

# **Active Subspaces & solving dynamic models on high-dimensional, irregularly-shaped state spaces**

Simon Scheidegger  
simon.scheidegger@unil.ch  
January 23<sup>th</sup>, 2019

Cowles Foundation – Yale University

# Today's Road-map

- I. Dimension-reduction with the active subspace method.
- II. Solving dynamic models on high-dimensional, (irregularly-shaped) state spaces.
- III. Wrap-up of the lecture-suite.

# Recall: GPR with noisy data

- In empirical setups, **measurement noise may arise** from our **inability to control all the influential factors** or from **irreducible (aleatory) uncertainties**.
- In **computer simulations**, measurement uncertainty may stem from quasi-random stochasticity, or chaotic behavior.
- Now let us consider the case where what we observe is a noisy version of the underlying function

$$y = f(x) + \epsilon, \text{ where } \epsilon \sim \mathcal{N}(0, \sigma_y^2)$$

# Recall: GPR with noisy data (II)

- In this case (presence of noise), the model is not required to interpolate the data, **but it must come “close”** to the observed data.
- The covariance of the observed noisy responses is

$$\text{cov}[y_p, y_q] = \kappa(\mathbf{x}_p, \mathbf{x}_q) + \sigma_y^2 \delta_{pq}$$

where  $\delta_{pq} = \mathbb{I}(p = q)$

- The second matrix is **diagonal** because we assumed the **noise terms were independently added to each observation**.

# Recall: The GPR with noisy data (III)

- The joint density of the observed data and the latent, noise-free function on the test points is given by

Latent function.  $\rightarrow$

$$\begin{pmatrix} \mathbf{y} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N} \left( \mathbf{0}, \begin{pmatrix} \mathbf{K}_y & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{pmatrix} \right)$$

Noise in the diagonal, rest is as before.  $\rightarrow$

- where we are assuming the mean is zero, for notational simplicity.
- Hence the posterior predictive density is

$$\begin{aligned} p(\mathbf{f}_* | \mathbf{X}_*, \mathbf{X}, \mathbf{y}) &= \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) \\ \boldsymbol{\mu}_* &= \mathbf{K}_*^T \mathbf{K}_y^{-1} \mathbf{y} \\ \boldsymbol{\Sigma}_* &= \mathbf{K}_{**} - \mathbf{K}_*^T \mathbf{K}_y^{-1} \mathbf{K}_* \end{aligned}$$

# Recall: Learning the kernel parameters

Marginal likelihood  $p(\mathbf{y}|\mathbf{X}) = \int p(\mathbf{y}|\mathbf{f}, \mathbf{X})p(\mathbf{f}|\mathbf{X})d\mathbf{f}$

Since  $p(\mathbf{f}|\mathbf{X}) = \mathcal{N}(\mathbf{f}|\mathbf{0}, \mathbf{K})$

and  $p(\mathbf{y}|\mathbf{f}) = \prod_i \mathcal{N}(y_i|f_i, \sigma_y^2)$

the (log-) marginal likelihood is given by

$$\log p(\mathbf{y}|\mathbf{X}) = \log \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}_y) = -\frac{1}{2}\mathbf{y}\mathbf{K}_y^{-1}\mathbf{y} - \frac{1}{2}\log |\mathbf{K}_y| - \frac{N}{2}\log(2\pi)$$

1<sup>st</sup> term: data fit term

2<sup>nd</sup> term: a model complexity term

3<sup>rd</sup> term: a constant.

# Recall: Maximizing the the marginal likelihood

- Let the kernel parameters (also called **hyper-parameters**) be denoted by  **$\theta$**
- One can show that the following holds.

$$\begin{aligned}\frac{\partial}{\partial \theta_j} \log p(\mathbf{y}|\mathbf{X}) &= \frac{1}{2} \mathbf{y}^T \mathbf{K}_y^{-1} \frac{\partial \mathbf{K}_y}{\partial \theta_j} \mathbf{K}_y^{-1} \mathbf{y} - \frac{1}{2} \text{tr}(\mathbf{K}_y^{-1} \frac{\partial \mathbf{K}_y}{\partial \theta_j}) \\ &= \frac{1}{2} \text{tr} \left( (\boldsymbol{\alpha} \boldsymbol{\alpha}^T - \mathbf{K}_y^{-1}) \frac{\partial \mathbf{K}_y}{\partial \theta_j} \right)\end{aligned}$$

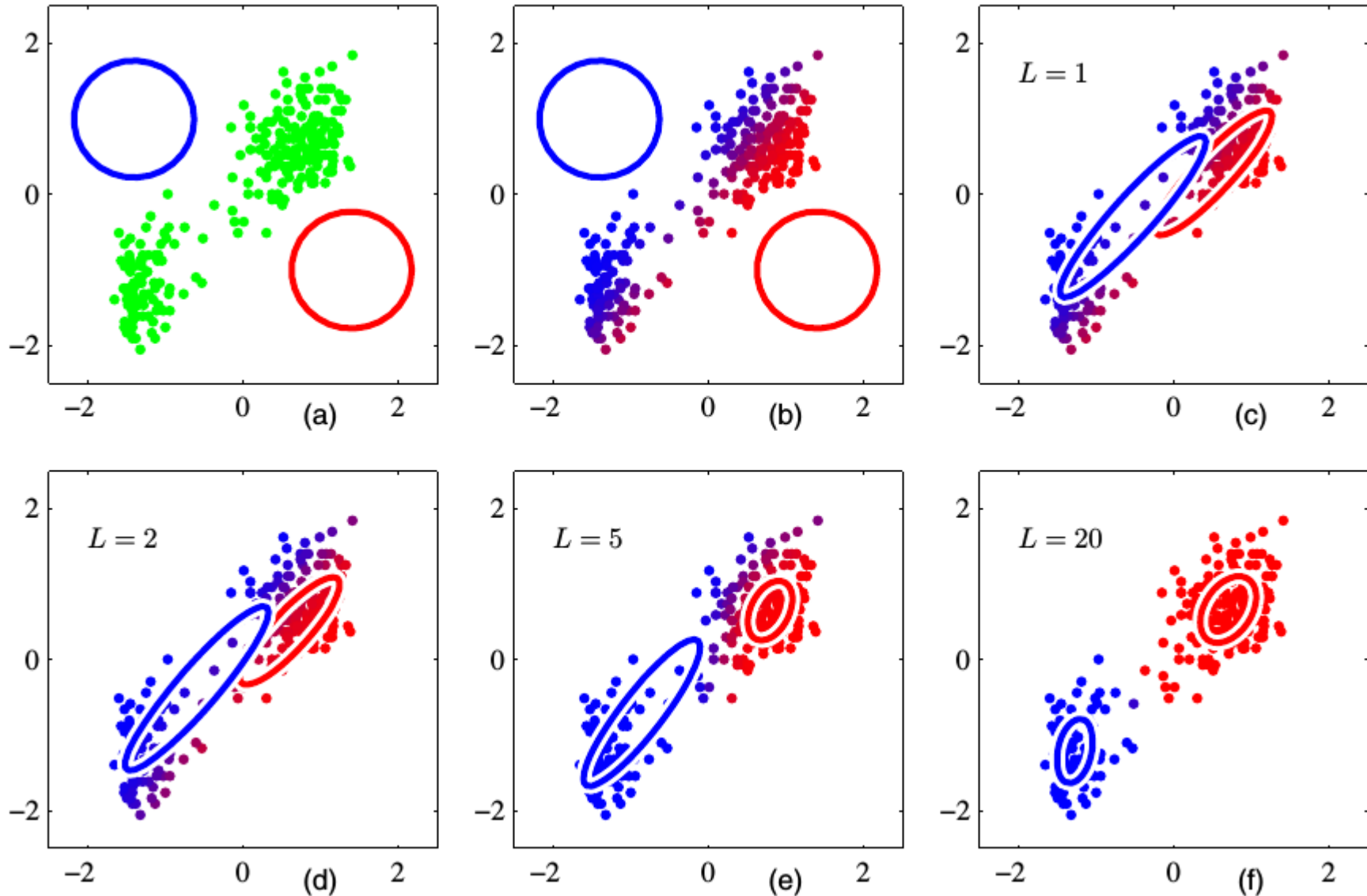
where  $\boldsymbol{\alpha} = \mathbf{K}_y^{-1} \mathbf{y}$

## **Computational complexity:**

- It takes  $O(N^3)$  time to compute  $\mathbf{K}_y^{-1}$
- $O(N^2)$  time per hyper-parameter to compute the gradient.

# Recall: Gaussian mixture models

See Bishop (2006) for details





# Recall: Superposition of Gaussians

- Formally, a GMM is:  $p(\mathbf{x}) = \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$  **(A)**
- Each Gaussian density  $\mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$  is called **a component of the mixture**.
- It has its own **mean  $\boldsymbol{\mu}_k$**  and **covariance  $\boldsymbol{\Sigma}_k$** .
- $\pi_k$  : mixing coefficients.**

→ If we integrate both sides of **(A)** with respect to  $\mathbf{x}$ , and note that both  $p(\mathbf{x})$  and the individual Gaussian components are normalized, one obtains:

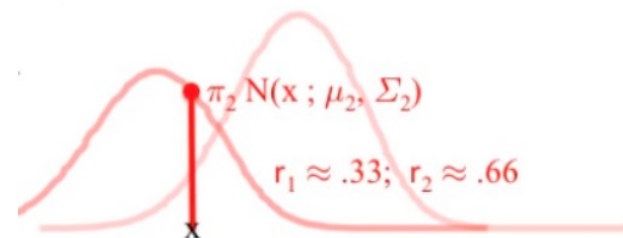
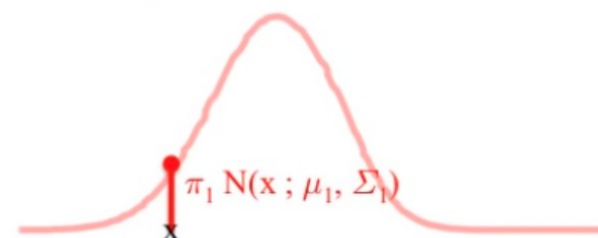
$$\sum_{k=1}^K \pi_k = 1. \quad 0 \leq \pi_k \leq 1.$$

$$\left[ p(\mathbf{x}) \geq 0 \quad \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \geq 0 \right]$$

# EM in the multidimensional case

- Start with parameters describing each cluster
- Mean  $\mu_c$ , Covariance  $\Sigma_c$ , “size”  $\pi_c$
- **E-step (“Expectation”):**
  - For **each observation/point**  $x_i$
  - Compute “ $r_{ic}$ ”, the probability that it belongs to cluster  $c$ .
    - Compute its probability under model  $c$ .
    - Normalize to sum to one (over clusters  $c$ ).

$$r_{ic} = \frac{\pi_c \mathcal{N}(x_i ; \mu_c, \Sigma_c)}{\sum_{c'} \pi_{c'} \mathcal{N}(x_i ; \mu_{c'}, \Sigma_{c'})}$$



- If  $x_i$  is very likely under the  $c$ -th Gaussian, it gets high weight.
- Denominator just makes  $r$ 's sum to one.

# Recall: Expectation Maximization

- **M-step (“Maximization step”):**

- For each cluster (Gaussian)  $z=c$
- Update its parameters using the (weighted) data points

$$N_c = \sum_i r_{ic}$$

Total responsibility allocated to cluster  $c$

$$\pi_c = \frac{N_c}{N}$$

Fraction of total assigned to cluster  $c$

$$\mu_c = \frac{1}{N_c} \sum_i r_{ic} x_i$$

Weighted mean of assigned data

$$\Sigma_c = \frac{1}{N_c} \sum_i r_{ic} (x_i - \mu_c)^T (x_i - \mu_c)$$

Weighted covariance of assigned data  
(use new weighted means here)

# Recall: Expectation-Maximization

- Likelihood of the data

$$P(x_1, \dots, x_N) = \prod_{i=1}^N \sum_{k=1}^K P(x_i|k)P(k)$$

- Each step increases the log-likelihood of our model

$$\ln p(\mathbf{X}|\boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^N \ln \left\{ \sum_{k=1}^K \pi_k \mathcal{N}(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \right\}$$

- Iterate until convergence
  - Convergence guaranteed – another ascent method.
- Cannot discover  $k$ .

# Value function iteration (on irregularly-shaped domains)

e.g. Bellman (1961), Stokey, Lucas & Prescott (1989), Judd (1998), ...

The solution is approached in the limit as  $j \rightarrow \infty$  by iterations on:

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

s.t.

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

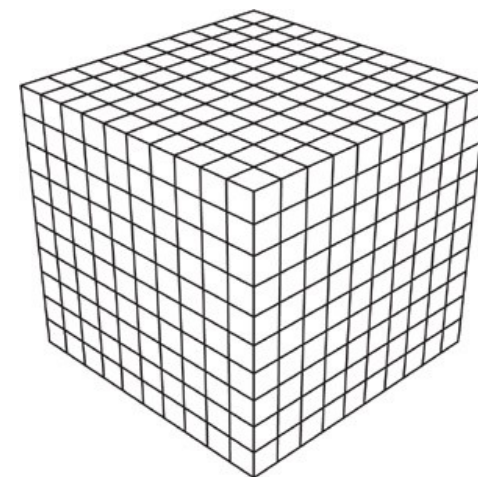
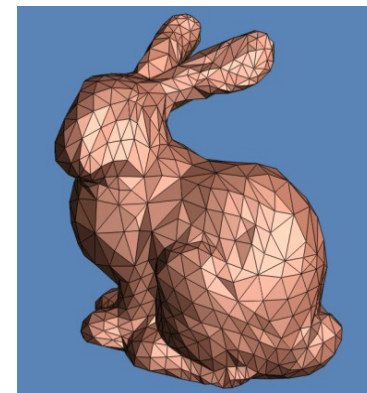
**x**: point in state space; describes your system.  
State-space potentially **irregularly-shaped**  
and **high-dimensional**.

`old solution':  
high-dimensional function on which **we interpolate**.

→ **N<sup>d</sup>** points in ordinary discretization schemes.

→ **“Curse of dimensionality”**

→ **Use-case for Gaussian process regression & active subspaces.**



# How many is dimensions is high dimensions?

Number of parameters (the dimension)	Number of model runs (at 10 points per dimension)	Time for parameter study (at 1 second per run)
1	10	10 sec
2	100	~ 1.6 min
3	1,000	~ 16 min
4	10,000	~ 2.7 hours
5	100,000	~ 1.1 days
6	1,000,000	~ 1.6 weeks
...	...	...
20	1e20	3 trillion years (240x age of the universe)

## Dimension reduction

*Exploit symmetries, e.g., via the active subspace method*

## Deal with #Points

*Adaptive Sparse Grids*

## High-performance computing

*Reduces time to solution, but not the problem size*

**Today's focus**

# The Curse of Dimensionality

- Why is it hard (impossible) to learn functions in high dimensions?
- Methods relying on **local similarity measures** (e.g., SVM, GP's) have severe problems.
- “The curse of highly variable functions for local kernel machines”, Bengio et al. (2006)).
  - The reason is that the **Euclidean distance** is **not a good “closeness” measure in high-dimensions**.

*Most of the volume of a high-dimensional orange is in the skin, not in the pulp. If a constant number of examples is distributed uniformly in a high-dimensional hypercube, beyond some dimensionality most examples are closer to a face of the hypercube than to their nearest neighbors\*.*

\* “A few useful things to know about machine learning”, Pedro Domingos, (2012)

# Volumes in high dimensions

- A lot of probability theory uses the idea of volumes.
- For example, the probability and expectation are usually defined as:

$$\mathbb{P}_\mu(\mathcal{A}) = \int_{\mathcal{A}} d\mu \quad ; \quad \mathbb{E}_\mu(f(X)) = \int_{\mathcal{X}} f(x) d\mu(x)$$

- where the differential element is usually a volume,  $d_\mu(x) = p(x) dx_1 \dots dx_d$ , and  $p(x)$  is a density, but volumes are really weird in higher dimensions.
- The main idea is that our ideas of distance no longer make sense.

