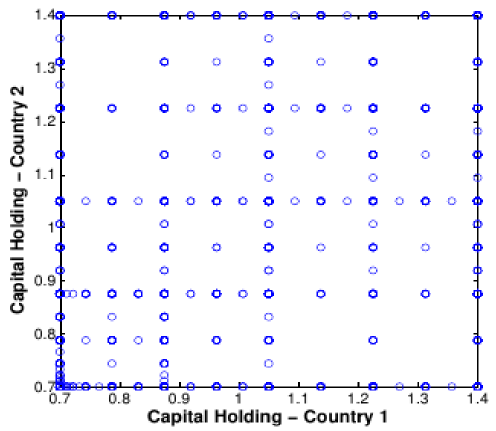
Solving Dynamic Models with Sparse Grids

Simon Scheidegger
simon.scheidegger@unil.ch
January 18th, 2019

Cowles Foundation – Yale University



Today’s Roadmap

I. Reminder – Dynamic Programming (DP)

II. A growth model solved by DP and Sparse Grids

III. Parallel Sparse Grid Dynamic Programming

IV. Time Iteration and SGs (applied to the Ramsey Model)

Recall: Infinite-Horizon Dynamic Programming

e.g. Stokey, Lucas & Prescott (1989), Judd (1998), ...

Want to choose an infinite sequence of "controls" $\{u_s\}_{s=0}^{\infty}$ to maximize

$$\sum_{t=0}^{\infty} \beta^t r\left(x_t, u_t\right) \quad \text{s.t.} \quad x_{t+1} = g(x_t, u_t) \qquad \beta \in (0,1)$$

(Discrete time) Dynamic programming seeks a **time-invariant policy function *h*** mapping the state x_t into the control u_t, such that the sequence $\{u_s\}_{s=0}^{\infty}$ generated by iterating

$$u_t = h\left(x_t\right)$$

$$x_{t+1} = g\left(x_t, u_t\right)$$

starting from an initial condition solves the original problem.

r in the economic context: often a so-called `utility function'.
r concave: reflects the notion "more is better"; marginal benefit tends to zero.

Recall: Infinite-Horizon Dynamic Programming

To find the policy function *h*, we need to know another function (`**Value Function**') that expresses optimal value of the original problem

$$V\left(x_0\right) = \max_{\{u_s\}_{s=0}^{\infty}} \sum_{t=0}^{\infty} \beta^t r\left(x_t, u_t\right)$$

→ Task: solve jointly for $V(x), h(x)$ that are linked by the **Bellman equation**

$$V\left(x\right) = \max_{u}\{r\left(x, u\right) + \beta V\left[g\left(x, u\right)\right]\} \qquad \text{(A)}$$

→ The maximizer of (A) is a policy function *h(x)* that satisfies

$$V\left(x\right) = r\left[x, h\left(x\right)\right] + \beta V\{g\left[x, h\left(x\right)\right]\}$$

Value Function Iteration

The solution is approached in the limit as $j \rightarrow \infty$ by iterations on at every coordinate of the discretized grid.

$$V_{j+1}\underline{(x)} = \max_{u}\{r(x,u) + \beta \underline{V_j(\tilde{x})}\}$$

s.t.

$$\tilde{x} = g(x,u)$$

x: grid point, describes your system. State-space potentially **high-dimensional.**

`old solution': high-dimensional function, approximated by sparse grid Interpolation method on which we Interpolate.

Use-case for (adaptive) sparse grids

Example: Infinite-Horizon Stochastic DP

If uncertainty is present, previous Bellman equation can be re-written as

$$V(x) = \max_{u}\{r(x,u) + \beta E[V[g(x,u,\epsilon)]\,|x]\}$$

s.t.

$$x_{t+1} = g(x_t, u_t, \epsilon_{t+1})$$

The solution is approached by iterations on

$$V_{j+1}(x) = \max_{u}\{r(x,u) + \beta E[V_j[g(x,u,\epsilon)]\,|x]\}$$

Note: If we have discrete shocks, we may have to carry around multiple sparse grids that need to be updated!

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

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

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

$$s.t.$$

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

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

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

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. $\mathbf{k}^{+} = \left(k_1^{+}, ..., k_D^{+}\right)$ 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:

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

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:

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

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

Parametrization

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

Map to sparse grid

Value function iteration

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

$$s.t.$$

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

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

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

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

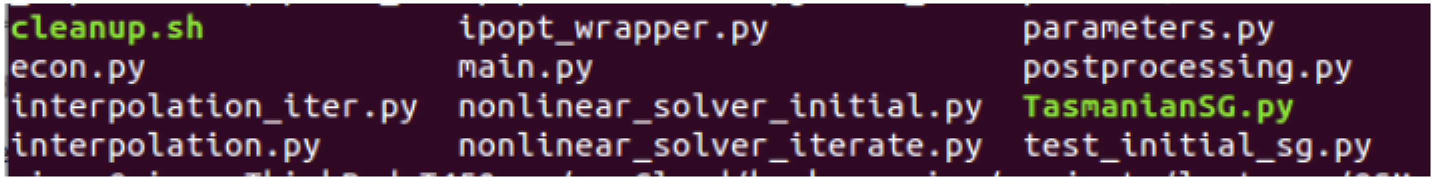
Convergence measures (due to contraction mapping)

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

Go here: global_solution_yale19/Lecture_2/SparseGridCode/growth_model/serial



```
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,...

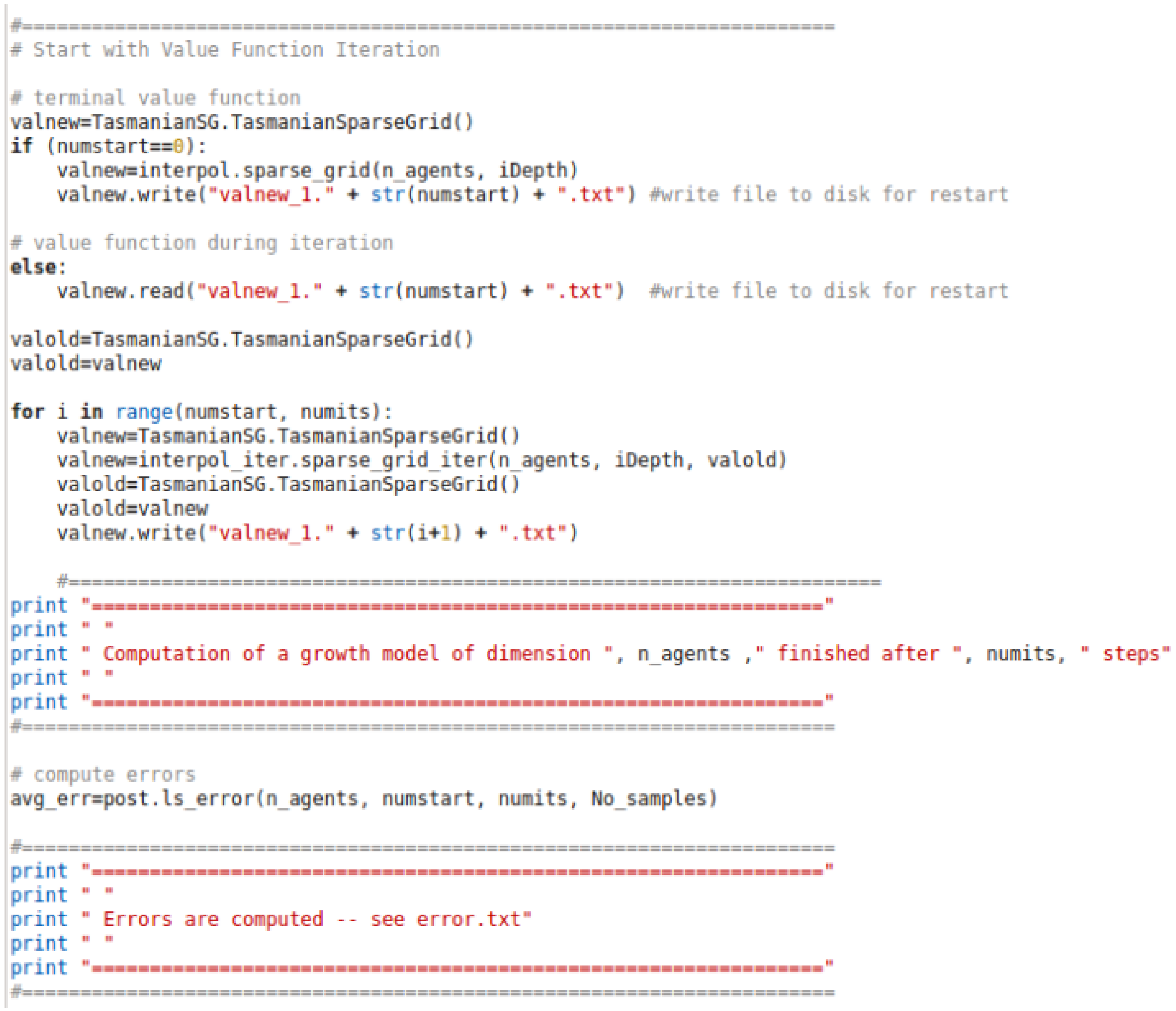
nonlinear_solver_initial/iterate.py: interface SG ↔ IPOPT (optimizer).

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

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

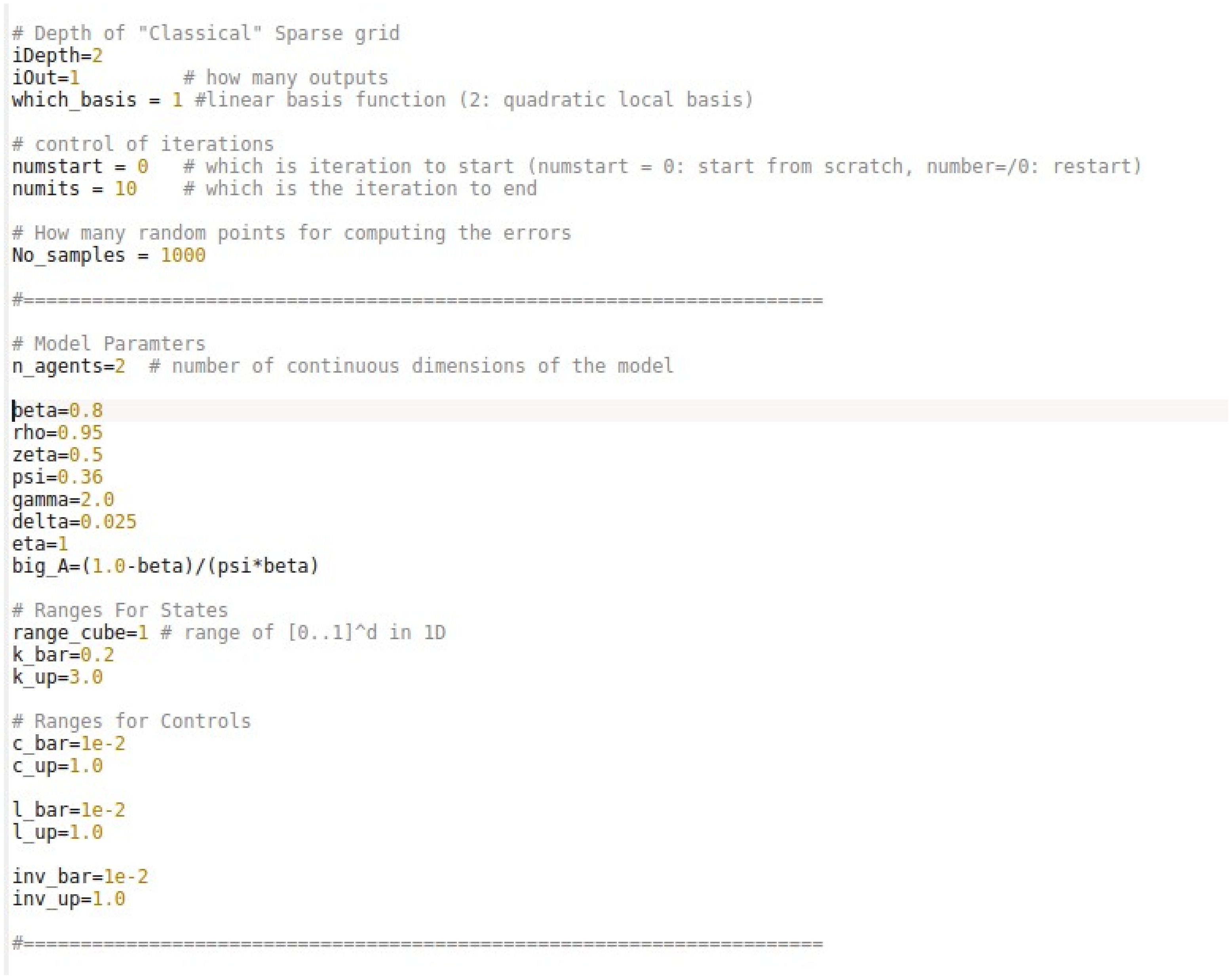
postprocessing.py: auxiliary routines, e.g., to compute the error.

Code snippet – main.py



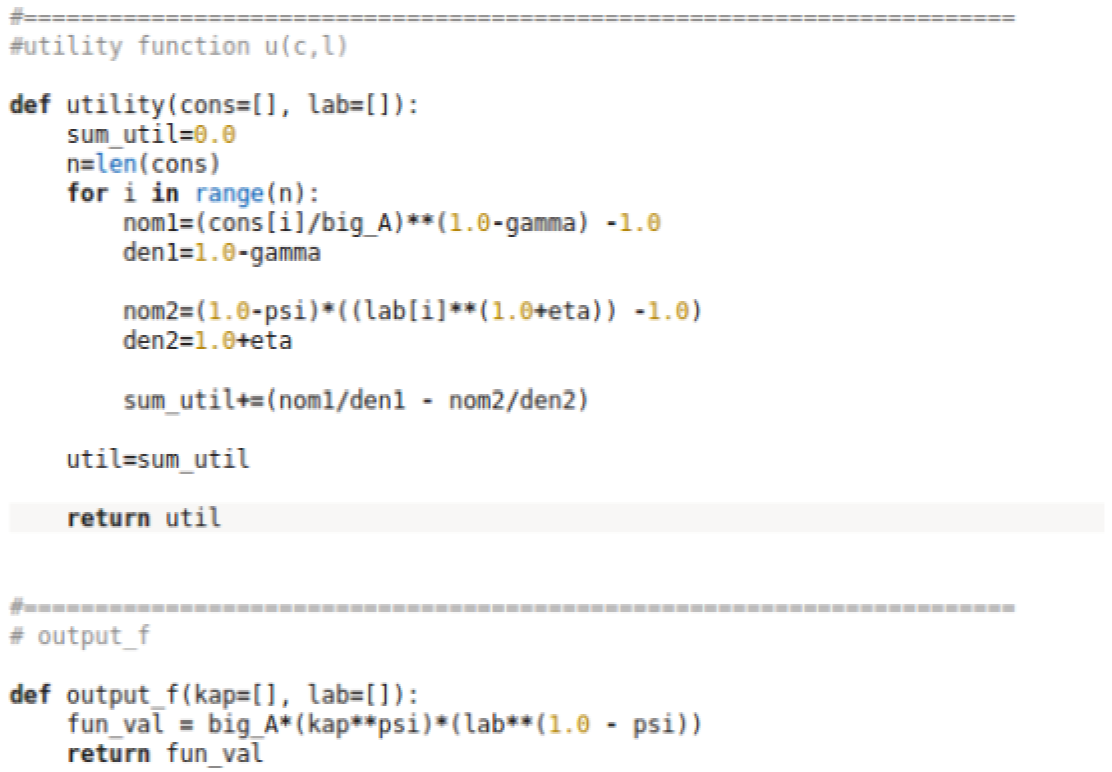
```
#======================================================================
# Start with Value Function Iteration

