Macroeconomics 3

TA session 2

Deterministic Global Solution Methods

Prof.: Marco Maffezzoli TA: Andrea Pasqualini

A.y. 2017-2018

PhD in Economics and Finance

Bocconi University, Milan

Clarification on classes, objects

- Think of Object-Oriented Programming (OOP) as an abstraction layer
  - Programmers can plan their code beforehand in terms of "what is this *thing* supposed to do?"
  - First *think* the main building blocks of your code, then *code* those blocks, finally put'em all together
  - Allows to temporarily forget about the sequence of specific instructions in the specific implementation (reusability)
- Python supports **inheritance**

```
1 class OLS:
2     # cannot 'raise' it
```

```
1 class MyError(Exception):
2     # can 'raise MyError'
```

See also: https://stackoverflow.com/questions/4015417/python-class-inherits-object

Back on track: today

- Value Function Iteration (VFI)
- Policy Function Iteration (PFI)
- Projection method

Value Function Iteration (VFI)

Given $V_0(\cdot)$, iterate over

$$V_{i+1}(x) = \max_{y \in D(x)} f(y) + \beta V_i(x')$$

Intuition: contraction mapping theorem ensures existence of fixed point (see also Ex. 9 in first exercise set)

- **Pros**
  - Extensive mathematical framework (it always converges)
  - Gives knowledge of the "utility" (e.g., welfare analysis)
- **Cons**
  - Converges slowly
  - Infeasible with high-dimensional state space (a.k.a. Achilles' Heel or Curse of Dimensionality)

Value Function Iteration: how?

Sketch of procedure:

1. Spawn a grid for the state variables X
2. Define an initial guess $V_0(x)$
3. Define a criterion (e.g., $crit = \max_x |V_{i+1}(x) - V_i(x)|$)
4. Compute $V_{i+1}(x) = \max_{y \in D(x)} f(y) + V_i(y(x))$ for each $x \in X$
5. If $crit < tol$ then stop; otherwise, repeat step 4

Catch:

- The fixed point is a function, not one number

Value Function Iteration: example

Consider the Neoclassical Growth Model in its deterministic version

$$\max_{c_t, k_{t+1}} \quad \sum_{t=0}^{\infty} \beta^t \frac{c_t^{1-\sigma}}{1-\sigma}$$

$$\text{s.t.} \quad \begin{cases} c_t + k_{t+1} = y_t \\ y_t = k_t^{\alpha} + (1-\delta)k_t \\ c_t \geq 0 \end{cases}$$

Combine constraints and write the problem in recursive formulation

$$V(k) = \max_{0 \leq k' \leq k^{\alpha} + (1-\delta)k} \frac{c^{1-\sigma}}{1-\sigma} + \beta V(k')$$

