Computational Methods HW1

Omer Lurie

May 2025

Exercise 2.1

Consider the matrix

$$A = \begin{bmatrix} 1 & 5 & 2 & 3 \\ 1 & 6 & 8 & 6 \\ 1 & 6 & 11 & 2 \\ 1 & 7 & 17 & 4 \end{bmatrix}$$

a)

Compute the matrices L and U applying the Gaussian elimination method by hand.

We perform LU decomposition with pivoting on the matrix A:

$$A = \begin{pmatrix} 1 & 5 & 2 & 3 \\ 1 & 6 & 8 & 6 \\ 1 & 6 & 11 & 2 \\ 1 & 7 & 17 & 4 \end{pmatrix}$$

The LU decomposition with pivoting results in the following matrices:

Permutation Matrix P

Python's scipy uses a permutation matrix P, so we will consider P first. The permutation matrix P, reflecting row swaps during the Gaussian elimination process, is:

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

This matrix reflects that the second and fourth rows of A were swapped, as well as the third and fourth rows, ensuring numerical stability during the process.

Lower Triangular Matrix L

The lower triangular matrix L is:

$$L = \begin{pmatrix} 1. & 0. & 0. & 0. \\ 1. & 1. & 0. & 0. \\ 1. & 0.5 & 1. & 0. \\ 1. & 0.5 & -1. & 1. \end{pmatrix}$$

The matrix L contains the multipliers used during the Gaussian elimination process. Each entry below the diagonal represents the multiplier used to eliminate values below the pivots in U.

Upper Triangular Matrix U

The upper triangular matrix U is:

$$U = \begin{pmatrix} 1. & 5. & 2. & 3. \\ 0. & 2. & 15. & 1. \\ 0. & 0. & 1.5 & -1.5 \\ 0. & 0. & 0. & 1. \end{pmatrix}$$

The matrix U contains the upper triangular form of the matrix A, obtained after performing Gaussian elimination on the permuted matrix PA.

LU Decomposition

Thus, the LU decomposition of A with pivoting is:

$$PA = LU$$

Where P is the permutation matrix, L is the lower triangular matrix, and U is the upper triangular matrix.

Check your results using the subroutine lu_dec (A, L, U) from the toolbox. Finally, inspect the matrix product of L and U.

```
P,L,U = lu(A)
print(f'product of L and U is:\n {np.matmul(L,U)}')
```

Snippet 1: LU Decomposition

This yields the same solutions as before:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0.5 & 1 & 0 \\ 1 & 0.5 & -1 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 5 & 2 & 3 \\ 0 & 2 & 15 & 0 \\ 0 & 0 & 1.5 & -1.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathrm{A} = \begin{bmatrix} 1 & 5 & 2 & 3 \\ 1 & 6 & 8 & 6 \\ 1 & 6 & 11 & 2 \\ 1 & 7 & 17 & 4 \end{bmatrix}$$

b)

Solve the linear equation system

$$Ax = b, \quad \text{with} \quad b = \begin{bmatrix} 1 \\ 2 \\ 1 \\ 1 \end{bmatrix}$$

```
#dot product
Pb=np.dot(P,b)
#solving for y
y = np.linalg.solve(L,Pb)
#solving for x
x = np.linalg.solve(U,y)
#check that the solution is correct
np.linalg.solve(A,b)
#dot product of A and x
np.dot(A,x)
```

Snippet 2: solving a linear equation system

This yields the solutions:

$$x = \begin{bmatrix} 36 \\ -8 \\ 1 \\ 1 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 2 \\ 1 \\ 1 \end{bmatrix}$$

Exercise 2.2

Consider the intertemporal household problem represented by the utility function:

$$U(c_1, c_2) = \frac{c_1^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} + \beta \frac{c_2^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}}$$

with c_1 and c_2 denoting consumption in the first and the second period, γ is the ELS (intertemporal elasticity of substitution) and β defines the time discount factor. The household receives labour income w in the first period and does not work in the second period, as is often seen in overlapping generation models. Consequently, the budget constraint is:

$$c_1 + \frac{c_2}{1+r} = w$$

where r is the interest rate.

a)

Define the Lagrangian for this specific optimization problem and derive the first-order conditions with respect to c_1, c_2, and λ. Solve the equation system analytically using parameter values $\gamma = 0.5$, $\beta = 1$, $r = 0$, and $w = 1$.

$$\mathcal{L}(c_1, c_2, \lambda) = \frac{c_1^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} + \beta \cdot \frac{c_2^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} + \lambda \left(w - c_1 - \frac{c_2}{1+r} \right)$$

FOCs:

$$\frac{\partial \mathcal{L}}{\partial c_1} = c_1^{-\frac{1}{\gamma}} - \lambda = 0$$
$$\Rightarrow \lambda = c_1^{-\frac{1}{\gamma}}$$

$$\frac{\partial \mathcal{L}}{\partial c_2} = \beta c_2^{-\frac{1}{\gamma}} - \lambda \cdot \frac{1}{1+r} = 0$$
$$\Rightarrow \lambda = \beta(1+r) c_2^{-\frac{1}{\gamma}}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = w - c_1 - \frac{c_2}{1+r} = 0$$

Euler-equation:
By inserting for λ we can derive the Euler Equation

$$c_1^{-\frac{1}{\gamma}} = \beta(1+r) c_2^{-\frac{1}{\gamma}} \quad \Rightarrow \quad \left(\frac{c_1}{c_2} \right)^{\frac{1}{\gamma}} = \beta(1+r)$$

The Euler equation expresses describes the households intertemporal consumption preferences and allow us to find the equilibrium between the allocations for c_1 and c_2 given r. (The allocation that sets the utility derived from consumption in periods 1 and 2 equal at the margins).

