplo_padrao2)
4 hist, valores = np.histogram(exemplo_padrao2, bins = np.arange(0, aux))
5 plt.plot(valores[:-1], hist, color="blue", linewidth=1)
6 plt.ylabel("Trabalhadores", fontsize = 10)
7 plt.xlabel("Tempo até encontrar emprego", fontsize = 10)
8 plt.show()
```

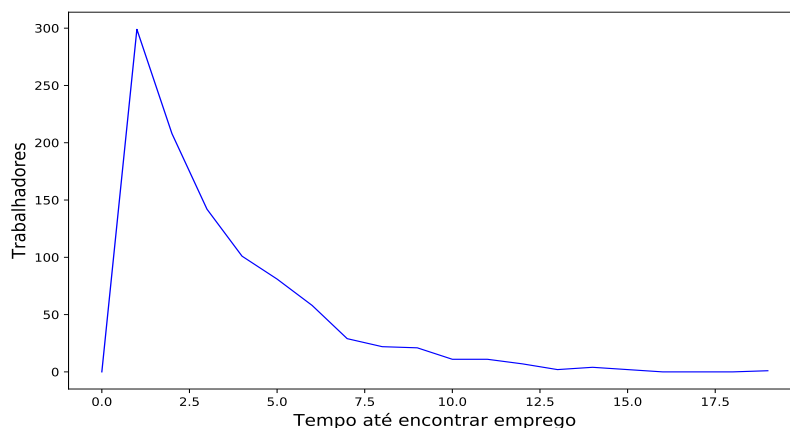


Figura 5.3: *Frequência dos trabalhadores encontrando emprego no tempo.*

Depois de rodar o código, podemos ver a forma do histograma através da figura 5.3. Por último, podemos realizar, assim como fizemos anteriormente, a estática comparativa do histograma ao variar a taxa de obtenção de emprego, λ . Isto é feito pelo código abaixo. Como era de

se esperar, uma maior taxa de obtenção de emprego aumenta a quantidade de trabalhadores que conseguem emprego em menor tempo.

```
1 exemplo_padrao3 = individual_path(Lambda = 0.2)
2 exemplo_padrao4 = individual_path(Lambda = 0.4)
3 exemplo_padrao5 = individual_path(Lambda = 0.65)
4 exemplo_padrao6 = individual_path(Lambda = 0.9)
5
6 aux = max(max(exemplo_padrao2), max(exemplo_padrao3),
7          max(exemplo_padrao4), max(exemplo_padrao5))
8
9 hist2, valores2 = np.histogram(exemplo_padrao3, bins = np.arange(0, aux))
10 hist3, valores3 = np.histogram(exemplo_padrao4, bins = np.arange(0, aux))
11 hist4, valores4 = np.histogram(exemplo_padrao5, bins = np.arange(0, aux))
12 hist5, valores5 = np.histogram(exemplo_padrao6, bins = np.arange(0, aux))
13
14 plt.plot(valores2[:-1], hist2, color="black", linewidth=1)
15 plt.plot(valores3[:-1], hist3, color="green", linewidth=1)
16 plt.plot(valores4[:-1], hist4, color="red", linewidth=1)
17 plt.plot(valores5[:-1], hist5, color="orange", linewidth=1)
18 plt.legend([r'\lambda = 10\%', r'\lambda = 40\%', r'\lambda = 70\%',
19            r'\lambda = 90\%'])
20 plt.ylabel("Trabalhadores", fontsize = 10)
21 plt.xlabel("Tempo até encontrar emprego", fontsize = 10)
22 plt.show()
```

Este código gerará os histogramas da figura 5.4.

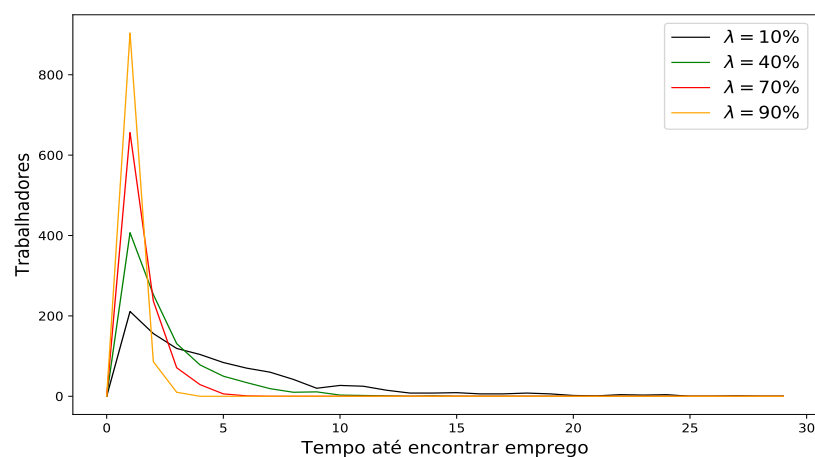


Figura 5.4: Estática comparativa: frequência de trabalhadores encontrando emprego no tempo.

Capítulo 6

Escolha Intertemporal

Frequentemente, as pessoas tomam decisões sobre quanto de sua renda elas devem consumir ao longo do tempo e quanto elas devem poupar para os próximos períodos. Neste contexto, os economistas estão interessados em determinar qual a distribuição ótima de consumo no tempo para um agente representativo através de um modelo matemático. A este problema de decisão damos o nome de *escolha intertemporal*.

Para um tratamento mais detalhado do assunto, o capítulo 10 de Varian (2012) e 16, seção 2, de Mankiw (2015) são boas referências. Se entendido o modelo, a implementação em Python torna-se simples. Talvez, a maior dificuldade é plotar os gráficos de forma a fazer os exemplos ficarem didáticos e bem entendidos.

6.1 O modelo

Suponha, primeiramente, a existência de uma economia com apenas dois períodos, t_1 e t_2 , e um consumidor que recebe uma dotação m_1 para gastar em t_1 e m_2 para gastar em t_2 . O agente representativo deve decidir a quantidade a ser consumida em t_1 e, assim, a quantidade de recursos poupada (ou não) para ser consumida em t_2 . Representa-se a distribuição de consumo no tempo pelo par ordenado (c_1, c_2) . Na ausência de um mercado de crédito, claramente $c_1 \in [0, m_1]$. Então, no segundo período temos que

$$c_2 = m_2 + (m_1 - c_1)$$

Portanto, ou $c_1 = m_1$ e o consumidor consome toda a dotação m_2 de t_2 neste último período (já que não há outros períodos da economia e, assim, não tem por que realizar poupança em t_2) ou $c_1 < m_1$ e então o consumidor consome c_2 uma quantidade maior do que sua dotação m_2 em t_2 .

Suponha, agora, a existência de um mercado de crédito nesta economia. É possível consu-

mir, no primeiro período, uma parte da renda superior a sua dotação inicial, bastando pegar um empréstimo a uma taxa de juros real, r . Portanto, escolhido o consumo no primeiro período, c_1 , o consumo no segundo período será dado por

$$c_2 = m_2 + (1 + r)(m_1 - c_1) \quad (6.1)$$

Se $m_1 = c_1$, então $m_2 = c_2$ (pelo mesmo motivo anterior); se $m_1 < c_1$, o indivíduo é um tomador de empréstimo e, assim, por (6.1), $c_2 < m_2$; por fim, se $m_1 > c_1$, o indivíduo é um poupador de recurso e, conseqüentemente, $c_2 > m_2$.

Observe que (6.1) é a equação da restrição orçamentária do indivíduo. Para ficar mais claro, note que ela é equivalente à expressão dada por

$$c_1 + \frac{c_2}{1 + r} = m_1 + \frac{m_2}{1 + r},$$

que representa a mesma restrição orçamentária mas a preços do primeiro período (isto é, em termos de valor presente). Poderíamos, ainda, multiplicar ambos os lados desta equação por $(1 + r)$ de modo a obter a restrição orçamentária em termos de valor futuro.

Uma vez derivada a restrição orçamentária, as preferências de consumo ao longo do tempo é representada por uma função utilidade. Como sempre, estaremos interessados no caso em que esta função é bem-comportada, no sentido de apresentar sobretudo a propriedade de convexidade. Assim, o consumidor sempre preferirá uma quantidade “média” de consumo em cada período a ter muito consumo em um só período e muito pouco no outro. Para os propósitos do algoritmo desenvolvido para este modelo, adotaremos a função com aversão ao risco relativo constante (CRRA), dada por

$$\begin{cases} u(c) = \frac{c^{1-\gamma}-1}{1-\gamma}, & \text{se } \gamma \neq 1 \\ \ln c, & \text{se } \gamma = 1, \end{cases}$$

em que $\gamma > 0$.

Consideraremos que o indivíduo estará interessado em escolher (c_1, c_2) de tal modo a maximizar a sua função utilidade. Se $U(c_1, c_2) \equiv u(c_1) + u(c_2)$, então o problema do consumidor pode ser representado algebricamente por

$$\begin{cases} \max_{c_1, c_2} & U(c_1, c_2) \\ \text{s.a.} & c_1 + \frac{c_2}{1+r} = m_1 + \frac{m_2}{1+r} \end{cases}$$

Com alguma álgebra, podemos encontrar que

$$(c_1^*, c_2^*) = \left(\frac{m_1(1+r) + m_2}{(1+r)[1 + (1+r)^{(1-\gamma)/\gamma}]}, (1+r)(m_1 - c_1^*) + m_2 \right), \quad (6.2)$$

o que dá a distribuição ótima de consumo nos tempos t_1 e t_2 .

6.2 Implementação computacional

As funções abaixo implementam as utilidades e a restrição orçamentária do consumidor. Além disso, a função `choice(a,b,r,m1,m2)` estabelece analiticamente o par (6.2), a escolha ótima.

```

1 #####
2 #                                     Funções                                     #
3 #####
4
5 def u(c, a):
6     if a > 0 and a != 1:
7         return (c**(1-a) - 1) / (1 - a)
8     else:
9         if a == 0:
10            return np.log(c)
11
12 def U(c1, c2, a, b):
13     return u(c1, a) + u(c2, b)
14
15 def budget(c1, m1, m2, r):
16     if m1 > 0 and m2 > 0 and r > 0 and r < 1:
17         return (1+r)*m1 + m2 - (1+r)*c1
18
19 def inverse_budget(c2, m1, m2, r):
20     if m1 > 0 and m2 > 0 and r > 0 and r < 1:
21         return (m2-c2)/(1+r) + m1
22
23 def choice(a, b, r, m1, m2):
24     if (a > 0 and a != 1 and b > 0 and b != 1 and m1 > 0 and m2 > 0 and r > 0
25         and r < 1):
26         c1_estrela = (m1+m2/(1+r)) / (1+(1+r)**((1-a)/a))
27         c2_estrela = (m1 - c1_estrela)*(1+r) + m2
28         funcao_valor = U(c1_estrela, c2_estrela, a, b)
29
30     return c1_estrela, c2_estrela, funcao_valor

```


Em seguida, um exemplo de economia é implementado pelo código

```
1 a, b, r, m1, m2 = 0.5, 0.5, 0.5, 50, 750
2 exemplo1 = choice(a, b, r, m1, m2)
```

A economia é caracterizada por uma dotação inicial $(m_1, m_2) = (50, 750)$, taxa de juros $r = 0.5$, e função de utilidade dada por

$$U(c_1, c_2) = u(c_1) + u(c_2) = \frac{c_1^{1/2} - 1}{1/2} + \frac{c_2^{1/2} - 1}{1/2} = 2(\sqrt{c_1} + \sqrt{c_2}) - 4.$$

A restrição orçamentária para esta dotação inicial é dada pela curva da Figura (6.1). Note que o ponto A está representando a dotação inicial.

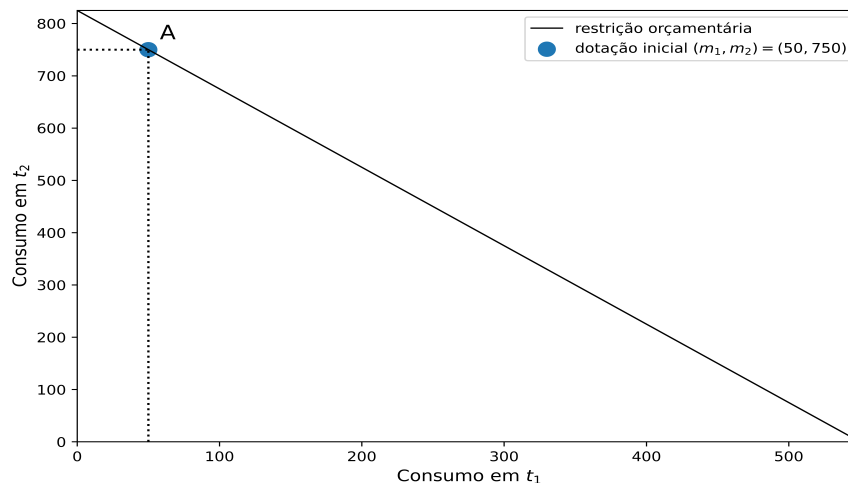


Figura 6.1: Restrição orçamentária.

Ainda, qualquer ponto sobre a reta orçamentária à direita do ponto A significa que o indivíduo está tomando um empréstimo (porque $c_1 > m_1$) e qualquer ponto sobre a reta orçamentária à esquerda de A significa que o indivíduo é um poupador (porque $c_1 < m_1$).

Rodando o programa para este exemplo de economia, temos que o consumo ótimo será dado pelo par

$$(c_1^*, c_2^*) = (220, 495), \tag{6.3}$$

com nível de utilidade $U \approx 70$. O programa mostra estes resultados por escrito. No entanto, estas informações podem ser melhor visualizadas através do gráfico que mostra a reta orçamentária tangenciando a curva de indiferença do consumidor (a condição necessária para a solução

do problema), e o código, ao final, faz exatamente isto. A Figura (6.2) mostra o gráfico exibido pelo programa.

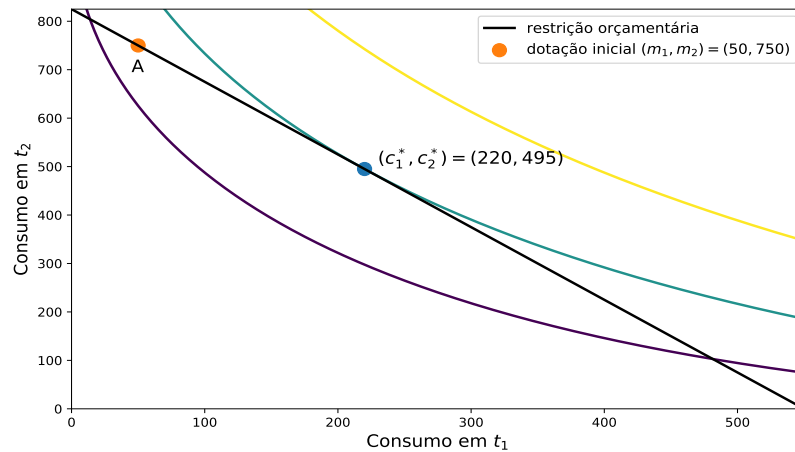


Figura 6.2: Consumo ótimo para um exemplo de economia.

O próximo passo do nosso algoritmo é verificar como a escolha ótima altera-se conforme variamos a taxa de juros do mercado de crédito. O seguinte código faz isso:

```

1  # Outros exemplos
2  a2, b2, r2, m1_2, m2_2 = 0.5, 0.5, 0.1, 50, 750
3  a3, b3, r3, m1_3, m2_3 = 0.5, 0.5, 0.9, 50, 750
4  exemplo2 = choice(a2, b2, r2, m1_2, m2_2)
5  exemplo3 = choice(a3, b3, r3, m1_3, m2_3)
6
7  # configuração exemplo1
8  budget1 = budget(eixo_x, m1, m2, r)
9
10 # configuração exemplo2
11 budget2 = budget(eixo_x, m1_2, m2_2, r2)
12
13 # configuração exemplo3
14 budget3 = budget(eixo_x, m1_3, m2_3, r3)

```

Note que o indivíduo no nosso exemplo é um tomador de empréstimo. Então, espera-se que aumentos na taxa de juros faça com que ele diminua o consumo no primeiro período em detrimento de maior consumo no segundo. Com lógica semelhante, diminuições na taxa de juros faria com que ele aumentasse o consumo no primeiro período (se endividasse mais) em detrimento de um menor consumo no segundo.

De fato, é o que ocorre. O programa, ao final, mostra como varia seu consumo no tempo e

seu respectivo nível de utilidade. Ao diminuirmos a taxa de juros para $r = 0.1$, o novo consumo ótimo será dado pelo par $(c_1^*, c_2^*) = (349, 422)$, com $U \approx 74$; ao aumentarmos para $r = 0.9$, a nova distribuição de consumo ótima é $(c_1^*, c_2^*) = (153, 554)$, com $U \approx 67$. A variação do nível de utilidade confirma outro fato: como o consumidor é um tomador de empréstimo, seu nível de utilidade altera-se na direção contrária à variação da taxa de juros. Toda esta análise é condensada na Figura (6.3).

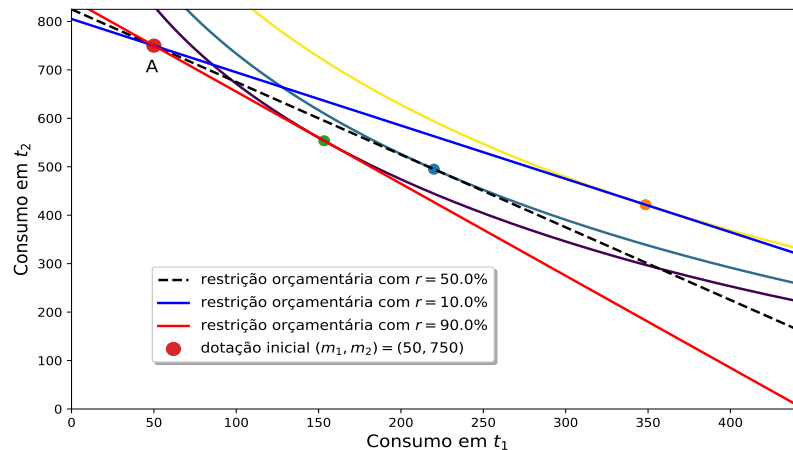


Figura 6.3: Consumo ótimo para diferentes taxas de juros.

Os gráficos (6.1) e (6.2) são implementados pelo código abaixo.

```

1 plt.rcParams["figure.figsize"] = (11, 6)
2 plt.rcParams["lines.linewidth"] = (2.5)
3
4 # plotando exemplo1 (mesmo que anterior)
5 Z1 = plt.contour(X, Y, U(X, Y, a, b), levels = [exemplo3[2], exemplo1[2],
6                                                exemplo2[2]], linewidths = 2)
7 plt.plot(eixo_x, budget1, color = "black", ls = "dashed", lw = 2,
8          label = f"restrição orçamentária com $r = {100*r}\%$")
9
10 plt.rcParams["figure.figsize"] = (11, 6)
11 plt.rcParams["lines.linewidth"] = (2.5)
12
13 # plotando exemplo2
14 plt.plot(eixo_x, budget2, color = "blue", lw = 2,
15          label = f"restrição orçamentária com $r = {100*r2}\%$")
16
17 # plotando exemplo3
18 plt.plot(eixo_x, budget3, color = "red", lw = 2,
19          label = f"restrição orçamentária com $r = {100*r3}\%$")

```

```

20
21 # Marcando pontos na tangência da reta orçamentária com as curvas de utilidade
22 plt.scatter([exemplo1[0]], [exemplo1[1]], s = 4*4**2)
23 plt.scatter([exemplo2[0]], [exemplo2[1]], s = 4*4**2)
24 plt.scatter([exemplo3[0]], [exemplo3[1]], s = 4*4**2)
25 plt.scatter([m1],[m2], s = 7*4**2,
26             label = f"dotação inicial $(m_1, m_2) = ({m1},{m2})$")
27
28 plt.annotate("A", (m1-5,m2-55), fontsize = 12)
29
30 # configuração geral plot
31 plt.legend(loc = 'lower left', bbox_to_anchor=(0.1, 0.1),
32           fancybox = True, shadow = True, fontsize = 11.5)
33 plt.ylim(0, (1+r)*m1 + m2)
34 plt.xlim(0, m1_3 + m2_3/(1+r3))
35 plt.ylabel(r"Consumo em $t_2$", fontsize = 12)
36 plt.xlabel(r"Consumo em $t_1$", fontsize = 12)
37 plt.show()

```


Capítulo 7

Escolha sob incerteza

Um dos principais objetivos dos economistas é entender o comportamento individual do consumidor diante de algumas situações específicas. Em teoria microeconômica, costuma-se estudar como o indivíduo com uma restrição orçamentária e preferências bem definidas faz suas escolhas no mercado de bens. Num cenário um pouco diferente, em que o consumidor preocupa-se com o quanto consumir em diferentes estados do tempo, novamente a teoria oferece um modelo para analisar a escolha ótima de consumo presente e futuro (ou poupança), como foi feito na seção anterior sobre escolha intertemporal.

No entanto, em nenhum dos modelos citados a presença de *risco* foi incorporada. Aqui, no modelo de *escolha sob incerteza*, estaremos interessados em entender como os indivíduos tomam decisões diante do risco que eles têm em perder parte do consumo futuro. Particularmente, estudaremos cenários em que este consumo futuro (que representaremos, por motivos de simplificação, pela riqueza do consumidor) *pode* (i.e., com alguma probabilidade) ser afetado. Para isto, um mercado de seguros será disponibilizado, de modo que o indivíduo possa alterar sua distribuição de consumo entre os estados contingentes.

Sem dúvida, do que foi visto até aqui, este é o modelo teórico mais sofisticado e, por isso, mais difícil de ser compreendido. Talvez, a abordagem presente no capítulo 12 de Varian (2012) não seja a mais indicada e, por isso, o leitor pode preferir acompanhar o capítulo 7, parte 3, de Nicholson and Snyder (2017). Entretanto, para manter o padrão de seguir a referência Varian (2012), desenvolveremos computacionalmente o modelo presente neste livro.

O leitor com um pouco mais de familiaridade em microeconomia observará que o problema *matemático* descrito é muito semelhante aos outros modelos de escolha do consumidor, mas a interpretação econômica apresenta diferenças substanciais. No entanto, uma vez entendido o modelo teórico, a implementação em Python não é difícil de ser entendida.

7.1 O modelo

Como dito, o modelo estudará a escolha de consumo do consumidor em ambientes de incerteza, isto é, com diferentes estados da natureza apresentando determinada distribuição de probabilidade. Para efeitos de simplicidade, o consumo será representado pelo estoque de ativos do indivíduo, que chamaremos de *riqueza*, num determinado período de tempo. Ainda, serão considerados apenas dois estados da natureza: um estado em que o consumidor tem parte de sua riqueza corroída (por exemplo: sua casa é queimada ou seu carro é danificado) e outro em que nada ocorre, isto é, sua riqueza permanece inalterada. Denotaremos o consumo no primeiro estado por c_b e o consumo no segundo estado por c_g . Adicionalmente, um mercado de seguros estará disponível ao consumidor. Assim, o indivíduo tem a escolha de contratar K unidades monetárias de seguro ao preço de γK , para $\gamma \in (0, 1)$.

Suponha que o consumidor tenha uma riqueza inicial x e que o estado ruim afete seu patrimônio em L unidades. Caso contrate o seguro, o indivíduo terá $x - \gamma K$ de consumo no estado bom ($c_g = x - \gamma K$), isto é, sua riqueza não é corroída e seu custo reside apenas no pagamento do prêmio do seguro, e $x - L + K - \gamma K$ no estado ruim em que sua riqueza é afetada por uma perda L mas também, por ter contratado o seguro, por um ganho $K - \gamma K$ (ou seja, $c_b = x - L + K - \gamma K$). A figura 7.1 ilustra este raciocínio.

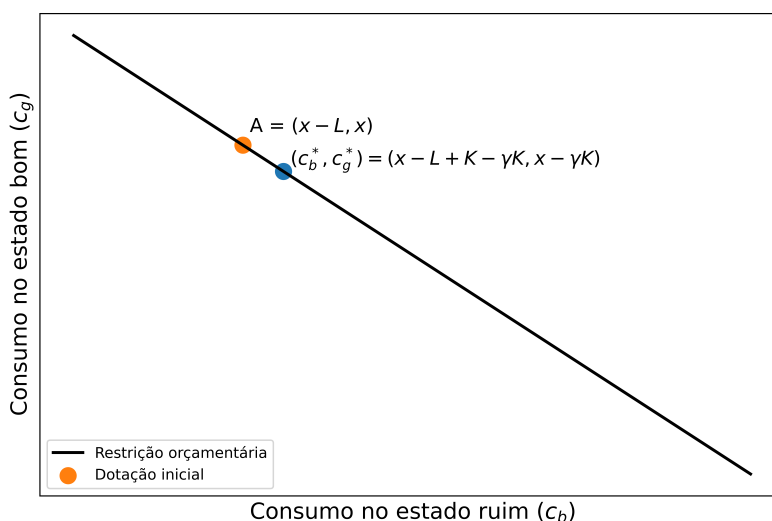


Figura 7.1: A restrição orçamentária.

Podemos descrever algebricamente a reta orçamentária. Através dos dois pontos dados, podemos ver que a inclinação da reta é

$$\frac{(x - (x - \gamma K))}{(x - L) - (x - L + K - \gamma K)} = \frac{\gamma K}{\gamma K - K} = \frac{-\gamma}{1 - \gamma}.$$

Assim, como sabemos as coordenadas do ponto de dotação inicial e sabemos a inclinação, podemos achar o intercepto do eixo vertical, que será dado por $b \equiv x + \frac{\gamma}{(1-\gamma)}(x - L)$. Então, a reta orçamentária pode ser descrita pela equação

$$c_g = -\frac{\gamma}{1-\gamma}c_b + b. \quad (7.1)$$

Quanto de seguro a pessoa contratará? Certamente, isto dependerá das *preferências* dela com respeito à distribuição de consumo entre os estados bom e ruim. Como de costume, mapeamos as preferências do consumidor de modo que as curvas de nível sejam convexas, e o ponto de ótimo será dado pela tangência entre a restrição orçamentária (7.1) e a curva de indiferença. Ou seja,

$$TMS = \frac{UMg_{c_b}}{UMg_{c_g}} = \frac{-\gamma}{1-\gamma}. \quad (7.2)$$

Podemos utilizar, como é de costume em escolha sob incerteza, a função de utilidade Neumann-Morgenstern (ou função de utilidade esperada), dada por

$$U(c_b, c_g) = \pi_b u(c_b) + \pi_g u(c_g), \quad (7.3)$$

em que π_i é a probabilidade de ocorrer o estado $i \in \{b, g\}$. Mais especificamente, utilizaremos $u(c) = \ln(c)$.

Com esta forma funcional mapeando as preferências do consumidor, temos que a condição (7.2) fica da seguinte forma

$$\frac{-\gamma}{1-\gamma} = \frac{-\pi_b c_g}{\pi_g c_b} \implies c_g^* = \frac{\pi_g}{\pi_b} \frac{\gamma}{1-\gamma} c_b^*. \quad (7.4)$$

Substituindo o valor de c_g^* em (7.4) na restrição orçamentária em (7.1), temos que:

$$b = c_b^* \frac{\pi_g}{\pi_b} \frac{\gamma}{1-\gamma} + c_b^* \frac{\gamma}{1-\gamma} \implies c_b^* = \pi_b b \frac{(1-\gamma)}{\gamma}, \quad (7.5)$$

em que $b = x - \frac{\gamma}{(1-\gamma)}(x - L)$. Assim, a partir de (7.5), (7.4) assume a forma $c_g^* = \pi_g b$.

Portanto, segue que a escolha ótima é dada por

$$(c_b^*, c_g^*) = \left(\frac{(1-\gamma)}{\gamma} \pi_b b, \pi_g b \right). \quad (7.6)$$

Esta solução analítica será útil na descrição do programa.

7.2 Implementação computacional

As funções úteis ao modelo são implementadas pelo código abaixo:

```
1 def utilidade(c_estado_b, c_estado_g, prob_estado_b, prob_estado_g):
2
3     return prob_estado_b*np.log(c_estado_b) + prob_estado_g*np.log(c_estado_g)
4
5 def restr_orcamentaria(c_estado_b, gamma, dot_estado_b, dot_estado_g):
6
7     b = dot_estado_g + dot_estado_b*gamma/(1-gamma)
8
9     return b - (gamma*c_estado_b)/(1-gamma)
10
11 def escolha(dot_estado_b, dot_estado_g, prob_estado_b, prob_estado_g, gamma):
12
13     b = dot_estado_g + dot_estado_b*gamma/(1-gamma)
14     c = prob_estado_g/prob_estado_b
15
16     c_b_estrela = b*prob_estado_b*(1-gamma)/gamma
17     c_g_estrela = c_b_estrela*c*(gamma/(1-gamma))
18     funcao_valor = utilidade(c_b_estrela, c_g_estrela, prob_estado_b,
19                             prob_estado_g)
20
21     return c_b_estrela, c_g_estrela, funcao_valor
22
23 def seguro_otimo(dot_estado_b, dot_estado_g, prob_estado_b, prob_estado_g,
24                 gamma):
25
26     escolha_otima = escolha(dot_estado_b, dot_estado_g, prob_estado_b,
27                             prob_estado_g, gamma)
28
29     return (dot_estado_g - escolha_otima[1]) / gamma
```

As três primeiras funções nada mais são que as funções de utilidade, restrição orçamentária e de escolha ótima representadas algebricamente pelas equações (7.3), (7.1) e (7.6), respectivamente. A função `seguro_otimo`, dada a escolha ótima, retorna a quantidade ótima de seguro contratada pelo consumidor.

Em seguida, o código implementa um exemplo de economia:

```
1 dot_estado_b1, dot_estado_g1 = 250, 350
2 gamma_1 = 0.4
3 prob_estado_b1, prob_estado_g1 = 0.4, 0.60
```

```

4
5 exemplo_1 = escolha(dot_estado_b1, dot_estado_g1, prob_estado_b1,
6                     prob_estado_g1, gamma_1)
7
8 seguro_estrela = seguro_otimo(dot_estado_b1, dot_estado_g1, prob_estado_b1,
9                               prob_estado_g1, gamma_1)

```

É considerada uma dotação inicial $(c_b^{dot}, c_g^{dot}) = (250, 350)$, um prêmio $\gamma = 0.4$, e uma distribuição de probabilidade $(\pi_b, \pi_g) = (0.4, 0.6)$. Para esta economia, a distribuição de consumo ótima será dada por

$$(c_b^*, c_g^*) = (310, 310).$$

Ainda, o programa fornece a quantidade ótima de seguro para este exemplo, dada por $K^* = 100$. Observe que o consumidor preferiu eliminar completamente o risco de corrosão de sua riqueza, distribuindo o consumo de forma equivalente entre os estados. De fato, isto não é uma coincidência: se o prêmio do seguro é igual à probabilidade de ocorrência do estado ruim, isto é, $\gamma = \pi_b$, então dizemos que o preço do seguro é justo e, assim, o consumidor avesso ao risco comprará seguro na exata quantidade para eliminar o risco.

Por fim, o programa mostra graficamente a escolha ótima de consumo nos estados, que corresponde, como sempre, à tangência entre a restrição orçamentária e a inclinação da curva de indiferença. A Figura (7.2) mostra tal gráfico gerado pelo código com o nível de utilidade alcançado pelo consumidor.

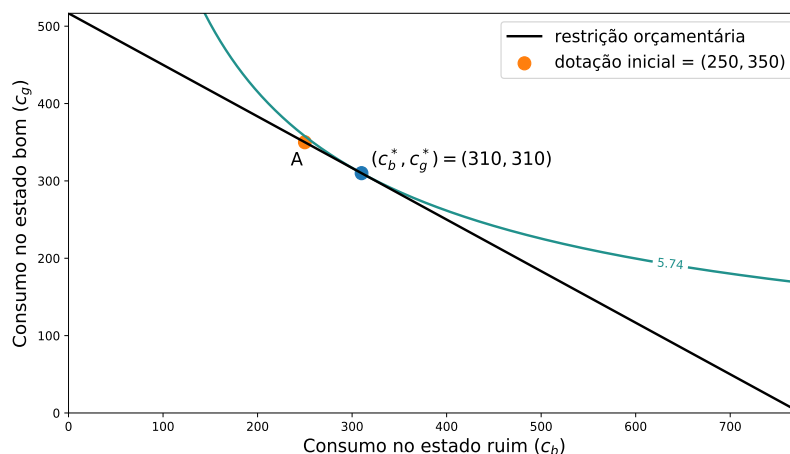


Figura 7.2: Escolha ótima do consumidor para um exemplo de economia.

O código que gera o gráfico é dado por:

```

1 c_estado_b1 = np.linspace(0.01, 1000, 1000)
2 c_estado_g1 = np.linspace(0.01, 1000, 1000)
3 orcamento = restr_orcamentaria(c_estado_b1, gamma_1, dot_estado_b1,
4                                 dot_estado_g1)
5 X, Y = np.meshgrid(c_estado_b1, c_estado_g1)
6
7 Z = plt.contour(X, Y, utilidade(X, Y, prob_estado_b1, prob_estado_g1),
8                 levels = [exemplo_1[2]-10, exemplo_1[2], exemplo_1[2]+10],
9                 linewidths = 2)
10 plt.plot(c_estado_b1, orcamento, color = "black", lw = 2,
11          label = "restrição orçamentária")
12 plt.clabel(Z, inline = 1, fontsize = 10)
13 plt.ylim(0, dot_estado_g1 + dot_estado_b1*gamma_1/(1-gamma_1))
14 plt.xlim(0, (1-gamma_1)*(dot_estado_g1 + dot_estado_b1*gamma_1/(1-gamma_1))
15          /gamma_1)
16 plt.ylabel(r"Consumo no estado bom ($c_g$)", fontsize = 15)
17 plt.xlabel(r"Consumo no estado ruim ($c_b$)", fontsize = 15)
18 plt.scatter([exemplo_1[0]], [exemplo_1[1]], s = 7*4**2)
19 plt.annotate(f"$c_b^*, c_g^* = ({exemplo_1[0]: .0f}, {exemplo_1[1]: .0f})$",
20             (exemplo_1[0]+10, exemplo_1[1]+10), fontsize = 13)
21 plt.scatter([dot_estado_b1], [dot_estado_g1], s = 7*4**2,
22            label = f"dotação inicial = $({dot_estado_b1}, {dot_estado_g1})$")
23 plt.annotate("A", (dot_estado_b1-15, dot_estado_g1-30), fontsize = 13)
24 plt.legend()
25 plt.show()

```

Capítulo 8

Stag Hunt Game (Rousseau)

Uma das principais áreas da teoria econômica moderna é a que estuda a interação estratégica entre os agentes econômicos, chamada Teoria dos Jogos. Pela complexidade do tema, não será possível explicar todos os conceitos necessários ao entendimento desta teoria. No entanto, algumas ideias necessárias ao entendimento do jogo *Stag Hunt* deverão ser tratadas. Para um estudo mais aprofundado do tema, ver Varian (2012), capítulos 28 e 29, ou Tadelis (2013) para uma abordagem ainda mais detalhada. Uma vez entendido o jogo e especialmente o seu conceito de *equilíbrio*, a implementação computacional torna-se fácil de ser entendida.

8.1 O modelo

De maneira simplificada, um jogo é definido por um conjunto de estratégias (ou ações) disponível aos jogadores e seus respectivos ganhos (que frequentemente chamaremos de *payoffs*). Como é de se esperar de uma interação estratégia entre indivíduos, estes *payoffs* dependem não somente da estratégia adotada por um jogador mas também das ações escolhidas pelos demais jogadores. Por simplicidade, assumiremos um número finito de estratégias e apenas dois jogadores.

É comum representarmos um jogo através de uma matriz de *payoffs*, indicando os ganhos dos jogadores para cada estado do jogo. Examinemos um exemplo. Suponha que dois cúmplices de um crime são pegos pela polícia com provas suficientes para incriminá-los e sentenciá-los a 1 ano de cadeia. A eles é dada a oportunidade de delatar o parceiro e, assim, ficar livre caso o outro não delate ou, caso ambos delatarem um ao outro, pegar 2 anos de prisão. Perceba que temos duas ações para cada jogador: delatar ou não delatar. Denote tais estratégias, respectivamente, por D e ND . Este jogo é o famoso Dilema dos Prisioneiros. Podemos representá-lo pela matriz de *payoffs* dada pela tabela 8.1.

Note que, por padrão, a primeira entrada do vetor dos *payoffs* representa os ganhos (ou perdas) do jogador 1, enquanto que a segunda entrada representa os ganhos (ou perdas) do

jogador 2. Assim, podemos ver na matriz o que foi dito acima: se ambos delatam, cada um pega 2 anos de prisão; se ambos não delatam, eles recebem 1 ano de prisão cada um; por fim, se um delatar e o outro não, o que delatou fica livre e o criminoso que não delatou recebe a pena máxima de 3 anos de cadeia.

		Jogador 2	
		<i>D</i>	<i>ND</i>
Jogador 1	<i>D</i>	(-2,-2)	(0,-3)
	<i>ND</i>	(-3,0)	(-1,-1)

Tabela 8.1: Matriz de *payoffs* do jogo Dilema dos Prisioneiros.

Alguém poderia perguntar: qual seria a melhor tomada de decisão para cada jogador, supondo-se que eles possuem crenças defensáveis sobre a estratégia que o outro jogador deve adotar? Para responder a esta pergunta, observe primeiramente que, do ponto de vista de um jogador qualquer, não delatar não é uma boa escolha, visto que independentemente da ação do “adversário”, o jogador sempre tem a ganhar mudando sua estratégia para delatar, recebendo menor tempo de prisão. Então, sabendo disso, a crença defensável de ambos os jogadores é supor que o oponente irá delatar e, assim, a melhor resposta de cada um é sempre delatar. Portanto, a situação em que ambos delatam constitui um *equilíbrio de Nash*. Em geral, o equilíbrio de Nash define-se pela situação em que ambos os jogadores escolhem a *melhor* estratégia, *dada* a escolha de ação do oponente.

Existem diversos tipos de jogos que podem ser estudados, mas em especial estaremos interessados no jogo chamado *Stag Hunt*. A origem deste jogo consiste numa história contada na obra Rousseau (2004) e que representa o conflito ou *trade-off* nada incomum entre segurança pessoal e cooperação social. Tadelis (2013) resume esta história da seguinte forma:

“Dois caçadores, que podemos chamar de jogadores 1 e 2, podem escolher entre caçar um veado, que oferece uma refeição maior e mais saborosa, ou caçar uma lebre, muito menos saciável. Caçar veados é desafiador e requer cooperação mútua. Se cada um caçador resolve caçar um veado sozinho, a chance de sucesso é ínfima, enquanto que caçar lebres é uma tarefa individual que não é feita em pares. Então, caçar veados é socialmente mais benéfico mas requer “confiança” entre os caçadores de forma que cada um acredite que o outro juntará forças com ele.”

Denotaremos a estratégia caçar veado (do inglês, “*stag*”) pela letra *s*, enquanto que caçar lebre (do inglês, “*hare*”) será representado pela letra *h*. De forma geral, o jogo pode ser descrito através da matriz de *payoffs* representada na tabela 8.2. Uma condição necessária para que a matriz represente o jogo é que $a > b \geq d > c$.

		Jogador 2	
		<i>s</i>	<i>h</i>
Jogador 1	<i>s</i>	(a,a)	(c,b)
	<i>h</i>	(b,c)	(d,d)

Tabela 8.2: Matriz geral de *payoffs* do jogo Stag Hunt.

Estamos interessados em implementar o jogo Stag Hunt computacionalmente e calcular o(s) equilíbrio(s) de Nash.

8.2 Implementação Computacional

Ao contrário dos outros programas, neste modelo precisaremos apenas do pacote `random`¹. Podemos importá-lo da seguinte forma:

```
1 import random as rd # Precisaremos desta pacote para o computador escolher
2                     # jogar o jogo Stag Hunt
```

A função que define a matriz de *payoffs* é dada por

```
1 def payoff(jogo):
2     """
3
4
5     Parameters
6     -----
7     jogo : Lista.
8         A primeira entrada é a jogada do primeiro jogador e a segunda entrada
9         é a jogada do segundo jogador. Para um jogo ser válido, ele deve estar
10        dentro da matriz de jogos possíveis.
11
12    Returns
13    -----
14    Lista
15        Retorna uma lista em que a primeira entrada representa o payoff do
16        primeiro jogador e a segunda entrada o payoff do segundo jogador.
17
18    """
19
20    if jogo in matriz:
```

¹A documentação desta biblioteca pode ser encontrada clicando aqui.

```

21
22     if jogo == ['s', 's']:
23         return [5, 5]
24     if jogo == ['s', 'h']:
25         return [0, 3]
26     if jogo == ['h', 's']:
27         return [3, 0]
28     if jogo == ['h', 'h']:
29         return [3, 3]
30
31     else:
32
33         return -1

```

Note que a função recebe uma lista com o estado do jogo (depois dos jogadores escolherem suas ações) e retornam um vetor de ganhos. Para rodar o jogo, utilizamos a biblioteca que implementamos acima. O comando `rd.choice(estrategia)` faz com que o computador escolha aleatoriamente um elemento da lista `estrategia` (que contém as ações possíveis de cada jogador). Se ambos os jogadores fazem este procedimento, um estado do jogo é determinado e podemos, assim, calcular o *payoff* de cada jogador pela função que acabamos de introduzir acima. O código da função que determina o jogo é dado abaixo.

```

1  def rodada():
2      """
3
4
5      Returns
6      -----
7      jogo : Lista.
8          Retorna uma lista contendo o jogo feito pela escolha das estratégias
9          pelos dois jogadores.
10
11      """
12
13     jogada_A, jogada_B = rd.choice(estrategia), rd.choice(estrategia)
14     jogo = [jogada_A, jogada_B]
15
16     return jogo

```

A última função é responsável por determinar o equilíbrio de Nash do jogo e é dada pelo seguinte código:

```

1 def nash_equilibrium(jogo):
2     """
3
4
5     Parameters
6     -----
7     jogo : Lista
8         Recebe uma lista do jogo determinado pelas ações (escolha das
9         estratégias) dos jogadores.
10
11     Returns
12     -----
13     Bool
14         Assim como na função anterior, calcula o equilíbrio de Nash. Aqui,
15         porém, fazemos um algoritmo em que o cálculo do equilíbrio baseia-se
16         na análise da mudança de estratégia unilateral de cada jogador: se
17         houver uma melhora no payoff dos jogadores, então não há equilíbrio;
18         caso contrário, há.
19
20     """
21
22     if jogo in matriz:
23
24         payoffs = payoff(jogo)
25         equilibrio = True
26
27         if jogo[0] == 'h':
28
29             jogo[0] = 's'
30             payoff_aux = payoff(jogo)
31
32             if payoff_aux[0] > payoffs[0]:
33
34                 equilibrio = False
35
36             jogo[0] = 'h'
37
38         if jogo[0] == 's':
39
40             jogo[0] = 'h'
41             payoff_aux = payoff(jogo)
42
43             if payoff_aux[0] > payoffs[0]:
44
45                 equilibrio = False

```



```

46
47     jogo[0] = 's'
48
49     if jogo[1] == 'h':
50
51         jogo[1] = 's'
52         payoff_aux = payoff(jogo)
53
54         if payoff_aux[1] > payoffs[1]:
55
56             equilibrio = False
57
58         jogo[1] = 'h'
59
60     if jogo[1] == 's':
61
62         jogo[1] = 'h'
63         payoff_aux = payoff(jogo)
64
65         if payoff_aux[1] > payoffs[1]:
66
67             equilibrio = False
68
69         jogo[1] = 's'
70
71     return equilibrio
72
73 else:
74
75     return -1

```

O leitor pode analisar com cuidado o que a função faz e verá que o procedimento foi explicado na parte teórica do modelo, quando o conceito de equilíbrio foi detalhado. Determinado um estado do jogo, o algoritmo da função muda as estratégias dos jogadores de maneira unilateral e, caso nenhum jogador seja capaz de melhorar sua situação (isto é, obter um *payoff* maior), então temos um equilíbrio de Nash.

Depois de definidas as funções, definimos a matriz de estados possíveis do jogo, bem como o vetor de estratégia:

```

1  estrategia = ['s', 'h'] # as possíveis ações dos jogadores são caçar veado
2                        # (jogar "s") ou caçar lebre (jogar "h").
3  matriz = [['s', 's'], ['s', 'h'],
4            ['h', 's'], ['h', 'h']] # jogos possíveis determinando pela matriz.

```

Por fim, programamos o jogo pelo código abaixo. Note que o loop `while` faz com que o jogo seja jogado pelo computador até que atinja um estado de equilíbrio. O usuário é informado sobre cada jogo implementado e quantos jogos são feitos até que se atinja o equilíbrio.

```
1  equilibrio = False # Para entrar no loop abaixo
2  i = 1 # contagem do número de jogos
3
4  while not equilibrio: #Jogos serão feitos até que se alcance um equilibrio
5
6      jogo = rodada() #Um jogo é uma lista com as jogadas dos dois jogadores
7      equilibrio = nash_equilibrium(jogo) #Confere se o jogo constitui equilibrio
8      payoffs = payoff(jogo) #Calcula o Payoff do jogo
9
10     print("\nJogaremos até encontrarmos um jogo com Equilíbrio de Nash.")
11
12     if jogo == ['s', 'h']:
13
14         print(f"\nJogo {i}: O jogador A falhou em tentar caçar", end = " ")
15         print("o veado sozinho; o jogador B conseguiu caçar", end = " ")
16         print("a lebre sozinho. Como o jogador A poderia", end = " ")
17         print("melhorar sua situação mudando sua estratégia e", end = " ")
18         print("caçar uma lebre sozinho, este jogo não", end = " ")
19         print("constitui um Equilíbrio de Nash.")
20
21     if jogo == ['h', 's']:
22
23         print(f"\nJogo {i}: O jogador B falhou em tentar caçar", end = " ")
24         print("o veado sozinho; o jogador A conseguiu caçar", end = " ")
25         print("a lebre sozinho. Como o jogador B poderia", end = " ")
26         print("melhorar sua situação mudando sua estratégia e", end = " ")
27         print("caçar uma lebre sozinho, este jogo não", end = " ")
28         print("constitui um Equilíbrio de Nash.")
29
30     if jogo == ['h', 'h']:
31
32         print(f"\nJogo {i}: Ambos os jogadores escolheram", end = " ")
33         print("caçar lebres e conseguiram. Como nenhum jogador", end = " ")
34         print("tem a ganhar mudando sua estratégia unilateralme", end = "")
35         print("nte para caçar um veado, então este jogo constit", end = "")
36         print("ui um Equilíbrio de Nash. Note, no entanto, que ambo", end = "")
37         print("s melhorariam sua situação se combinassem de caçar um veado.")
38
39     if jogo == ['s', 's']:
40
41         print(f"\nJogo {i}: Ambos os jogadores escolheram", end = " ")
```