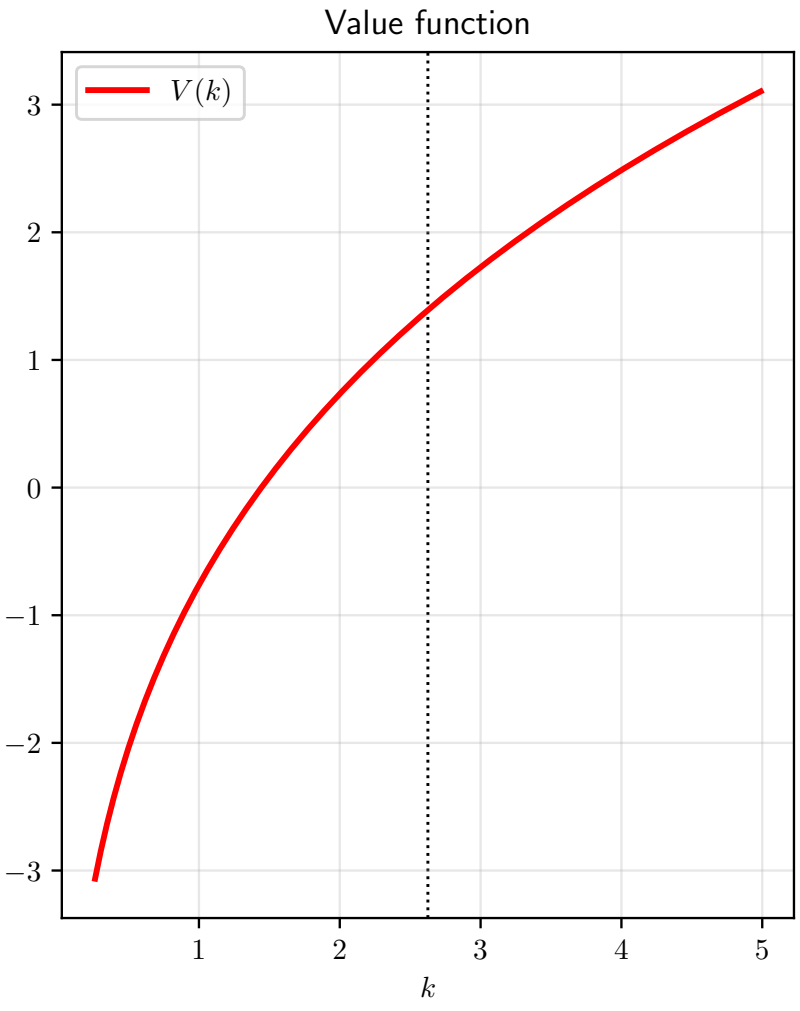
with $c = k^{\alpha} + (1-\delta)k - k'$

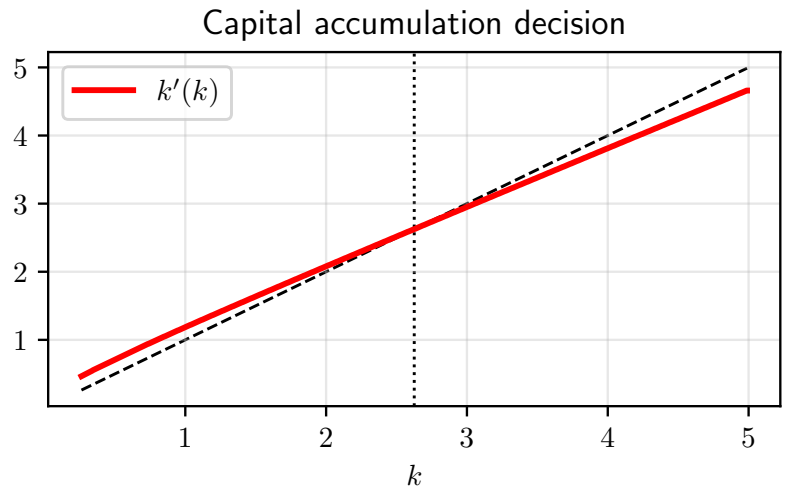
Value Function Iteration: example

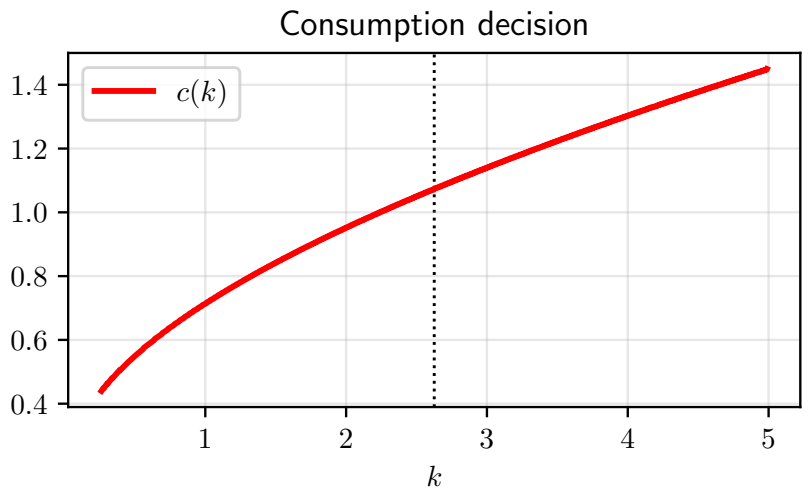
Sketch of code implementing VFI

```
import numpy as np
# define N, alpha, beta, delta, sigma
K = np.linspace(0.01, 5, num=N)   # grid for state variable
V0 = np.zeros((N, 1))             # preallocation (efficiency)
V1 = np.zeros((N, 1))             # preallocation (efficiency)
crit = 1.                         # dummy value to get 'while' started
tol = 1e-6
while crit > tol:
    for i in range(N):
        C = K[i]**alpha + (1-delta) * K[i] - K
        C[C < 0] = np.nan
        util = C ** (1-sigma) / (1-sigma)
        V1[i] = np.nanmax(util + beta * V0)
    crit = np.max(np.abs(V1 - V0))  # using sup norm
    V1[:] = V0      # 'deep copy' of array to avoid pass by reference
```

