# CEE 598: Uncertainty Quantification

## Assignment Six

**Noah Garfinkle (garfink2@illinois edu)**

**03 May 2020**

## Imports and Set Up

In [1]:
```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import seaborn as sns
import pytest
import chaospy as cp
import pandas as pd
from scipy.integrate import odeint
```

## References

In [2]:
```
# https://chaospy.readthedocs.io/en/master/tutorial.html
# https://chaospy.readthedocs.io/en/master/_modules/chaospy/quadrature/clenshaw_cur
tis.html
# https://chaospy.readthedocs.io/en/master/quadrature.html#module-chaospy.quadratur
e.clenshaw_curtis
# https://chaospy.readthedocs.io/en/master/quadrature.html#module-chaospy.quadratur
e.sparse_grid
# https://chaospy.readthedocs.io/en/master/recipes/galerkin.html
```

## General Approach

Building off of assignment five, I have opted to again utilize the third path of learning to utilize and confirming an existing open source library. As with Assignment Five, I have opted to utilize Chaospy for Python.

In order to evaluate the integral, I have opted to utilize an approach similar to Monte Carlo approximations of the integral, represented by

$$\int f(X) \approx \sum_{n=1}^{N} W_n f(X_n)$$

Where $N$ is the number of samples, $X_n$ is each individual sample, $f(X_n)$ is the evaluation of the target function at the sample coordinates, and $W_n$ is the weight of the sample as defined by the rule utilized.

## Problem 1.

Consider the following target function

$$f(x) = f(x_1, x_2, \ldots, x_5) = \prod_{i=1}^{5} sin(ix_i)$$

where $x_i \in [0, 1], \forall_i$. We would like to approximate $\int_{[0,1]^5} f(x)dx$ using quadrature. Compare the following multidimensional quadrature rules:

As with assignment five, because I am using an existing library I have emphasized 1) learning to use the library correctly, and 2) developing helper functions in order to aid me in applying the algorithms to this and future problems.

In [23]:
```
# The function f(x) specified in the problem formulation
def func(x):
    x1,x2,x3,x4,x5 = x
    value = np.sin(x1) * np.sin(2*x2) * np.sin(3*x3) * np.sin(4*x4) * np.sin(5*x5)
    return value
```

I have assumed that $x_i$ are each uniformally distributed.

In [4]:
```
distribution_x = cp.J(cp.Uniform(0,1),cp.Uniform(0,1),cp.Uniform(0,1),cp.Uniform(0,
1),cp.Uniform(0,1))
```

In [25]:
```
"""
Utilizes Monte Carlo approximation of an integral, with the sample points determine
d per appropriate rules.
Please note that I have chosen, for simplicity sake, to hard code the distribution
and function.

Inputs:
order (Int) -> The level of approximation to utilize
sampleRule (Str) -> The quadrature rule to be utilized.  For this assignment, clens
haw_curtis and gauss_legendre are utilized
sparse (Bool) -> Whether Chaospy should apply a Sparse Grid
growth (Bool) -> An additional Chaospy parameter which is only exposed for transpar
ency, as it defaults to True in Chaospy if Sparse==True

Returns:
integral (Float) -> Estimate of the function Integral
"""
def calculateIntegral(order,sampleRule="clenshaw_curtis",sparse=False,growth=Fals
e):
    abscissas, weights = cp.generate_quadrature(order, distribution_x, rule=sampleR
ule,sparse=sparse,growth=growth)
    samples = abscissas.T
    integral = 0.0
    for i,sample in enumerate(samples):
        weight = weights[i]
        integral += weight * func(sample)
    return integral
```

In [27]:
```
"""
Simply a wrapper to call the previous function at multiple levels and neatly compil
e the results into
a single dataframe for analysis and plotting
"""
def evaluateIntegralAtMultipleLevels(sampleRule="clenshaw_curtis",sparse=False,grow
th=False,levels=[1,2,3]):
    orders = []
    integrals = []
    rules = []
    sparses = []
    for order in levels:
        orders.append(order)
        integrals.append(calculateIntegral(order,sampleRule=sampleRule,sparse=spars
e,growth=growth))
        rules.append(sampleRule)
        sparses.append(sparse)
    df = pd.DataFrame({"Order":orders,"Integral":integrals,"Rule":rules,"Sparse":sp
arses})
    return df
```

