Modelos econômicos aplicados em Python[1]

Matheus L. Carrijo[2]
LEMC-FEARP/USP & DCM-FFCLRP/USP

22 de dezembro de 2021

[1]O 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.

[2]Faculdade 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

Lista de Figuras

Lista de Tabelas

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,

[1] Cabe 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:

```
def demanda(p, a = -2, b = 7):
    # Setting demand function (slope coefficient must be negative!)

    assert a < 0 and b > 0

    global a2, b2 # estabelece as variáveis como globais, isto é, elas são
                  # reconhecidas em todo o programa, não só dentro das funções
    a2, b2 = a, b

    return a*p + b

```

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

```python
def oferta(p, c = 4, d = 1):
    # Setting supply function (slope coefficient must be posive!)
    # d in (0,b) é suficiente pra equilibrio

    assert c > 0 and d > 0 and b2 > d

    global c2, d2
    c2, d2 = c, d

    return c*p + d

def excesso_demanda(p):
   # Setting excess demand function. It must be clear that equlibrium requires
   # excesso_demanda(p) = 0, for some especific p

   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:

```python
print(f"\nFaremos o cálculo para a função demanda D = {a2}p + {b2}", end = " ")
print(f"e oferta S = {c2}p + {d2}\n")

p_grid = np.linspace(-1000, 1000, 1000)
q_grid = np.linspace(-1000, 1000, 1000)

E_abs = []
for i in p_grid:
    E_abs.append(abs(excesso_demanda(i)))

indice_min = E_abs.index(min(E_abs))
p_equilibrio = p_grid[indice_min]
q_equilibrio = oferta(p_equilibrio)

print("O preço e a quantidade de equilíbrio são", end = "")
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:

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

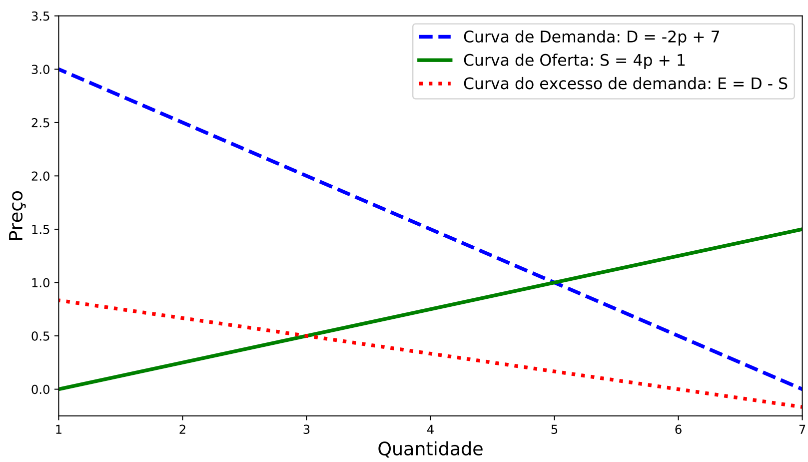

Figura 2.1: *Equilíbrio entre oferta e demanda.*

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 renda[1], 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 $f(k) \equiv F(K/L, 1)$.

[1] O 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

$$i = sy = sf(k), \tag{3.2}$$

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

$$\Delta k = sf(k) - \delta k, \tag{3.3}$$

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

```python
# Set the Cobb-Douglas production function:
def produto_agregado(k_agregado, l_agregado = 1):
    return np.sqrt(k_agregado*l_agregado)

# Set the Cobb-Douglas production function per worker:
def produto_por_trabalhador(k_agregado, l_agregado = 1):
    return produto_agregado(k_agregado)/l_agregado

```

```python
# Depreciation is a linear function of k per worker, i.e., k=K/L
def depreciacao(k_agregado, l_agregado = 1, taxa_depreciacao = 0.1):
    return taxa_depreciacao*(k_agregado/l_agregado)

def investimento(k_agregado, taxa_poupanca = 0.3):
    return taxa_poupanca*produto_por_trabalhador(k_agregado)

def consumo(k_agregado, taxa_poupanca = 0.3):
    return (1 - taxa_poupanca)*produto_por_trabalhador(k_agregado)

def k_variacao(k_agregado, taxa_poupanca = 0.3):
    return (taxa_poupanca*produto_por_trabalhador(k_agregado)
            - depreciacao(k_agregado))
```

O programa principal é dado pelo código abaixo:

```python
        ################################################################
        #                   Programa principal                         #
        ################################################################

taxa_depreciacao, taxa_poupanca, l_agregado, k_t0 = 0.1, 0.3, 1, 4

print("Faremos uma exemplo em que a taxa de poupança é", end = " ")
print(f"{taxa_poupanca}%, a taxa de depreciação é", end = " ")
print(f"{taxa_depreciacao}%, e que a economia comece com uma", end = " ")
print("relação de 4 capital por trabalhador")

tolerancia = 0.001
norma = 10 # número apenas para entrar no While
while norma > tolerancia:
    k_t1 = (taxa_poupanca*produto_por_trabalhador(k_t0) +
            k_t0*(1-taxa_depreciacao))
    norma = abs(k_t1 - k_t0)
    k_t0 = k_t1
```

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

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

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:

```
        ################################################################
        #                          Gráfico                             #
        ################################################################

k_grid = np.linspace(0, 2*k_t1, 1000)

plt.plot(k_grid, depreciacao(k_grid), color="blue", linewidth=3.0,
         ls = "dashed")
plt.plot(k_grid, investimento(k_grid), color="green", linewidth=3.0)
plt.xlim(0, 2*k_t1) # entre -5 e 5 por exemplo
plt.ylim(0, 2*investimento(k_t1))
plt.legend([f'Depreciação, {taxa_depreciacao}k',
            f'Investimento, {taxa_poupanca}f(k)'])
plt.ylabel("Investimento e Depreciação", fontsize = 15)
plt.xlabel("Capital por trabalhador, k", fontsize = 15)
plt.show()
```

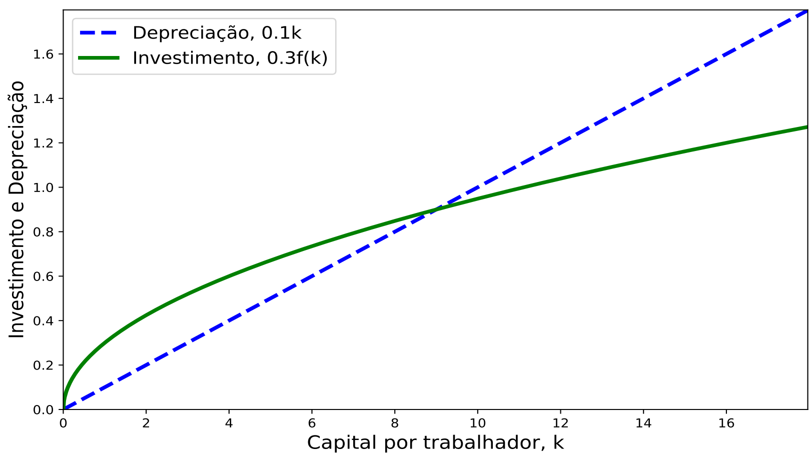

Figura 3.1: *Investimento, depreciação e estado estacionário.*

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

$$Y = C + I + G. \tag{4.1}$$

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,

$$
\begin{aligned}
C &= C(Y - T) \\
I &= I(r) \\
G &= \bar{G},
\end{aligned}
$$

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

$$
\begin{aligned}
C &= C_1(Y - T) + C_0 = C_1(Y - tY) + C_0 = C_1 Y(1 - t) + C_0 \\
I &= I_0 + I_1 r,
\end{aligned}
$$

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

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

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

[1] É 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:

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

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

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

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:

```python
        ################################################################
        #                       Definindo funções                      #
        ################################################################

def investimento(taxa_juros):
    """
    Dada a taxa de juros, calcula o nível de investimento. Deve ser
    negativamente relacionada com a taxa de juros. Por simplicidade,
    consideramos o caso linear.
    """

    if a < 0 and b > 0:
        return a*taxa_juros + b

def demanda_monetaria(taxa_juros, produto):
    """
    Modelamos a demanda monetária tentando incorporar linearmente os motivos
    precaução, especulação, e transação, respectivamente pelas variáveis
    e, c, d.
    """

    return c*produto + d*taxa_juros + e

def consumo(produto):
    """
    Consideraremos o consumo como uma função linear da renda disponível, de
    modo que o coeficiente de inclinação é a propensão marginal a consumir e
    a constante é o consumo autônomo.
    """

    return (propensao_marginal_a_consumir * (produto - aliquota_imposto)
            + consumo_autonomo)

def produto_IS(taxa_juros):
    """
    Construímos a curva IS através da equação Y = C + I + G, em que C é a
    função consumo (que depende da renda disponível), I é o investimento,
    dependente da taxa de juros, e G são os gastos do governo. Como o consumo
    depende do nível de renda, esta equação é manipulada até que tenhamos uma
    expressão do produto em relação à taxa de juros
    """

    I = investimento(taxa_juros)

    return ((consumo_autonomo + I + gastos_governo) /
```

```
            (1 - propensao_marginal_a_consumir
            * (1 - aliquota_imposto)))


def produto_LM(taxa_juros):
    """
    Construímos a curva LM através da equação
    oferta_monetaria = demanda_monetaria = c*produto + d*taxa_juros + e,
    em que a oferta é exógena e a demanda calculada pela função. Então, a
    expressão pode ser manipulada até que tenhamos uma função de Y em função
    da taxa de juros.
    """

    return ((oferta_monetaria/nivel_precos - e) / c - taxa_juros * (d / c))

def excesso_ISLM(taxa_juros):
    """
    Função para calcular o equilíbrio fazendo excesso_ISLM = 0, isto é,
    o par (r*, Y*) ótimo que nos dá a intersecção entre as curvas IS e LM.
    """

    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.

$$\text{Consumo: } C = 0.3Y + 100$$

$$\text{Investimento: } I = -20r + 5$$

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

$$\text{Oferta monetária: } (M/P)^S = 10$$

$$\text{Função Consumo: } C = 0.3Y + 100$$

$$\text{Política fiscal: } (G, T) = (50, 0.2Y)$$

```
        ################################################################
        #                   Programa principal                         #
        ################################################################

a, b, propensao_marginal_a_consumir, consumo_autonomo = -20, 5, 0.3, 100
gastos_governo, aliquota_imposto, c, d, e = 50, .2, 10, -300, 50
nivel_precos, oferta_monetaria = 10, 100

print("\nFaremos uma exemplo em que a função investimento é", end = " ")
print(f"dada por {a}r + {b};", end = " ")
print("a função demanda monetária, por", end = " ")
print(f"{c}Y + {d}i + {e};", end = " ")
print("a função consumo, por", end = " ")
print(f"{propensao_marginal_a_consumir}Y + {consumo_autonomo};", end = " ")
print(f"e a política fiscal por G = {gastos_governo} e", end = " ")
print(f"T = {aliquota_imposto}Y.", end = " ")
print("Ainda, a oferta monetária real é dada por M/P =", end = " ")
print(f"{oferta_monetaria/nivel_precos}.")


# Calculo do equilibrio

taxa_juros_grid = np.linspace(0, 1000, 100000)

Excesso_abs = []

for i in taxa_juros_grid:

    Excesso_abs.append(abs(excesso_ISLM(i)))

indice_min = Excesso_abs.index(min(Excesso_abs))
juros_equilibrio = taxa_juros_grid[indice_min]
produto_equilibrio = produto_IS(juros_equilibrio)
```

```
print("\nPortanto, a taxa de juros e produto de equilíbrio são,", end = "")
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.

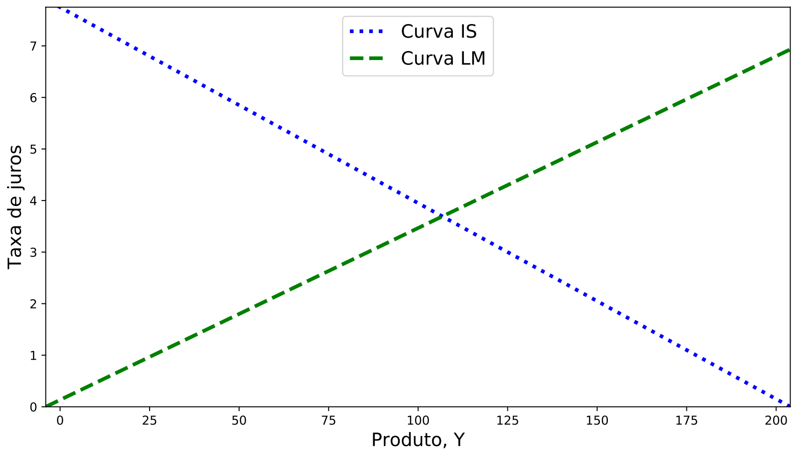

Figura 4.1: *O equilíbrio simultâneo nos mercados de bens e serviços e monetário.*

O código responsável por gerar este gráfico é mostrado abaixo.

```
        ##################################################################
        #                          Gráfico                               #
        ##################################################################

valores_produto = []

for i in taxa_juros_grid:

    valores_produto.append(abs(produto_IS(i)))

y = taxa_juros_grid[np.argmin(valores_produto)]

# O código acima foi para saber a taxa de juros (no grid) que torna a curva IS
# nula. Isto será útil para determinar a escala do gráfico no eixo y (vertical)

```

```python
# Gráficos:

plt.plot(produto_IS(taxa_juros_grid), taxa_juros_grid, color="blue",
         linewidth=3.0, ls = "dotted")
plt.plot(produto_LM(taxa_juros_grid), taxa_juros_grid, color="green",
         linewidth=3.0, ls = "dashed")
plt.ylim(0, y)
plt.xlim(min(produto_LM(taxa_juros_grid)),
         max(produto_IS(taxa_juros_grid)))
plt.legend(['Curva IS', 'Curva LM'])
# plt.title("Modelo IS-LM", fontsize = 20)
plt.ylabel("Taxa de juros", fontsize = 15)
plt.xlabel("Produto, Y", fontsize = 15)
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 Pyhton 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,

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

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:

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

```python

    #Definindo as variáveis com o valor dos parâmetros

    forca_trabalho = empregados + desempregados

    A = np.array([[1 - Lambda, Alpha    ],
                  [Lambda     , 1 - Alpha]])
    x_inicial = np.array([[desempregados/forca_trabalho],
                          [empregados/forca_trabalho]])

    tempo = []
    eixo_taxa_desemprego = [x_inicial[0][0]]
    eixo_taxa_emprego = [x_inicial[1][0]]
    t = 0
    tol = 1e-04
    dif = tol + 1

    # Calculando o nível estacionário

    while dif > tol:

        x = A @ x_inicial
        tempo.append(t)
        eixo_taxa_desemprego.append(x[0][0])
        eixo_taxa_emprego.append(x[1][0])
        dif = np.max(np.abs(x - x_inicial))
        x_inicial = x
        t += 1
    tempo.append(t)

    # Informando ao usuário

    print(f"\nPara uma economia com {empregados} empregados,", end = " ")
    print(f"{desempregados} desempregados, uma taxa de obtenção de", end = " ")
    print(f"emprego de {100*Lambda: .2f}%, e uma taxa de demissão", end = " ")
    print(f"de {100*Alpha: .2f}%, a taxa de desemprego no nível", end = " ")
    print(f"estacionário é dada por {100*x[0][0]: .2f}% \n")

    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:

$$\text{Taxa de obtenção de emprego: } \lambda = 0.391$$
$$\text{Taxa de demissão: } \alpha = 0.027$$
$$\text{Número de empregados na economia: } E = 250.000$$
$$\text{Número de desempregados na economia: } U = 50.000$$

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.

