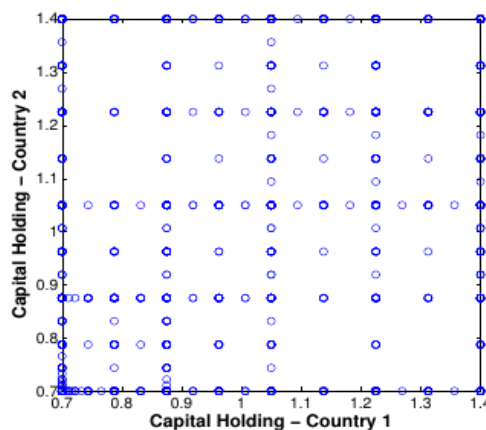


# Dynamic Programming with Sparse Grids

Simon Scheidegger  
simon.scheidegger@gmail.com  
July 10<sup>th</sup>, 2018

Open Source Macroeconomics Laboratory – BFI/UChicago



# Growth Model & Dynamic Programming & ASG

To demonstrate the capabilities of sparse grids, we consider an **infinite-horizon discrete-time multi-dimensional optimal growth model** (see, e.g., Scheidegger & Bilonis (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**

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

and elastic **labour supply**

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

# Stochastic growth model

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

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

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

- 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

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

*s.t.*

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

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

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

## Model (II)

Convex adjustment cost of sector  $j$ :  $\Gamma_t = (\Gamma_{t,1}, \dots, \Gamma_{t,D})$

Capital depreciation:  $\delta$

Discount factor:  $\beta$

# 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.*

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

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

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

where we indicate the next period's variables with a superscript “+”.  $\mathbf{k} = (k_1, \dots, k_D)$  represents the state vector,  $\mathbf{l} = (l_1, \dots, l_D)$ ,  $\mathbf{c} = (c_1, \dots, c_D)$ , and  $\mathbf{I} = (I_1, \dots, I_D)$  are  $3D$  control variables.  $\mathbf{k}^+ = (k_1^+, \dots, k_D^+)$  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  $\underline{\mathbf{k}} = (\underline{k}_1, \dots, \underline{k}_D)$ , and the upper bound is given by  $\overline{\mathbf{k}} = (\overline{k}_1, \dots, \overline{k}_D)$ . 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
$\beta$	0.8
$\delta$	0.025
$\zeta$	0.5
$[\underline{\mathbf{k}}, \overline{\mathbf{k}}]^D$	$[0.2, 3.0]^D$
$\psi$	0.36
$A$	$(1 - \beta)/(\psi \cdot \beta)$
$\gamma$	2
$\eta$	1



# Value function iteration

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

*s.t.*

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

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

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

**State  $\mathbf{k}$ :** sparse grid coordinates

$V_{\text{next}}$  : 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  $\leftrightarrow$  IPOPT (optimizer).

**ipopt\_wrapper.py**: specifies the optimization problem  
(objective function,...).

**interpolation.py**: interface value function iteration  $\leftrightarrow$  sparse grid.

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

# Code snippet – main.py

```
#=====
# Start with Value Function Iteration

# terminal value function
valnew=TasmanianSG.TasmanianSparseGrid()
if (numstart==0):
    valnew=interpol.sparse_grid(n_agents, iDepth)
    valnew.write("valnew_1." + str(numstart) + ".txt") #write file to disk for restart

# value function during iteration
else:
    valnew.read("valnew_1." + str(numstart) + ".txt") #write file to disk for restart

valold=TasmanianSG.TasmanianSparseGrid()
valold=valnew

for i in range(numstart, numits):
    valnew=TasmanianSG.TasmanianSparseGrid()
    valnew=interpol_iter.sparse_grid_iter(n_agents, iDepth, valold)
    valold=TasmanianSG.TasmanianSparseGrid()
    valold=valnew
    valnew.write("valnew_1." + str(i+1) + ".txt")

#=====
print "=====
print " "
print " Computation of a growth model of dimension ", n_agents , " finished after ", numits, " steps"
print " "
print "=====
#=====

# compute errors
avg_err=post.ls_error(n_agents, numstart, numits, No_samples)

#=====
print "=====
print " "
print " Errors are computed -- see error.txt"
print " "
print "=====
#=====
```