# 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



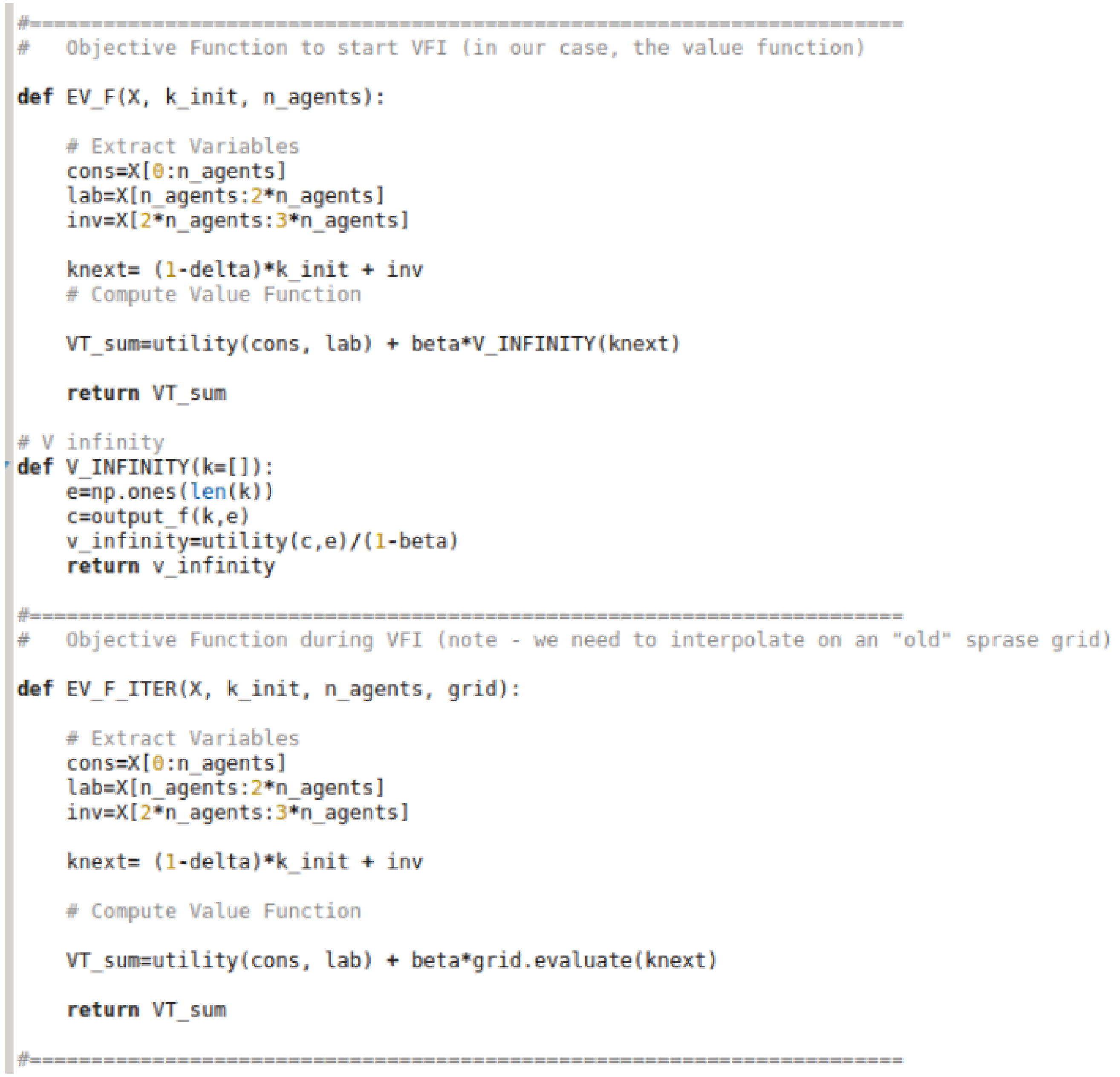
```
# Depth of "Classical" Sparse grid
iDepth=2
iOut=1          # how many outputs
which_basis = 1 #linear basis function (2: quadratic local basis)

# 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=1.0

l_bar=1e-2
l_up=1.0

inv_bar=1e-2
inv_up=1.0

#=======================================================================
```

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" sprase 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)

→ global_solution_yale19/Lecture_2/SparseGridCode/growth_model/serial_growth

→ run with

```
>python main.py
```

Recall

On Yale’s HPC cluster GRACE (ssh -X NETID@grace.hpc.yale.edu)

```
>cd ~
>vi .bashrc
```

→ add the following lines to the .bashrc

```
module load MPI/OpenMPI/2.1.1-intel15
module load Langs/Python/2.7.15-anaconda
module load Libs/MPI4PY
```

Installed software on “GRACE”

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

An example is given here:

```
> cd global_solution_yale19/Lecture_2/SparseGridCode/pyipopt_midway/pyipopt/examples
> python hs071.py
```

2) TASMANIAN (http://tasmanian.ornl.gov/)

```
>cd global_solution_yale19/Lecture_2/SparseGridCode/TasmanianSparseGrids/InterfacePython/
>python example.py
```

A stochastic growth model

→ Model with stochastic production

$$f(k_i, l_i, \theta_i) \;=\; \theta_i A k_i^{\psi} l_i^{1-\psi}$$

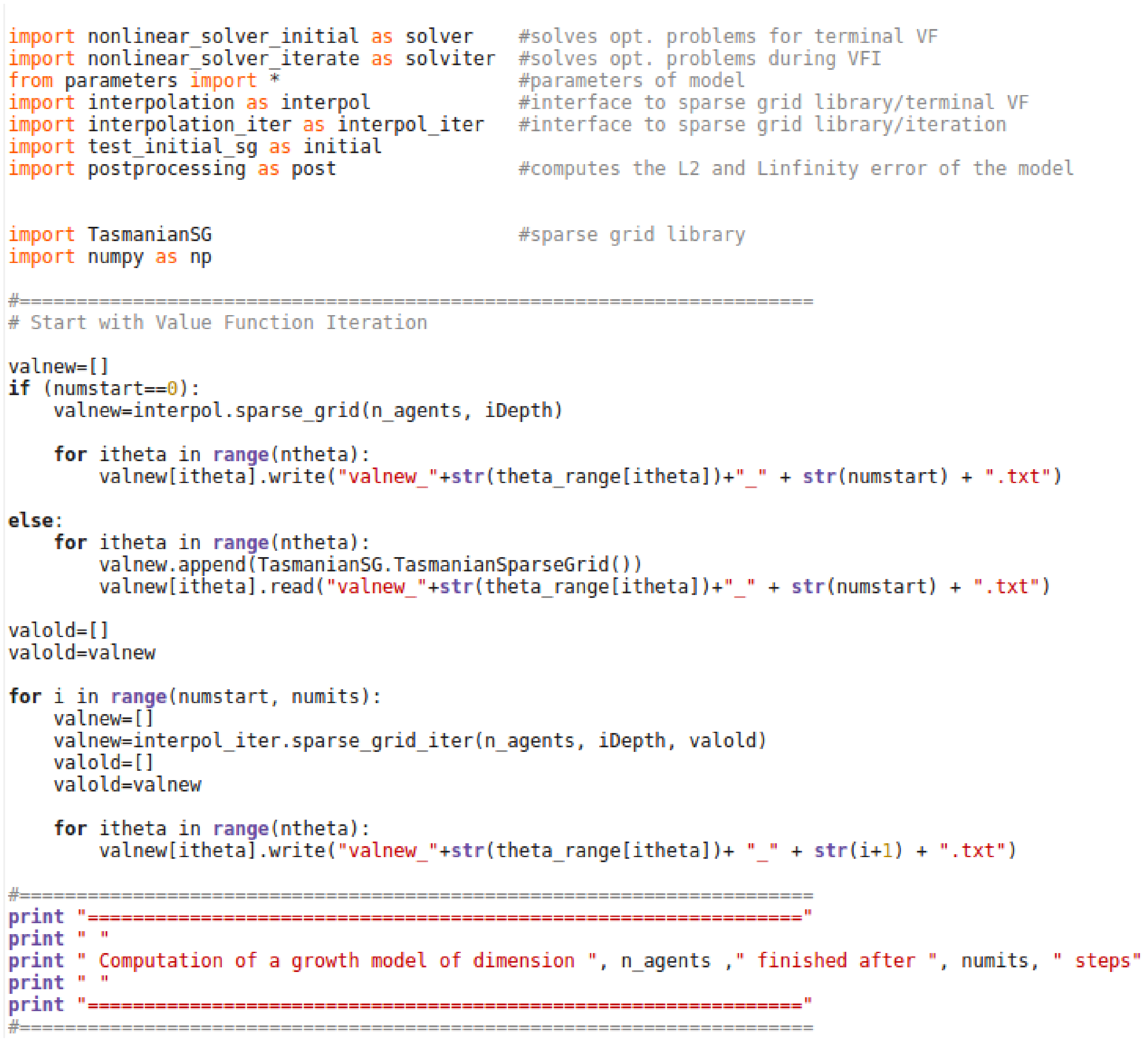
→ Here we assume 5 possible values of
Θ_i = {0.9, 0.95, 1.00, 1.05, 1.10}