```

42     print("caçar veados e conseguiram. Como nenhum jogador", end = " ")
43     print("tem a ganhar mudando sua estratégia unilateralme", end = "")
44     print("nte para caçar uma lebre, então este jogo constit", end = "")
45     print("ui um Equilíbrio de Nash. Note que este é o melhor", end = " ")
46     print("resultado que os jogadores poderiam ter.")
47
48     i += 1
49
50     i -= 1

```

Note que a implementação computacional trata de um caso específico do jogo *Stag Hunt*, em que $a = 5$, $b = d = 3$, e $c = 0$. Podemos escrever, então, a matriz payoff da seguinte maneira:

		Jogador 2	
		s	h
Jogador 1	s	(5,5)	(0,3)
	h	(3,0)	(3,3)

Tabela 8.3: Matriz particular de *payoffs* do jogo *Stag Hunt*.

Note que este jogo, ao contrário do dilema dos prisioneiros, terá mais de um equilíbrio. Tanto a situação em que os dois jogadores caçam veados ou caçam lebres constitui um Equilíbrio de Nash. De fato, note que em ambos os casos, nenhum jogador tem a ganhar mudando sua estratégia *unilateralmente*. Se eles caçam veados, todos recebem um *payoff* igual a 5, e a mudança *unilateral* para caçar lebre faz com que o jogador piore a sua própria situação e passe a receber um *payoff* igual a 3. Com raciocínio semelhante, se ambos caçam lebres, a mudança *unilateral* para caçar veado não é benéfica, visto que o jogador passa de um *payoff* de 3 para um *payoff* nulo.

Ao rodar o programa, jogos são feitos de forma aleatória até que se jogue uma situação de equilíbrio, isto é, quando ambos caçam lebre ou veado. Uma discussão interessante que este jogo levanta é por que ambos os jogadores caçariam lebres se eles podem obter maior benefício combinando de, juntos, caçarem veados.² Isto ocorre pois não necessariamente há confiança mútua entre os caçadores para juntarem forças e realizar a tarefa. Se um deles antecipa que o outro não irá se arriscar a caçar um veado, então claramente a melhor coisa a se fazer é também caçar lebres. Este tipo de dilema mostra precisamente a força do conceito de Equilíbrio de Nash, que depende das crenças dos jogadores.

²Dizemos, neste caso, que (s, s) Pareto domina (h, h) .

Capítulo 9

Oligopólio (competição imperfeita)

Em economia, as *estruturas de mercado* correspondem às formas como os mercados são organizados. Frequentemente, ouvimos falar sobre as estruturas de concorrência perfeita, em que existe um número suficientemente grande de firmas incapazes de influenciar o preço de mercado, e as de monopólio, em que uma única firma domina o mercado e, assim, determina o preço e a quantidade ofertada. No entanto, uma situação intermediária a estes dois casos é a que chamamos de *oligopólio*. Nesta, existe um número considerável de concorrentes, mas todos com poder de influenciar o preço de mercado. Entretanto, para os nossos propósitos será suficiente considerar o caso de um duopólio, quando somente duas empresas atuam no mercado.

Uma consequência da organização do mercado em oligopólio é a interação estratégica entre as firmas.¹ Existem vários modelos que tentam descrever o comportamento das firmas no âmbito dos oligopólios. Nesta seção, focaremos no estudo do chamado *modelo de Cournot*, caracterizado por um duopólio em que as empresas devem maximizar seu lucro fazendo previsões sobre a produção da concorrente. O *equilíbrio de Cournot*, assim, é a situação em que ambas as firmas veem suas previsões (sobre as produções da empresa concorrente) confirmadas na realidade. A implementação computacional não apresenta maiores dificuldades se a teoria é entendida. Para mais sobre o estudo de oligopólios, ver capítulo 27 de Varian (2012).

9.1 O modelo

Supomos, de início, que ambas as firmas do duopólio são idênticas. Desta maneira, podemos nos restringir a estudar o problema de apenas uma firma representativa. Assim, considere o problema da empresa 1. Primeiramente, denote y_2^e a produção da empresa 2 esperada pela empresa 1. A produção total da economia esperada pela empresa 1 é de $Y = y_1 + y_2^e$ (y_1 é a

¹A área de Teoria dos Jogos que vimos na seção anterior, por exemplo, é extremamente importante neste estudo da tomada de decisão das firmas no âmbito dos oligopólios.

produção da empresa 1), que gerará, por sua vez, um preço de mercado $p(Y) = p(y_1 + y_2^e)$. O problema de maximização do lucro da empresa 1 será:

$$\max_{y_1} p(y_1 + y_2^e)y_1 - c(y_1). \quad (9.1)$$

Por simplicidade, admitiremos um custo nulo, $c(y_1) = 0$, e uma curva de demanda inversa linear $p(Y) = a - bY$, de tal modo que maximizar o lucro é maximizar a receita $[a - b(y_1 + y_2^e)]y_1$. Observe pelo problema de maximização que a escolha ótima de produção y_1^* dependerá da expectativa da empresa 1 sobre a produção da empresa 2. Então, pode-se escrever que

$$y_1 = f_1(y_2^e). \quad (9.2)$$

Chamamos esta função de *curva de reação* da empresa 1. Como dissemos anteriormente, a análise desenvolvida para a empresa 1 também serve para a empresa 2. Assim, esta última também terá uma curva de reação, dada por

$$y_2 = f_2(y_1^e). \quad (9.3)$$

Desta forma, o equilíbrio de Cournot será dado pelo par (y_1^*, y_2^*) satisfazendo

$$\begin{cases} y_1^* = f_1(y_2^*) \\ y_2^* = f_2(y_1^*), \end{cases}$$

ou seja, o equilíbrio de Cournot é dado pelo par de produção no qual as curvas de reação se cruzam.

De acordo com Varian (2012), em equilíbrio

“cada empresa maximiza seus lucros de acordo com suas expectativas sobre a escolha de produção da outra empresa e, além disso, essas expectativas são confirmadas em equilíbrio: cada empresa escolhe de forma ótima fabricar a quantidade que a outra empresa espera que ela fabrique. Num equilíbrio de Cournot, nenhuma empresa achará lucrativo mudar sua produção, uma vez que descubra a escolha realmente feita pela outra empresa.”

Com custo nulo e função de demanda inversa dada por $p(Y) = a - bY$, podemos resolver o problema (9.1) de ambas as firmas, que ficará da seguinte forma:

$$\max_{y_1} [a - b(y_1 + y_2^e)]y_1.$$

A solução deve satisfazer a condição de primeira ordem (basta, como sempre, derivar a função objetivo e igualar a zero) e será dada por

$$y_1 = \frac{a - by_2^e}{2b}.$$

Igualmente, como assumimos que ambas as firmas são idênticas, segue que

$$y_2 = \frac{a - by_1^e}{2b}.$$

Para determinar o equilíbrio de Cournot, faça $y_1^e = y_1$ e $y_2^e = y_2$ e resolva o sistema

$$\begin{cases} y_1 = \frac{a - by_2}{2b} \\ y_2 = \frac{a - by_1}{2b} \end{cases} \quad (9.4)$$

Então, teremos a solução dada por

$$(y_1^*, y_2^*) = \left(\frac{a}{3b}, \frac{a}{3b} \right),$$

de forma que a produção total da economia sob este duopólio será $Y = y_1^* + y_2^* = 2a/3b$.

9.2 Implementação Computacional

Devemos, primeiramente, implementar funções para a demanda inversa e para a receita. O seguinte código faz exatamente isso.

```

1 def demanda_inversa(Y, a = 1000, b = 1): #preço em função da quantidade
2     """
3     Definindo a demanda inversa, que depende da produção total da economia e
4     dos parâmetros 'a' e 'b'. O parâmetro Y é uma lista em que a entrada i
5     representa a produção da empresa i.
6     """
7
8     assert a > 0 and b > 0, "Os parâmetros da demanda devem ser positivos"
9
10    global c, d
11    c, d = a, b
12
13    return a - b*(sum(Y))
14
15 def receita(y, Y):
16     """

```

```

17 Receita = preço*quantidade. No caso do oligopólio, o preço é dado pela
18 demanda inversa e a quantidade determinada pela firma.
19 """
20
21 return y*demanda_inversa(Y)

```

Por padrão, consideramos a demanda inversa $p(Y) = 1000 - Y = 1000 - (y_1 + y_2^e)$. O parâmetro `Y` é uma lista em que a entrada $i \in \{1, 2\}$ representa a produção da empresa i . Em seguida, a função `curva_reacao()` resolve o problema de maximização da firma representativa (1 ou 2) em função da produção que ela espera da outra firma, ou seja, a função gera a curva de melhor resposta:

```

1 def curva_reacao():
2     """
3     Serve para o caso em que numero_firmas = 2. A função resolve o problema de
4     maximização da firma arbitrária (1 ou 2) em função da produção que ela
5     espera da outra firma, ou seja, a função gera a curva de melhor resposta.
6     O equilíbrio de Cournot será a intersecção entre ambas as curvas.
7     """
8
9     demanda_inversa([0,1]) # Chamo a função apenas para definir o parâmetro 'c'
10
11     # Resolvendo o problema para uma empresa arbitrária (empresa 1 ou 2):
12
13     receita_valores = []
14     y_argmax = []
15     y, y_e = np.linspace(0, c, c+1), np.linspace(0, c, c+1)
16
17     for j in y_e:
18
19         for i in y:
20
21             receita_valores.append(receita(i, [i,j]))
22
23             indice_max = np.argmax(receita_valores) # Guarda o índice da maior
24                                     # receita
25             y_argmax.append(y[indice_max]) # Adiciona a uma lista o nível de
26                                     # produção y que maximiza o lucro
27             receita_valores = [] # "zera" a lista de receita para o algoritmo
28                                     # resolver novamente o problema de maximizacao de
29                                     # lucro para outro nível de produção esperado
30
31     return y_e, y_argmax

```

Observe que o algoritmo acima serve apenas para o duopólio (apenas duas firmas), caso em que a visualização gráfica das curvas de reação é possível. A última função a ser implementada é justamente a que calcula o equilíbrio de Cournot e serve para um número arbitrário de empresas, embora por padrão a implementação seja para apenas duas:

```

1 def equilibrio_cournot(numero_firmas = 2):
2     """
3     Esta função serve para o caso geral em que numero_firmas = n. A solução é
4     implementada algebricamente. As firmas são idênticas e o custo é nulo. O
5     cálculo é realizado para o caso especial de demanda inversa linear.
6     """
7
8     coeficientes = np.array([[2 if j==i else 1 for i in range(numero_firmas)]
9                               for j in range(numero_firmas)])
10    inversa_coeficientes = np.linalg.inv(coeficientes)
11
12    constantes = np.array([c/d for i in range(numero_firmas)])
13
14    vetor_producao = np.dot(inversa_coeficientes, constantes)
15
16    return vetor_producao

```

Note que a função acima é responsável por resolver um sistema linear de n equações e, assim, determinar o equilíbrio de Cournot. Embora seja uma extensão simples do sistema (9.4), omitiremos a explicação da álgebra necessária para a solução. No entanto, o leitor é convidado a tentar generalizar analiticamente o sistema (9.4) para n firmas idênticas.

O programa utiliza um exemplo que segue o caso de demanda inversa linear e custo nulo que detalhamos anteriormente na explicação do modelo. Embora o programa comporte alterações, os parâmetros da demanda inversa são $a = 1000$ e $b = 1$ tais que $p(Y) = 1000 - Y = 1000 - (y_1 + y_2^e)$. Assim, os problemas das empresas podem ser representados por

$$\begin{cases} \max_{y_1} [1000 - (y_1 + y_2^e)]y_1 \\ \max_{y_2} [1000 - (y_2 + y_1^e)]y_2. \end{cases}$$

As curvas de reação serão $y_1 = 500 - y_2^e/2$ e $y_2 = 500 - y_1^e/2$. Assim, fazendo $y_1^e = y_1$ e $y_2^e = y_2$ e resolvendo o sistema acima, o equilíbrio de Cournot é dado pelo par $(y_1^*, y_2^*) = (1000/3, 1000/3)$. Para achar o preço de equilíbrio desta economia de duopólio, basta usar o vetor de produção das firmas na função de demanda inversa. O seguinte código faz uso das funções para implementar este exemplo, e a figura (9.1) ilustra graficamente.


```

1 curva_de_reacao = curva_reacao()
2
3 equilibrio_cournot1 = equilibrio_cournot() # Para n=2 firmas
4
5 quantidade_equilibrio = sum(equilibrio_cournot1)
6
7 preco_equilibrio = demanda_inversa(equilibrio_cournot1)

```

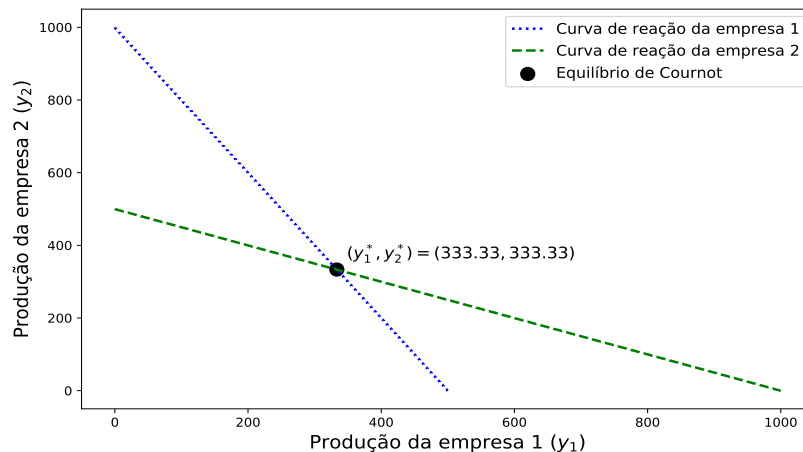


Figura 9.1: O equilíbrio de Cournot.

O código que gera este gráfico é dado a seguir.

```

1 # Gráfico para o exemplo de 2 firmas (curvas de reação):
2
3 plt.plot(curva_de_reacao[1], curva_de_reacao[0], color="blue", linewidth=2.0,
4          ls = "dotted", label = "Curva de reação da empresa 1")
5 plt.plot(curva_de_reacao[0], curva_de_reacao[1], color="green", linewidth=2.0,
6          ls = "dashed", label = "Curva de reação da empresa 2")
7 plt.xlabel(r"Produção da empresa 1 ($y_1$)", fontsize = 15)
8 plt.ylabel(r"Produção da empresa 2 ($y_2$)", fontsize = 15)
9 plt.scatter(equilibrio_cournot1[0], equilibrio_cournot1[1], s = 7*4**2,
10             label = "Equilíbrio de Cournot", color = "black")
11 plt.annotate(f"${y_1^*, y_2^*} = ({c/3*d: .2f}, {c/3*d: .2f})$",
12             ((c/3*d)+15, (c/3*d)+30), fontsize = 13)
13 plt.legend()
14 plt.show()

```

Podemos, ainda, analisar como a economia se comporta com o aumento do número de firmas neste oligopólio. O código abaixo implementa esta análise calculando as quantidades e

preços de equilíbrio conforme aumentamos o número de firmas, além de plotar o gráfico que dá os diferentes níveis de produção e preço para diferentes quantidades de empresas no oligopólio.

```
1 preco2 = []
2 quantidade2 = []
3 n_firmas = 100
4
5 for i in range(1, n_firmas + 1): # para cada numero de firmas, calcular as
6                                     # a quantidade e preço de equilíbrio da
7                                     # economia.
8
9     equilibrio_cournot2 = equilibrio_cournot(i)
10    quantidade2.append(sum(equilibrio_cournot2))
11    preco2.append(demanda_inversa(equilibrio_cournot2))
12
13 # Gráfico para a análise da entrada de firmas:
14
15 plt.plot(np.linspace(1, n_firmas, n_firmas), quantidade2, color="blue",
16          linewidth=2.0, ls = "dotted", label = "Quantidade")
17 plt.plot(np.linspace(1, n_firmas+1, n_firmas), preco2, color="green",
18          linewidth=2.0, ls = "dashed", label = "Preço")
19 plt.xlabel(r"Número de firmas", fontsize = 15)
20 plt.legend()
21 plt.show()
```

A figura 9.2 mostra uma consequência interessante: a produção da economia aumenta conforme aumentamos o tamanho do oligopólio, enquanto que o preço converge ao custo marginal nulo.

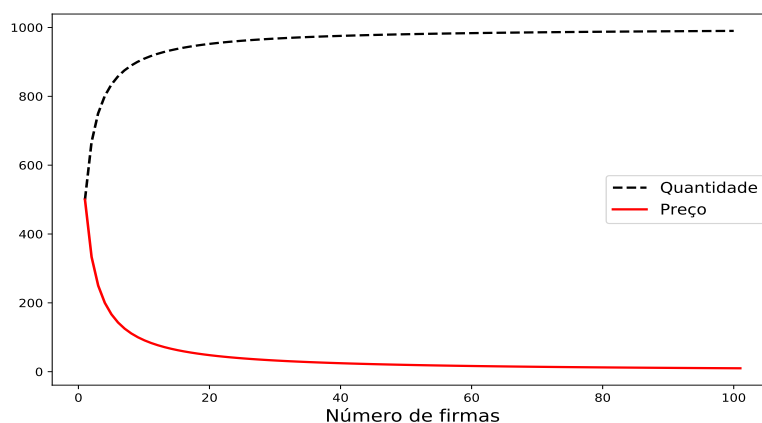


Figura 9.2: Efeito do aumento no número de firmas nos níveis de produção e preço.

Capítulo 10

Caixa de Edgeworth: uma análise para o equilíbrio geral

No estudo de equilíbrio geral para uma economia de trocas com dois agentes e dois mercados, é comumente útil representar as alocações possíveis numa ferramenta gráfica conhecida como *caixa de Edgeworth*. É possível ilustrar também as curvas de preferências de ambos os indivíduos, bem como a dotação inicial, de maneira que facilita o estudo das trocas feitas na economia e que levam ao equilíbrio de mercado.

Considero o presente modelo teórico com um grau de dificuldade acima dos anteriores. Talvez isto decorra do fato de que a análise de equilíbrio geral requer um entendimento da dinâmica de trocas que não está presente nos modelos mais básicos de equilíbrio parcial. Já com respeito à implementação computacional, talvez a maior novidade seja a representação gráfica da caixa de Edgeworth, que vai além da forma usual de como lemos gráficos de duas dimensões. Para o estudo mais aprofundado do tema, a referência recomendada é o capítulo 31 de Varian (2012).

10.1 O modelo

Estaremos interessados em estudar uma economia competitiva de trocas puras (sem produção) com dois consumidores e dois bens (ou dois mercados). Denote os dois consumidores por A e B e ambos os bens por x e y . Suponha que o indivíduo A comece com uma dotação $\omega_A = (\omega_A^x, \omega_A^y)$ e o consumidor B com dotação $\omega_B = (\omega_B^x, \omega_B^y)$. Entende-se por *alocação* um par qualquer de cestas de consumo $\Omega = ((x_A, y_A), (x_B, y_B))$ e diz-se que a alocação é *factível* se a demanda total por um bem corresponde à sua oferta inicial, isto é, para $k \in \{x, y\}$,

$$k_A + k_B = \omega_A^k + \omega_B^k. \quad (10.1)$$

Dada a dotação, ambos os agentes podem melhorar sua situação inicial fazendo trocas entre si, sempre com o objetivo usual de maximizar sua satisfação. Consideraremos que as preferências de um consumidor $j \in \{A, B\}$ qualquer são representadas por uma função de utilidade $U(x_j, y_j) \equiv U_j(x, y)$. Assim, o problema do consumidor j consiste em maximizar sua utilidade respeitando a restrição orçamentária, ou seja,

$$\begin{cases} \max_{x_j, y_j} U(x_j, y_j) \\ \text{s. a. } p_x x_j + p_y y_j = p_x \omega_j^x + p_y \omega_j^y. \end{cases}$$

Lembremos que este é o clássico problema do consumidor e a solução será dada quando a taxa marginal de substituição entre os bens igualar-se à razão de preços (que supomos dados, por se tratar de mercados perfeitamente competitivos). Como consequência da solução, teremos as *demandas brutas* individuais pelo bem $k \in \{x, y\}$, que dependerão dos preços e das dotações iniciais. Denotaremos esta demanda por $k_j^d(p, \omega_j)$, em que $p = (p_x, p_y)$ é o vetor de preços e ω_j é o vetor de dotação do indivíduo j . Definiremos, ainda, a *demanda líquida* ou *demanda excedente* de um agente j por um bem k como a diferença entre sua demanda bruta e sua dotação inicial, isto é,

$$e_j^k(p, \omega_j) \equiv k_j^d(p, \omega_j) - \omega_j^k.$$

Qual deverá ser o equilíbrio nesta economia? É natural definirmos que o equilíbrio de mercado (ou equilíbrio *walrasiano*) seja dado pelo vetor de preços $p^* = (p_x^*, p_y^*)$ e pela alocação $\Omega^* = ((x_A^*, y_A^*), (x_B^*, y_B^*))$ tais que os consumidores demandam as cestas de Ω^* e a alocação é factível, isto é, a igualdade (10.1) é satisfeita. Podemos, ainda, definir a *demanda líquida agregada* (também chamada de *demanda excedente agregada*) de um bem $k \in \{x, y\}$ como a soma de suas demandas líquidas:

$$z_k(p) \equiv e_A^k(p) + e_B^k(p),$$

em que $p = (p_x, p_y)$.

Observe que a restrição do problema do consumidor, dada por

$$\begin{aligned} p_x x_j + p_y y_j = p_x \omega_j^x + p_y \omega_j^y &\iff p_x(x_j - \omega_j^x) + p_y(y_j - \omega_j^y) = 0 \\ &\iff p_x e_j^x(p, \omega_j) + p_y e_j^y(p, \omega_j) = 0, \end{aligned}$$

tem que valer para todos os indivíduos $j \in \{A, B\}$. Somando as equações da restrição de ambos os consumidores, temos que

$$\begin{aligned} &[p_x e_A^x(p, \omega_A) + p_y e_A^y(p, \omega_A)] + [p_x e_B^x(p, \omega_B) + p_y e_B^y(p, \omega_B)] \\ &= p_x(e_A^x(p, \omega_A) + e_B^x(p, \omega_B)) + p_y(e_A^y(p, \omega_A) + e_B^y(p, \omega_B)) \end{aligned}$$

$$= p_x z_x(p) + p_y z_y(p) = 0. \quad (10.2)$$

A *Lei de Walras* afirma exatamente o que está escrito na igualdade (10.2). Em outras palavras, o valor da demanda excedente agregada é nulo, independentemente da escolha do vetor de preços p . Este é um resultado extremamente importante uma vez que, para acharmos o equilíbrio nos dois mercados, basta o encontrarmos em um específico e o outro mercado consequentemente estará também em equilíbrio. Em particular, é suficiente fazer de x um bem numérico (i.e., fixar seu preço em uma unidade monetária) e encontrar o preço do bem y , p_y^* , de tal forma que $p_y^* z_y(p^*) = 0$, em que $p^* = (1, p_y^*)$. Como consequência direta, $z_x(p^*) = 0$ também e, portanto, (10.2) é satisfeito.¹

Geometricamente, a alocação de equilíbrio tem como característica a tangência de ambas as curvas de indiferença na caixa de Edgeworth. A figura 10.1 ilustra o raciocínio.

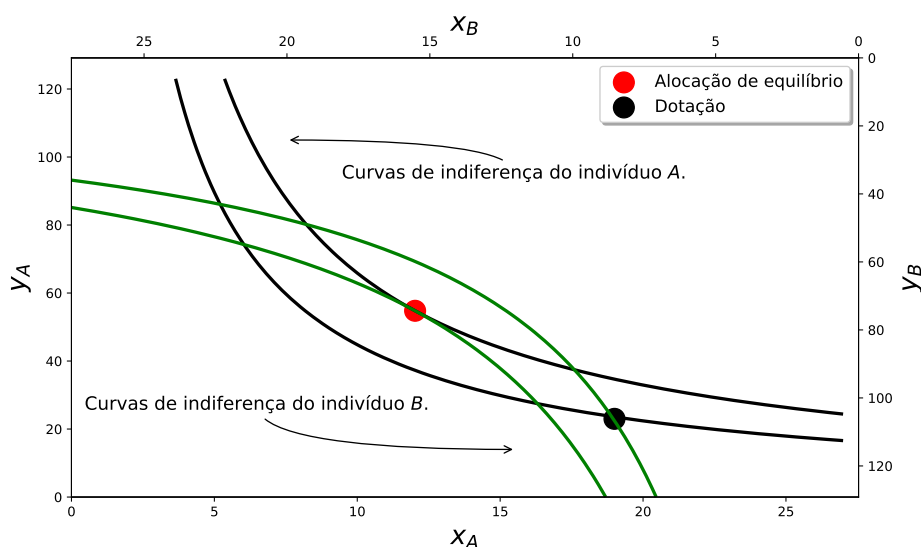


Figura 10.1: Caixa de Edgeworth.

Observe que as curvas de indiferença se cruzam na dotação inicial. Portanto, ambos os indivíduos ficarão melhor caso troquem bens de tal modo que eles se movam para qualquer ponto mais próximo à alocação de equilíbrio, representada pelo ponto vermelho na figura. De fato, eles trocarão bens até que atinjam *exatamente* a alocação de tangência das curvas. Para ver este fato, note que caso as curvas de indiferença não sejam tangentes numa alocação interior, então elas devem se cruzar. Mas, então, sempre haverá como melhorar a situação de pelo menos um dos agentes através das trocas sem piorar o outro. O processo ocorre até que a alocação resultante das trocas seja aquela em que as curvas de indiferença sejam tangentes, momento em que é exaurida também a possibilidade de melhora de um dos agentes sem que o

¹De fato, se existir n mercados, é suficiente encontrar um vetor de preços que equilibra os $n - 1$ mercados de bens. A Lei de Walras nos ensina que, neste caso, o último mercado estará automaticamente em equilíbrio.

outro pior.

Percebe-se, pelo raciocínio acima, que toda alocação de equilíbrio é um *ótimo no sentido de Pareto*, já que, como dito, não há como melhorar um dos agentes sem piorar o outro. De fato, para várias dotações, pode-se chegar a várias alocações ótimas e, portanto, em várias pontos eficientes no sentido de Pareto. Diz-se que conjunto de todas as alocações eficientes no sentido de Pareto na caixa de Edgeworth é a *curva de contrato*, já que elas são alocações ótimas finais desta economia. Este conceito é fundamental e, a seguir, estaremos interessados em implementar computacionalmente um subconjunto da curva de contrato para um exemplo de economia.

10.2 Implementação Computacional

Pode-se começar o programa definindo as funções de utilidade e as demandas (bruta, líquida e de excesso agregado) pelo seguinte código:

```
1 def utilidade(x, y, alpha = 0.5):
2     """
3     Função de utilidade Cobb-Douglas dos agentes (ambos terão a mesma).
4     """
5
6     return (x**alpha)*(y**(1-alpha))
7
8 def demanda(p, dot, alpha = 0.5):
9     """
10    Resolve o problema do consumidor de forma analítica, estabelecendo a
11    demanda pelos bens x e y. Como as funções de utilidade são as mesmas para
12    ambos os agentes, a demanda será igual para valores arbitrários da dotação
13    inicial. O parâmetro 'dot' é uma lista em que a primeira entrada
14    corresponde à dotação inicial do bem x; e a segunda para o bem y. O
15    parâmetro preço é uma lista em que a primeira entrada corresponde ao preço
16    do bem x e a segunda ao preço do bem y.
17    """
18
19    x = alpha*(p[0]*dot[0] + p[1]*dot[1]) / p[0]
20    y = (1 - alpha)*(p[0]*dot[0] + p[1]*dot[1]) / p[1]
21
22    return x, y
23
24 def demanda_liq(p, dot):
25     """
26    Calcula o excesso de demanda para os bens x e y, isto é, a diferença entre
27    a demanda total e a dotação inicial dos agentes. O parâmetro p é uma lista
```

```

28     com os preços de  $x$  e  $y$ , respect., enquanto que  $\text{dot}$  é uma lista com as
29     dotações iniciais dos bens  $x$  e  $y$ , respect..
30     """
31
32      $x, y = \text{demanda}(p, \text{dot})$ 
33      $\text{demanda\_liq\_x} = x - \text{dot}[0]$ 
34      $\text{demanda\_liq\_y} = y - \text{dot}[1]$ 
35
36     return  $\text{demanda\_liq\_x}, \text{demanda\_liq\_y}$ 
37
38 def demanda_liq_agreg( $p, \text{dot\_A}, \text{dot\_B}$ ):
39     """
40     Calcula a demanda líquida agregada para os bens  $x$  e  $y$ . A demanda líquida
41     agregada nada mais é que a soma das demandas líquidas individuais.
42     """
43
44      $\text{demanda\_liq\_agreg\_x} = \text{demanda\_liq}(p, \text{dot\_A})[0] + \text{demanda\_liq}(p, \text{dot\_B})[0]$ 
45      $\text{demanda\_liq\_agreg\_y} = \text{demanda\_liq}(p, \text{dot\_A})[1] + \text{demanda\_liq}(p, \text{dot\_B})[1]$ 
46
47     return  $\text{demanda\_liq\_agreg\_x}, \text{demanda\_liq\_agreg\_y}$ 

```

Por simplicidade, observe que utilizamos uma função de utilidade Cobb-Douglas da forma $U(x, y) = x^\alpha y^{1-\alpha}$ para ambos os agentes, em que $\alpha = 0.5$ por padrão. Sabemos, pela teoria usual do consumidor, que para este tipo de função o indivíduo terá suas demandas brutas dadas por

$$x(p_x, p_y, m) = \alpha m / p_x \quad \text{e} \quad y(p_x, p_y, m) = \alpha m / p_y,$$

em que $m = p \cdot \omega_j$ é o produto interno dos vetores de preço e dotação inicial do indivíduo $j \in \{A, B\}$. Assim, estas demandas pelos bens são implementadas diretamente (depois da solução analítica) na função `demanda($p, \text{dot}, \alpha = 0.5$)`. As duas funções que aparecem no código em seguida, dadas por `demanda_liq_agreg($p, \text{dot_A}, \text{dot_B}$)` e `demanda_liq(p, dot)`, calculam respectivamente o excesso de demanda agregada e a demanda líquida. Ambas recebem como *inputs* os vetores (listas, em Python) de preços e dotações.

As duas últimas funções talvez sejam as mais importantes. Em primeiro lugar, a função `equilibrio($\text{dot_A}, \text{dot_B}$)` tem como objetivo calcular a alocação de equilíbrio desta economia. Isto é feito usando o método que discutimos anteriormente: fixa-se o preço do bem x em uma unidade e acha-se o preço p_y que zera o excesso de demanda agregada do bem y . Por consequência, também é nulo o excesso de demanda agregada pelo bem x .

