Introduction to Projects

Simon Scheidegger
simon.scheidegger@gmail.com
July 12th, 2018
Open Source Macroeconomics Laboratory – BFI/UChicago

Outline

Project 1:
Option Pricing

Project 2:
High-dimensional dynamic programming

**Project 3:**
Discrete State Dynamic Programming

Project 1: Option pricing

1a – Pricing of an European Option

- We consider the calculation of the expected present value of the pay-off of a call option on a stock.

- Let S(t) denote the price of the stock at time t. Consider a call option granting the holder the right to buy the stock at a fixed price K at a fixed time T in the future.

- The current time is t = 0. If at time T the stock price S(T) exceeds the strike price K, the holder exercises the option for a profit of **S(T) − K**

- If, on the other hand, S(T) ≤ K, the option expires worthless.

- The pay-off to the option holder at time T is given by $(S(T) - K)^+ = \max\{0, S(T) - K\}.$

- We denote the expected present value by $\mathsf{E}[e^{-rT}(S(T) - K)^+]$

- The Black-Scholes model describes the evolution of the stock price through the stochastic differential equation.

$$\frac{dS(t)}{S(t)} = r\,dt + \sigma\,dW(t),$$

Long Call
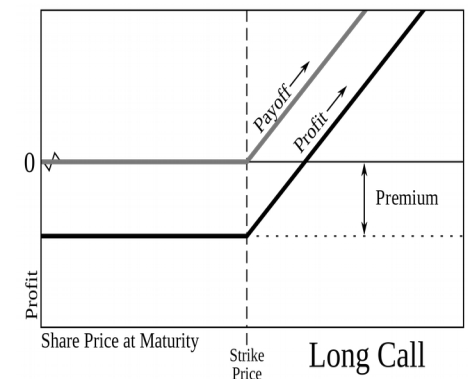
1a – Pricing of an European Option

- W (T) is normally distributed with mean 0 and variance T

- This is also the distribution of $\sqrt{T}Z$ if Z is a standard normal random variable (mean 0, variance 1).

- We may therefore represent the terminal stock price as: $S(T) = S(0) \exp\left([r - \tfrac{1}{2}\sigma^2]T + \sigma\sqrt{T}Z\right)$.

- We see that to draw samples of the terminal stock price S(T) it suffices to have a mechanism for drawing samples from the standard normal distribution.

- We simply assume the ability to produce a sequence Z1 , Z2 , . . . of independent standard normal random variables. Given a mechanism for generating the Zi , we can estimate $\mathsf{E}[e^{-rT}(S(T) - K)^+]$ using the following algorithm:

$$
\begin{aligned}
&\text{for } i = 1, \ldots, n \\
&\quad \text{generate } Z_i \\
&\quad \text{set } S_i(T) = S(0) \exp\left([r - \tfrac{1}{2}\sigma^2]T + \sigma\sqrt{T}Z_i\right) \\
&\quad \text{set } C_i = e^{-rT}(S(T) - K)^+ \\
&\text{set } \hat{C}_n = (C_1 + \cdots + C_n)/n
\end{aligned}
$$

Compile & run

1. OSM2018/day2/Projects/OptionsPricing/serial_option

2. First time compilation:

> g++ BS.cpp -o BS.exec

3. run

> ./BS.exec

→ **Let's have a look at the code together**

Project 1b – Path-dependent option

- The pay-off of a standard European call option is determined by the terminal stock price S(T) and does not otherwise depend on the evolution of S(t) between times 0 and T.

- In estimating $\mathsf{E}[e^{-rT}(S(T) - K)^+],$ we were able to jump directly from time 0 to time T to sample values of S(T).

- Each simulated "path" of the underlying asset thus consists of just the two points S(0) and S(T).

- In valuing more complicated derivative securities using more complicated models of the dynamics of the underlying assets, it is often necessary to simulate paths over multiple intermediate dates and not just at the initial and terminal dates.

- Two considerations may make this necessary:
    - → the pay-off of a derivative security may depend explicitly on the values of underlying assets at multiple dates.