→ for simplicity, we assume **Π(*,*) = 1/5** (no Markov chain)

→ solve

$$V_t(k,\theta) = \max_{c,l,I} u(c,l) + \beta \mathbb{E}\left\{ V_{t+1}(k^+, \theta^+) \mid \theta \right\}$$

Code snippet – main.py



```
import nonlinear_solver_initial as solver     #solves opt. problems for terminal VF
import nonlinear_solver_iterate as solviter   #solves opt. problems during VFI
from parameters import *                      #parameters of model
import interpolation as interpol              #interface to sparse grid library/terminal VF
import interpolation_iter as interpol_iter    #interface to sparse grid library/iteration
import test_initial_sg as initial
import postprocessing as post                 #computes the L2 and Linfinity error of the model


import TasmanianSG                            #sparse grid library
import numpy as np

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

valnew=[]
if (numstart==0):
    valnew=interpol.sparse_grid(n_agents, iDepth)

    for itheta in range(ntheta):
        valnew[itheta].write("valnew_"+str(theta_range[itheta])+"_" + str(numstart) + ".txt")

else:
    for itheta in range(ntheta):
        valnew.append(TasmanianSG.TasmanianSparseGrid())
        valnew[itheta].read("valnew_"+str(theta_range[itheta])+"_" + str(numstart) + ".txt")

valold=[]
valold=valnew

for i in range(numstart, numits):
    valnew=[]
    valnew=interpol_iter.sparse_grid_iter(n_agents, iDepth, valold)
    valold=[]
    valold=valnew

    for itheta in range(ntheta):
        valnew[itheta].write("valnew_"+str(theta_range[itheta])+ "_" + str(i+1) + ".txt")

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

Code snippet – IPOPT_wrapper.py

```python
#=======================================================================
#   Objective Function during VFI (note - we need to interpolate on an "old" sprase grid)
def EV_F_ITER(X, k_init, theta_init, n_agents, grid_list):

    # 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 E[V(next, theta)]
    exp_v=0.0

    for itheta in range(ntheta):
        theta_next=theta_range[itheta]
        exp_v+=prob(theta_init, theta_next)*grid_list[itheta].evaluate(knext)


