Polynomial chaos expansions part 2: Practical implementation

Jonathan Feinberg and Simen Tennøe

Kalkulo AS

January 23, 2015

Relevant links

A very basic introduction to scientific Python programming:
http://hplgit.github.io/bumpy/doc/pub/sphinx-basics/index.html

Installation instructions:
https://github.com/hplgit/chaospy

Repetition of our model problem

We have a simple differential equation

$$\frac{du(x)}{dx} = -au(x), \qquad u(0) = I$$

with the solution

$$u(x) = Ie^{-ax}$$

with two random input variables:

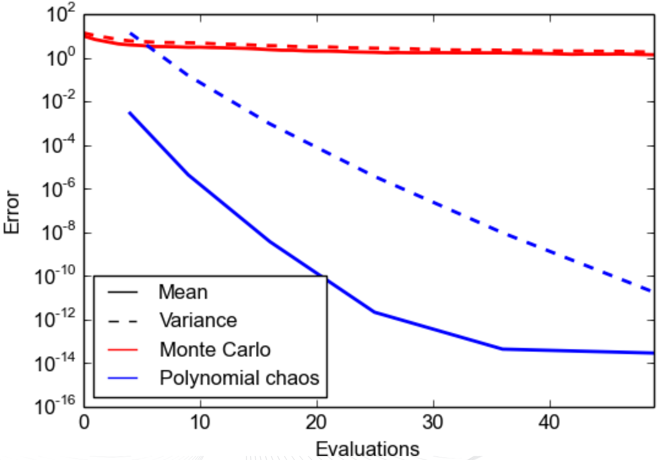
$$a \sim \text{Uniform}(0, 0.1), \qquad I \sim \text{Uniform}(8, 10)$$

Want to compute $\mathrm{E}(u)$ and $\mathrm{Var}(u)$

Repetition of the Chaospy code

```
dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
dist = cp.J(a,I)

P = cp.orth_ttr(2, dist)
```

Polynomial chaos expansions have a very fast convergence rate



The computational essence of polynomial chaos

With $\hat{u}_M(x;q) = \sum_{n=0}^{N} c_n(x) P_n(q)$ and orthogonal polynomials, least squares minimization leads to a formula for c_n:

$$c_n(x) = \frac{\langle u, P_n \rangle_Q}{\|P_n\|_Q^2} = \frac{\mathrm{E}(uP_n)}{\mathrm{E}(P_n^2)}$$

$$= \frac{1}{\mathrm{E}(P_n^2)} \int u(x;q) P_n(q) f_Q(q) dq \approx$$

$$\hat{c}_n(x) = \frac{1}{\mathrm{E}(P_n^2)} \sum_{k=0}^{K} P_n(q_k) u(x;q_k) f(q_k) \omega_k$$

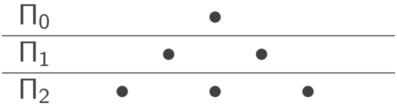
The numerical integral approximation is named *pseudo-spectral method*.

q_k quadrature nodes, ω_k quadrature weights

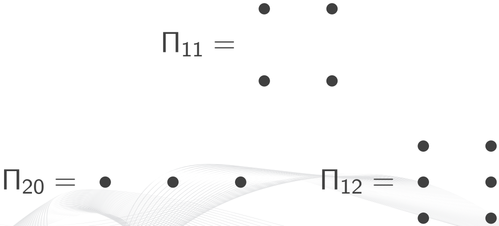
Generating nodes and weights in Chaospy

```
dist = cp.Normal()
nodes, weights = cp.generate_quadrature(2, dist, rule="G")

print nodes
[[-1.73205081  0.          1.73205081]]
print weights
[ 0.16666667  0.66666667  0.16666667]
```

Quadrature rule Π



Multivariate combinations:



K Total number of quadrature nodes

L Quadrature order along an axis

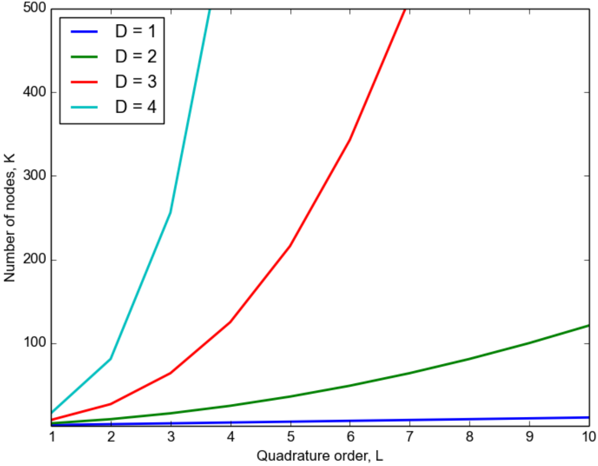
Generating multivariate integration rules in Chaospy

```
# joint multivariate dist
dist = cp.J(cp.Uniform(), cp.Uniform())
nodes, weights = cp.generate_quadrature((1,2), \
    dist, rule="G")

print nodes
[[0.211324 0.211324 0.211324 0.788675 0.788675 0.788675]
 [0.112701 0.5      0.887298 0.112701 0.5      0.887298]]

print weights
 [0.138888 0.222222 0.138889 0.138889 0.222222 0.138889]
```

A full implementation of pseudo-spectral projection in Chaospy

```
dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
dist = cp.J(a,I)

P = cp.orth_ttr(2, dist)

nodes, weights = cp.generate_quadrature(3, dist)

x = np.linspace(0, 10, 100)
samples_u = [u(x, *node) for node in nodes.T]

u_hat = cp.fit_quadrature(P, nodes, weights, samples_u)

mean, var = cp.E(u_hat, dist), cp.Var(u_hat, dist)
```

Number of quadrature nodes K grows exponentially with dimension D



Smolyak sparse grids can drastically reduce the number of nodes

Full tensor basis:

y^2	y^2x	y^2x^2
y	yx	yx^2
1	x	x^2

Smolyak sparse grid:

y^2		
y	xy	
1	x	x^2

$=$

1	x	x^2

$+$

y^2
y
1

$+$

y	yx
1	x

$-$

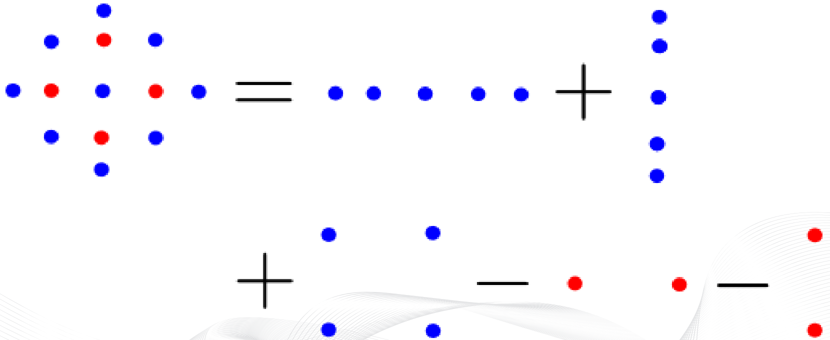
1	x

$-$

y
1

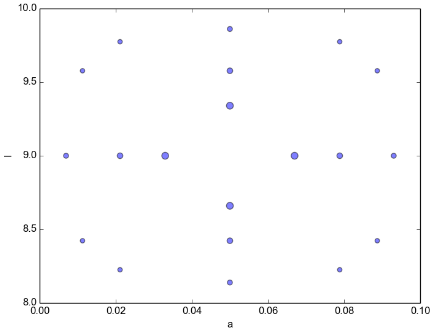
$$\Pi_{20} + \Pi_{11} + \Pi_{02} - \Pi_{10} - \Pi_{01}$$

Example of a Smolyak node placement

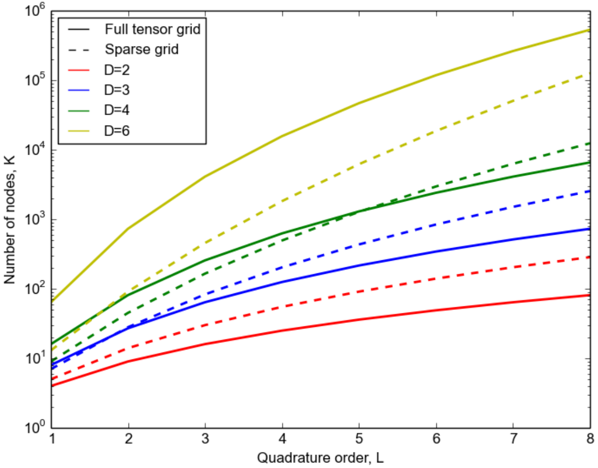


Creating sparse grid nodes in Chaospy

```
nodes, weights =
 cp.generate_quadrature(k, dist, rule="G",
                        sparse=True)
```