```

1 def equilibrio( $\text{dot\_A}, \text{dot\_B}$ ):
2     """
3     Calcula o equilíbrio geral para um mercado competitivo de dois bens e dois

```



```

4      indivíduos. Seguindo o que diz a Lei de Walras, o preço do bem x pode ser
5      fixado em 1 (bem numerário) enquanto que achamos o preço do bem 2 que zera
6      o excesso de demanda agregado do bem 2.
7      """
8
9      # Procurando o zero da demanda agregada excedente para o bem y
10     # pelo método da bissecção:
11
12     global px
13
14     px = 1
15     tol = 1e-10
16     p_alto = 1000
17     p_baixo = 0
18
19     # Analisando a fronteira:
20     tentativa1 = demanda_liq_agreg([px, p_alto], dot_A, dot_B)[1]
21     if abs(tentativa1) < tol:
22         return p_alto
23
24     # atualiza preco pela média
25     preco = p_alto / 2
26     excesso_demanda_agreg = demanda_liq_agreg([px, preco], dot_A, dot_B)[1]
27
28     while abs(excesso_demanda_agreg) > tol:
29
30         if excesso_demanda_agreg > 0:
31
32             p_baixo = preco
33
34         else:
35
36             p_alto = preco
37
38         preco = (p_baixo + p_alto) / 2
39         excesso_demanda_agreg = demanda_liq_agreg([px, preco], dot_A, dot_B)[1]
40
41     return preco

```

Observe pelo código acima que o método de busca no *grid* para encontrar o zero de alguma função não mais foi utilizado. Esta é uma técnica pouco eficiente computacionalmente. De maneira alternativa, foi usado o método da bissecção, levemente mais eficiente mas também extremamente fácil de ser implementado.

A última função definida no programa é responsável por encontrar uma reta que é um sub-

conjunto da curva de contrato da economia:

```
1 def curva_contrato():
2     """
3     Calcula um subconjunto da curva de contrato variando as dotações.
4     """
5
6     precos_equilibrio = [] # Lista para adicionar os preços de equilíbrio
7                             # conforme variamos as dotações.
8
9     y_aux = np.linspace(0, total_dot_y, 5)
10    # Não estou "varrendo" toda a caixa de Edgeworth para calcular as
11    # demandas de equilíbrio. Passo apenas inteiramente por um eixo (eixo
12    # x, por ex) e vario alguns pontos do eixo y. Por este motivo, a curva
13    # resultante é um subconjunto da curva de contrato.
14
15    for j in y_aux:
16
17        for i in x:
18
19            precos_equilibrio.append(equilibrio([i, j], [total_dot_x - i,
20                                                         total_dot_y - j]))
21
22    x_estrela = []
23    y_estrela = []
24
25    for k in precos_equilibrio:
26
27        for m in y_aux:
28
29            for l in x:
30
31                x_estrela.append(demanda([1, k], [1, m])[0])
32                y_estrela.append(demanda([1, k], [1, m])[1])
33
34    return [x_estrela, y_estrela]
```

Basicamente, a função varia as dotações iniciais e encontra a alocação de equilíbrio para cada uma destas dotações. Observe que o algoritmo não passa por todos os pontos da caixa de Edgeworth e, portanto, rigorosamente falando, não se pode dizer que a reta resultante é a curva de contrato — que, por definição, é o conjunto de todas as alocações ótimas de Pareto da economia. A função apenas implementa o cálculo de equilíbrio para um número suficientemente grande de dotações iniciais. No entanto, a curva resultante pode ser vista como uma boa aproximação da curva de contrato e passaremos a se referir a ela por este nome. A figura

10.2 exibe graficamente o resultado.

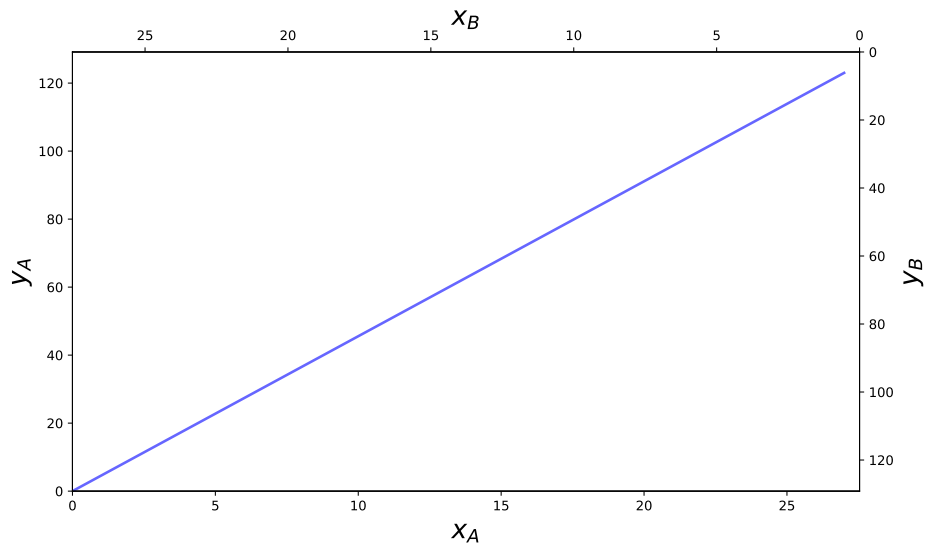


Figura 10.2: Curva de contrato.

No programa principal, o código define alguns parâmetros que serão usados nas funções, como as dotações iniciais de cada consumidor. Em seguida, as funções são invocadas para implementar um exemplo de economia:

```

1 #####
2 #                               Programa principal                               #
3 #####
4
5 #dot_A = [100, 25] # dotação indivíduo A
6 #dot_B = [50, 150] # dotação indivíduo B
7
8 #dot_A = [23, 33] # dotação indivíduo A
9 #dot_B = [14, 32] # dotação indivíduo B
10
11 dot_A = [230, 100] # dotação indivíduo A
12 dot_B = [400, 90] # dotação indivíduo B
13
14 preco_equilibrio_y = equilibrio(dot_A, dot_B)
15 p_equilibrio = [px, preco_equilibrio_y]
16
17 #Calcula as demandas de equilíbrio para os indivíduos A e B, respect.:
18 demanda_eq_A = demanda(p_equilibrio, dot_A)
19 demanda_eq_B = demanda(p_equilibrio, dot_B)
20
21 # Utilidade dos agentes em equilíbrio:
22 u_valor_A = utilidade(demanda_eq_A[0], demanda_eq_A[1])

```

```

23 u_valor_B = utilidade(demanda_eq_B[0], demanda_eq_B[1])
24
25
26 total_dot_x = dot_A[0] + dot_B[0] # pode-se considerar o tamanho do eixo
27                                     # horizontal da caixa de Edgeworth
28 total_dot_y = dot_A[1] + dot_B[1] # pode-se considerar o tamanho do eixo
29                                     # vertical da caixa de Edgeworth
30
31 x = np.linspace(0.0001, total_dot_x, 100) # grid da quantidade do bem x
32 y = np.linspace(0.0001, total_dot_y, 100) # grid da quantidade do bem y
33
34 contrato = curva_contrato()

```

Note que o código guarda como comentário exemplos alternativos de dotações que podem ser implementados pelo leitor. Ao rodar o programa, uma mensagem informando as dotações iniciais e a alocação de equilíbrio resultante é mostrada na tela. Para o exemplo destacado, com dotações dos indivíduos *A* e *B* dadas respectivamente por (230, 100) e (400, 90), a alocação ótima desta economia é dada por $\Omega^* \approx ((281, 85), (349, 105))$. A figura 10.3 ilustra graficamente o exemplo. Observe como a curva de contrato passa por todas as tangências das curvas de indiferença dos consumidores (como não poderia deixar de ser).

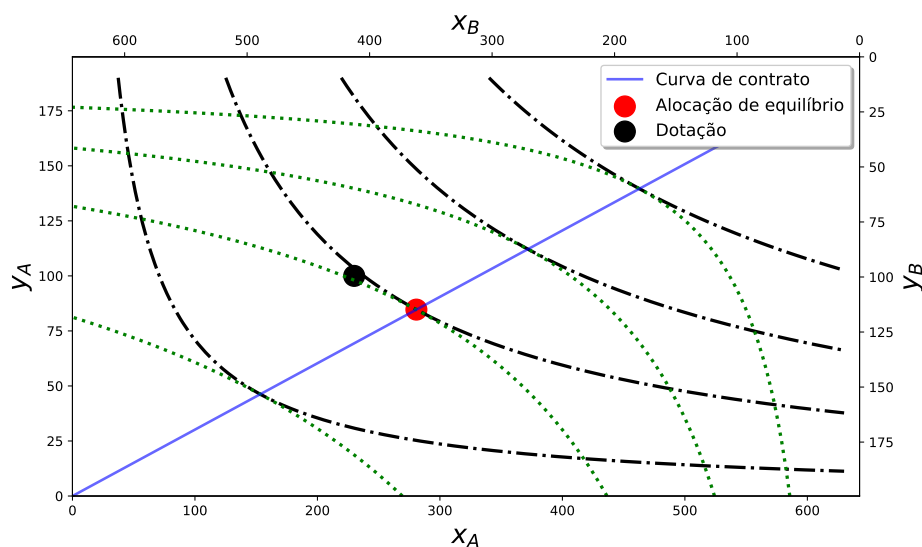


Figura 10.3: Caixa de Edgeworth para um exemplo de economia.

Este gráfico gerado pode ser implementado utilizando o seguinte código:

```

1 plt.rcParams["figure.figsize"] = (10, 6)
2 plt.rcParams["lines.linewidth"] = (2.5)
3

```

```

4 fig, ax = plt.subplots(tight_layout = True)
5
6 # Cria o produto cartesiano das quantidades  $x_1$  e  $x_2$ .
7 xc_1, xc_2 = np.meshgrid(x, y)
8
9 # Coloca label nos eixos originais
10 ax.set_xlabel('$x_A$', fontsize = 20)
11 ax.set_ylabel('$y_A$', fontsize = 20)
12
13 ax.axis([0, 1.02*total_dot_x, 0, 1.05*total_dot_y])
14
15 # Calcula a utilidade do consumidor A
16 uA = utilidade(xc_1, xc_2) # Calcula a matriz de utilidades do
17                               # consumidor A.
18
19 # Calcula a utilidade do consumidor B. Como queremos que o gráfico tenha origem
20 # em  $x = x_0$  e  $y = y_0$ , fazemos o ajuste nos argumentos da função.
21 uB = utilidade(total_dot_x - xc_1, total_dot_y - xc_2) # Calcula a matriz de
22                                                         # utilidades do
23                                                         # consumidor B.
24
25 # Criamos um novo eixo y (apenas para repetir os ticks e labels do eixo y
26 #                               original)
27 ax2 = ax.twinx()
28
29 # Copiamos os limites do eixo y original
30 orig_ylim = ax.get_ylim()
31
32 # Replicamos os limites do eixo y original no novo eixo
33 ax2.set_ylim(orig_ylim)
34
35 # Definimos um label para o novo eixo y
36 ax2.set_ylabel('$y_B$', fontsize = 20)
37
38 # Invertamos o sentido do texto do eixo y original
39 ax2.invert_yaxis()
40
41 # Criamos um novo eixo x (apenas para repetir os ticks e labels do eixo x
42 #                               original)
43 ax3 = ax.twiny()
44
45 # Copiamos os limites do eixo x original
46 orig_xlim = ax.get_xlim()
47
48 # Replicamos os limites do eixo x original no novo eixo
49 ax3.set_xlim(orig_xlim)

```

```

50
51 # Definimos um label para o novo eixo x
52 ax3.set_xlabel('$x_B$', fontsize = 20)
53
54 # Invertamos o sentido do texto do eixo x original
55 ax3.invert_xaxis()
56
57 # Curvas de indiferença no nível indicado
58 CS1 = ax.contour(xc_1, xc_2, uA, levels = [u_valor_A - 70, u_valor_A,
59                                           u_valor_A + 50, u_valor_A + 100],
60                 linestyle = 'dashdot', colors = 'k', alpha = 1,
61                 extend = 'both')
62 CS2 = ax.contour(xc_1, xc_2, uB, levels = [u_valor_B - 100, u_valor_B - 50,
63                                           u_valor_B, u_valor_B + 70],
64                 linestyle = 'dotted', colors = 'g', alpha = 1,
65                 extend = 'both')
66
67 # Curva de contrato:
68 ax.plot(contrato[0], contrato[1], 'blue', linewidth = 2, alpha = 0.6,
69         label = 'Curva de contrato')
70
71 #Alocação de equilíbrio:
72 ax.scatter(demanda_eq_A[0], demanda_eq_A[1], s = 10*5**2,
73           label = "Alocação de equilíbrio", alpha = 1, c = "r")
74 #ax.plot(demanda_eq_A[0], demanda_eq_A[1], 'r', marker = ".", markersize = 25,
75 #        label = "Alocação de equilíbrio")
76
77 # Dotação inicial:
78 ax.scatter(dot_A[0], dot_A[1], s = 10*5**2, label = "Dotação", alpha = 1,
79           c = "black")
80 #ax.plot(dot_A[0], dot_A[1], 'black', marker = ".", markersize = 25,
81 #        label = "Dotação")
82 ax.legend(loc = "upper right", fancybox = True, shadow = True, fontsize = 13)
83 plt.show()

```


Capítulo 11

Matching (estabilidade de casamentos)

Se quisermos parear (fazer um *matching* entre) elementos de dois diferentes grupos — tais como faculdades e alunos, empregadores e empregados, homens e mulheres —, qual seria o algoritmo que estabeleceria um resultado previsto ou desejado? Como poderíamos, por exemplo, construir um mecanismo para estabelecer os melhores casamentos *estáveis* para os homens com base em suas preferências sobre o grupo de mulheres (ou o contrário)? Descreveremos, neste capítulo, um algoritmo simples para responder a esta pergunta. Embora nossa atenção esteja no *matching* entre homens e mulheres, podemos aplicar os mesmos conceitos para parear quaisquer dois grupos que se encaixam nas hipóteses que faremos para a construção do algoritmo.

Este tipo de problema normalmente é descrito em teoria econômica como um *mercado de dois lados*, em que os consumidores (ou agentes, de modo geral) buscam o *matching* entre si. Para o problema específico que estamos considerando, isto é, o *problema da estabilidade de casamentos*, a prova da existência de solução foi dada por Gale and Shapley (1962), que ofereceram um algoritmo para tal. A implementação em Python não é complicada se o leitor é habituado a operações com listas de listas (ou matrizes). Uma introdução ao assunto pode ser vista em Varian (2012), capítulo 17, seção 7.

11.1 O modelo

Considere a existência de n mulheres e n homens, de modo que cada homem e cada mulher ordenam as pessoas do sexo oposto com base em suas preferências — isto é, estabelecem uma classificação das pessoas com quem eles preferem se casar. Suponha também que, nesta ordenação, cada pessoa pode preferir ficar solteira a ficar com uma pessoa do sexo oposto. Assim, cada pessoa possui $n + 1$ elementos de ordenação: as n pessoas do sexo oposto *mais* a opção de ficar solteira.

Dissemos anteriormente que queríamos um algoritmo que produzisse casamentos *estáveis*.

Por definição, um casamento estável é aquele em que não há pessoas em casamentos diferentes que prefiram umas às outras. Isto é, se uma pessoa, digamos x_1 , é casada com x_2 e prefere a pessoa y_2 do sexo oposto — que, por sua vez, é casada com y_1 — então y_2 não deve preferir a pessoa x_1 em detrimento de seu parceiro y_1 . O algoritmo de Gale and Shapley (1962) que garante o casamento estável pode ser descrito da seguinte forma¹:

1. Cada homem propõe casamento para a mulher que ele mais prefere e que ainda não o rejeitou, desde que a mulher seja preferível a ficar solteiro — caso contrário, o homem fica solteiro e não faz mais propostas.
2. Cada mulher aceita o casamento com o homem que seja mais preferível a ficar solteira — caso contrário, ela fica solteira —, e troca de marido se o homem que propõe é preferível ao seu atual parceiro.
3. Volte ao passo 1 se ainda existirem homens solteiros involuntários ou termine o algoritmo.

Como dito, o algoritmo garante a estabilidade dos casamentos e, ainda, o melhor casamento estável para os homens.

11.2 Implementação Computacional

O programa é composto apenas de uma função que implementa o *matching* entre homens e mulheres baseado no algoritmo descrito acima. Podemos defini-la como no código abaixo. Depois, todas as linhas do algoritmo serão colocadas dentro desta função. Observe que dadas as preferências de homens e mulheres uns sobre os outros, a função retorna os casais estáveis formados. Após a descrição do algoritmo, mostraremos um exemplo de preferências e os casais formados a partir destas.

```
1 def matching_com_solteiros(preferencia_H, preferencia_M):  
2  
3     return casais
```

Note que `preferencia_H` e `preferencia_M` são matrizes (lista de listas) em que a linha i representa o homem ou a mulher $i \in \{0, 1, \dots, n - 1\}$, e as colunas, para uma dada linha i , representam a ordenação da pessoa i sobre o sexo oposto e sobre a opção de ficar solteiro. Portanto, a matriz de preferência terá n linhas e $n + 1$ colunas. Por exemplo, a matriz abaixo representa as preferências de um determinado sexo.

¹Originalmente, o algoritmo não considera a possibilidade das pessoas ficarem solteiras. Adaptamos o algoritmo para incluir este caso.

```

1 preferencia = [[0, 1, 2, -1],
2               [1, -1, 2, 0],
3               [2, 1, 0, -1]]

```

Suponhamos que ela represente as preferências dos homens. Assim, o homem 0 (linha 0) tem a ordenação de suas preferências dada por

$$0 \succ_0 1 \succ_0 2 \succ_0 -1,$$

em que -1 representa a opção de ficar solteiro e a relação $a \succ_0 b$ significa “para o homem 0, a mulher a é estritamente preferida à mulher b ”. Já os homens 1 e 2 (linhas 1 e 2, respectivamente) têm as suas preferências dadas por

$$\begin{aligned} 1 \succ_1 -1 \succ_1 2 \succ_1 0 \\ 2 \succ_2 1 \succ_2 0 \succ_2 -1. \end{aligned}$$

Explicada a representação das preferências, começaremos a construir a função (que definimos acima) com a definição de alguns objetos:

```

1  # preferencia_H são listas de listas, em que a i-ésima lista nos dá a
2  # preferência (ranking de mulheres) do homem i (o mesmo raciocínio vale
3  # para preferencia_M).
4
5  # Garante que o número de homens deve ser igual ao de mulheres:
6  assert len(preferencia_H) == len(preferencia_M)
7
8  n = len(preferencia_H) # n = numero de mulheres e homens
9  casais = [] # registra os casais formados. É uma lista de listas.
10 h_solteiros = [h for h in range(n)] # Registra os homens solteiros.
11 m_solteiras = [m for m in range(n)] # Registra as mulheres solteiras.
12
13 # Lista para a situação dos casais (mulher = coluna j; homem = linha i)
14 # 0 = mulher j não recebeu proposta
15 # 1 = mulher j recebeu proposta do homem j
16 # 2 = mulher j aceitou a proposta do homem j
17 estado = np.zeros((n, n), dtype = int) # Ajuda na formatação do gráfico
18
19 # Cria a estrutura do gráfico que usaremos depois
20 fig, ax = plt.subplots(nrows = 2, ncols = 5, sharex = False,
21                        sharey = True, tight_layout = True,
22                        figsize = (15, 8), dpi = 1000)

```

O leitor deve atentar-se aos comentários no código para que obtenha explicações dos objetos definidos. Após esta etapa, queremos estabelecer um *loop* para as propostas dos homens, de modo que cada um deles irá propor casamento para todas as mulheres que ainda não o rejeitou até que não haja mais homens solteiros de modo involuntário (lembre-se: a depender das preferências, eles podem *desejar* ser solteiros!). Dentro deste *loop*, devemos permitir que os homens façam as propostas e as mulheres, por sua vez, aceitam ou não, sempre conforme suas preferências. Em suma, implementaremos os passos 1, 2 e 3 descritos no algoritmo exposto na seção anterior. Em seguida, ainda dentro do *loop*, plotaremos os gráficos que indicarão as propostas feitas e os aceites das mulheres. O código responsável por tal é dado a seguir:

```
1      ##### Enquanto houver homem solteiro, haverá novas propostas #####
2
3      while h_solteiros:
4
5
6          for m in range(n): # quando m mudar, significa nova rodada (todos
7                              # farão proposta pra mulher mais preferida, depois
8                              # para a segunda mais preferida, e assim sucess.)
9
10             propostas = [] # Lista de listas. O elemento [h, m] significa que
11                             # o homem h fez proposta para a mulher m.
12
13             solteiros_indice = []
14             solteiras_indice = []
15
16             ##### Mulheres que querem ficar solteiras: #####
17
18             for k in m_solteiras:
19
20                 if preferencia_M[k][m] == -1:
21
22                     casais.append([-1, k])
23                     solteiras_indice.append(k)
24
25             for j in solteiras_indice:
26
27                 m_solteiras.remove(j)
28
29             ##### rodada de propostas #####
30
31             for h in h_solteiros: # faz proposta ou escolhe ficar solteiro
32
33                 if (preferencia_H[h][m] != -1): # faz proposta
34
```

```

35         propostas.append([h, preferencia_H[h][m]])
36
37         if estado[h][preferencia_H[h][m]] != 2:
38             estado[h][preferencia_H[h][m]] = 1
39
40         else: # fica solteiro
41
42             casais.append([h, -1])
43             solteiros_indice.append(h)
44
45     for j in solteiros_indice:
46
47         h_solteiros.remove(j)
48
49     ##### Análise de propostas #####
50
51     for h, m2 in propostas:
52
53         # se a mulher está casada:
54         if m2 in [casais[k][1] for k in range(len(casais))]:
55
56             # ----- achando o índice do parceiro da mulher m2 ----- #
57
58             for i in range(len(casais)):
59
60                 if casais[i][1] == m2:
61
62                     indice_parceiro = i
63
64             # ----- #
65
66             # se o homem fazendo a proposta é preferido ao atual
67             # parceiro da mulher, então ela troca de parceiro:
68             if (preferencia_M[m2].index(casais[indice_parceiro][0]) >
69                 preferencia_M[m2].index(h)):
70
71                 estado[h][m2] = 2
72                 estado[casais[indice_parceiro][0]][m2] = 0
73
74                 h_solteiros.append(casais[indice_parceiro][0])
75
76                 casais[indice_parceiro][0] = h
77                 h_solteiros.remove(h)
78                 h_solteiros.sort()
79
80     # Se a mulher está solteira, ela aceita a proposta:

```