    # Compute Value Function
    VT_sum=utility(cons, lab) + beta*exp_v

    return VT_sum
```

Run the Growth model code

→Model implemented in Python (TASMANIAN)

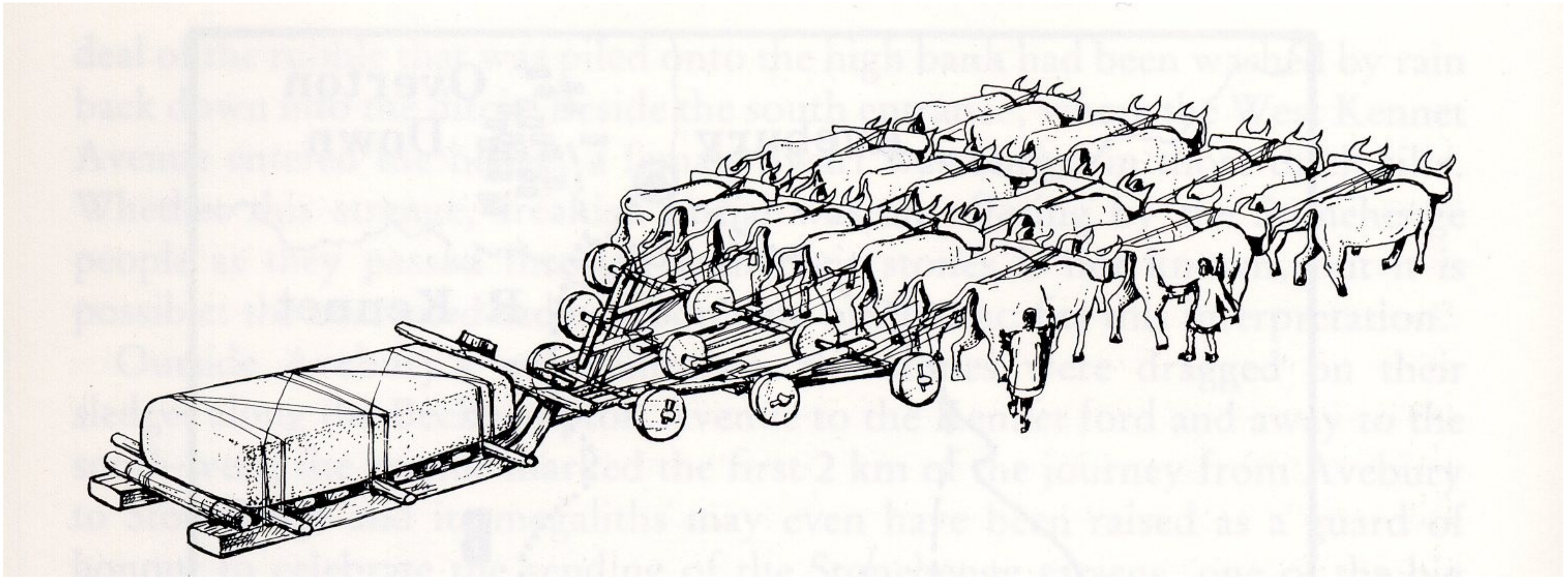
→ Optimizer used: IPOPT & PYIPOPT (python interface)

→ global_solution_yale19/Lecture_2/SparseGridCode/growth_model/serial_stochastic

→ run with

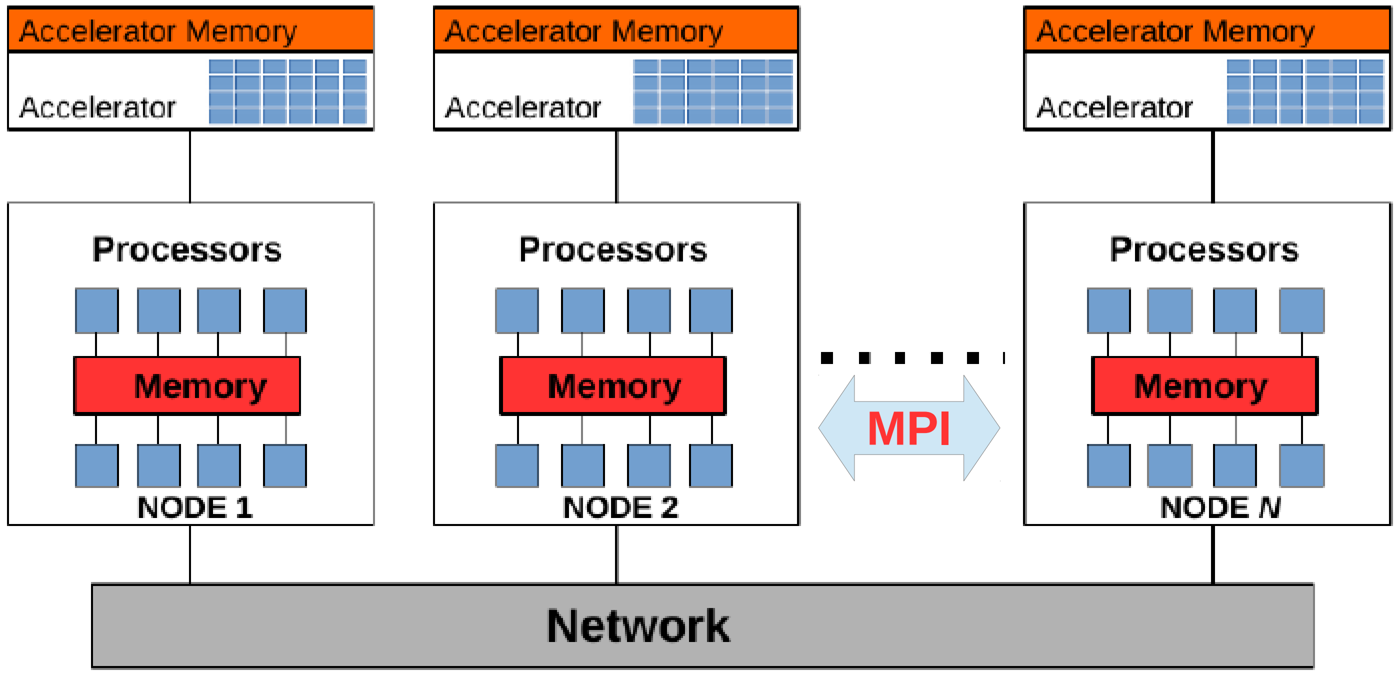
```
>python main.py
```

“To pull a bigger wagon, it is easier to add more oxen than to grow a gigantic ox” (Skjellum et al. 1999)



“To pull a bigger wagon, it is easier to add more oxen than to grow a gigantic ox” (Skjellum et al. 1999)

Recall: today's HPC systems

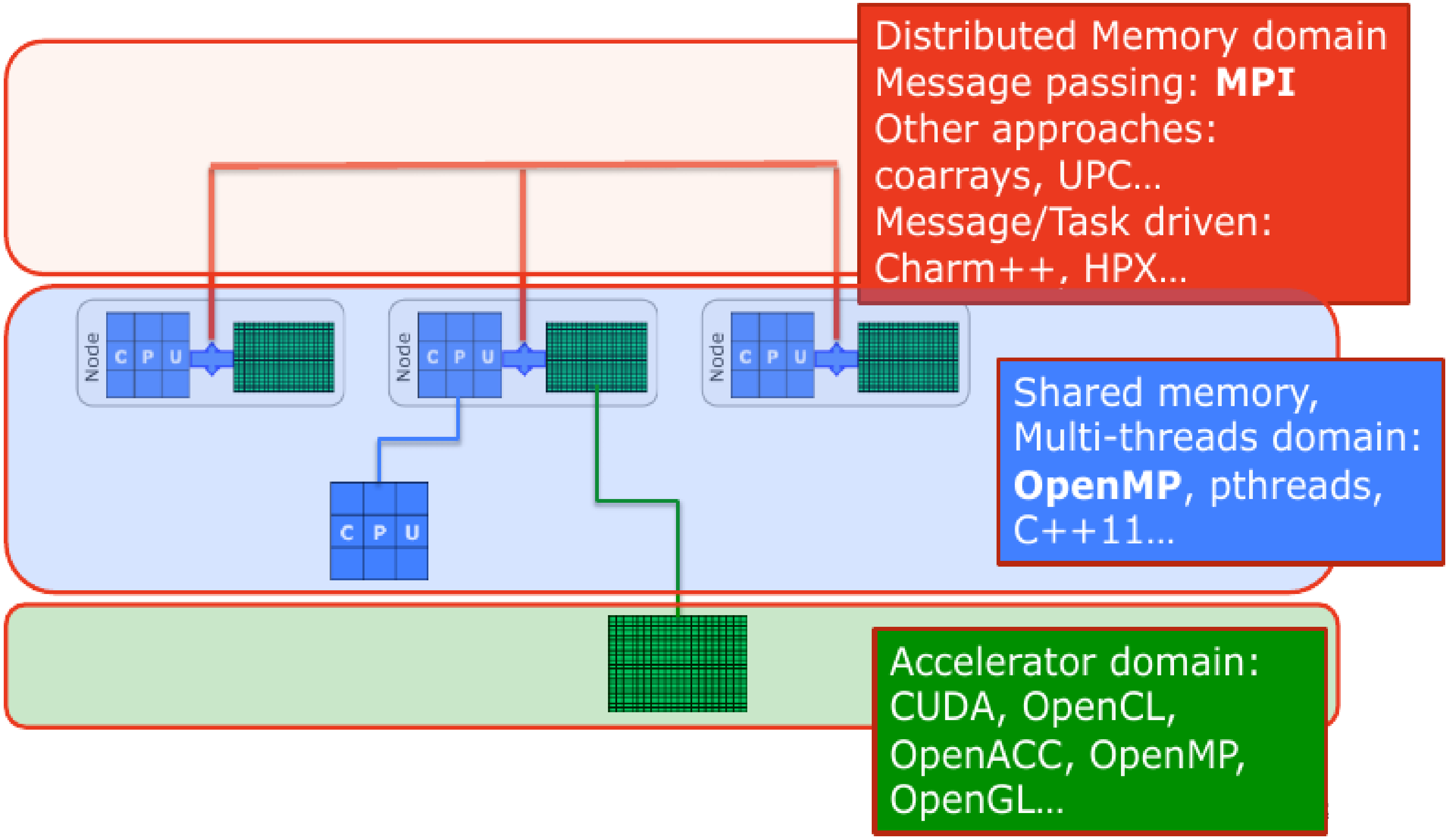


- Ever since parallel computers hit the HPC market, there was an intense discussion about what should be an appropriate programming model for them.

- Message passing is required if a parallel computer is of the distributed memory type, i.e. if there is no way for one processor to directly access the address space of another.

- **The use of explicit message passing (MP), i.e., communication between processes, is surely the most tedious and complicated but also the most flexible parallelization method.**

Overall picture of programming models



Distributed Memory domain
Message passing: **MPI**
Other approaches:
coarrays, UPC...
Message/Task driven:
Charm++, HPX...

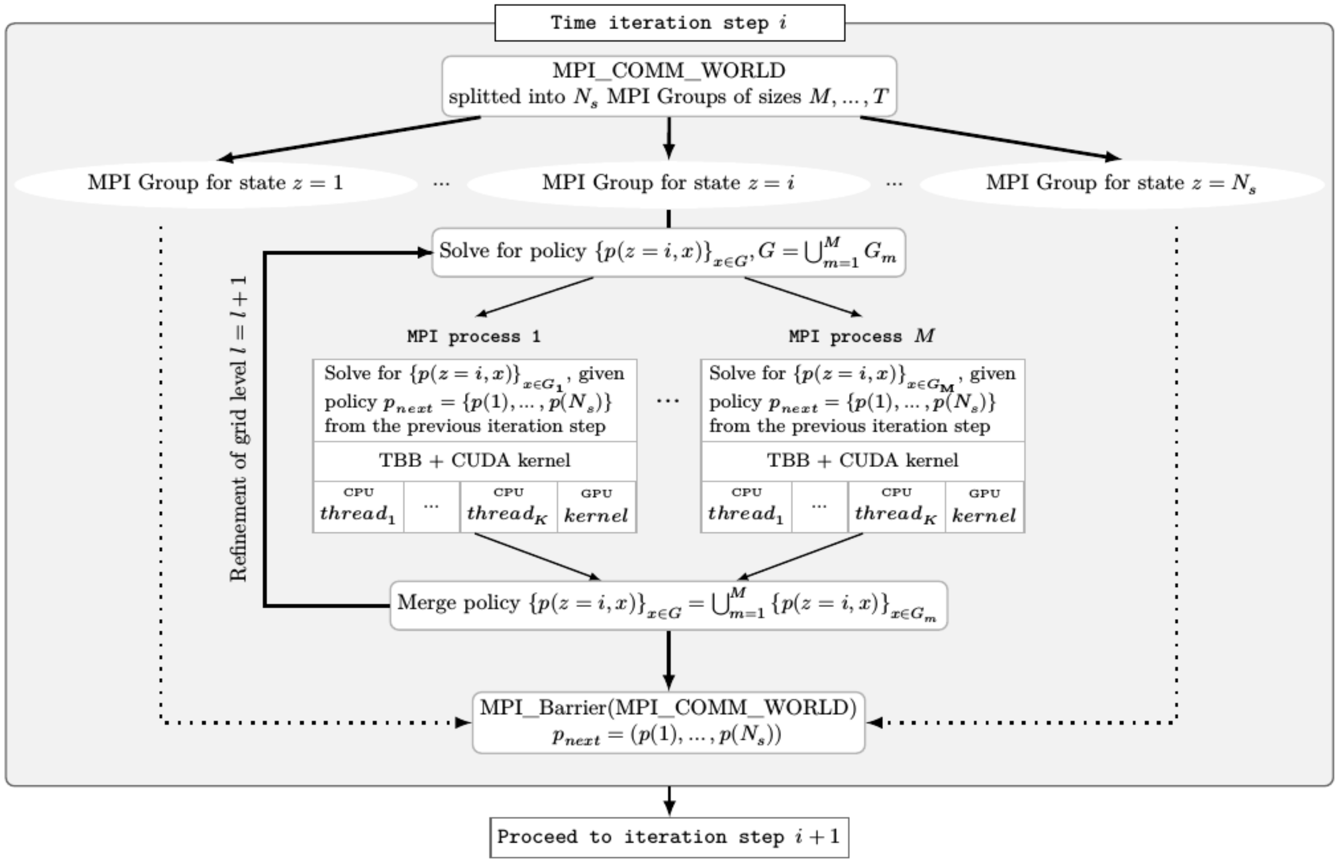
Shared memory,
Multi-threads domain:
OpenMP, pthreads,
C++11...

Accelerator domain:
CUDA, OpenCL,
OpenACC, OpenMP,
OpenGL...

(Slide from C. Gheller)

A generic parallelization scheme for ASGs

Scheidegger et al. (2018)


Time iteration step i
MPI_COMM_WORLD splitted into N_s MPI Groups of sizes $M, \dots, T$
MPI Group for state $z = 1$
MPI Group for state $z = i$
MPI Group for state $z = N_s$
Solve for policy $\{p(z=i,x)\}_{x \in G}, G = \bigcup_{m=1}^{M} G_m$
Refinement of grid level $l = l + 1$
MPI process 1
MPI process M
Solve for $\{p(z=i,x)\}_{x \in G_1}$, given policy $p_{next} = \{p(1), \dots, p(N_s)\}$ from the previous iteration step
Solve for $\{p(z=i,x)\}_{x \in G_M}$, given policy $p_{next} = \{p(1), \dots, p(N_s)\}$ from the previous iteration step
TBB + CUDA kernel
CPU $thread_1$
CPU $thread_K$
GPU $kernel$
Merge policy $\{p(z=i,x)\}_{x \in G} = \bigcup_{m=1}^{M} \{p(z=i,x)\}_{x \in G_m}$
MPI_Barrier(MPI_COMM_WORLD) $p_{next} = (p(1), \dots, p(N_s))$
Proceed to iteration step $i+1$

→ At every grid point in every state, an optimization problem (or a set of nonlinear equations) needs to be solved.

→ Sizes of the individual adaptive sparse grids may be very different.

→ We need to carefully ensure workload balance.

Strong scaling on “Piz Daint” at CSCS

Scheidegger et al. (2018)

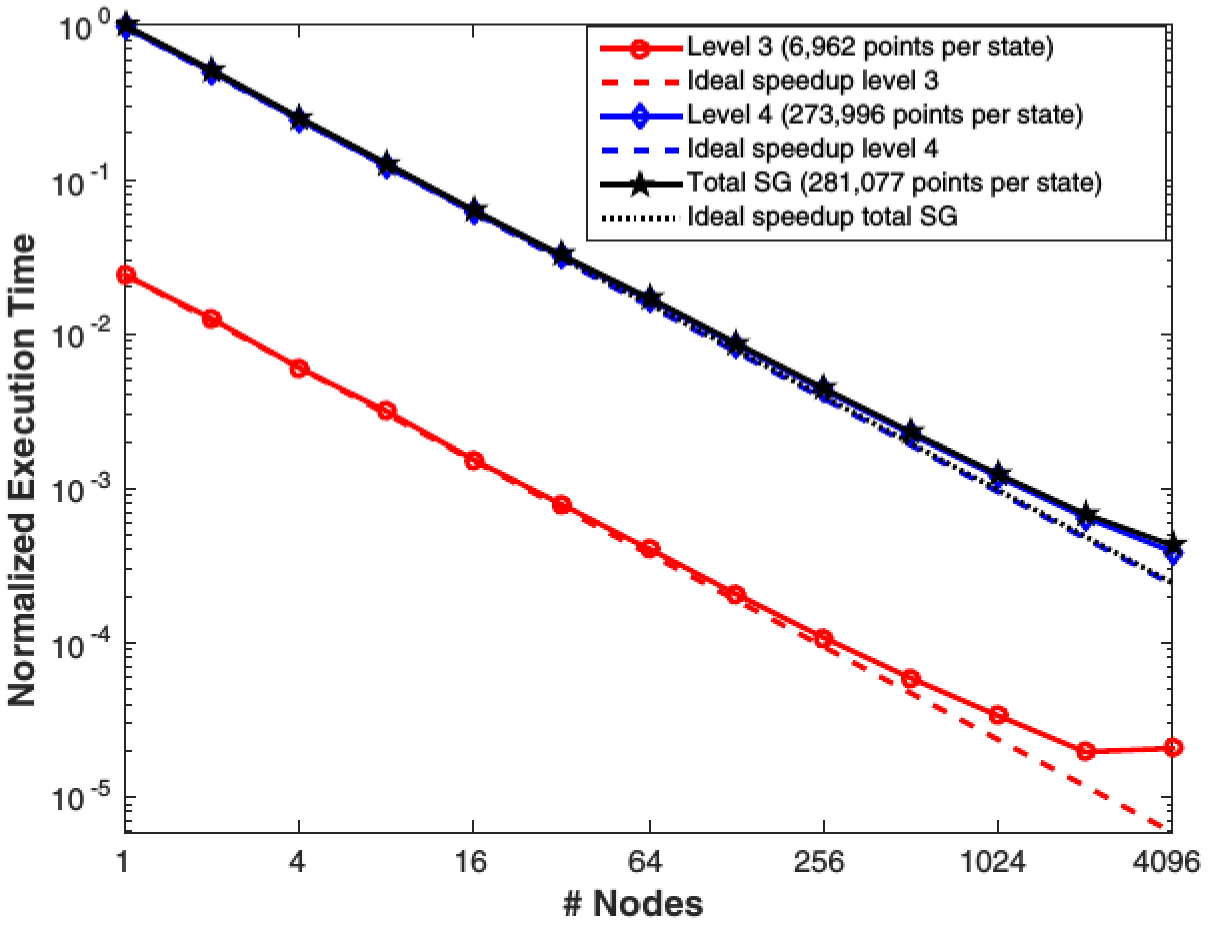
- Test on Cray XC50

- 16 x 281,077 = 4,497,232 points.

- 265,336,688 unknowns.

- 70% efficiency on 4,096 nodes.

- Speed-up limitations: few points in lower grid levels.



We focus today on MPI

- Resources are LOCAL (different from shared memory).

- Each process runs in an “isolated” environment. Interactions requires **Messages** to be exchanged.

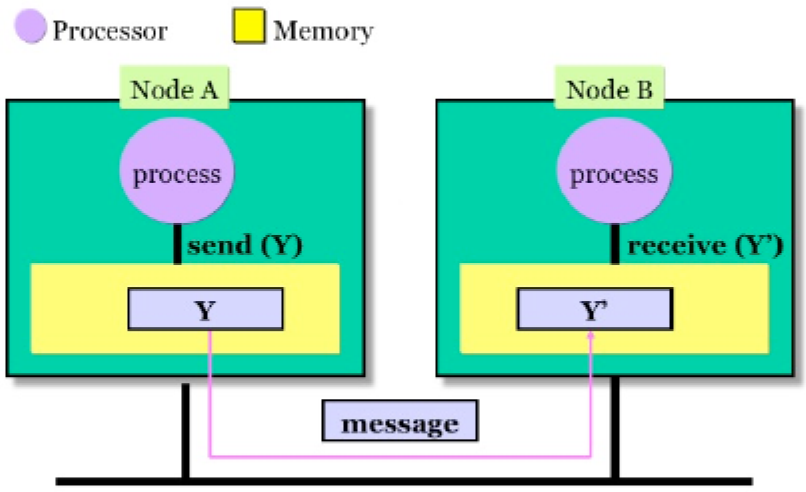
- Messages can be: **instructions, data, synchronization.**

- **MPI works also on Shared Memory systems**.

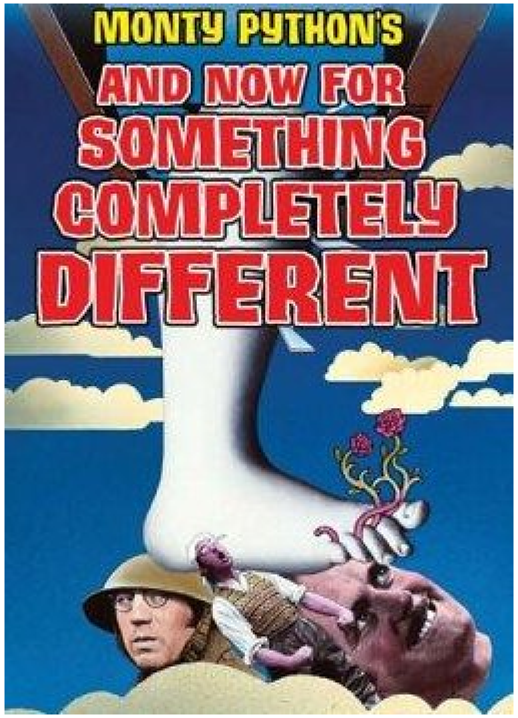
- Time to exchange messages is much larger than accessing local memory.

→ **Massage Passing is a COOPERATIVE Approach, based on 3 operations:**

- **SEND** (a message)
- **RECEIVE** (a message)
- **SYNCHRONIZE**



Detour: MPI and Python



Detour: MPI in Python

Recall: https://github.com/sischei/YaleParallel2018
See **https://mpi4py.scipy.org**

→ **MPI for Python** supports convenient, pickle-based communication of generic Python object as well as fast, near C-speed, direct array data communication of buffer-provider objects (e.g., NumPy arrays).

Communication of generic Python objects:
You have to use **all-lowercase methods** (of the Comm class), like send(), recv(), bcast(). Note that isend() is available, but irecv() is not.

Collective calls like scatter(), gather(), allgather(), alltoall() expect/return a sequence of Comm.size elements at the root or all process. They return a single value, a list of Comm.size elements, or None.

Global reduction operations reduce() and allreduce() are naively implemented, the reduction is actually done at the designated root process or all processes.

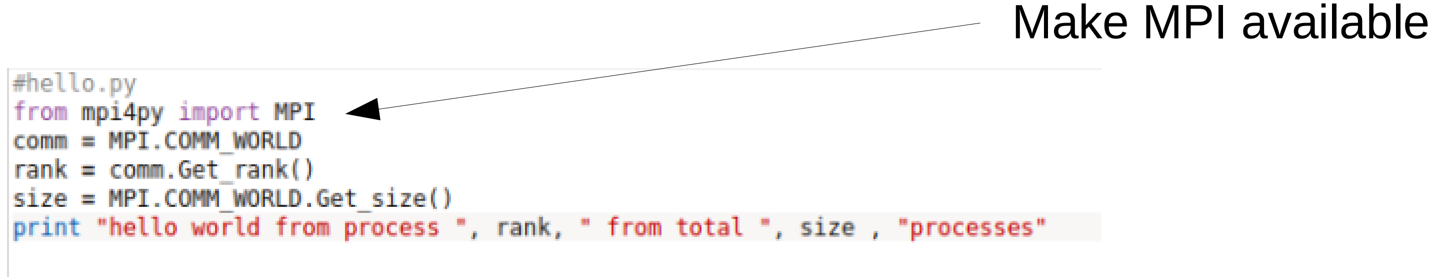
“Hello World” in Python

Recall: https://github.com/sischei/YaleParallel2018

→ **YaleParallel2018/day3/code/MPI4PY**

Run with

> mpirun -np 4 python hello.py

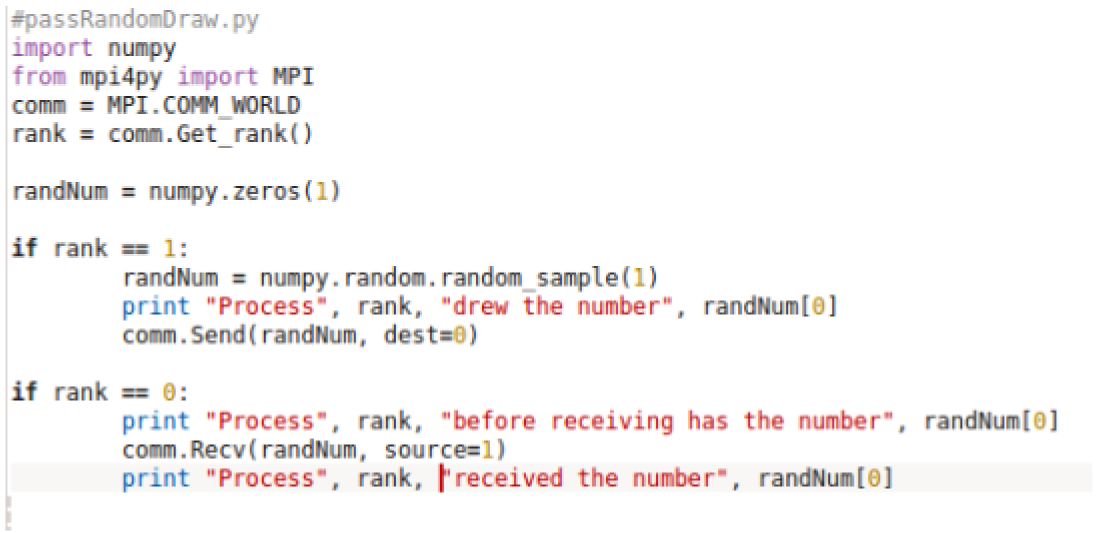


Make MPI available