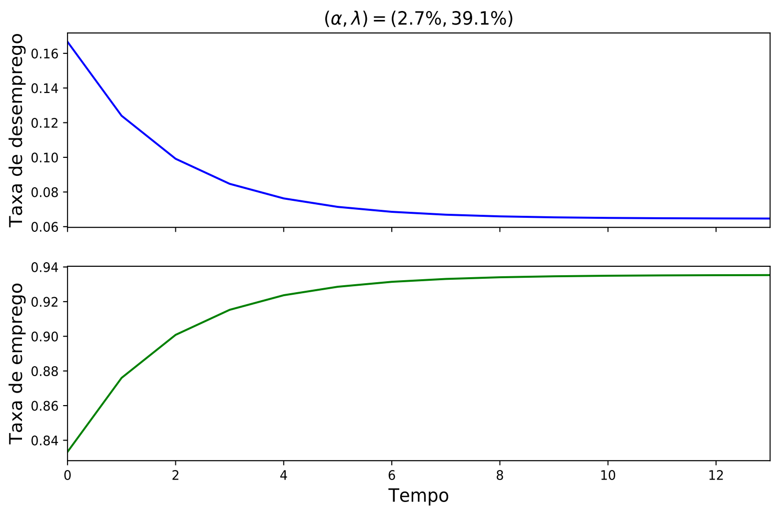

Figura 5.1: *Exemplo de uma economia em seu nível estacionário de desemprego.*

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

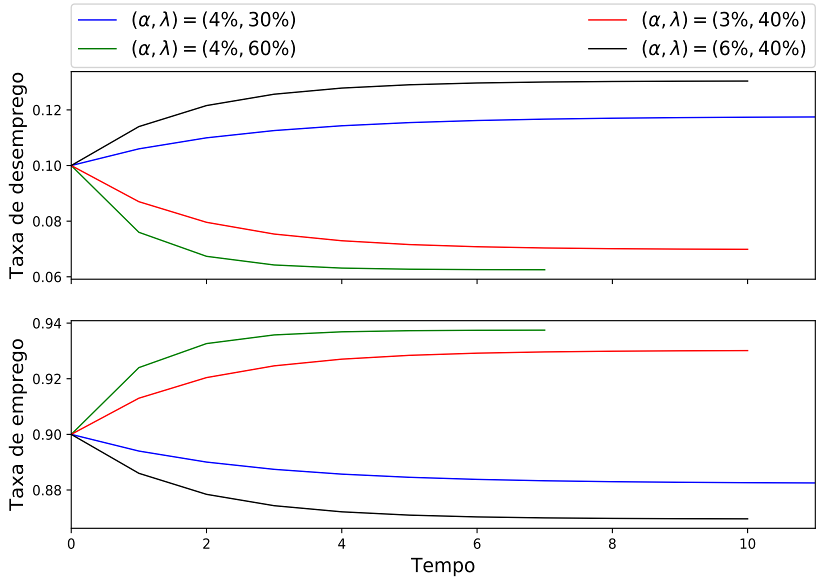

Figura 5.2: *Estática comparativa das taxas de desemprego e emprego.*

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:

```
        ###############################################################
        #                Uma economia com:                            #
        #                - empregados = 250_000                       #
        #                - desempregados = 50_000                     #
        #                - Alpha = .027                               #
        #                - Lambda = .391                              #
        ###############################################################

exemplo_padrao = nivel_estacionario()

        ###############################################################
        #                         Gráfico                             #
        ###############################################################

fig, ax = plt.subplots(nrows = 2, ncols = 1, sharex  = True)

# plot desemprego:

ax[0].plot(exemplo_padrao[1], exemplo_padrao[2], color = "blue", lw = 1.5)
ax[0].set_xlim(0, exemplo_padrao[1][-1])
```

```python
ax[0].set_ylabel("Taxa de desemprego")
#ax[0].legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower center',
#              ncol=2, mode="expand", borderaxespad=0.)
ax[0].set_title(r'$(\alpha, \lambda) = (2.7\%, 39.1\%)$', fontsize = 10)

# plot emprego:

ax[1].plot(exemplo_padrao[1], exemplo_padrao[3], color = "green", lw = 1.5)
ax[1].set_xlabel("tempo")
ax[1].set_ylabel("Taxa de emprego")

        ################################################################
        #         Variando parâmetros (estática comparativa)           #
        ################################################################

# Variando Lambda

estatica1 = nivel_estacionario(450_000, 50_000, .04, .3)
estatica2 = nivel_estacionario(450_000, 50_000, .04, .6)

# Variando Alpha

estatica3 = nivel_estacionario(450_000, 50_000, .03, .4)
estatica4 = nivel_estacionario(450_000, 50_000, .06, .4)

        ################################################################
        #                          Gráfico                             #
        ################################################################

fig, ax = plt.subplots(nrows = 2, ncols = 1, sharex  = True)

t_max = max(estatica1[1][-1], estatica2[1][-1], estatica3[1][-1],
            estatica4[1][-1]) # limite eixo x

# plot desemprego:

ax[0].plot(estatica1[1], estatica1[2], color = "blue", lw = 1,
           label = r'$(\alpha, \lambda) = (4\%, 30\%)$')
ax[0].plot(estatica2[1], estatica2[2], color = "green", lw = 1,
           label = r'$(\alpha, \lambda) = (4\%, 60\%)$')
ax[0].plot(estatica3[1], estatica3[2], color = "red", lw = 1,
           label = r'$(\alpha, \lambda) = (3\%, 40\%)$')
ax[0].plot(estatica4[1], estatica4[2], color = "black", lw = 1,
           label = r'$(\alpha, \lambda) = (6\%, 40\%)$')
ax[0].set_xlim(0, t_max)
ax[0].set_ylabel("Taxa de desemprego")
```

```python
ax[0].legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower center',
             ncol=2, mode="expand", borderaxespad=0.)

# plot emprego:

ax[1].plot(estatica1[1], estatica1[3], color = "blue", lw = 1)
ax[1].plot(estatica2[1], estatica2[3], color = "green", lw = 1)
ax[1].plot(estatica3[1], estatica3[3], color = "red", lw = 1)
ax[1].plot(estatica4[1], estatica4[3], color = "black", lw = 1)
ax[1].set_xlim(0, t_max)
ax[1].set_xlabel("tempo")
ax[1].set_ylabel("Taxa de emprego")
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:

```python
def individual_path(Lambda = 0.3, Alpha = .027, desempregado = True):
    """

    Parameters
    ----------
    Lambda : float, optional
        Taxa de obtenção de emprego. The default is 0.3.
    Alpha : float, optional
        Taxa de demissão. The default is .027.
    desempregado : bool, optional
        informa se os indivíduos estão desempregados. The default is True.

    Returns
    -------
    tempo : list
        lista em que a i-ésima entrada corresponde ao tempo que o i-esimo
        trabalhador demorou pra encontrar emprego (caso esteja desempregado)
        ou que demorou pra ficar desempregado (caso esteja empregado).

    """

```

```
    if desempregado:

        tempo = []

        j = 1
        while j <= 1000: # amostra de 1000 individuos

            encontrou_emprego = False
            t = 0
            while not encontrou_emprego:
            # enquanto a pessoa j não encontrar emprego, ela continua
            # procurando

                r = np.random.rand()

                if r <= Lambda: # desempregado encontra emprego

                    encontrou_emprego = True

                t += 1

            tempo.append(t)
            j += 1

    else:

        tempo = []

        j = 1
        while j <= 1000: # amostra de 1000 individuos

            ficou_desempregado = False
            t = 0
            while not ficou_desempregado: # fica no emprego até ser demitido

                r = np.random.rand()

                if r <= Alpha: # empregado ficou desempregado

                    ficou_desempregado = True
                    t += 1

                t += 1

            tempo.append(t)
            j += 1
```

```python
    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 $t, \forall t \in \{1, 2, ...\}$.

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:

```python
exemplo_padrao2 = individual_path()

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

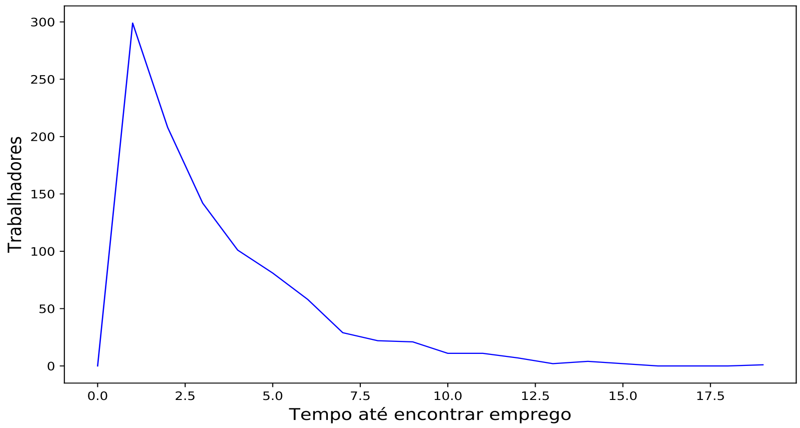

Figura 5.3: *Frequência dos trabalhadores encontrando emprego no tempo.*

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.

```
exemplo_padrao3 = individual_path(Lambda = 0.2)
exemplo_padrao4 = individual_path(Lambda = 0.4)
exemplo_padrao5 = individual_path(Lambda = 0.65)
exemplo_padrao6 = individual_path(Lambda = 0.9)

aux = max(max(exemplo_padrao2), max(exemplo_padrao3),
          max(exemplo_padrao4), max(exemplo_padrao5))

hist2, valores2 = np.histogram(exemplo_padrao3, bins = np.arange(0, aux))
hist3, valores3 = np.histogram(exemplo_padrao4, bins = np.arange(0, aux))
hist4, valores4 = np.histogram(exemplo_padrao5, bins = np.arange(0, aux))
hist5, valores5 = np.histogram(exemplo_padrao6, bins = np.arange(0, aux))

plt.plot(valores2[:-1], hist2, color="black", linewidth=1)
plt.plot(valores3[:-1], hist3, color="green", linewidth=1)
plt.plot(valores4[:-1], hist4, color="red", linewidth=1)
plt.plot(valores5[:-1], hist5, color="orange", linewidth=1)
plt.legend([r'$\lambda = 10\%$', r'$\lambda = 40\%$', r'$\lambda = 70\%$',
            r'$\lambda = 90\%$'])
plt.ylabel("Trabalhadores", fontsize = 10)
plt.xlabel("Tempo até encontrar emprego", fontsize = 10)
plt.show()
```

Este código gerará os histogramas da figura 5.4.

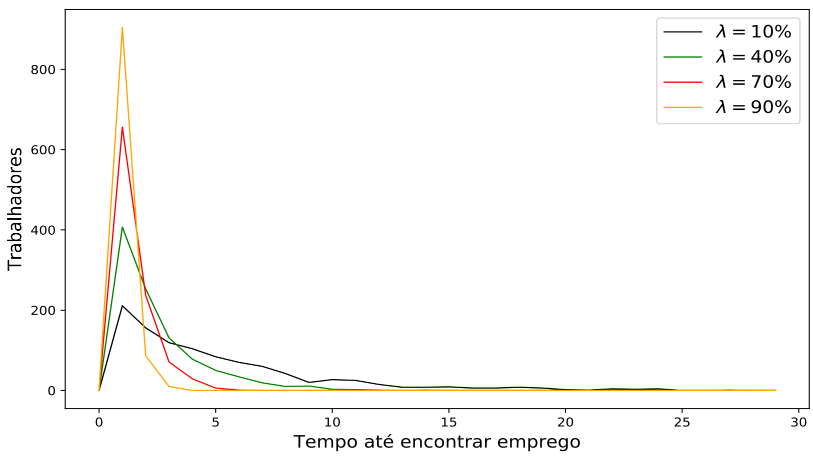

Figura 5.4: *Estática comparativa: frequência de trabalhadores encontrando emprego no tempo.*

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 consumir, 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

$$c_2 = m_2 + (1 + r)(m_1 - c_1) \tag{6.1}$$

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, consequentemente, $c_2 > m_2$.

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

$$(c_1^*, c_2^*) = \left(\frac{m_1(1+r) + m_2}{(1+r)[1+(1+r)^{(1-\gamma)/\gamma}]}, (1+r)(m_1 - c_1^*) + m2 \right), \tag{6.2}$$

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.

```
        ################################################################
        #                          Funções                             #
        ################################################################

def u(c, a):
    if a > 0 and a != 1:
        return (c**(1-a) - 1) / (1 - a)
    else:
        if a == 0:
            return np.log(c)

def U(c1, c2, a, b):
    return u(c1, a) + u(c2, b)

def budget(c1, m1, m2, r):
    if m1 > 0 and m2 > 0 and r > 0 and r < 1:
        return (1+r)*m1 + m2 - (1+r)*c1

def inverse_budget(c2, m1, m2, r):
    if m1 > 0 and m2 > 0 and r > 0 and r < 1:
        return (m2-c2)/(1+r) + m1

def choice(a, b, r, m1, m2):
    if (a > 0 and a != 1 and b > 0 and b != 1 and m1 > 0 and m2 > 0 and r > 0
        and r < 1):
        c1_estrela = (m1+m2/(1+r)) / (1+(1+r)**((1-a)/a))
        c2_estrela = (m1 - c1_estrela)*(1+r) + m2
        funcao_valor = U(c1_estrela, c2_estrela, a, b)

    return c1_estrela, c2_estrela, funcao_valor
```

Em seguida, um exemplo de economia é implementado pelo código

```
a, b, r, m1, m2 = 0.5, 0.5, 0.5, 50, 750
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.

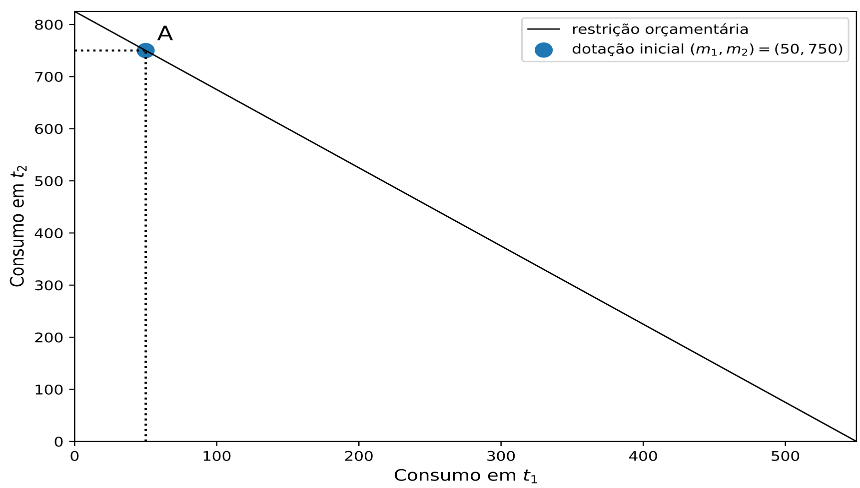

Figura 6.1: *Restrição orçamentária.*

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.