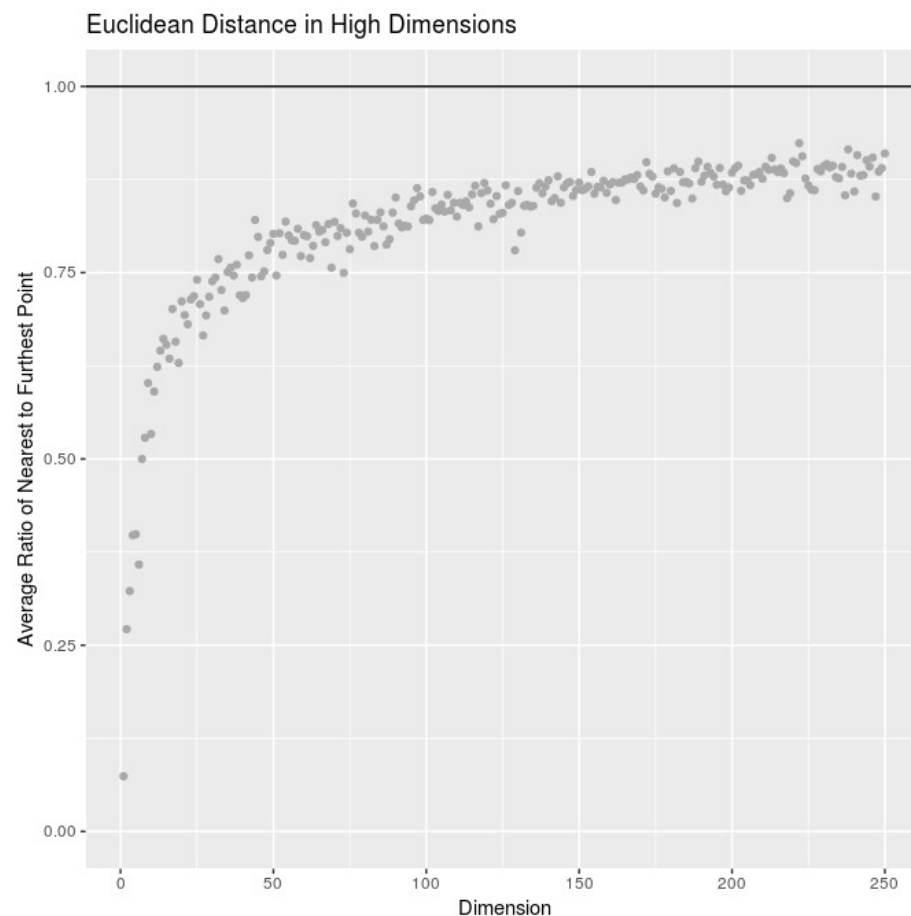


# Volumes in high dimensions (II)

- Consider the following:  
Let's simulate a bunch of random normal points in many dimensions and plot the average ratio of the distance to nearest point to the distance to farthest point

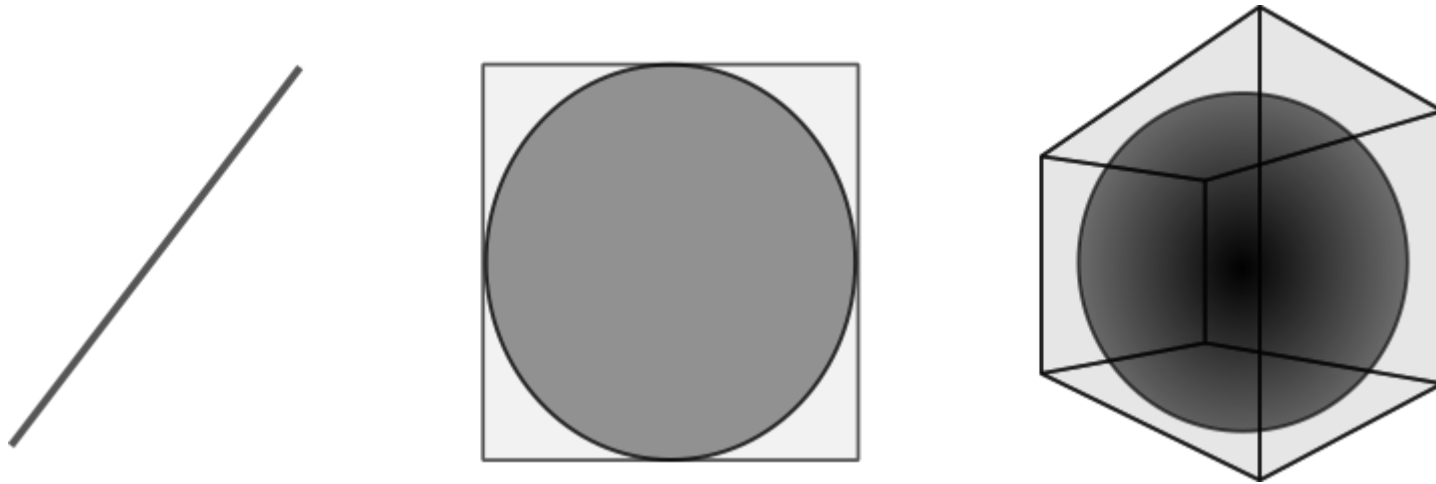
$$D = \sqrt{\sum_{i=1}^d (x_i)^2}$$

- This ratio tends to one, all points “look alike”.
  - No good notion of “locality”.
  - Problems, e.g., for GPR.

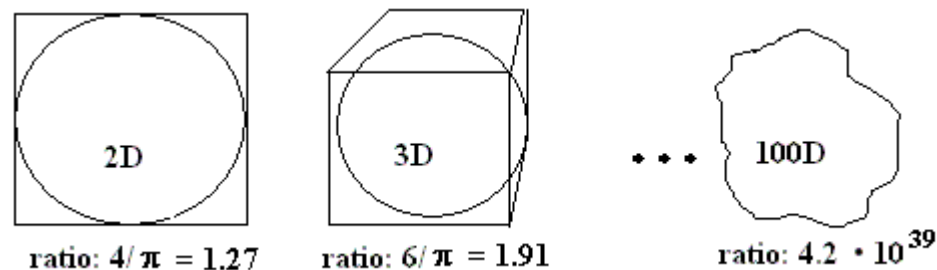


# Volumes in high dimensions (III)

- On the same subject, consider a cube of unit lengths containing a sphere of unit radius in higher dimensions:



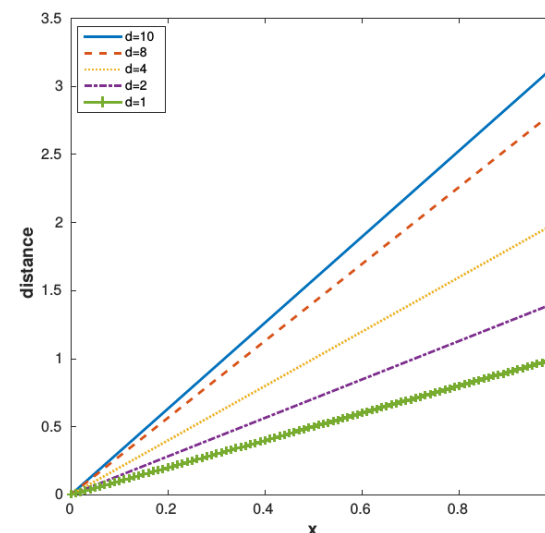
- The edges of the cube keep more and more volume away from the sphere (**cf. Sparse Grids!!!**).



# Uninformative Euclidian Distance

- If a constant number of examples is distributed uniformly in a high-dimensional hypercube, beyond **some dimensionality most examples are closer to a face of the hypercube than to their nearest neighbor.**

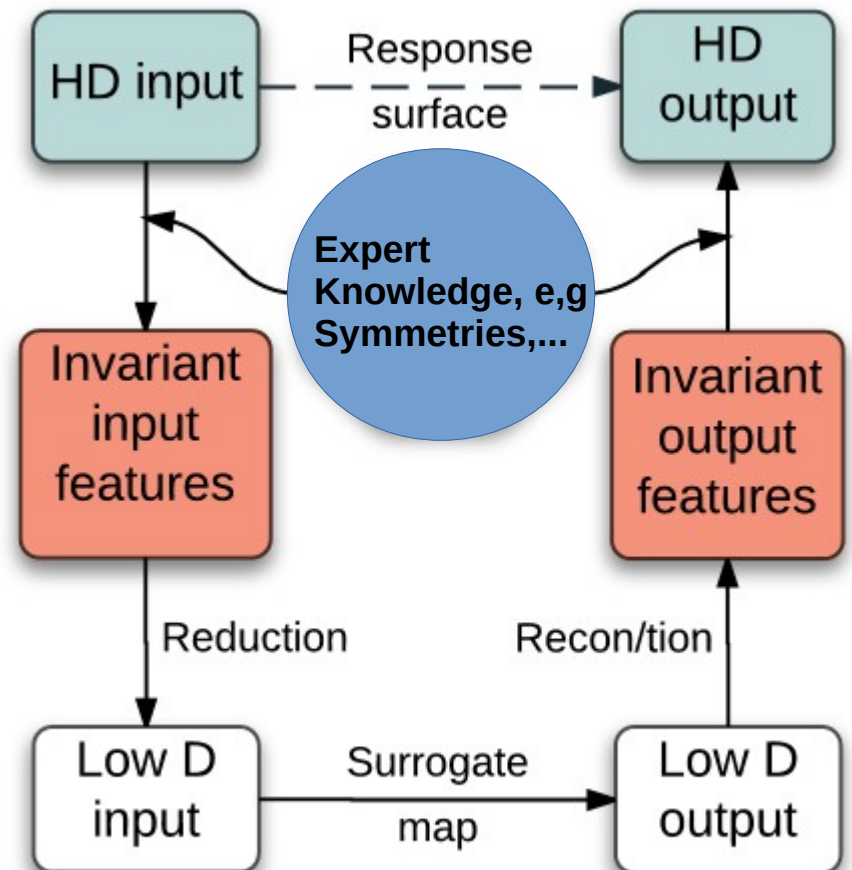
$$D = \sqrt{\sum_{i=1}^d (x_i)^2}$$



- Furthermore, the distance between the center and the corners is  $r\sqrt{d}$  which increases without bound for fixed  $r$ .
- In this sense, **nearly all of the high-dimensional space is "far away" from the center.**
- To put it another way, the high-dimensional unit hypercube can be said to consist almost **entirely of the "corners" of the hypercube, with almost no "middle".**

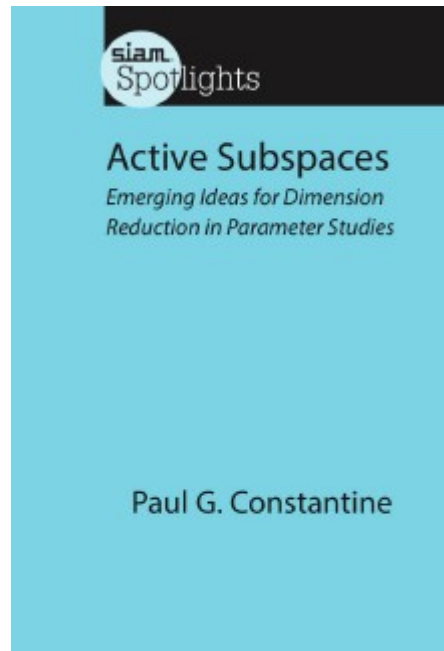
# Dealing with high dimensions

- Generic term approach:
- **Exploit invariances and symmetries**  
Induced by knowledge about the economics problem.
- Discover and exploit (non)-linear manifolds over which the **response exhibits maximal variability**.
- Here, we focus on:
  - HD input (already satisfying symmetries).
  - Discover and exploit linear
  - manifold of maximal variability.



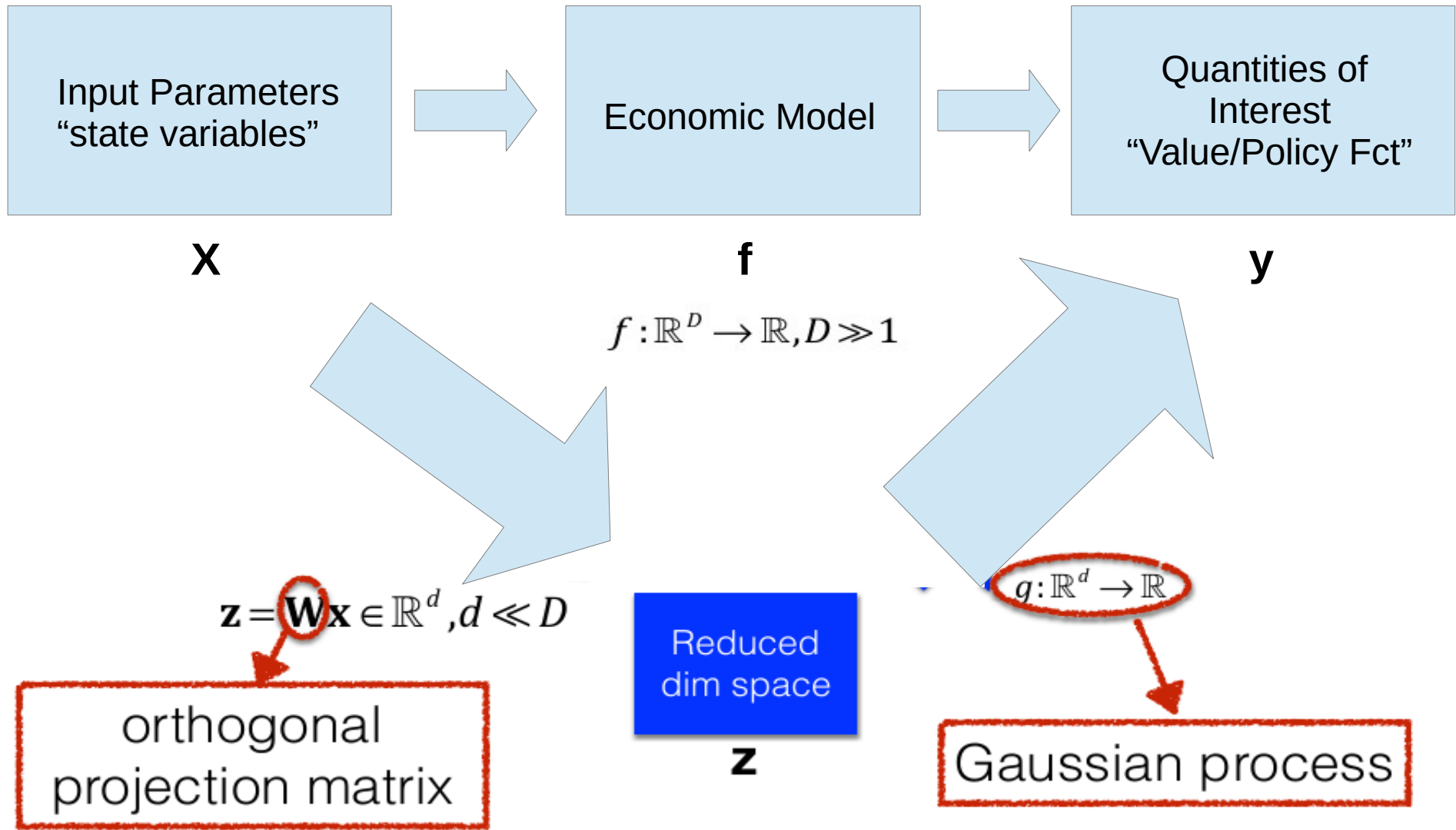
# Active Subspaces

<http://bookstore.siam.org/sl02/>

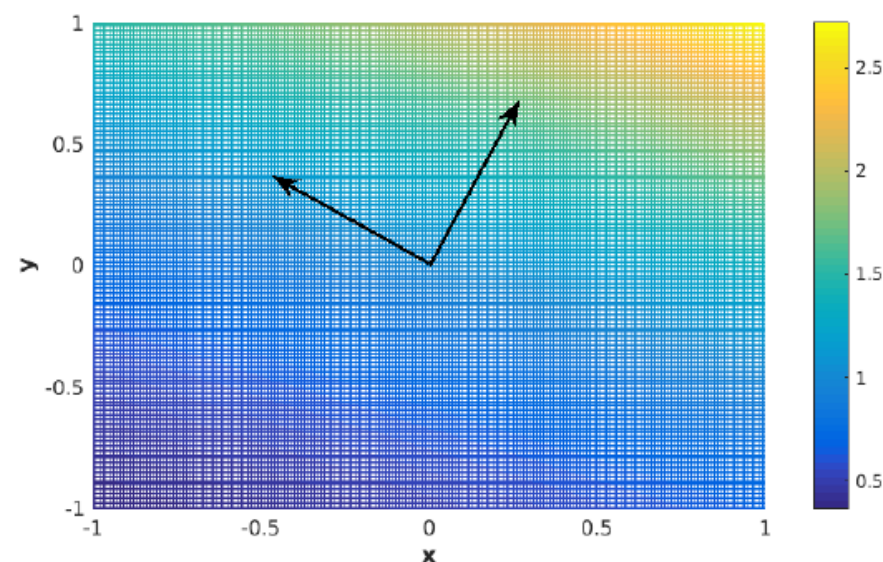
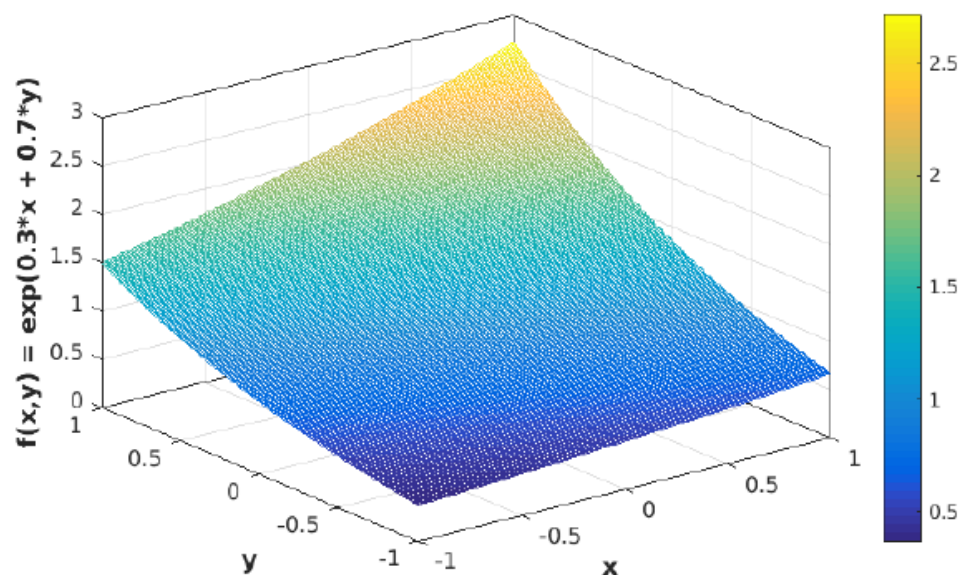


# High Dimensions → Active Subspaces

(Constantine et. al (2013); Constantine (2015); Bilonis et al. (2016), with references therein)



# Active Subspaces – intuitive example



Function varies most along  $[0.3, 0.7]$ , and is constant in the orthogonal direction.