Inserting the parameters:

$$\left(\frac{c_1}{c_2}\right)^{0.5} = 1 \quad \rightarrow \quad c_1 = c_2$$

β being one implies the household is indifferent between consumption in periods 1 and 2. With r = 0 saving today does not increase consumption tomorrow by a factor of more than 1. Therefore, the household will set consumption in periods 1 and 2 equal to each other.

plugging in the 3rd FOC: $c_1 = c_2 = 0.5$ Using the first or second FOC we get $\lambda = 4$

b)

Solve the equation system resulting from a) using function fzero from the toolbox. Print the results and compare the numerical results with the analytical solutions.

To solve numerically we define the FOC's as follows:

```
# Defining equations
def equations(vars):
    c1, c2, lam = vars
    eq1 = 1 / c1**2 - lam
    eq2 = 1 / c2**2 - lam
    eq3 = c1 + c2 - 1
    return [eq1, eq2, eq3]
```

Snippet 3: Defining the equations

defining an initial guess matrix and solving numerically:

```
# Initial guess
initial_guess = [0.4, 0.6, 1.0] #These are arbitrary values, they can be
    anything. Guesses that are far away from the solution will take longer
    to converge.

# Solve the system
solution = fsolve(equations, initial_guess) #fsolve is a function that
    solves a system of nonlinear equations. It uses the newton method to
    find the solution.

# Print results
c1, c2, lam = solution #solution is an array of the solutions to the system
     of equations.
print(f"Numerical solution using :")
print(f"c1 = {c1:.4f}")
print(f"c2 = {c2:.4f}")
print(f"lambda = {lam:.4f}")'
```

Snippet 4: Numerical solution

This yields the same solutions, namely $c_1 = 0.5,\ c_2 = 0.5,\ \lambda = 4$

c)

Solve the household problem using the subroutine fminsearch and compare the results. To solve with the equivalent of fminsearch we first have to insert the budget constraint in the utility function:

$$U(c_1, c_2) = \frac{c_1^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} + \beta \frac{((w-c_1)(1+r))^{1-\frac{1}{\gamma}}}{1-\frac{1}{\gamma}} \tag{2}$$

We set up the negative utility function (since we are minimizing), and the budget constraint as follows:

```
# Objective function (negative utility to perform maximization)
def neg_utility(c):
    c1, c2 = c #c is an array of initial guesses for c1 and c2
    if c1 <= 0 or c2 <= 0: #check if the initial guesses are positive
        return np.inf   #if not, return infinity
    return 1 / c1 + 1 / c2  # Negative of (-1/c1 - 1/c2)

# Constraint: c1 + c2 = 1
constraint = {
    'type': 'eq', #constraint type (equality)
    'fun': lambda c: c[0] + c[1] - 1 #constraint function (c1 + c2 = 1)
}
```

Snippet 5: utility function

We provide a budget constraint and call the minimizing function.

```
# Initial guess
initial_guess = [0.4, 0.6] #initial guess for c1 and c2

# Call the optimizer
result = minimize(neg_utility, initial_guess, method='SLSQP',
constraints=constraint) #minimize is a function that minimizes a function.
#It uses the SLSQP method to find the solution.
#SLSQP is a Sequential Least Squares Programming method.
#It is a method for solving constrained optimization problems.

# Extract results
c1_opt, c2_opt = result.x
#result.x is an array of the solutions to the optimization problem.

print("Solution using minimization :")
print(f"c1 = {c1_opt:.4f}")
print(f"c2 = {c2_opt:.4f}")
print(f"Utility = {-neg_utility([c1_opt, c2_opt]):.4f}")
```

Snippet 6: Minimize

This again yields the same results $c_1 = c_2 = 0.5$.

Exercise 2.3

We consider the problem of minimizing the function
over the interval . The problem is divided into three parts:

a)

Write a function `minimize(a, b)` of type `real*8`, which computes the local minimum of using Golden Search on the interval [a,b].

The golden-section search is a technique for finding the extremum (minimum or maximum) of a function inside a specified interval. We implement the method below:

```
#Function to Minimize
def function (x):
    return x*np.cos(x**2)

#golden search on interval
def golden_search_minimize (function, lower_bound, upper_bound): #function
    is the function to minimize, lower_bound is the lower bound of the
    interval, upper_bound is the upper bound of the interval

    return golden(function,brack=(lower_bound,upper_bound)) #golden is a
    function that finds the minimum of a function on an interval.
```

Snippet 7: Golden-section search minimizer

b)

Next split up the interval [0, 5] in n subintervals $[x_i, x_{i+1}]$ of identical length and compute for each interval the local minimum using the function `minimize`.