```
#hello.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = MPI.COMM_WORLD.Get_size()
print "hello world from process ", rank, " from total ", size , "processes"
```

Point-to-Point Communication

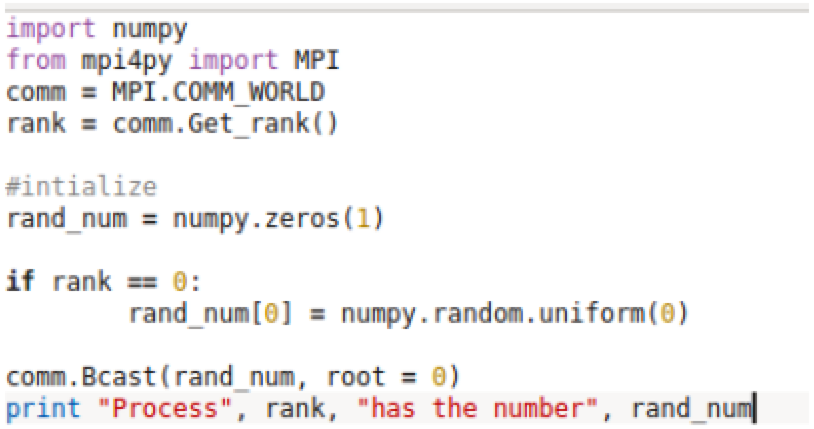
Go to **YaleParallel2018/day3/code/MPI4PY/pointtopoint.py**



```
#passRandomDraw.py
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

randNum = numpy.zeros(1)

if rank == 1:
        randNum = numpy.random.random_sample(1)
        print "Process", rank, "drew the number", randNum[0]
        comm.Send(randNum, dest=0)

if rank == 0:
        print "Process", rank, "before receiving has the number", randNum[0]
        comm.Recv(randNum, source=1)
        print "Process", rank, "received the number", randNum[0]
```

MPI Broadcast in Python

Go to **YaleParallel2018/day3/code/MPI4PY/bcast.py**



```
import numpy
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

#intialize
rand_num = numpy.zeros(1)

if rank == 0:
        rand_num[0] = numpy.random.uniform(0)