# Code snippet – parameters.py

```
#=====
# Depth of "Classical" Sparse grid
iDepth=1
iOut=1      # how many outputs
which_basis = 1 #linear basis function (2: quadratic local basis)

# control of iterations
numstart = 0  # which is iteration to start (numstart = 0: start from scratch, number=/0: restart)
numits = 10   # which is the iteration to end

# How many random points for computing the errors
No_samples = 1000

#=====

# Model Paramters

n_agents=2 # number of continuous dimensions of the model

beta=0.8
rho=0.95
zeta=0.5
psi=0.36
gamma=2.0
delta=0.025
eta=1
big_A=(1.0-beta)/(psi*beta)

# Ranges For States
range_cube=1 # range of [0..1]^d in 1D
k_bar=0.2
k_up=3.0

# Ranges for Controls
c_bar=1e-2
c_up=10000.0

l_bar=1e-2
l_up=1.0

inv_bar=1e-2
inv_up=10000.0
```

# Code snippet – econ.py

```
#=====
#utility function u(c,l)

def utility(cons=[], lab=[]):
    sum_util=0.0
    n=len(cons)
    for i in range(n):
        nom1=(cons[i]/big_A)**(1.0-gamma) -1.0
        den1=1.0-gamma

        nom2=(1.0-psi)*((lab[i]**(1.0+eta)) -1.0)
        den2=1.0+eta

        sum_util+=(nom1/den1 - nom2/den2)

    util=sum_util

    return util

#=====
# output_f

def output_f(kap=[], lab=[]):
    fun_val = big_A*(kap**psi)*(lab**(1.0 - psi))
    return fun_val
```

# Code snippet – ipopt\_wrapper.py

```
#=====
# Objective Function to start VFI (in our case, the value function)

def EV_F(X, k_init, n_agents):

    # Extract Variables
    cons=X[0:n_agents]
    lab=X[n_agents:2*n_agents]
    inv=X[2*n_agents:3*n_agents]

    knext= (1-delta)*k_init + inv
    # Compute Value Function

    VT_sum=utility(cons, lab) + beta*V_INFINITY(knext)

    return VT_sum

# V infinity
def V_INFINITY(k=[]):
    e=np.ones(len(k))
    c=output_f(k,e)
    v_infinity=utility(c,e)/(1-beta)
    return v_infinity

#=====
# Objective Function during VFI (note - we need to interpolate on an "old" sparse grid)

def EV_F_ITER(X, k_init, n_agents, grid):

    # Extract Variables
    cons=X[0:n_agents]
    lab=X[n_agents:2*n_agents]
    inv=X[2*n_agents:3*n_agents]

    knext= (1-delta)*k_init + inv

    # Compute Value Function

    VT_sum=utility(cons, lab) + beta*grid.evaluate(knext)

    return VT_sum

#=====
```

# Run the Growth model code

- Model implemented in Python (TASMANIAN)
- Optimizer used: IPOPT & PYIPOPT (python interface)
- OSM2018/day1\_SparseGrid/SparseGridCode/growth\_model
- run with

**>python main.py**





# Installed software on Midway

- 1) PYIPOPT (<https://github.com/xuy/pyipopt>)  
→ Python Interface to IPOPT

An example is given here:

```
> cd OSM2018/day1_SparseGrid/SparseGridCode/pyipopt_midway/pyipopt/examples  
> python hs071.py
```

- 2) TASMANIAN (<http://tasmanian.ornl.gov/>)

```
>cd OSM2018/day1_SparseGrid/SparseGridCode/TasmanianSparseGrids/InterfacePython/  
>python example.py
```