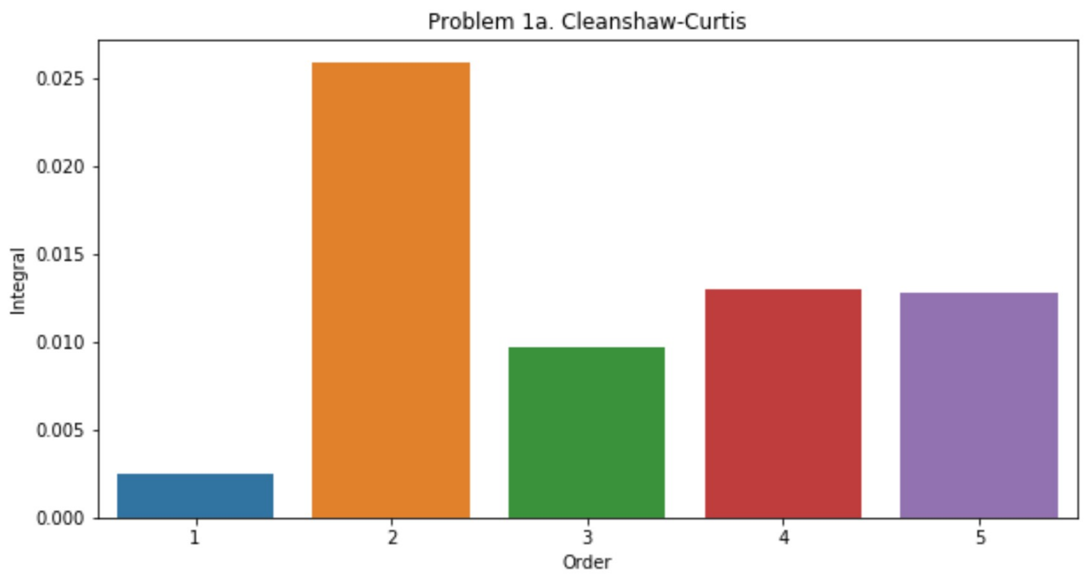
## Problem 1a. Tensor product using Clenshaw-Curtis (CC) points at levels 1,2, and 3.

Note: Clenshaw-Curtis points and their weights are given on Slide 15 of Lecture 20

While the problem prompt only requires the first three levels to be analyzed, I have chosen to present the first five because it makes the comparison of later results (which call for five levels) more consistent.

In [7]:
```
df_1a = evaluateIntegralAtMultipleLevels(levels=[1,2,3,4,5])
fig,ax = plt.subplots(figsize=(10,5))
sns.barplot(x="Order",y="Integral",data=df_1a,ax=ax)
ax.set_title("Problem 1a. Cleanshaw-Curtis")
```