# Discover active subspaces

(see, e.g., Constantine (2015), with references therein)

Step 1: Find  $\mathbf{W}$

→  $\mathbf{C} = \mathbb{E}[\nabla_{\mathbf{x}} f(\mathbf{x}) \nabla_{\mathbf{x}} f(\mathbf{x})^T] \approx \frac{1}{N} \sum_{i=1}^N \nabla_{\mathbf{x}} f(\mathbf{x}^{(i)}) \nabla_{\mathbf{x}} f(\mathbf{x}^{(i)})^T$

“Mean-square directional derivative”

Note: there are also derivative-free ways of constructing  $\mathbf{W}$ .

$$\mathbf{C} = \mathbf{W} \mathbf{\Lambda} \mathbf{W}^T$$

Gradient info

Partition the eigendecomposition  $\left\{ \begin{array}{l} \rightarrow \text{Compute Eigenvalues, order them.} \\ \rightarrow \text{Look for “gaps.”} \end{array} \right.$

$$\mathbf{\Lambda} = \begin{bmatrix} \mathbf{\Lambda}_1 & \\ & \mathbf{\Lambda}_2 \end{bmatrix}, \quad \mathbf{W} = [\mathbf{W}_1 \quad \mathbf{W}_2], \quad \mathbf{W}_1 \in \mathbb{R}^{m \times n}$$

Create a rotated coordinate system

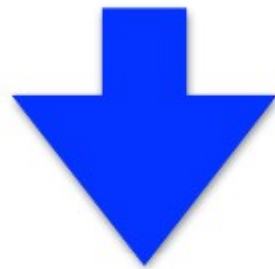
$$\mathbf{x} = \mathbf{W} \mathbf{W}^T \mathbf{x} = \mathbf{W}_1 \mathbf{W}_1^T \mathbf{x} + \mathbf{W}_2 \mathbf{W}_2^T \mathbf{x} = \mathbf{W}_1 \mathbf{q} + \cancel{\mathbf{W}_2 \mathbf{y}}$$



# GPs in active subspace

Step 2: Regression  $\longrightarrow f(\mathbf{x}) \approx g(\mathbf{W}_1^T \mathbf{x}).$

$$\mathcal{D}_{\text{projected}} = \left\{ \left( \mathbf{z}^{(i)} = \mathbf{W} \mathbf{x}^{(i)}, y^{(i)} = f(\mathbf{x}^{(i)}) \right) \right\}_{i=1}^N$$

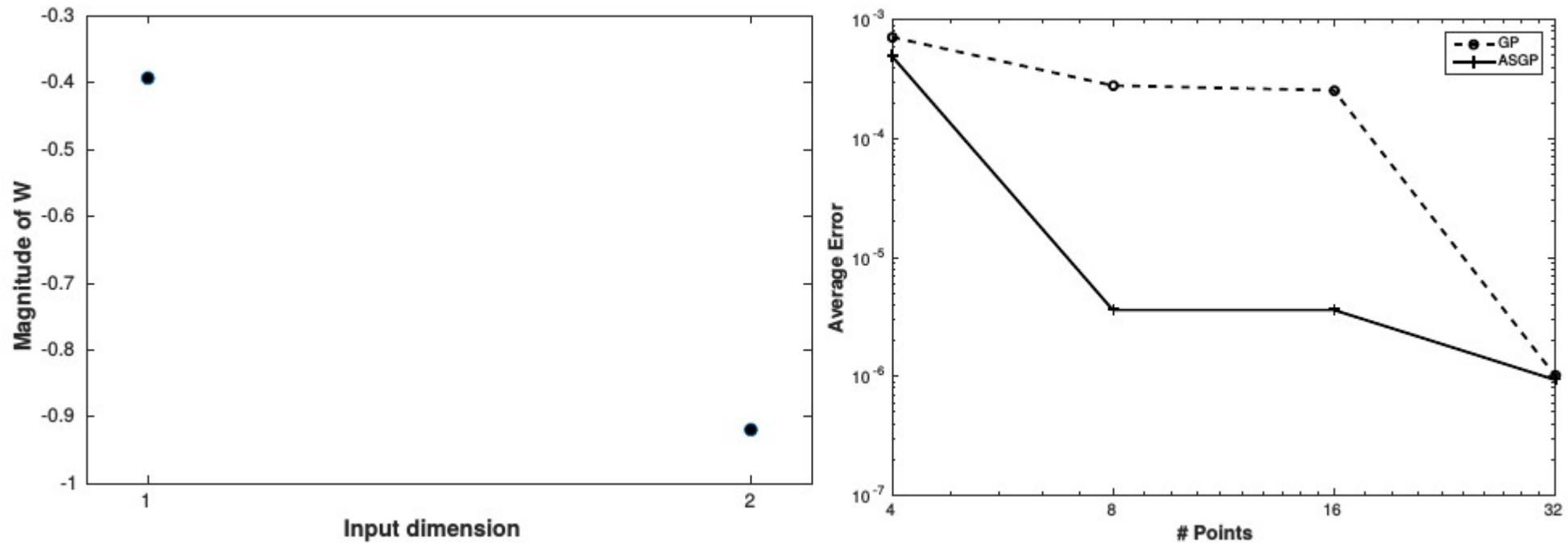


$$g(\cdot) \sim \text{GP}\left(g(\cdot) | m(\cdot), k_0(\cdot, \cdot)\right).$$

$$g(\cdot) | \mathcal{D}_{\text{projected}} \sim \text{GP}\left(g(\cdot) | m^*(\cdot), k_0^*(\cdot, \cdot)\right)$$

# Analytical example in 2d

cf. [global\\_solution\\_yale19/Lecture\\_6/code/AS\\_ex1.py](#)



$$f(x, y) = \exp(0.3x + 0.7y)$$

**The left panel** shows the projection matrix  $W$  of the 1-dimensional AS.

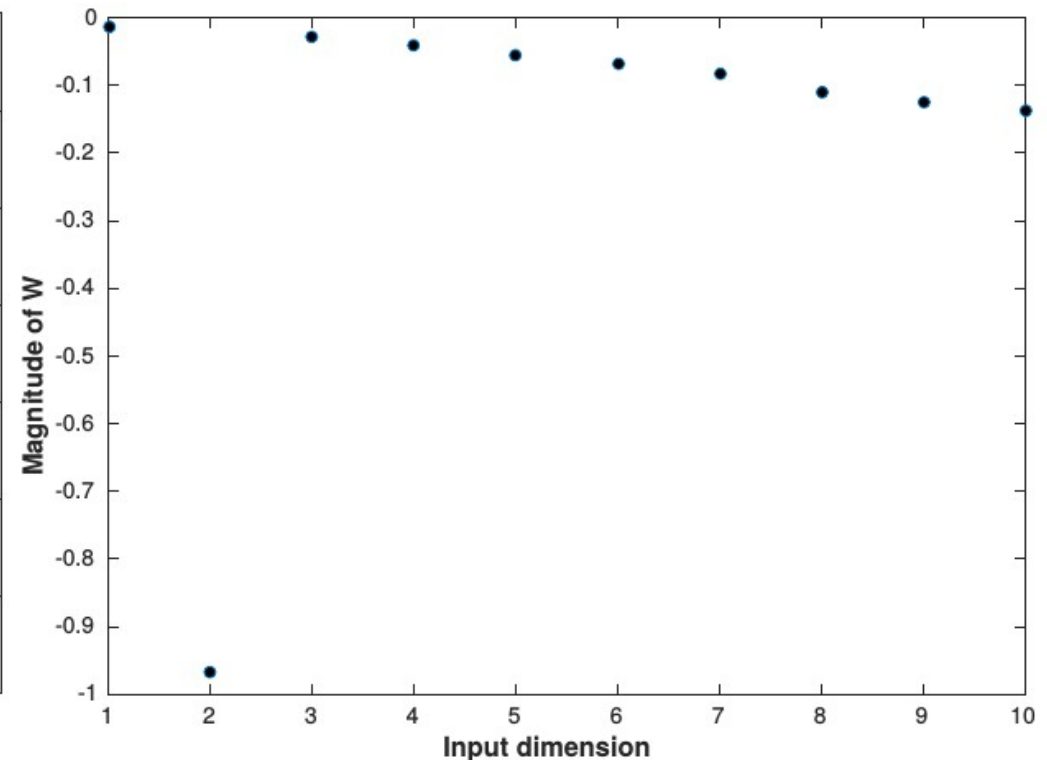
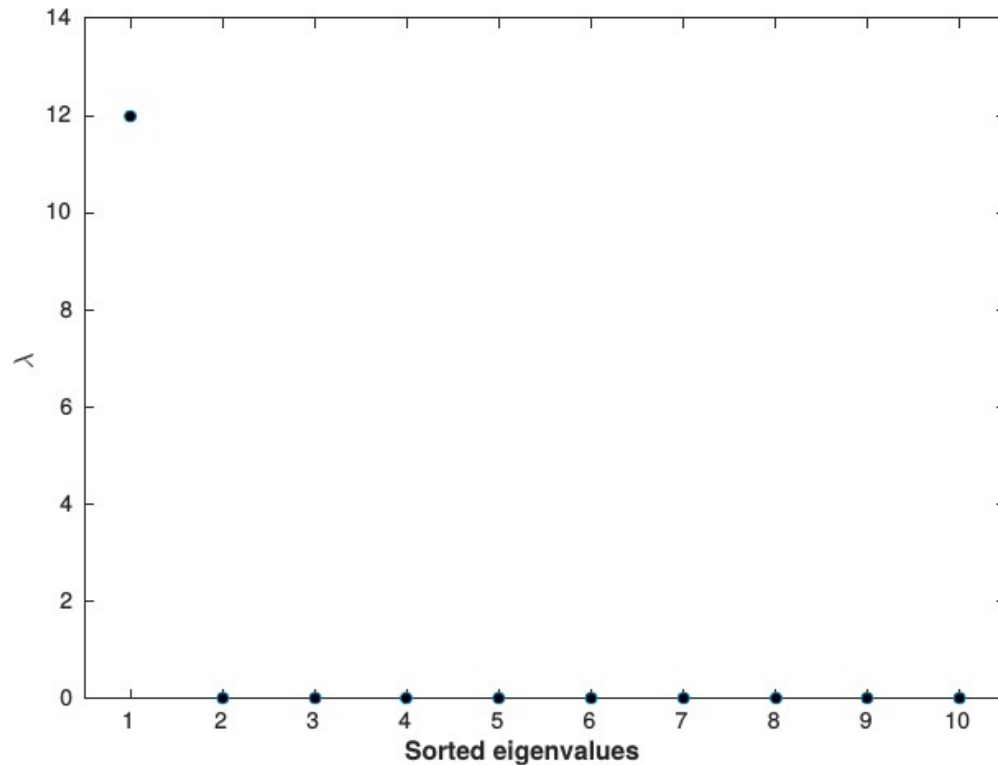
**The right panel** displays a comparison of the interpolation error for 2-dimensional GPs and an 1-dimensional AS of varying resolution, respectively.

# 10d analytical example

cf. [global\\_solution\\_yale19/Lecture\\_6/code/AS\\_ex2.py](#)

$$f : [-1, 1]^{10} \rightarrow \mathbb{R}$$

$$f(x_1, \dots, x_{10}) = \exp(0.01x_1 + 0.7x_2 + 0.02x_3 + 0.03x_4 + 0.04x_5 + 0.05x_6 + 0.06x_7 + 0.08x_8 + 0.09x_9 + 0.1x_{10})$$



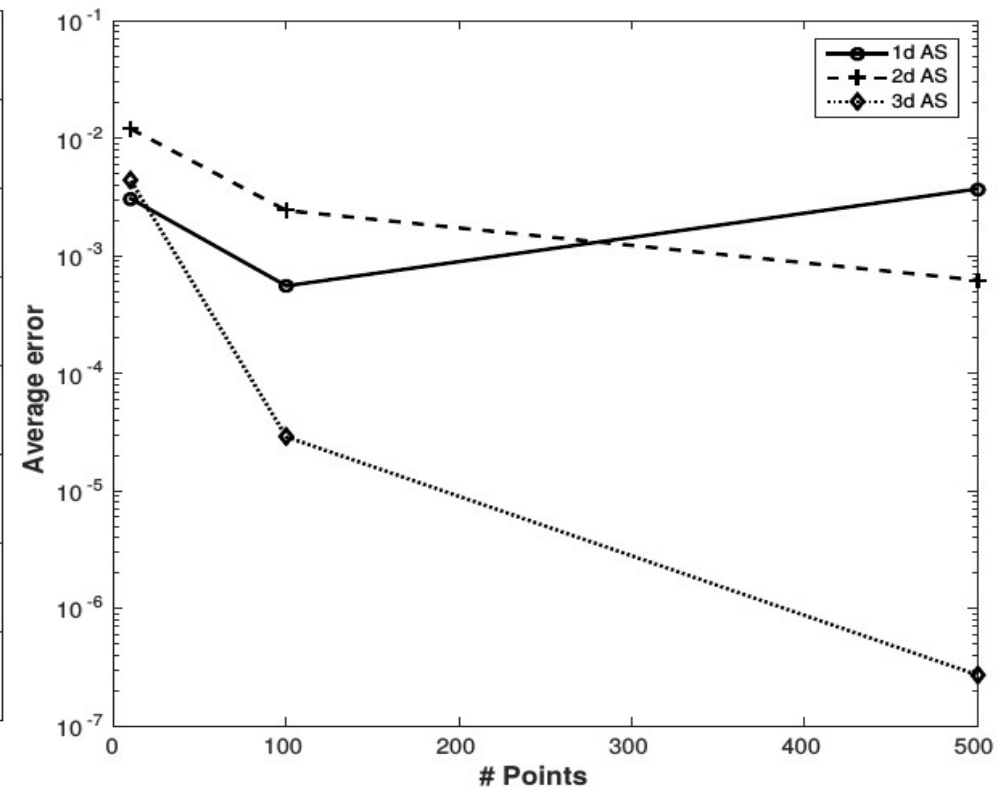
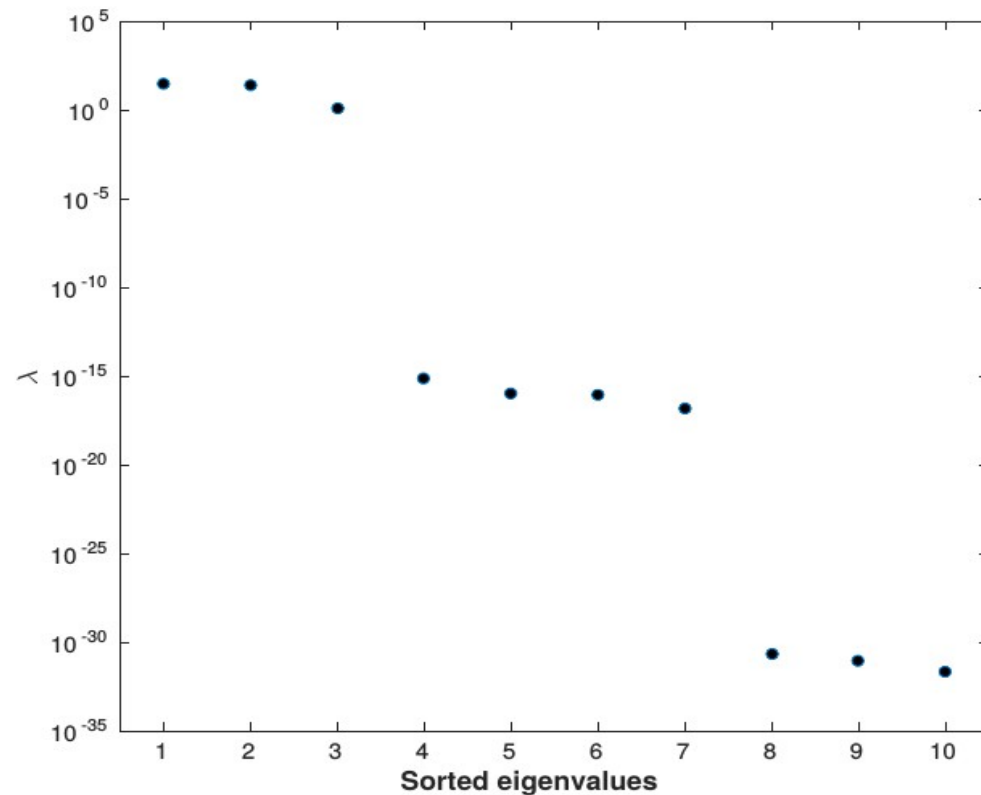
# 10d analytical example: Gap