Value Function Iteration: example







Policy Function Iteration (PFI), a.k.a. Howard improvement

- Sometimes you want to see what happens if you mess around with parameters (e.g., sensitivity analysis, robustness checks). This implies that you have to solve your model lots of times. Time consuming, and your computer will start sweating...[1]
- Intuition (why it works): VFI assumes the policy function regards *tomorrow* as a function of *today*, forgetting what happens the day after tomorrow. PFI assumes that the policy function is always the same, spanning the entire horizon.
- Intuition (how it works): have a guess on the policy function. Does it solve the dynamic program? No? Use the policy function in a clever way to get a new proposal for it. Rinse and repeat.

[1] Treat well your tech, they're wonderful creatures: they're always right!

Policy Function Iteration (PFI), a.k.a. Howard improvement

- **Pros**
  - Flexible for counterfactuals (i.e., different parameters)
  - Still allows for recovering the value function
  - Typically faster than VFI (but see below)
- **Cons**
  - You have to prove convergence
  - Assumes $f_t(\cdot) = f(\cdot)$ for all t (e.g., bad for models with ownership of durables)
  - Requires the policy function to be sufficiently "regular" (e.g., bad for discrete choice models)
- Is it really faster than VFI? It depends...
  - Yes, if β is very close to one (...like, always in Econ)
  - No, if inverting $[I - \beta Q]$ is large (see next slide)

Policy Function Iteration: how?

Sketch of procedure:

1. Spawn a grid for the state variables X
2. Define an initial guess for policy function $f_0(x)$
3. Define a criterion for convergence
4. Given $f_i(x)$
   - Invert the VF to get $V(x) = [I - \beta Q]^{-1} u(f_i(x), x)$
   - Compute $f_{i+1}(x) = \arg\max_y u(y, x) + \beta V(x')$
5. If $crit < tol$ then stop; otherwise go back to 4

Catch: what's Q?

- $Q_{j,k} = \mathbf{1}\left(x_k = h(f_{i+1}(x_j), x_j)\right)$
- **Sparse** matrix regulating transitions from state x to x'

Policy Function Iteration: example

Going back to the Neoclassical Growth Model

$$V(k) = \max_{0 \leq k' \leq k^{\alpha} + (1-\delta)k} \frac{c^{1-\sigma}}{1-\sigma} + \beta V(k')$$

with $c = k^{\alpha} + (1-\delta)k - k'$

- Here we can take a shortcut and forget about $V(\cdot)$
- Obtain the Euler equation

$$c(k) = c(k'(k)) \left[\beta \left[\alpha \big(k'(k)\big)^{\alpha - 1} + 1 - \delta \right] \right]^{-1/\sigma}$$