    - → we may not know how to sample transitions of the underlying assets exactly and thus need to divide a time interval [0, T] into smaller subintervals to obtain a more accurate approximation to sampling from the distribution at time T.

Asian Option as an Example

Asian options are options with pay-offs that depend on the average level of the underlying asset. This includes, for example, the pay-off $(\bar{S} - K)^+$

$$\bar{S} = \frac{1}{m} \sum_{j=1}^{m} S(t_j)$$

for some fixed set of dates $0 = t_0 < t_1 < \cdots < t_m = T$, with T the date at which the pay-off is received.

To calculate the expected discounted pay-off, we need to be able to generate **samples of the average S**.

The simplest way to do this is to simulate the path S(t1), . . . , S(tm) and then compute the average along the path. We saw before how to simulate S(T) given S(0); simulating $S(t_{j+1})$ from $S(t_j)$ works the same way:

$$S(t_{j+1}) = S(t_j) \exp\left([r - \tfrac{1}{2}\sigma^2](t_{j+1} - t_j) + \sigma\sqrt{t_{j+1} - t_j}Z_{j+1}\right)$$

Pricing an Asian Option

The following algorithm illustrates the steps in **simulating n paths of m transitions each**. To be explicit, we use Z_{ij} to denote the j-th draw from the normal distribution along the i-th path:

$$
\begin{aligned}
&\text{for } i = 1, \ldots, n \\
&\quad \text{for } j = 1, \ldots, m \\
&\qquad \text{generate } Z_{ij} \\
&\qquad \text{set } S_i(t_j) = S_i(t_{j-1}) \exp\left([r - \tfrac{1}{2}\sigma^2](t_j - t_{j-1}) + \sigma\sqrt{(t_j - t_{j-1})}Z_{ij} \right) \\
&\quad \text{set } \bar{S} = (S_i(t_1) + \cdots + S_i(t_m))/m \\
&\quad \text{set } C_i = e^{-rT}(\bar{S} - K)^+ \\
&\text{set } \hat{C}_n = (C_1 + \cdots + C_n)/n
\end{aligned}
$$

<u>Project 1a – Overall tasks</u>

- Write a makefile instead of compiling the file by hand

- Parallelize the problem with OpenMP
  (look at BS.cpp – take care of the random seeds!).

- **Check the speed-up** for combinations of threads and a variety of **sample points** on one node of MIDWAY (use slurm)

- Ensure that the serial and parallel code return the same results.

- Compute the global maximum over all sample points.

OpenMP

- Parallelize the example with MPI (look at BS.cpp).

- Check the speed-up for combinations of MPI processes and a variety of sample points (on one node on MIDWAY (use slurm).

- Ensure that the serial and parallel return the same results.

MPI

- Compute the global maximum over all sample points.

Hybrid

- Can you parallelize it in hybrid (mixed OpenMP/MPI)?

- If yes, run the resulting code with 2 MPI processes and max. # threads/node each

<u>Project 1a – continued*</u>

- Write a script that dispatches 3 options pricers at once (on slurm – low latency parallelism)
  - → This is done by using a **"chain job"** on slurm.

  - → each of the options should be computed on one node in shared memory mode.
  - → Choose for each of the 3 call/put options a specification. The base configuration is

    double S = 100.0; // Option price
    double K = 100.0; // Strike price
    double r = 0.05;   // Risk-free rate (5%)
    double v = 0.2;   // Volatility of the underlying (20%)
    double T = 1.0;   // One year until expiry

*not mandatory

Tasks 2b

- Adjust pricer from 2a to Asian Options.

- Then repeat the same steps as in 2a.