cf. [global\\_solution\\_yale19/Lecture\\_6/code/AS\\_ex3.py](#)

$$f : [-1, 1]^{10} \rightarrow \mathbb{R}$$

$$f(x_1, \dots, x_{10}) = x_2 \cdot x_3 \cdot \exp(0.01x_1 + 0.7x_2 + 0.02x_3 + 0.03x_4 + 0.04x_5 + 0.05x_6 + 0.06x_7 + 0.08x_8 + 0.09x_9 + 0.1x_{10}).$$

ASGP surrogates of dimension  $d = \{1, 2, 3\}$



# Active Subspaces versus PCA

- Some comment on the similarities and discrepancies between PCA and AS:
- PCA identifies a projection of the input space.
- The goal of this projection, however, is not the same as in AS.
- PCA picks the projection that minimizes the mean square reconstruction error of the input  $\mathbf{x}$  that is, it minimizes  $E[\|\mathbf{x} - \mathbf{W}(\mathbf{W}^T \mathbf{x})\|^2]$ , where the expectation is with respect to the input-generating distribution.
- AS has an objective that is very different to that of PCA:  
AS focuses on finding a  $\mathbf{W}$  that allows us to approximate  $f(\mathbf{x})$  with a function of the form  $h(\mathbf{W}\mathbf{x})$  as well as possible.
- Even though AS has not (yet) been formulated to directly minimize the mean square error  $E[(f(\mathbf{x}) - h(\mathbf{W}\mathbf{x}))^2]$ , it was shown by Constantine that the mean square error is bounded by a term proportional to the sum of the neglected eigenvalues of  $\mathbf{C}$ .
  - PCA focuses on finding the best linear projection that allows the reconstruction of the input, whereas AS focuses on the search for the best linear projection that enables the reconstruction of the response surface  $f(\mathbf{x})$ .

## II. Switching gears: Dynamic Programming



# Test model – growth model

To demonstrate the capabilities of our algorithm, we choose an **infinite-horizon discrete-time multi-dimensional stochastic optimal growth model** (see, e.g, Cai & Judd (2014), and references therein).

Workhorse for studying methods for solving high-dimensional economic models.

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 (=dim)** considered.

→ there are  $D$  sectors with **capital**  $\mathbf{k}_t = (k_{t,1}, \dots, k_{t,D})$

and elastic **labour supply**  $\mathbf{l}_t = (l_{t,1}, \dots, l_{t,D})$

## Growth model (II)

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**.



# The formal multi-*d* Growth Model

See, e.g. Cai & Judd (2014) and references therein

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

$$s.t.$$
$$k_j^+ = (1 - \delta) \cdot k_j + I_j + \epsilon_j, \quad j = 1, \dots, D$$
$$\Gamma_j = \frac{\zeta}{2} k_j \left( \frac{I_j}{k_j} - \delta \right)^2, \quad j = 1, \dots, D$$
$$\sum_{j=1}^D (c_j + I_j - \delta \cdot k_j) = \sum_{j=1}^D (f(k_j, l_j) - \Gamma_j),$$

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

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

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

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
$\sigma$	0.01
$D$	$\{1, \dots, 500\}$

**k: continuous states**

$$\epsilon_{t,j} \sim \mathcal{N}(0, \sigma^2)$$

# Value function iteration

**Data:** Initial guess  $V_{next}$  for the next period's value function. Approximation accuracy  $\bar{\eta}$ .  
**Result:** The (approximate) equilibrium 3D policy functions  $\xi^* = \{\xi_1^*, \dots, \xi_{3D}^*\}$  and the corresponding value function  $V^*$ .

Set iteration step  $s = 1$ .

**while**  $\eta > \bar{\eta}$  **do**

Generate  $n$  training inputs  $\mathbf{X} = \{\mathbf{k}_i^s : 1 \leq i \leq n\} \in [\underline{\mathbf{k}}, \bar{\mathbf{k}}]^D$

**for**  $\mathbf{k}_i^s \in \mathbf{X}$  **do**

Compute the maximization problem

$$\begin{aligned} V(\mathbf{k}_i^s) &= \max_{\mathbf{l}, \mathbf{c}, \mathbf{l}} \left( u(\mathbf{c}, \mathbf{l}) + \beta \mathbb{E} \left\{ V_{next}(\mathbf{k}_i^+) \right\} \right), \text{ s.t.} \\ k_j^+ &= (1 - \delta) \cdot k_j + I_j + \epsilon_j, \quad j = 1, \dots, D \\ \Gamma_j &= \frac{\zeta}{2} k_j \left( \frac{I_j}{k_j} - \delta \right)^2, \quad j = 1, \dots, D \\ \sum_{j=1}^D (c_j + I_j - \delta \cdot k_j) &= \sum_{j=1}^D (f(k_j, l_j) - \Gamma_j) \end{aligned}$$

given the next period's value function  $V_{next}$ .  
Set the training targets for the value function:  $t_i = V(\mathbf{k}_i^s)$ .  
If required, set the training targets to learn the policy function  $\xi_{ji}(\mathbf{k}_i^s) \in \arg \max_{p_j} V(\mathbf{k}_i^s)$ .

**end**

Set  $\mathbf{t} = \{t_i : 1 \leq i \leq n\}$ .

Given  $\{\mathbf{X}, \mathbf{t}\}$ , learn a surrogate  $V_{surrogate}$  of  $V$  with ASGP (or GP).

Set  $\xi_j = \{\xi_{ji} : 1 \leq i \leq n\}$ .

Given  $\{\mathbf{X}, \xi_j\}$ , learn a surrogate of the policy  $\xi_j$  with ASGP (or GP).

Calculate (an approximation for) the error, e.g.,  $\eta = \|V_{surrogate} - V_{next}\|_\infty$ .

Set  $V_{next} = V_{surrogate}$ .

Set  $s = s + 1$ .

**end**

$V^* = V_{surrogate}$ .

$\xi^* = \{\xi_1, \dots, \xi_{3D}\}$ .

**“Hybrid parallel”**  
Implementation  
(Shared & distributed memory parallelism)

# Algorithmic Complexity of GPR-DP

**N**: number of observations.

The computational cost of standard GPR is dominated by the need to construct the Cholesky decomposition of the  $N \times N$  covariance matrix at each step of the likelihood optimization that is –  $O(N^3)$ .

The computational cost of **AS** arises from a single SVD of an  $N \times N$  matrix, plus the cost of a standard GP regression that is, it is also  $O(N^3)$ .

In both cases, the right number of observations  $N$  depends on the function smoothness and the input dimensionality:

**D** for the GP regression and **d** for the ASGP regression.

The complete details of this **relationship are not entirely understood theoretically** and are well beyond the scope of the present lecture. However, we observe in numerical experiments that the number of samples required by GP regression, is much larger than the number of samples required by AS-GPR,

$$N_{\text{GPR}} \gg N_{\text{AS-GPR}} \text{ when } D \gg d.$$

# Setup of Code

global\_solution\_yale19/Lecture\_6/code/growth\_model\_GPR

```
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 GPR  $\leftrightarrow$  IPOPT.

**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

```

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 postprocessing as post                #computes the L2 and Linfinity error of the model
import numpy as np

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

for i in range(numstart, numits):
    # terminal value function
    if (i==1):
        print "start with Value Function Iteration"
        interpol.GPR_init(i)

    else:
        print "Now, we are in Value Function Iteration step", i
        interpol_iter.GPR_iter(i)

#=====
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_postprocess)

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

```

# Code snippet – parameters.py

```
#=====
# How many training points for GPR
n_agents= 1 # number of continuous dimensions of the model
No_samples = 10*n_agents

# control of iterations
numstart = 1 # which is iteration to start (numstart = 1: start from scratch, number=/0: restart)
numits = 7 # which is the iteration to end

filename = "restart/restart_file_step_" #folder with the restart/result files

#=====

# Model Paramters

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
k_bar=0.2
k_up=3.0
range_cube = k_up - k_bar # range of  $[0..1]^d$  in 1D

# Ranges for Controls
c_bar=1e-2
c_up=10.0

l_bar=1e-2
l_up=10.0

inv_bar=1e-2
inv_up=10.0

#=====

# Number of test points to compute the error in the postprocessing
No_samples_postprocess = 20
```

# 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" GPR)

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

    # 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

    #transform to comp. domain of the model
    knext_cube = box_to_cube(knext)

    # initialize correct data format for training point
    s = (1,n_agents)
    Xtest = np.zeros(s)
    Xtest[0,:] = knext_cube

    # interpolate the function, and get the point-wise std.
    V_old, sigma_test = gp_old.predict(Xtest, return_std=True)

    VT_sum = utility(cons, lab) + beta*V_old

    return VT_sum

```



# Code snippet – interpolate\_iter.py

```
def GPR_iter(iteration):

    # Load the model from the previous iteration step
    restart_data = filename + str(iteration-1) + ".pcl"
    with open(restart_data, 'rb') as fd_old:
        gp_old = pickle.load(fd_old)
        print "data from iteration step ", iteration -1 , "loaded from disk"
    fd_old.close()

    ##generate sample aPoints
    np.random.seed(666) #fix seed
    dim = n_agents
    Xtraining = np.random.uniform(k_bar, k_up, (No_samples, dim))
    y = np.zeros(No_samples, float) # training targets

    # solve bellman equations at training points
    for iI in range(len(Xtraining)):
        y[iI] = solver.iterate(Xtraining[iI], n_agents, gp_old)[0]

    #for iI in range(len(Xtraining)):
        #print Xtraining[iI], y[iI]

    # Instantiate a Gaussian Process model
    #kernel = 1.0 * RBF(length_scale=1.0, length_scale_bounds=(1e-1, 10.0))

    | kernel = 1.0 * RBF(length_scale=100.0, length_scale_bounds=(1e-1, 2e2)) \
      + WhiteKernel(noise_level=1, noise_level_bounds=(1e-3, 1e+0))

    #kernel = 1.0 * RBF(length_scale=100.0, length_scale_bounds=(1e-1, 2e2))
    #kernel = 1.0 * Matern(length_scale=1.0, length_scale_bounds=(1e-1, 10.0), nu=1.5)

    gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10)

    # Fit to data using Maximum Likelihood Estimation of the parameters
    gp.fit(Xtraining, y)

    ##save the model to a file
    output_file = filename + str(iteration) + ".pcl"
    print output_file
    with open(output_file, 'wb') as fd:
        pickle.dump(gp, fd, protocol=pickle.HIGHEST_PROTOCOL)
        print "data of step ", iteration , " written to disk"
        print " -----"
    fd.close()
```



# 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

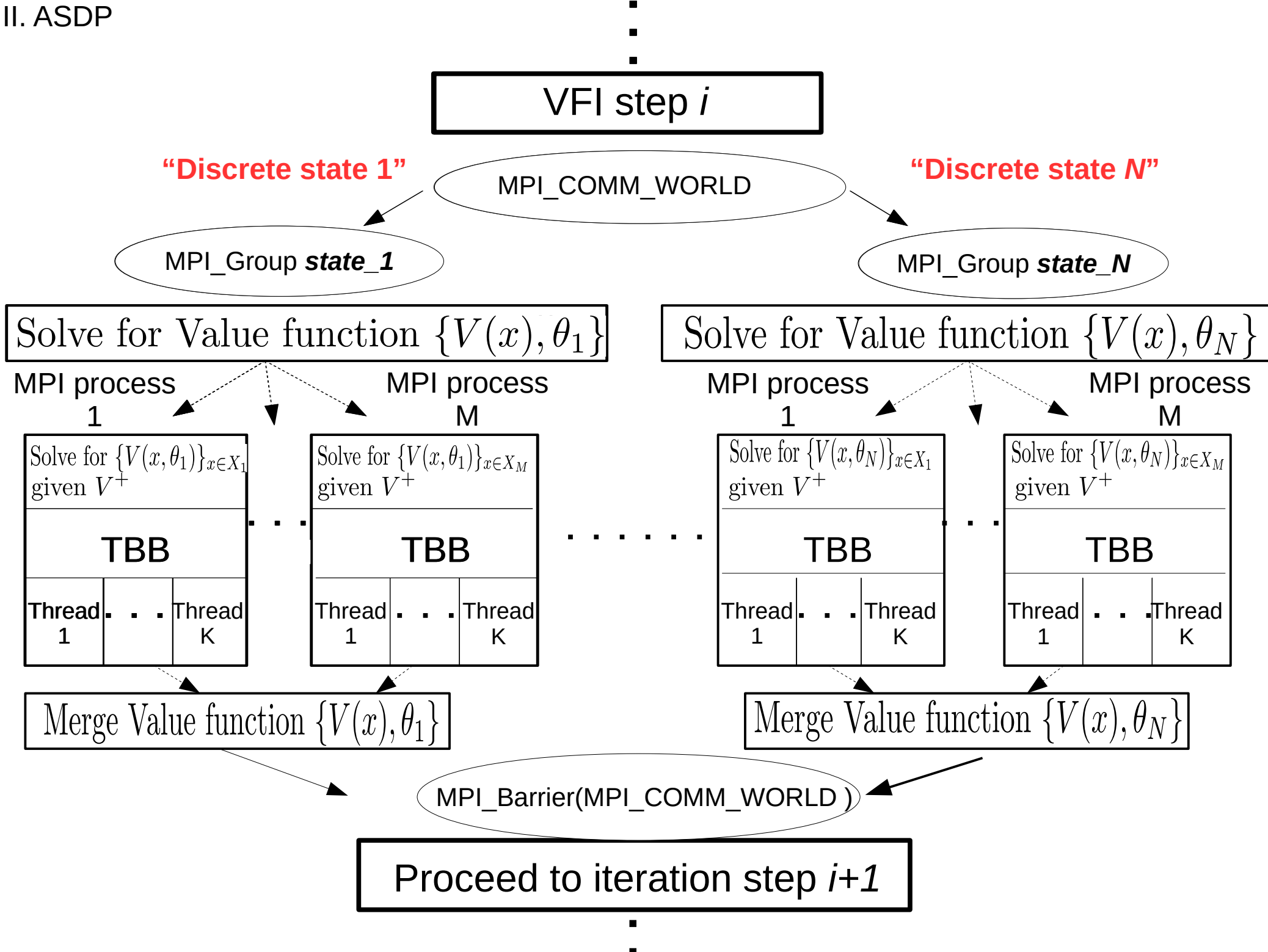
On Yale's HPC cluster GRACE (ssh -X NETID@grace.hpc.yale.edu)

```
>cd ~
```

```
>vi .bashrc
```