```

81         else:
82
83             # Mas apenas se ela não prefere estar solteira a estar com
84             # h
85             if (preferencia_M[m2].index(-1) >
86                 preferencia_M[m2].index(h)):
87
88                 casais.append([h, m2])
89                 h_solteiros.remove(h)
90                 h_solteiros.sort()
91                 m_solteiras.remove(m2)
92                 m_solteiras.sort()
93
94                 estado[h][m2] = 2
95
96             # Removendo -1 da lista de solteiros caso tenha algum
97             if -1 in h_solteiros:
98
99                 h_solteiros.remove(-1)
100
101             # Removendo [-1, -1] da lista casais pois não significa nada
102             if [-1, -1] in casais:
103
104                 casais.remove([-1,-1])
105
106
107             #####
108             # Gráfico: mostra casais formados a cada rodada #
109             #####
110
111             # Código RGB para as cores
112             cores = {0: np.array([255, 255, 255]), # branco
113                     1: np.array([100, 100, 255]), # azul (não usaremos)
114                     2: np.array([255, 100, 100])} # vermelho
115
116             # Matriz que mapeia o estado nas cores correspondentes
117             mapa = []
118
119             for homem in estado:
120
121                 linha = []
122
123                 for situacao in homem:
124
125                     cor = cores[situacao]
126                     linha.append(cor)

```

```

127
128         mapa.append(linha)
129
130     #Labels para os homens e mulheres
131     label_homens = ["H0", "H1", "H2", "H3", "H4", "H5", "H6", "H7",
132                    "H8", "H9"]
133     label_mulheres = ["M0", "M1", "M2", "M3", "M4", "M5", "M6", "M7",
134                      "M8", "M9"]
135
136     if m in range(5):
137
138         ax[0][m].matshow(mapa)
139         # Cria ticks para traçar as linhas do grid
140         ax[0][m].set_xticks(np.arange(-.5, len(label_homens)),
141                             minor = True)
142         ax[0][m].set_yticks(np.arange(-.5, len(label_mulheres)),
143                             minor = True)
144         ax[0][m].grid(True, which = 'minor', color = 'k', ls = ':',
145                       linewidth = 1)
146         # Define a posição dos ticks principais e altera labels
147         ax[0][m].set_xticks(np.arange(len(label_homens)))
148         ax[0][m].set_yticks(np.arange(len(label_mulheres)))
149         ax[0][m].set_xticklabels(label_mulheres)
150         ax[0][m].set_yticklabels(label_homens)
151         ax[0][m].title.set_text(f"Rodada {m+1}")
152
153     if m == 5:
154
155         ax[1][0].matshow(mapa)
156         # Cria ticks para traçar as linhas do grid
157         ax[1][0].set_xticks(np.arange(-.5, len(label_homens)),
158                             minor = True)
159         ax[1][0].set_yticks(np.arange(-.5, len(label_mulheres)),
160                             minor = True)
161         ax[1][0].grid(True, which = 'minor', color = 'k', ls = ':',
162                       linewidth = 1)
163         # Define a posição dos ticks principais e altera labels
164         ax[1][0].set_xticks(np.arange(len(label_homens)))
165         ax[1][0].set_yticks(np.arange(len(label_mulheres)))
166         ax[1][0].set_xticklabels(label_mulheres)
167         ax[1][0].set_yticklabels(label_homens)
168         ax[1][0].title.set_text(f"Rodada {m+1}")
169
170     if m == 6:
171
172         ax[1][1].matshow(mapa)

```

```

173         # Cria ticks para traçar as linhas do grid
174         ax[1][1].set_xticks(np.arange(-.5, len(label_homens)),
175                             minor = True)
176         ax[1][1].set_yticks(np.arange(-.5, len(label_mulheres)),
177                             minor = True)
178         ax[1][1].grid(True, which = 'minor', color = 'k', ls = ':',
179                       linewidth = 1)
180         # Define a posição dos ticks principais e altera labels
181         ax[1][1].set_xticks(np.arange(len(label_homens)))
182         ax[1][1].set_yticks(np.arange(len(label_mulheres)))
183         ax[1][1].set_xticklabels(label_mulheres)
184         ax[1][1].set_yticklabels(label_homens)
185         ax[1][1].title.set_text(f"Rodada {m+1}")
186
187     if m == 7:
188
189         ax[1][2].matshow(mapa)
190         # Cria ticks para traçar as linhas do grid
191         ax[1][2].set_xticks(np.arange(-.5, len(label_homens)),
192                             minor = True)
193         ax[1][2].set_yticks(np.arange(-.5, len(label_mulheres)),
194                             minor = True)
195         ax[1][2].grid(True, which = 'minor', color = 'k', ls = ':',
196                       linewidth = 1)
197         # Define a posição dos ticks principais e altera labels
198         ax[1][2].set_xticks(np.arange(len(label_homens)))
199         ax[1][2].set_yticks(np.arange(len(label_mulheres)))
200         ax[1][2].set_xticklabels(label_mulheres)
201         ax[1][2].set_yticklabels(label_homens)
202         ax[1][2].title.set_text(f"Rodada {m+1}")
203
204     if m == 8:
205
206         ax[1][3].matshow(mapa)
207         # Cria ticks para traçar as linhas do grid
208         ax[1][3].set_xticks(np.arange(-.5, len(label_homens)),
209                             minor = True)
210         ax[1][3].set_yticks(np.arange(-.5, len(label_mulheres)),
211                             minor = True)
212         ax[1][3].grid(True, which = 'minor', color = 'k', ls = ':',
213                       linewidth = 1)
214         # Define a posição dos ticks principais e altera labels
215         ax[1][3].set_xticks(np.arange(len(label_homens)))
216         ax[1][3].set_yticks(np.arange(len(label_mulheres)))
217         ax[1][3].set_xticklabels(label_mulheres)
218         ax[1][3].set_yticklabels(label_homens)

```

```

219         ax[1][3].title.set_text(f"Rodada {m+1}")
220
221     if m == 9:
222
223         ax[1][4].matshow(mapa)
224         # Cria ticks para traçar as linhas do grid
225         ax[1][4].set_xticks(np.arange(-.5, len(label_homens)),
226                             minor = True)
227         ax[1][4].set_yticks(np.arange(-.5, len(label_mulheres)),
228                             minor = True)
229         ax[1][4].grid(True, which = 'minor', color = 'k', ls = ':',
230                       linewidth = 1)
231         # Define a posição dos ticks principais e altera labels
232         ax[1][4].set_xticks(np.arange(len(label_homens)))
233         ax[1][4].set_yticks(np.arange(len(label_mulheres)))
234         ax[1][4].set_xticklabels(label_mulheres)
235         ax[1][4].set_yticklabels(label_homens)
236         ax[1][4].title.set_text(f"Rodada {m+1}")

```

Observe que o código para a formação dos casais não é tão extenso quanto parece. Isto porque a construção (que começa em torno da linha 110) do gráfico das rodadas de propostas (que mostraremos abaixo) ocupa boa parte do espaço do programa.

Após a implementação da função `matching_com_solteiros()` descrita acima, podemos usar um exemplo de preferência para observar os casais formados. O programa completo, que pode ser conferido na seção A.10 do apêndice, oferece um exemplo em que não há a opção de estar solteiro (ou, de forma equivalente, todos preferem estar acompanhados a estar solteiros). Aqui, no entanto, analisaremos apenas o caso mais geral em que existe a possibilidade de pessoas (homens ou mulheres) preferirem ficar solteiros ao invés de estar casadas com outras pessoas.

Assim, usaremos as seguintes matrizes de preferências:

```

1      #####
2      #                               #
3      #####
4
5      # Com solteiros:
6
7      preferencia_H = \
8          [[0, 4, 2, 8, -1, 9, 3, 5, 1, 7, 6], [2, -1, 7, 0, 3, 4, 5, 1, 9, 8, 6],
9           [7, 4, 0, 3, 1, 5, 8, 6, -1, 2, 9], [-1, 8, 5, 3, 6, 7, 4, 9, 1, 2, 0],
10          [9, 3, 1, 2, 5, -1, 4, 0, 8, 7, 6], [1, 0, -1, 3, 6, 4, 8, 2, 9, 7, 5],
11          [6, 4, 8, 1, 2, 0, 3, 7, 9, 5, -1], [0, 4, 7, 5, 8, 2, 9, 1, 6, 3, -1],
12          [7, 2, -1, 3, 6, 1, 0, 5, 8, 9, 4], [0, 5, 9, 6, 4, -1, 1, 3, 2, 8, 7]]

```



```

13
14 preferencia_M = \
15     [[-1, 1, 5, 9, 6, 8, 0, 3, 4, 2, 7], [1, 0, -1, 2, 5, 6, 3, 8, 4, 9, 7],
16     [5, -1, 1, 4, 6, 7, 2, 8, 0, 3, 9], [5, 9, 2, 0, -1, 8, 7, 6, 3, 1, 4],
17     [9, 7, 5, 3, 0, 6, 2, 4, 8, 1, -1], [1, 0, 4, -1, 8, 9, 3, 5, 6, 2, 7],
18     [9, 6, 7, 5, 1, 0, 2, 4, 3, -1, 8], [6, 9, 1, 0, -1, 8, 3, 7, 4, 2, 5],
19     [8, 2, 7, 6, 5, 1, -1, 0, 4, 9, 3], [4, -1, 7, 6, 0, 1, 9, 2, 8, 5, 3]]

```

Para rodar a função que construímos com estas preferências específicas, basta chamá-la com os argumentos `preferencia_H` e `preferencia_M` definidos acima:

```

1 casais_com_solteiros = matching_com_solteiros(preferencia_H, preferencia_M)

```

Então, armazenaremos na variável `casais_com_solteiros` a matriz de casais formados a partir destas preferências. Além disso, o programa mostra uma representação gráfica de cada rodada de propostas. Se representarmos as propostas dos homens para as mulheres pela cor azul e os aceites pela cor laranja, a figura 11.1 mostra o movimento de propostas e aceites ao longo das rodadas. Note que os homens são listados no eixo vertical e as mulheres no horizontal.

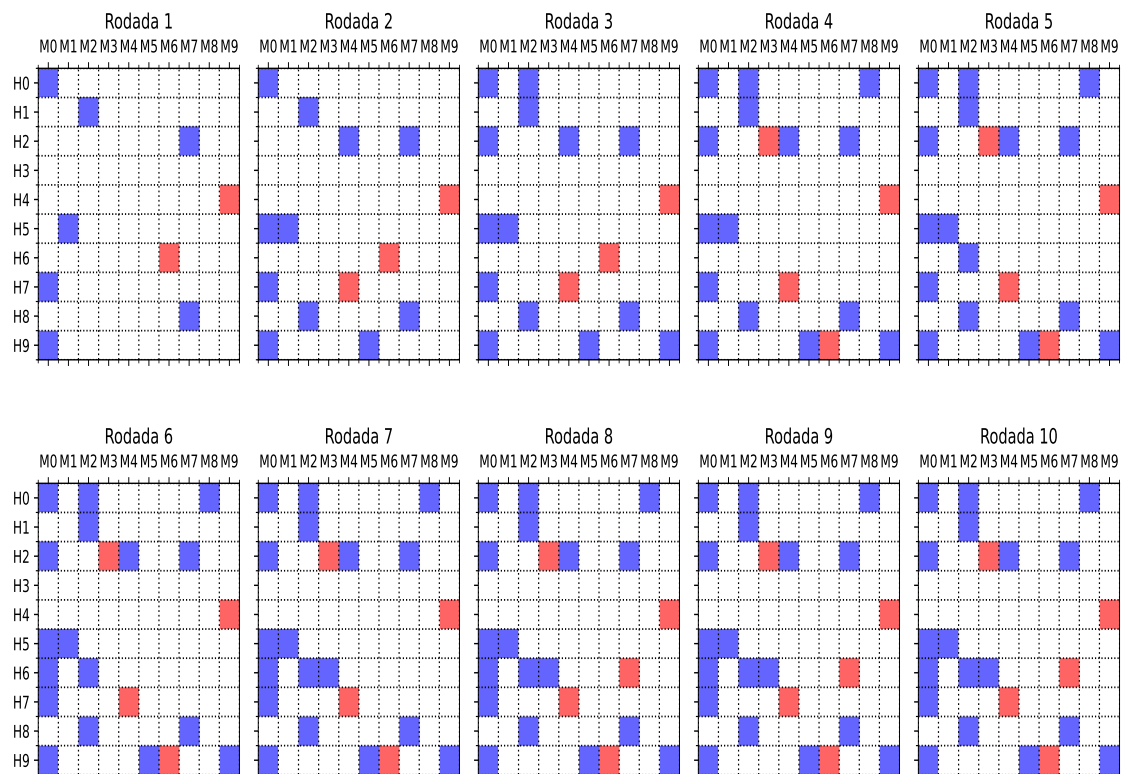


Figura 11.1: Propostas e casais formados por rodada.

Pela figura acima podemos ver que os únicos casais formados são dados pelos pares $(4, 9)$, $(2, 3)$, $(7, 4)$, $(9, 6)$ e $(6, 7)$, em que a primeira entrada do vetor representa o homem e a segunda entrada indica a mulher. Portanto, o conjunto dos homens solteiros poderia ser representado por $\{0, 1, 3, 5, 8\}$ e o conjunto das mulheres solteiras poderia ser dado por $\{0, 1, 2, 5, 8\}$.

Capítulo 12

Equilíbrio Geral

12.1 O modelo

12.2 Implementação Computacional

Capítulo 13

Desenho de Mecanismo

13.1 O modelo

13.2 Implementação Computacional

Referências Bibliográficas

N. Mankiw. Principles of economics. South-Western College Pub, 6th edition., 2011. ISBN 0538453052.

Hal R. Varian. Microeconomia: uma abordagem moderna. Elsevier Editora Ltda., trad. da 8ª ed., 2012. ISBN 978-85-352-5143-2.

T.J Sargent and J. Stachurski. Quantitative Economics with Python. Open Source, 2021. Disponível em <https://python.quantecon.org/intro.html>.

N. Mankiw. Macroeconomia. LTC, trad. da 8ª ed., 2015. ISBN 9788521615767.

W. Nicholson and C. Snyder. Microeconomic Theory: Basic Principles & Extensions. Cengage Learning, 12ª ed., 2017. ISBN-13: 978-1-305-50579-7.

S. Tadelis. Game theory: an introduction. Princeton University Press, 2013. ISBN 978-0-691-12908-2.

J-J. Rousseau. Discourse on the origin of inequality. Dover Publications, 2004. ISBN 9780486434148.

David Gale and Lloyd Shapley. College admissions and the stability of marriage. American Mathematical Monthly, 69:9–15, 1962.

Apêndice A

Programas completos

Este apêndice exibe completamente os programas implementados no texto, mostrando algumas partes dos códigos que foram omitidas (como, por exemplo, a implementação dos pacotes usuais) para não cansar o leitor com repetições desnecessárias. Deste modo, a consulta deste apêndice pode servir para sanar possíveis dúvidas que possam surgir durante o acompanhamento do texto principal.

A.1 Oferta e Demanda

```
1  # -*- coding: utf-8 -*-
2  """
3  @author: Matheus L. Carrijo
4  """
5
6  import matplotlib.pyplot as plt #importing graph package
7  plt.figure(figsize=(9, 6), dpi=100) #set default figure size in a better way
8  import numpy as np #importing numpy
9
10 def demanda(p):
11     assert a < 0 and b > 0
12     return a*p + b
13
14 def oferta(p):
15     assert c > 0 and d > 0 and b > d
16     # d in (0,b) é suficiente pra equilibrio
17     return c*p + d
18
19 def excesso_demanda(p):
20     # Setting excess demand function. It must be clear that equilibrium requires
21     # excesso_demanda(p) = 0, for some especific p
```

```

22     return demanda(p) - oferta(p)
23
24 parametro_bool = False
25
26 teste = "diferente de y e n"
27 while teste != "y" and teste != "n":
28     print("\nVocê quer entrar com os parâmetros das funções oferta", end = "")
29     teste = input("e demanda? [y/n]: ")
30
31 if teste == 'y':
32     parametro_bool = True
33
34 if parametro_bool:
35
36     print("\nDadas as funções lineares de oferta e demanda,", end = " ")
37     print("queremos calcular o preço e a quantidade de equilíbrio")
38     a = float(input("Digite o coeficiente de inclinação da demanda: "))
39     b = float(input("Digite a constante da curva de demanda: "))
40     c = float(input("Digite o coeficiente de inclinação da oferta: "))
41     d = float(input("Digite a constante da curva de oferta: "))
42
43     p_grid = np.linspace(-1000, 1000, 100000)
44     q_grid = np.linspace(-1000, 1000, 100000)
45
46     E_abs = []
47     for i in p_grid:
48         E_abs.append(abs(excesso_demanda(i)))
49
50     indice_min = E_abs.index(min(E_abs))
51     p_equilibrio = p_grid[indice_min]
52     q_equilibrio = oferta(p_equilibrio)
53
54     print("O preço e a quantidade de equilíbrio são", end = "")
55     print(f", respectivamente, {p_equilibrio: .2f} e {q_equilibrio: .2f}")
56
57     q1, q2, q3 = demanda(p_grid), oferta(p_grid), excesso_demanda(p_grid)
58     plt.plot(q1, p_grid, color="blue", linewidth=1.0, ls = "dashed")
59     plt.plot(q2, p_grid, color="green", linewidth=1.0)
60     plt.plot(q3, p_grid, color="red", linewidth=1.0, ls = "dotted")
61     plt.xlim(d, b) # entre -5 e 5 por exemplo
62     plt.ylim(-d/c, (-b) / a)
63     plt.legend([f'Curva de Demanda: D = {a}p + {b}',
64                f'Curva de Oferta: S = {c}p + {d}',
65                'Curva do excesso de demanda: E = D - S'])
66     plt.ylabel("Preço", fontsize = 15)
67     plt.xlabel("Quantidade", fontsize = 15)

```

```

68     plt.show()
69
70 else:
71     print("\nFaremos o cálculo para a função demanda  $D = -2p + 7$ ", end = " ")
72     print("e oferta  $S = 4p + 1$ \n")
73     a, b, c, d = -2, 7, 4, 1
74
75     p_grid = np.linspace(-1000, 1000, 1000)
76     q_grid = np.linspace(-1000, 1000, 1000)
77
78     E_abs = []
79     for i in p_grid:
80         E_abs.append(abs(excesso_demanda(i)))
81
82     indice_min = E_abs.index(min(E_abs))
83     p_equilibrio = p_grid[indice_min]
84     q_equilibrio = oferta(p_equilibrio)
85
86     print("O preço e a quantidade de equilíbrio são", end = "")
87     print(f", respectivamente, {p_equilibrio: .2f} e {q_equilibrio: .2f}")
88
89     q1, q2, q3 = demanda(p_grid), oferta(p_grid), excesso_demanda(p_grid)
90     plt.plot(q1, p_grid, color="blue", linewidth=3.0, ls = "dashed")
91     plt.plot(q2, p_grid, color="green", linewidth=3.0)
92     plt.plot(q3, p_grid, color="red", linewidth=3.0, ls = "dotted")
93     plt.xlim(d, b) # entre -5 e 5 por exemplo
94     plt.ylim(-d/c, (-b) / a)
95     plt.legend([f'Curva de Demanda:  $D = \{a\}p + \{b\}$ ',
96                f'Curva de Oferta:  $S = \{c\}p + \{d\}$ ',
97                'Curva do excesso de demanda:  $E = D - S$ '])
98     plt.ylabel("Preço", fontsize = 15)
99     plt.xlabel("Quantidade", fontsize = 15)
100    plt.show()

```

A.2 Crescimento Econômico: Modelo de Solow

```

1  # -*- coding: utf-8 -*-
2  """
3  @author: Matheus L. Carrijo
4  """
5
6  import matplotlib.pyplot as plt #importing graph package

```

```

7 plt.figure(figsize=(9, 6), dpi=100) #set default figure size in a better way
8 #plt.rcParams["figure.figsize"] = (8, 6) #set default figure size
9 import numpy as np #importing numpy
10
11 # Set the Cobb-Douglas production function:
12 def produto_agregado(k_agregado):
13     return np.sqrt(k_agregado*l_agregado)
14
15 # Set the Cobb-Douglas production function per worker:
16 def produto_por_trabalhador(k_agregado):
17     return produto_agregado(k_agregado)/l_agregado
18
19 # Depreciation is a linear function of k per worker, i.e.,  $k=K/L$ 
20 def depreciacao(k_agregado):
21     return taxa_depreciacao*(k_agregado/l_agregado)
22
23 def investimento(k_agregado):
24     return taxa_poupanca*produto_por_trabalhador(k_agregado)
25
26 def consumo(k_agregado):
27     return (1 - taxa_poupanca)*produto_por_trabalhador(k_agregado)
28
29 def k_variacao(k_agregado):
30     return (taxa_poupanca*produto_por_trabalhador(k_agregado)
31             - depreciacao(k_agregado))
32
33 resposta = "qualquer_coisa"
34 while resposta != "y" and resposta != "n":
35     print("Deseja entrar com os parâmetros? [y/n]", end = " ")
36     resposta = input(" ")
37
38 if resposta == "y":
39     taxa_depreciacao, taxa_poupanca = 2, 2 # just to enter the while
40
41     # The rates must be between 0 and 1. The following loop guarantee this.
42     while (taxa_depreciacao < 0 or taxa_depreciacao > 1
43            or taxa_depreciacao == 0):
44         print("A taxa de depreciação não nula deve estar entre", end = " ")
45         print("0 e 1", end = " ")
46         taxa_depreciacao = float(input("Entre com a taxa de depreciação: "))
47
48     while taxa_poupanca < 0 or taxa_poupanca > 1 or taxa_poupanca == 0:
49         print("A taxa de poupança não nula deve estar entre 0 e 1.", end = ' ')
50         taxa_poupanca = float(input("Entre com a taxa de poupança: "))
51
52

```

```

53     print("Entre com o nível de mão de obra desta economia: ", end = " ")
54     l_agregado = float(input(" "))
55
56     print("Entre com o nível inicial de capital agregado desta economia: ",
57           end = " ")
58     k_t0 = float(input(" "))/l_agregado
59
60     tolerancia = 0.001 # ask the user for enter this parameter?
61     norma = 10 #just to enter in the while
62     while norma > tolerancia:
63         k_t1 = (taxa_poupanca*produto_por_trabalhador(k_t0) +
64               k_t0*(1-taxa_depreciacao))
65         norma = abs(k_t1 - k_t0)
66         k_t0 = k_t1
67
68     print(f"O produto de equilibrio é k* = {k_t1: .4}")
69
70     # Here is the graph settings
71     k_grid = np.linspace(0, 2*k_t1, 1000)
72     plt.plot(k_grid, depreciacao(k_grid), color="blue", linewidth=3.0,
73             ls = "dashed")
74     plt.plot(k_grid, investimento(k_grid), color="green", linewidth=3.0)
75     plt.xlim(0, 2*k_t1) # entre -5 e 5 por exemplo
76     plt.ylim(0, 2*investimento(k_t1))
77     plt.legend([f'Depreciação, {taxa_depreciacao}k',
78               f'Investimento, {taxa_poupanca}f(k)'])
79     plt.ylabel("Investimento e Depreciação", fontsize = 15)
80     plt.xlabel("Capital por trabalhador, k", fontsize = 15)
81     plt.show()
82
83 elif resposta == "n":
84     print("Faremos uma exemplo em que a taxa de poupança é 30%", end = " ")
85     print("a taxa de depreciação é 10%, e que a economia comece", end = " ")
86     print("com uma relação de 4 capital por trabalhador")
87
88     taxa_depreciacao, taxa_poupanca, l_agregado, k_t0 = 0.1, 0.3, 1, 4
89
90     tolerancia = 0.001 # ask the user for enter this parameter?
91     norma = 10 #just to enter in the while
92     while norma > tolerancia:
93         k_t1 = (taxa_poupanca*produto_por_trabalhador(k_t0) +
94               k_t0*(1-taxa_depreciacao))
95         norma = abs(k_t1 - k_t0)
96         k_t0 = k_t1
97
98     print(f"O produto de equilibrio é k* = {k_t1: .4}")

```