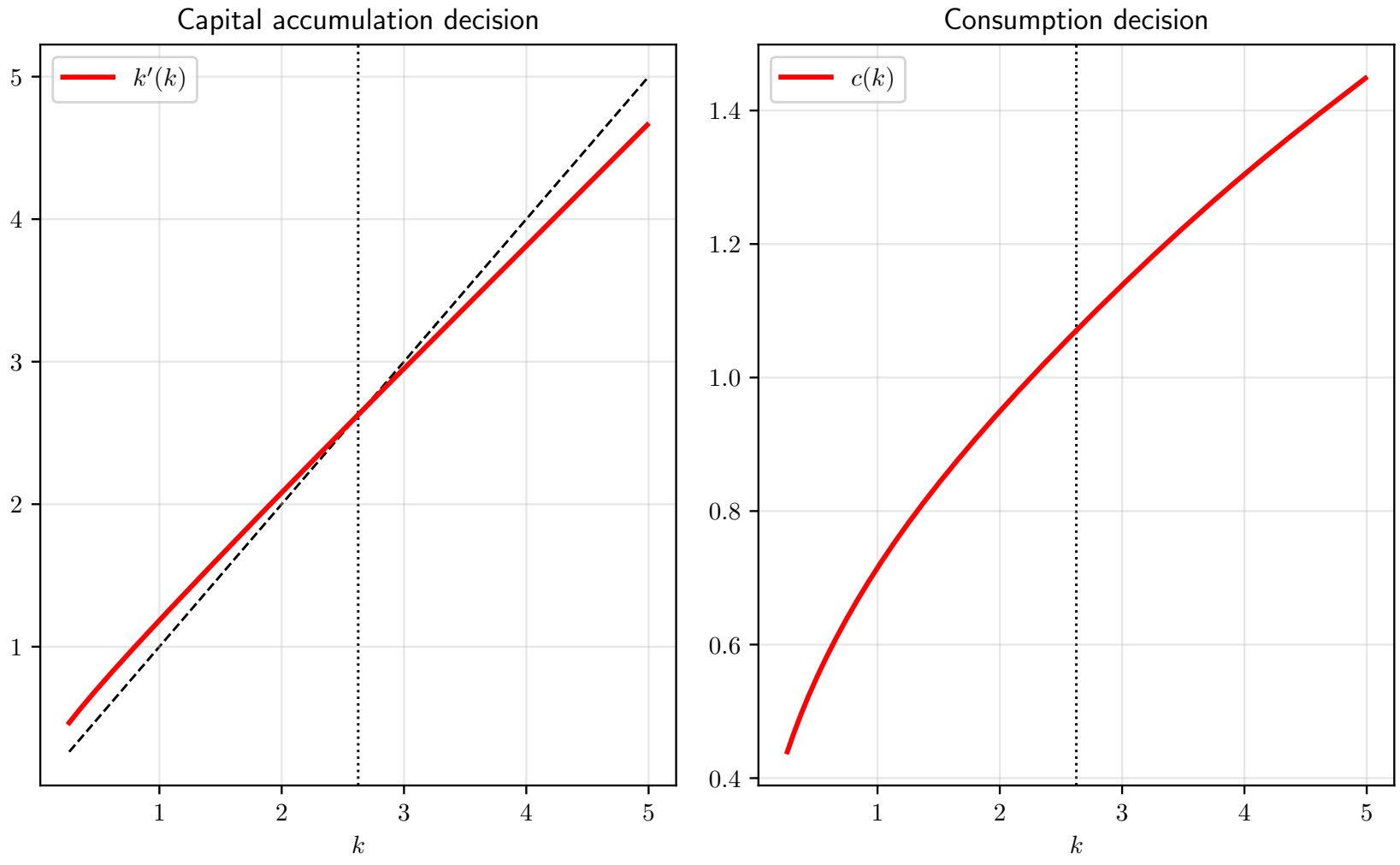
- Note that choosing $c(k)$ implies choosing $k'(k)$ through the budget constraint
- Can iterate and check consistency between $c(k)$ and $k'(k)$

Policy Function Iteration: example

Sketch of code implementing PFI

```
import numpy as np
# define N, alpha, beta, delta, sigma
K = np.linspace(0.01, 5, num=N)
C0 = 0.1 * np.ones((N,))
crit = 1.
tol = 1e-6
while crit > tol:
    k1 = K ** alpha - C0 + (1 - delta) * K
    pc = np.polyfit(K, C0, 5)    # fitting a polynomial for c(k)
    Ctp1 = np.polyval(pc, k1)    # fitted values for c(k'(k))
    opr = alpha * k1**(alpha-1) + 1 - delta  # one plus r
    C1 = Ctp1 * (beta * opr) ** (-1 / sigma)
    crit = np.max(np.abs(C1 - C0))
    C0[:] = C1  # deep copy of array
```

Policy Function Iteration: example



Projection method

Sometimes your system of (non-linear) equations is simple enough that you can solve it using standard numerical routines (e.g., quasi-Newton methods). In general, you can write models as

$$F(x_{t-1}, x_t, x_{t+1}) = 0,$$

and the goal is to get policy functions $g(x_{t-1}, x_t)$ such that $F(\cdot) = 0$. See Hall (2017, AER) for a practical example.