Out[7]:
```
Text(0.5, 1.0, 'Problem 1a. Cleanshaw-Curtis')
```

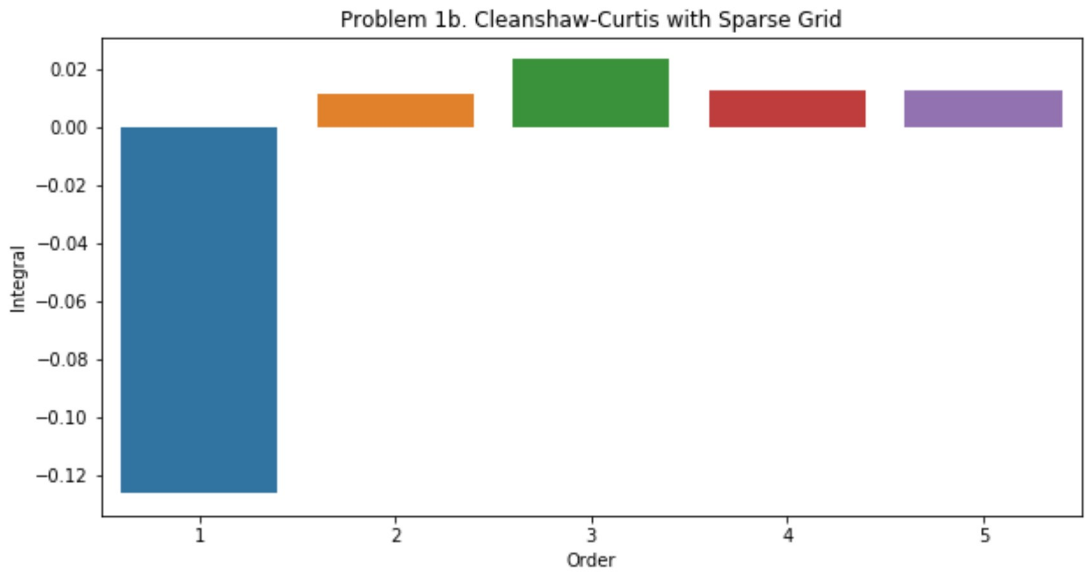


## Problem 1b. Sparse grid using CC points at levels 1, 2, and 3.

```
In [8]: df_1b = evaluateIntegralAtMultipleLevels(sparse=True,growth=True,levels=[1,2,3,4,
        5])
        fig,ax = plt.subplots(figsize=(10,5))
        sns.barplot(x="Order",y="Integral",data=df_1b,ax=ax)
        ax.set_title("Problem 1b. Cleanshaw-Curtis with Sparse Grid")
```

```
C:\Users\garfink2\AppData\Local\Continuum\anaconda3\envs\phd\lib\site-packages\n
umpoly\poly_function\monomial\cross_truncation.py:49: RuntimeWarning: divide by
zero encountered in true_divide
  out = numpy.sum((indices/bound)**norm, axis=-1)**(1./norm) <= 1
C:\Users\garfink2\AppData\Local\Continuum\anaconda3\envs\phd\lib\site-packages\n
umpoly\poly_function\monomial\cross_truncation.py:49: RuntimeWarning: invalid va
lue encountered in true_divide
  out = numpy.sum((indices/bound)**norm, axis=-1)**(1./norm) <= 1
C:\Users\garfink2\AppData\Local\Continuum\anaconda3\envs\phd\lib\site-packages\n
umpoly\poly_function\monomial\cross_truncation.py:49: RuntimeWarning: invalid va
lue encountered in less_equal
  out = numpy.sum((indices/bound)**norm, axis=-1)**(1./norm) <= 1
```

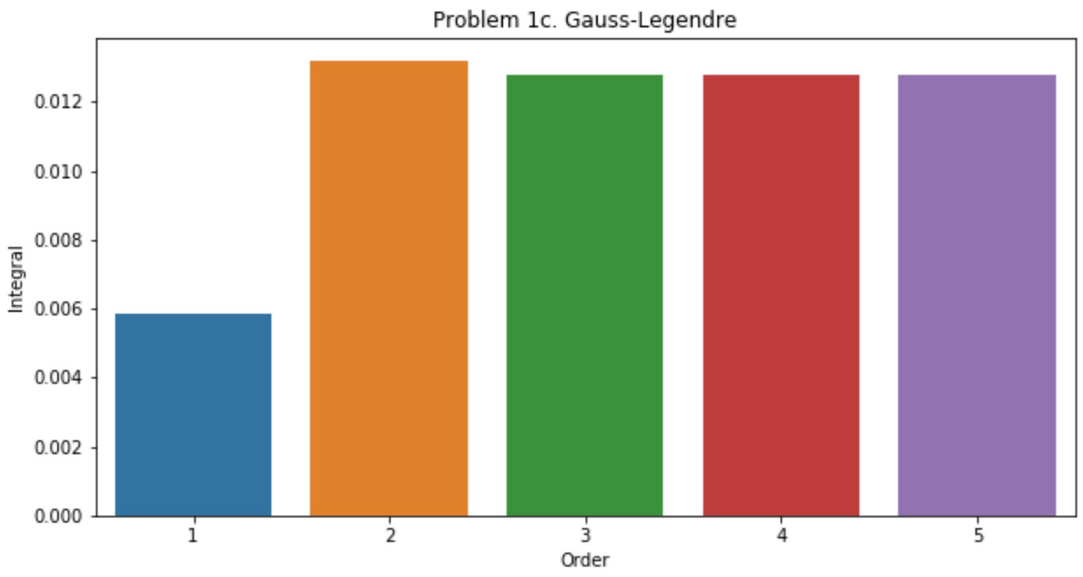
```
Out[8]: Text(0.5, 1.0, 'Problem 1b. Cleanshaw-Curtis with Sparse Grid')
```