→ add the following lines to the .bashrc

```
module load Langs/Python/2.7.15-anaconda
```

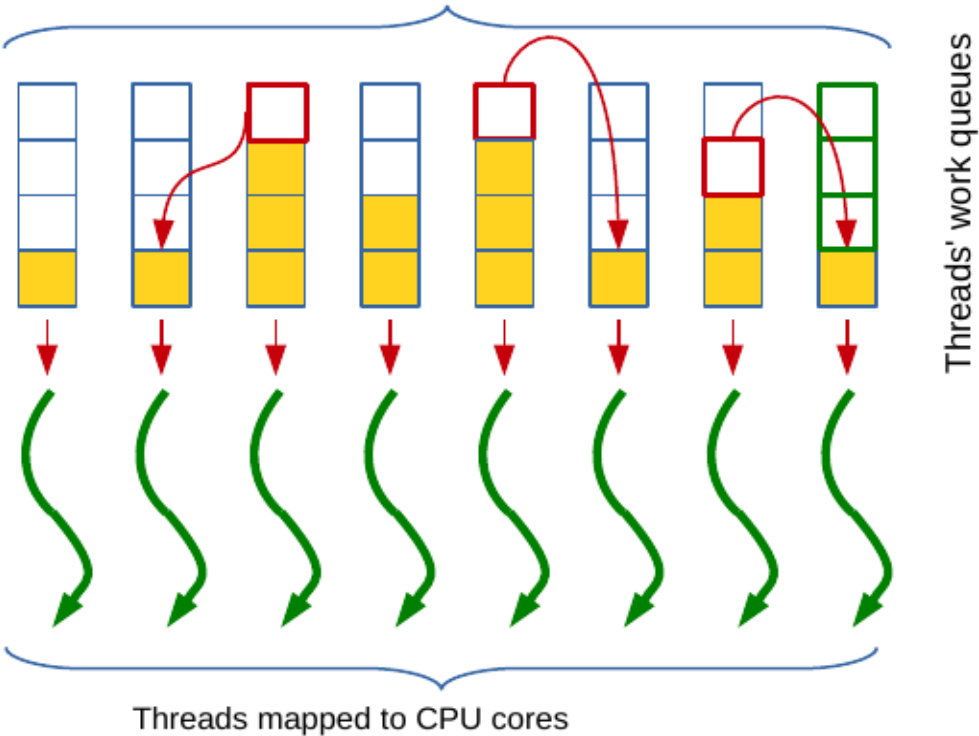


# Node-level parallelization

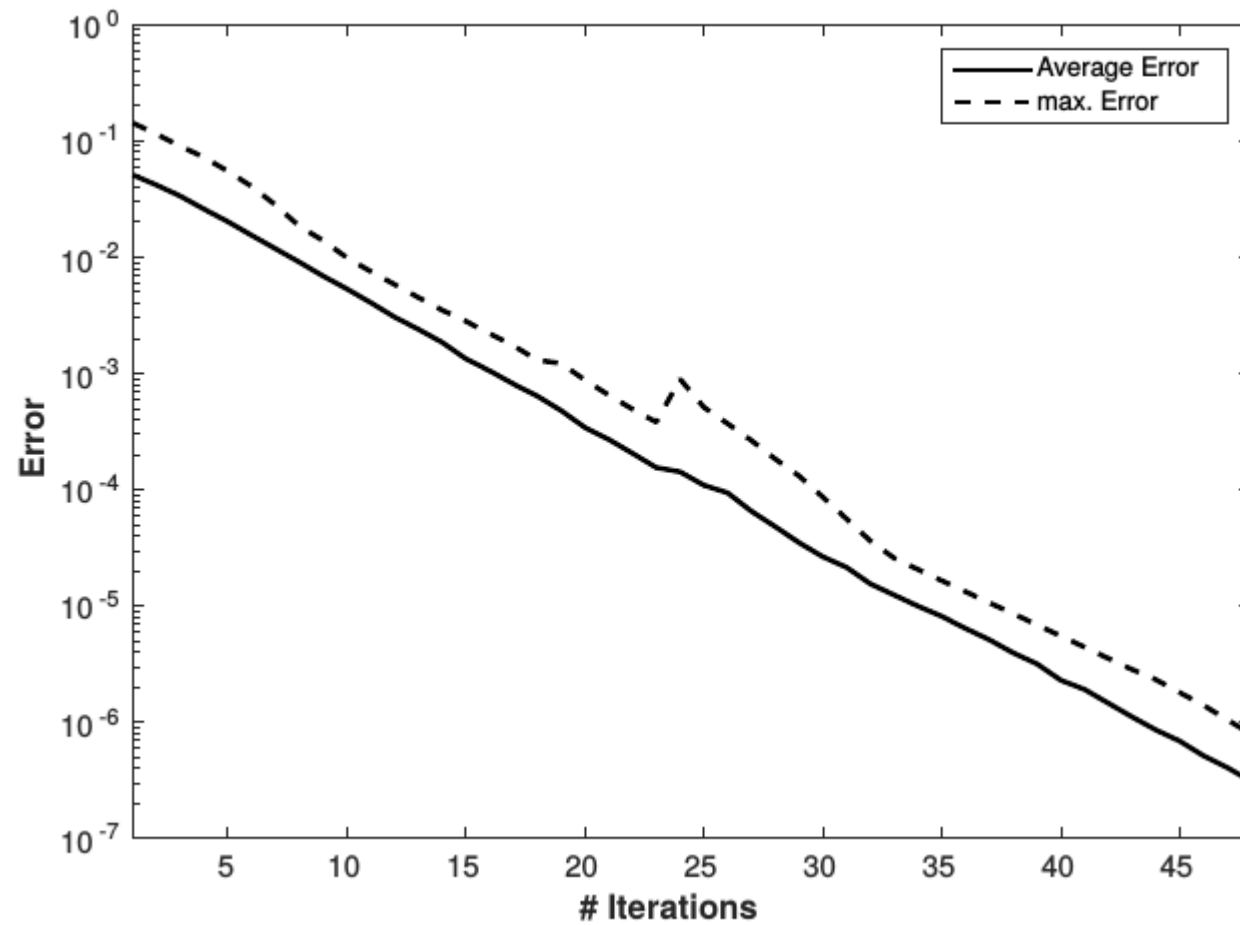
Individual work tasks – points within a grid refinement level



TBB initially distributes work equally among parallel threads



# 1d Growth Model – converges



# Pin-down a 1d VF by GPs

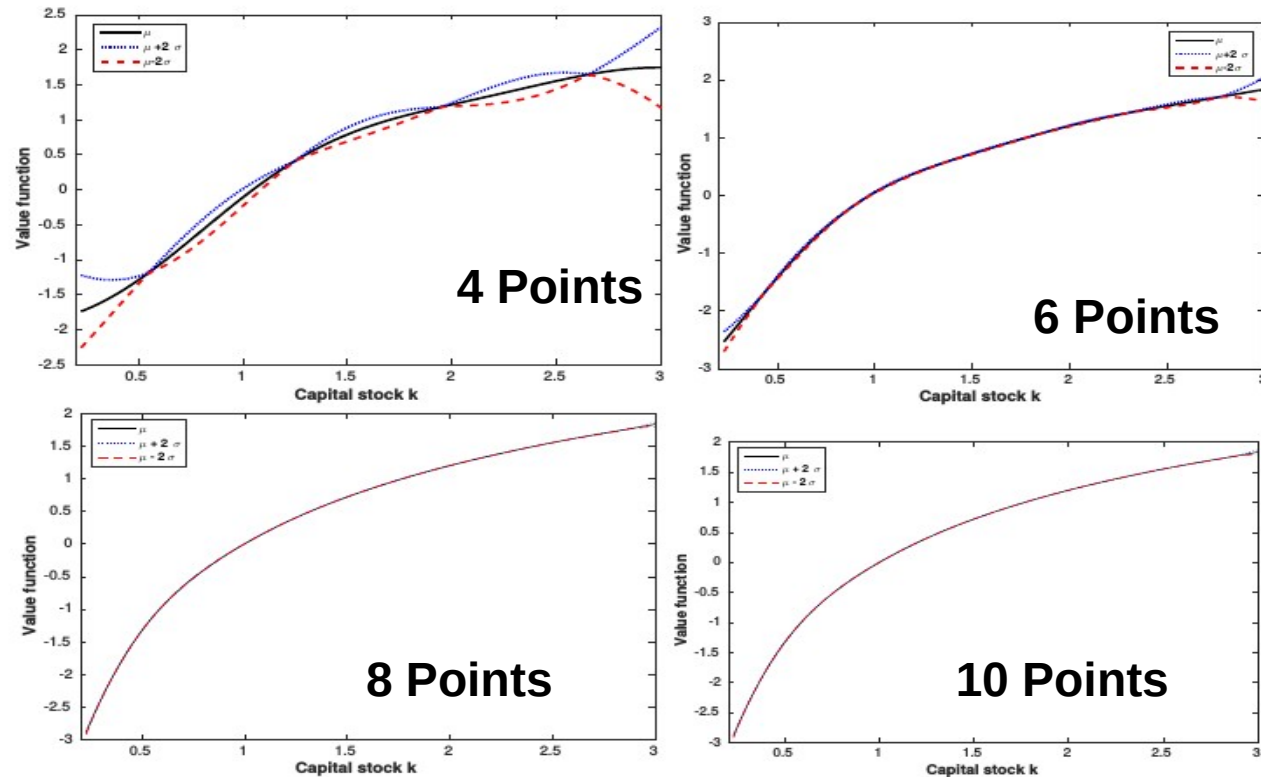
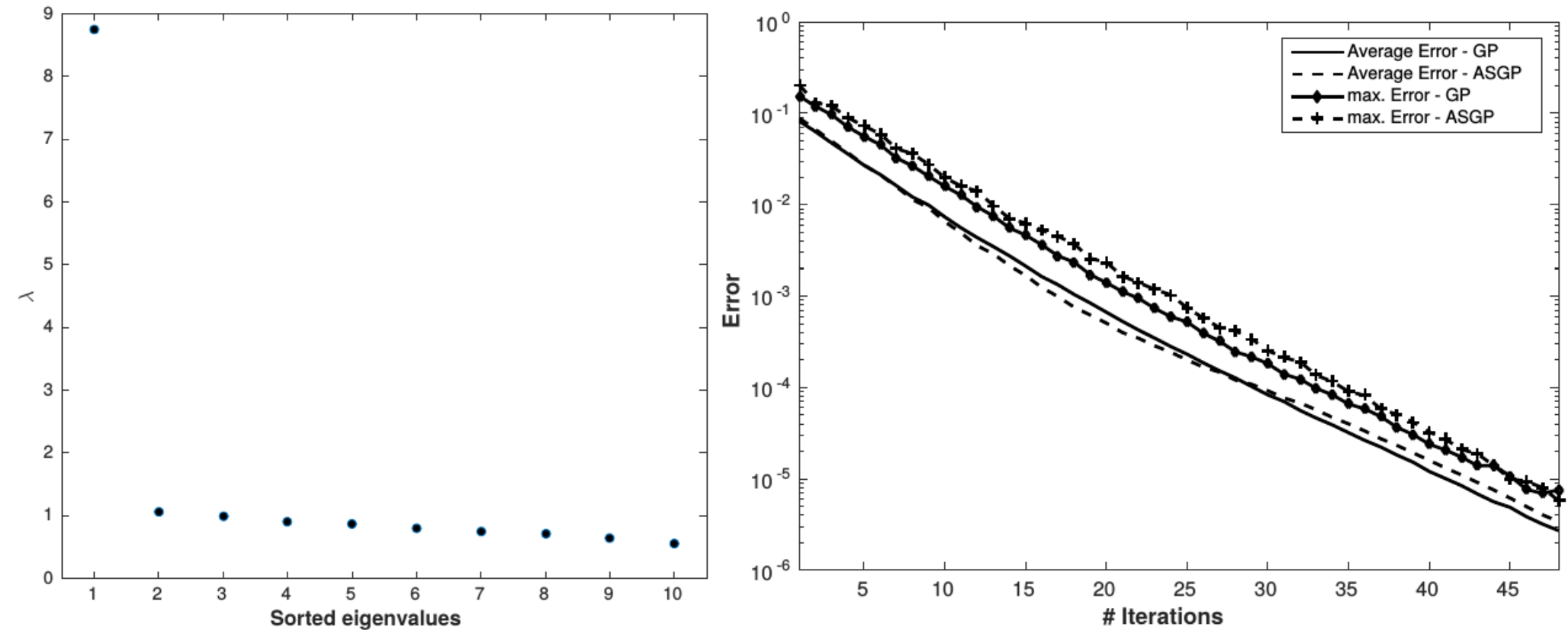


Figure 5: Above's four panels display the predictive mean value function from a 1-dimensional growth model at convergence, and the 95% confidence interval. The upper left figure was constructed based on only 4 sample points in the state space, the upper right on 6 sample points. The lower left was a result of 8 sample points, and the lower right is based on 10 sample points that pin down the function.

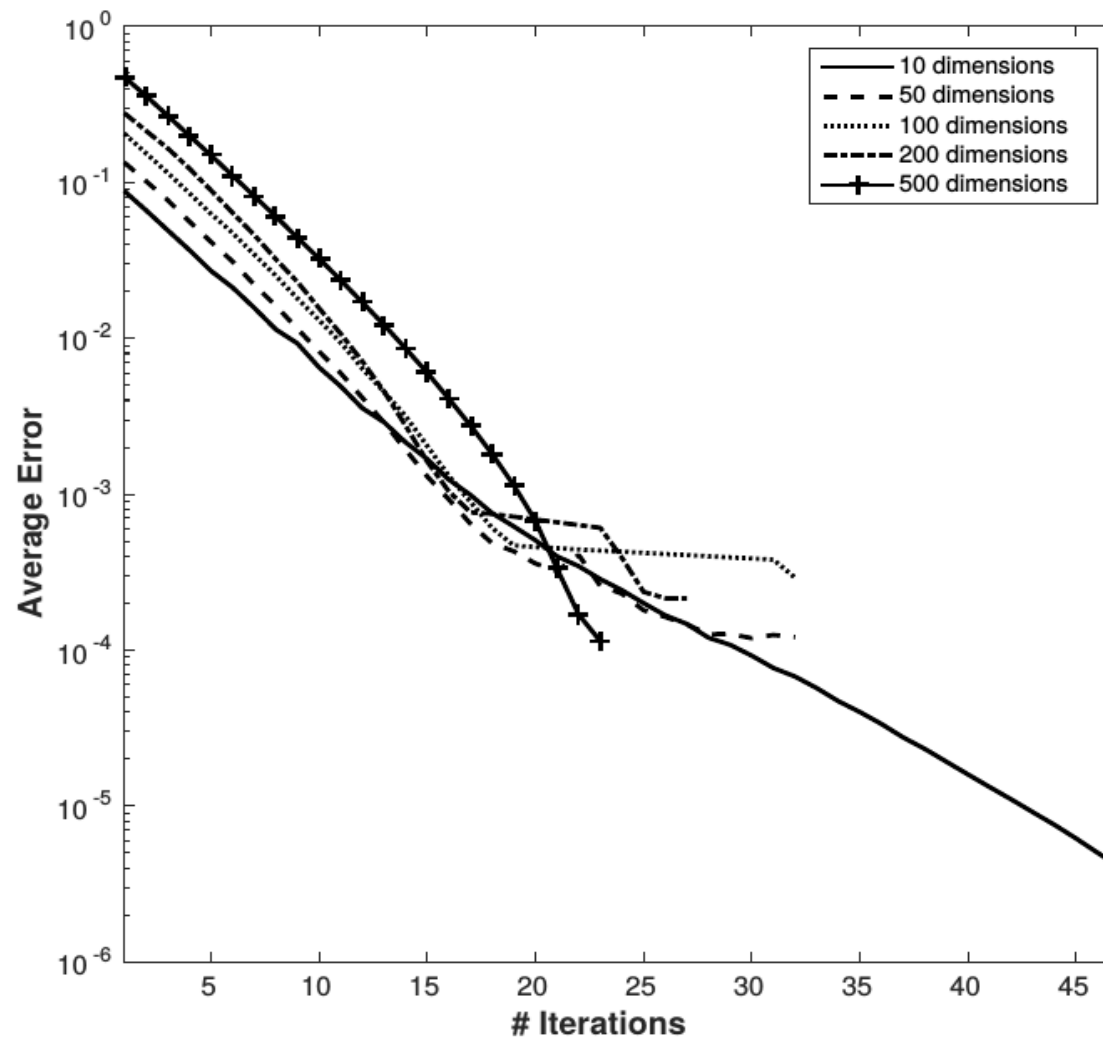
# 10d Growth Model – converges



**Left panel:** sorted eigenvalues of of the 10-dimensional OG model.

**Right panel:** decreasing maximum and average error for 10-D OG model, computed either by GPs or ASGP with an AS of dimension 1, respectively.