```

99
100     # Here is the graph settings
101     k_grid = np.linspace(0, 2*k_t1, 1000)
102     plt.plot(k_grid, depreciacao(k_grid), color="blue", linewidth=3.0,
103              ls = "dashed")
104     plt.plot(k_grid, investimento(k_grid), color="green", linewidth=3.0)
105     plt.xlim(0, 2*k_t1) # entre -5 e 5 por exemplo
106     plt.ylim(0, 2*investimento(k_t1))
107     plt.legend([f'Depreciação, {taxa_depreciacao}k',
108                f'Investimento, {taxa_poupanca}f(k)'])
109     plt.ylabel("Investimento e Depreciação", fontsize = 15)
110     plt.xlabel("Capital por trabalhador, k", fontsize = 15)
111     plt.show()

```

A.3 Renda Nacional: Modelo IS-LM

```

1  # -*- coding: utf-8 -*-
2  """
3  @author: Matheus L. Carrijo
4  """
5
6  import matplotlib.pyplot as plt #importing graph package
7  plt.figure(figsize=(9, 6), dpi=100) #set default figure size
8  import numpy as np
9
10 def investimento(taxa_juros):
11
12     """
13     Dada a taxa de juros, calcula o nível de investimento. Deve ser
14     negativamente relacionada com a taxa de juros. Por simplicidade,
15     consideramos o caso linear.
16     """
17
18     if a < 0 and b > 0:
19         return a*taxa_juros + b
20
21 def demanda_monetaria(taxa_juros, produto):
22
23     """
24     Modelamos a demanda monetária tentando incorporar linearmente os motivos
25     precaução, especulação, e transação, respectivamente pelas variáveis
26     e, c, d.

```

```

27     """
28
29     return c*produto + d*taxa_juros + e
30
31 def consumo(produto):
32
33     """
34     Consideraremos o consumo como uma função linear da renda disponível, de
35     modo que o coeficiente de inclinação é a propensão marginal a consumir e
36     a constante é o consumo autónomo.
37     """
38
39     return (propensao_marginal_a_consumir * (produto - aliquota_imposto)
40            + consumo_autonomo)
41
42 def produto_IS(taxa_juros):
43
44     """
45     Construímos a curva IS através da equação  $Y = C + I + G$ , em que  $C$  é a
46     função consumo (que depende da renda disponível),  $I$  é o investimento,
47     dependente da taxa de juros, e  $G$  são os gastos do governo. Como o consumo
48     depende do nível de renda, esta equação é manipulada até que tenhamos uma
49     expressão do produto em relação à taxa de juros
50     """
51
52     I = investimento(taxa_juros)
53
54     return ((consumo_autonomo + I + gastos_governo) /
55            (1 - propensao_marginal_a_consumir
56             * (1 - aliquota_imposto)))
57
58
59 def produto_LM(taxa_juros):
60
61     """
62     Construímos a curva LM através da equação
63     oferta_monetaria = demanda_monetaria =  $c*produto + d*taxa\_juros + e$ ,
64     em que a oferta é exógena e a demanda calculada pela função. Então, a
65     expressão pode ser manipulada até que tenhamos uma função de  $Y$  em função
66     da taxa de juros.
67     """
68
69     return ((oferta_monetaria/nivel_precos - e) / c - taxa_juros * (d / c))
70
71 def excesso_ISLM(taxa_juros):
72

```



```

73     """
74     Função para calcular o equilíbrio fazendo excesso_ISLM = 0, isto é,
75     o par (r*, Y*) ótimo que nos dá a intersecção entre as curvas IS e LM.
76     """
77
78     return produto_IS(taxa_juros) - produto_LM(taxa_juros)
79
80 resposta = "qualquer_coisa"
81
82 while resposta != "y" and resposta != "n":
83
84     print("Deseja entrar com os parâmetros? [y/n]", end = " ")
85     resposta = input(" ")
86
87 if resposta == "n":
88
89     a, b, propensao_marginal_a_consumir, consumo_autonomo = -20, 5, 0.3, 100
90     gastos_governo, aliquota_imposto, c, d, e = 50, .2, 10, -300, 50
91     nivel_precos, oferta_monetaria = 10, 100
92
93     print("\nFaremos uma exemplo em que a função investimento é", end = " ")
94     print(f"dada por {a}r + {b};", end = " ")
95     print("a função demanda monetária, por", end = " ")
96     print(f"{c}Y + {d}i + {e};", end = " ")
97     print("a função consumo, por", end = " ")
98     print(f"{propensao_marginal_a_consumir}Y + {consumo_autonomo};", end = " ")
99     print(f"e a política fiscal por G = {gastos_governo} e", end = " ")
100    print(f"T = {aliquota_imposto}Y.", end = " ")
101    print("Ainda, a oferta monetária real é dada por M/P =", end = " ")
102    print(f"{oferta_monetaria/nivel_precos}.")
103
104
105    # Calculo do equilibrio
106
107    taxa_juros_grid = np.linspace(0, 1000, 100000)
108
109    Excesso_abs = []
110
111    for i in taxa_juros_grid:
112
113        Excesso_abs.append(abs(excesso_ISLM(i)))
114
115    indice_min = Excesso_abs.index(min(Excesso_abs))
116    juros_equilibrio = taxa_juros_grid[indice_min]
117    produto_equilibrio = produto_IS(juros_equilibrio)
118

```

```

119     print("\nPortanto, a taxa de juros e produto de equilíbrio são,", end = "")
120     print(f" respect., {juros_equilibrio: .2f} e {produto_equilibrio: .2f}")
121
122     valores_produto = []
123
124     for i in taxa_juros_grid:
125
126         valores_produto.append(abs(produto_IS(i)))
127
128     y = taxa_juros_grid[np.argmin(valores_produto)]
129
130     # graph settings
131
132     plt.plot(produto_IS(taxa_juros_grid), taxa_juros_grid, color="blue",
133              linewidth=3.0, ls = "dotted")
134     plt.plot(produto_LM(taxa_juros_grid), taxa_juros_grid, color="green",
135              linewidth=3.0, ls = "dashed")
136     plt.ylim(0, y)
137     plt.xlim(min(produto_LM(taxa_juros_grid)),
138              max(produto_IS(taxa_juros_grid)))
139     plt.legend(['Curva IS', 'Curva LM'])
140     # plt.title("Modelo IS-LM", fontsize = 20)
141     plt.ylabel("Taxa de juros", fontsize = 15)
142     plt.xlabel("Produto, Y", fontsize = 15)
143     plt.show()
144
145 elif resposta == "y":
146
147     # Entrando com os parâmetros da função investimento:
148
149     a, b = 1, -1 # Valor para entrar no loop
150
151     while a >= 0:
152
153         print("\n0 coeficiente de inclinação da função", end = " ")
154         print("investimento deve ser negativo.")
155         print("Entre com o coeficiente de inclinação da função", end = " ")
156         a = float(input("linear de investimento: "))
157
158     while b < 0:
159
160         print("\nA constante da função investimento deve ser", end = " ")
161         print("positiva.\n")
162         print("Entre com a constante da função linear de", end = " ")
163         b = float(input("investimento: "))
164

```

```

165     # Entrando com os parâmetros da função demanda monetária:
166
167     c, d, e = -1, 1, -1 # Valor para entrar no loop
168
169     while c < 0:
170
171         print("\nO coeficiente de transação da demanda monetária", end = " ")
172         print("deve ser positivo.")
173         print("Entre com o coeficiente de transação da demanda", end = " ")
174         c = float(input("monetária: "))
175
176     while d > 0:
177
178         print("\nO coeficiente de especulação da demanda monetária", end = " ")
179         print("deve ser negativo.")
180         print("Entre com o coeficiente de especulação da demanda", end = " ")
181         d = float(input("monetária: "))
182
183     while e < 0:
184
185         print("\nO coeficiente de precaução da demanda monetária", end = " ")
186         print("deve ser positivo.")
187         print("Entre com o coeficiente de precaução da demanda", end = " ")
188         e = float(input("monetária: "))
189
190     # Entrando com a oferta monetária real:
191
192     nivel_precos, oferta_monetaria = -1, -1 # Valor para entrar no loop
193
194     while nivel_precos <= 0:
195
196         print("\nO nível de preços deve ser positivo.")
197         print("Entre com o nível de preços:", end = " ")
198         nivel_precos = float(input(""))
199
200     while oferta_monetaria <= 0:
201
202         print("\nA oferta monetária deve ser positiva.")
203         print("Entre com a oferta monetária:", end = " ")
204         oferta_monetaria = float(input(""))
205
206     # Entrando com os parâmetros da função consumo:
207
208     propensao_marginal_a_consumir, consumo_autonomo = -1, -1
209
210     while (propensao_marginal_a_consumir < 0 or

```

```

211         propensao_marginal_a_consumir > 1):
212
213     print("\nA propensão marginal a consumir deve estar entre", end = " ")
214     print("0 e 1.")
215     print("Entre com a propensão marginal a consumir: ", end = " ")
216     propensao_marginal_a_consumir = float(input(""))
217
218     while consumo_autonomo < 0:
219
220         print("\nO consumo autônomo deve ser positivo.")
221         print("Entre com o consumo autônomo: ", end = " ")
222         consumo_autonomo = float(input(""))
223
224     # Entrando com a política fiscal:
225
226     gastos_governo, aliquota_imposto = -1, -1 # Valor para entrar no loop
227
228     while gastos_governo < 0:
229
230         print("\nOs gastos do governo devem ser não negativo.")
231         print("Entre com os gastos do governo: ", end = " ")
232         gastos_governo = float(input(""))
233
234     while aliquota_imposto < 0 or aliquota_imposto > 1:
235
236         print("\nA alíquota de imposto deve estar entre 0 e 1.")
237         print("Entre com os impostos cobrados pelo governo: ", end = " ")
238         aliquota_imposto = float(input(""))
239
240     taxa_juros_grid = np.linspace(0, 1000, 100000)
241
242     Excesso_abs = []
243
244     for i in taxa_juros_grid:
245
246         Excesso_abs.append(abs(excesso_ISLM(i)))
247
248     indice_min = Excesso_abs.index(min(Excesso_abs))
249     juros_equilibrio = taxa_juros_grid[indice_min]
250     produto_equilibrio = produto_IS(juros_equilibrio)
251
252     print("\nPortanto, a taxa de juros e produto de equilíbrio são,", end = "")
253     print(f" respect., {juros_equilibrio: .2f} e {produto_equilibrio: .2f}")
254
255     valores_produto = []
256

```

```

257     for i in taxa_juros_grid:
258
259         valores_produto.append(abs(produto_IS(i)))
260
261     y = taxa_juros_grid[np.argmin(valores_produto)]
262
263     # graph settings
264
265     plt.plot(produto_IS(taxa_juros_grid), taxa_juros_grid, color="blue",
266             linewidth=3.0, ls = "dotted")
267     plt.plot(produto_LM(taxa_juros_grid), taxa_juros_grid, color="green",
268             linewidth=3.0, ls = "dashed")
269     plt.ylim(0, y)
270     plt.xlim(min(produto_LM(taxa_juros_grid)),
271             max(produto_IS(taxa_juros_grid)))
272     plt.legend(['Curva IS', 'Curva LM'])
273     #plt.title("Modelo IS-LM", fontsize = 20)
274     plt.ylabel("Taxa de juros", fontsize = 15)
275     plt.xlabel("Produto, Y", fontsize = 15)
276     plt.show()

```

A.4 Desemprego: Lake Model

```

1  # -*- coding: utf-8 -*-
2  """
3  @author: Matheus L. Carrijo
4  """
5
6  #####
7  #                               Importando pacotes                               #
8  #####
9
10 import matplotlib.pyplot as plt #importing graph package
11 plt.figure(figsize=(9, 6), dpi=1000) #set default figure size
12 import numpy as np
13
14 #####
15 #                               Funções                               #
16 #####
17
18 def nivel_estacionario(empregados = 250_000, desempregados = 50_000,
19                       Alpha = .027, Lambda = .391):

```

```

20
21 #Definindo as variáveis com o valor dos parâmetros
22
23 forca_trabalho = empregados + desempregados
24
25 A = np.array([[1 - Lambda, Alpha ],
26               [Lambda      , 1 - Alpha]])
27 x_inicial = np.array([[desempregados/forca_trabalho],
28                       [empregados/forca_trabalho]])
29
30 tempo = []
31 eixo_taxa_desemprego = [x_inicial[0][0]]
32 eixo_taxa_emprego = [x_inicial[1][0]]
33 t = 0
34 tol = 1e-04
35 dif = tol + 1
36
37 # Calculando o nível estacionário
38
39 while dif > tol:
40
41     x = A @ x_inicial
42     tempo.append(t)
43     eixo_taxa_desemprego.append(x[0][0])
44     eixo_taxa_emprego.append(x[1][0])
45     dif = np.max(np.abs(x - x_inicial))
46     x_inicial = x
47     t += 1
48 tempo.append(t)
49
50 # Informando ao usuário
51
52 print(f"\nPara uma economia com {empregados} empregados,", end = " ")
53 print(f"{desempregados} desempregados, uma taxa de obtenção de", end = " ")
54 print(f"emprego de {100*Lambda: .2f}%, e uma taxa de demissão", end = " ")
55 print(f"de {100*Alpha: .2f}%, a taxa de desemprego no nível", end = " ")
56 print(f"estacionário é dada por {100*x[0][0]: .2f}% \n")
57
58 return x, tempo, eixo_taxa_desemprego, eixo_taxa_emprego
59
60
61 def individual_path(Lambda = 0.3, Alpha = .027, desempregado = True):
62     """
63
64     Parameters
65     -----

```

```

66     Lambda : float, optional
67         Taxa de obtenção de emprego. The default is 0.3.
68     Alpha : float, optional
69         Taxa de demissão. The default is .027.
70     desempregado : bool, optional
71         informa se os indivíduos estão desempregados. The default is True.
72
73     Returns
74     -----
75     tempo : list
76         lista em que a i-ésima entrada corresponde ao tempo que o i-esimo
77         trabalhador demorou pra encontrar emprego (caso esteja desempregado)
78         ou que demorou pra ficar desempregado (caso esteja empregado).
79
80     """
81
82     if desempregado:
83
84         tempo = []
85
86         j = 1
87         while j <= 1000: # amostra de 1000 individuos
88
89             encontrou_emprego = False
90             t = 0
91             while not encontrou_emprego:
92                 # enquanto a pessoa j não encontrar emprego, ela continua
93                 # procurando
94
95                 r = np.random.rand()
96
97                 if r <= Lambda: # desempregado encontra emprego
98
99                     encontrou_emprego = True
100
101                 t += 1
102
103                 tempo.append(t)
104                 j += 1
105
106     else:
107
108         tempo = []
109
110         j = 1
111         while j <= 1000: # amostra de 1000 individuos

```

```

112
113     ficou_desempregado = False
114     t = 0
115     while not ficou_desempregado: # fica no emprego até ser demitido
116
117         r = np.random.rand()
118
119         if r <= Alpha: # empregado ficou desempregado
120
121             ficou_desempregado = True
122             t += 1
123
124         t += 1
125
126     tempo.append(t)
127     j += 1
128
129     return tempo
130
131
132     #####
133     #                Uma economia com:                #
134     #                - empregados = 250_000            #
135     #                - desempregados = 50_000          #
136     #                - Alpha = .027                    #
137     #                - Lambda = .391                   #
138     #####
139
140 exemplo_padrao = nivel_estacionario()
141
142 fig, ax = plt.subplots(nrows = 2, ncols = 1, sharex = True)
143
144 # plot desemprego:
145
146 ax[0].plot(exemplo_padrao[1], exemplo_padrao[2], color = "blue", lw = 1.5)
147 ax[0].set_xlim(0, exemplo_padrao[1][-1])
148 ax[0].set_ylabel("Taxa de desemprego")
149 #ax[0].legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower center',
150 #             ncol=2, mode="expand", borderaxespad=0.)
151 ax[0].set_title(r'$(\alpha, \lambda) = (2.7\%, 39.1\%)$', fontsize = 10)
152
153 # plot emprego:
154
155 ax[1].plot(exemplo_padrao[1], exemplo_padrao[3], color = "green", lw = 1.5)
156 ax[1].set_xlabel("tempo")
157 ax[1].set_ylabel("Taxa de emprego")

```



```

158
159 #####
160 #          Variando parâmetros (estática comparativa)          #
161 #####
162
163 # Variando Lambda
164
165 estatica1 = nivel_estacionario(450_000, 50_000, .04, .3)
166 estatica2 = nivel_estacionario(450_000, 50_000, .04, .6)
167
168 # Variando Alpha
169
170 estatica3 = nivel_estacionario(450_000, 50_000, .03, .4)
171 estatica4 = nivel_estacionario(450_000, 50_000, .06, .4)
172
173
174 fig, ax = plt.subplots(nrows = 2, ncols = 1, sharex = True)
175
176 t_max = max(estatica1[1][-1], estatica2[1][-1], estatica3[1][-1],
177             estatica4[1][-1]) # limite eixo x
178
179 # plot desemprego:
180
181 ax[0].plot(estatica1[1], estatica1[2], color = "blue", lw = 1,
182            label = r'$(\alpha, \lambda) = (4\%, 30\%)$')
183 ax[0].plot(estatica2[1], estatica2[2], color = "green", lw = 1,
184            label = r'$(\alpha, \lambda) = (4\%, 60\%)$')
185 ax[0].plot(estatica3[1], estatica3[2], color = "red", lw = 1,
186            label = r'$(\alpha, \lambda) = (3\%, 40\%)$')
187 ax[0].plot(estatica4[1], estatica4[2], color = "black", lw = 1,
188            label = r'$(\alpha, \lambda) = (6\%, 40\%)$')
189 ax[0].set_xlim(0, t_max)
190 ax[0].set_ylabel("Taxa de desemprego")
191 ax[0].legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower center',
192             ncol=2, mode="expand", borderaxespad=0.)
193
194 # plot emprego:
195
196 ax[1].plot(estatica1[1], estatica1[3], color = "blue", lw = 1)
197 ax[1].plot(estatica2[1], estatica2[3], color = "green", lw = 1)
198 ax[1].plot(estatica3[1], estatica3[3], color = "red", lw = 1)
199 ax[1].plot(estatica4[1], estatica4[3], color = "black", lw = 1)
200 ax[1].set_xlim(0, t_max)
201 ax[1].set_xlabel("tempo")
202 ax[1].set_ylabel("Taxa de emprego")
203 plt.show()

```

```

204
205 #####
206 #          Trajetória de indivíduos com:          #
207 #          - Lambda = .391                        #
208 #          - Alpha = .027                          #
209 #####
210
211 exemplo_padrao2 = individual_path()
212
213 aux = max(exemplo_padrao2)
214 hist, valores = np.histogram(exemplo_padrao2, bins = np.arange(0, aux))
215 plt.plot(valores[:-1], hist, color="blue", linewidth=1)
216 plt.ylabel("Trabalhadores", fontsize = 10)
217 plt.xlabel("Tempo até encontrar emprego", fontsize = 10)
218 #plt.title('Histograma', fontsize = 13)
219 plt.show()
220
221 #####
222 #          Estática comparativa para desemprego          #
223 #####
224
225 exemplo_padrao3 = individual_path(Lambda = 0.2)
226 exemplo_padrao4 = individual_path(Lambda = 0.4)
227 exemplo_padrao5 = individual_path(Lambda = 0.65)
228 exemplo_padrao6 = individual_path(Lambda = 0.9)
229
230 aux = max(max(exemplo_padrao2), max(exemplo_padrao3),
231           max(exemplo_padrao4), max(exemplo_padrao5))
232 hist2, valores2 = np.histogram(exemplo_padrao3, bins = np.arange(0, aux))
233 hist3, valores3 = np.histogram(exemplo_padrao4, bins = np.arange(0, aux))
234 hist4, valores4 = np.histogram(exemplo_padrao5, bins = np.arange(0, aux))
235 hist5, valores5 = np.histogram(exemplo_padrao6, bins = np.arange(0, aux))
236 plt.plot(valores2[:-1], hist2, color="black", linewidth=1)
237 plt.plot(valores3[:-1], hist3, color="green", linewidth=1)
238 plt.plot(valores4[:-1], hist4, color="red", linewidth=1)
239 plt.plot(valores5[:-1], hist5, color="orange", linewidth=1)
240 plt.legend([r'$\lambda = 10\%$', r'$\lambda = 40\%$', r'$\lambda = 70\%$',
241            r'$\lambda = 90\%$'])
242 plt.ylabel("Trabalhadores", fontsize = 10)
243 plt.xlabel("Tempo até encontrar emprego", fontsize = 10)
244 #plt.title('Histogramas', fontsize = 13)
245 plt.show()

```

A.5 Escolha Intertemporal

```
1  # -*- coding: utf-8 -*-
2  """
3  @author: Matheus L. Carrijo
4  """
5
6  #####
7  #                               Importando pacotes                               #
8  #####
9
10 import matplotlib.pyplot as plt #importing graph package
11 plt.figure(figsize=(9, 6), dpi=1000) #set default figure size
12 import numpy as np
13
14 #####
15 #                               Funções                               #
16 #####
17
18 def u(c, a):
19     if a > 0 and a != 1:
20         return (c**(1-a) - 1) / (1 - a)
21     else:
22         if a == 0:
23             return np.log(c)
24
25 def U(c1, c2, a, b):
26     return u(c1, a) + u(c2, b)
27
28 def budget(c1, m1, m2, r):
29     if m1 > 0 and m2 > 0 and r > 0 and r < 1:
30         return (1+r)*m1 + m2 - (1+r)*c1
31
32 def inverse_budget(c2, m1, m2, r):
33     if m1 > 0 and m2 > 0 and r > 0 and r < 1:
34         return (m2-c2)/(1+r) + m1
35
36 def choice(a, b, r, m1, m2):
37     if (a > 0 and a != 1 and b > 0 and b != 1 and m1 > 0 and m2 > 0 and r > 0
38         and r < 1):
39         c1_estrela = (m1+m2/(1+r)) / (1+(1+r)**((1-a)/a))
40         c2_estrela = (m1 - c1_estrela)*(1+r) + m2
41         funcao_valor = U(c1_estrela, c2_estrela, a, b)
42
```

```

43     return c1_estrela, c2_estrela, funcao_valor
44
45     #####
46     #                                     Exemplo                                     #
47     #####
48
49     a, b, r, m1, m2 = 0.5, 0.5, 0.5, 50, 750
50     exemplo1 = choice(a, b, r, m1, m2)
51
52     #####
53     #                                     Informando ao usuário                                     #
54     #####
55
56     print(f"\nCom uma taxa de juros de {100*r}%, renda em t1 de {m1}", end = " ")
57     print(f"e renda {m2} em t2, a distribuição de consumo no tempo que", end = " ")
58     print("maximiza a utilidade do agente é dada por", end = " ")
59     print(f'(c1, c2) = ({exemplo1[0]}, {exemplo1[1]}), com nível de', end = " ")
60     print(f"utilidade dado por U = {exemplo1[2]}")
61
62     #####
63     #                                     Gráfico                                     #
64     #####
65
66     eixo_x, eixo_y = np.linspace(0, 1000, 1000), np.linspace(0, 1000, 1000)
67     budget1 = budget(eixo_x, m1, m2, r)
68     X, Y = np.meshgrid(eixo_x, eixo_y)
69
70     Z = plt.contour(X, Y, U(X, Y, a, b),
71                     levels = [exemplo1[2]-10, exemplo1[2], exemplo1[2]+10],
72                     linewidths = 2)
73     plt.plot(eixo_x, budget1, color = "black", lw = 2,
74             label = "restrição orçamentária")
75     plt.clabel(Z, inline = 1, fontsize = 10)
76     plt.ylim(0, (1+r)*m1 + m2)
77     plt.xlim(0, m1 + m2/(1+r))
78     plt.ylabel(r"Consumo em $t_2$", fontsize = 15)
79     plt.xlabel(r"Consumo em $t_1$", fontsize = 15)
80     plt.scatter([exemplo1[0]], [exemplo1[1]], s = 7*4**2)
81     plt.annotate(f"$ (c_1^*, c_2^*) = ({exemplo1[0]: .0f}, {exemplo1[1]: .0f})$",
82                 (exemplo1[0]+10, exemplo1[1]+10), fontsize = 13)
83     plt.scatter([m1], [m2], s = 7*4**2,
84                 label = f"dotação inicial $(m_1, m_2) = ({m1}, {m2})$")
85     plt.annotate("A", (m1-5, m2-55), fontsize = 13)
86     plt.legend()
87     plt.show()
88

```