For low dimension D, tensor grid is best; for high dimension D, sparse grid is more efficient

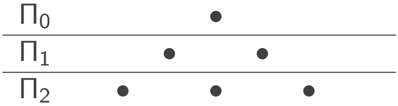


Different problems require different schemes

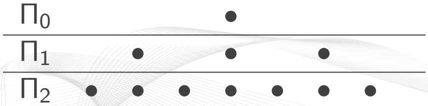
Key		Description
"Gaussian"	"G"	Optimal Gaussian quadrature.
"Legendre"	"E"	Gauss-Legendre quadrature
"Clenshaw"	"C"	Clenshaw-Curtis quadrature.
"Leja"	"J"	Leja quadrature.
"Genz"	"Z"	Hermite Genz-Keizter 16 rule.
"Patterson"	"P"	Gauss-Patterson quadrature rule.

Nested sparse grids use overlapping nodes to further reduce the number of nodes

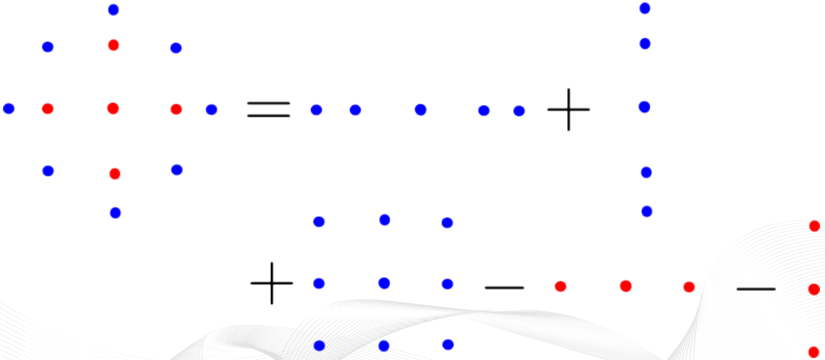
Clenshaw-Curtis:



Nested Clenshaw-Curtis:



Nested smolyak sparse grid in practice



The number of overlapping nodes grows quickly

1

1 3 1

1 3 5 3 1

1 3 1

1

Mapping between polynomial order M and quadrature order L

For nested Clenshaw-Curtis

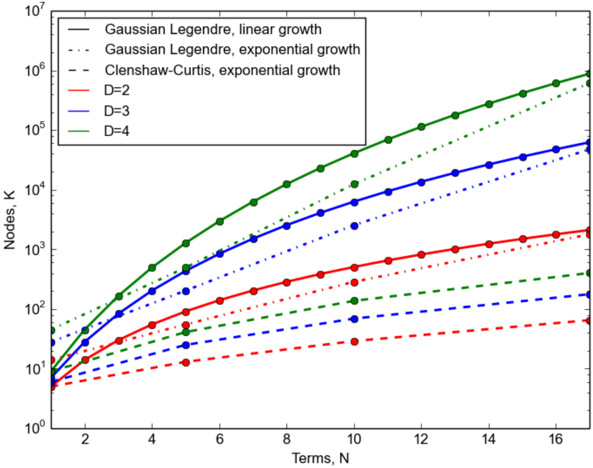
Quadrature order, L	0	1	2	3	4	5	6	7	8	Quadrature order,
Number of nodes, K	1	4	9	16	25	36	49	64	81	Number of nodes,
Polynomial terms, N	1	3	6	10	15	21	28	37	47	Polynomial terms,
Polynomial order, M	0	1	2	3	4	5	6	7	8	Polynomial order,

Suggestion:

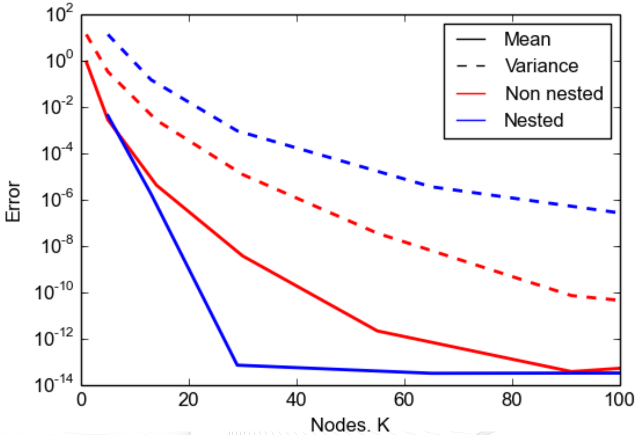
Linear growth rule: $L = 2M - 1$

Exponential growth rule: $L = 2^M - 1$

Comparing three sparse grids



Nested sparse grid converges faster than a non nested sparse grid



Gaussian qudrature approximates integrals with weighting functions

$$\int W(q)u(x,q)dq \approx \sum_k \omega_k u(x,q_k)$$

We need weighting function $W(q)$ to be the joint probability distribution $f_Q(q)$

$$\int f_Q(q)u(x,q)dq \approx \sum_k \omega_k u(x,q_k)$$

The point collocation method is alternative to the pseudo-spectral method

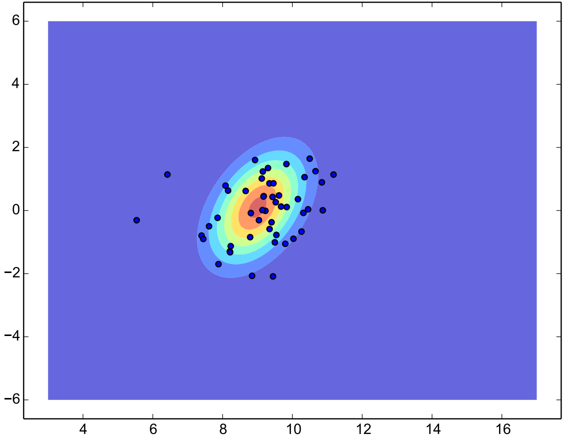
1. Psuedo-spectral method:
   1.1 Determine polynomial approximation of model by least squares minimization in a space weighted with the probability distribution
   1.2 Approximate integrals in c_n by quadrature rules
2. Point collocation method:
   2.1 Determine polynomial approximation of model by least squares minimization in a vector space as in regression (or overdetermined matrix systems)
   2.2 Need to choose a set of nodes (regression points)

The point collocation method: estimate c_n using linear regression

$$\mathbf{c} = \begin{bmatrix} c_0(x) \\ \vdots \\ c_N(x) \end{bmatrix} \quad \mathbf{P} = \begin{bmatrix} P_0(q_0) & \cdots & P_N(q_0) \\ \vdots & & \vdots \\ P_0(q_K) & \cdots & P_N(q_K) \end{bmatrix} \quad \mathbf{u} = \begin{bmatrix} u(x; q_0) \\ \vdots \\ u(x, q_K) \end{bmatrix}$$

$$\begin{aligned} \hat{\mathbf{c}} &= \underset{\mathbf{c}}{\operatorname{argmin}} \|\mathbf{P}\mathbf{c} - \mathbf{u}\|_2^2 \\ &= (\mathbf{P}^T\mathbf{P})^{-1}\mathbf{P}^T\mathbf{u} \end{aligned}$$

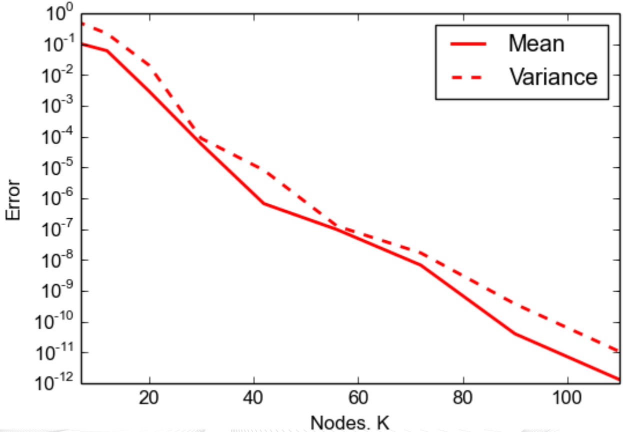
Collocation nodes should be placed where probability is high



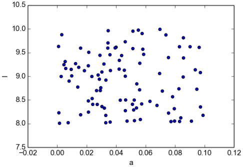
Code for least square minimization

```
def u(x, a, I):
  return I*np.exp(-a*x)

dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
dist = cp.J(dist_a, dist_I)

x = np.linspace(0, 10, 100)

P = cp.orth_ttr(3, dist)
nodes = dist.sample(2*len(P))
samples_u = [u(x, *node) for node in nodes.T]
u_hat = cp.fit_regression(P, nodes, samples_u)
```