We divide the interval into 6 equally spaced subintervals and apply the minimizer to each:

```
lower_bound = 0
upper_bound = 5
subinterval_num = 6
interval = [lower_bound,upper_bound]
subinterval_bounds = np.linspace(lower_bound,upper_bound,subinterval_num) #
    creates an array of subinterval bounds
subintervals = [(subinterval_bounds[i],subinterval_bounds[i+1]) for i in
    range(len(subinterval_bounds)-1)] #creates an array of subintervals

minima = [golden_search_minimize(function,subintervals[i][0],subintervals[i
    ][1]) for i in range(len(subintervals))] #finds the minimum of the
    function on each subinterval
for i in range(len(subintervals)):
    print(f'The local minimum on [{subintervals[i][0]},{subintervals[i][1]}
     is {minima[i]:.4f}]\n') #prints the local minimum of the function on
    each subinterval
```

Snippet 8: Subinterval minimization

This yields the following minima:
Local minimum on [0.0,1.0 is -0.8083]
Local minimum on [1,2] is 1.8145
Local minimum on [2,3] is 4.6919
Local minimum on [3, 4] is 3.9673
Local minimum on [4,5] is 3.9673

c)

Sort out the global minimum from the set of computed local minima using the function `minloc(array, 1)`.

We use `numpy.argmin` to identify the index of the global minimum:

```
global_minimum_location = np.argmin(minima) #finds the index of the minimum
    of the function on the interval
print(f'\nThe global minimum is {minima[global_minimum_location]}') #prints
    the global minimum of the function on the interval
```

Snippet 9: Finding global minimum

This procedure identifies the overall global minimum on [0,5], namely -0.8082.

Exercise 2.4

We consider the following intertemporal household optimization problem. The household has a utility function: $U(c_1, c_2, c_3) = \sum_{i=1}^{3} \beta^{i-1} u(c_i)$, where $u(c_i) = \frac{c_i^{1-1/\gamma}}{1-1/\gamma}$ denotes consumption in period c_i, β is the time discount rate, and $\gamma > 0$ is the coefficient of relative risk aversion.

The household receives labor income in periods 1 and 2 and consumes all savings in period 3. The intertemporal budget constraint is: $\sum_{i=1}^{3} \frac{c_i}{(1+r)^{i-1}} = \sum_{i=1}^{2} \frac{w}{(1+r)^{i-1}}$ and $r \geq$ is the real interest rate.

To solve this, we proceed with the following steps:

(a) Reduction of Dimensionality

We substitute the budget constraint into the utility function, eliminating c_1 and expressing it in terms of c_2 and c_3 : $c_1 = w + \frac{w}{1+r} - \frac{c_2}{1+r} - \frac{c_3}{(1+r)^2}$ By substituting the budget constraint, we get:

$U(c_2, c_3) = \frac{c_1^{1-1/\gamma}}{1-1/\gamma} + \beta \frac{c_2^{1-1/\gamma}}{1-1/\gamma} + \beta^2 \frac{c_3^{1-1/\gamma}}{1-1/\gamma}$

b)

Minimize the negative utility function to determine the optimal c_2 and c_3.

We implement this using the SLSQP method for constrained optimization:

```
# Parameters
gamma = 0.5
beta = 1

# Objective: Negative utility w.r.t c2, c3
def neg_utility_c2_c3(c, r, w):
    c2, c3 = c #c is an array of initial guesses for c2 and c3
    rhs = w + w / (1 + r) #right hand side of the budget constraint
    c1 = rhs - c2 / (1 + r) - c3 / ((1 + r)**2) #solving for c1
    if c1 <= 0 or c2 <= 0 or c3 <= 0: #check if the initial guesses are
    positive
        return np.inf  # handle invalid values
    u1 = c1**(1 - 1/gamma) / (1 - 1/gamma) #utility function w.r.t c1
    u2 = beta * c2**(1 - 1/gamma) / (1 - 1/gamma)#utility function w.r.t c2
    u3 = beta**2 * c3**(1 - 1/gamma) / (1 - 1/gamma)#utility function w.r.t
     c3
    return -(u1 + u2 + u3) #negative utility function

def optimize_given_r_w(r, w): #function to optimize the utility function
    given r and w
    result = minimize(neg_utility_c2_c3, x0=[0.5, 0.5], args=(r, w), method
    ='SLSQP') #minimize is a function that minimizes a function. It uses the
     SLSQP method to find the solution.
    if result.success: #check if the optimization was successful
        c2_opt, c3_opt = result.x #result.x is an array of the solutions to
     the optimization problem.
```

```
        rhs = w + w / (1 + r) #right hand side of the budget constraint
        c1_opt = rhs - c2_opt / (1 + r) - c3_opt / ((1 + r)**2) #solving
    for c1
        utility = -neg_utility_c2_c3((c2_opt, c3_opt), r, w)  # flip sign
        return utility
    else:
        return np.nan #if the optimization was not successful, return nan
```

Snippet 10: Negative utility and optimization function

c)

Compute the optimal c_1 using the budget constraint.