## Problem 1c. Sparse grid using Gauss-Legendre quadrature points at sparse grid levels 1,2,3, and 4. Note that the level for one dimensional GL quadrature, compared to 1D CC quadrature, involves different numbers of points. For example, level-2 CC has 3 points, but level-2 GL has 2 points. So, it is expected that at the same sparse grid level, the number of multidimensional points are not the same between CC and GL.

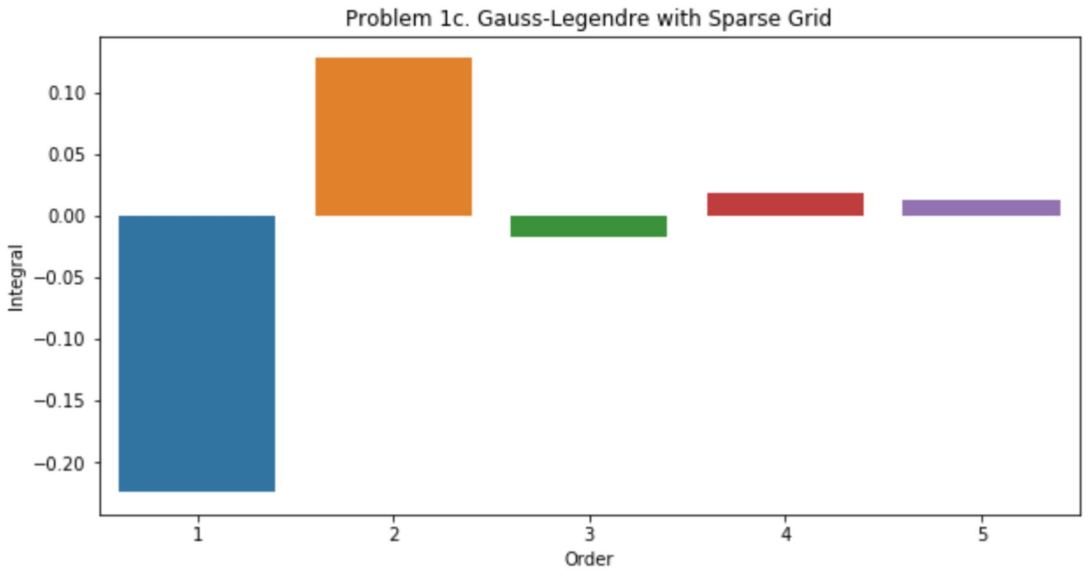
```
In [9]: df_1c = evaluateIntegralAtMultipleLevels(sampleRule="gauss_legendre",sparse=False,g
        rowth=False,levels=[1,2,3,4,5])
        fig,ax = plt.subplots(figsize=(10,5))
        sns.barplot(x="Order",y="Integral",data=df_1c,ax=ax)
        ax.set_title("Problem 1c. Gauss-Legendre")

Out[9]: Text(0.5, 1.0, 'Problem 1c. Gauss-Legendre')
```



```
In [10]: df_1c2 = evaluateIntegralAtMultipleLevels(sampleRule="gauss_legendre",sparse=True,g
         rowth=True,levels=[1,2,3,4,5])
         fig,ax = plt.subplots(figsize=(10,5))
         sns.barplot(x="Order",y="Integral",data=df_1c2,ax=ax)
         ax.set_title("Problem 1c. Gauss-Legendre with Sparse Grid")

Out[10]: Text(0.5, 1.0, 'Problem 1c. Gauss-Legendre with Sparse Grid')
```



In order to more fairly compare the four approaches presented above, I have run each approach out to five levels, which are combined and compared below.