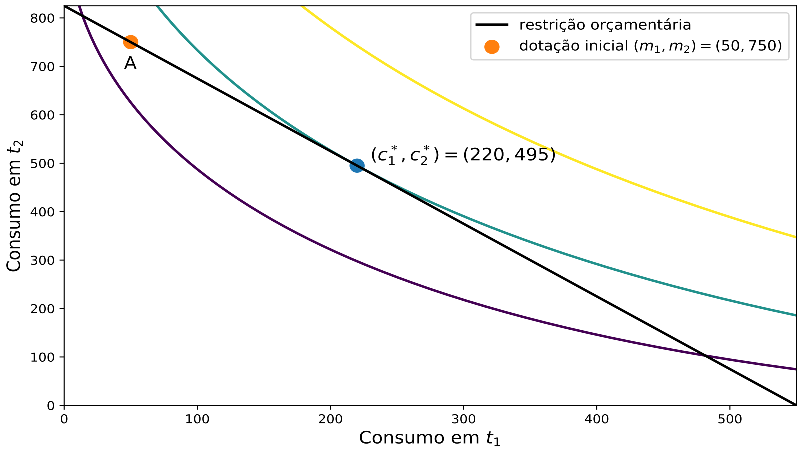

Figura 6.2: *Consumo ótimo para um exemplo de economia.*

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:

```
# Outros exemplos
a2, b2, r2, m1_2, m2_2 = 0.5, 0.5, 0.1, 50, 750
a3, b3, r3, m1_3, m2_3 = 0.5, 0.5, 0.9, 50, 750
exemplo2 = choice(a2, b2, r2, m1_2, m2_2)
exemplo3 = choice(a3, b3, r3, m1_3, m2_3)

# configuração exemplo1
budget1 = budget(eixo_x, m1, m2, r)

# configuração exemplo2
budget2 = budget(eixo_x, m1_2, m2_2, r2)

# configuração exemplo3
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).

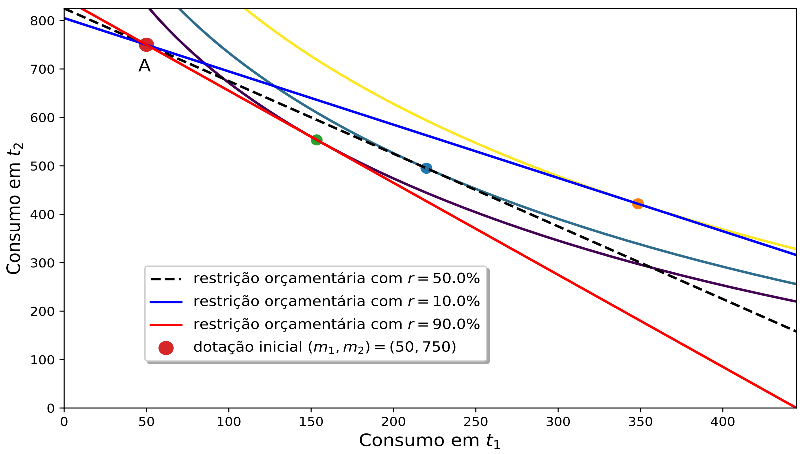

Figura 6.3: *Consumo ótimo para diferentes taxas de juros.*

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

```
plt.rcParams["figure.figsize"] = (11, 6)
plt.rcParams["lines.linewidth"] = (2.5)

# plotando exemplo1 (mesmo que anterior)
Z1 = plt.contour(X, Y, U(X, Y, a, b), levels = [exemplo3[2], exemplo1[2],
                                                  exemplo2[2]], linewidths = 2)
plt.plot(eixo_x, budget1, color = "black", ls = "dashed", lw = 2,
         label = f"restrição orçamentária com $r = {100*r}\%$")

plt.rcParams["figure.figsize"] = (11, 6)
plt.rcParams["lines.linewidth"] = (2.5)

# plotando exemplo2
plt.plot(eixo_x, budget2, color = "blue", lw = 2,
         label = f"restrição orçamentária com $r = {100*r2}\%$")

# plotando exemplo3
plt.plot(eixo_x, budget3, color = "red", lw = 2,
         label = f"restrição orçamentária com $r = {100*r3}\%$")
```

```python

# Marcando pontos na tangência da reta orçamentária com as curvas de utilidade
plt.scatter([exemplo1[0]], [exemplo1[1]], s = 4*4**2)
plt.scatter([exemplo2[0]], [exemplo2[1]], s = 4*4**2)
plt.scatter([exemplo3[0]], [exemplo3[1]], s = 4*4**2)
plt.scatter([m1],[m2], s = 7*4**2,
            label = f"dotação inicial $(m_1, m_2) = ({m1},{m2})$")

plt.annotate("A", (m1-5,m2-55), fontsize = 12)

# configuração geral plot
plt.legend(loc = 'lower left', bbox_to_anchor=(0.1, 0.1),
           fancybox = True, shadow = True, fontsize = 11.5)
plt.ylim(0, (1+r)*m1 + m2)
plt.xlim(0, m1_3 + m2_3/(1+r3))
plt.ylabel(r"Consumo em $t_2$", fontsize = 12)
plt.xlabel(r"Consumo em $t_1$", fontsize = 12)
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.

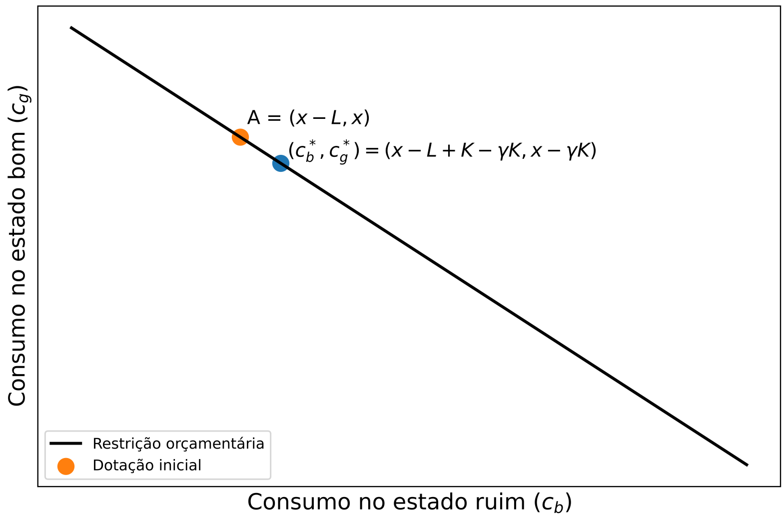

Figura 7.1: *A restrição orçamentária.*

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

$$c_g = -\frac{\gamma}{1-\gamma}c_b + b. \tag{7.1}$$

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,

$$TMS_{12} = \frac{UMg_{c_b}}{UMg_{c_g}} = \frac{-\gamma}{1-\gamma}. \tag{7.2}$$

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

$$U(c_b, c_g) = \pi_b u(c_b) + \pi_g u(c_g), \tag{7.3}$$

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

$$\frac{-\gamma}{1-\gamma} = \frac{-\pi_b}{\pi_g}\frac{c_g}{c_b} \Longrightarrow c_g^* = \frac{\pi_g}{\pi_b}\frac{\gamma}{1-\gamma}c_b^*. \tag{7.4}$$

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

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

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

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

Esta solução analítica será útil na descrição do programa.

7.2 Implementação computacional

As funções úteis ao modelo são implementadas pelo código abaixo:

```
def utilidade(c_estado_b, c_estado_g, prob_estado_b, prob_estado_g):

    return prob_estado_b*np.log(c_estado_b) + prob_estado_g*np.log(c_estado_g)

def restr_orcamentaria(c_estado_b, gamma, dot_estado_b, dot_estado_g):

    b = dot_estado_g + dot_estado_b*gamma/(1-gamma)

    return b - (gamma*c_estado_b)/(1-gamma)

def escolha(dot_estado_b, dot_estado_g, prob_estado_b, prob_estado_g, gamma):

    b = dot_estado_g + dot_estado_b*gamma/(1-gamma)
    c = prob_estado_g/prob_estado_b

    c_b_estrela = b*prob_estado_b*(1-gamma)/gamma
    c_g_estrela = c_b_estrela*c*(gamma/(1-gamma))
    funcao_valor = utilidade(c_b_estrela, c_g_estrela, prob_estado_b,
                             prob_estado_g)

    return c_b_estrela, c_g_estrela, funcao_valor

def seguro_otimo(dot_estado_b, dot_estado_g, prob_estado_b, prob_estado_g,
                 gamma):

    escolha_otima = escolha(dot_estado_b, dot_estado_g, prob_estado_b,
                            prob_estado_g, gamma)

    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:

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

```
exemplo_1 = escolha(dot_estado_b1, dot_estado_g1, prob_estado_b1,
                    prob_estado_g1, gamma_1)

seguro_estrela = seguro_otimo(dot_estado_b1, dot_estado_g1, prob_estado_b1,
                    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.

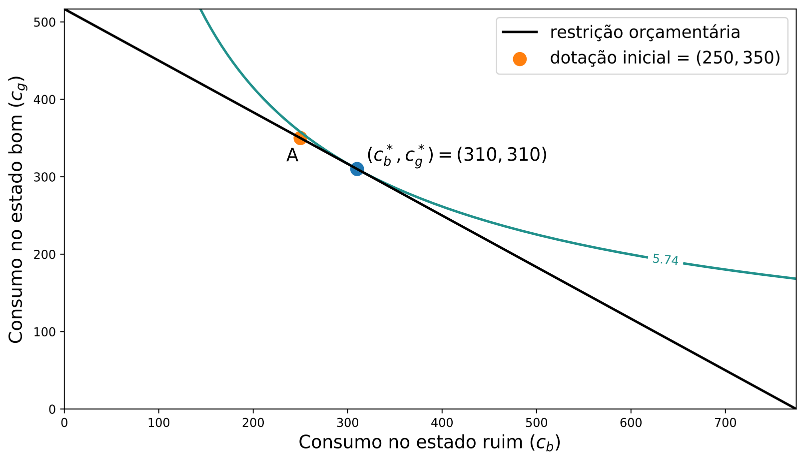

Figura 7.2: *Escolha ótima do consumidor para um exemplo de economia.*

O código que gera o gráfico é dado por:

```
c_estado_b1 = np.linspace(0.01, 1000, 1000)
c_estado_g1 = np.linspace(0.01, 1000, 1000)
orcamento = restr_orcamentaria(c_estado_b1, gamma_1, dot_estado_b1,
                               dot_estado_g1)
X, Y = np.meshgrid(c_estado_b1, c_estado_g1)

Z = plt.contour(X, Y, utilidade(X, Y, prob_estado_b1, prob_estado_g1),
                levels = [exemplo_1[2]-10, exemplo_1[2], exemplo_1[2]+10],
                linewidths = 2)
plt.plot(c_estado_b1, orcamento, color = "black", lw = 2,
         label = "restrição orçamentária")
plt.clabel(Z, inline = 1, fontsize = 10)
plt.ylim(0, dot_estado_g1 + dot_estado_b1*gamma_1/(1-gamma_1))
plt.xlim(0, (1-gamma_1)*(dot_estado_g1 + dot_estado_b1*gamma_1/(1-gamma_1))
         /gamma_1)
plt.ylabel(r"Consumo no estado bom ($c_g$)", fontsize = 15)
plt.xlabel(r"Consumo no estado ruim ($c_b$)", fontsize = 15)
plt.scatter([exemplo_1[0]], [exemplo_1[1]], s = 7*4**2)
plt.annotate(f"$(c_b^*, c_g^*) = ({exemplo_1[0]: .0f}, {exemplo_1[1]: .0f})$",
             (exemplo_1[0]+10, exemplo_1[1]+10), fontsize = 13)
plt.scatter([dot_estado_b1],[dot_estado_g1], s = 7*4**2,
            label = f"dotação inicial = $({dot_estado_b1},{dot_estado_g1})$")
plt.annotate("A", (dot_estado_b1-15,dot_estado_g1-30), fontsize = 13)
plt.legend()
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	
		D	ND
Jogador 1	D	(-2,-2)	(0,-3)
	ND	(-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	
		s	h
Jogador 1	s	(a,a)	(c,b)
	h	(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:

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

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

```python
def payoff(jogo):
    """



    Parameters
    ----------
    jogo : Lista.
        A primeira entrada é a jogada do primeiro jogador e a segunda entrada
        é a jogada do segundo jogador. Para um jogo ser válido, ele deve estar
        dentro da matriz de jogos possíveis.

    Returns
    -------
    Lista
        Retorna uma lista em que a primeira entrada representa o payoff do
        primeiro jogador e a segunda entrada o payoff do segundo jogador.

    """

    if jogo in matriz:
```

[1] A documentação desta biblioteca pode ser encontrata clicando aqui.

```python

        if jogo == ['s', 's']:
            return [5, 5]
        if jogo == ['s', 'h']:
            return [0, 3]
        if jogo == ['h', 's']:
            return [3, 0]
        if jogo == ['h', 'h']:
            return [3, 3]

    else:

        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.

```python
def rodada():
    """



    Returns
    -------
    jogo : Lista.
        Retorna uma lista contendo o jogo feito pela escolha das estratégias
        pelos dois jogadores.

    """

    jogada_A, jogada_B = rd.choice(estrategia), rd.choice(estrategia)
    jogo = [jogada_A, jogada_B]

    return jogo
```

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

```
def nash_equilibrium(jogo):
    """


    Parameters
    ----------
    jogo : Lista
        Recebe uma lista do jogo determinado pelas ações (escolha das
        estratégias) dos jogadores.

    Returns
    -------
    Bool
        Assim como na função anterior, calcula o equilíbrio de Nash. Aqui,
        porém, fazemos um algorítmo em que o cálculo do equilíbrio baseia-se
        na análise da mudança de estratégia unilateral de cada jogador: se
        houver uma melhora no payoff dos jogadores, então não há equilibrio;
        caso contrário, há.

    """

    if jogo in matriz:

        payoffs = payoff(jogo)
        equilibrio = True

        if jogo[0] == 'h':

            jogo[0] = 's'
            payoff_aux = payoff(jogo)

            if payoff_aux[0] > payoffs[0]:

                equilibrio = False

            jogo[0] = 'h'

        if jogo[0] == 's':

            jogo[0] = 'h'
            payoff_aux = payoff(jogo)

            if payoff_aux[0] > payoffs[0]:

                equilibrio = False
```

```
            jogo[0] = 's'

        if jogo[1] == 'h':

            jogo[1] = 's'
            payoff_aux = payoff(jogo)

            if payoff_aux[1] > payoffs[1]:

                equilibrio = False

            jogo[1] = 'h'

        if jogo[1] == 's':

            jogo[1] = 'h'
            payoff_aux = payoff(jogo)

            if payoff_aux[1] > payoffs[1]:

                equilibrio = False

            jogo[1] = 's'

        return equilibrio

    else:

        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:

```
estrategia = ['s', 'h'] # as possíveis ações dos jogadores são caçar veado
                        # (jogar "s") ou caçar lebre (jogar "h").
matriz = [['s', 's'], ['s', 'h'],
          ['h', 's'], ['h', 'h']] # jogos possíveis determiandos 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.

```
equilibrio = False # Para entrar no loop abaixo
i = 1 # contagem do número de jogos

while not equilibrio: #Jogos serão feitos até que se alcance um equilibrio

    jogo = rodada() #Um jogo é uma lista com as jogadas dos dois jogadores
    equilibrio = nash_equilibrium(jogo) #Confere se o jogo constitui equilibrio
    payoffs = payoff(jogo) #Calcula o Payoff do jogo

    print("\nJogaremos até encontrarmos um jogo com Equilíbrio de Nash.")

    if jogo == ['s', 'h']:

        print(f"\nJogo {i}: O jogador A falhou em tentar caçar", end = " ")
        print("o veado sozinho; o jogador B conseguiu caçar", end = " ")
        print("a lebre sozinho. Como o jogador A poderia", end = " ")
        print("melhorar sua situação mudando sua estratégia e", end = " ")
        print("caçar uma lebre sozinho, este jogo não", end = " ")
        print("constitui um Equilíbrio de Nash.")

    if jogo == ['h', 's']:

        print(f"\nJogo {i}: O jogador B falhou em tentar caçar", end = " ")
        print("o veado sozinho; o jogador A conseguiu caçar", end = " ")
        print("a lebre sozinho. Como o jogador B poderia", end = " ")
        print("melhorar sua situação mudando sua estratégia e", end = " ")
        print("caçar uma lebre sozinho, este jogo não", end = " ")
        print("constitui um Equilíbrio de Nash.")

    if jogo == ['h', 'h']:

        print(f"\nJogo {i}: Ambos os jogadores escolheram", end = " ")
        print("caçar lebres e conseguiram. Como nenhum jogador", end = " ")
        print("tem a ganhar mudando sua estratégia unilateralme", end = "")
        print("nte para caçar um veado, então este jogo constit", end = "")
        print("ui um Equilíbrio de Nash. Note, no entanto, que ambo", end = "")
        print("s melhorariam sua situação se combinassem de caçar um veado.")

    if jogo == ['s', 's']:

        print(f"\nJogo {i}: Ambos os jogadores escolheram", end = " ")
```

```
        print("caçar veados e conseguiram. Como nenhum jogador", end = " ")
        print("tem a ganhar mudando sua estratégia unilateralme", end = "")
        print("nte para caçar uma lebre, então este jogo constit", end = "")
        print("ui um Equilíbrio de Nash. Note que este é o melhor", end = " ")
        print("resultado que os jogadores poderiam ter.")

    i += 1

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 raciocíno 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.

[2]Dizemos, 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

[1] A 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:

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

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

$$y_1 = f_1(y_2^e). \tag{9.2}$$

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

$$y_2 = f_2(y_1^e). \tag{9.3}$$

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

$$\begin{cases} y_1 = \frac{a-by_2}{2b} \\ y_2 = \frac{a-by_1}{2b}. \end{cases} \tag{9.4}$$

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.