We compute the optimal values as follows:

```
 --- Single optimization at baseline ---
r = 0
w = 1
rhs = w + w / (1 + r) #right hand side of the budget constraint

initial_guess = [0.3, 0.3] #initial guess for c2 and c3
result = minimize(neg_utility_c2_c3, initial_guess, args=(r, w), method='
    SLSQP') #minimize is a function that minimizes a function. It uses the
    SLSQP method to find the solution.


# Extract results
optimal_c2, optimal_c3 = result.x #result.x is an array of the solutions to
     the optimization problem.
optimal_c1 = rhs - optimal_c2 / (1 + r) - optimal_c3 / ((1 + r)**2) #
    solving for c1
optimal_utility = neg_utility_c2_c3([optimal_c2, optimal_c3],r,w) #negative
     utility function

print(f"Optimal consumption:\noptimal c1 = {optimal_c1:.4f}, optimal c2 = {
    optimal_c2:.4f}, optimal c3 = {optimal_c3:.4f}\nMaximum utility = {-
    optimal_utility:.4f}") #print the optimal consumption and maximum
    utility
```

Snippet 11: Optimal consumption levels and utility

Analysis of Parameter Changes

Effect of changing the interest rate: To observe how utility responds to varying interest rates:

```
r_vals = np.linspace(0.001, 0.2, 50) #create an array of interest rates
utilities_r = [optimize_given_r_w(r, w=1.0) for r in r_vals] #optimize the
    utility function given the interest rate and wage rate

plt.figure(figsize=(8, 5)) #create a figure
```

```
plt.plot(r_vals, utilities_r, label='Utility vs. Interest Rate', color='
    blue') #plot the utility function vs the interest rate
plt.xlabel('Interest Rate (r)') #x-axis label
plt.ylabel('Max Utility') #y-axis label
plt.title('Utility as a Function of Interest Rate') #title
plt.grid(True) #grid
plt.legend() #legend
plt.tight_layout()
plt.savefig('utility_vs_interest_rate.png', dpi=300, bbox_inches='tight')
plt.show()
```

Snippet 12: Utility as a function of interest rate

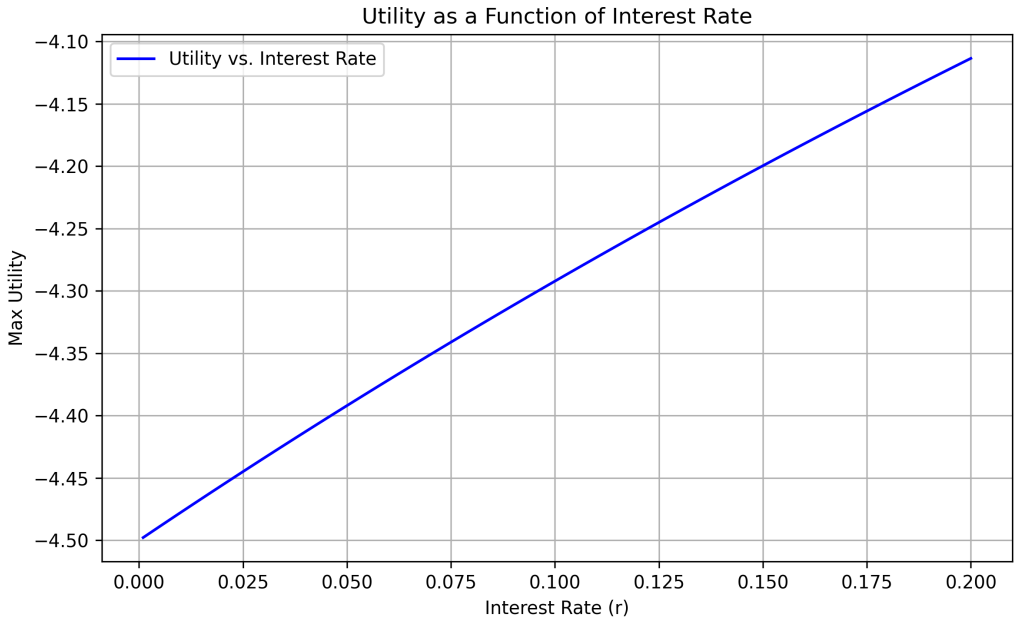

Figure 1: Utility as a function of the interest rate

Effect of changing the wage rate:

```
w_vals = np.linspace(0.5, 2.0, 50) #create an array of wage rates
utilities_w = [optimize_given_r_w(r=0.05, w=w) for w in w_vals] #optimize
    the utility function given the interest rate and wage rate

plt.figure(figsize=(8, 5)) #create a figure
plt.plot(w_vals, utilities_w, label='Utility vs. Wage Rate', color='green')
    #plot the utility function vs the wage rate
plt.xlabel('Wage Rate (w)') #x-axis label
plt.ylabel('Max Utility') #y-axis label
plt.title('Utility as a Function of Wage Rate') #title
```

```
plt.grid(True) #grid
plt.legend() #legend
plt.tight_layout()
plt.savefig('utility_vs_wage_rate.png', dpi=300, bbox_inches='tight')
plt.show()
```

Snippet 13: Utility as a function of wage rate

Figure 2: Utility as a function of the wage rate

A higher interest rate increases the return on deferred consumption, incentivizing the household to allocate a greater share of consumption to later periods. This intertemporal substitution effect leads to an initial rise in lifetime utility. However, the precise effect depends on the elasticity of intertemporal substitution governed by γ.