comm.Bcast(rand_num, root = 0)
print "Process", rank, "has the number", rand_num
```

Recall VFI – there are many **k's**

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

$$s.t.$$

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

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

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

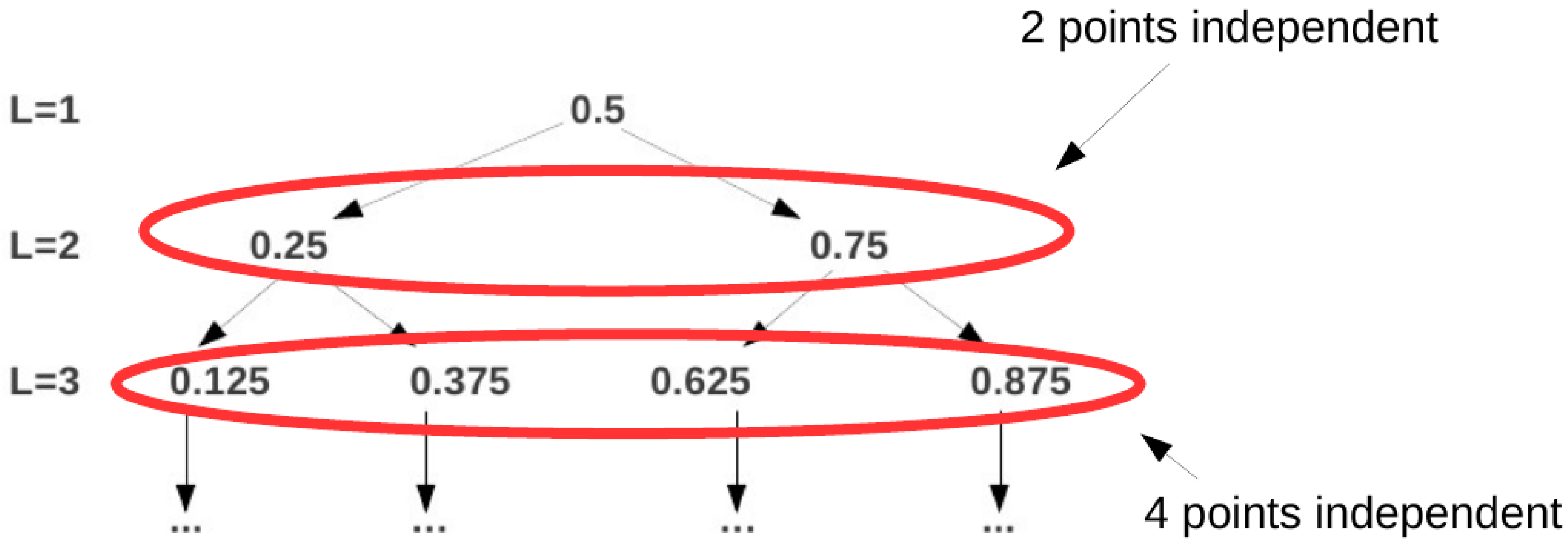
State k: sparse grid coordinates

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

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

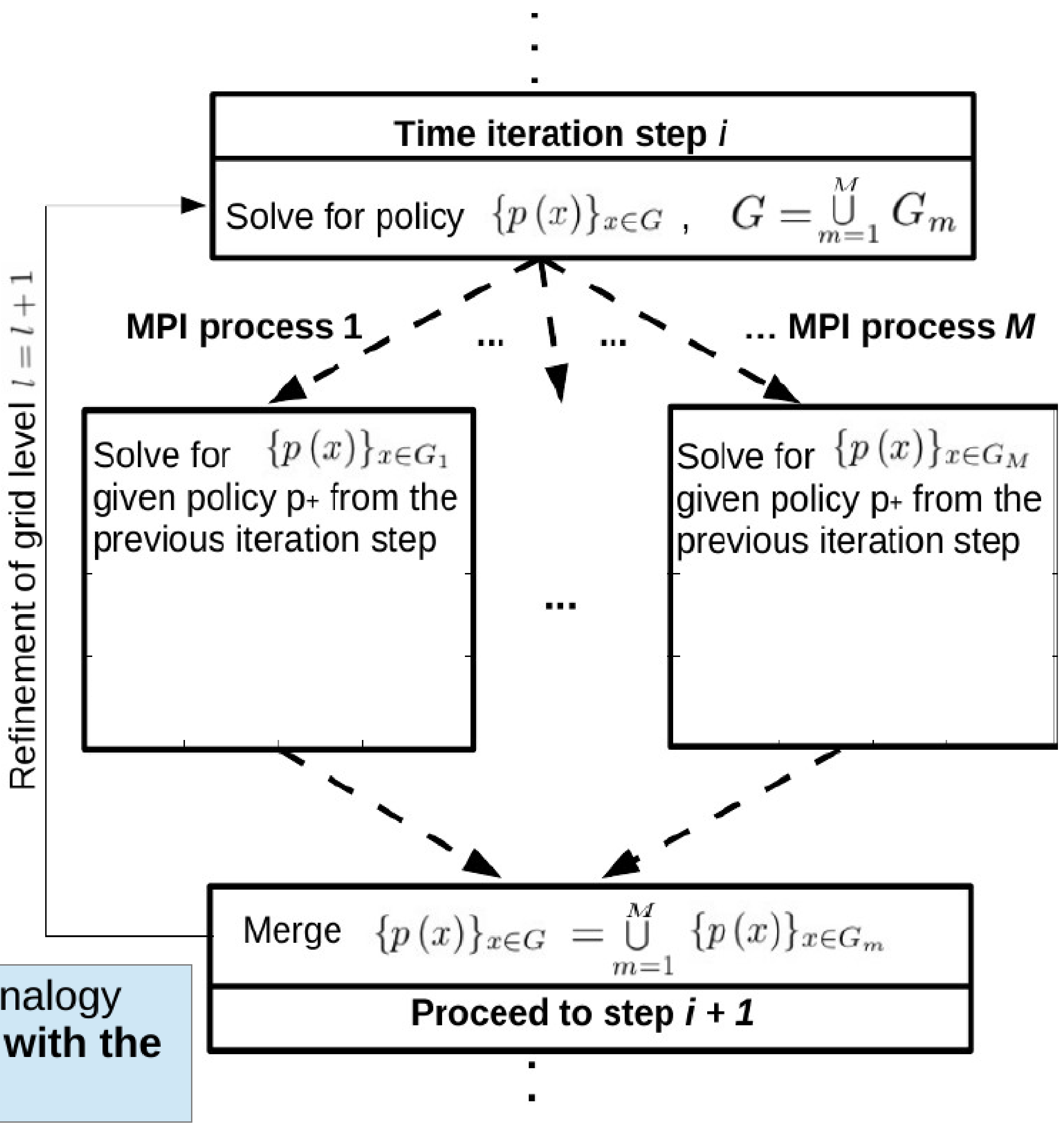
!!! All the individual optimization problems are independent !!!

The parallelization scheme (1d SG)



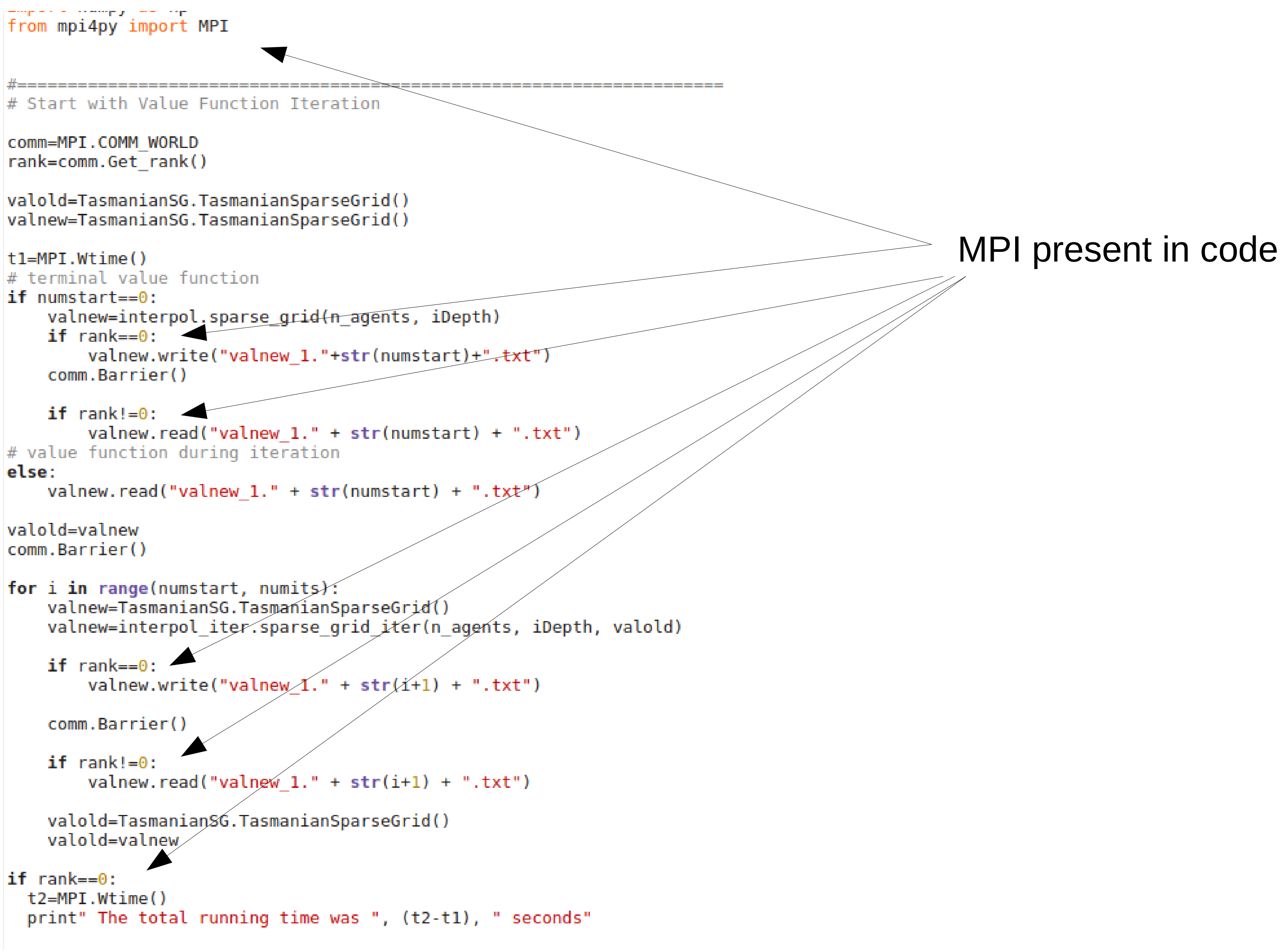
- All newly generated points within a refinement level are independent and have to be distributed equally among different MPI processes.

Parallelization scheme (cont'd)


Time iteration step *i*
Solve for policy $\{p(x)\}_{x \in G}$, $G = \bigcup_{m=1}^{M} G_m$
MPI process 1
...
...
... MPI process *M*
Solve for $\{p(x)\}_{x \in G_1}$ given policy p+ from the previous iteration step
...
Solve for $\{p(x)\}_{x \in G_M}$ given policy p+ from the previous iteration step
Merge $\{p(x)\}_{x \in G} = \bigcup_{m=1}^{M} \{p(x)\}_{x \in G_m}$
Proceed to step *i + 1*
Refinement of grid level $l = l + 1$

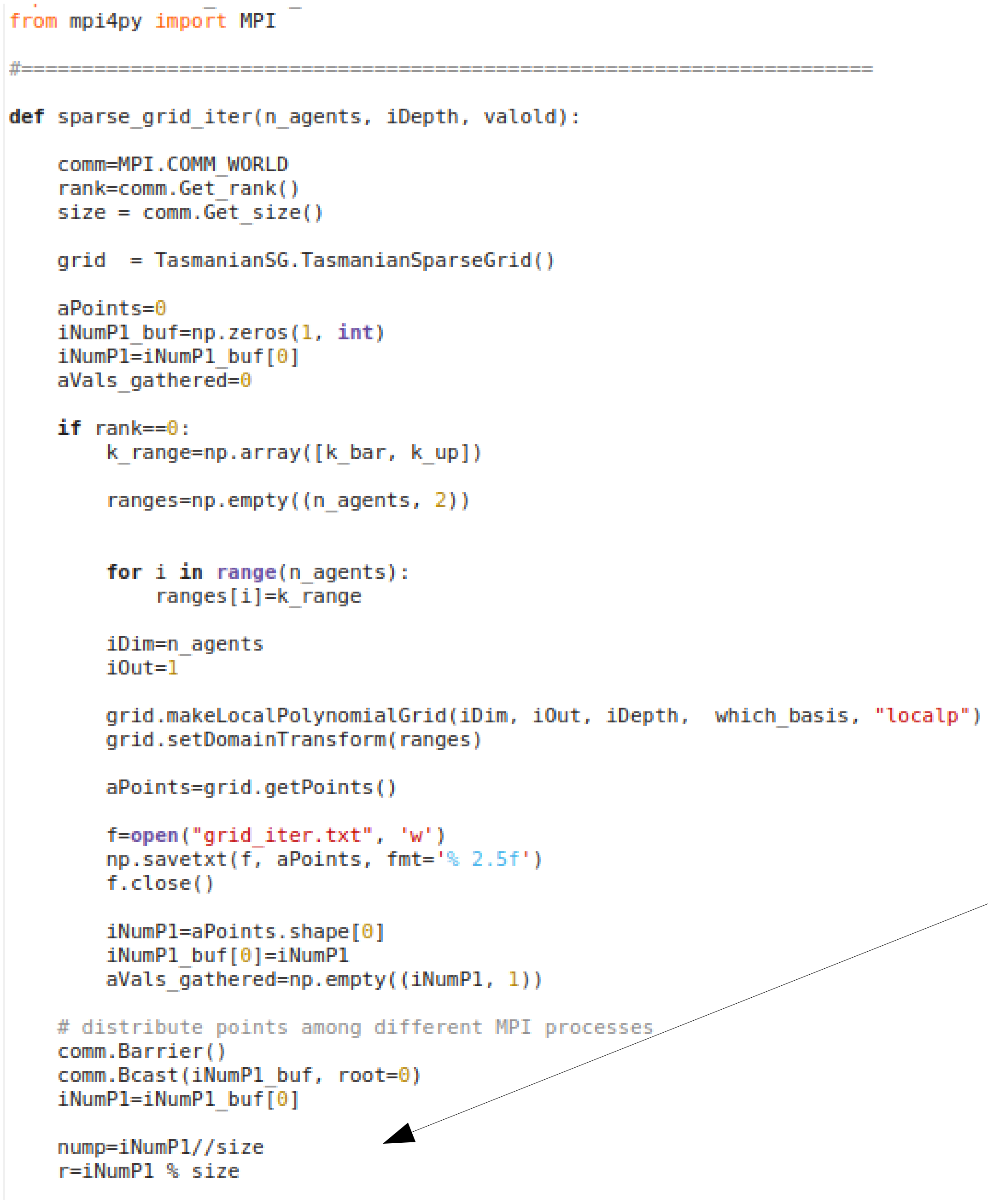
Note: VFI works in analogy
→ **replace policy p with the value function**.

Parallel DP: Code Snippet – main.py



```
from mpi4py import MPI


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

comm=MPI.COMM_WORLD
rank=comm.Get_rank()

valold=TasmanianSG.TasmanianSparseGrid()
valnew=TasmanianSG.TasmanianSparseGrid()

t1=MPI.Wtime()
# terminal value function
if numstart==0:
    valnew=interpol.sparse_grid(n_agents, iDepth)
    if rank==0:
        valnew.write("valnew_1."+str(numstart)+".txt")
    comm.Barrier()

    if rank!=0:
        valnew.read("valnew_1." + str(numstart) + ".txt")
# value function during iteration
else:
    valnew.read("valnew_1." + str(numstart) + ".txt")

valold=valnew
comm.Barrier()

for i in range(numstart, numits):
    valnew=TasmanianSG.TasmanianSparseGrid()
    valnew=interpol_iter.sparse_grid_iter(n_agents, iDepth, valold)

    if rank==0:
        valnew.write("valnew_1." + str(i+1) + ".txt")

    comm.Barrier()

    if rank!=0:
        valnew.read("valnew_1." + str(i+1) + ".txt")