Convergence using least square minimization

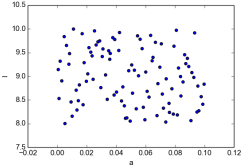


(Pseudo-)Random sampling schemes for choosing nodes



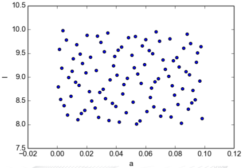
(Pseudo-)Random sampling:

```
nodes = dist.sample(100)
```



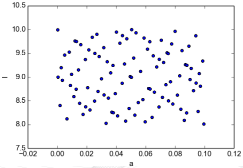
Latin Hypercube sampling:

```
nodes = dist.sample(100, "L")
```



Halton sampling

```
nodes = dist.sample(100, "H")
```



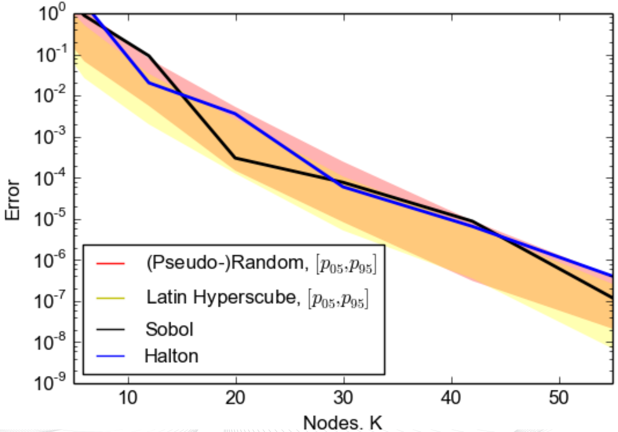
Sobol sampling

```
nodes = dist.sample(100, "S")
```

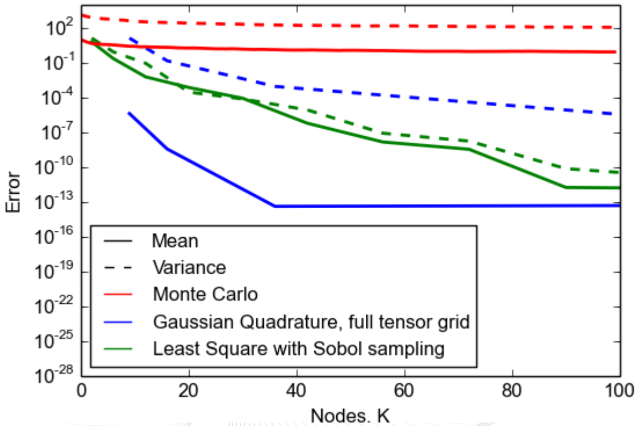
Sampling schemes in Chaospy

Key	Name	Nested
K	Korobov	no
R	(Pseudo-)Random	no
L	Latin hypercube	no
S	Sobol	yes
H	Halton	yes
M	Hammersley	yes
C	Clenshaw Curtis	no
G	Gaussian quadrature	no
E	Gauss-Legendre	no

Convergence using different sampling schemes



What is best of pseudo-spectral and point collocation method? It's problem dependent!



Which method to choose for your problem

	Pseudo-spectral	Point collocation	Monte Carlo
Efficiency	Highest	Very high	Very low
Stability	Low	Medium	Very high
Dimension-independence	Lowest	Low	Highest

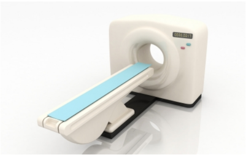
A surrogate model allows for computational cheap statistical analysis

```
u_hat, c_hat = cp.fit_quadrature(
        P, nodes, weights, solves, retall=True)

mean =  cp.E(u_hat, dist)
var = cp.Var(u_hat, dist)
```

```
mean = c_hat[0]
norms2 = cp.E(P**2, dist)[1:]
c2 = c_hat[1:]**2
var = np.sum(c2*norms2)
```

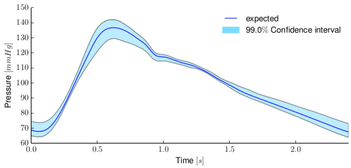
```
samples_q = dist.sample(10**6)
samples_u = u_hat(*samples_q)
mean = np.mean(samples_u,1)
var = np.var(samples_u,1)
```