An increase in the wage level expands the feasible consumption set, resulting in a monotonic increase in utility. When wages are asymmetric across periods, the marginal value of income varies by timing, altering the optimal consumption allocation across periods.

Changes in the time preference parameter β influence the household's intertemporal preferences: lower β values increase present-bias, shifting optimal consumption toward earlier periods, while higher values smooth consumption more evenly across time.

Exercise 2.7: Tax Revenue Maximization

The government has raised labor income tax rates in recent years, but observed revenues did not increase monotonically. Given the data in the table below, we aim to determine the tax rate τ that maximizes tax revenue $T(\tau)$. We approach this using polynomial interpolation.

Year	Tax Rate (τ)	Tax Revenue (bn.)
2012	37	198.875
2013	42	199.500
2014	45	196.875

Table 1: Observed tax rates and corresponding revenues.

(a) Polynomial Interpolation and Revenue Maximization

We fit a quadratic polynomial $T(\tau) = a\tau^2 + b\tau + c$ through the three data points, then maximize it over the interval $\tau \in [35, 45]$. This captures the assumed concave nature of the tax-revenue relationship.

```
#data decleration
tax_rates = np.array([37, 42, 45]) #array of observed tax rates
tax_revenue = np.array([198.875, 199.5, 196.875]) #array of observed tax
    revenue

#polinomial interpolation
coefficients = np.polyfit(tax_rates,tax_revenue,deg=2) #fit a polynomial of
     degree 2 to the data
polynomial = np.poly1d(coefficients) #returns a representation of the
    polynomial's coefficients, in decreasing powers.
                                    #For example, poly1d([1, 2, 3])
    returns an object that represents x**2 + 2*x + 3

print(f'polynomial: {polynomial}')#print the polynomial

#evaluatie the function at different values
tax_rate_range = np.linspace(35, 45, 200) #create an array of tax rates
revenue_vals = polynomial(tax_rate_range) #evaluate the polynomial at the
    tax rates

# Maximizing revenue by minimizing the negative tax revenue function
result = minimize_scalar(lambda t: -polynomial(t), bounds=(35, 45), method=
    'bounded') #minimize the negative tax revenue function
optimal_tax_rate = result.x #result.x is the tax rate that maximizes the
    revenue
revenue_max = polynomial(optimal_tax_rate) #evaluate the polynomial at the
    optimal tax rate
```

Snippet 14: Polynomial interpolation and tax revenue maximization

The resulting quadratic revenue function is:

$$T(\tau) = \texttt{-0.125}\tau^2 + \texttt{10}\tau - \texttt{1.175e-11}$$

where the coefficients were computed from the data. The revenue-maximizing tax rate is:

$$\tau^* = 40, \quad T(\tau^*) = 200 \text{ billion}$$

(b) Visualization

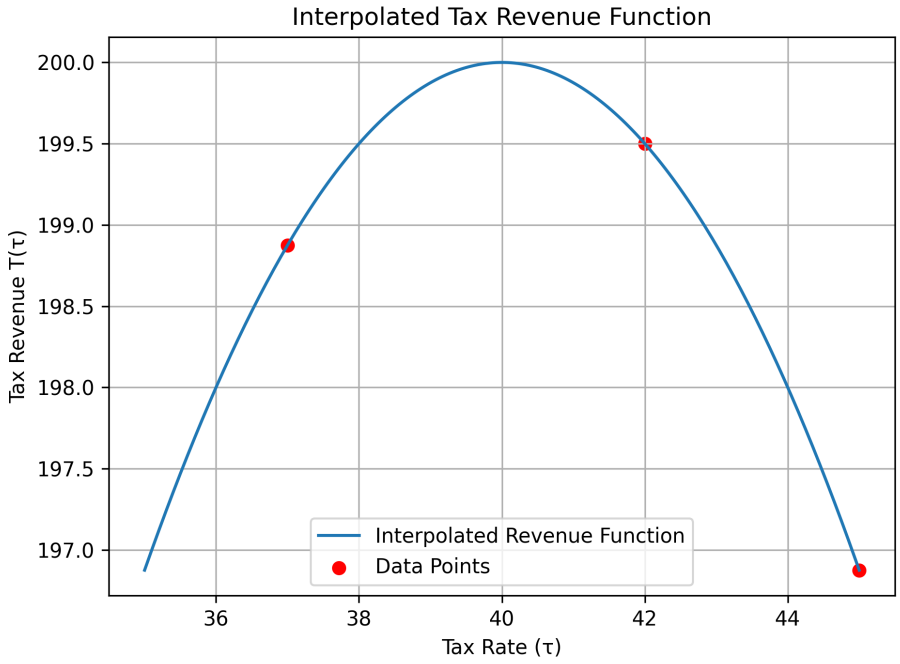

Figure 3: Interpolated tax revenue function with observed data points.

(c) Why Not Use Spline Interpolation?

Spline or piecewise linear interpolation is not suitable here because:

- **Piecewise Linear Interpolation** fails to capture the curvature in the revenue function and can misrepresent the maximum location.
- **Cubic Spline Interpolation** preserves local smoothness but may not respect the global concavity needed to find a well-defined maximum between sparse data points.

A global polynomial interpolation (specifically quadratic, since we have three points) is the most natural choice to capture the expected concave functional form.