Project 2: High-dimensional Dynamic Programming
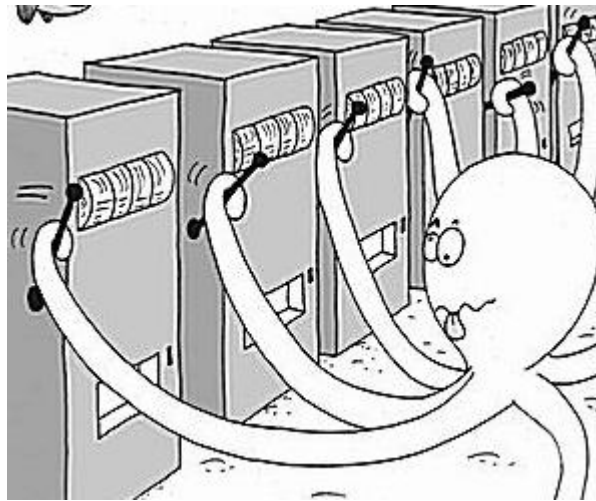
<u>Growth Model & Dynamic Programming & ASG</u>

To demonstrate the capabilities of sparse grids, we consider an **infinite-horizon discrete-time multi-dimensional optimal growth model**
(see, e.g, Cai & Judd (2014), Scheidegger & Bilionis (2017), and references therein).

The model has few parameters and is relatively easy to explain, whereas the dimensionality of the problem can be scaled up in a straightforward but meaningful way

→ state-space depends linearly on the number of **D sectors** considered.

→ there are D sectors with **capital**

$$\mathbf{k}_t = (k_{t,1}, ..., k_{t,D})$$

and elastic **labour supply**

$$\mathbf{l}_t = (l_{t,1}, ..., l_{t,D})$$

Stochastic growth model

The production function of sector **i** at time **t** is $f(k_{t,i}, l_{t,i})$, for $i = 1, ..., D$.

Consumption: $\mathbf{c}_t = (c_{t,1}, ..., c_{t,D})$

Investment of the sectors at time *t:* $\mathbf{I}_t = (I_{t,1}, ..., I_{t,D})$

→ The goal now is to find **optimal consumption** and **labour supply decisions** such that expected total utility over an infinite time horizon is maximized.

Model

$$V_0(\mathbf{k}_0) = \max_{\mathbf{k}_t, \mathbf{I}_t, \mathbf{c}_t, \mathbf{l}_t, \mathbf{\Gamma}_t} \left\{ \sum_{t=0}^{\infty} \beta^t \cdot u(\mathbf{c}_t, \mathbf{l}_t)) \right\},$$

$$s.t.$$

$$k_{t+1,j} = (1 - \delta) \cdot k_{t,j} + I_{t,j} \qquad j = 1, ..., D$$

$$\Gamma_{t,j} = \frac{\zeta}{2} k_{t,j} \left(\frac{I_{t,j}}{k_{t,j}} - \delta \right)^2, \qquad j = 1, ..., D$$

$$\sum_{j=1}^{D} (c_{t,j} + I_{t,j} - \delta \cdot k_{t,j}) = \sum_{j=1}^{D} (f(k_{t,j}, l_{t,j}) - \Gamma_{t,j})$$

Model (II)

Convex adjustment cost of sector j: $\mathbf{\Gamma}_t = (\Gamma_{t,1,}, ..., \Gamma_{t,D})$

Capital depreciation: δ

Discount factor: β

Recursive formulation

$$V(\mathbf{k}) = \max_{\mathbf{I},\mathbf{c},\mathbf{l}} \left(u(c,l) + \beta \ \left\{ V_{next}(k^+) \right\} \right),$$

$$s.t.$$

$$k_j^+ = (1 - \delta) \cdot k_j + I_j \qquad j = 1, ..., D$$

$$\Gamma_j = \frac{\zeta}{2} k_j \left(\frac{I_j}{k_j} - \delta \right)^2, \quad j = 1, ..., D$$

$$\sum_{j=1}^{D} (c_j + I_j - \delta \cdot k_j) = \sum_{j=1}^{D} (f(k_j, l_j) - \Gamma_j)$$

where we indicate the next period's variables with a superscript "+". $\mathbf{k} = (k_1, ..., k_D)$ represents the state vector, $\mathbf{l} = (l_1, ..., l_D)$, $\mathbf{c} = (c_1, ..., c_D)$, and $\mathbf{I} = (I_1, ..., I_D)$ are $3D$ control variables. $\mathbf{k}^+ = (k_1^+, ..., k_D^+)$ is the vector of next period's variables. Today's and tomorrow's states are restricted to the finite range $[\underline{\mathbf{k}}, \overline{\mathbf{k}}]^D$, where the lower edge of the computational domain is given by $\underline{\mathbf{k}} = (\underline{k_1}, ..., \underline{k_D})$, and the upper bound is given by $\overline{\mathbf{k}} = (\overline{k_1}, ..., \overline{k_D})$. Moreover, $\mathbf{c} > 0$ and $\mathbf{l} > 0$ holds component-wise.