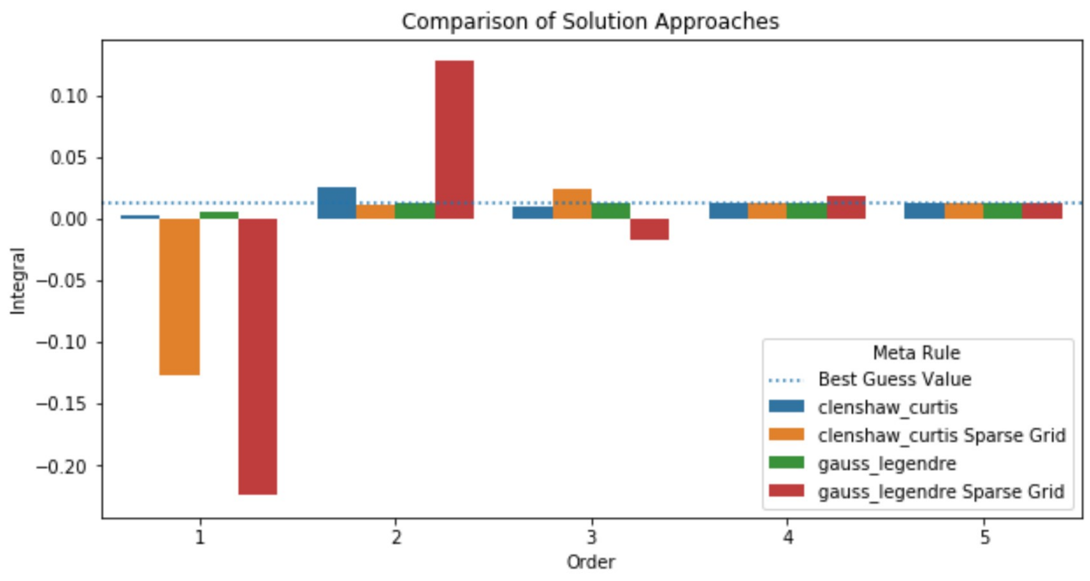
In [11]:
```
df_stacked = pd.concat([df_1a,df_1b,df_1c,df_1c2])
metaRule = []
for i,row in df_stacked.iterrows():
    sparse = ""
    if row["Sparse"] == True:
        sparse = "Sparse Grid"
    rule = f"{row['Rule']} {sparse}"
    metaRule.append(rule)
df_stacked["Meta Rule"] = metaRule
```

In [12]:
```
# Run a high order problem to try to fit a better estimate
bestGuessIntegral = calculateIntegral(10,sampleRule="clenshaw_curtis",sparse=False,
growth=False)
print(bestGuessIntegral)
```

```
0.012788234790550164
```

In [13]:
```
fig,ax = plt.subplots(figsize=(10,5))
ax.axhline(bestGuessIntegral,ls=":",label="Best Guess Value")
sns.barplot(x="Order",y="Integral",hue="Meta Rule",data=df_stacked,ax=ax)
ax.set_title("Comparison of Solution Approaches")
```

Out[13]:
```
Text(0.5, 1.0, 'Comparison of Solution Approaches')
```



The Gauss-Legendre rule appears to perform best, which I believe is reasonable as it is known to be a good solution approach for uniform distributions as we have with this problem, while the Clenshaw-Curtis rule is more general but often faster.

In spite of my expectations, sparse grids as implemented do not improve my solution until allowing a greater order, suggesting that I have not implemented them correctly.

## Problem 2.

Using the Stochastic Galerkin approach, find a PCE approximation for $y$ given by

$$ay = bc + d$$

where

$$a = 3 - 0.4\Psi_1(z) + 0.02\Psi_2(z), b = -5 + 0.4\Psi_1(z) + 0.4\Psi_2(z), c = 3 + 0.02\Psi_1(z), and\ d = 11 - 3\Psi_1(z)$$

are four random parameters, where $z$ is a uniform random variable in $[0, 1]$ and $\Psi_i$ are Legendre polynomials.

```
In [22]: distribution_z = cp.J(cp.Uniform(0,1))
```

### General Approach to Problem Two

Per the course notes, we attempt to approximate

$$y(z) = \sum_{i=0}^{K} c_i \Psi_i(z)$$

Where $\Psi_i(z)$ represents our basis function.