Pros

- Easy to implement
- Faster than VFI (not always)
- Convenient for on-the-go simulations

Cons

- Super-sensitive to initial guess
- Might be numerically inaccurate
- Good luck recovering the Value Function

Projection method: how?

Sketch of procedure:

- Spawn a grid for the state variables X
- Define a function that represents your system of equations (e.g., typically FOCs and BCs) and write is as $F(x) = 0$
- Invoke your favorite numerical solver, feeding initial guess and vector X
- Work out the remaining equations, if any

Catch:

- Think of your model as equations in the form $F(x) = 0$, e.g., Euler Equation

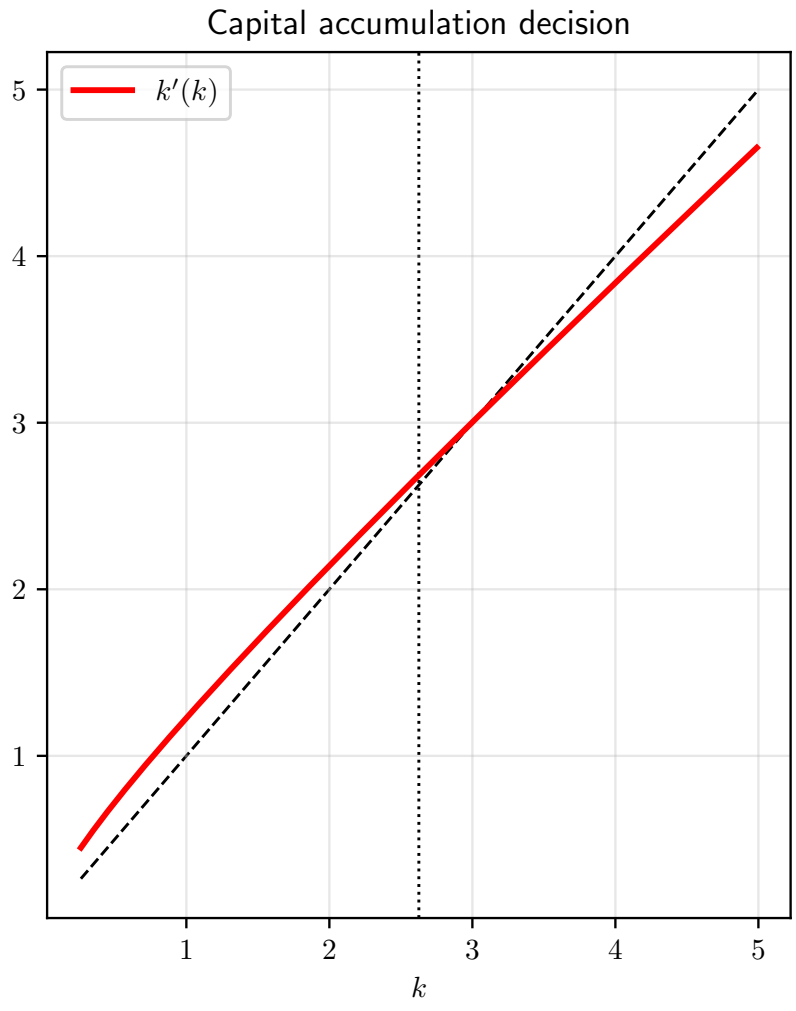
$$c(k) - c(k'(k)) \left[\beta \left[\alpha \big(k'(k)\big)^{\alpha - 1} + 1 - \delta \right] \right]^{-1/\sigma} = 0$$

