Lab 1

Introduction to Python Programming for Economics & Finance

Richard Foltyn
University of Glasgow

May 22, 2023

Contents

Two-period consumption-savings problem

Consider the following standard consumption-savings problem over two periods with lifetime utility $U(c_1, c_2)$ given by

$$\max_{c_1,c_2} \frac{c_1^{1-\gamma}}{1-\gamma} + \beta \frac{c_2^{1-\gamma}}{1-\gamma} \qquad \text{s.t.} \qquad c_1 + \frac{c_2}{1+r} = w$$

where γ is the RRA coefficient, β is the discount factor, r is the interest rate, w is initial wealth, and (c_1, c_2) is the optimal consumption allocation to be determined.

1 Per-period utility function

Write a function `util(c, gamma)` which takes as arguments the consumption c and the risk-aversion γ and returns the per-period utility given by $u(c) = \frac{c^{1-\gamma}}{1-\gamma}$. Make sure that the function works with log preferences ($\gamma = 1$) as well as general CRRA preferences with $\gamma \neq 1$.

Hint: Use `np.log()` from the NumPy package to evaluate logs.

Solution.

[1]:
```python
import numpy as np

def util(c, gamma):
    if gamma == 1:
        # Utility for log preferences
        u = np.log(c)
    else:
        # Utility for general CRRA preferences
        u = c**(1.0 - gamma) / (1.0 - gamma)

    return u
```

2 Solving the problem using grid search

In a first step, you are going to solve the household problem using grid search, a basic algorithm that evaluates the objective function (lifetime utility) for every possible value of (c_1, c_2) on a grid of candidate consumption levels.

2.1 Objective function (lifetime utility)

Write the objective function `objective(c1, c2, beta, gamma)` which takes the candidate consumption choices and parameters as arguments and returns the associated lifetime utility. This function should call the per-period utility function `util(c, gamma)` you wrote above.

Solution.

[2]:
```python
# Objective function for maximisation problem

def objective(c1, c2, beta, gamma):
    # Use per-period utility functions to compute lifetime utility
    U = util(c1, gamma) + beta * util(c2, gamma)

    return U
```

2.2 Candidate consumption grid

Assume that the problem is parametrised using the following values:

[3]:
```python
# Parameters
r = 0.04
beta = 0.96
gamma = 1.0

# Initial wealth
wealth = 1.0
```

Create a uniformly spaced grid for candidate period-1 consumption levels c_1 called `c1_grid` with 20 points. What is the interval of feasible values for c_1?

Hint: Use `np.linspace()` to create a linearly (uniformly) spaced array.

Solution.

[4]:
```python
# Grid size
N = 20

# Candidate grid for period-1 consumption
c1_grid = np.linspace(0.0, wealth, N)
```

2.3 Grid search algorithm

Write a function `find_optimum(c1_grid, beta, gamma, r, wealth)` which takes as arguments the candidate first-period consumption levels as well as parameters and returns the following objects:

1. An array containing the candidate lifetime utility for each candidate c_1.
2. An integer index identifying the maximizer on that array.

Use this function to find the optimal consumption levels.

Hint: Loop over the points in `c1_grid`. For each c_1 on the grid, use the budget constraint to obtain the implied c_2. Then use the `objective(c1, c2, ...)` function to evaluate lifetime utility and keep track of the maximum.

Solution.

[5]:
```python
# Function to locate optimal consumption allocation
def find_optimum(c1_grid, beta, gamma, r, wealth):

    imax = 0

    # Initialize with -Inf
    U_max = - np.inf

    # Lifetime utility evaluated on candidate grid points
    U_grid = np.zeros(len(c1_grid))

    for i, c1 in enumerate(c1_grid):
        # Compute implied period-2 consumption from budget constraint
        c2 = (1.0 + r) * (wealth - c1)

        # Lifetime utility level for current consumption choice (c1, c2)
        U_try = objective(c1, c2, beta, gamma)

        # Store lifetime utility in result array
        U_grid[i] = U_try

        # Check for new maximum
        if U_try > U_max:
            # Update best allocation
            imax = i
            U_max = U_try

    # Return lifetime utilities for all candidate points and index of max
    return U_grid, imax
```

[6]:
```python
# Evaluate lifetime utility at each grid point and get optimal index
U_grid, imax = find_optimum(c1_grid, beta, gamma, r, wealth)
```

```
/tmp/ipykernel_301156/3699509581.py:6: RuntimeWarning: divide by zero
encountered in log
  u = np.log(c)
```

2.4 Reporting the results

Write a loop that prints the allocation (c_1, c_2) and the implied utility level $U(c_1, c_2)$ for each point on the candidate grid.

Solution.