Exercise 2.9

We consider a Cournot oligopoly with m identical firms competing in quantities. Market demand is given by the inverse function $D(P) = P^{-\eta}$, where $\eta > 0$ is the price elasticity of demand. Each firm chooses its quantity to equate marginal revenue with marginal cost. The marginal cost function is specified as:

$$MC(q) = \alpha\sqrt{q} + q^2,$$

where $\alpha > 0$ captures fixed cost intensity and convexity arises through the quadratic term. Under Cournot competition, each firm treats its competitors' output as fixed, so the change in price with respect to a single firm's output satisfies:

$$\frac{dP}{dq} = \frac{1}{D'(P)}.$$

(a) Grid and Individual Supply

We construct an equidistant price grid $P_0, \ldots, P_N \in [0.1, 3.0]$ using `np.linspace` and solve for each firm's optimal output $q(P)$ at each price via root-finding, using the `fsolve` (the Pythonic equivalent of `fzero`) method applied to the first-order condition:

$$P + q \cdot \frac{1}{mD'(P)} = MC(q).$$

```
#2.9
# Parameters
alpha = 1.0 #alpha represents the cost of production
eta = 1.5 #eta represents the elasticity of demand
m = 3 #m represents the number of firms
N = 10 #N represents the number of price points
NP = 1000 #NP represents the number of points for the fine price grid

# Price grid: equidistant in [0.1, 3.0]
P_grid = np.linspace(0.1, 3.0, N + 1) #create a price grid

# Demand function
def D(P):
    return P ** (-eta)

# Derivative of demand w.r.t price
def D_prime(P):
    return -eta * P ** (-eta - 1)

# Marginal cost function
def marginal_cost(q):
    return alpha * np.sqrt(q) + q**2

def firm_quantity(P):
    def F(q):
```

```
        if q < 0: #check if the quantity is negative
            return 1e6  # penalize negative q
        MB = P + q * (1 / D_prime(P)) * (1 / m) #marginal benefit
        MC = marginal_cost(q) #marginal cost
        return MB - MC

    q_guess = D(P) / m  # initial guess based on equal split
    q_sol, = fsolve(F, q_guess) #solve for the quantity
    return max(q_sol, 0)  # ensure non-negative quantity

# Compute quantities for each price
q_values = np.array([firm_quantity(P) for P in P_grid])
```

Snippet 15: Cournot market simulation with spline interpolation

(b) Interpolation of Supply and Plotting

We interpolate the firm-level supply function $q(P)$ using cubic splines, and scale it by m to obtain the aggregate market supply $Q(P) = m \cdot q(P)$. The figure below compares this supply curve with the market demand:

```
# Interpolation using CubicSpline
spline_q = CubicSpline(P_grid, q_values)

# Fine price grid for plotting
P_fine = np.linspace(0.1, 3.0, NP) #create a fine price grid
q_interp = spline_q(P_fine) #interpolate the quantity

# Plot demand and individual supply
plt.figure(figsize=(10, 6)) #create a figure
plt.plot(P_fine, D(P_fine), label="Market Demand D(P)") #plot the demand
plt.plot(P_fine, m * q_interp, label="Aggregate Supply (m*q(P))") #plot the
    aggregate supply
plt.xlabel("Price") #x-axis label
plt.ylabel("Quantity") #y-axis label
plt.title("Market Demand and Aggregate Supply") #title
plt.legend() #legend
plt.grid(True) #grid
plt.show()
```

Snippet 16: spline interpolation and supply plotting

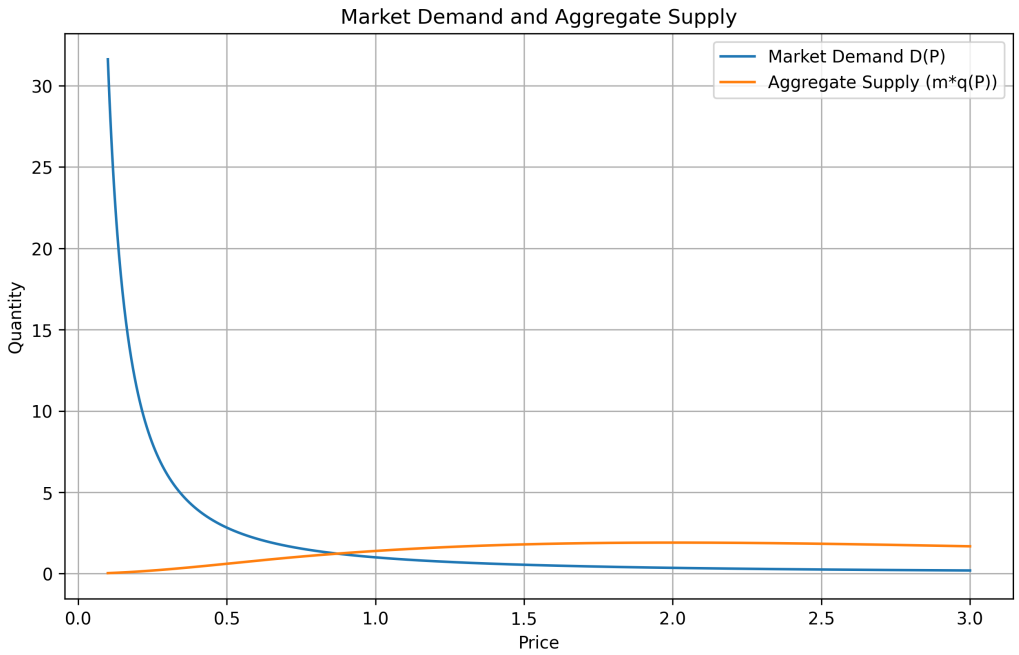

Figure 4: Market demand and aggregate supply in the Cournot oligopoly.