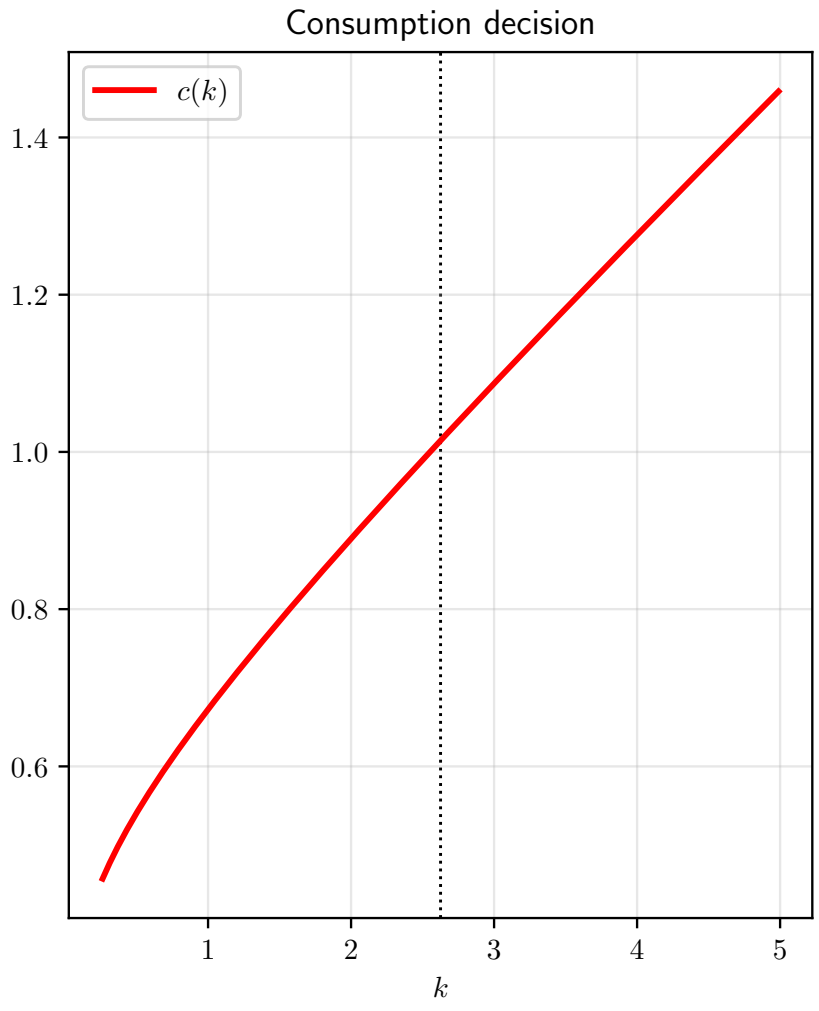
Projection method: example

Sketch of code implementing the projection method

```
import numpy as np
from scipy.optimize import fsolve
# define N, alpha, beta, delta, sigma
K = np.linspace(0.01, 5, num=N)
C0 = 0.4 + 0.35 * K - 0.02 * K**2  # helping solver with good guess
def euler(C0, K, alpha, beta, delta, sigma):
    k1 = K**alpha - C0 + (1-delta) * K
    pc = np.polyfit(K, C0, 1)
    Ctp1 = np.polyval(pc, k1)
    opr = alpha * k1 ** (alpha-1) + 1 - delta
    resid = C0 - Ctp1 * (beta * opr) ** (-1/sigma)
    return resid
C1 = fsolve(euler, C0, args=(K, alpha, beta, delta, sigma), xtol=1e-6)
K_opt = K ** alpha - C1 + (1-delta) * K
```

Projection method: example





Wrap-up

- In general, can choose among VFI, PFI and Projection methods
- Each has pros and cons
- My personal take on this: use VFI
  - Never fails (and you don't have to prove it!!!)
  - Suitable to "exotic" models
  - Be smart with your state variables and keep them in check
- Next time: VFI, PFI and Projection methods with uncertainty
  - Today we discretized a deterministic state space
  - Will need to discretize a stochastic state space: how do we find the "right" probabilities?