[7]:
```
# Grid for implied period-2 consumption
c2_grid = (1.0 + r) * (wealth - c1_grid)

# Print lifetime utility for all candidate levels
for i in range(N):
    c1, c2, U = c1_grid[i], c2_grid[i], U_grid[i]
    # Print consumption using 4 decimal digits, use scientific notation
    # for utility
    print(f'Utility for c1={c1:.4f}, c2={c2:.4f}: {U:.5e}')

# Print optimal allocation
c1_opt, c2_opt, U_opt = c1_grid[imax], c2_grid[imax], U_grid[imax]
print(f'Utility at optimal c1={c1_opt:.4f}, c2={c2_opt:.4f}: {U_opt:.5e}')
```

```
Utility for c1=0.0000, c2=1.0400: -inf
Utility for c1=0.0526, c2=0.9853: -2.95869e+00
Utility for c1=0.1053, c2=0.9305: -2.32042e+00
Utility for c1=0.1579, c2=0.8758: -1.97315e+00
Utility for c1=0.2105, c2=0.8211: -1.74743e+00
Utility for c1=0.2632, c2=0.7663: -1.59052e+00
Utility for c1=0.3158, c2=0.7116: -1.47934e+00
Utility for c1=0.3684, c2=0.6568: -1.40203e+00
Utility for c1=0.4211, c2=0.6021: -1.35203e+00
Utility for c1=0.4737, c2=0.5474: -1.32574e+00
Utility for c1=0.5263, c2=0.4926: -1.32153e+00
Utility for c1=0.5789, c2=0.4379: -1.33929e+00
Utility for c1=0.6316, c2=0.3832: -1.38047e+00
Utility for c1=0.6842, c2=0.3284: -1.44841e+00
Utility for c1=0.7368, c2=0.2737: -1.54933e+00
Utility for c1=0.7895, c2=0.2189: -1.69456e+00
Utility for c1=0.8421, c2=0.1642: -1.90619e+00
Utility for c1=0.8947, c2=0.1095: -2.23481e+00
Utility for c1=0.9474, c2=0.0547: -2.84308e+00
Utility for c1=1.0000, c2=0.0000: -inf
Utility at optimal c1=0.5263, c2=0.4926: -1.32153e+00
```

2.5 Vectorized solution method

In "proper" quantitative work we hardly ever want to loop over elements of an array and perform computations individually. Instead, we prefer to operate on the whole array at once which is called *vectorization*. This is an essential technique when using languages such as Python and Matlab.

To illustrate the vectorized grid search algorithm, we proceed as follows:

1. Compute grid of implied period-2 consumption levels using the budget constraint.
2. Evaluate the lifetime utility for each point of the grid.
3. Locate the maximum on this grid of lifetime utilities.

[8]:
```
# Grid of period-2 consumption levels
c2_grid = (1.0 + r) * (wealth - c1_grid)
```

```
# Array of implied lifetime utilities
U_grid = objective(c1_grid, c2_grid, beta, gamma)
```

```
/tmp/ipykernel_301156/3699509581.py:6: RuntimeWarning: divide by zero
encountered in log
  u = np.log(c)
```

Note that we have already coded the function `objective()` in a way that it supports array-valued arguments for consumption.

Once we have the lifetime utility for each candidate consumption allocation, we can find the maximum and its location using `np.amax()` and `np.argmax()`:

[9]:
```
# max. lifetime utility
U_opt = np.amax(U_grid)

# Index where max. lifetime utility is attained
imax = np.argmax(U_grid)
```

Reassuringly, the results are identical to what we found using loops:

[10]:
```
# Print optimal allocation
c1_opt, c2_opt = c1_grid[imax], c2_grid[imax]
print(f'Utility at optimal c1={c1_opt:.4f}, c2={c2_opt:.4f}: {U_opt:.5e}')
```

```
Utility at optimal c1=0.5263, c2=0.4926: -1.32153e+00
```

---

3 Analytical solution

Compute the analytical solution for this problem and contrast it with what you found above. Why are the values not identical?

Hint: The analytical solution can be trivially derived from the first-order conditions given by the Euler equation and the budget constraint:

$$\begin{aligned} c_1^{-\gamma} &= \beta(1+r)c_2^{-\gamma} \\ c_1 + \frac{c_2}{1+r} &= w \end{aligned}$$

The optimal c_1 is then given by

$$c_1 = \alpha \cdot w \qquad \text{with} \qquad \alpha = \left[1 + \beta^{\frac{1}{\gamma}}(1+r)^{\frac{1}{\gamma}-1}\right]^{-1}$$

where α is the fraction of initial wealth consumption in period 1.

Solution.