Utility function etc.

Productivity:
$$f(k_j, l_j) = A \cdot k_i^{\psi} \cdot l_i^{1-\psi}$$

Utility:
$$u(\mathbf{c}, \mathbf{l}) = \sum_{i=1}^{d} \left[\frac{(c_i/A)^{1-\gamma} - 1}{1 - \gamma} - (1 - \psi)\frac{l_i^{1+\eta} - 1}{1 + \eta} \right]$$

Terminal Value function:
$$V^{\infty}(\mathbf{k}) = u(f(k, \mathbf{e}), \mathbf{e})/(1 - \beta)$$

where $\mathbf{e}$ is the unit vector

Parametrization

Parameter	Value
β	0.8
δ	0.025
ζ	0.5
$\left[\underline{\mathbf{k}}, \overline{\mathbf{k}}\right]^{D}$	$[0.2, 3.0]^{D}$
ψ	0.36
A	$(1 - \beta)/(\psi \cdot \beta)$
γ	2
η	1

Value function iteration

$$V(\mathbf{k}) = \max_{\mathbf{I,c,l}} \left(u(c,l) + \beta \; \left\{ V_{next}(k^+) \right\} \right),$$

$$s.t.$$

$$k_j^+ = (1-\delta) \cdot k_j + I_j \qquad , \qquad j = 1, ..., D$$

$$\Gamma_j = \frac{\zeta}{2} k_j \left(\frac{I_j}{k_j} - \delta \right)^2, \qquad j = 1, ..., D$$

$$\sum_{j=1}^{D} (c_j + I_j - \delta \cdot k_j) = \sum_{j=1}^{D} (f(k_j, l_j) - \Gamma_j)$$

State k: sparse grid coordinates

V_{next} : sparse grid interpolator from the previous iteration step

Solve this optimization problem at every point in the sparse grid!

Attention: Take care of the econ domain/ sparse grid domain

Convergence measures
(due to contraction mapping)

Average error:

$$e^s = \frac{1}{N} \sum_{i=1}^{N} |V^s(\mathbf{x^i}) - V^{s-1}(\mathbf{x^i})|$$

Max. error:

$$a^s = \max_{i=1,N} |V^s(\mathbf{x^i}) - V^{s-1}(\mathbf{x^i})|$$

Setup of Code

```
cleanup.sh              ipopt_wrapper.py           parameters.py
econ.py                 main.py                    postprocessing.py
interpolation_iter.py   nonlinear_solver_initial.py TasmanianSG.py
interpolation.py        nonlinear_solver_iterate.py test_initial_sg.py
```

main.py: driver routine

econ.py: contains production function, utility,...

nonlinear_solver_initial/iterate.py: interface SG ↔ IPOPT (optimizer).

ipopt_wrapper.py: specifies the optimization problem
(objective function,...).

interpolation.py: interface value function iteration ↔ sparse grid.

postprocessing.py: auxiliary routines, e.g., to compute the error.

Run the Growth model code

→ Model implemented in Python (TASMANIAN)

→ Optimizer used: IPOPT & PYIPOPT (python interface)