```

89 #####
90 #                               Estática Comparativa                               #
91 #                               (alterando a taxa de juros)                               #
92 #####
93
94 # Outros exemplos
95 a2, b2, r2, m1_2, m2_2 = 0.5, 0.5, 0.1, 50, 750
96 a3, b3, r3, m1_3, m2_3 = 0.5, 0.5, 0.9, 50, 750
97 exemplo2 = choice(a2, b2, r2, m1_2, m2_2)
98 exemplo3 = choice(a3, b3, r3, m1_3, m2_3)
99
100 # Saída para o usuário
101 print(f"\nDiminuindo a taxa de juros para {100*r2}%, temos", end = " ")
102 print("que a nova distribuição de consumo no tempo que maximiza a", end = " ")
103 print("utilidade do agente é dada por", end = " ")
104 print(f'(c1*, c2*) = ({exemplo2[0]}, {exemplo2[1]}), com nível de', end = " ")
105 print(f"utilidade dado por U = {exemplo2[2]}. Se aumentarmos a", end = " ")
106 print(f"taxa de juros para {100*r3}%, a distribuição de consumo no", end = " ")
107 print("tempo é dada por (c1*, c2*) =", end = " ")
108 print(f"({exemplo3[0]}, {exemplo3[1]}), com nível de utilidade", end = " ")
109 print(f"U = {exemplo3[2]}. Podemos conferir estes resultados graficamente.")
110
111 # configuração exemplo1
112 budget1 = budget(eixo_x, m1, m2, r)
113
114 # configuração exemplo2
115 budget2 = budget(eixo_x, m1_2, m2_2, r2)
116
117 # configuração exemplo3
118 budget3 = budget(eixo_x, m1_3, m2_3, r3)
119
120
121 # plotando exemplo1 (mesmo que anterior)
122 Z1 = plt.contour(X, Y, U(X, Y, a, b), levels = [exemplo3[2], exemplo1[2],
123                                                exemplo2[2]], linewidths = 2)
124 plt.plot(eixo_x, budget1, color = "black", ls = "dashed", lw = 2,
125          label = f"restrição orçamentária com $r = {100*r}\%$")
126
127 # plotando exemplo2
128 plt.plot(eixo_x, budget2, color = "blue", lw = 2,
129          label = f"restrição orçamentária com $r = {100*r2}\%$")
130
131 # plotando exemplo3
132 plt.plot(eixo_x, budget3, color = "red", lw = 2,
133          label = f"restrição orçamentária com $r = {100*r3}\%$")
134

```

```

135 # Marcando pontos na tangência da reta orçamentária com as curvas de utilidade
136 plt.scatter([exemplo1[0]], [exemplo1[1]], s = 4*4**2)
137 plt.scatter([exemplo2[0]], [exemplo2[1]], s = 4*4**2)
138 plt.scatter([exemplo3[0]], [exemplo3[1]], s = 4*4**2)
139 plt.scatter([m1],[m2], s = 7*4**2,
140             label = f"dotação inicial $(m_1, m_2) = ({m1},{m2})$")
141
142 plt.annotate("A", (m1-5,m2-55), fontsize = 12)
143
144 # configuração geral plot
145 plt.legend(loc = 'lower left', bbox_to_anchor=(0.1, 0.1),
146           fancybox = True, shadow = True)
147 plt.ylim(0, (1+r)*m1 + m2)
148 plt.xlim(0, m1_3 + m2_3/(1+r3))
149 plt.ylabel(r"Consumo em $t_2$", fontsize = 12)
150 plt.xlabel(r"Consumo em $t_1$", fontsize = 12)
151 plt.show()

```

A.6 Escolha sob Incerteza

```

1 # -*- coding: utf-8 -*-
2 """
3 @author: Matheus L. Carrijo
4 """
5
6 #####
7 #                               Importando pacotes                               #
8 #####
9
10 import matplotlib.pyplot as plt #importing graph package
11 plt.figure(figsize=(9, 6), dpi=1000) #set default figure size
12 import numpy as np
13
14 #####
15 #                               Funções                                           #
16 #####
17
18 def utilidade(c_estado_b, c_estado_g, prob_estado_b, prob_estado_g):
19
20     return prob_estado_b*np.log(c_estado_b) + prob_estado_g*np.log(c_estado_g)
21
22 def restr_orcamentaria(c_estado_b, gamma, dot_estado_b, dot_estado_g):

```

```

23
24     b = dot_estado_g + dot_estado_b*gamma/(1-gamma)
25
26     return b - (gamma*c_estado_b)/(1-gamma)
27
28 def escolha(dot_estado_b, dot_estado_g, prob_estado_b, prob_estado_g, gamma):
29
30     b = dot_estado_g + dot_estado_b*gamma/(1-gamma)
31     c = prob_estado_g/prob_estado_b
32
33     c_b_estrela = b*prob_estado_b*(1-gamma)/gamma
34     c_g_estrela = c_b_estrela*c*(gamma/(1-gamma))
35     funcao_valor = utilidade(c_b_estrela, c_g_estrela, prob_estado_b,
36                             prob_estado_g)
37
38     return c_b_estrela, c_g_estrela, funcao_valor
39
40 def seguro_otimo(dot_estado_b, dot_estado_g, prob_estado_b, prob_estado_g,
41                 gamma):
42
43     escolha_otima = escolha(dot_estado_b, dot_estado_g, prob_estado_b,
44                             prob_estado_g, gamma)
45
46     return (dot_estado_g - escolha_otima[1]) / gamma
47
48     #####
49     #                               Exemplo                               #
50     #####
51
52 dot_estado_b1, dot_estado_g1 = 250, 350
53 gamma_1 = .4
54 prob_estado_b1, prob_estado_g1 = .4, .6
55
56 exemplo_1 = escolha(dot_estado_b1, dot_estado_g1, prob_estado_b1,
57                     prob_estado_g1, gamma_1)
58
59 seguro_estrela = seguro_otimo(dot_estado_b1, dot_estado_g1, prob_estado_b1,
60                               prob_estado_g1, gamma_1)
61
62     #####
63     #                               Informando ao usuário                               #
64     #####
65
66 print(f"\nCom dotação ({dot_estado_b1}, {dot_estado_g1});", end = " ")
67 print("distribuição de probabilidade dada por", end = " ")
68 print(f"({prob_estado_b1*100}%,{prob_estado_g1*100}%)", end = " ")

```

```

69 print(f"e premio {gamma_1*100}%, a distribuicao de consumo ótimo", end = " ")
70 print(f"será c_b = {exemplo_1[0]: .1f} e c_g = {exemplo_1[1]: .1f}.")
71 print(f"Ainda, a quantidade de seguro ótima é {seguro_estrela: .1f}.")
72
73 #####
74 #                                     Gráfico                                     #
75 #####
76
77 c_estado_b1 = np.linspace(0.01, 1000, 1000)
78 c_estado_g1 = np.linspace(0.01, 1000, 1000)
79 orcamento = restr_orcamentaria(c_estado_b1, gamma_1, dot_estado_b1,
80                                dot_estado_g1)
81 X, Y = np.meshgrid(c_estado_b1, c_estado_g1)
82
83 Z = plt.contour(X, Y, utilidade(X, Y, prob_estado_b1, prob_estado_g1),
84                levels = [exemplo_1[2]-10, exemplo_1[2], exemplo_1[2]+10],
85                linewidths = 2)
86 plt.plot(c_estado_b1, orcamento, color = "black", lw = 2,
87          label = "restrição orçamentária")
88 plt.clabel(Z, inline = 1, fontsize = 10)
89 plt.ylim(0, dot_estado_g1 + dot_estado_b1*gamma_1/(1-gamma_1))
90 plt.xlim(0, (1-gamma_1)*(dot_estado_g1 + dot_estado_b1*gamma_1/(1-gamma_1))
91          /gamma_1)
92 plt.ylabel(r"Consumo no estado 0 (bom)", fontsize = 15)
93 plt.xlabel(r"Consumo no estado 1 (ruim)", fontsize = 15)
94 plt.scatter([exemplo_1[0]], [exemplo_1[1]], s = 7*4**2)
95 plt.annotate(f"${c_1}^*, {c_2}^* = ({exemplo_1[0]: .0f}, {exemplo_1[1]: .0f})$",
96             (exemplo_1[0]+10, exemplo_1[1]+10), fontsize = 13)
97 plt.scatter([dot_estado_b1], [dot_estado_g1], s = 7*4**2,
98            label = f"dotação inicial = $({dot_estado_b1},{dot_estado_g1})$")
99 plt.annotate("A", (dot_estado_b1-15, dot_estado_g1-30), fontsize = 13)
100 plt.legend()
101 plt.show()

```

A.7 Stag Hunt Game (Rousseau)

```

1 # -*- coding: utf-8 -*-
2
3 """
4 @author: Matheus L. Carrijo
5 """
6 #####

```



```

7      #                                Importando pacotes                                #
8      #####
9
10     import numpy as np
11     import random as rd # Precisaremos desta pacote para o computador escolher
12                          # jogar o jogo Stag Hunt
13     import matplotlib.pyplot as plt
14     #####
15     #                                Definindo funções                                #
16     #####
17
18     def payoff(jogo):
19         """
20
21         Parameters
22         -----
23         jogo : Lista.
24             A primeira entrada é a jogada do primeiro jogador e a segunda entrada
25             é a jogada do segundo jogador. Para um jogo ser válido, ele deve estar
26             dentro da matriz de jogos possíveis.
27
28         Returns
29         -----
30         Lista
31             Retorna uma lista em que a primeira entrada representa o payoff do
32             primeiro jogador e a segunda entrada o payoff do segundo jogador.
33
34         """
35
36
37         if jogo in matriz:
38
39             if jogo == ['s', 's']:
40                 return [5, 5]
41             if jogo == ['s', 'h']:
42                 return [0, 3]
43             if jogo == ['h', 's']:
44                 return [3, 0]
45             if jogo == ['h', 'h']:
46                 return [3, 3]
47
48         else:
49
50             return -1
51
52     def rodada():

```

```

53     """
54
55
56     Returns
57     -----
58     jogo : Lista.
59     Retorna uma lista contendo o jogo feito pela escolha das estratégias
60     pelos dois jogadores.
61
62     """
63
64     jogada_A, jogada_B = rd.choice(estrategia), rd.choice(estrategia)
65     jogo = [jogada_A, jogada_B]
66
67     return jogo
68
69
70 def nash_equilibrium_analitico(jogo):
71     """
72
73
74     Parameters
75     -----
76     jogo : Lista.
77     Recebe uma lista do jogo definido pelas ações dos jogadores.
78
79     Returns
80     -----
81     bool
82     Calcula o equilíbrio de forma analítica do jogo, isto é, sabemos de
83     antemão que o equilíbrio consiste nos jogos em que ambos os jogadores
84     escolhem ao mesmo tempo caçar lebre ou veado.
85
86     """
87
88     if jogo in matriz:
89
90         payoffs = payoff(jogo)
91
92         if payoffs == [5, 5] or payoffs == [3, 3]:
93
94             equilibrio = True
95
96         else:
97
98             equilibrio = False

```

```

99
100     return equilibrio
101
102 else:
103
104     return -1
105
106
107 def nash_equilibrium(jogo):
108     """
109
110
111     Parameters
112     -----
113     jogo : Lista
114         Recebe uma lista do jogo determinado pelas ações (escolha das
115         estratégias) dos jogadores.
116
117     Returns
118     -----
119     Bool
120         Assim como na função anterior, calcula o equilíbrio de Nash. Aqui,
121         porém, fazemos um algoritmo em que o cálculo do equilíbrio baseia-se
122         na análise da mudança de estratégia unilateral de cada jogador: se
123         houver uma melhora no payoff dos jogadores, então não há equilíbrio;
124         caso contrário, há.
125
126     """
127
128     if jogo in matriz:
129
130         payoffs = payoff(jogo)
131         equilibrio = True
132
133         if jogo[0] == 'h':
134
135             jogo[0] = 's'
136             payoff_aux = payoff(jogo)
137
138             if payoff_aux[0] > payoffs[0]:
139
140                 equilibrio = False
141
142             jogo[0] = 'h'
143
144         if jogo[0] == 's':

```

```

145
146     jogo[0] = 'h'
147     payoff_aux = payoff(jogo)
148
149     if payoff_aux[0] > payoffs[0]:
150
151         equilibrio = False
152
153     jogo[0] = 's'
154
155     if jogo[1] == 'h':
156
157         jogo[1] = 's'
158         payoff_aux = payoff(jogo)
159
160         if payoff_aux[1] > payoffs[1]:
161
162             equilibrio = False
163
164         jogo[1] = 'h'
165
166     if jogo[1] == 's':
167
168         jogo[1] = 'h'
169         payoff_aux = payoff(jogo)
170
171         if payoff_aux[1] > payoffs[1]:
172
173             equilibrio = False
174
175         jogo[1] = 's'
176
177     return equilibrio
178
179 else:
180
181     return -1
182
183 def tabela_latex():
184
185     global tabela
186
187     matriz0 = np.array([[5,5], [0,3], [3,0], [3,3]])
188     matriz = matriz0.flatten() # Transforma a matriz de payoffs em uma lista
189
190     # Cria a tabela usando a sintaxe do latex:

```

191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236

```
# (o processo não aceita uma string com multiplas linhas; o que eu faço
# aqui é quebrar a string simples usando \ no final da linha, apenas pra
# facilitar a nossa visualização)

tabela = f"\\begin{{table}}[H]\\centering\\begin{{tabular}}{{c c | c | c |}} \\
& \\multicolumn{{1}}{{c}}{{}} & \\multicolumn{{2}}{{c}}{{Jogador 2}}\\\\ \\
& \\multicolumn{{1}}{{c}}{{}} & \\multicolumn{{1}}{{c}}{{\\$\\$}} & \\multicolumn{{1}}{{c}}{{\\$\\$}} \\
\\cline{{3-4}} \\multirow{{2}}*{{Jogador 1}} & \\$\\$ & ({{matriz[0]}},{{matriz[1]}}) & ({{matriz[2]}},{{matriz[3]}}) \\
\\cline{{3-4}} & \\$\\$ & ({{matriz[4]}},{{matriz[5]}}) & ({{matriz[6]}},{{matriz[7]}}) \\\\ \\
\\cline{{3-4}} \\
\\end{{tabular}}\\end{{table}}"

# Define o estilo cosmético da figura.
plt.style.use('_classic_test_patch')

# Plot da tabela:
# Habilita o uso do código em tex
plt.rc("text", usetex = True)

# Habilita alguns pacotes do latex
plt.rc("text.latex", preamble=r'\\usepackage{multirow,array,float}')

# Inicializa a figura p/ plot
plt.figure(figsize=(1,1), dpi = 500)

# Cria eixos p/ plot
ax = plt.subplot2grid((1,1),(0,0))

# Desativa o frame criado pelo passo anterior
ax.axis("off")

# Passa o conteúdo da variável ``tex_table`` como texto ao gráfico
ax.text(0.5, 0.5, tabela, ha = "center", va = "center")
# Mostra a figura

#plt.savefig('Inserir nome.pdf')
plt.show()

# Desabilita o uso do código em tex
plt.rc("text", usetex = False)

#####
#                                Jogando                                #
#####

estrategia = ['s', 'h'] # as possíveis ações dos jogadores são caçar veado
```

```

237         # (jogar "s") ou caçar lebre (jogar "h").
238 matriz = [['s', 's'], ['s', 'h'],
239           ['h', 's'], ['h', 'h']] # jogos possíveis determinando pela matriz.
240
241 equilibrio = False # Para entrar no loop abaixo
242 i = 1 # contagem do número de jogos
243
244 while not equilibrio: #Jogos serão feitos até que se alcance um equilibrio
245
246     jogo = rodada() #Um jogo é uma lista com as jogadas dos dois jogadores
247     equilibrio = nash_equilibrium(jogo) #Confere se o jogo constitui equilibrio
248     payoffs = payoff(jogo) #Calcula o Payoff do jogo
249
250     print("\nJogaremos até encontrarmos um jogo com Equilíbrio de Nash.")
251
252     if jogo == ['s', 'h']:
253
254         print(f"\nJogo {i}: O jogador A falhou em tentar caçar", end = " ")
255         print("o veado sozinho; o jogador B conseguiu caçar", end = " ")
256         print("a lebre sozinho. Como o jogador A poderia", end = " ")
257         print("melhorar sua situação mudando sua estratégia e", end = " ")
258         print("caçar uma lebre sozinho, este jogo não", end = " ")
259         print("constitui um Equilíbrio de Nash.")
260
261     if jogo == ['h', 's']:
262
263         print(f"\nJogo {i}: O jogador B falhou em tentar caçar", end = " ")
264         print("o veado sozinho; o jogador A conseguiu caçar", end = " ")
265         print("a lebre sozinho. Como o jogador B poderia", end = " ")
266         print("melhorar sua situação mudando sua estratégia e", end = " ")
267         print("caçar uma lebre sozinho, este jogo não", end = " ")
268         print("constitui um Equilíbrio de Nash.")
269
270     if jogo == ['h', 'h']:
271
272         print(f"\nJogo {i}: Ambos os jogadores escolheram", end = " ")
273         print("caçar lebres e conseguiram. Como nenhum jogador", end = " ")
274         print("tem a ganhar mudando sua estratégia unilateralme", end = "")
275         print("nte para caçar um veado, então este jogo constit", end = "")
276         print("ui um Equilíbrio de Nash. Note, no entanto, que ambo", end = "")
277         print("s melhorariam sua situação se combinassem de caçar um veado.")
278
279     if jogo == ['s', 's']:
280
281         print(f"\nJogo {i}: Ambos os jogadores escolheram", end = " ")
282         print("caçar veados e conseguiram. Como nenhum jogador", end = " ")

```

```

283     print("tem a ganhar mudando sua estratégia unilateralme", end = "")
284     print("nte para caçar uma lebre, então este jogo constit", end = "")
285     print("ui um Equilíbrio de Nash. Note que este é o melhor", end = " ")
286     print("resultado que os jogadores poderiam ter.")
287
288     i += 1
289
290 i -= 1
291
292     #####
293     #                               Invocando a tabela                               #
294     #####
295
296 tabela_latex()

```

A.8 Oligopólio

```

1  # -*- coding: utf-8 -*-
2
3  """
4  @author: Matheus L. Carrijo
5
6  Objetivo do programa: implementar computacionalmente o modelo que descreve a
7  competição de oligopólio com apenas duas empresas idênticas (duopólio) e com
8  apenas um período, chamado Modelo de Cournot. Cada empresa toma a decisão
9  de quanto produzir baseada na expectativa de produção da outra empresa, de
10 maneira a maximizar o lucro. O equilíbrio de previsões é a situação em que cada
11 empresa vê sua crença sobre a outra confirmada. Por simplicidade, também
12 considero custos nulos. Depois, o modelo é ampliado para o caso com n empresas
13 """
14
15     #####
16     #                               Importando pacotes                               #
17     #####
18
19 import matplotlib.pyplot as plt #importing graph package
20 plt.figure(figsize=(10, 6), dpi=1000) #set default figure size
21 import numpy as np #importing numpy
22
23     #####
24     #                               Definindo funções                               #
25     #####

```

```

26
27 def demanda_inversa(Y, a = 1000, b = 1): #preço em função da quantidade
28     """
29     Definindo a demanda inversa, que depende da produção total da economia e
30     dos parâmetros 'a' e 'b'. O parâmetro Y é uma lista em que a entrada i
31     representa a produção da empresa i.
32     """
33
34     assert a > 0 and b > 0, "Os parâmetros da demanda devem ser positivos"
35
36     global c, d
37     c, d = a, b
38
39     return a - b*(sum(Y))
40
41 def receita(y, Y):
42     """
43     Receita = preço*quantidade. No caso do oligopólio, o preço é dado pela
44     demanda inversa e a quantidade determinada pela firma.
45     """
46
47     return y*demanda_inversa(Y)
48
49 def curva_reacao():
50     """
51     Serve para o caso em que numero_firmas = 2. A função resolve o problema de
52     maximização da firma arbitrária (1 ou 2) em função da produção que ela
53     espera da outra firma, ou seja, a função gera a curva de melhor resposta.
54     O equilíbrio de Cournot será a intersecção entre ambas as curvas.
55     """
56
57     demanda_inversa([0,1]) # Chamo a função apenas para definir o parâmetro 'c'
58
59     # Resolvendo o problema para uma empresa arbitrária (empresa 1 ou 2):
60
61     receita_valores = []
62     y_argmax = []
63     y, y_e = np.linspace(0, c, c+1), np.linspace(0, c, c+1)
64
65     for j in y_e:
66
67         for i in y:
68
69             receita_valores.append(receita(i, [i,j]))
70
71     indice_max = np.argmax(receita_valores) # Guarda o índice da maior

```



```

72                                     # receita
73     y_argmax.append(y[indice_max]) # Adiciona a uma lista o nível de
74                                     # produção y que maximiza o lucro
75     receita_valores = [] # "zera" a lista de receita para o algoritmo
76                           # resolver novamente o problema de maximização de
77                           # lucro para outro nível de produção esperado
78
79     return y_e, y_argmax
80
81 def equilibrio_cournot(numero_firmas = 2):
82     """
83     Esta função serve para o caso geral em que numero_firmas = n. A solução é
84     implementada algebricamente. As firmas são idênticas e o custo é nulo. O
85     cálculo é realizado para o caso especial de demanda inversa linear.
86     """
87
88     coeficientes = np.array([[2 if j==i else 1 for i in range(numero_firmas)]
89                               for j in range(numero_firmas)])
90     inversa_coeficientes = np.linalg.inv(coeficientes)
91
92     constantes = np.array([c/d for i in range(numero_firmas)])
93
94     vetor_producao = np.dot(inversa_coeficientes, constantes)
95
96     return vetor_producao
97
98     #####
99     #                               Programa principal                               #
100    #                               (chamando as funções)                               #
101    #####
102
103    curva_de_reacao = curva_reacao()
104
105    equilibrio_cournot1 = equilibrio_cournot() # Para n=2 firmas
106
107    quantidade_equilibrio = sum(equilibrio_cournot1)
108
109    preco_equilibrio = demanda_inversa(equilibrio_cournot1)
110
111
112    ##### Analisando o efeito do aumento de firmas #####
113
114    preco2 = []
115    quantidade2 = []
116    n_firmas = 100
117

```

```

118 for i in range(1, n_firmas + 1): # para cada numero de firmas, calcular as
119                                     # a quantidade e preço de equilíbrio da
120                                     # economia.
121
122     equilibrio_cournot2 = equilibrio_cournot(i)
123     quantidade2.append(sum(equilibrio_cournot2))
124     preco2.append(demanda_inversa(equilibrio_cournot2))
125
126     #####
127     #                                     informando ao usuário                                     #
128     #####
129
130 print("O preço e a quantidade de equilíbrio são dados respectivamente por:")
131 print(f"{quantidade_equilibrio: .2f} e {preco_equilibrio: .2f}")
132
133     #####
134     #                                     Gráficos                                     #
135     #####
136
137 # Gráfico para o exemplo de 2 firmas (curvas de reacao):
138
139 plt.plot(curva_de_reacao[1], curva_de_reacao[0], color="blue", linewidth=2.0,
140          ls = "dotted", label = "Curva de reação da empresa 1")
141 plt.plot(curva_de_reacao[0], curva_de_reacao[1], color="green", linewidth=2.0,
142          ls = "dashed", label = "Curva de reação da empresa 2")
143 plt.xlabel(r"Produção da empresa 1 ($y_1$)", fontsize = 15)
144 plt.ylabel(r"Produção da empresa 2 ($y_2$)", fontsize = 15)
145 plt.scatter(equilibrio_cournot1[0], equilibrio_cournot1[1], s = 7*4**2,
146             label = "Equilíbrio de Cournot", color = "black")
147 plt.annotate(f"${y_1^*}, {y_2^*} = ({c/3*d: .2f}, {c/3*d: .2f})$",
148             ((c/3*d)+15, (c/3*d)+30), fontsize = 13)
149 plt.legend()
150 plt.show()
151
152 # Gráfico para a análise da entrada de firmas:
153
154 plt.plot(np.linspace(1, n_firmas, n_firmas), quantidade2, color="k",
155          linewidth=2.0, ls = "dashed", label = "Quantidade")
156 plt.plot(np.linspace(1, n_firmas+1, n_firmas), preco2, color="r",
157          linewidth=2.0, label = "Preço")
158 plt.xlabel(r"Número de firmas", fontsize = 15)
159 plt.legend(fontsize = 13)
160 plt.show()

```

A.9 Caixa de Edgeworth

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Wed Dec 15 22:59:27 2021
4
5  @author: Matheus L. Carrijo
6  """
7
8  #####
9  #                               Importando pacotes                               #
10 #####
11
12 import numpy as np #importing numpy
13 import matplotlib.pyplot as plt #importing graph package
14 #plt.figure(figsize=(9, 6), dpi=1000) #set default figure size
15
16 #####
17 #                               Definindo funções                               #
18 #####
19
20 def utilidade(x, y, alpha = 0.5):
21     """
22     Função de utilidade Cobb-Douglas dos agentes (ambos terão a mesma).
23     """
24
25     return (x**alpha)*(y**(1-alpha))
26
27 def demanda(p, dot, alpha = 0.5):
28     """
29     Resolve o problema do consumidor de forma analítica, estabelecendo a
30     demanda pelos bens x e y. Como as funções de utilidade são as mesmas para
31     ambos os agentes, a demanda será igual para valores arbitrários da dotação
32     inicial. O parâmetro 'dot' é uma lista em que a primeira entrada
33     corresponde à dotação inicial do bem x; e a segunda para o bem y. O
34     parâmetro preço é uma lista em que a primeira entrada corresponde ao preço
35     do bem x e a segunda o preço do bem y.
36     """
37
38     x = alpha*(p[0]*dot[0] + p[1]*dot[1]) / p[0]
39     y = (1 - alpha)*(p[0]*dot[0] + p[1]*dot[1]) / p[1]
40
41     return x, y
42
```

```

43 def demanda_liq(p, dot):
44     """
45     Calcula o excesso de demanda para os bens x e y, isto é, a diferença entre
46     a demanda total e a dotação inicial dos agentes. O parâmetro p é uma lista
47     com os preços de x e y, respect., enquanto que dot é uma lista com as
48     dotações iniciais dos bens x e y, respect..
49     """
50
51     x, y = demanda(p, dot)
52     demanda_liq_x = x - dot[0]
53     demanda_liq_y = y - dot[1]
54
55     return demanda_liq_x, demanda_liq_y
56
57 def demanda_liq_agreg(p, dot_A, dot_B):
58     """
59     Calcula a demanda líquida agregada para os bens x e y. A demanda líquida
60     agregada nada mais é que a soma das demandas líquidas individuais.
61     """
62
63     demanda_liq_agreg_x = demanda_liq(p, dot_A)[0] + demanda_liq(p, dot_B)[0]
64     demanda_liq_agreg_y = demanda_liq(p, dot_A)[1] + demanda_liq(p, dot_B)[1]
65
66     return demanda_liq_agreg_x, demanda_liq_agreg_y
67
68 def equilibrio(dot_A, dot_B):
69     """
70     Calcula o equilíbrio geral para um mercado competitivo de dois bens e dois
71     indivíduos. Seguindo o que diz a Lei de Walras, o preço do bem x pode ser
72     fixado em 1 (bem numérico) enquanto que achamos o preço do bem 2 que zera
73     o excesso de demanda agregado do bem 2.
74     """
75
76     # Procurando o zero da demanda agregada excedente para o bem y
77     # pelo método da bissecção:
78
79     global px
80
81     px = 1
82     tol = 1e-10
83     p_alto = 1000
84     p_baixo = 0
85
86     # Analisando a fronteira:
87     tentativa1 = demanda_liq_agreg([px, p_alto], dot_A, dot_B)[1]
88     if abs(tentativa1) < tol:

```

```

89         return p_alto
90
91     # atualiza preco pela média
92     preco = p_alto / 2
93     excesso_demanda_agreg = demanda_liq_agreg([px, preco], dot_A, dot_B)[1]
94
95     while abs(excesso_demanda_agreg) > tol:
96
97         if excesso_demanda_agreg > 0:
98
99             p_baixo = preco
100
101         else:
102
103             p_alto = preco
104
105             preco = (p_baixo + p_alto) / 2
106             excesso_demanda_agreg = demanda_liq_agreg([px, preco], dot_A, dot_B)[1]
107
108     return preco
109
110 def curva_contrato():
111     """
112     Calcula um subconjunto da curva de contrato variando as dotações.
113     """
114
115     precos_equilibrio = [] # Lista para adicionar os preços de equilíbrio
116                        # conforme variamos as dotações.
117
118     y_aux = np.linspace(0, total_dot_y, 5)
119     # Não estou "varrendo" toda a caixa de Edgeworth para calcular as
120     # demandas de equilíbrio. Passo apenas inteiramente por um eixo (eixo
121     # x, por ex) e vario alguns pontos do eixo y. Por este motivo, a curva
122     # resultante é um subconjunto da curva de contrato.
123
124     for j in y_aux:
125
126         for i in x:
127
128             precos_equilibrio.append(equilibrio([i, j], [total_dot_x - i,
129                                                         total_dot_y - j]))
130
131     x_estrela = []
132     y_estrela = []
133
134     for k in precos_equilibrio:

```

```

135
136     for m in y_aux:
137
138         for l in x:
139
140             x_estrela.append(demanda([1, k], [1, m])[0])
141             y_estrela.append(demanda([1, k], [1, m])[1])
142
143     return [x_estrela, y_estrela]
144
145     #####
146     #                                     Programa principal                                #
147     #####
148
149     #dot_A = [100, 25] # dotação indivíduo A
150     #dot_B = [50, 150] # dotação indivíduo B
151
152     #dot_A = [23, 33] # dotação indivíduo A
153     #dot_B = [14, 32] # dotação indivíduo B
154
155     dot_A = [230, 100] # dotação indivíduo A
156     dot_B = [400, 90] # dotação indivíduo B
157
158     preco_equilibrio_y = equilibrio(dot_A, dot_B)
159     p_equilibrio = [px, preco_equilibrio_y]
160
161     #Calcula as demandas de equilíbrio para os indivíduos A e B, respect.:
162     demanda_eq_A = demanda(p_equilibrio, dot_A)
163     demanda_eq_B = demanda(p_equilibrio, dot_B)
164
165     # Utilidade dos agentes em equilíbrio:
166     u_valor_A = utilidade(demanda_eq_A[0], demanda_eq_A[1])
167     u_valor_B = utilidade(demanda_eq_B[0], demanda_eq_B[1])
168
169
170     total_dot_x = dot_A[0] + dot_B[0] # pode-se considerar o tamanho do eixo
171                                     # horizontal da caixa de Edgeworth
172     total_dot_y = dot_A[1] + dot_B[1] # pode-se considerar o tamanho do eixo
173                                     # vertical da caixa de Edgeworth
174
175     x = np.linspace(0.0001, total_dot_x, 100) # grid da quantidade do bem x
176     y = np.linspace(0.0001, total_dot_y, 100) # grid da quantidade do bem y
177
178     contrato = curva_contrato()
179
180     #####

```

```

181         # Informando ao usuário #
182         #####
183
184     print(f"\nCom dotações dos agentes A e B dadas por {dot_A} e", end = " ")
185     print(f"{dot_B} as alocações de equilíbrio para os indivíduos $A$", end = " ")
186     print(f"e e $$ são, respect., ({demanda_eq_A[0]:.2f},", end = " ")
187     print(f"{demanda_eq_A[1]:.2f}) e ({total_dot_x-demanda_eq_A[0]:.2f}", end = "")
188     print(f", {total_dot_y-demanda_eq_A[1]:.2f}).\n")
189     print("A curva de contrato na figura mostra as diferentes", end = " ")
190     print("alocações eficientes para variações nas dotações iniciais.\n")
191
192     #####
193     # Gráfico #
194     #####
195
196     #plt.style.use('classic')
197     #plt.style.use('default')
198     plt.rcParams["figure.figsize"] = (10, 6)
199     plt.rcParams["lines.linewidth"] = (2.5)
200
201     fig, ax = plt.subplots(tight_layout = True)
202
203     # Cria o produto cartesiano das quantidades  $x_1$  e  $x_2$ .
204     xc_1, xc_2 = np.meshgrid(x, y)
205
206     # Coloca label nos eixos originais
207     ax.set_xlabel('$x_A$', fontsize = 20)
208     ax.set_ylabel('$y_A$', fontsize = 20)
209
210     ax.axis([0, 1.02*total_dot_x, 0, 1.05*total_dot_y])
211
212     # Calcula a utilidade do consumidor A
213     uA = utilidade(xc_1, xc_2) # Calcula a matriz de utilidades do
214                                # consumidor A.
215
216     # Calcula a utilidade do consumidor B. Como queremos que o gráfico tenha origem
217     # em  $x = x_0$  e  $y = y_0$ , fazemos o ajuste nos argumentos da função.
218     uB = utilidade(total_dot_x - xc_1, total_dot_y - xc_2) # Calcula a matriz de
219                                                            # utilidades do
220                                                            # consumidor B.
221
222     # Criamos um novo eixo y (apenas para repetir os ticks e labels do eixo y
223     # original)
224     ax2 = ax.twinx()
225
226     # Copiamos os limites do eixo y original

```

```

227 orig_ylim = ax.get_ylim()
228
229 # Replicamos os limites do eixo y original no novo eixo
230 ax2.set_ylim(orig_ylim)
231
232 # Definimos um label para o novo eixo y
233 ax2.set_ylabel('$y_B$', fontsize = 20)
234
235 # Invertemos o sentido do texto do eixo y original
236 ax2.invert_yaxis()
237
238 # Criamos um novo eixo x (apenas para repetir os ticks e labels do eixo x
239 # original)
240 ax3 = ax.twinx()
241
242 # Copiamos os limites do eixo x original
243 orig_xlim = ax.get_xlim()
244
245 # Replicamos os limites do eixo x original no novo eixo
246 ax3.set_xlim(orig_xlim)
247
248 # Definimos um label para o novo eixo x
249 ax3.set_xlabel('$x_B$', fontsize = 20)
250
251 # Invertemos o sentido do texto do eixo x original
252 ax3.invert_xaxis()
253
254 # Curvas de indiferença no nível indicado
255 #CS1 = ax.contour(xc_1, xc_2, uA, levels = [u_valor_A], linestyles = 'dashdot',
256 # colors = 'black', alpha = 1, extend = 'both')
257 #CS2 = ax.contour(xc_1, xc_2, uB, levels = [u_valor_B], linestyles = 'dotted',
258 # colors = 'forestgreen', alpha = 1, extend = 'both')
259
260 CS1 = ax.contour(xc_1, xc_2, uA, levels = [u_valor_A - 70, u_valor_A,
261 u_valor_A + 50, u_valor_A + 100],
262 linestyles = 'dashdot', colors = 'k', alpha = 1,
263 extend = 'both')
264 CS2 = ax.contour(xc_1, xc_2, uB, levels = [u_valor_B - 100, u_valor_B - 50,
265 u_valor_B, u_valor_B + 70],
266 linestyles = 'dotted', colors = 'g', alpha = 1,
267 extend = 'both')
268
269 # Curva de contrato:
270 ax.plot(contrato[0], contrato[1], 'blue', linewidth = 2, alpha = 0.6,
271 label = 'Curva de contrato')
272

```



```

273 #Alocação de equilíbrio:
274 ax.scatter(demanda_eq_A[0], demanda_eq_A[1], s = 10*5**2,
275           label = "Alocação de equilíbrio", alpha = 1, c = "r")
276 #ax.plot(demanda_eq_A[0], demanda_eq_A[1], 'r', marker = ".", markersize = 25,
277 #        label = "Alocação de equilíbrio")
278
279 # Dotação inicial:
280 ax.scatter(dot_A[0], dot_A[1], s = 10*5**2, label = "Dotação", alpha = 1,
281           c = "black")
282 #ax.plot(dot_A[0], dot_A[1], 'black', marker = ".", markersize = 25,
283 #        label = "Dotação")
284 ax.legend(loc = "upper right", fancybox = True, shadow = True, fontsize = 13)
285 plt.show()

```

A.10 Matching

```

1  # -*- coding: utf-8 -*-
2  """
3  @author: Matheus Lopes Carrijo
4  """
5
6  #####
7  #                                Importando pacotes                                #
8  #####
9
10 import numpy as np
11 import matplotlib.pyplot as plt
12
13 #####
14 #                                Definindo funções                                #
15 #####
16
17 def matching_com_solteiros(preferencia_H, preferencia_M):
18
19     # preferencia_H são listas de listas, em que a i-ésima lista nos dá a
20     # preferência (ranking de mulheres) do homem i (o mesmo raciocínio vale
21     # para preferencia_M).
22
23     # Garante que o número de homens deve ser igual ao de mulheres:
24     assert len(preferencia_H) == len(preferencia_M)
25
26     n = len(preferencia_H) # n = numero de mulheres e homens

```

```

27
28 casais = [] # registra os casais formados. É uma lista de listas.
29 h_solteiros = [h for h in range(n)] # Registra os homens solteiros.
30 m_solteiras = [m for m in range(n)] # Registra as mulheres solteiras.
31 # Lista para a situacao dos casais (mulher = coluna j; homem = linha i)
32 # 0 = mulher j não recebeu proposta
33 # 1 = mulher j recebeu proposta do homem j
34 # 2 = mulher j aceitou a proposta do homem j
35 estado = np.zeros((n, n), dtype = int) # Ajuda na formatação do gráfico
36
37 # Cria a estrutura do gráfico que usaremos depois
38 fig, ax = plt.subplots(nrows = 2, ncols = 5, sharex = False,
39                        sharey = True, tight_layout = True,
40                        figsize = (13, 7), dpi = 800)
41
42 ##### Enquanto houver homem solteiro, haverá novas propostas #####
43
44 while h_solteiros:
45
46
47     for m in range(n): # quando m mudar, significa nova rodada (todos
48                         # farão proposta pra mulher mais preferida, depois
49                         # para a segunda mais preferida, e assim sucess.)
50
51         propostas = [] # Lista de listas. O elemento [h, m] significa que
52                         # o homem h fez proposta para a mulher m.
53
54         solteiros_indice = []
55         solteiras_indice = []
56
57         ##### Mulheres que querem ficar solteiras: #####
58
59         for k in m_solteiras:
60
61             if preferencia_M[k][m] == -1:
62
63                 casais.append([-1, k])
64                 solteiras_indice.append(k)
65
66         for j in solteiras_indice:
67
68             m_solteiras.remove(j)
69
70         ##### rodada de propostas #####
71
72         for h in h_solteiros: # faz proposta ou escolhe ficar solteiro

```

```

73
74     if (preferencia_H[h][m] != -1): # faz proposta
75
76         propostas.append([h, preferencia_H[h][m]])
77
78         if estado[h][preferencia_H[h][m]] != 2:
79             estado[h][preferencia_H[h][m]] = 1
80
81     else: # fica solteiro
82
83         casais.append([h, -1])
84         solteiros_indice.append(h)
85
86 for j in solteiros_indice:
87
88     h_solteiros.remove(j)
89
90 ##### Análise de propostas #####
91
92 for h, m2 in propostas:
93
94     # se a mulher está casada:
95     if m2 in [casais[k][1] for k in range(len(casais))]:
96
97         # ----- achando o índice do parceiro da mulher m2 ----- #
98
99         for i in range(len(casais)):
100
101             if casais[i][1] == m2:
102
103                 indice_parceiro = i
104
105                 # ----- #
106
107                 # se o homem fazendo a proposta é preferido ao atual
108                 # parceiro da mulher, então ela troca de parceiro:
109                 if (preferencia_M[m2].index(casais[indice_parceiro][0]) >
110                     preferencia_M[m2].index(h)):
111
112                     estado[h][m2] = 2
113                     estado[casais[indice_parceiro][0]][m2] = 0
114
115                     h_solteiros.append(casais[indice_parceiro][0])
116
117                     casais[indice_parceiro][0] = h
118                     h_solteiros.remove(h)

```

```

119         h_solteiros.sort()
120
121         # Se a mulher está solteira, ela aceita a proposta:
122         else:
123
124             # Mas apenas se ela não prefere estar solteira a estar com
125             # h
126             if (preferencia_M[m2].index(-1) >
127                 preferencia_M[m2].index(h)):
128
129                 casais.append([h, m2])
130                 h_solteiros.remove(h)
131                 h_solteiros.sort()
132                 m_solteiras.remove(m2)
133                 m_solteiras.sort()
134
135                 estado[h][m2] = 2
136
137         # Removendo -1 da lista de solteiros caso tenha algum
138         if -1 in h_solteiros:
139
140             h_solteiros.remove(-1)
141
142         # Removendo [-1, -1] da lista casais pois não significa nada
143         if [-1, -1] in casais:
144
145             casais.remove([-1, -1])
146
147
148         #####
149         # Gráfico: mostra casais formados a cada rodada #
150         #####
151
152         # Código RGB para as cores
153         cores = {0: np.array([255, 255, 255]), # branco
154                  1: np.array([100, 100, 255]), # azul (não usaremos)
155                  2: np.array([255, 100, 100])} # vermelho
156
157         # Matriz que mapeia o estado nas cores correspondentes
158         mapa = []
159
160         for homem in estado:
161
162             linha = []
163
164             for situacao in homem:

```

```

165         cor = cores[situacao]
166         linha.append(cor)
167
168
169     mapa.append(linha)
170
171     #Labels para os homens e mulheres
172     label_homens = ["H0", "H1", "H2", "H3", "H4", "H5", "H6", "H7",
173                    "H8", "H9"]
174     label_mulheres = ["M0", "M1", "M2", "M3", "M4", "M5", "M6", "M7",
175                      "M8", "M9"]
176
177     if m in range(5):
178
179         ax[0][m].matshow(mapa)
180         # Cria ticks para traçar as linhas do grid
181         ax[0][m].set_xticks(np.arange(-.5, len(label_homens)),
182                             minor = True)
183         ax[0][m].set_yticks(np.arange(-.5, len(label_mulheres)),
184                             minor = True)
185         ax[0][m].grid(True, which = 'minor', color = 'k', ls = ':',
186                       linewidth = 1)
187         # Define a posição dos ticks principais e altera labels
188         ax[0][m].set_xticks(np.arange(len(label_homens)))
189         ax[0][m].set_yticks(np.arange(len(label_mulheres)))
190         ax[0][m].set_xticklabels(label_mulheres)
191         ax[0][m].set_yticklabels(label_homens)
192         ax[0][m].title.set_text(f"Rodada {m+1}")
193
194     if m == 5:
195
196         ax[1][0].matshow(mapa)
197         # Cria ticks para traçar as linhas do grid
198         ax[1][0].set_xticks(np.arange(-.5, len(label_homens)),
199                             minor = True)
200         ax[1][0].set_yticks(np.arange(-.5, len(label_mulheres)),
201                             minor = True)
202         ax[1][0].grid(True, which = 'minor', color = 'k', ls = ':',
203                       linewidth = 1)
204         # Define a posição dos ticks principais e altera labels
205         ax[1][0].set_xticks(np.arange(len(label_homens)))
206         ax[1][0].set_yticks(np.arange(len(label_mulheres)))
207         ax[1][0].set_xticklabels(label_mulheres)
208         ax[1][0].set_yticklabels(label_homens)
209         ax[1][0].title.set_text(f"Rodada {m+1}")
210

```

```

211     if m == 6:
212
213         ax[1][1].matshow(mapa)
214         # Cria ticks para traçar as linhas do grid
215         ax[1][1].set_xticks(np.arange(-.5, len(label_homens)),
216                             minor = True)
217         ax[1][1].set_yticks(np.arange(-.5, len(label_mulheres)),
218                             minor = True)
219         ax[1][1].grid(True, which = 'minor', color = 'k', ls = ':',
220                       linewidth = 1)
221         # Define a posição dos ticks principais e altera labels
222         ax[1][1].set_xticks(np.arange(len(label_homens)))
223         ax[1][1].set_yticks(np.arange(len(label_mulheres)))
224         ax[1][1].set_xticklabels(label_mulheres)
225         ax[1][1].set_yticklabels(label_homens)
226         ax[1][1].title.set_text(f"Rodada {m+1}")
227
228     if m == 7:
229
230         ax[1][2].matshow(mapa)
231         # Cria ticks para traçar as linhas do grid
232         ax[1][2].set_xticks(np.arange(-.5, len(label_homens)),
233                             minor = True)
234         ax[1][2].set_yticks(np.arange(-.5, len(label_mulheres)),
235                             minor = True)
236         ax[1][2].grid(True, which = 'minor', color = 'k', ls = ':',
237                       linewidth = 1)
238         # Define a posição dos ticks principais e altera labels
239         ax[1][2].set_xticks(np.arange(len(label_homens)))
240         ax[1][2].set_yticks(np.arange(len(label_mulheres)))
241         ax[1][2].set_xticklabels(label_mulheres)
242         ax[1][2].set_yticklabels(label_homens)
243         ax[1][2].title.set_text(f"Rodada {m+1}")
244
245     if m == 8:
246
247         ax[1][3].matshow(mapa)
248         # Cria ticks para traçar as linhas do grid
249         ax[1][3].set_xticks(np.arange(-.5, len(label_homens)),
250                             minor = True)
251         ax[1][3].set_yticks(np.arange(-.5, len(label_mulheres)),
252                             minor = True)
253         ax[1][3].grid(True, which = 'minor', color = 'k', ls = ':',
254                       linewidth = 1)
255         # Define a posição dos ticks principais e altera labels
256         ax[1][3].set_xticks(np.arange(len(label_homens)))

```

```

257         ax[1][3].set_yticks(np.arange(len(label_mulheres)))
258         ax[1][3].set_xticklabels(label_mulheres)
259         ax[1][3].set_yticklabels(label_homens)
260         ax[1][3].title.set_text(f"Rodada {m+1}")
261
262     if m == 9:
263
264         ax[1][4].matshow(mapa)
265         # Cria ticks para traçar as linhas do grid
266         ax[1][4].set_xticks(np.arange(-.5, len(label_homens)),
267                             minor = True)
268         ax[1][4].set_yticks(np.arange(-.5, len(label_mulheres)),
269                             minor = True)
270         ax[1][4].grid(True, which = 'minor', color = 'k', ls = ':',
271                       linewidth = 1)
272         # Define a posição dos ticks principais e altera labels
273         ax[1][4].set_xticks(np.arange(len(label_homens)))
274         ax[1][4].set_yticks(np.arange(len(label_mulheres)))
275         ax[1][4].set_xticklabels(label_mulheres)
276         ax[1][4].set_yticklabels(label_homens)
277         ax[1][4].title.set_text(f"Rodada {m+1}")
278
279     return casais
280
281     #####
282     #                                     Programa principal                                #
283     #####
284
285     # Sem solteiros (note que se todos preferem estar com alguém a ficar solteiro
286     # então é o mesmo caso em que não há opção de ficar solteiro)
287
288     preferencia_H2 = \
289         [[0, 4, 2, 8, 9, 3, 5, 1, 7, 6, -1], [2, 7, 0, 3, 4, 5, 1, 9, 8, 6, -1],
290          [7, 4, 0, 3, 1, 5, 8, 6, 2, 9, -1], [8, 5, 3, 6, 7, 4, 9, 1, 2, 0, -1],
291          [9, 3, 1, 2, 5, 4, 0, 8, 7, 6, -1], [1, 0, 3, 6, 4, 8, 2, 9, 7, 5, -1],
292          [6, 4, 8, 1, 2, 0, 3, 7, 9, 5, -1], [0, 4, 7, 5, 8, 2, 9, 1, 6, 3, -1],
293          [7, 2, 3, 6, 1, 0, 5, 8, 9, 4, -1], [0, 5, 9, 6, 4, 1, 3, 2, 8, 7, -1]]
294
295     preferencia_M2 = \
296         [[1, 5, 9, 6, 8, 0, 3, 4, 2, 7, -1], [1, 0, 2, 5, 6, 3, 8, 4, 9, 7, -1],
297          [5, 1, 4, 6, 7, 2, 8, 0, 3, 9, -1], [5, 9, 2, 0, 8, 7, 6, 3, 1, 4, -1],
298          [9, 7, 5, 3, 0, 6, 2, 4, 8, 1, -1], [1, 0, 4, 8, 9, 3, 5, 6, 2, 7, -1],
299          [9, 6, 7, 5, 1, 0, 2, 4, 3, 8, -1], [6, 9, 1, 0, 8, 3, 7, 4, 2, 5, -1],
300          [8, 2, 7, 6, 5, 1, 0, 4, 9, 3, -1], [4, 7, 6, 0, 1, 9, 2, 8, 5, 3, -1]]
301
302     # Com solteiros:

```

```

303
304 preferencia_H = \
305     [[0, 4, 2, 8, -1, 9, 3, 5, 1, 7, 6], [2, -1, 7, 0, 3, 4, 5, 1, 9, 8, 6],
306     [7, 4, 0, 3, 1, 5, 8, 6, -1, 2, 9], [-1, 8, 5, 3, 6, 7, 4, 9, 1, 2, 0],
307     [9, 3, 1, 2, 5, -1, 4, 0, 8, 7, 6], [1, 0, -1, 3, 6, 4, 8, 2, 9, 7, 5],
308     [6, 4, 8, 1, 2, 0, 3, 7, 9, 5, -1], [0, 4, 7, 5, 8, 2, 9, 1, 6, 3, -1],
309     [7, 2, -1, 3, 6, 1, 0, 5, 8, 9, 4], [0, 5, 9, 6, 4, -1, 1, 3, 2, 8, 7]]
310
311 preferencia_M = \
312     [[-1, 1, 5, 9, 6, 8, 0, 3, 4, 2, 7], [1, 0, -1, 2, 5, 6, 3, 8, 4, 9, 7],
313     [5, -1, 1, 4, 6, 7, 2, 8, 0, 3, 9], [5, 9, 2, 0, -1, 8, 7, 6, 3, 1, 4],
314     [9, 7, 5, 3, 0, 6, 2, 4, 8, 1, -1], [1, 0, 4, -1, 8, 9, 3, 5, 6, 2, 7],
315     [9, 6, 7, 5, 1, 0, 2, 4, 3, -1, 8], [6, 9, 1, 0, -1, 8, 3, 7, 4, 2, 5],
316     [8, 2, 7, 6, 5, 1, -1, 0, 4, 9, 3], [4, -1, 7, 6, 0, 1, 9, 2, 8, 5, 3]]
317
318 casais_com_solteiros = matching_com_solteiros(preferencia_H, preferencia_M)

```

A.11 Equilíbrio Geral

A.12 Desenho de Mecanismo