# Growth model – 1 active subspace





# Context

- **500d growth model**: every individual observation used to train the ASGP consists of solving an optimization problem with **500 continuous states** and **1,500 controls**.
- **far beyond what has been done so far in the literature** in the context of computing global solutions to economic problems. Cai & Judd (2014), for example or up to **4 continuous states**.
- Note that **adaptive sparse grid-based solution algorithms** (e.g. Brumm & Scheidegger (2017)) as well as algorithms that are based on Smolyak's method (Judd et. al. (2014), Kruger & Kubler (2004)) **would not be able solve models of this size**.
- Brumm & Scheidegger (2017) were able to compute global solutions for models up to **100 dimensions** with adaptive sparse grids and a massively parallelized code, whereas Kruger & Kubler (2004) deal with up to **20 continuous states** when employing Smolyak's method.
- Their underlying **data structures of sparse grids become so complex** and too slow to operate on that they in practice become **un - operational** for problems of this size.
- the very high dimensionality required e.g. in large-scale multi-country OLG models.

## Dynamic Programming on irregularly-shaped geometries

- Natural modelling geometry of economic models often not a hypercube, but rather **a simplex** (e.g., Brumm & Kubler (2014)), or a **hyper-ball** kind of ergodic set (Judd et. al. (2014), Maliar & Maliar (2015)).
  - We need to be able to perform value function iteration or time iteration on “arbitrary/irregularly-shaped” geometries.
  - Want to focus on the region of interest (**limited computational budget**).

# Recall – GPs


(e.g. Murphy (2012), Rasmussen & Williams (2006), with references therein)

GPs: grid-free way of constructing interpolators

→ I can add geometry-free observations to  $D$ !

Training set:  $D = \{(\mathbf{x}_i, y_i) | i = 1, \dots, n\}$

$$\begin{pmatrix} \mathbf{f} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N} \left( \begin{pmatrix} \boldsymbol{\mu} \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{pmatrix} \right) \quad \begin{aligned} p(\mathbf{f}_* | \mathbf{X}_*, \mathbf{X}, \mathbf{f}) &= \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) \\ \boldsymbol{\mu}_* &= \boldsymbol{\mu}(\mathbf{X}_*) + \mathbf{K}_*^T \mathbf{K}^{-1} (\mathbf{f} - \boldsymbol{\mu}(\mathbf{X})) \\ \boldsymbol{\Sigma}_* &= \mathbf{K}_{**} - \mathbf{K}_*^T \mathbf{K}^{-1} \mathbf{K}_* \end{aligned}$$


 Test point = interpolation at  $\mathbf{X}^*$

# Simulate the economy

Judd et. al. (2014), Maliar & Maliar (2015)

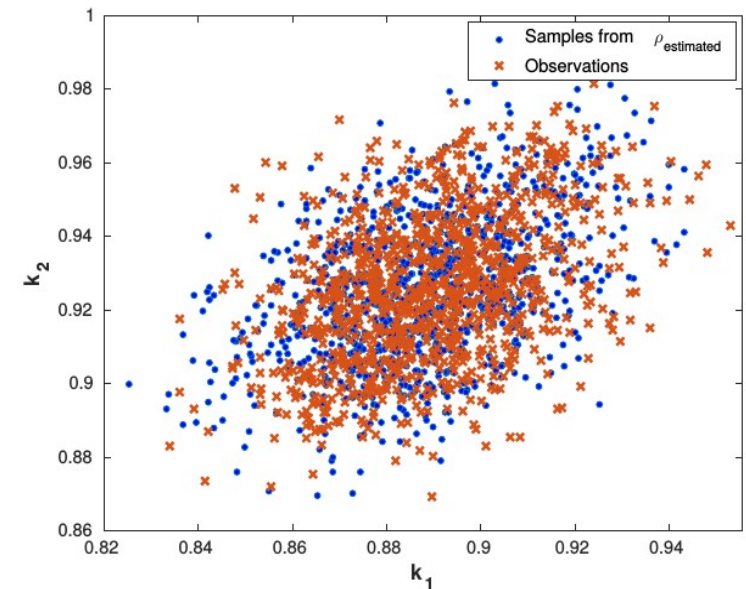
## 1. Simulate economy at few points, learn the policy

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

## 2. Approximate density with mixture of Gaussians (e.g. Rasmussen (2000), Blei & Jordan (2005))

$$\rho_{estimated}(\mathbf{x}) = \sum_{m=1}^M \pi_m \mathcal{N}(\mathbf{x} | \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m)$$

→ generate training data; plug them into VFI



# DP on ergodic sets

**Data:** Initial guess  $V_{next}$  for the next period's value function. Approximation accuracy  $\bar{\eta}$ .

**Result:** The  $m$  (approximate) equilibrium policy functions  $\xi^*$  and the corresponding value function  $V^*$  on  $\Omega_{ergodic}$ .

Set iteration step  $s = 1$ .

**while**  $\eta > \bar{\eta}$  **do**

    Determine  $\Omega_{ergodic}$  by using (64) and (65).

    Generate  $n$  training inputs  $\mathbf{X}_{ergodic} = \{\mathbf{k}_i^s : 1 \leq i \leq n\} \in \Omega_{ergodic}$

**for**  $\mathbf{k}_i^s \in \mathbf{X}_{ergodic}$  **do**

        Evaluate the Bellman operator  $TV^s(\mathbf{k}_i^s)$  (see Eq. (5))

        given next period's value function  $V_{next}$ .

        Set the training targets for the value function:  $t_i = TV(\mathbf{k}_i^s)$ .

        If required, set the training targets to learn the  $j$ -th policy function:

$\xi_{j_i}(\mathbf{k}_i^s) \in \arg \max_{p_j} TV(\mathbf{k}_i^s)$ .

**end**

    Set  $\mathbf{t}_{ergodic} = \{t_i : 1 \leq i \leq n\}$ .

    Given  $\{\mathbf{X}_{ergodic}, \mathbf{t}_{ergodic}\}$ , learn a surrogate  $V_{surrogate}$  of  $V$ .

    Set  $\xi_j = \{\xi_{j_i} : 1 \leq i \leq n\}$ .

    Given  $\{\mathbf{X}_{ergodic}, \xi_j\}$ , learn a surrogate of the policy  $\xi_j$ .

    Calculate (an approximation for) the error, e.g.:  $\eta = \|V_{surrogate} - V_{next}\|_\infty$ .

    Set  $V_{next} = V_{surrogate}$ .

    Set  $s = s + 1$ .

**end**

$V^* = V_{surrogate}$ .

$u^* = \{\xi_1, \dots, \xi_m\}$ .

► Probably only every 5 to 10 steps...

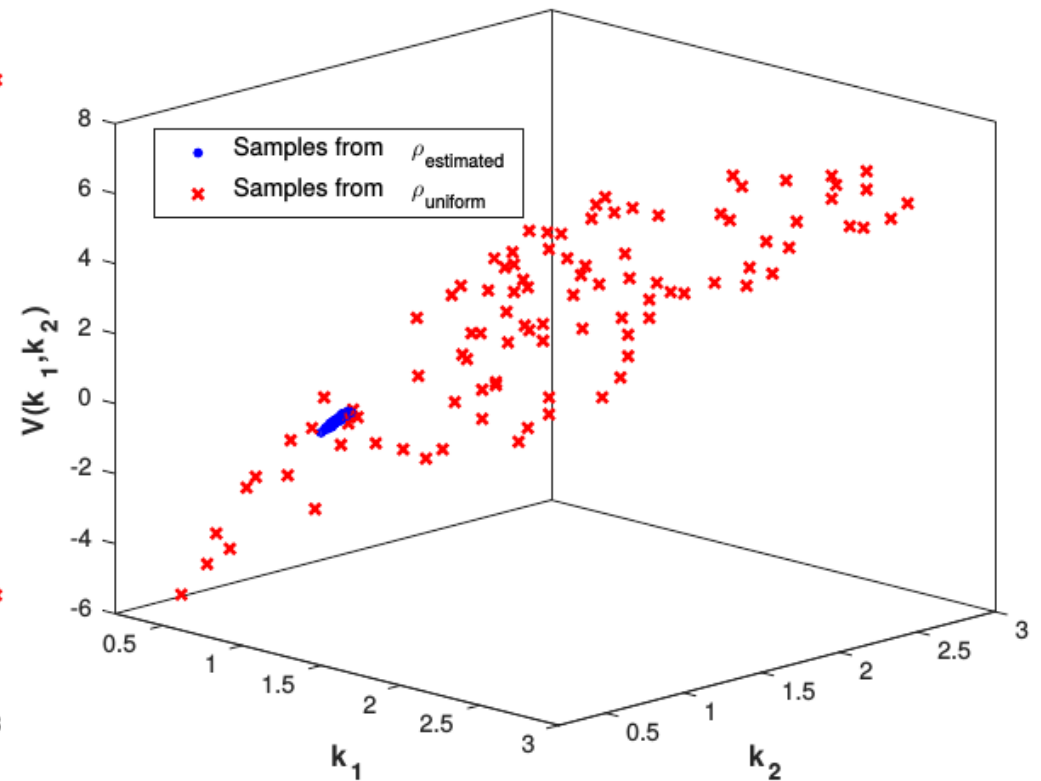
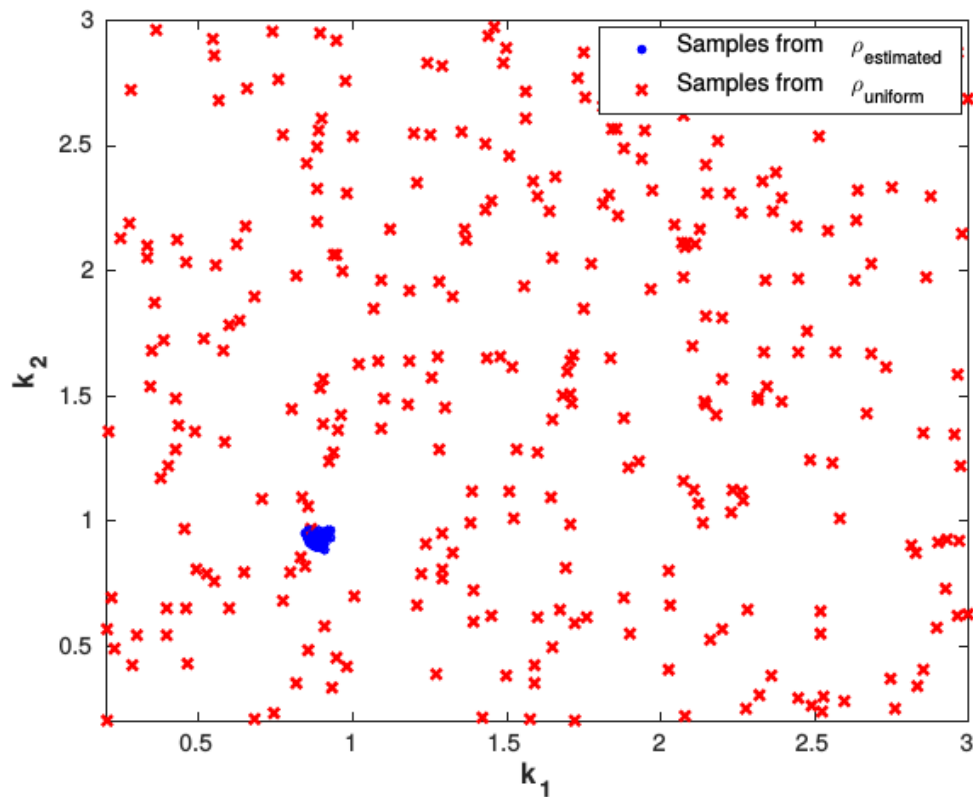
$$\mathbf{X}_{ergodic} = \left\{ \mathbf{x}_{ergodic}^{(1)}, \dots, \mathbf{x}_{ergodic}^{(N)} \right\}$$

$$\mathbf{t}_{ergodic} = \left\{ t_{ergodic}^{(1)}, \dots, t_{ergodic}^{(N)} \right\}$$

$$\{\mathbf{X}_{ergodic}, \mathbf{t}_{ergodic}\} \in \Omega_{ergodic}$$

**Algorithm 2:** Overview of the critical steps of the VFI algorithm that operates on the ergodic set  $\Omega_{ergodic}$ .

# Focus resources where needed



# Uncertainty Quantification (UQ)

- Uncertainty quantification (UQ) is a field of applied mathematics concerned with the **quantification and propagation of uncertainties** through computational models.
- In the quantitative economics community, UQ should be of paramount interest, as it can help to address questions such as **which parameters are driving the conclusions derived from an economic model**.
- Answering them, in turn, can inform the researcher for example **on which parts of the model she needs to focus** on when calibrating it.

There are various sources of uncertainty that can enter economic models, including:

- parameter uncertainty** (robustness of results)
- interpolation uncertainty** (more data reduce uncertainty)

# Enlarge the state space

- Standard approaches require a **large number of model evaluations**.
- **We get away with a single model evaluation!**
- Since we can deal with very high-dimensional problems, we simply **enlarge the state space** by the parameters of interest, e.g.:

$$\tilde{S} = (\mathbf{k}_t, \gamma)$$

In the growth model, we simply set it to:

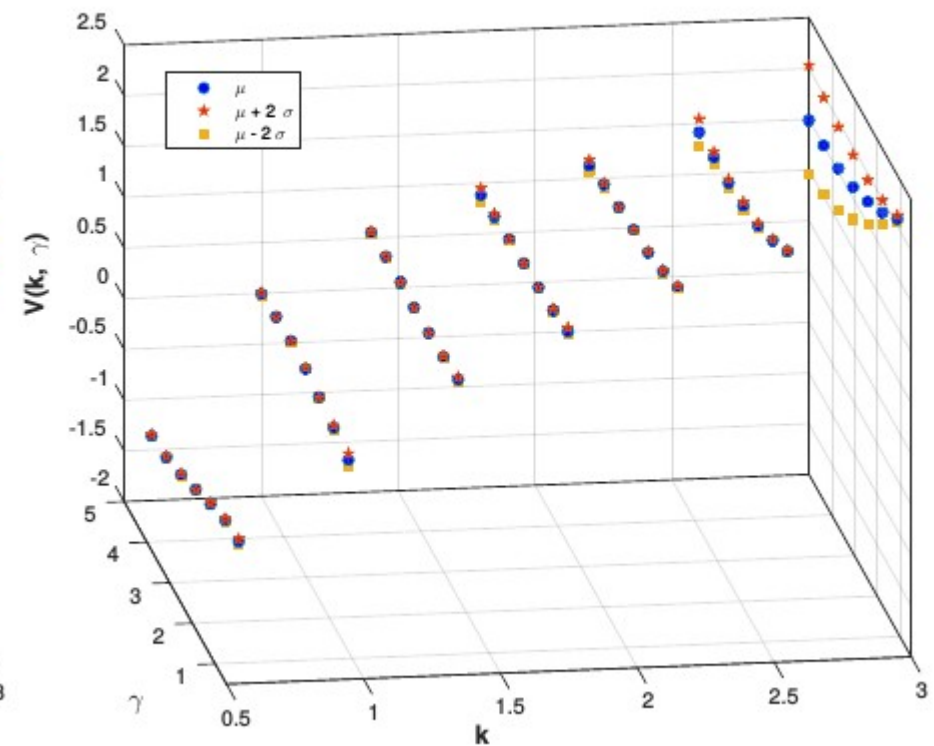
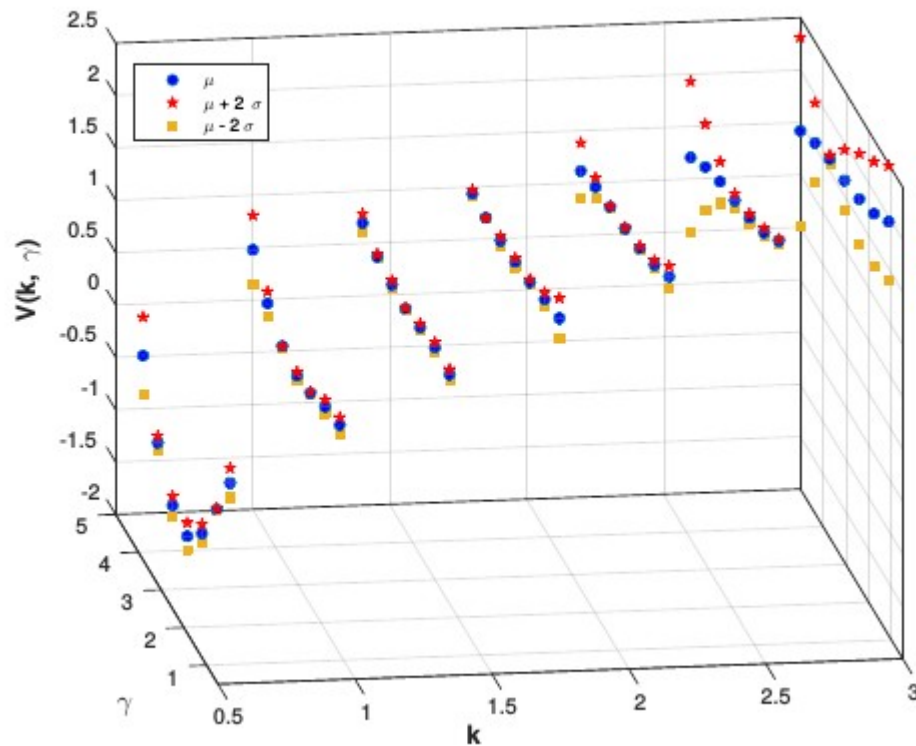
$$[\underline{\mathbf{k}}, \overline{\mathbf{k}}]^D \times [\underline{\gamma}, \overline{\gamma}] = [0.2, 3]^D \times [0.5, 5]$$