→ OSM2018/SparseGrid/SparseGridCode/growth_model

→ run serially with

>python main.py

Code snippet – interpolation.py

```python
import TasmanianSG
import numpy as np
from parameters import *
import nonlinear_solver_initial as solver

#====================================================================

def sparse_grid(n_agents, iDepth):

    grid  = TasmanianSG.TasmanianSparseGrid()

    k_range=np.array([k_bar, k_up])

    ranges=np.empty((n_agents, 2))


    for i in range(n_agents):
        ranges[i]=k_range

    iDim=n_agents

    grid.makeLocalPolynomialGrid(iDim, iOut, iDepth, which_basis, "localp")
    grid.setDomainTransform(ranges)

    aPoints=grid.getPoints()
    iNumP1=aPoints.shape[0]
    aVals=np.empty([iNumP1, 1])

    file=open("comparison0.txt", 'w')
    for iI in range(iNumP1):
        aVals[iI]=solver.initial(aPoints[iI], n_agents)[0]
        v=aVals[iI]*np.ones((1,1))
        to_print=np.hstack((aPoints[iI].reshape(1,n_agents), v))
        np.savetxt(file, to_print, fmt='%2.16f')

    file.close()
    grid.loadNeededPoints(aVals)

    f=open("grid.txt", 'w')
    np.savetxt(f, aPoints, fmt='% 2.16f')
    f.close()

    return grid
#====================================================================
```

The function evaluation at ever point in the sparse grid are independent.

Project 2 – Tasks

- Parallelize the non-stochastic model in the **non-adaptive Sparse grid** case with MPI

- Parallelize the **adaptive Sparse grid** with MPI

- **Check the speed-up** for combinations of MPI processes and a variety of **the SG level** on one node of **MIDWAY** (use slurm)

- Ensure that the serial and parallel return the same results.

MPI

- Parallelize the **stochastic model** in the **non-adaptive Sparse grid** case with MPI (split the MPI communicator – one communicator per discrete shock).

- Parallelize the **adaptive Sparse grid** with MPI.

- Check the speed-up for combinations of MPI processes and avariety of **the SG level** on one node of **MIDWAY** (use slurm)

- Ensure that the serial and parallel return the same results.

Splitting the MPI communicator

- Compute the global maximum over all sample points (a reduction operation).

3. Project: Discrete State Dynamic Programming

Innocence could be considered a discrete state of mind.

— John Shirley —

AZ QUOTES

Setup:

- We consider a infinite-horizon discrete-state dynamic programming problem similar to the example described in Judd (1998) – Numerical Methods in Economics, 12.1.3.

- Here, we have a stochastic growth problem with a stochastic component to the output.
  The rate of consumption is denoted by **c**, and **f (k,Θ)** is the net-of-depreciation output when capital stock is **k**, implying that capital evolves according to $k_{t+1} = k_t + f(k_t, \Theta_t) - c_t$

- We assume a time-separable utility function, and **β < 1**.

The optimal growth problem yields:

$$V(k, \Theta) = \max_{c_t} \mathbb{E} \sum_{t=0}^{\infty} \beta^t u(c_t)$$

$$\text{s.t.} \quad k_{t+1} = f(k, \Theta_t) - c_t$$

$$\Theta_{t+1} = g(\Theta_t)$$

Dynamic Programming

$$V_{new}(k, \Theta) = \max_c \left(u(c) + \beta \mathbb{E}\{V_{old}(k_{next}, \Theta_{next})\} \right)$$

$$\text{s.t.} \quad k_{next} = f(k, \Theta_{next}) - c$$

$$\Theta_{next} = g(\Theta)$$

States of the model:
- k : today's capital stock
- Θ: today's productivity state

Choices of the model:
- k$_{next}$

→ k, k$_{next}$, Θ and Θ$_{next}$ are limited to a finite number of values

Specifications

Gross output:

$$f(k,\Theta) = k + \Theta(1 - \beta)k^\alpha /(\alpha\beta)$$

Utility function:

$$u(c) = c^{1-\gamma} /1-\gamma$$

choose γ > 0 in order for u(c) to be concave.

Parameters

$\alpha = 0.25$

$\beta = 0.9$

$\gamma = 5.0$

Range for capital stock: $[k_{min}, k_{max}] = [0.85, 1.15]$

Discrete number of capital stocks: n_k

Discretiztion of capital axis: kappa $= (k_{max} - k_{min})/(n_k - 1)$

Productivity states: $n_\Theta = 5$

$\Theta_{grid} = \{0.9, 0.95, 1.0, 1.05, 1.1\}$

Transition Matrix: $\pi(i,j) = 1.0/n_\Theta = 1/5$ (could be any other specification)

Computation

Since we solve a discrete-state DP problem,
the value function is a 2-dimensional matrix: **Val(k,Θ)**

→ We create **two Matrices Val$_{old}$ and Val$_{new}$** and update them with value
function iteration.