```
def demanda_inversa(Y, a = 1000, b = 1): #preço em função da quantidade
    """
    Definindo a demanda inversa, que depende da produção total da economia e
    dos parâmetros 'a' e 'b'. O parâmetro Y é uma lista em que a entrada i
    representa a produção da empresa i.
    """

    assert a > 0 and b > 0, "Os parâmetros da demanda devem ser positivos"

    global c, d
    c, d = a, b

    return a - b*(sum(Y))

def receita(y, Y):
    """
```

```
    Receita = preço*quantidade. No caso do oligopólio, o preço é dado pela
    demanda inversa e a quantidade determinada pela firma.
    """

    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:

```
def curva_reacao():
    """
    Serve para o caso em que numero_firmas = 2. A função resolve o problema de
    maximização da firma arbitrária (1 ou 2) em função da produção que ela
    espera da outra firma, ou seja, a função gera a curva de melhor resposta.
    O equiilibrio de Cournot será a intersecção entre ambas as curvas.
    """

    demanda_inversa([0,1]) # Chamo a função apenas para definir o parâmetro 'c'

    # Resolvendo o problema para uma empresa arbitrária (empresa 1 ou 2):

    receita_valores = []
    y_argmax = []
    y, y_e = np.linspace(0, c, c+1), np.linspace(0, c, c+1)

    for j in y_e:

        for i in y:

            receita_valores.append(receita(i, [i,j]))

        indice_max = np.argmax(receita_valores) # Guarda o índice da maior
                                                # receita
        y_argmax.append(y[indice_max]) # Adiciona a uma lista o nível de
                                       # produção y que maximiza o lucro
        receita_valores = [] # "zera" a lista de receita para o algoritmo
                             # resolver novamente o problema de maximizacao de
                             # lucro para outro nível de produção esperado

    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:

```python
def equilibrio_cournot(numero_firmas = 2):
    """
    Esta função serve para o caso geral em que numero_firmas = n. A solução é
    implementada algebricamente. As firmas são idênticas e o custo é nulo. O
    cálculo é realizado para o caso especial de demanda inversa linear.
    """

    coeficientes = np.array([[2 if j==i else 1 for i in range(numero_firmas)]
                    for j in range(numero_firmas)])
    inversa_coeficientes = np.linalg.inv(coeficientes)

    constantes = np.array([c/d for i in range(numero_firmas)])

    vetor_producao = np.dot(inversa_coeficientes, constantes)

    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.

```python
curva_de_reacao = curva_reacao()

equilibrio_cournot1 = equilibrio_cournot() # Para n=2 firmas

quantidade_equilibrio = sum(equilibrio_cournot1)

preco_equilibrio = demanda_inversa(equilibrio_cournot1)
```

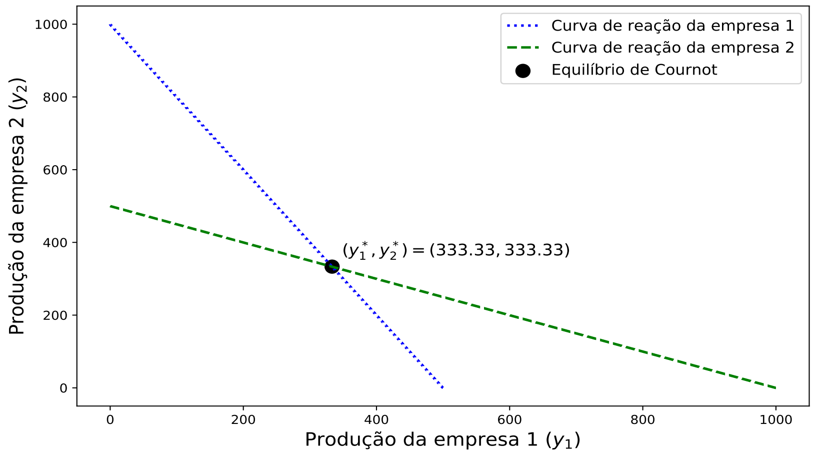

Figura 9.1: *O equilíbrio de Cournot.*

O código que gera este gráfico é dado a seguir.

```python
# Gráfico para o exemplo de 2 firmas (curvas de reacao):

plt.plot(curva_de_reacao[1], curva_de_reacao[0], color="blue", linewidth=2.0,
         ls = "dotted", label = "Curva de reação da empresa 1")
plt.plot(curva_de_reacao[0], curva_de_reacao[1], color="green", linewidth=2.0,
         ls = "dashed", label = "Curva de reação da empresa 2")
plt.xlabel(r"Produção da empresa 1 ($y_1$)", fontsize = 15)
plt.ylabel(r"Produção da empresa 2 ($y_2$)", fontsize = 15)
plt.scatter(equilibrio_cournot1[0], equilibrio_cournot1[1], s = 7*4**2,
            label = "Equilíbrio de Cournot", color = "black")
plt.annotate(f"$(y_1^*, y_2^*) = ({c/3*d: .2f}, {c/3*d: .2f})$",
             ((c/3*d)+15,(c/3*d)+30), fontsize = 13)
plt.legend()
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.

```
preco2 = []
quantidade2 = []
n_firmas = 100

for i in range(1, n_firmas + 1): # para cada numero de firmas, calcular as
                                 # a quantidade e preço de equilíbrio da
                                 # economia.

    equilibrio_cournot2 = equilibrio_cournot(i)
    quantidade2.append(sum(equilibrio_cournot2))
    preco2.append(demanda_inversa(equilibrio_cournot2))

# Gráfico para a análise da entrada de firmas:

plt.plot(np.linspace(1, n_firmas, n_firmas), quantidade2, color="blue",
         linewidth=2.0, ls = "dotted", label = "Quantidade")
plt.plot(np.linspace(1, n_firmas+1, n_firmas), preco2, color="green",
         linewidth=2.0, ls = "dashed", label = "Preço")
plt.xlabel(r"Número de firmas", fontsize = 15)
plt.legend()
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.

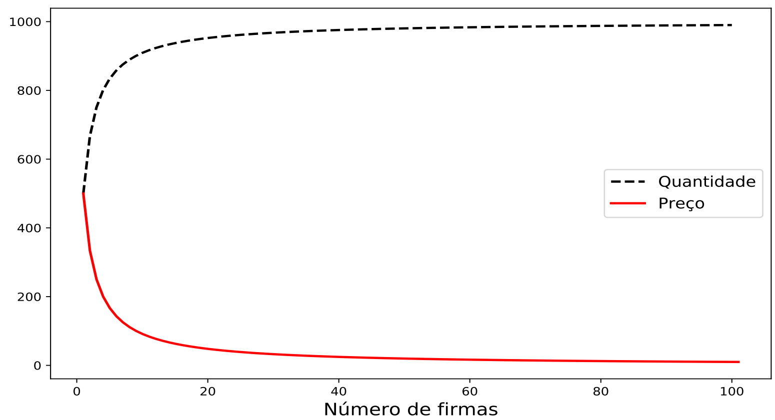

Figura 9.2: *Efeito do aumento no número de firmas nos níveis de produção e preço.*

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 $k \in \{x, y\}$,

$$k_A + k_B = \omega_A^k + \omega_B^k. \tag{10.1}$$

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

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

$$= p_x z_x(p) + p_y z_y(p) = 0. \tag{10.2}$$

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 numerário (*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.

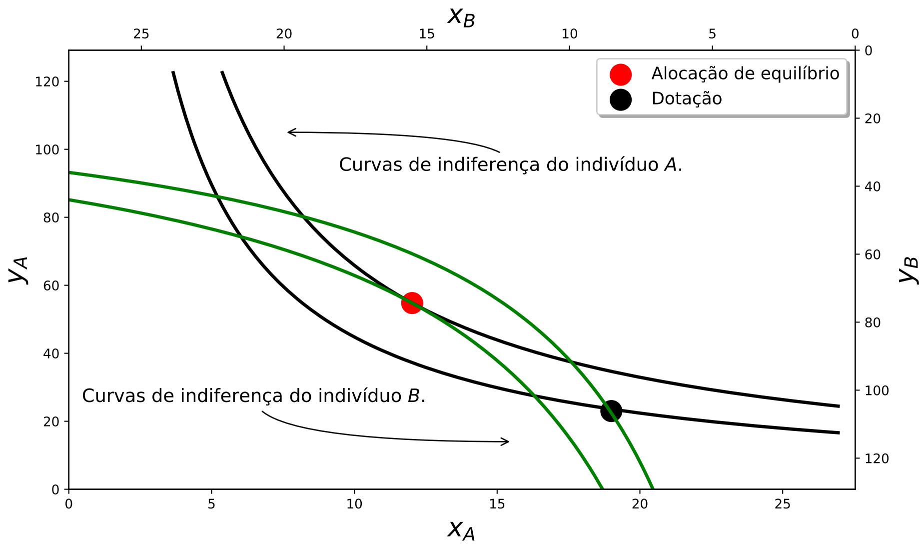

Figura 10.1: *Caixa de Edgeworth.*

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

[1] De 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 piore.

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:

```
def utilidade(x, y, alpha = 0.5):
    """
    Função de utilidade Cobb-Douglas dos agentes (ambos terão a mesma).
    """

    return (x**alpha)*(y**(1-alpha))

def demanda(p, dot, alpha = 0.5):
    """
    Resolve o problema do consumidor de forma analítica, estabelecendo a
    demanda pelos bens x e y. Como as funções de utilidade são as mesmas para
    ambos os agentes, a demanda será igual para valores arbitrários da dotação
    inicial. O parâmetro 'dot' é uma lista em que a primeira entrada
    corresponde à dotação inicial do bem x; e a segunda para o bem y. O
    parâmetro preço é uma lista em que a primeira entrada corresponde ao preço
    do bem x e a segunda ao preço do bem y.
    """

    x = alpha*(p[0]*dot[0] + p[1]*dot[1]) / p[0]
    y = (1 - alpha)*(p[0]*dot[0] + p[1]*dot[1]) / p[1]

    return x, y

def demanda_liq(p, dot):
    """
    Calcula o excesso de demanda para os bens x e y, isto é, a diferença entre
    a demanda total e a dotação inicial dos agentes. O parâmetro p é uma lista
```

```
    com os preços de x e y, respect., enquanto que dot é uma lista com as
    dotações iniciais dos bens x e y, respect..
    """

    x, y = demanda(p, dot)
    demanda_liq_x = x - dot[0]
    demanda_liq_y = y - dot[1]

    return demanda_liq_x, demanda_liq_y

def demanda_liq_agreg(p, dot_A, dot_B):
    """
    Calcula a demanda líquida agregada para os bens x e y. A demanda líquida
    agregada nada mais é que a soma das demandas líquidas individuais.
    """

    demanda_liq_agreg_x = demanda_liq(p, dot_A)[0] + demanda_liq(p, dot_B)[0]
    demanda_liq_agreg_y = demanda_liq(p, dot_A)[1] + demanda_liq(p, dot_B)[1]

    return demanda_liq_agreg_x, 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, dot, α = 0.5)`. As duas funções que aparecem no código em seguida, dadas por `demanda_liq_agreg(p, dot_A, 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(dot_A, 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.

```
def equilibrio(dot_A, dot_B):
    """
    Calcula o equilíbrio geral para um mercado competitivo de dois bens e dois
```

```
    indivíduos. Seguindo o que diz a Lei de Walras, o preço do bem x pode ser
    fixado em 1 (bem numerário) enquanto que achamos o preço do bem 2 que zera
    o excesso de demanda agregado do bem 2.
    """

    # Procurando o zero da demanda agregada excedente para o bem y
    # pelo método da bissecção:

    global px

    px = 1
    tol = 1e-10
    p_alto = 1000
    p_baixo = 0

    # Analisando a fronteira:
    tentativa1 = demanda_liq_agreg([px, p_alto], dot_A, dot_B)[1]
    if abs(tentativa1) < tol:
        return p_alto

    # atualiza preco pela média
    preco = p_alto / 2
    excesso_demanda_agreg = demanda_liq_agreg([px, preco], dot_A, dot_B)[1]

    while abs(excesso_demanda_agreg) > tol:

        if excesso_demanda_agreg > 0:

            p_baixo = preco

        else:

            p_alto = preco

        preco = (p_baixo + p_alto) / 2
        excesso_demanda_agreg = demanda_liq_agreg([px, preco], dot_A, dot_B)[1]

    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 bisseção, 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 subconjunto da curva de contrato da economia:

```
def curva_contrato():
    """
    Calcula um subconjunto da curva de contrato variando as dotações.
    """

    precos_equilibrio = [] # Lista para adicionar os preços de equilíbrio
                           # conforme variamos as dotações.

    y_aux = np.linspace(0, total_dot_y, 5)
    # Não estou "varrendo" toda a caixa de Edgeworth para calcular as
    # demandas de equilíbrio. Passo apenas inteiramente por um eixo (eixo
    # x, por ex) e vario alguns pontos do eixo y. Por este motivo, a curva
    # resultante é um subconjunto da curva de contrato.

    for j in y_aux:

        for i in x:

            precos_equilibrio.append(equilibrio([i, j], [total_dot_x - i,
                                                          total_dot_y - j]))

    x_estrela = []
    y_estrela = []

    for k in precos_equilibrio:

        for m in y_aux:

            for l in x:

                x_estrela.append(demanda([1, k], [l, m])[0])
                y_estrela.append(demanda([1, k], [l, m])[1])

    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.

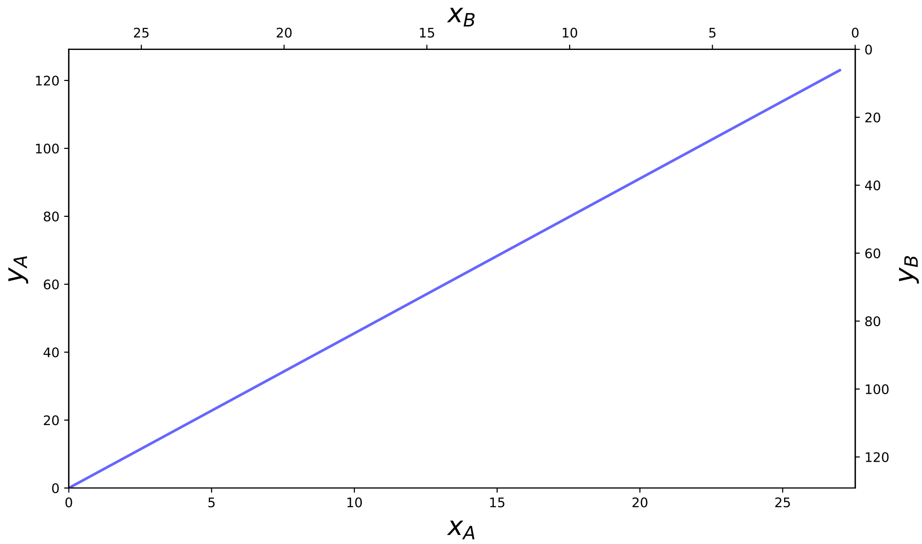

Figura 10.2: *Curva de contrato.*

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:

```
        ################################################################
        #                    Programa principal                       #
        ################################################################

#dot_A = [100, 25] # dotação indivíduo A
#dot_B = [50, 150] # dotação indivíduo B

#dot_A = [23, 33] # dotação indivíduo A
#dot_B = [14, 32] # dotação indivíduo B

dot_A = [230, 100] # dotação indivíduo A
dot_B = [400, 90] # dotação indivíduo B

preco_equilibrio_y = equilibrio(dot_A, dot_B)
p_equilibrio = [px, preco_equilibrio_y]

#Calcula as demandas de equilíbrio para os indivíduos A e B, respect.:
demanda_eq_A = demanda(p_equilibrio, dot_A)
demanda_eq_B = demanda(p_equilibrio, dot_B)

# Utilidade dos agentes em equilíbrio:
u_valor_A = utilidade(demanda_eq_A[0], demanda_eq_A[1])
```

```
u_valor_B = utilidade(demanda_eq_B[0], demanda_eq_B[1])


total_dot_x = dot_A[0] + dot_B[0] # pode-se considerar o tamanho do eixo
                                  # horizontal da caixa de Edgeworth
total_dot_y = dot_A[1] + dot_B[1] # pode-se considerar o tamanho do eixo
                                  # vertical da caixa de Edgeworth

x = np.linspace(0.0001, total_dot_x, 100) # grid da quantidade do bem x
y = np.linspace(0.0001, total_dot_y, 100) # grid da quantidade do bem y

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).

