Intro to Numerical Methods — Homework 1 Guide

Class I: The Neoclassical Investment Model and VFI

Alessandro T. Villa

Fall 2025

Learning Goals

In this first assignment you will:

1. Learn how to represent a simple deterministic growth model in MATLAB.
2. Implement Value Function Iteration (VFI) using both grid search and continuous optimization.
3. Use functions, loops, interpolation, and plotting to visualize results.

1 Model Description

We study a representative household maximizing lifetime utility:

$$\max_{\{C_t, K_{t+1}\}} \sum_{t=0}^{\infty} \beta^t \, u(C_t), \quad u(C_t) = C_t,$$

subject to

$$C_t + K_{t+1} = Z K_t^{\alpha} + (1-\delta) K_t.$$

The Bellman equation is

$$V(K) = \max_{K'} \{ u(C(K, K')) + \beta \, V(K') \}.$$

We assume normalized labor $N = 1$ and deterministic productivity $Z = 1$.

The steady-state capital implied by the Euler equation is:

$$K^* = \left[\frac{1/\beta - 1 + \delta}{Z\alpha} \right]^{1/(\alpha-1)}.$$

2 Model Parameters

Set the parameters as follows:

$$\beta = 0.96, \qquad \alpha = \tfrac{1}{3}, \qquad \delta = 0.1, \qquad Z = 1.$$

```
% Parameters
p.beta  = 0.96;
p.alpha = 1/3;
p.delta = 0.10;
p.Z     = 1;

% Steady-state capital (analytical)
Kstar = ((1/p.beta - 1 + p.delta)*(1/(p.Z*p.alpha)))^(1/(p.alpha - 1));
```

Listing 1: Parameter block

—

3 Part 1 — Brute-Force Value Function Iteration

In this first part, you will solve the deterministic model by evaluating all possible choices of K' on a fixed grid.

Steps

a) Create a grid of capital values around K^*:

$$K \in [K^*(1 - \texttt{Explore}),\, K^*(1 + \texttt{Explore})].$$

Use `linspace` with `NumNodes = 11`.

b) Initialize the value function $V(K)$ to zeros:

```
V = zeros(NumNodes,1);
```

c) Define production, investment, and consumption as anonymous functions:

```
F = @(K) p.Z * K.^p.alpha;
I = @(K,Kp) Kp - (1 - p.delta)*K;
C = @(K,Kp) F(K) - I(K,Kp);
```

d) For each K on the grid:

i. Loop over all possible K' values on the same grid.
ii. Compute consumption $C(K, K')$.
iii. If $C \leq 0$, assign a large negative utility (e.g. `-1e10`).
iv. Otherwise, compute utility $u(C)$ and the continuation value $\beta V(K')$.
v. Keep track of the maximizing K' and its value.

e) Iterate until the value function converges:

$$\max_{K_i} |V^{(t+1)}(K_i) - V^{(t)}(K_i)| < 10^{-5}.$$

Suggested structure (pseudocode)

```
while err > tol
for each K in grid
for each Kprime in grid
% Compute consumption and check feasibility
% Compute utility + discounted continuation value
end
% Take max over Kprime choices
end
% Update value function and convergence criterion
end
```

Plotting

After convergence, plot:

- The policy function $K'(K)$.
- The value function $V(K)$.

Use subplot and LaTeX labels for clarity. For example:

```
subplot(2,1,1)
plot(K_grid, Kprime_policy)
xlabel('$K$','interpreter','latex')
ylabel('$K''(K)$','interpreter','latex')
title('Policy Function')
```

—

4 Part 2 — VFI with Interpolation and Golden Section Search

Now allow the choice of K' to be continuous in the interval $[K_{\min}, K_{\max}]$. Instead of evaluating over all grid points, use a **golden-section search** to find the maximizing K'.

Key idea

For each K:

$$K'^*(K) = \arg \max_{K' \in [K_{\min}, K_{\max}]} \Big\{ u(C(K, K')) + \beta V(K') \Big\}.$$

You will:

- Use `interp1` to evaluate $V(K')$ at non-grid points.
- Write a helper function `goldenx.m` that performs 1D maximization.

Pseudocode structure

```
for each K in K_grid
obj = @(Kp) u_safe(C(K,Kp), p) + p.beta * interp1(K_grid, V, Kp);
[Kp_opt, Vnew(i)] = goldenx(obj, Kmin, Kmax);
Kprime_policy(i) = Kp_opt;
end
```

Repeat until the value function converges. Then plot the new policy and value functions.

Hints

- Check that your policy function stabilizes around K^*.
- Use a tolerance of 10^{-5} and print progress every few iterations.
- Use breakpoints and debug your code!

Soft Nonnegativity via a Quadratic Penalty

During numerical policy iteration, candidate choices may occasionally imply negative consumption due to exploratory updates or actual lack of resources. Instead of imposing a hard constraint $C \geq 0$—which introduces kinks and branching logic that can hinder convergence—we use a *soft penalty* that heavily discourages $C < 0$ while remaining smooth and differentiable.

We define

$$u_{\text{safe}}(C) = C - 10 \cdot \mathbf{1}\{C < 0\}\, C^2,$$

implemented in `Matlab` as:

```
function val = u_safe(C)
val = C - (C<0).*10.*(C).^2;
end
```

Interpretation. For feasible values $C \geq 0$, utility is unchanged: $u_{\text{safe}}(C) = C$. When $C < 0$, utility is sharply reduced by a quadratic term $10C^2$. This acts as a standard quadratic penalty that *mimics the inequality constraint* $C \geq 0$ without introducing discontinuities in the objective.

Smoothness and Concavity. The function is continuous and continuously differentiable at $C = 0$:

$$u'_{\text{safe}}(C) = \begin{cases} 1 - 20C, & C < 0, \\ 1, & C \geq 0, \end{cases} \qquad u''_{\text{safe}}(C) = \begin{cases} -20, & C < 0, \\ 0, & C > 0, \end{cases}$$

so that $u'_{\text{safe}}(0^-) = u'_{\text{safe}}(0^+) = 1$. The function is therefore C^1 and concave everywhere, which helps maintain standard monotonicity and contraction properties in value or policy iteration.

Numerical Advantages.

- **Corrective incentives:** For $C < 0$, the marginal utility is $u'_{\text{safe}}(C) = 1 - 20C > 1$, pushing the algorithm quickly back toward feasible $C \geq 0$.
- **Efficient implementation:** The logical mask $(C < 0)$ applies the penalty elementwise without branching, allowing for fully vectorized and fast evaluation. This is particularly handy for grid search.

Tuning the Penalty. The coefficient (here 10) controls how strongly the constraint binds. Larger values make violations costlier and rarer. In practice:

- Start with 10; if negative C values persist (e.g., $C_{\min} < -10^{-6}$), increase to 50 or 100.
- After convergence, verify feasibility ($\min C \geq -10^{-5}$). If small violations remain, tighten the penalty or project C to $\max\{C, 0\}$ in post-processing.

Caveats. Because the penalty is finite, tiny negative C values may remain if they marginally improve the objective elsewhere. With a large enough penalty and tight tolerance (10^{-5}), these are negligible.

Summary. The quadratic penalty provides a smooth, concave, and numerically stable way to enforce approximate nonnegativity of consumption. It preserves the correct behavior on the feasible region $C \geq 0$ while preventing the algorithm from exploring infeasible or unstable states.