1. We begin by creating the closed form of our residual (error) function

$$\delta(y, z) = a(z)y(z) - b(z)c(z) - d(z)$$

Which can be expanded to

$$\delta(y, z) = (3 - 0.4\Psi_1(z) + 0.02\Psi_2(z))y(z) - (-5 + 0.4\Psi_1(z) + 0.4\Psi_2(z))(3 + 0.02\Psi_1(z)) - (11 - 3\Psi_1(z))$$

and can be further exapnded to

$$\delta(y, z) = 3y(z) - 0.4\Psi_1(z)y(z) + 0.02\Psi_2(z)y(z) - 0.008\Psi_1(z)\Psi_2(z) - 0.008\Psi_1(z)^2 + 1.9\Psi_1(z) - 1.2\Psi_2(z) + 4$$

We then substitute our approximation of $y(z)$ into this expansion, resulting in

$$\delta(y, z) = 3\left(\sum_{i=0}^{K} c_i \Psi_i(z)\right) - 0.4\Psi_1(z)\left(\sum_{i=0}^{K} c_i \Psi_i(z)\right) + 0.02\Psi_2(z)\left(\sum_{i=0}^{K} c_i \Psi_i(z)\right) - 0.008\Psi_1(z)\Psi_2(z) - 0.008\Psi$$
$$+ 4$$

Which is now a function of strictly z.

Per page 216 of "Uncertainty Quantification," our objective is to formulate an approximate solution $u^K(z, \Psi)$ which satisfies

$$0 = \langle \delta(y, z), \Psi_i \rangle$$

$$\therefore 0 = \langle 3\left(\sum_{i=0}^{K} c_i \Psi_i(z)\right) - 0.4\Psi_1(z)\left(\sum_{i=0}^{K} c_i \Psi_i(z)\right) + 0.02\Psi_2(z)\left(\sum_{i=0}^{K} c_i \Psi_i(z)\right) - 0.008\Psi_1(z)\Psi_2(z) - 0.008\Psi_1($$
$$+ 4, \Psi_i(z)\rangle, for\ i = 0, \ldots, K$$

Per class discussion, this inner product is used to generate a system of $K + 1$ equations in order to solve for $K + 1$ unknowns, which results in our solution vector $c$. Note that $K$ is the order of our response function.

For $K = 0$,

$$0 = \langle 0.008\Psi_1(z)\Psi_2(z) - 0.008\Psi_1(z)^2 + 1.9\Psi_1(z) - 1.2\Psi_2(z) + 4, \Psi_i(z)\rangle$$

For $K = 1$,

$$0 = \langle 3\left(c_1\Psi_1(z)\right) - 0.4\Psi_1(z)\left(c_1\Psi_1(z)\right) + 0.02\Psi_2(z)\left(c_1\Psi_1(z)\right) - 0.008\Psi_1(z)\Psi_2(z) - 0.008\Psi_1(z)^2 + 1.9\Psi_1(z) -$$

For $K = 2$,

$$0 = \langle 3\left(c_1\Psi_1(z) + c_2\Psi_2(z)\right) - 0.4\Psi_1(z)\left(c_1\Psi_1(z) + c_2\Psi_2(z)\right) + 0.02\Psi_2(z)\left(c_1\Psi_1(z) + c_2\Psi_2(z)\right) - 0.008\Psi_1(z)\Psi_2($$
$$- 1.2\Psi_2(z) + 4, \Psi_i(z)\rangle$$

For the sake of time I have opted to submit my initial approach to problem two, which I understand to be incomplete. To be honest, I am uncertain about how to proceed from here. First, because only two $\Psi$ are specified, I have opted to terminate at $K = 2$, although I am uncertain if this choice is defensible. Second, I am not confident about formulating the inner products into the typically required form $Ax = b$.

My understanding in order to complete the problem is that I must evaluate the inner product for each level $K$, resulting in a system of equations which can then be evaluated in order to solve for the vector $c$.