

Endogenous Gridpoints Method bug: Implications for Heterogeneous Agents models

Abitesh Ranjan*

October 5, 2026

Abstract

A common mistake made while numerically implementing Endogenous Gridpoints Method (EGM) is to assume that the rate of return is constant across time. In this paper, I solve EGM numerically without assuming constant rate of return and compare the result when EGM is implemented with constant rate of return. This exercise will enable researchers to identify the bug early in the project.

*Indian Institute of Management Sirmaur

1 Introduction

The Endogenous Gridpoints Method was proposed by [Carroll \(2006\)](#). This method offers a significant improvement in speed relative to the conventional Value Function Iteration. It has been used extensively to solve heterogeneous agents model, for example, [Guvenen et al. \(2026\)](#), [Bayer et al. \(2026\)](#) and HANK models, such as [Auclert et al. \(2023\)](#), etc.

A common mistake made while implementing numerically Endogenous Gridpoints Method (EGM) is to assume that the rate of return is constant. In this paper, I solve EGM numerically without assuming constant rate of return and compare the result when EGM is implemented with constant rate of return. This exercise will enable researchers to identify the bug early.

Since an AI tool generates incorrect EGM code, therefore it will be safe to assume that the human written EGM codes, that were used for AI learning, are also be incorrect.

2 Model

For illustration, I consider a stochastic consumption-saving model, in which, a representative agent is endowed with stochastic income, e , which can take N_e values. This income follows a Markov process with transition probabilities $\Pi(e'|e)$, where e' is the next period income. The agent can save in a risk-free bond with return R . The consumer maximizes

$$\mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t u(c_t)$$

$$s.t. \quad c_t + a_{t+1} = e_t + Ra_t,$$

$$and \quad a_{t+1} \geq 0,$$

where $u(c) = \frac{c^{1-\sigma}}{1-\sigma}$. The Bellman equation is

$$V(a, e) = \max_{a' \geq 0} \{u(c) + \beta \mathbb{E}V(a', e')\}$$

$$s.t. \quad c + a' = e + Ra,$$

where the expectation is taken over all possible realizations of e' conditional on e . The first order and envelope conditions are:

$$u'(c) \geq \beta \mathbb{E} V_a(a', e'),$$

$$V_a(a, e) = R u'(c).$$

The income process

I assume that the income of the agent is drawn from a lognormal distribution with mean μ and variance σ^2 .

3 Analysis and Comparison

Since by definition, the rate of return is constant in a stationary state, there will be no effect of the bug on stationary distribution, further the bug will make no difference when there is an endowment shock as the interest rate remains constant. In what follows we will look at the case of partial equilibrium dynamic setup with shock to the rate of return. The common parameter values are shown in the Table 1. And the result of a 10% shock to the interest rate is shown in Figure 1 for the correct and the incorrect algorithm.

Parameter	Symbol	Value
Time	T	10
Discount rate	β	0.96
Elasticity of Substitution	σ	1
Steady state return	R_{ss}	1.02
Minimum asset level	$\underline{a}$	0
Maximum asset level	$\bar{a}$	5
Number of asset grids	N_a	101
Minimum endowment level	$\underline{e}$	0.5
Maximum endowment level	$\bar{e}$	1
Number of endowment grids	N_e	2

Table 1: Parameter values

Explanation

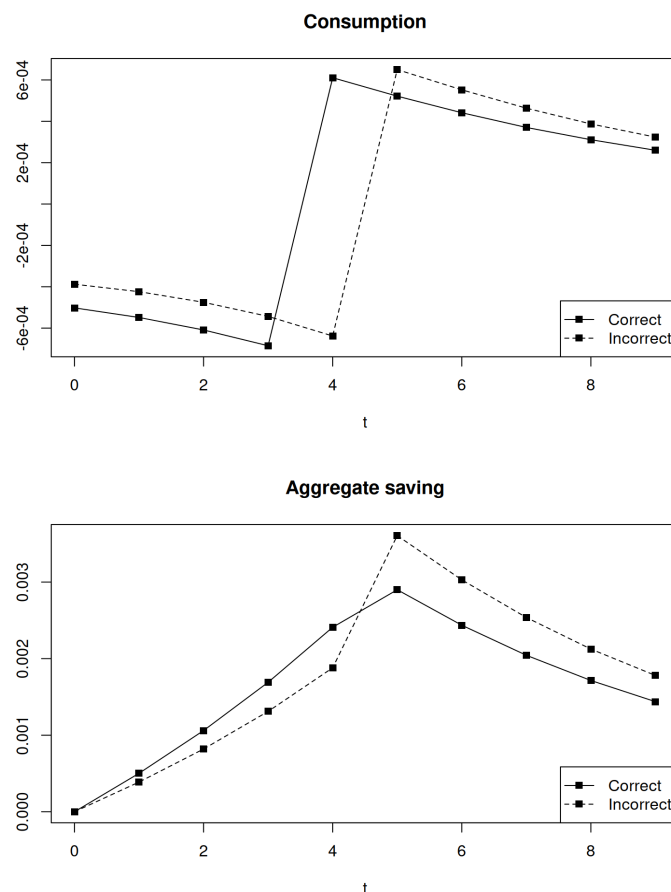


Figure 1: Response to 10% interest rate shock at $t=4$.

Bibliography

Adrien Auclert, Bence Bardoczy, and Mathew Rognlie. MPCs, MPEs, and Multipliers: A Trilemma for New Keynesian Models. *The Review of Economics and Statistics*, 105:700–712, 2023.