→ In our problem, the control k$_{next}$ is discrete. Hence, the maximization
problem reduces to computing a list of values implied by all possible
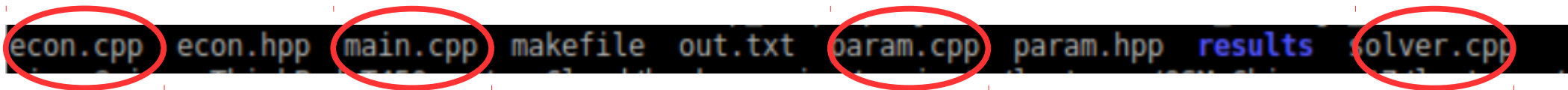choices, and taking the maximum (no optimizer needed).

Do i = 1, n$_{\Theta}$
  Do j = 1, kappa
    Val$_{new}$ (i,j) = max(...)
  End do
End do
Val$_{old}$ = Val$_{new}$

**Spot the huge
potential for parallelization**

Setup of Code

1. cd OSM2018/day2/Projects/DynamicProgramming/serial_DP

The folder contains the following important files:

main.cpp: the "driver routine" of the code.

solver.cpp: in this file, the actual value function iteration happens (→ parallelization).

econ.cpp: contains economic functions like utility, output.

parameters.cpp: specifies the parameters of the problem.

makefile: compiles the code

solver.cpp

```cpp
for (int itheta=0; itheta<ntheta; itheta++) {

    /*
     Given the theta state, we now determine the new values and optimal policies corresponding to each
     capital state.
    */

    for (int ik=0; ik<nk; ik++) {

        // Compute the consumption quantities implied by each policy choice
        c=f(kgrid(ik), thetagrid(itheta))-kgrid;

        // Compute the list of values implied implied by each policy choice
        temp=util(c) + beta*ValOld*p(thetagrid(itheta));

        /* Take the max of temp and store its location.
         The max is the new value corresponding to (ik, itheta).
         The location corresponds to the index of the optimal policy choice in kgrid.
        */
        ValNew(ik, itheta)=temp.maxCoeff(&maxIndex);

        Policy(ik, itheta)=kgrid(maxIndex);

    }
}
```

loops to worry about

parameters.cpp

```cpp
// Preference Parameters
double alpha = 0.25;
double gamm  = 5.0;
double beta  = 0.9;

// Capital Stock Interval
double kmin = 0.85; //minimum capital
double kmax = 1.15; //max. capital

// Choose nk, the number of capital stocks, and kappa, the discretization level
int nk   =  31;
double kappa  =  (kmax-kmin)/(nk-1.0);

// Number of theta states and the minimum and maximum theta values
int ntheta = 5;
double theta_min  = 1-10.0/100;
double theta_step = 5.0/100;

// Capital and Theta grids
ArrayXd kgrid=fill_kgrid();
ArrayXd thetagrid=fill_thetagrid();

// Number of Iterations
int numstart = 0; // start iterating at this timestep
int Numits = 3;   // stop  iterating at this timestep

// Maximum difference between succesive Value Function Iterates (initialized to 0)
double errmax = 0;

// Number of Columns for output
int nout = 3;

// Frequency with which data will be printed, e.g. 1=every timestep, 10=every ten iteration steps
int datafreq = 1;
```

Project 3 – Tasks

- Parallelize the problem with OpenMP
  (look at solver.cpp).

- **Check the speed-up** for combinations of threads and a variety of **the discretization level**
  on one node of **MIDWAY** (use slurm)

- Ensure that the serial and parallel return the same results.

OpenMP

- Parallelize the example with MPI (look at solver.cpp).

- Check the speed-up for combinations of MPI processes and a variety of sample points (set in parameters.cpp) on one node on MIDWAY.

- Ensure that the serial and parallel return the same results.

MPI

- Compute the global maximum over all sample points.

- Can you parallelize it in hybrid (mixed OpenMP/MPI)?

Hybrid