(c) Equilibrium Determination

The market equilibrium occurs at price P^* satisfying:

$$D(P^*) = Q(P^*) = m \cdot q(P^*).$$

We compute this equilibrium by solving the fixed-point condition numerically:

```
def market_excess(P):
    return m * spline_q(P) - D(P)

P_eq, = fsolve(market_excess, 1.0)
Q_eq = D(P_eq)

print(f"Equilibrium price: {P_eq:.4f}, Quantity: {Q_eq:.4f}")
```

Snippet 17: Solving for market equilibrium

and get the following equilibrium:

$$Q(P^*) = 1.2285$$

$$P^* = 0..8718$$

(d) Comparative Statics

We analyze how the equilibrium responds to changes in parameters:

- Increasing m intensifies competition, pushing price down and increasing total output.
- Higher α raises marginal cost, thereby reducing each firm's output and increasing price.
- Larger η makes demand more elastic, flattening the demand curve. This increases the responsiveness of quantity to price changes and compresses markups.

Exercise 2.10: Monopoly Pricing with Two-Sided Market

(a) Spline Approximation and Optimal Price Combination

The firm observes the following profit values $G(p_R, p_A)$ from different combinations of reader price p_R and advertiser price p_A:

$p_R \backslash p_A$	0.5	4.5	8.5	12.5
0.5	11.5	70.9	98.3	93.7
4.5	31.1	82.5	101.9	89.3
8.5	18.7	62.1	73.5	52.9
12.5	-25.7	9.7	13.1	-15.5

We construct a bivariate spline interpolation of the profit matrix using `RectBivariateSpline`, evaluate it on a fine grid, and determine the profit-maximizing price combination.

```
#Price grids
reader_prices = np.linspace(0.5, 12.5, 4)  # Reader price grid (rows)
ad_prices = np.linspace(0.5, 12.5, 4)  # Ad price grid (columns)
#Profit matrix G(pR, pA)
G = np.array([
    [11.5, 70.9, 98.3, 93.7],
    [31.1, 82.5, 101.9, 89.3],
    [18.7, 62.1, 73.5, 52.9],
    [-25.7, 9.7, 13.1, -15.5]
])
#spline interpolation
spline_G = interpolate.RectBivariateSpline(reader_prices, ad_prices, G)
#Evaluate the profit matrix at the fine price grid
Nplot = 100 #number of points for the fine price grid
reader_prices_fine = np.linspace(0.5, 12.5, Nplot) #create a fine price
    grid for the reader
ad_prices_fine = np.linspace(0.5, 12.5, Nplot) #create a fine price grid
    for the ad
profit_matrix = spline_G(reader_prices_fine, ad_prices_fine) #evaluate the
    profit matrix at the fine price grid
# maximum profit location
max_idx = np.unravel_index(np.argmax(profit_matrix), profit_matrix.shape) #
    Returns the indices of the maximum value in the 2D matrix (using argmax
    would have returned the flattened index)
reader_price_optimal = reader_prices_fine[max_idx[0]] #optimal reader price
ad_price_optimal = ad_prices_fine[max_idx[1]] #optimal ad price
profit_optimal = profit_matrix[max_idx] #optimal profit
print(f"(a) Optimal prices from interpolation: pR* = {reader_price_optimal
    :.2f}, pA* = {ad_price_optimal:.2f}")
print(f"    Maximum approximated profit: G(pR*, pA*) = {profit_optimal:.2f}
    ")
```

Snippet 18: Spline interpolation and optimal pricing from approximation

From this evaluation we get:

$$P_R^* = 2.68$$
$$P_A^* = 9.35$$
$$\pi(P_R^*, P_A^*) = 105$$

(b) Analytical Maximization with Demand Functions

Given demand functions:

$$x_R = 10 - p_R, \quad x_A = 20 - p_A - 0.5p_R$$

and constant marginal costs $c = 0.1$, we define the true profit function and use numerical optimization.

```
#profit function
def true_profit(prices):
    reader_price, ad_price = prices
    reader_demand = 10 - reader_price #demand for the reader
    ad_demand = 20 - ad_price - 0.5 * reader_price #demand for the ad
    if reader_demand < 0 or ad_demand < 0:#check if the demand is positive
        return -1e6  # Penalize infeasible demand
    profit = (reader_price - 0.1) * reader_demand + (ad_price - 0.1) *
    ad_demand #profit
    return -profit  # sign change to maximize profit

# Minimize the function
res = minimize(true_profit, x0=[5.0, 5.0], bounds=[(0.5, 12.5), (0.5, 12.5)
    ])
reader_price_optimal, ad_price_optimal = res.x
profit_optimal = -res.fun #sign change to maximize profit

print(f"(b) Optimal prices from true profit: pR* = {reader_price_optimal:.2
    f}, pA* = {ad_price_optimal:.2f}")
print(f"    Maximum true profit: G(pR*, pA*) = {profit_optimal:.2f}")
```