- no calibration exercise needed!
- evaluate, e.g., univariate impact of a parameter on the model conclusion.
- **Bring the model and the data together.**



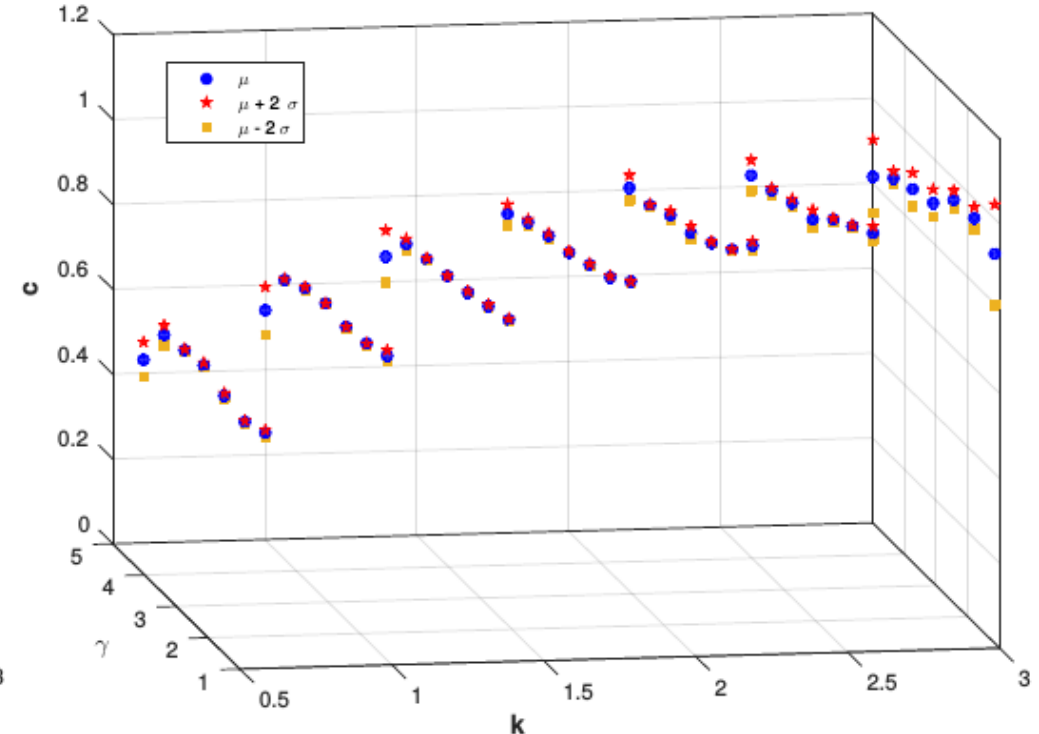
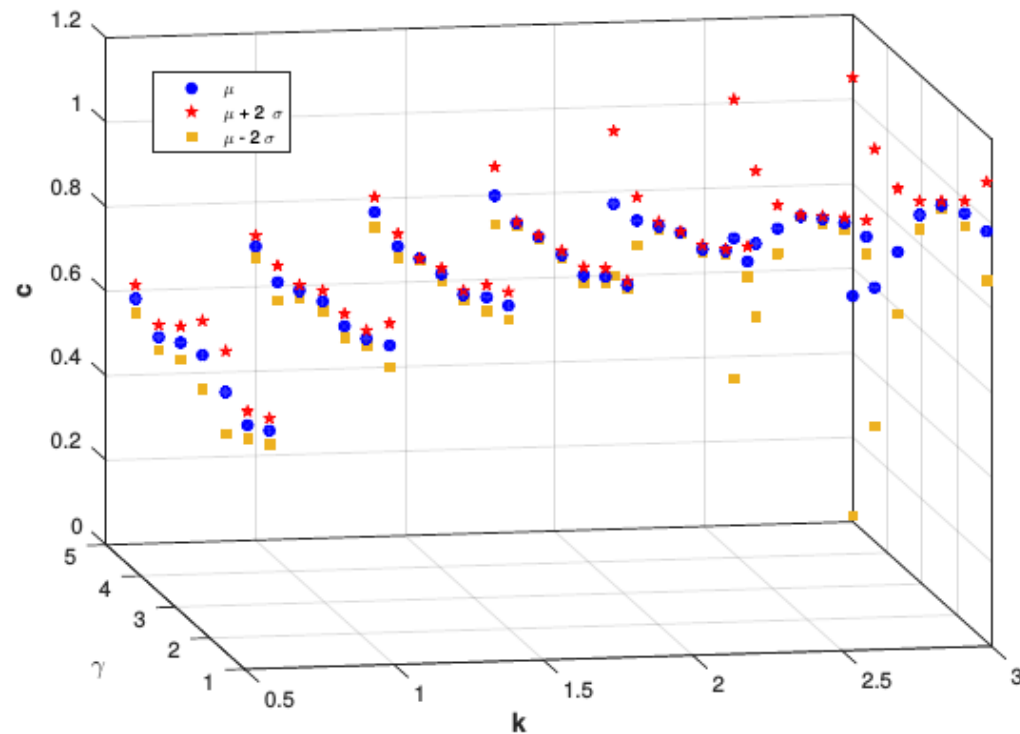
# Value function $V(k, \gamma)$

Solve a DP problem with  $\gamma$  (risk aversion) uniformly sampled from [0.5: 5]  
 Note: other distributions for parameters possible!



**Left:** predictive mean and 95% quantile of the value function (20 sample points)  
**Right:** predictive mean and 95% quantile of the value function (40 sample points)

# Consumption $c(k, \gamma)$



**Left:** predictive mean and 95% quantile of the value function (20 sample points)  
**Right:** predictive mean and 95% quantile of the value function (40 sample points)

# Convergence of ASGDP

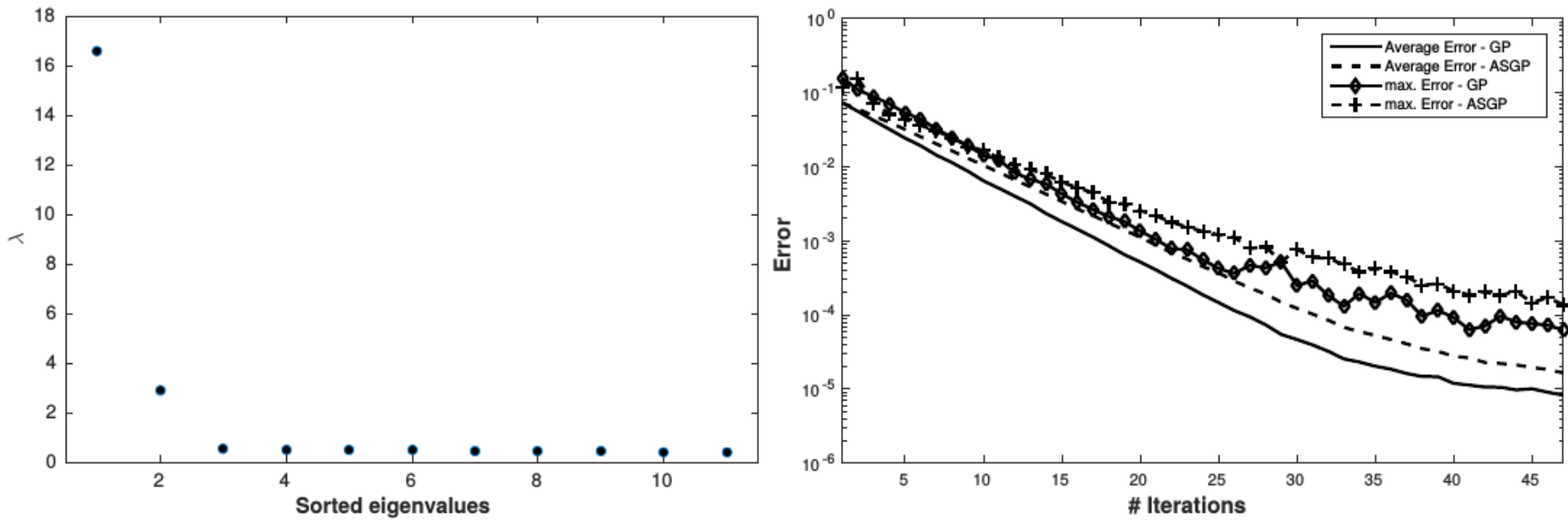


Figure 13: Left panel—Sorted eigenvalues of  $\mathbf{C}_N$  (see Eq. (42)) for the 11-dimensional OG problem with continuous states  $\tilde{S} = (k_1, \dots, k_{10}, \gamma)$ . Right panel—Decreasing maximum and average error for the two 11-dimensional OG models, computed either by GPs or by ASGP with an AS of dimension 1, respectively.)

# Quantity of interest – an example

Jaynes (1982); Harenberg (2017), with references therein

Parameters:

$$\chi = \{\gamma, \delta, \psi, \zeta, \eta\}$$

Parameter	Baseline value	lower bound for $\chi_i$	upper bound for $\bar{\chi}_i$
$\gamma$	2.0	0.5	5.0
$\delta$	0.025	0.02	0.03
$\psi$	0.36	0.32	0.4
$\zeta$	0.5	0.0	1.0
$\eta$	1.0	0.0	2.0

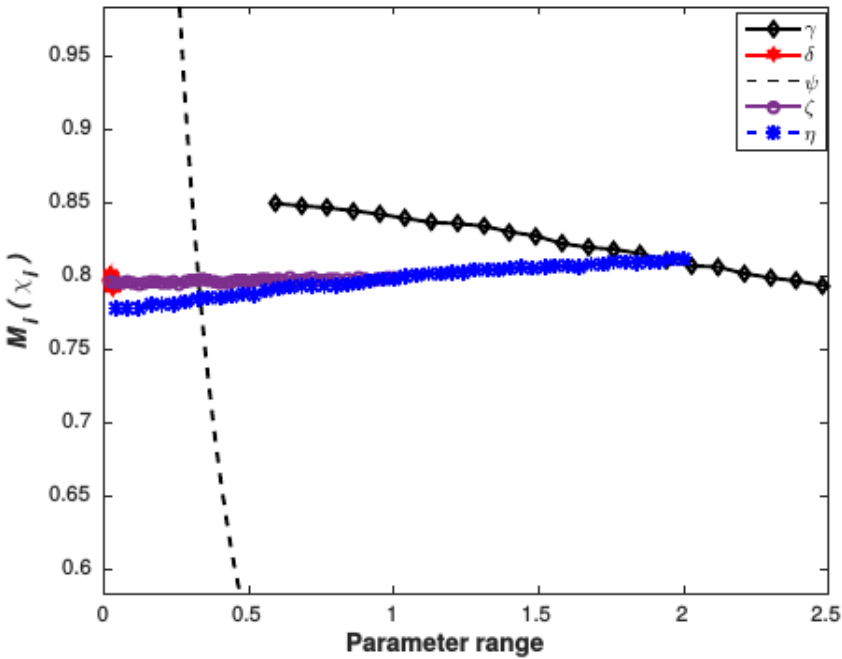
Extended state space:  $\tilde{S} = (\mathbf{k}, \chi) \rightarrow [\underline{\mathbf{k}}, \bar{\mathbf{k}}] \times [\underline{\gamma}, \bar{\gamma}] \times [\underline{\delta}, \bar{\delta}] \times [\underline{\psi}, \bar{\psi}] \times [\underline{\zeta}, \bar{\zeta}] \times [\underline{\eta}, \bar{\eta}]$ .

QoI: aggregate production:  $\mathcal{M}(\chi) = \mathbb{E}[f]$

Univariate effects:

$$\mathcal{M}_i(\chi_i) = \mathbb{E}[\mathcal{M}(\Theta | \Theta_i = \chi_i)]$$

- commonly interpreted as a robust relationship between an input parameter and the QoI.
- **A global solution method required here.**



### III. Wrap-up of the lecture-suite



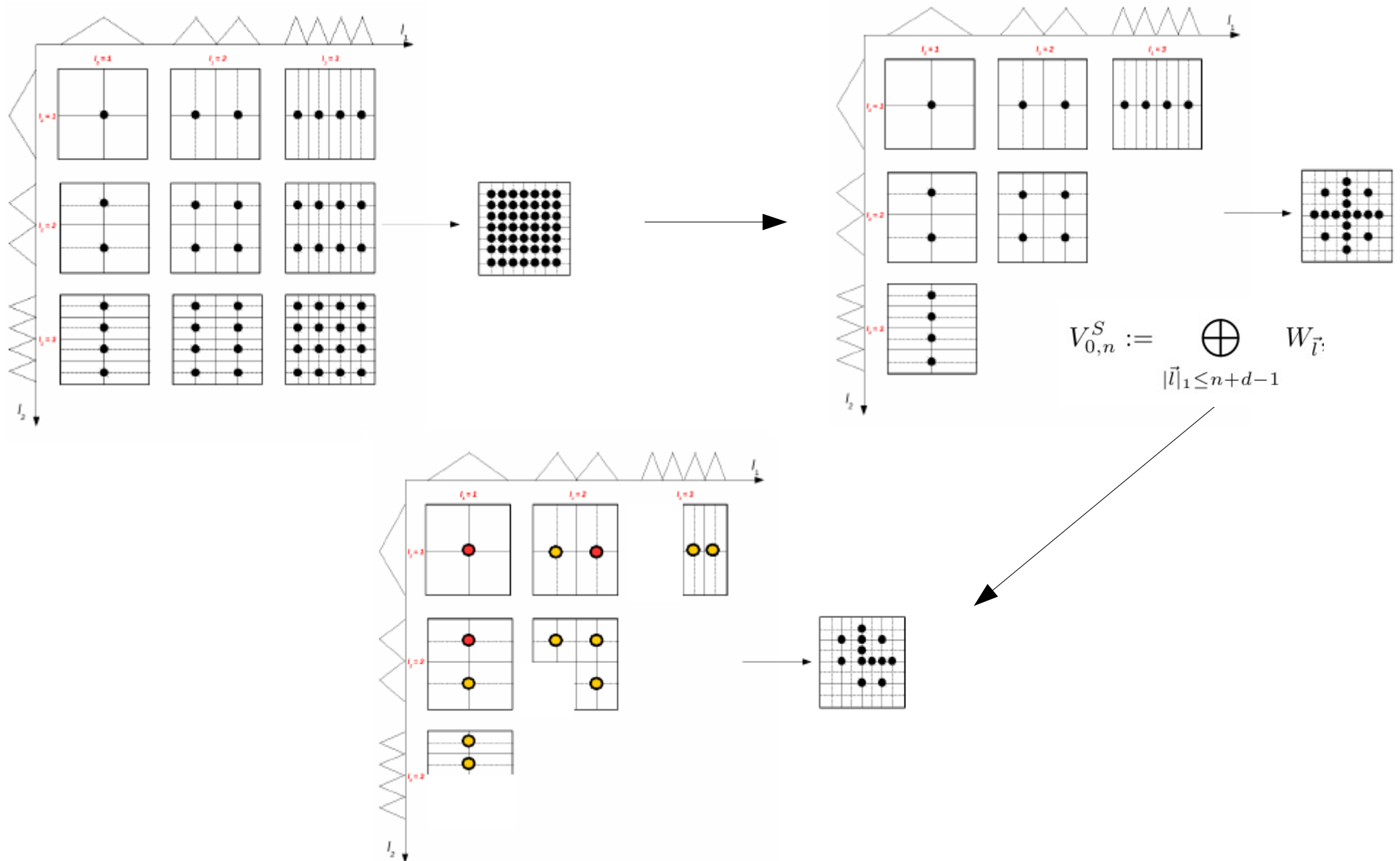
# Lecture 1: A Crash Course in Python

## I. First steps in Python.

Lists, dict, loops, clauses, functions, libraries:  
numpy, scipy, matplotlib,...