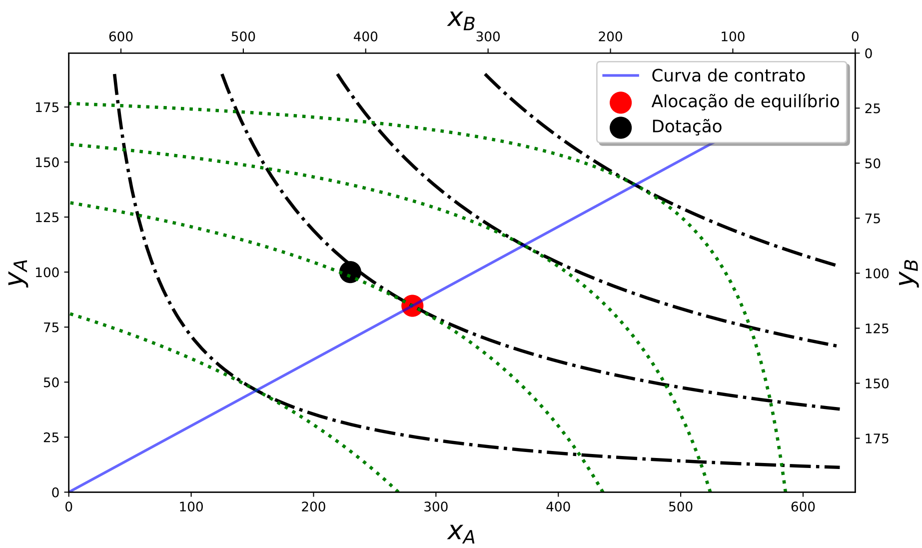

Figura 10.3: *Caixa de Edgeworth para um exemplo de economia.*

Este gráfico gerado pode ser implementado utilizando o seguinte código:

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

```

```python
fig, ax = plt.subplots(tight_layout = True)

# Cria o produto cartesiano das quantidades x_1 e x_2.
xc_1, xc_2 = np.meshgrid(x, y)

# Coloca label nos eixos originais
ax.set_xlabel('$x_A$', fontsize = 20)
ax.set_ylabel('$y_A$', fontsize = 20)

ax.axis([0, 1.02*total_dot_x, 0, 1.05*total_dot_y])

# Calcula a utilidade do consumidor A
uA = utilidade(xc_1, xc_2) # Calcula a matriz de utilidades do
                           # consumidor A.

# Calcula a utilidade do consumidor B. Como queremos que o gráfico tenha origem
# em x = x0 e y = y0, fazemos o ajuste nos argumentos da função.
uB = utilidade(total_dot_x - xc_1, total_dot_y - xc_2) # Calcula a matriz de
                                                       # utilidades do
                                                       # consumidor B.

# Criamos um novo eixo y (apenas para repetir os ticks e labels do eixo y
#                          original)
ax2 = ax.twinx()

# Copiamos os limites do eixo y original
orig_ylim = ax.get_ylim()

# Replicamos os limites do eixo y original no novo eixo
ax2.set_ylim(orig_ylim)

# Definimos um label para o novo eixo y
ax2.set_ylabel('$y_B$', fontsize = 20)

# Invertemos o sentido do texto do eixo y original
ax2.invert_yaxis()

# Criamos um novo eixo x (apenas para repetir os ticks e labels do eixo x
#                          original)
ax3 = ax.twiny()

# Copiamos os limites do eixo x original
orig_xlim = ax.get_xlim()

# Replicamos os limites do eixo x original no novo eixo
ax3.set_xlim(orig_xlim)
```

```python

# Definimos um label para o novo eixo x
ax3.set_xlabel('$x_B$', fontsize = 20)

# Invertemos o sentido do texto do eixo x original
ax3.invert_xaxis()

# Curvas de indiferença no nível indicado
CS1 = ax.contour(xc_1, xc_2, uA, levels = [u_valor_A - 70, u_valor_A,
                                           u_valor_A + 50, u_valor_A + 100],
                 linestyles = 'dashdot', colors = 'k', alpha = 1,
                 extend = 'both')
CS2 = ax.contour(xc_1, xc_2, uB, levels = [u_valor_B - 100, u_valor_B - 50,
                                           u_valor_B, u_valor_B + 70],
                 linestyles = 'dotted', colors = 'g', alpha = 1,
                 extend = 'both')

# Curva de contrato:
ax.plot(contrato[0], contrato[1], 'blue', linewidth = 2, alpha = 0.6,
        label = 'Curva de contrato')

#Alocação de equilíbrio:
ax.scatter(demanda_eq_A[0], demanda_eq_A[1], s = 10*5**2,
           label = "Alocação de equilíbrio", alpha = 1, c = "r")
#ax.plot(demanda_eq_A[0], demanda_eq_A[1], 'r', marker = ".", markersize = 25,
#        label = "Alocação de equilíbrio")

# Dotação inicial:
ax.scatter(dot_A[0], dot_A[1], s = 10*5**2, label = "Dotação", alpha = 1,
           c = "black")
#ax.plot(dot_A[0], dot_A[1], 'black', marker = ".", markersize = 25,
#        label = "Dotação")
ax.legend(loc = "upper right", fancybox = True, shadow = True, fontsize = 13)
plt.show()
```

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.

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

```python
# -*- coding: utf-8 -*-
"""
@author: Matheus L. Carrijo
"""

import matplotlib.pyplot as plt #importing graph package
plt.figure(figsize=(9, 6), dpi=100) #set default figure size in a better way
import numpy as np #importing numpy

def demanda(p):
    assert a < 0 and b > 0
    return a*p + b

def oferta(p):
    assert c > 0 and d > 0 and b > d
    # d in (0,b) é suficiente pra equilibrio
    return c*p + d

def excesso_demanda(p):
   # Setting excess demand function. It must be clear that equlibrium requires
   # excesso_demanda(p) = 0, for some especific p
```

```python
    return demanda(p) - oferta(p)

parametro_bool = False

teste = "diferente de y e n"
while teste != "y" and teste != "n":
    print("\nVocê quer entrar com os parâmetros das funções oferta", end = "")
    teste = input("e demanda? [y/n]: ")

if teste == 'y':
    parametro_bool = True

if parametro_bool:

    print("\nDadas as funções lineares de oferta e demanda,", end = " ")
    print("queremos calcular o preço e a quantidade de equilíbrio")
    a = float(input("Digite o coeficiente de inclinação da demanda: "))
    b = float(input("Digite a constante da curva de demanda: "))
    c = float(input("Digite o coeficiente de inclinação da oferta: "))
    d = float(input("Digite a constante da curva de oferta: "))

    p_grid = np.linspace(-1000, 1000, 100000)
    q_grid = np.linspace(-1000, 1000, 100000)

    E_abs = []
    for i in p_grid:
        E_abs.append(abs(excesso_demanda(i)))

    indice_min = E_abs.index(min(E_abs))
    p_equilibrio = p_grid[indice_min]
    q_equilibrio = oferta(p_equilibrio)

    print("O preço e a quantidade de equilíbrio são", end = "")
    print(f", respectivamente,{p_equilibrio: .2f} e {q_equilibrio: .2f}")

    q1, q2, q3 = demanda(p_grid), oferta(p_grid), excesso_demanda(p_grid)
    plt.plot(q1, p_grid, color="blue", linewidth=1.0, ls = "dashed")
    plt.plot(q2, p_grid, color="green", linewidth=1.0)
    plt.plot(q3, p_grid, color="red", linewidth=1.0, ls = "dotted")
    plt.xlim(d, b) # entre -5 e 5 por exemplo
    plt.ylim(-d/c, (-b) / a)
    plt.legend([f'Curva de Demanda: D = {a}p + {b}',
                f'Curva de Oferta: S = {c}p + {d}',
                'Curva do excesso de demanda: E = D - S'])
    plt.ylabel("Preço", fontsize = 15)
    plt.xlabel("Quantidade", fontsize = 15)
```

```python
    plt.show()

else:
    print("\nFaremos o cálculo para a função demanda D = -2p + 7", end = " ")
    print("e oferta S = 4p + 1\n")
    a, b, c, d = -2, 7, 4, 1

    p_grid = np.linspace(-1000, 1000, 1000)
    q_grid = np.linspace(-1000, 1000, 1000)

    E_abs = []
    for i in p_grid:
        E_abs.append(abs(excesso_demanda(i)))

    indice_min = E_abs.index(min(E_abs))
    p_equilibrio = p_grid[indice_min]
    q_equilibrio = oferta(p_equilibrio)

    print("O preço e a quantidade de equilíbrio são", end = "")
    print(f", respectivamente,{p_equilibrio: .2f} e {q_equilibrio: .2f}")

    q1, q2, q3 = demanda(p_grid), oferta(p_grid), excesso_demanda(p_grid)
    plt.plot(q1, p_grid, color="blue", linewidth=3.0 , ls = "dashed")
    plt.plot(q2, p_grid, color="green", linewidth=3.0)
    plt.plot(q3, p_grid, color="red", linewidth=3.0, ls = "dotted")
    plt.xlim(d, b) # entre -5 e 5 por exemplo
    plt.ylim(-d/c, (-b) / a)
    plt.legend([f'Curva de Demanda: D = {a}p + {b}',
                f'Curva de Oferta: S = {c}p + {d}',
                'Curva do excesso de demanda: E = D - S'])
    plt.ylabel("Preço", fontsize = 15)
    plt.xlabel("Quantidade", fontsize = 15)
    plt.show()
```

A.2 Crescimento Econômico: Modelo de Solow

```python
# -*- coding: utf-8 -*-
"""
@author: Matheus L. Carrijo
"""

import matplotlib.pyplot as plt #importing graph package
```

```python
plt.figure(figsize=(9, 6), dpi=100) #set default figure size in a better way
#plt.rcParams["figure.figsize"] = (8, 6) #set default figure size
import numpy as np #importing numpy

# Set the Cobb-Douglas production function:
def produto_agregado(k_agregado):
    return np.sqrt(k_agregado*l_agregado)

# Set the Cobb-Douglas production function per worker:
def produto_por_trabalhador(k_agregado):
    return produto_agregado(k_agregado)/l_agregado

# Depreciation is a linear function of k per worker, i.e., k=K/L
def depreciacao(k_agregado):
    return taxa_depreciacao*(k_agregado/l_agregado)

def investimento(k_agregado):
    return taxa_poupanca*produto_por_trabalhador(k_agregado)

def consumo(k_agregado):
    return (1 - taxa_poupanca)*produto_por_trabalhador(k_agregado)

def k_variacao(k_agregado):
    return (taxa_poupanca*produto_por_trabalhador(k_agregado)
            - depreciacao(k_agregado))

resposta = "qualquer_coisa"
while resposta != "y" and resposta != "n":
    print("Deseja entrar com os parâmetros? [y/n]", end = " ")
    resposta = input(" ")

if resposta == "y":
    taxa_depreciacao, taxa_poupanca = 2, 2 # just to enter the while

    # The rates must be between 0 and 1. The following loop guarantee this.
    while (taxa_depreciacao < 0 or taxa_depreciacao > 1
           or taxa_depreciacao == 0):
        print("A taxa de depreciação não nula deve estar entre", end = " ")
        print("0 e 1", end = " ")
        taxa_depreciacao = float(input("Entre com a taxa de depreciação: "))

    while taxa_poupanca < 0 or taxa_poupanca > 1 or taxa_poupanca == 0:
        print("A taxa de poupança não nula deve estar entre 0 e 1.", end = ' ')
        taxa_poupanca = float(input("Entre com a taxa de poupança: "))


```

```
    print("Entre com o nível de mão de obra desta economia: ", end = " ")
    l_agregado = float(input(" "))

    print("Entre com o nível inicial de capital agregado desta economia: ",
          end = " ")
    k_t0 = float(input(" "))/l_agregado

    tolerancia = 0.001 # ask the user for enter this parameter?
    norma = 10 #just to enter in the while
    while norma > tolerancia:
        k_t1 = (taxa_poupanca*produto_por_trabalhador(k_t0) +
                k_t0*(1-taxa_depreciacao))
        norma = abs(k_t1 - k_t0)
        k_t0 = k_t1

    print(f"O produto de equilibrio é k* = {k_t1: .4}")

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

elif resposta == "n":
    print("Faremos uma exemplo em que a taxa de poupança é 30%,", end = " ")
    print("a taxa de depreciação é 10%, e que a economia comece", end = " ")
    print("com uma relação de 4 capital por trabalhador")

    taxa_depreciacao, taxa_poupanca, l_agregado, k_t0 = 0.1, 0.3, 1, 4

    tolerancia = 0.001 # ask the user for enter this parameter?
    norma = 10 #just to enter in the while
    while norma > tolerancia:
        k_t1 = (taxa_poupanca*produto_por_trabalhador(k_t0) +
                k_t0*(1-taxa_depreciacao))
        norma = abs(k_t1 - k_t0)
        k_t0 = k_t1

    print(f"O produto de equilibrio é k* = {k_t1: .4}")
```

```

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

A.3 Renda Nacional: Modelo IS-LM

```
# -*- coding: utf-8 -*-
"""
@author: Matheus L. Carrijo
"""

import matplotlib.pyplot as plt #importing graph package
plt.figure(figsize=(9, 6), dpi=100) #set default figure size
import numpy as np

def investimento(taxa_juros):

    """
    Dada a taxa de juros, calcula o nível de investimento. Deve ser
    negativamente relacionada com a taxa de juros. Por simplicidade,
    consideramos o caso linear.
    """

    if a < 0 and b > 0:
        return a*taxa_juros + b

def demanda_monetaria(taxa_juros, produto):

    """
    Modelamos a demanda monetária tentando incorporar linearmente os motivos
    precaução, especulação, e transação, respectivamente pelas variáveis
    e, c, d.
```

```python
    """

    return c*produto + d*taxa_juros + e

def consumo(produto):

    """
    Consideraremos o consumo como uma função linear da renda disponível, de
    modo que o coeficiente de inclinação é a propensão marginal a consumir e
    a constante é o consumo autônomo.
    """

    return (propensao_marginal_a_consumir * (produto - aliquota_imposto)
            + consumo_autonomo)

def produto_IS(taxa_juros):

    """
    Construímos a curva IS através da equação Y = C + I + G, em que C é a
    função consumo (que depende da renda disponível), I é o investimento,
    dependente da taxa de juros, e G são os gastos do governo. Como o consumo
    depende do nível de renda, esta equação é manipulada até que tenhamos uma
    expressão do produto em relação à taxa de juros
    """

    I = investimento(taxa_juros)

    return ((consumo_autonomo + I + gastos_governo) /
            (1 - propensao_marginal_a_consumir
            * (1 - aliquota_imposto)))


def produto_LM(taxa_juros):

    """
    Construímos a curva LM através da equação
    oferta_monetaria = demanda_monetaria = c*produto + d*taxa_juros + e,
    em que a oferta é exógena e a demanda calculada pela função. Então, a
    expressão pode ser manipulada até que tenhamos uma função de Y em função
    da taxa de juros.
    """

    return ((oferta_monetaria/nivel_precos - e) / c - taxa_juros * (d / c))

def excesso_ISLM(taxa_juros):

```

```
    """
    Função para calcular o equilíbrio fazendo excesso_ISLM = 0, isto é,
    o par (r*, Y*) ótimo que nos dá a intersecção entre as curvas IS e LM.
    """

    return produto_IS(taxa_juros) - produto_LM(taxa_juros)

resposta = "qualquer_coisa"

while resposta != "y" and resposta != "n":

    print("Deseja entrar com os parâmetros? [y/n]", end = " ")
    resposta = input(" ")

if resposta == "n":

    a, b, propensao_marginal_a_consumir, consumo_autonomo = -20, 5, 0.3, 100
    gastos_governo, aliquota_imposto, c, d, e = 50, .2, 10, -300, 50
    nivel_precos, oferta_monetaria = 10, 100

    print("\nFaremos uma exemplo em que a função investimento é", end = " ")
    print(f"dada por {a}r + {b};", end = " ")
    print("a função demanda monetária, por", end = " ")
    print(f"{c}Y + {d}i + {e};", end = " ")
    print("a função consumo, por", end = " ")
    print(f"{propensao_marginal_a_consumir}Y + {consumo_autonomo};", end = " ")
    print(f"e a política fiscal por G = {gastos_governo} e", end = " ")
    print(f"T = {aliquota_imposto}Y.", end = " ")
    print("Ainda, a oferta monetária real é dada por M/P =", end = " ")
    print(f"{oferta_monetaria/nivel_precos}.")


    # Calculo do equilibrio

    taxa_juros_grid = np.linspace(0, 1000, 100000)

    Excesso_abs = []

    for i in taxa_juros_grid:

        Excesso_abs.append(abs(excesso_ISLM(i)))

    indice_min = Excesso_abs.index(min(Excesso_abs))
    juros_equilibrio = taxa_juros_grid[indice_min]
    produto_equilibrio = produto_IS(juros_equilibrio)