    valold=TasmanianSG.TasmanianSparseGrid()
    valold=valnew

if rank==0:
  t2=MPI.Wtime()
  print" The total running time was ", (t2-t1), " seconds"
```

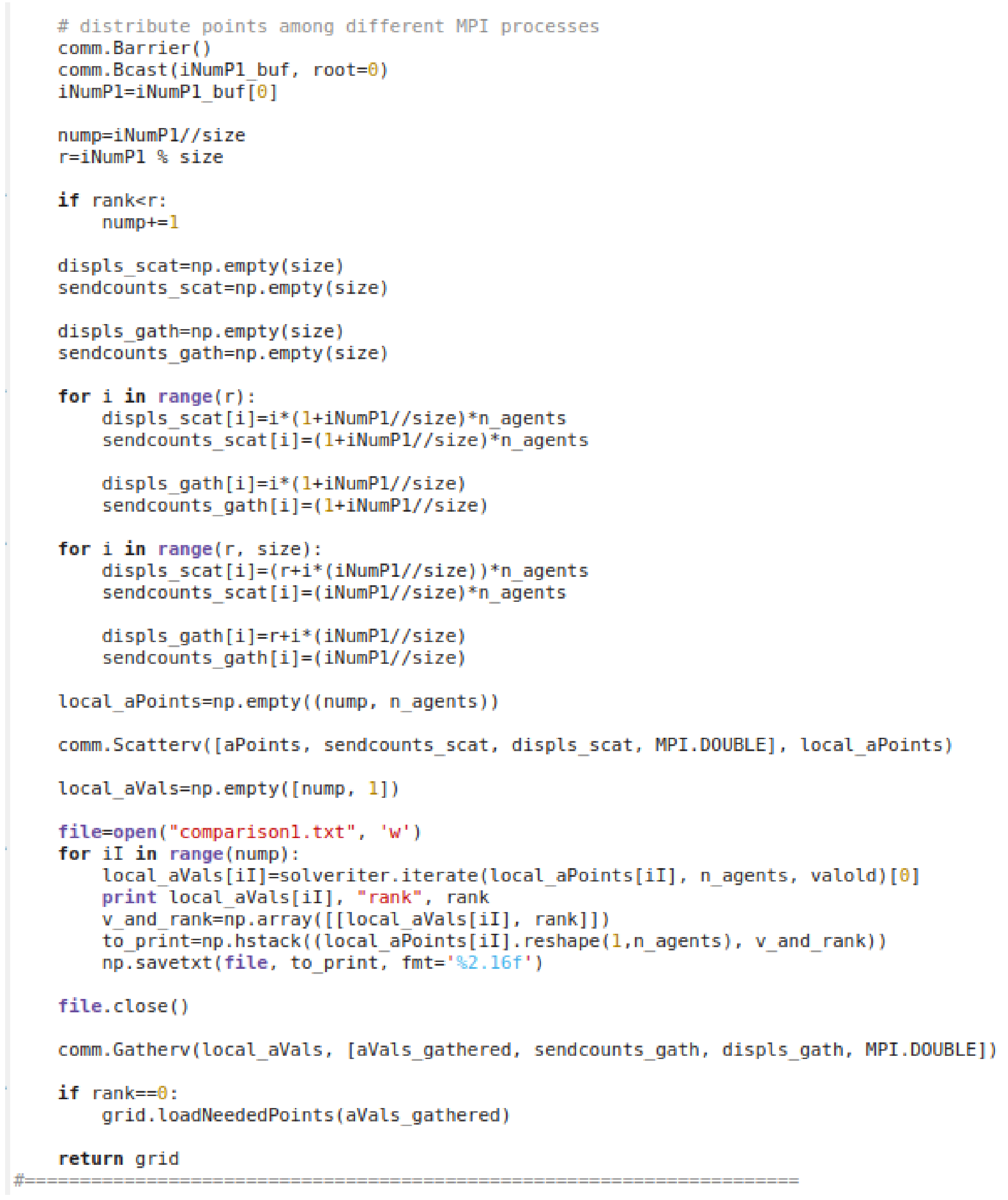
Code Snippet – interpolation_iter.py (1)



```
from mpi4py import MPI

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

def sparse_grid_iter(n_agents, iDepth, valold):

    comm=MPI.COMM_WORLD
    rank=comm.Get_rank()
    size = comm.Get_size()

    grid  = TasmanianSG.TasmanianSparseGrid()

    aPoints=0
    iNumP1_buf=np.zeros(1, int)
    iNumP1=iNumP1_buf[0]
    aVals_gathered=0

    if rank==0:
        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
        iOut=1

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

        aPoints=grid.getPoints()

        f=open("grid_iter.txt", 'w')
        np.savetxt(f, aPoints, fmt='% 2.5f')
        f.close()

        iNumP1=aPoints.shape[0]
        iNumP1_buf[0]=iNumP1
        aVals_gathered=np.empty((iNumP1, 1))

    # distribute points among different MPI processes
    comm.Barrier()
    comm.Bcast(iNumP1_buf, root=0)
    iNumP1=iNumP1_buf[0]

    nump=iNumP1//size
    r=iNumP1 % size
```

Split work among processes

Code Snippet – interpolation_iter.py (2)



```
    # distribute points among different MPI processes
    comm.Barrier()
    comm.Bcast(iNumP1_buf, root=0)
    iNumP1=iNumP1_buf[0]

    nump=iNumP1//size
    r=iNumP1 % size

    if rank<r:
        nump+=1

    displs_scat=np.empty(size)
    sendcounts_scat=np.empty(size)

    displs_gath=np.empty(size)
    sendcounts_gath=np.empty(size)

    for i in range(r):
        displs_scat[i]=i*(1+iNumP1//size)*n_agents
        sendcounts_scat[i]=(1+iNumP1//size)*n_agents

        displs_gath[i]=i*(1+iNumP1//size)
        sendcounts_gath[i]=(1+iNumP1//size)

    for i in range(r, size):
        displs_scat[i]=(r+i*(iNumP1//size))*n_agents
        sendcounts_scat[i]=(iNumP1//size)*n_agents

        displs_gath[i]=r+i*(iNumP1//size)
        sendcounts_gath[i]=(iNumP1//size)

    local_aPoints=np.empty((nump, n_agents))

    comm.Scatterv([aPoints, sendcounts_scat, displs_scat, MPI.DOUBLE], local_aPoints)

    local_aVals=np.empty([nump, 1])

    file=open("comparison1.txt", 'w')
    for iI in range(nump):
        local_aVals[iI]=solveriter.iterate(local_aPoints[iI], n_agents, valold)[0]
        print local_aVals[iI], "rank", rank
        v_and_rank=np.array([[local_aVals[iI], rank]])
        to_print=np.hstack((local_aPoints[iI].reshape(1,n_agents), v_and_rank))
        np.savetxt(file, to_print, fmt='%2.16f')

    file.close()

    comm.Gatherv(local_aVals, [aVals_gathered, sendcounts_gath, displs_gath, MPI.DOUBLE])

    if rank==0:
        grid.loadNeededPoints(aVals_gathered)

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

Perform “local” work

Collect results

Run the Growth model code in parallel

→Model implemented in Python (TASMANIAN)

→ Optimizer used: IPOPT & PYIPOPT (python interface)

→ global_solution_yale19/Lecture_2/SparseGridCode/growth_model/mpi_growth

→ request multiple cores (n a two core interactive job (srun -n 2 --pty bash))

```
$srun -n 2 python main.py
```

IV. Time Iteration and SGs (applied to the Ramsey Model)

We choose the Ramsey model as our first example because:

- it is the most canonical infinite horizon optimization problem.

- it is very simple (in its basic form).

- its simplicity allows us to focus on the solution method.

- however, it can be extended to include many interesting features.

The deterministic Ramsey Model

$$\max U(\{c_t\}_{t=0}^{\infty}) \quad \text{s.t. } c_t + k_{t+1} \leq \underbrace{f(k_t) + (1-\delta)k_t}_{\equiv \bar{f}(k_t)}$$

$$c_t \geq 0,\ k_{t+1} \geq 0\ \forall t \in \mathbb{N}_0,\ k_0 \text{ given}$$

Where:

c_t is consumption at time t

$U(\{c_t\}_{t=0}^{\infty})$ is utility of the consumption stream $\{c_t\}_{t=0}^{\infty}$

k_t is the capital stock at time t, and k_0 the initial capital stock

$f(\cdot)$ is the production function

$\bar{f}(\cdot)$ is production including non-depreciated capital

δ is depreciation

Standard Assumptions

Production

Neoclassical Production:

$f(0) = 0,\ f \in C^2(\mathbb{R}),$
$f'(k) > 0,\ f''(k) < 0,$
$\lim_{k\to 0} f'(k) = \infty,$
$\lim_{k\to \infty} f'(k) = 0$

Special Case:

$$f(k) = k^{\alpha}$$

Cobb-Douglas with capital share α and fixed labor supply (normalized or intensive form)

Preferences

Time-separable utility:

$$U(\{t\}_{t=0}^{\infty}) = \sum_{t=0}^{\infty} \beta^t u(c_t)$$

with discount factor $0 < \beta < 1$ and $\lim_{c\to 0} U'(c) = \infty$.

Special Case:

$$u(c_t) = \begin{cases} \ln(c_t), & \gamma = 1 \\ \frac{c_t^{1-\gamma}}{1-\gamma}, & \gamma \in \mathbb{R}_+ \setminus \{1\} \end{cases}$$

CRRA utility

The Euler Equation

Due to the above assumptions:

$c_t \geq 0, \; k_{t+1} \geq 0$ are never binding

the budget constraint is always binding: $c_t = \bar{f}(k_t) - k_{t+1}$

Therefore, the Lagrangian of the maximization problem simplifies to:

$$\mathcal{L} = \sum_{t=0}^{\infty} \beta^t [u(c_t) + \lambda_t(\bar{f}(k_t) - c_t - k_{t+1})]$$

$$\frac{\partial \mathcal{L}}{\partial c_t} = 0 \Leftrightarrow u'(c_t) = \lambda_t; \; \frac{\partial \mathcal{L}}{\partial k_{t+1}} = 0 \Leftrightarrow \lambda_t = \beta \lambda_{t+1} \bar{f}'(k_{t+1})$$

Combining, we get the Euler equation(s):

$$\boxed{u'(\bar{f}(k_t) - k_{t+1}) = \beta \bar{f}'(k_{t+1}) u'(\bar{f}(k_{t+1}) - k_{t+2}) \quad \forall t \in \mathbb{N}_0}$$

Recursive Equilibrium

Hard to solve for an infinite sequence directly!

⇒ Reduce problem to two periods: ‘**today**’ and ‘**tomorrow**’

⇒ Suppose optimal choice does not depend on ***t*** directly, just on $\boldsymbol{k_t}$

⇒ Look for recursive equilibrium with capital **k** as endogenous state

⇒ A recursive equilibrium policy function p(k) must satisfy:

$$\boxed{u'\left(\bar{f}(k) - p(k)\right) = \beta \cdot \bar{f}'(p(k)) \cdot u'\left(\bar{f}(p(k)) - p(p(k))\right)}$$

Time Iteration (TI): The Idea

(see Judd (1998))

- Start with a guess for the policy function ‘tomorrow’

- Find policy ‘today’ that is optimal given that policy function ‘tomorrow’

- Use this policy as new guess and iterate.

- Hope that this procedure converges, i.e. that the policy does (almost) not change any more.

- The final policy (almost) satisfies the Euler equation when used ‘today’ and ‘tomorrow’.

-Then we have found an (approximate) recursive equilibrium.

Time Iteration Algorithm for the deterministic Ramsey model

1) Initial Step *(Set grid, initial policy, and error tolerance)*

a) Set capital grid $K = [K_1 \ K_2 \ \dots \ K_n] \in \mathbb{R}^n_+, \ K_j < K_{j+1} \ \forall \ j$
b) Set guess for policy function $p : [K_1, K_n] \to [K_1, K_n]$
c) Set error tolerance for time iteration $\bar{\epsilon} > 0$

2) Main Step *(Update policy function)*

a) For all $1 \leq j \leq n$:
Solve Euler equation

$$u'(\bar{f}(k) - k^+) - \beta \cdot \bar{f}'(k^+) \cdot u'(\bar{f}(k^+) - k^{++}) = 0$$

for optimal k^+ given $k = K_j$ and $k^{++} = p(k^+)$. Then, set $K_j^+ = k^+$.

b) **Approximate** new policy $\tilde{p}$ using the data points $\left\{K_j, K_j^+\right\}_{j=1}^n$.

3) Final Step *(Check error criterion)*

a) Calculate error: $\epsilon = \|\tilde{p} - p\|_\infty / \|p\|_\infty$
b) Set $p = \tilde{p}$.
c) If $\epsilon < \bar{\epsilon}$, then stop and report results; otherwise go to step 1.

Measuring Accuracy: Recall Euler Errors I

We want a policy function that satisfies the Euler equation

$$u'(C(k)) = \beta \cdot \bar{f}'(\bar{f}(k) - C(k)) \cdot u'(C(\bar{f}(k) - C(k)))$$

at all $k \in [k_{min}, k_{max}]$, not only at k^*. We proceed as follows:

Create many points $\{\tilde{k}_i\}_{i=1}^{I} : \tilde{k}_i \in [k_{min}, k_{max}]$

Compute consumption implied by approximate policy: $\hat{c}_i = \hat{C}(\tilde{k}_i)$.