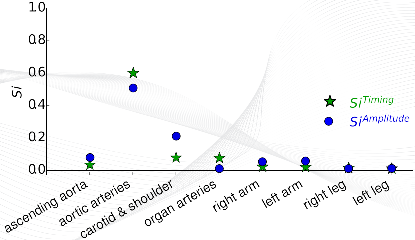
Modeling bloodflow requires sensitivity analysis



STARFiSh
STochastic ARterial Flow Simulations

ChaosPy





Want to have a sensitivity measure to judge the impact of various input parameters

Variance based sensitivity:

$$S_{T_i} = \frac{\mathrm{E}(\mathrm{Var}(u \mid \mathbf{Q} \setminus Q_i))}{\mathrm{Var}(u)} = 1 - \frac{\mathrm{Var}(\mathrm{E}(u \mid \mathbf{Q} \setminus Q_i))}{\mathrm{Var}(u)}$$

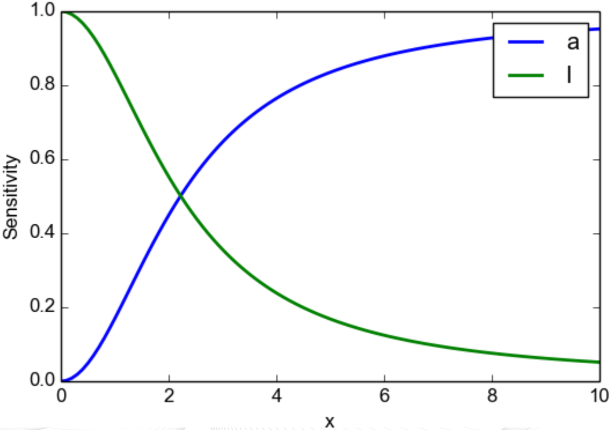
Chaospy:

```
sensitivity_Q = cp.Sens_t(u_hat, dist)
```

Manual code:

```
V = cp.Var(u_hat, dist)
sensetivity_a = 1-cp.Var(cp.E_cond(u_hat, [0,1], dist), dist)/V
sensetivity_I = 1-cp.Var(cp.E_cond(u_hat, [1,0], dist), dist)/V
```

Variance based sensitivity of our example

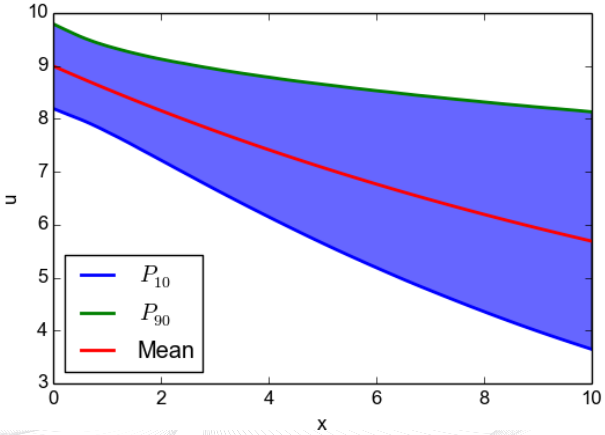


Various statistical metrics are easy to construct in Chaospy

Some statistical metrics have analytical formulas, others can easily be implemented by using Monte Carlo on the surrogate model:

```
samples_Q = dist.samples(10**5)
samples_u = P(*samples_Q)

p_10 = np.percentile(samples_u, 10, axis=0)
p_90 = np.percentile(samples_u, 90, axis=0)
```

Confidence interval



Summary

```
x = np.linspace(0, 10, 100)
def u(x, a, I):
  return I*np.exp(-a*x)

dist_a = cp.Uniform(0, 0.1)
dist_I = cp.Uniform(8, 10)
dist = cp.J(dist_a, dist_I)

P = cp.orth_ttr(3, dist)

nodes, weights = cp.generate_quadrature(4, dist)

samples_u = [u(x, *node) for node in nodes.T]

u_hat= cp.fit_quadrature(P, nodes, weights, samples_u)

mean = cp.E(u_hat, dist)
var = cp.Var(u_hat, dist)
```

Thank you

A very basic introduction to scientific Python programming:
http://hplgit.github.io/bumpy/doc/pub/sphinx-basics/index.html

Installation instructions:
https://github.com/hplgit/chaospy