```

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

    valores_produto = []

    for i in taxa_juros_grid:

        valores_produto.append(abs(produto_IS(i)))

    y = taxa_juros_grid[np.argmin(valores_produto)]

    # graph settings

    plt.plot(produto_IS(taxa_juros_grid), taxa_juros_grid, color="blue",
             linewidth=3.0, ls = "dotted")
    plt.plot(produto_LM(taxa_juros_grid), taxa_juros_grid, color="green",
             linewidth=3.0, ls = "dashed")
    plt.ylim(0, y)
    plt.xlim(min(produto_LM(taxa_juros_grid)),
             max(produto_IS(taxa_juros_grid)))
    plt.legend(['Curva IS', 'Curva LM'])
    # plt.title("Modelo IS-LM", fontsize = 20)
    plt.ylabel("Taxa de juros", fontsize = 15)
    plt.xlabel("Produto, Y", fontsize = 15)
    plt.show()

elif resposta == "y":

    # Entrando com os parâmetros da função investimento:

    a, b = 1, -1 # Valor para entrar no loop

    while a >= 0:

        print("\nO coeficiente de inclinação da função", end = " ")
        print("investimento deve ser negativo.")
        print("Entre com o coeficiente de inclinação da função", end = " ")
        a = float(input("linear de investimento: "))

    while b < 0:

        print("\nA constante da função investimento deve ser", end = " ")
        print("positiva.\n")
        print("Entre com a constante da função linear de", end = " ")
        b = float(input("investimento: "))

```

```
    # Entrando com os parâmetros da função demanda monetária:

    c, d, e = -1, 1, -1 # Valor para entrar no loop

    while c < 0:

        print("\nO coeficiente de transação da demanda moentária", end = " ")
        print("deve ser positivo.")
        print("Entre com o coeficiente de transação da demanda", end = " ")
        c = float(input("monetária: "))

    while d > 0:

        print("\nO coeficiente de especulação da demanda monetária", end = " ")
        print("deve ser negativo.")
        print("Entre com o coeficiente de especulação da demanda", end = " ")
        d = float(input("monetária: "))

    while e < 0:

        print("\nO coeficiente de precaução da demanda monetária", end = " ")
        print("deve ser positivo.")
        print("Entre com o coeficiente de precaução da demanda", end = " ")
        e = float(input("monetária: "))

    # Entrando com a oferta monetária real:

    nivel_precos, oferta_monetaria = -1, -1 # Valor para entrar no loop

    while nivel_precos <= 0:

        print("\nO nível de preços deve ser positivo.")
        print("Entre com o nível de preços:", end = " ")
        nivel_precos = float(input(""))

    while oferta_monetaria <= 0:

        print("\nA oferta monetária deve ser positiva.")
        print("Entre com a oferta monetária:", end = " ")
        oferta_monetaria = float(input(""))

    # Entrando com os parâmetros da função consumo:

    propensao_marginal_a_consumir, consumo_autonomo = -1, -1

    while (propensao_marginal_a_consumir < 0 or
```

```
           propensao_marginal_a_consumir > 1):

        print("\nA propensão marginal a consumir deve estar entre", end = " ")
        print("0 e 1.")
        print("Entre com a propensão marginal a consumir: ", end = " ")
        propensao_marginal_a_consumir = float(input(""))

    while consumo_autonomo < 0:

        print("\nO consumo autônomo deve ser positivo.")
        print("Entre com o consumo autônomo: ", end = " ")
        consumo_autonomo = float(input(""))

    # Entrando com a política fiscal:

    gastos_governo, aliquota_imposto = -1, -1 # Valor para entrar no loop

    while gastos_governo < 0:

        print("\nOs gastos do governo devem ser não negativo.")
        print("Entre com os gastos do governo: ", end = " ")
        gastos_governo = float(input(""))

    while aliquota_imposto < 0 or aliquota_imposto > 1:

        print("\nA alíquota de imposto deve estar entre 0 e 1.")
        print("Entre com os impostos cobrados pelo governo: ", end = " ")
        aliquota_imposto = float(input(""))

    taxa_juros_grid = np.linspace(0, 1000, 100000)

    Excesso_abs = []

    for i in taxa_juros_grid:

        Excesso_abs.append(abs(excesso_ISLM(i)))

    indice_min = Excesso_abs.index(min(Excesso_abs))
    juros_equilibrio = taxa_juros_grid[indice_min]
    produto_equilibrio = produto_IS(juros_equilibrio)

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

    valores_produto = []

```

```python
    for i in taxa_juros_grid:

        valores_produto.append(abs(produto_IS(i)))

    y = taxa_juros_grid[np.argmin(valores_produto)]

    # graph settings

    plt.plot(produto_IS(taxa_juros_grid), taxa_juros_grid, color="blue",
             linewidth=3.0, ls = "dotted")
    plt.plot(produto_LM(taxa_juros_grid), taxa_juros_grid, color="green",
             linewidth=3.0, ls = "dashed")
    plt.ylim(0, y)
    plt.xlim(min(produto_LM(taxa_juros_grid)),
             max(produto_IS(taxa_juros_grid)))
    plt.legend(['Curva IS', 'Curva LM'])
    #plt.title("Modelo IS-LM", fontsize = 20)
    plt.ylabel("Taxa de juros", fontsize = 15)
    plt.xlabel("Produto, Y", fontsize = 15)
    plt.show()
```

A.4 Desemprego: *Lake Model*

```python
# -*- coding: utf-8 -*-
"""
@author: Matheus L. Carrijo
"""

        ################################################################
        #                     Importando pacotes                       #
        ################################################################

import matplotlib.pyplot as plt #importing graph package
plt.figure(figsize=(9, 6), dpi=1000) #set default figure size
import numpy as np

        ################################################################
        #                          Funções                             #
        ################################################################

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

```python

    #Definindo as variáveis com o valor dos parâmetros

    forca_trabalho = empregados + desempregados

    A = np.array([[1 - Lambda, Alpha    ],
                  [Lambda     , 1 - Alpha]])
    x_inicial = np.array([[desempregados/forca_trabalho],
                          [empregados/forca_trabalho]])

    tempo = []
    eixo_taxa_desemprego = [x_inicial[0][0]]
    eixo_taxa_emprego = [x_inicial[1][0]]
    t = 0
    tol = 1e-04
    dif = tol + 1

    # Calculando o nível estacionário

    while dif > tol:

        x = A @ x_inicial
        tempo.append(t)
        eixo_taxa_desemprego.append(x[0][0])
        eixo_taxa_emprego.append(x[1][0])
        dif = np.max(np.abs(x - x_inicial))
        x_inicial = x
        t += 1
    tempo.append(t)

    # Informando ao usuário

    print(f"\nPara uma economia com {empregados} empregados,", end = " ")
    print(f"{desempregados} desempregados, uma taxa de obtenção de", end = " ")
    print(f"emprego de {100*Lambda: .2f}%, e uma taxa de demissão", end = " ")
    print(f"de {100*Alpha: .2f}%, a taxa de desemprego no nível", end = " ")
    print(f"estacionário é dada por {100*x[0][0]: .2f}% \n")

    return x, tempo, eixo_taxa_desemprego, eixo_taxa_emprego


def individual_path(Lambda = 0.3, Alpha = .027, desempregado = True):
    """

    Parameters
    ----------
```

```
    Lambda : float, optional
        Taxa de obtenção de emprego. The default is 0.3.
    Alpha : float, optional
        Taxa de demissão. The default is .027.
    desempregado : bool, optional
        informa se os indivíduos estão desempregados. The default is True.

    Returns
    -------
    tempo : list
        lista em que a i-ésima entrada corresponde ao tempo que o i-esimo
        trabalhador demorou pra encontrar emprego (caso esteja desempregado)
        ou que demorou pra ficar desempregado (caso esteja empregado).

    """

    if desempregado:

        tempo = []

        j = 1
        while j <= 1000: # amostra de 1000 individuos

            encontrou_emprego = False
            t = 0
            while not encontrou_emprego:
            # enquanto a pessoa j não encontrar emprego, ela continua
            # procurando

                r = np.random.rand()

                if r <= Lambda: # desempregado encontra emprego

                    encontrou_emprego = True

                t += 1

            tempo.append(t)
            j += 1

    else:

        tempo = []

        j = 1
        while j <= 1000: # amostra de 1000 individuos
```

```

            ficou_desempregado = False
            t = 0
            while not ficou_desempregado: # fica no emprego até ser demitido

                r = np.random.rand()

                if r <= Alpha: # empregado ficou desempregado

                    ficou_desempregado = True
                    t += 1

                t += 1

            tempo.append(t)
            j += 1

    return tempo


        ################################################################
        #                Uma economia com:                             #
        #                - empregados = 250_000                        #
        #                - desempregados = 50_000                      #
        #                - Alpha = .027                                #
        #                - Lambda = .391                               #
        ################################################################

exemplo_padrao = nivel_estacionario()

fig, ax = plt.subplots(nrows = 2, ncols = 1, sharex  = True)

# plot desemprego:

ax[0].plot(exemplo_padrao[1], exemplo_padrao[2], color = "blue", lw = 1.5)
ax[0].set_xlim(0, exemplo_padrao[1][-1])
ax[0].set_ylabel("Taxa de desemprego")
#ax[0].legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower center',
#             ncol=2, mode="expand", borderaxespad=0.)
ax[0].set_title(r'$(\alpha, \lambda) = (2.7\%, 39.1\%)$', fontsize = 10)

# plot emprego:

ax[1].plot(exemplo_padrao[1], exemplo_padrao[3], color = "green", lw = 1.5)
ax[1].set_xlabel("tempo")
ax[1].set_ylabel("Taxa de emprego")
```

```python

        ###############################################################
        #         Variando parâmetros (estática comparativa)          #
        ###############################################################

# Variando Lambda

estatica1 = nivel_estacionario(450_000, 50_000, .04, .3)
estatica2 = nivel_estacionario(450_000, 50_000, .04, .6)

# Variando Alpha

estatica3 = nivel_estacionario(450_000, 50_000, .03, .4)
estatica4 = nivel_estacionario(450_000, 50_000, .06, .4)


fig, ax = plt.subplots(nrows = 2, ncols = 1, sharex  = True)

t_max = max(estatica1[1][-1], estatica2[1][-1], estatica3[1][-1],
            estatica4[1][-1]) # limite eixo x

# plot desemprego:

ax[0].plot(estatica1[1], estatica1[2], color = "blue", lw = 1,
           label = r'$(\alpha, \lambda) = (4\%, 30\%)$')
ax[0].plot(estatica2[1], estatica2[2], color = "green", lw = 1,
           label = r'$(\alpha, \lambda) = (4\%, 60\%)$')
ax[0].plot(estatica3[1], estatica3[2], color = "red", lw = 1,
           label = r'$(\alpha, \lambda) = (3\%, 40\%)$')
ax[0].plot(estatica4[1], estatica4[2], color = "black", lw = 1,
           label = r'$(\alpha, \lambda) = (6\%, 40\%)$')
ax[0].set_xlim(0, t_max)
ax[0].set_ylabel("Taxa de desemprego")
ax[0].legend(bbox_to_anchor=(0., 1.02, 1., .102), loc='lower center',
             ncol=2, mode="expand", borderaxespad=0.)

# plot emprego:

ax[1].plot(estatica1[1], estatica1[3], color = "blue", lw = 1)
ax[1].plot(estatica2[1], estatica2[3], color = "green", lw = 1)
ax[1].plot(estatica3[1], estatica3[3], color = "red", lw = 1)
ax[1].plot(estatica4[1], estatica4[3], color = "black", lw = 1)
ax[1].set_xlim(0, t_max)
ax[1].set_xlabel("tempo")
ax[1].set_ylabel("Taxa de emprego")
plt.show()
```

```
        ################################################################
        #             Trajetória de indivíduos com:                    #
        #                - Lambda = .391                               #
        #                - Alpha = .027                                #
        ################################################################

exemplo_padrao2 = individual_path()

aux = max(exemplo_padrao2)
hist, valores = np.histogram(exemplo_padrao2, bins = np.arange(0, aux))
plt.plot(valores[:-1], hist, color="blue", linewidth=1)
plt.ylabel("Trabalhadores", fontsize = 10)
plt.xlabel("Tempo até encontrar emprego", fontsize = 10)
#plt.title('Histograma', fontsize = 13)
plt.show()

        ################################################################
        #            Estática comparativa para desemprego              #
        ################################################################

exemplo_padrao3 = individual_path(Lambda = 0.2)
exemplo_padrao4 = individual_path(Lambda = 0.4)
exemplo_padrao5 = individual_path(Lambda = 0.65)
exemplo_padrao6 = individual_path(Lambda = 0.9)

aux = max(max(exemplo_padrao2), max(exemplo_padrao3),
          max(exemplo_padrao4), max(exemplo_padrao5))
hist2, valores2 = np.histogram(exemplo_padrao3, bins = np.arange(0, aux))
hist3, valores3 = np.histogram(exemplo_padrao4, bins = np.arange(0, aux))
hist4, valores4 = np.histogram(exemplo_padrao5, bins = np.arange(0, aux))
hist5, valores5 = np.histogram(exemplo_padrao6, bins = np.arange(0, aux))
plt.plot(valores2[:-1], hist2, color="black", linewidth=1)
plt.plot(valores3[:-1], hist3, color="green", linewidth=1)
plt.plot(valores4[:-1], hist4, color="red", linewidth=1)
plt.plot(valores5[:-1], hist5, color="orange", linewidth=1)
plt.legend([r'$\lambda = 10\%$', r'$\lambda = 40\%$', r'$\lambda = 70\%$',
            r'$\lambda = 90\%$'])
plt.ylabel("Trabalhadores", fontsize = 10)
plt.xlabel("Tempo até encontrar emprego", fontsize = 10)
#plt.title('Histogramas', fontsize = 13)
plt.show()
```

A.5 Escolha Intertemporal

```python
# -*- coding: utf-8 -*-
"""
@author: Matheus L. Carrijo
"""

        #################################################################
        #                     Importando pacotes                               #
        #################################################################

import matplotlib.pyplot as plt #importing graph package
plt.figure(figsize=(9, 6), dpi=1000) #set default figure size
import numpy as np

        #################################################################
        #                          Funções                                     #
        #################################################################

def u(c, a):
    if a > 0 and a != 1:
        return (c**(1-a) - 1) / (1 - a)
    else:
        if a == 0:
            return np.log(c)

def U(c1, c2, a, b):
    return u(c1, a) + u(c2, b)

def budget(c1, m1, m2, r):
    if m1 > 0 and m2 > 0 and r > 0 and r < 1:
        return (1+r)*m1 + m2 - (1+r)*c1

def inverse_budget(c2, m1, m2, r):
    if m1 > 0 and m2 > 0 and r > 0 and r < 1:
        return (m2-c2)/(1+r) + m1

def choice(a, b, r, m1, m2):
    if (a > 0 and a != 1 and b > 0 and b != 1 and m1 > 0 and m2 > 0 and r > 0
        and r < 1):
        c1_estrela = (m1+m2/(1+r)) / (1+(1+r)**((1-a)/a))
        c2_estrela = (m1 - c1_estrela)*(1+r) + m2
        funcao_valor = U(c1_estrela, c2_estrela, a, b)