## II. Nonlinear equations and optimization.

# Lecture 2: Introduction to Sparse Grids and Adaptive Sparse Grids



# Lecture 3: Dynamic Programming and Time Iteration with Sparse Grids

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

*s.t.*

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

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

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

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

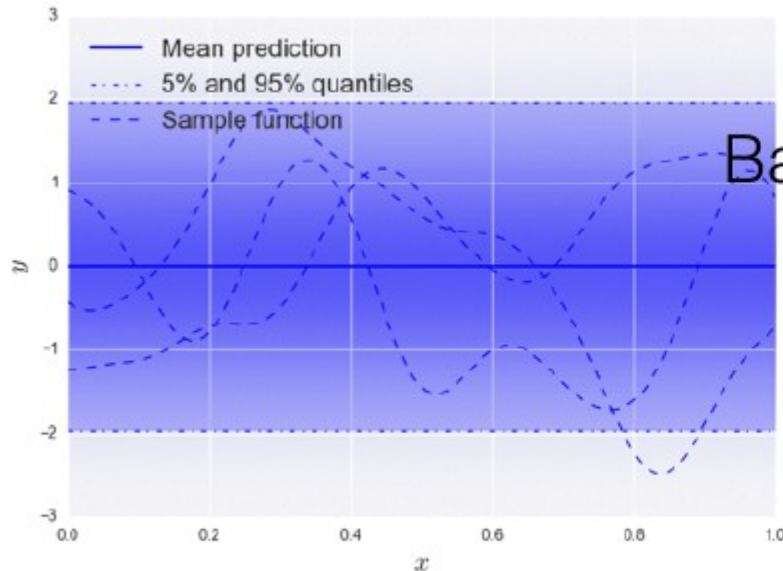
$V_{\text{next}}$  : sparse grid interpolator from the previous iteration step

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

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



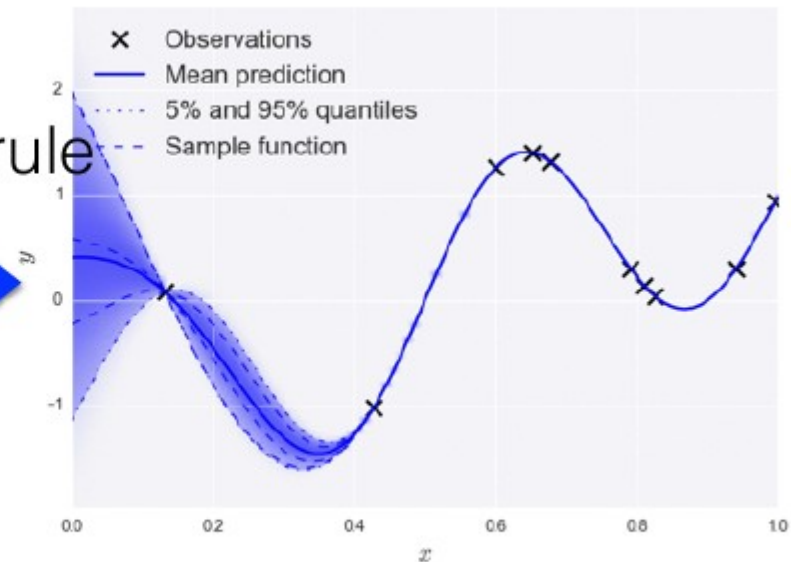
# Lecture 4: Intro to ML and Basics on Gaussian Process Regression



Prior GP



Bayes rule



Posterior GP

Training set:  $D = \{(\mathbf{x}_i, y_i) | i = 1, \dots, n\}$

$$\begin{pmatrix} \mathbf{f} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N} \left( \begin{pmatrix} \boldsymbol{\mu} \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{pmatrix} \right) \quad p(\mathbf{f}_* | \mathbf{X}_*, \mathbf{X}, \mathbf{f}) = \mathcal{N}(\mathbf{f}_* | \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$$

$$\boxed{\boldsymbol{\mu}_* = \boldsymbol{\mu}(\mathbf{X}_*) + \mathbf{K}_*^T \mathbf{K}^{-1} (\mathbf{f} - \boldsymbol{\mu}(\mathbf{X}))}$$

$$\boldsymbol{\Sigma}_* = \mathbf{K}_{**} - \mathbf{K}_*^T \mathbf{K}^{-1} \mathbf{K}_*$$

Test point = interpolation at  $\mathbf{X}^*$

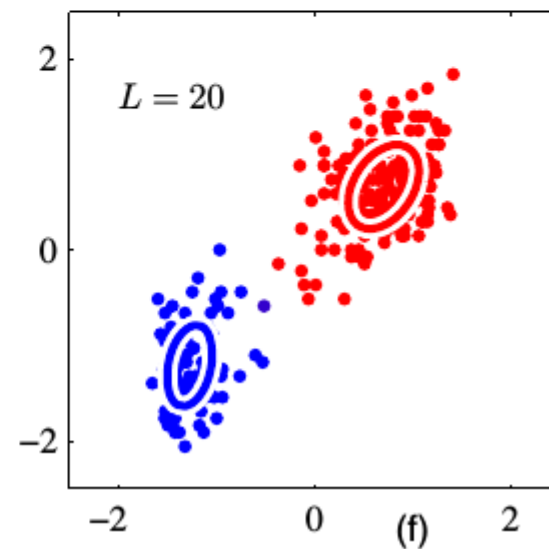
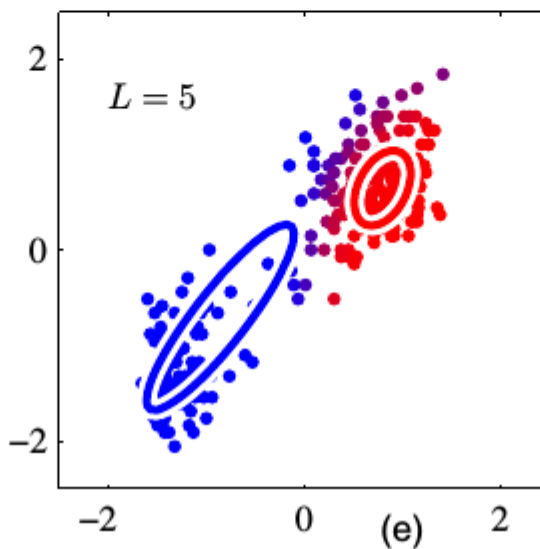
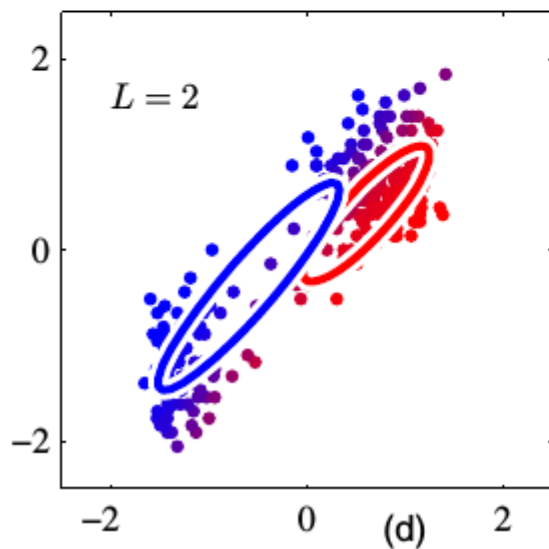
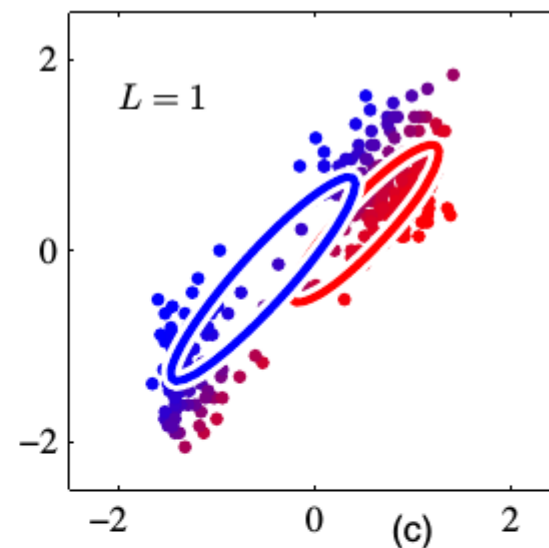
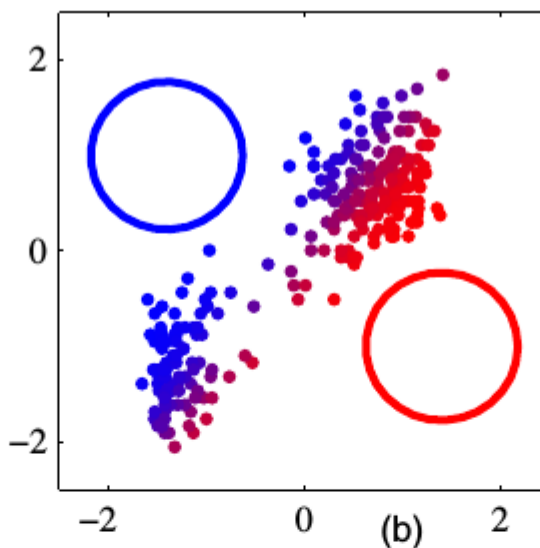
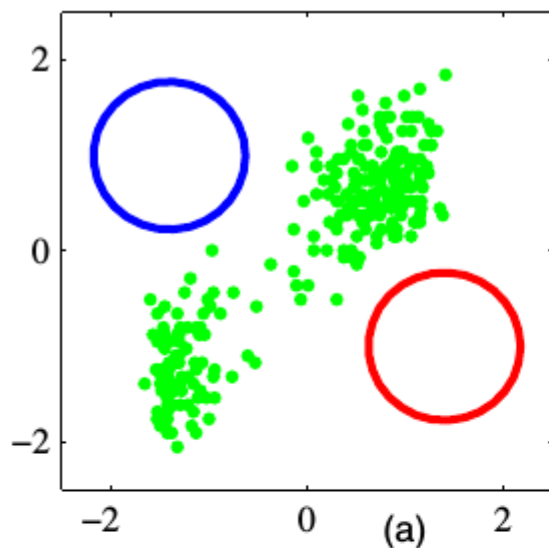
→ **predictive mean**  $\mu_* = \mathbb{E}(f_*)$

→ **Confidence Intervals!**

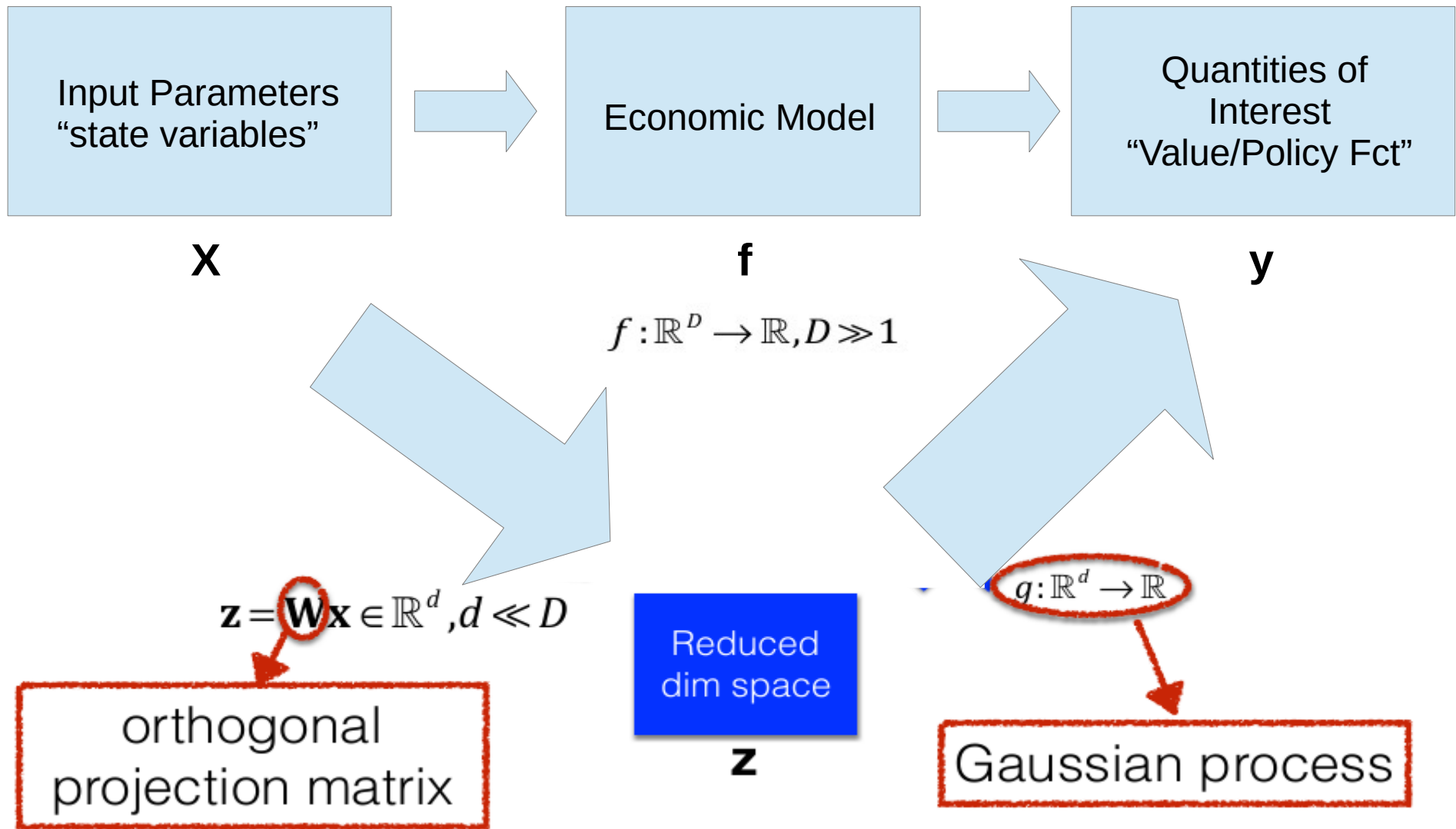
Where we have data, we have high confidence in our predictions.

Where we do not have data, we cannot be too confident about our predictions.

# Lecture 5: GPR and Intro to GMM



# Lecture 6: Introduction to Active Subspaces and DP on irregularly-shaped domains



# Summerschool: OSE Lab at U Chicago

- ♦ **July 1<sup>st</sup> – August 2<sup>nd</sup> .**
- ♦ Summer school on Computational Economics.
- ♦ ~25 Students.
- ♦ Travel, Housing, & Stipend.
- ♦ Ideal for 2<sup>nd</sup> year Ph.D. students.
  - Apply, the portal opens with the next few days!

Last year's site: <https://bfi.uchicago.edu/osm18>

Program 2018: <https://github.com/OpenSourceMacro/BootCamp2018>