Snippet 19: True profit maximization using demand functions

From this evaluation we get:

$$P_R^* = 2.73$$
$$P_A^* = 9.37$$
$$\pi(P_R^*, P_A^*) = 105.01$$

(c) Accuracy with Different Resolutions

To analyze how resolution affects the interpolation, we vary the grid size $N_{plot} \in \{100, 1000, 10000\}$ and observe the consistency of optimal values.

```
#Considering different Nplots for smoothing
for Nplot in [100, 1000, 10000]:
    reader_prices_fine = np.linspace(0.5, 12.5, Nplot)
    ad_prices_fine = np.linspace(0.5, 12.5, Nplot)
    profit_matrix = spline_G(reader_prices_fine, ad_prices_fine)
    max_idx = np.unravel_index(np.argmax(profit_matrix), profit_matrix.
    shape)
    reader_price_optimal = reader_prices_fine[max_idx[0]]
    ad_price_optimal = ad_prices_fine[max_idx[1]]
    profit_optimal = profit_matrix[max_idx]
    error = abs(profit_optimal - profit_optimal)
    print(f"Nplot = {Nplot}: Approx Profit = {profit_optimal:.2f}, Error =
    {error:.4f}")
```

Snippet 20: Error analysis with different Nplot values

From this evaluation we get the following:
Nplot = 100: Approx profit = 105.00
Nplot = 1000: Approx profit = 105.01
Nplot = 10000: Approx profit = 105.01

Conclusion: The spline approximation provides a solution that converges to the analytical optimum with sufficiently fine resolution.

Exercise 2.11: Optimal Transport from Gravel Pits to Building Sites

Three gravel pits A1, A2, and A3 store 11, 13, and 10 tons of gravel, respectively. Four building sites B1, B2, B3, and B4 require 5, 7, 13, and 6 tons of gravel. The transport cost of one ton of gravel from pit A_i to site B_j is given in the table below:

	B1	B2	B3	B4
A1	10	70	100	80
A2	130	90	120	110
A3	50	30	80	10

Since total supply (34 tons) exceeds total demand (31 tons), we introduce a dummy destination B_5 with zero transport cost to absorb the surplus of 3 tons. We formulate and solve the transportation problem using a linear programming approach.

```
# Cost matrix (flattened row-wise). Os at the end for the dummy destination
costs = np.array([
    10, 70, 100, 80, 0,    # A1
    130, 90, 120, 110, 0, # A2
    50, 30, 80, 10, 0      # A3
])

# Supply from A1, A2, A3
supply = [11, 13, 10]

# Demand from B1, B2, B3, B4 + dummy B5 for the excess supply
demand = [5, 7, 13, 6, 3]

#constraint matrix
num_sources = 3
num_destinations = 5
n_variables = num_sources * num_destinations #number of variables

# Supply constraints (rows: each source)
supply_constraints   = np.zeros((num_sources, n_variables)) #creating a
    matrix of zeros
for i in range(num_sources):
    for j in range(num_destinations):
        supply_constraints[i, i * num_destinations + j] = 1 #setting the
    supply constraints

# Demand constraints (columns: each destination)
demand_constraints = np.zeros((num_destinations, n_variables)) #creating a
    matrix of zeros
for j in range(num_destinations):
    for i in range(num_sources):
        demand_constraints[j, i * num_destinations + j] = 1 #setting the
    demand constraints
```

```
# Combine all constraints
A_eq = np.vstack([supply_constraints, demand_constraints]) #stacking the
    supply and demand
b_eq = np.array(supply + demand) #creating a vector of constraints

# Bounds
bounds = [(0, None)] * n_variables #setting the bounds

# Solve
res = linprog(c=costs, A_eq=A_eq, b_eq=b_eq, bounds=bounds, method='highs')
     #solving the linear program

# Extract result
if res.success:
    X = res.x.reshape((num_sources, num_destinations)) #reshaping the
    solution
    print("Optimal transport plan (tons from Ai to Bj):")
    for i in range(num_sources):
        for j in range(num_destinations):
            if X[i, j] > 0:#check if the solution is positive
                print(f"A{i+1} -> B{j+1}: {X[i, j]:.2f} tons") #print the
    solution
    print(f"\nMinimum total cost: {res.fun:.2f}") #print the minimum total
    cost
else:
    print("Optimization failed.")
```

Snippet 21: Solving the transportation problem via linear programming

Results

The simplex algorithm finds the following optimal transport plan:

- A1 → B1: 5.00 tons
- A1 → B3: 6.00 tons
- A2 → B2: 3.00 tons
- A2 → B3: 7.00 tons
- A2 → B5 (dummy): 3.00 tons
- A3 → B2: 4.00 tons
- A3 → B4: 6.00 tons

The minimum total cost is:

$$\boxed{1{,}940.00}$$

This solution satisfies all demand and supply constraints, allocating the 3 tons of surplus to the dummy destination B_5 at zero cost, while minimizing total transportation costs.