Compute consumption implied by Euler equation and approximate policy 'tomorrow': $c_i^* = u_c^{-1}\left[\beta \bar{f}'(\bar{f}(\tilde{k}_i) - \hat{c}_i) \cdot u_c\left(\hat{C}(\bar{f}(\tilde{k}_i) - \hat{c}_i)\right)\right]$

The (relative) error that the agent makes 'today' given his choice 'tomorrow' is the Euler error:

$$E_i = |\frac{\hat{c}_i}{c_i^*} - 1|$$

Measuring Accuracy: Recall Euler Errors II

Choose points $\{\tilde{k}_i\}_{i=1}^{I}$ either

- randomly (uniformly distributed) in $[k_{min}, k_{max}]$, or
- as a very fine (equidistant) grid on $[k_{min}, k_{max}]$

Later we will also look at Euler errors along a simulation path

'Bounded rationality' interpretation: The Euler error

$$E_i = |\frac{\hat{c}_i}{c_i^*} - 1|$$

is the fraction by which the approximate consumption choice today differs from the optimal one (given the approximate consumption choice tomorrow). For instance, $E_i = 0.05$ means that consumption is 5% too high or too low relative to the optimum

The Stochastic Ramsey Model

$$
\begin{aligned}
\max \mathbb{E}_0[U(\{c_t\})] \quad \text{s.t. } & c_t + k_{t+1} \leq \underbrace{a_t f(k_t) + (1-\delta)k_t}_{\equiv \bar{f}(a_t, k_t)}, \\
& c_t \geq 0,\ k_{t+1} \geq 0 \ \forall t \in \mathbb{N}_0 \\
& \ln a_{t+1} = \rho \ln a_t + \epsilon_{t+1}, \epsilon_{t+1} \sim \mathcal{N}(0, \sigma_a) \\
& k_0, a_0 \text{ given}
\end{aligned}
$$

where the expectation is over the sequence of stocks $\{a_t\}_{t=1}^{\infty}$ given a_0.

A recursive equilibrium (capital) policy function $p(k, a)$ must satisfy:

$$
u'\left(\bar{f}(k,a) - p(k,a)\right) = \beta \mathbb{E}[\bar{f}'(p(k,a)) \cdot u'\left(\bar{f}(p(k,a)) - p(p(k,a), a^+)\right)]
$$

Time Iteration for Stochastic Ramsey Model

1) Initial Step *(Set grid, initial policy, and error tolerance)*
   a) Set grid $G = [(K_1, A_1) \ \ldots \ (K_n, A_n)] \in \mathbb{R}_+^{n\times 2}$
   b) Set guess for policy function $p : [K_1, K_n] \times [A_1, A_n] \to [K_1, K_n]$
   c) Set error tolerance for time iteration $\bar{\epsilon} > 0$

2) Main Step *(Update policy function)*
   a) For all $1 \leq j \leq n$:
      Solve Euler equation

$$u'(\bar{f}(k, a) - k^+) - \beta \mathbb{E}[\bar{f}'(k^+, a^+) \cdot u'(\bar{f}(k^+, a^+) - k^{++})] = 0$$

      for optimal k^+ given $k = K_j$ and $k^{++} = p(k^+, a^+)$. Then, set $K_j^+ = k^+$.
   b) **Approximate** new policy $\tilde{p}$ using the data points $\left\{K_j, K_j^+\right\}_{j=1}^n$.

3) Final Step *(Check error criterion)*
   a) Calculate error: $\epsilon = \|\tilde{p} - p\|_\infty / \|p\|_\infty$
   b) Set $p = \tilde{p}$.
   c) If $\epsilon < \bar{\epsilon}$, then stop and report results; otherwise go to step 1.

Evaluating the Expectation: Recall Quadrature

- Each time we solve the first order conditions, we have to evaluate:

$$\mathbb{E}[\bar{f}'(k^+, a^+) \cdot u'(\bar{f}(k^+, a^+) - p(k^+, a^+))]$$

- To transform the expectation into a sum we use a quadrature method.

- We choose Gauss-Hermite quadrature (see Judd 1998, p.262).

→ Note: Integration always becomes an issue with increasing dim.

Choosing Equation Solver and Function Approximation

To implement policy function iteration, we need to choose:

→ A method for solving equations, namely the Euler equation

→ A method for approximating functions, namely the policy function

We choose for equation solving:

→ **fsolve** (from Matlab’s Optimization Toolbox)

→ for interpolation: **sparse grids**, in particular spinterp
(from Klimke’s Sparse Grid Interpolation Toolbox – we want to
avoid to introduce more complex optimizers)

Let’s look at the code and see how that works ...

Comparing Sparse Grids

We solve the stochastic Ramsey model with different sparse grids:

Grid Type (in **spinterp**)	Clenshaw-Curtis	Chebyshev
Basis Function	Piecewise Linear	Global Polynomial
Points / Avg EE / Time (sec.)	$13 \ / \ 7 \cdot 10^{-3} \ / \ 6$	$13 \ / \ 7 \cdot 10^{-4} \ / \ 7$
Points / Avg EE / Time (sec.)	$145 \ / \ 6 \cdot 10^{-4} \ / \ 132$	$29 \ / \ 6 \cdot 10^{-5} \ / \ 17$
Points / Avg EE / Time (sec.)	$321 \ / \ 3 \cdot 10^{-4} \ / \ 389$	$65 \ / \ 7 \cdot 10^{-6} \ / \ 45$

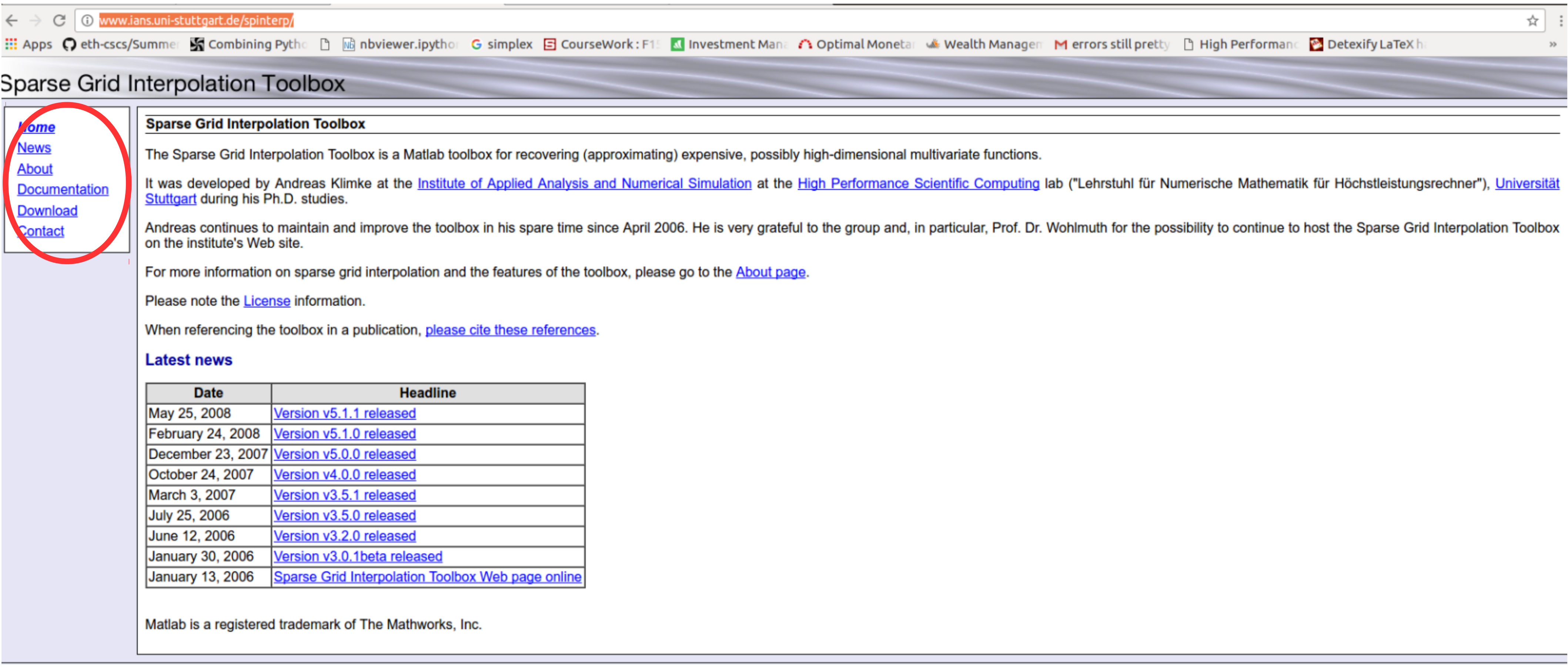
- Increasing the number of grid points substantially reduces the Euler errors for both types of grids.

- The global polynomial approximation performs much better in our (smooth) Application.

- For models with kinks (e.g. from occasionally binding constraints) local basis functions are preferable.

SPINTERP

http://www.ians.uni-stuttgart.de/spinterp/



Sparse Grid Interpolation Toolbox

The Sparse Grid Interpolation Toolbox is a Matlab toolbox for recovering (approximating) expensive, possibly high-dimensional multivariate functions.

It was developed by Andreas Klimke at the Institute of Applied Analysis and Numerical Simulation at the High Performance Scientific Computing lab ("Lehrstuhl für Numerische Mathematik für Höchstleistungsrechner"), Universität Stuttgart during his Ph.D. studies.

Andreas continues to maintain and improve the toolbox in his spare time since April 2006. He is very grateful to the group and, in particular, Prof. Dr. Wohlmuth for the possibility to continue to host the Sparse Grid Interpolation Toolbox on the institute's Web site.

For more information on sparse grid interpolation and the features of the toolbox, please go to the About page.

Please note the License information.

When referencing the toolbox in a publication, please cite these references.

Latest news

Date	Headline
May 25, 2008	Version v5.1.1 released
February 24, 2008	Version v5.1.0 released
December 23, 2007	Version v5.0.0 released
October 24, 2007	Version v4.0.0 released
March 3, 2007	Version v3.5.1 released
July 25, 2006	Version v3.5.0 released
June 12, 2006	Version v3.2.0 released
January 30, 2006	Version v3.0.1beta released
January 13, 2006	Sparse Grid Interpolation Toolbox Web page online

Matlab is a registered trademark of The Mathworks, Inc.

Run Example Code on GRACE

→ **Log on to GRACE**

```
> ssh -X NETID@grace.hpc.yale.edu
```

→ **Load matlab**

```
> module load Apps/Matlab
```

→ **Start MATLAB without graphical interface**

```
> matlab -nojvm
```

→ **Go to example and run it.**

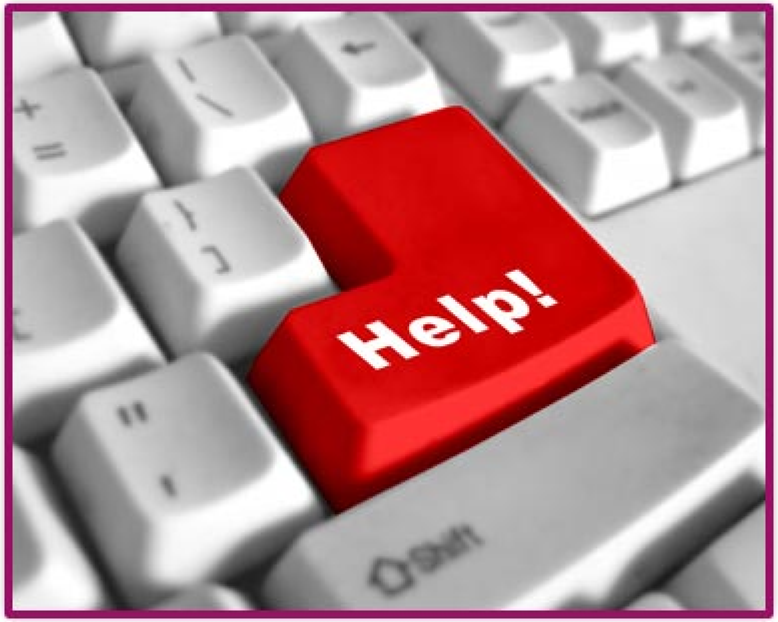
```
> cd global_solution_yale19/Lecture_2/SparseGridCode/Econ_example_ramsey
> TimeIterationWithSparseGrids
```

→ Play with settings (basis functions, accuracy, etc...)

INTERACTIVE JOBS ON GRACE:

```
srun --pty --x11 -p interactive bash
```

See https://research.computing.yale.edu/support/hpc/user-guide/slurm


Help!
Shift