```

```python
    return c1_estrela, c2_estrela, funcao_valor

        ################################################################
        #                          Exemplo                                    #
        ################################################################

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

        ################################################################
        #                   Informando ao usuário                             #
        ################################################################

print(f"\nCom uma taxa de juros de {100*r}%, renda em t1 de {m1}", end = " ")
print(f"e renda {m2} em t2, a distribuição de consumo no tempo que", end = " ")
print("maximiza a utilidade do agente é dada por", end = " ")
print(f'(c1, c2) = ({exemplo1[0]}, {exemplo1[1]}), com nível de', end = " ")
print(f"utilidade dado por U = {exemplo1[2]}")

        ################################################################
        #                          Gráfico                                   #
        ################################################################

eixo_x, eixo_y = np.linspace(0, 1000, 1000), np.linspace(0, 1000, 1000)
budget1 = budget(eixo_x, m1, m2, r)
X, Y = np.meshgrid(eixo_x, eixo_y)

Z = plt.contour(X, Y, U(X, Y, a, b),
                levels = [exemplo1[2]-10, exemplo1[2], exemplo1[2]+10],
                linewidths = 2)
plt.plot(eixo_x, budget1, color = "black", lw = 2,
         label = "restrição orçamentária")
plt.clabel(Z, inline = 1, fontsize = 10)
plt.ylim(0, (1+r)*m1 + m2)
plt.xlim(0, m1 + m2/(1+r))
plt.ylabel(r"Consumo em $t_2$", fontsize = 15)
plt.xlabel(r"Consumo em $t_1$", fontsize = 15)
plt.scatter([exemplo1[0]], [exemplo1[1]], s = 7*4**2)
plt.annotate(f"$(c_1^*, c_2^*) = ({exemplo1[0]: .0f}, {exemplo1[1]: .0f})$",
             (exemplo1[0]+10, exemplo1[1]+10), fontsize = 13)
plt.scatter([m1],[m2], s = 7*4**2,
            label = f"dotação inicial $(m_1, m_2) = ({m1},{m2})$")
plt.annotate("A", (m1-5,m2-55), fontsize = 13)
plt.legend()
plt.show()
```

```
        ################################################################
        #                    Estática Comparativa                      #
        #                (alterando a taxa de juros)                   #
        ################################################################

# Outros exemplos
a2, b2, r2, m1_2, m2_2 = 0.5, 0.5, 0.1, 50, 750
a3, b3, r3, m1_3, m2_3 = 0.5, 0.5, 0.9, 50, 750
exemplo2 = choice(a2, b2, r2, m1_2, m2_2)
exemplo3 = choice(a3, b3, r3, m1_3, m2_3)

# Saída para o usuário
print(f"\nDiminuindo a taxa de juros para {100*r2}%, temos", end = " ")
print("que a nova distribuição de consumo no tempo que maximiza a", end = " ")
print("utilidade do agente é dada por", end = " ")
print(f'(c1*, c2*) = ({exemplo2[0]}, {exemplo2[1]}), com nível de', end = " ")
print(f"utilidade dado por U = {exemplo2[2]}. Se aumentarmos a", end = " ")
print(f"taxa de juros para {100*r3}%, a distribuição de consumo no", end = " ")
print("tempo é dada por (c1*, c2*) =", end = " ")
print(f"({exemplo3[0]}, {exemplo3[1]}), com nível de utilidade", end = " ")
print(f"U = {exemplo3[2]}. Podemos conferir estes resultados graficamente.")

# configuração exemplo1
budget1 = budget(eixo_x, m1, m2, r)

# configuração exemplo2
budget2 = budget(eixo_x, m1_2, m2_2, r2)

# configuração exemplo3
budget3 = budget(eixo_x, m1_3, m2_3, r3)


# plotando exemplo1 (mesmo que anterior)
Z1 = plt.contour(X, Y, U(X, Y, a, b), levels = [exemplo3[2], exemplo1[2],
                                                exemplo2[2]], linewidths = 2)
plt.plot(eixo_x, budget1, color = "black", ls = "dashed", lw = 2,
         label = f"restrição orçamentária com $r = {100*r}\%$")

# plotando exemplo2
plt.plot(eixo_x, budget2, color = "blue", lw = 2,
         label = f"restrição orçamentária com $r = {100*r2}\%$")

# plotando exemplo3
plt.plot(eixo_x, budget3, color = "red", lw = 2,
         label = f"restrição orçamentária com $r = {100*r3}\%$")

```

```
# Marcando pontos na tangência da reta orçamentária com as curvas de utilidade
plt.scatter([exemplo1[0]], [exemplo1[1]], s = 4*4**2)
plt.scatter([exemplo2[0]], [exemplo2[1]], s = 4*4**2)
plt.scatter([exemplo3[0]], [exemplo3[1]], s = 4*4**2)
plt.scatter([m1],[m2], s = 7*4**2,
            label = f"dotação inicial $(m_1, m_2) = ({m1},{m2})$")

plt.annotate("A", (m1-5,m2-55), fontsize = 12)

# configuração geral plot
plt.legend(loc = 'lower left', bbox_to_anchor=(0.1, 0.1,),
           fancybox = True, shadow = True)
plt.ylim(0, (1+r)*m1 + m2)
plt.xlim(0, m1_3 + m2_3/(1+r3))
plt.ylabel(r"Consumo em $t_2$", fontsize = 12)
plt.xlabel(r"Consumo em $t_1$", fontsize = 12)
plt.show()
```

A.6 Escolha sob Incerteza

```
# -*- coding: utf-8 -*-
"""
@author: Matheus L. Carrijo
"""

        ################################################################
        #                     Importando pacotes                       #
        ################################################################

import matplotlib.pyplot as plt #importing graph package
plt.figure(figsize=(9, 6), dpi=1000) #set default figure size
import numpy as np

        ################################################################
        #                          Funções                             #
        ################################################################

def utilidade(c_estado_b, c_estado_g, prob_estado_b, prob_estado_g):

    return prob_estado_b*np.log(c_estado_b) + prob_estado_g*np.log(c_estado_g)

def restr_orcamentaria(c_estado_b, gamma, dot_estado_b, dot_estado_g):
```

```

    b = dot_estado_g + dot_estado_b*gamma/(1-gamma)

    return b - (gamma*c_estado_b)/(1-gamma)

def escolha(dot_estado_b, dot_estado_g, prob_estado_b, prob_estado_g, gamma):

    b = dot_estado_g + dot_estado_b*gamma/(1-gamma)
    c = prob_estado_g/prob_estado_b

    c_b_estrela = b*prob_estado_b*(1-gamma)/gamma
    c_g_estrela = c_b_estrela*c*(gamma/(1-gamma))
    funcao_valor = utilidade(c_b_estrela, c_g_estrela, prob_estado_b,
                             prob_estado_g)

    return c_b_estrela, c_g_estrela, funcao_valor

def seguro_otimo(dot_estado_b, dot_estado_g, prob_estado_b, prob_estado_g,
                 gamma):

    escolha_otima = escolha(dot_estado_b, dot_estado_g, prob_estado_b,
                            prob_estado_g, gamma)

    return (dot_estado_g - escolha_otima[1]) / gamma

        ################################################################
        #                          Exemplo                             #
        ################################################################

dot_estado_b1, dot_estado_g1 = 250, 350
gamma_1 = .4
prob_estado_b1, prob_estado_g1 = .4, .6

exemplo_1 = escolha(dot_estado_b1, dot_estado_g1, prob_estado_b1,
                    prob_estado_g1, gamma_1)

seguro_estrela = seguro_otimo(dot_estado_b1, dot_estado_g1, prob_estado_b1,
                    prob_estado_g1, gamma_1)

        ################################################################
        #                   Informando ao usuário                      #
        ################################################################

print(f"\nCom dotação ({dot_estado_b1}, {dot_estado_g1});", end = " ")
print("distribuição de probabilidade dada por", end = " ")
print(f"({prob_estado_b1*100}%,{prob_estado_g1*100}%),", end = " ")
```

```python
print(f"e premio {gamma_1*100}%, a distribuicao de consumo ótimo", end = " ")
print(f"será c_b = {exemplo_1[0]: .1f} e c_g = {exemplo_1[1]: .1f}.")
print(f"Ainda, a quantidade de seguro ótima é {seguro_estrela: .1f}.")

        ##################################################################
        #                          Gráfico                               #
        ##################################################################

c_estado_b1 = np.linspace(0.01, 1000, 1000)
c_estado_g1 = np.linspace(0.01, 1000, 1000)
orcamento = restr_orcamentaria(c_estado_b1, gamma_1, dot_estado_b1,
                               dot_estado_g1)
X, Y = np.meshgrid(c_estado_b1, c_estado_g1)

Z = plt.contour(X, Y, utilidade(X, Y, prob_estado_b1, prob_estado_g1),
                levels = [exemplo_1[2]-10, exemplo_1[2], exemplo_1[2]+10],
                linewidths = 2)
plt.plot(c_estado_b1, orcamento, color = "black", lw = 2,
         label = "restrição orçamentária")
plt.clabel(Z, inline = 1, fontsize = 10)
plt.ylim(0, dot_estado_g1 + dot_estado_b1*gamma_1/(1-gamma_1))
plt.xlim(0, (1-gamma_1)*(dot_estado_g1 + dot_estado_b1*gamma_1/(1-gamma_1))
         /gamma_1)
plt.ylabel(r"Consumo no estado 0 (bom)", fontsize = 15)
plt.xlabel(r"Consumo no estado 1 (ruim)", fontsize = 15)
plt.scatter([exemplo_1[0]], [exemplo_1[1]], s = 7*4**2)
plt.annotate(f"$(c_1^*, c_2^*) = ({exemplo_1[0]: .0f}, {exemplo_1[1]: .0f})$",
             (exemplo_1[0]+10, exemplo_1[1]+10), fontsize = 13)
plt.scatter([dot_estado_b1],[dot_estado_g1], s = 7*4**2,
            label = f"dotação inicial = $({dot_estado_b1},{dot_estado_g1})$")
plt.annotate("A", (dot_estado_b1-15,dot_estado_g1-30), fontsize = 13)
plt.legend()
plt.show()
```

A.7 Stag Hunt Game (Rousseau)

```python
# -*- coding: utf-8 -*-

"""
@author: Matheus L. Carrijo
"""
        ##################################################################
```

```
        #                       Importando pacotes                         #
        ####################################################################

import numpy as np
import random as rd # Precisaremos desta pacote para o computador escolher
                    # jogar o jogo Stag Hunt
import matplotlib.pyplot as plt
        ####################################################################
        #                         Definindo funções                        #
        ####################################################################

def payoff(jogo):
    """


    Parameters
    ----------
    jogo : Lista.
        A primeira entrada é a jogada do primeiro jogador e a segunda entrada
        é a jogada do segundo jogador. Para um jogo ser válido, ele deve estar
        dentro da matriz de jogos possíveis.

    Returns
    -------
    Lista
        Retorna uma lista em que a primeira entrada representa o payoff do
        primeiro jogador e a segunda entrada o payoff do segundo jogador.

    """

    if jogo in matriz:

        if jogo == ['s', 's']:
            return [5, 5]
        if jogo == ['s', 'h']:
            return [0, 3]
        if jogo == ['h', 's']:
            return [3, 0]
        if jogo == ['h', 'h']:
            return [3, 3]

    else:

        return -1

def rodada():
```

```
    """


    Returns
    -------
    jogo : Lista.
        Retorna uma lista contendo o jogo feito pela escolha das estratégias
        pelos dois jogadores.

    """

    jogada_A, jogada_B = rd.choice(estrategia), rd.choice(estrategia)
    jogo = [jogada_A, jogada_B]

    return jogo


def nash_equilibrium_analitico(jogo):
    """


    Parameters
    ----------
    jogo : Lista.
        Recebe uma lista do jogo definido pelas ações dos jogadores.

    Returns
    -------
    bool
        Calcula o equilíbrio de forma analítica do jogo, isto é, sabemos de
        antemão que o equilíbrio consiste nos jogos em que ambos os jogadores
        escolhem ao mesmo tempo caçar lebre ou veado.

    """

    if jogo in matriz:

        payoffs = payoff(jogo)

        if payoffs == [5, 5] or payoffs == [3,3]:

            equilibrio = True

        else:

            equilibrio = False
```

```python

            return equilibrio

        else:

            return -1


def nash_equilibrium(jogo):
    """


    Parameters
    ----------
    jogo : Lista
        Recebe uma lista do jogo determinado pelas ações (escolha das
        estratégias) dos jogadores.

    Returns
    -------
    Bool
        Assim como na função anterior, calcula o equilíbrio de Nash. Aqui,
        porém, fazemos um algorítmo em que o cálculo do equilíbrio baseia-se
        na análise da mudança de estratégia unilateral de cada jogador: se
        houver uma melhora no payoff dos jogadores, então não há equilibrio;
        caso contrário, há.

    """

    if jogo in matriz:

        payoffs = payoff(jogo)
        equilibrio = True

        if jogo[0] == 'h':

            jogo[0] = 's'
            payoff_aux = payoff(jogo)

            if payoff_aux[0] > payoffs[0]:

                equilibrio = False

            jogo[0] = 'h'

        if jogo[0] == 's':
```

```

            jogo[0] = 'h'
            payoff_aux = payoff(jogo)

            if payoff_aux[0] > payoffs[0]:

                equilibrio = False

            jogo[0] = 's'

        if jogo[1] == 'h':

            jogo[1] = 's'
            payoff_aux = payoff(jogo)

            if payoff_aux[1] > payoffs[1]:

                equilibrio = False

            jogo[1] = 'h'

        if jogo[1] == 's':

            jogo[1] = 'h'
            payoff_aux = payoff(jogo)

            if payoff_aux[1] > payoffs[1]:

                equilibrio = False

            jogo[1] = 's'

        return equilibrio

    else:

        return -1

def tabela_latex():

    global tabela

    matriz0 = np.array([[[5,5], [0,3], [3,0], [3,3]]])
    matriz = matriz0.flatten() # Transforma a matriz de payoffs em uma lista

    # Cria a tabela usando a sintaxe do latex:
```

```
    # (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}}{{$s$}}  & \\multicolumn{{1}}{{c}}{{$
        \\cline{{3-4}} \\multirow{{2}}*{{Jogador 1}}  & $s$ & ({matriz[0]},{matriz[1]}) & ({matriz[2
        \\cline{{3-4}} & $h$ & ({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
```

```python
                        # (jogar "s") ou caçar lebre (jogar "h").
matriz = [['s', 's'], ['s', 'h'],
          ['h', 's'], ['h', 'h']] # jogos possíveis determiandos pela matriz.

equilibrio = False # Para entrar no loop abaixo
i = 1 # contagem do número de jogos

while not equilibrio: #Jogos serão feitos até que se alcance um equilibrio

    jogo = rodada() #Um jogo é uma lista com as jogadas dos dois jogadores
    equilibrio = nash_equilibrium(jogo) #Confere se o jogo constitui equilibrio
    payoffs = payoff(jogo) #Calcula o Payoff do jogo

    print("\nJogaremos até encontrarmos um jogo com Equilíbrio de Nash.")

    if jogo == ['s', 'h']:

        print(f"\nJogo {i}: O jogador A falhou em tentar caçar", end = " ")
        print("o veado sozinho; o jogador B conseguiu caçar", end = " ")
        print("a lebre sozinho. Como o jogador A poderia", end = " ")
        print("melhorar sua situação mudando sua estratégia e", end = " ")
        print("caçar uma lebre sozinho, este jogo não", end = " ")
        print("constitui um Equilíbrio de Nash.")

    if jogo == ['h', 's']:

        print(f"\nJogo {i}: O jogador B falhou em tentar caçar", end = " ")
        print("o veado sozinho; o jogador A conseguiu caçar", end = " ")
        print("a lebre sozinho. Como o jogador B poderia", end = " ")
        print("melhorar sua situação mudando sua estratégia e", end = " ")
        print("caçar uma lebre sozinho, este jogo não", end = " ")
        print("constitui um Equilíbrio de Nash.")

    if jogo == ['h', 'h']:

        print(f"\nJogo {i}: Ambos os jogadores escolheram", end = " ")
        print("caçar lebres e conseguiram. Como nenhum jogador", end = " ")
        print("tem a ganhar mudando sua estratégia unilateralme", end = "")
        print("nte para caçar um veado, então este jogo constit", end = "")
        print("ui um Equilíbrio de Nash. Note, no entanto, que ambo", end = "")
        print("s melhorariam sua situação se combinassem de caçar um veado.")

    if jogo == ['s', 's']:

        print(f"\nJogo {i}: Ambos os jogadores escolheram", end = " ")
        print("caçar veados e conseguiram. Como nenhum jogador", end = " ")
```

```
        print("tem a ganhar mudando sua estratégia unilateralme", end = "")
        print("nte para caçar uma lebre, então este jogo constit", end = "")
        print("ui um Equilíbrio de Nash. Note que este é o melhor", end = " ")
        print("resultado que os jogadores poderiam ter.")

    i += 1

i -= 1

        ################################################################
        #                     Invocando a tabela                       #
        ################################################################

tabela_latex()
```