[11]:
```
# Fraction of wealth consumed in period 1
alpha = (1.0 + beta**(1.0/gamma) * (1.0 + r)**(1.0/gamma - 1.0))**(-1.0)

# Optimal period-1 consumption
c1_opt = alpha * wealth

# Optimal period-2 consumption follows from Euler equation
# or from budget constraint
c2_opt = (beta * (1.0 + r))**(1.0/gamma) * c1_opt
```

```
# Implied lifetime utility
U_opt = objective(c1_opt, c2_opt, beta, gamma)

print(f'Utility at optimal c1={c1_opt:.4f}, c2={c2_opt:.4f}: {U_opt:.5e}')
```

```
Utility at optimal c1=0.5102, c2=0.5094: -1.32051e+00
```

The values are unlikely to be identical to the result we found using grid search since it is unlikely that the optimal c_1 was included in the grid. We can of course get closer to the analytical value by increasing the grid size which is trivial in this setting but might not be computationally feasible in a more realistic problem.

4 Solving the problem using a minimizer

We usually don't use grid search to find an optimum since the method is often slow and imprecise. In this part you will therefore explore how the optimal allocation can be found using SciPy's optimization routines.

4.1 Objective function

Since this is a scalar maximization problem (in either c_1 or c_2 as the other is implied by the budget constraint), you first need to write a modified objective function that is compatible with SciPy's optimisation routines. To this end, define a function `objective_scipy(c1, beta, gamma, r, wealth)` which takes c_1 and parameters as arguments and evaluates lifetime utility. Because we are going to run a **minimizer** implemented in `minimize_scalar()` to find the optimum, your objective function needs to return the **negative** lifetime utility.

Hint: Use the function `util(c, gamma)` you implemented previously to evaluate the objective function.

Solution.

[12]:
```
def objective_scipy(c1, beta, gamma, r, wealth):

    # Implied period-2 consumption
    c2 = (1.0 + r) * (wealth - c1)
    if c1 < 0.0 or c2 < 0.0:
        return - np.inf

    U = util(c1, gamma) + beta * util(c2, gamma)

    # Return negative utility since we are running minimizer
    return -U
```

4.2 Running the minimizer

To run the scalar minimizer, you need to import the function `minimize_scalar` as follows:

```
from scipy.optimize import minimize_scalar
```

In this particular case, `minimize_scalar()` takes the following arguments:

1. The objective function
2. The `method` parameter specifying the minimization algorithm (use `method='bounded'`)
3. The `bounds` parameter specifying the range of admissible values.

Note that because your objective function takes 4 arguments but SciPy expects a function with a single argument, you'll need to use a `lambda` function to pass any additional arguments.

Write the call to `minimize_scalar()` as follows:

```
res = minimize_scalar(..., method='bounded', bounds=... )
```

Alternatively, you can skip the `lambda` function and pass any additional parameters as a `tuple` using the `args` argument as follows:

```
res = minimize_scalar(..., method='bounded', bounds=..., args=...)
```

Run the minimizer and inspect the attributes of the resulting object `res`. You'll see that the maximizer is returned as `res.x` and the objective is stored in `res.fun`.

Solution.

[13]:
```
from scipy.optimize import minimize_scalar

# Variant 1: Call the minimizer wrapping the objective inside
# a lambda expression
res = minimize_scalar(
    lambda x: objective_scipy(x, beta, gamma, r, wealth),
    method='bounded',
    bounds=[0.0, wealth]
)
```

[14]:
```
# Variant 2: Use args to pass any additional arguments
# to objective function
res = minimize_scalar(
    objective_scipy,
    method='bounded',
    bounds=[0.0, wealth],
    args=(beta, gamma, r, wealth)
)
```

[15]:
```
# Inspect attributes of result object
res
```

[15]:
```
 message: Solution found.
 success: True
  status: 0
     fun: 1.3205083976674494
       x: 0.5102040863551822
     nit: 9
    nfev: 9
```

From the output we see the minimizer (or maximizer), the objective, the number of iterations (`nit`) and function evaluations (`nfev`) and the exit status.

4.3 Reporting the results

Print the optimal allocation (c_1, c_2) and the associated lifetime utility obtained by the minimizer. Does it differ from the result you found with grid search?

Solution.

[16]:
```
c1_opt = res.x
c2_opt = (1.0 + r) * (wealth - c1_opt)
U_opt = - res.fun

print(f'Utility at optimal c1={c1_opt:.4f}, c2={c2_opt:.4f}: {U_opt:.5e}')
```

```
Utility at optimal c1=0.5102, c2=0.5094: -1.32051e+00
```

These results are somewhat different from those we found using grid search. In fact, they are very close to the analytical solution.