Christian Bayer, Ralph Luetticke, Maximilian Weiss, and Yannik Winkelmann. An endogenous gridpoint method for distributional dynamics. *Journal of Monetary Economics*, 158, 2026.

Christopher D. Carroll. The method of endogenous gridpoints for solving dynamic stochastic optimization problems. *Economics Letters*, 91:312–320, 2006.

Fatih Guvenen, Alisdair McKay, and Conor Ryan. A tractable income process for business cycle analysis. *Review of Economic Dynamics*, 61, 2026.

A Gemini(AI) generated work

Prompt: code for endogenous gridpoints method.

Response:

The **Endogenous Gridpoint Method (EGM)**, pioneered by Christopher Carroll in 2006, is a brilliant trick for solving dynamic stochastic optimization problems (like the classic cake-eating or consumption-savings problem) much faster than traditional Value Function Iteration (VFI).

Instead of guessing next period's assets and searching for the optimal current consumption using a costly non-linear solver, EGM defines a grid over **end-of-period savings** (post-decision state) and works **backward** to compute the corresponding beginning-of-period wealth endogenously.

Here is a clean, readable implementation in **Python** using 'numpy' and 'scipy'.

Python Implementation of EGM

This code solves a standard infinite-horizon consumption-savings model with income shocks:

$$\max \mathbb{E}_0 \sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\gamma}}{1-\gamma}$$

subject to $a_{t+1} = (1+r)(a_t - c_t) + y_{t+1}$.

Python

```
import numpy as np

from scipy.interpolate import interp1d

def solve_egm(params, grid_size=100, max_iter=500, tol=1e-6):
    # 1. Unpack parameters
```

```

beta  = params[,beta,]    # Discount factor
gamma = params[,gamma,]   # CRRA coefficient (risk aversion)
R      = params[,R,]       # Gross interest rate (1 + r)
y      = params[,y,]       # Income states (deterministic or Markov)
pi     = params[,pi,]      # Probabilities for income states

# CRRA Marginal Utility functions
def u_prime(c):
    return c ** (-gamma)

def u_prime_inv(mu):
    return mu ** (-1 / gamma)

# 2. Set up exogenous grid for end-of-period assets (savings)
# We choose where we want to end up, then back out where we started.
a_prime_grid = np.linspace(0.0, 50.0, grid_size)

# Initialize consumption policy function guess:  $c = 0.1 * \text{wealth} + \text{income}$ 
# Rows = wealth grid points, Columns = income states
c_policy = np.zeros((grid_size, len(y)))
for j in range(len(y)):
    c_policy[:, j] = 0.1 * a_prime_grid + y[j]

iteration = 0
pol_diff = 1.0

# 3. Main EGM Loop
while pol_diff > tol and iteration < max_iter:
    c_policy_old = c_policy.copy()

```

```

c_policy_new = np.zeros_like(c_policy)
x_grid_endogenous = np.zeros_like(c_policy)

# Loop over current income state today
for j in range(len(y)):

    # Compute Expected Marginal Utility tomorrow, conditional on saving a,
    rhs = np.zeros(grid_size)
    for j_next in range(len(y)):
        # Future cash-in-hand next period
        x_next = R * a_prime_grid + y[j_next]

        # Interpolate future consumption based on old policy
        c_next = np.interp(x_next, a_prime_grid + y[j_next], c_policy_old[:, j_next])

        # Accumulate expected marginal utility
        rhs += pi[j, j_next] * u_prime(c_next)

    # Euler Equation:  $u, (c) = \beta * R * E[u, (c_{next})]$ 
    mu_today = beta * R * rhs

    # Invert marginal utility to get current consumption directly! (No solver needed)
    c_today = u_prime_inv(mu_today)

    # Endogenous grid for cash-in-hand today:  $x = a, + c$ 
    x_today = a_prime_grid + c_today

    # Save endogenous grid and mapping
    x_grid_endogenous[:, j] = x_today

```

```

c_policy_new[:, j] = c_today

# Update policy function by evaluating it back on a fixed, common grid if nec
# or just monitor convergence on the structural mapping.
pol_diff = np.max(np.abs(c_policy_new - c_policy_old))
c_policy = c_policy_new
iteration += 1

print(f"EGM converged in {iteration} iterations.")
return a_prime_grid, x_grid_endogenous, c_policy

# --- Example Usage ---
if __name__ == "__main__":
    # Parameters setup
    params = {
        ,beta,: 0.96,
        ,gamma,: 2.0,
        ,R,: 1.02,
        ,y,: np.array([0.5, 1.5]), # Low / High income states
        ,pi,: np.array([[0.7, 0.3], [0.3, 0.7]]) # Transition matrix
    }

    a_grid, x_endo, c_pol = solve_egm(params)

    # x_endo[:, j] gives the cash-in-hand levels corresponding to c_pol[:, j]
    print("\nSample Output (First 5 points for Low Income state):")
    print("Cash-in-hand (x):", x_endo[:5, 0])
    print("Consumption (c):", c_pol[:5, 0])

```

Why this is incredibly fast:

In normal Value Function Iteration, you pick a point on a grid for cash-in-hand (x), and then evaluate ‘`scipy.optimize.minimize_scalar`’ to find the exact asset split (a') that satisfies the Euler equation.

EGM flips the timeline:

1. You **choose** a grid of future assets (a').
2. You compute the right-hand side of the Euler Equation via simple expectation.
3. You **invert** the utility function to get consumption ($c_t = (u')^{-1}(\text{RHS})$).
4. You add them together to get the grid point you *must have started at*: $x_t = a' + c_t$.

Because step 3 uses an analytical inverse instead of an iterative optimizer loop, EGM runs roughly **10x to 100x** faster than standard VFI.