A.8 Oligopólio

```
# -*- coding: utf-8 -*-

"""
@author: Matheus L. Carrijo

Objetivo do programa: implementar computacionalmente o modelo que descreve a
competição de oligopólio com apenas duas empresas idênticas (duopólio) e com
apenas um período, chamado Modelo de Cournot. Cada empresa toma a decisão
de quanto produzir baseada na expectativa de produção da outra empresa, de
maneira a maximizar o lucro. O equilíbrio de previsões é a situação em que cada
empresa vê sua crença sobre a outra confirmada. Por simplicidade, também
considero custos nulos. Depois, o modelo é ampliado para o caso com n empresas
"""

        ################################################################
        #                     Importando pacotes                       #
        ################################################################

import matplotlib.pyplot as plt #importing graph package
plt.figure(figsize=(10, 6), dpi=1000) #set default figure size
import numpy as np #importing numpy

        ################################################################
        #                      Definindo funções                       #
        ################################################################
```

```python

def demanda_inversa(Y, a = 1000, b = 1): #preço em função da quantidade
    """
    Definindo a demanda inversa, que depende da produção total da economia e
    dos parâmetros 'a' e 'b'. O parâmetro Y é uma lista em que a entrada i
    representa a produção da empresa i.
    """

    assert a > 0 and b > 0, "Os parâmetros da demanda devem ser positivos"

    global c, d
    c, d = a, b

    return a - b*(sum(Y))

def receita(y, Y):
    """
    Receita = preço*quantidade. No caso do oligopólio, o preço é dado pela
    demanda inversa e a quantidade determinada pela firma.
    """

    return y*demanda_inversa(Y)

def curva_reacao():
    """
    Serve para o caso em que numero_firmas = 2. A função resolve o problema de
    maximização da firma arbitrária (1 ou 2) em função da produção que ela
    espera da outra firma, ou seja, a função gera a curva de melhor resposta.
    O equiilibrio de Cournot será a intersecção entre ambas as curvas.
    """

    demanda_inversa([0,1]) # Chamo a função apenas para definir o parâmetro 'c'

    # Resolvendo o problema para uma empresa arbitrária (empresa 1 ou 2):

    receita_valores = []
    y_argmax = []
    y, y_e = np.linspace(0, c, c+1), np.linspace(0, c, c+1)

    for j in y_e:

        for i in y:

            receita_valores.append(receita(i, [i,j]))

        indice_max = np.argmax(receita_valores) # Guarda o índice da maior
```

```
                                                # receita
        y_argmax.append(y[indice_max]) # Adiciona a uma lista o nível de
                                       # produção y que maximiza o lucro
        receita_valores = [] # "zera" a lista de receita para o algoritmo
                             # resolver novamente o problema de maximizacao de
                             # lucro para outro nível de produção esperado

    return y_e, y_argmax

def equilibrio_cournot(numero_firmas = 2):
    """
    Esta função serve para o caso geral em que numero_firmas = n. A solução é
    implementada algebricamente. As firmas são idênticas e o custo é nulo. O
    cálculo é realizado para o caso especial de demanda inversa linear.
    """

    coeficientes = np.array([[2 if j==i else 1 for i in range(numero_firmas)]
                    for j in range(numero_firmas)])
    inversa_coeficientes = np.linalg.inv(coeficientes)

    constantes = np.array([c/d for i in range(numero_firmas)])

    vetor_producao = np.dot(inversa_coeficientes, constantes)

    return vetor_producao

        ################################################################
        #                  Programa principal                          #
        #                 (chamando as funções)                        #
        ################################################################

curva_de_reacao = curva_reacao()

equilibrio_cournot1 = equilibrio_cournot() # Para n=2 firmas

quantidade_equilibrio = sum(equilibrio_cournot1)

preco_equilibrio = demanda_inversa(equilibrio_cournot1)


     ########## Analisando o efeito do aumento de firmas ############

preco2 = []
quantidade2 = []
n_firmas = 100

```

```
for i in range(1, n_firmas + 1): # para cada numero de firmas, calcular as
                                 # a quantidade e preço de equilíbrio da
                                 # economia.

    equilibrio_cournot2 = equilibrio_cournot(i)
    quantidade2.append(sum(equilibrio_cournot2))
    preco2.append(demanda_inversa(equilibrio_cournot2))

        ###############################################################
        #                   informando ao usuário                     #
        ###############################################################

print("O preço e a quantidade de equilíbrio são dados respectivamente por:")
print(f"{quantidade_equilibrio: .2f} e {preco_equilibrio: .2f}")

        ###############################################################
        #                        Gráficos                             #
        ###############################################################

# Gráfico para o exemplo de 2 firmas (curvas de reacao):

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

# Gráfico para a análise da entrada de firmas:

plt.plot(np.linspace(1, n_firmas, n_firmas), quantidade2, color="k",
         linewidth=2.0, ls = "dashed", label = "Quantidade")
plt.plot(np.linspace(1, n_firmas+1, n_firmas), preco2, color="r",
         linewidth=2.0, label = "Preço")
plt.xlabel(r"Número de firmas", fontsize = 15)
plt.legend(fontsize = 13)
plt.show()
```

A.9 Caixa de Edgeworth

```python
# -*- coding: utf-8 -*-
"""
Created on Wed Dec 15 22:59:27 2021

@author: Matheus L. Carrijo
"""

        ################################################################
        #                     Importando pacotes                       #
        ################################################################

import numpy as np #importing numpy
import matplotlib.pyplot as plt #importing graph package
#plt.figure(figsize=(9, 6), dpi=1000) #set default figure size

        ################################################################
        #                       Definindo funções                      #
        ################################################################

def utilidade(x, y, alpha = 0.5):
    """
    Função de utilidade Cobb-Douglas dos agentes (ambos terão a mesma).
    """

    return (x**alpha)*(y**(1-alpha))

def demanda(p, dot, alpha = 0.5):
    """
    Resolve o problema do consumidor de forma analítica, estabelecendo a
    demanda pelos bens x e y. Como as funções de utilidade são as mesmas para
    ambos os agentes, a demanda será igual para valores arbitrários da dotação
    inicial. O parâmetro 'dot' é uma lista em que a primeira entrada
    corresponde à dotação inicial do bem x; e a segunda para o bem y. O
    parâmetro preço é uma lista em que a primeira entrada corresponde ao preço
    do bem x e a segunda o preço do bem y.
    """

    x = alpha*(p[0]*dot[0] + p[1]*dot[1]) / p[0]
    y = (1 - alpha)*(p[0]*dot[0] + p[1]*dot[1]) / p[1]

    return x, y

```

```
def demanda_liq(p, dot):
    """
    Calcula o excesso de demanda para os bens x e y, isto é, a diferença entre
    a demanda total e a dotação inicial dos agentes. O parâmetro p é uma lista
    com os preços de x e y, respect., enquanto que dot é uma lista com as
    dotações iniciais dos bens x e y, respect..
    """

    x, y = demanda(p, dot)
    demanda_liq_x = x - dot[0]
    demanda_liq_y = y - dot[1]

    return demanda_liq_x, demanda_liq_y

def demanda_liq_agreg(p, dot_A, dot_B):
    """
    Calcula a demanda líquida agregada para os bens x e y. A demanda líquida
    agregada nada mais é que a soma das demandas líquidas individuais.
    """

    demanda_liq_agreg_x = demanda_liq(p, dot_A)[0] + demanda_liq(p, dot_B)[0]
    demanda_liq_agreg_y = demanda_liq(p, dot_A)[1] + demanda_liq(p, dot_B)[1]

    return demanda_liq_agreg_x, demanda_liq_agreg_y

def equilibrio(dot_A, dot_B):
    """
    Calcula o equilíbrio geral para um mercado competitivo de dois bens e dois
    indivíduos. Seguindo o que diz a Lei de Walras, o preço do bem x pode ser
    fixado em 1 (bem numerário) enquanto que achamos o preço do bem 2 que zera
    o excesso de demanda agregado do bem 2.
    """

    # Procurando o zero da demanda agregada excedente para o bem y
    # pelo método da bissecção:

    global px

    px = 1
    tol = 1e-10
    p_alto = 1000
    p_baixo = 0

    # Analisando a fronteira:
    tentativa1 = demanda_liq_agreg([px, p_alto], dot_A, dot_B)[1]
    if abs(tentativa1) < tol:
```

```python
            return p_alto

    # atualiza preco pela média
    preco = p_alto / 2
    excesso_demanda_agreg = demanda_liq_agreg([px, preco], dot_A, dot_B)[1]

    while abs(excesso_demanda_agreg) > tol:

        if excesso_demanda_agreg > 0:

            p_baixo = preco

        else:

            p_alto = preco

        preco = (p_baixo + p_alto) / 2
        excesso_demanda_agreg = demanda_liq_agreg([px, preco], dot_A, dot_B)[1]

    return preco

def curva_contrato():
    """
    Calcula um subconjunto da curva de contrato variando as dotações.
    """

    precos_equilibrio = [] # Lista para adicionar os preços de equilíbrio
                           # conforme variamos as dotações.

    y_aux = np.linspace(0, total_dot_y, 5)
    # Não estou "varrendo" toda a caixa de Edgeworth para calcular as
    # demandas de equilíbrio. Passo apenas inteiramente por um eixo (eixo
    # x, por ex) e vario alguns pontos do eixo y. Por este motivo, a curva
    # resultante é um subconjunto da curva de contrato.

    for j in y_aux:

        for i in x:

            precos_equilibrio.append(equilibrio([i, j], [total_dot_x - i,
                                                          total_dot_y - j]))

    x_estrela = []
    y_estrela = []

    for k in precos_equilibrio:
```

```python
        for m in y_aux:

            for l in x:

                x_estrela.append(demanda([1, k], [l, m])[0])
                y_estrela.append(demanda([1, k], [l, m])[1])

    return [x_estrela, y_estrela]

        #################################################################
        #                    Programa principal                         #
        #################################################################

#dot_A = [100, 25] # dotação indivíduo A
#dot_B = [50, 150] # dotação indivíduo B

#dot_A = [23, 33] # dotação indivíduo A
#dot_B = [14, 32] # dotação indivíduo B

dot_A = [230, 100] # dotação indivíduo A
dot_B = [400, 90] # dotação indivíduo B

preco_equilibrio_y = equilibrio(dot_A, dot_B)
p_equilibrio = [px, preco_equilibrio_y]

#Calcula as demandas de equilíbrio para os indivíduos A e B, respect.:
demanda_eq_A = demanda(p_equilibrio, dot_A)
demanda_eq_B = demanda(p_equilibrio, dot_B)

# Utilidade dos agentes em equilíbrio:
u_valor_A = utilidade(demanda_eq_A[0], demanda_eq_A[1])
u_valor_B = utilidade(demanda_eq_B[0], demanda_eq_B[1])


total_dot_x = dot_A[0] + dot_B[0] # pode-se considerar o tamanho do eixo
                                  # horizontal da caixa de Edgeworth
total_dot_y = dot_A[1] + dot_B[1] # pode-se considerar o tamanho do eixo
                                  # vertical da caixa de Edgeworth

x = np.linspace(0.0001, total_dot_x, 100) # grid da quantidade do bem x
y = np.linspace(0.0001, total_dot_y, 100) # grid da quantidade do bem y

contrato = curva_contrato()

        #################################################################
```

```
        #                  Informando ao usuário                        #
        ################################################################

print(f"\nCom dotações dos agentes A e B dadas por {dot_A} e", end = " ")
print(f"{dot_B} as alocações de equilíbrio para os indivíduos $A$", end = " ")
print(f"e e $B$ são, respect., ({demanda_eq_A[0]:.2f},", end = " ")
print(f"{demanda_eq_A[1]:.2f}) e ({total_dot_x-demanda_eq_A[0]:.2f}", end = "")
print(f", {total_dot_y-demanda_eq_A[1]:.2f}).\n")
print("A curva de contrato na figura mostra as diferentes", end = " ")
print("alocações eficientes para variações nas dotações iniciais.\n")

        ################################################################
        #                         Gráfico                              #
        ################################################################

#plt.style.use('classic')
#plt.style.use('default')
plt.rcParams["figure.figsize"] = (10, 6)
plt.rcParams["lines.linewidth"] = (2.5)

fig, ax = plt.subplots(tight_layout = True)

# Cria o produto cartesiano das quantidades x_1 e x_2.
xc_1, xc_2 = np.meshgrid(x, y)

# Coloca label nos eixos originais
ax.set_xlabel('$x_A$', fontsize = 20)
ax.set_ylabel('$y_A$', fontsize = 20)

ax.axis([0, 1.02*total_dot_x, 0, 1.05*total_dot_y])

# Calcula a utilidade do consumidor A
uA = utilidade(xc_1, xc_2) # Calcula a matriz de utilidades do
                           # consumidor A.

# Calcula a utilidade do consumidor B. Como queremos que o gráfico tenha origem
# em x = x0 e y = y0, fazemos o ajuste nos argumentos da função.
uB = utilidade(total_dot_x - xc_1, total_dot_y - xc_2) # Calcula a matriz de
                                                       # utilidades do
                                                       # consumidor B.

# Criamos um novo eixo y (apenas para repetir os ticks e labels do eixo y
#                          original)
ax2 = ax.twinx()

# Copiamos os limites do eixo y original
```

```
orig_ylim = ax.get_ylim()

# Replicamos os limites do eixo y original no novo eixo
ax2.set_ylim(orig_ylim)

# Definimos um label para o novo eixo y
ax2.set_ylabel('$y_B$', fontsize = 20)

# Invertemos o sentido do texto do eixo y original
ax2.invert_yaxis()

# Criamos um novo eixo x (apenas para repetir os ticks e labels do eixo x
#                          original)
ax3 = ax.twiny()

# Copiamos os limites do eixo x original
orig_xlim = ax.get_xlim()

# Replicamos os limites do eixo x original no novo eixo
ax3.set_xlim(orig_xlim)

# Definimos um label para o novo eixo x
ax3.set_xlabel('$x_B$', fontsize = 20)

# Invertemos o sentido do texto do eixo x original
ax3.invert_xaxis()

# Curvas de indiferença no nível indicado
#CS1 = ax.contour(xc_1, xc_2, uA, levels = [u_valor_A], linestyles = 'dashdot',
#                 colors = 'black', alpha = 1, extend = 'both')
#CS2 = ax.contour(xc_1, xc_2, uB, levels = [u_valor_B], linestyles = 'dotted',
#                 colors = 'forestgreen', alpha = 1, extend = 'both')

CS1 = ax.contour(xc_1, xc_2, uA, levels = [u_valor_A - 70, u_valor_A,
                                           u_valor_A + 50, u_valor_A + 100],
                 linestyles = 'dashdot', colors = 'k', alpha = 1,
                 extend = 'both')
CS2 = ax.contour(xc_1, xc_2, uB, levels = [u_valor_B - 100, u_valor_B - 50,
                                           u_valor_B, u_valor_B + 70],
                 linestyles = 'dotted', colors = 'g', alpha = 1,
                 extend = 'both')

# Curva de contrato:
ax.plot(contrato[0], contrato[1], 'blue', linewidth = 2, alpha = 0.6,
        label = 'Curva de contrato')

```

```
#Alocação de equilíbrio:
ax.scatter(demanda_eq_A[0], demanda_eq_A[1], s = 10*5**2,
           label = "Alocação de equilíbrio", alpha = 1, c = "r")
#ax.plot(demanda_eq_A[0], demanda_eq_A[1], 'r', marker = ".", markersize = 25,
#        label = "Alocação de equilíbrio")

# Dotação inicial:
ax.scatter(dot_A[0], dot_A[1], s = 10*5**2, label = "Dotação", alpha = 1,
           c = "black")
#ax.plot(dot_A[0], dot_A[1], 'black', marker = ".", markersize = 25,
#        label = "Dotação")
ax.legend(loc = "upper right", fancybox = True, shadow = True, fontsize = 13)
plt.show()
```