Numerical Integration

by Richard W. Evans (https://sites.google.com/site/rickecon/), January 2020

The code in this Jupyter notebook was written using Python 3.7. All references cited in this notebook are fully cited in Section 7 at the end of this notebook.

1. Introduction

Integrals of the form $\int_a^b g(x)dx$ arise often in economic models. One example is the aggregating of consumption amounts of a continuum of differentiated goods $c_t(i)$,

$$C_t = \left(\int_0^1 \alpha_i^{\frac{1}{\varepsilon}} c_t(i)^{\frac{\varepsilon-1}{\varepsilon}} di\right)^{\frac{\varepsilon}{\varepsilon-1}}$$

where C_t is aggregate consumption, α_i is a weight on the particular amount of consumption of good i, ε is the constant elasticity of substitution between different goods i, and the measure of goods is normalized to be between 0 and 1, without loss of generality. This consumption aggregator is often called the Armington aggregator as it was first proposed in Armington (1969). It is also known as a Dixit-Stiglitz aggregator after its use in Dixit and Stiglitz (1977). Another key example of an integral that often occurs in macroeconomics is the expectations operator on the right-hand-side of the standard intertermporal Euler equation,

$$u'(c_t) = \beta E_{z_{t+1}|z_t}\left[(1 + r_{t+1} - \delta)u'(c_{t+1})\right]$$

$$\Rightarrow \quad u'(c_t) = \beta \int_a^b \Big(1 + r_{t+1}(z_{t+1}) - \delta\Big)u'\Big(c_{t+1}(z_{t+1})\Big) f(z_{t+1}|z_t)dz_{t+1}$$

where a and b are the bounds of the support of z_{t+1} and $f(z_{t+1}|z_t)$ is the pdf of z_{t+1} that could potentially be conditional on z_t.

It is a rare convenience when these integrals can be evaluated analytically. However, it does not take much richness in functional form to render analytical solutions impossible for many integrals in economic models. In these cases, the integral must be computed numerically. The following discussion of numerical integration draws from the great treatments of the subject in Heer and Maussner (2009, pp. 598-603), Judd (1998, ch. 7), and Adda and Cooper (2003, pp. 55-60).

2. Newton-Cotes Quadrature

Newton-Cotes quadrature forumalas approximate the integral of a function $\int_a^b g(x)dx$ by evaluating the function at N equally spaced nodes $\{x_1, x_2, \ldots x_N\}$ and weighting those nodes with N weights $\{\omega_1, \omega_2, \ldots \omega_N\}$. The general form of Newton-Cotes quadrature forumulas is

$$\int_a^b g(x)dx \approx \sum_{n=1}^{N} \omega_n g(x_n)$$

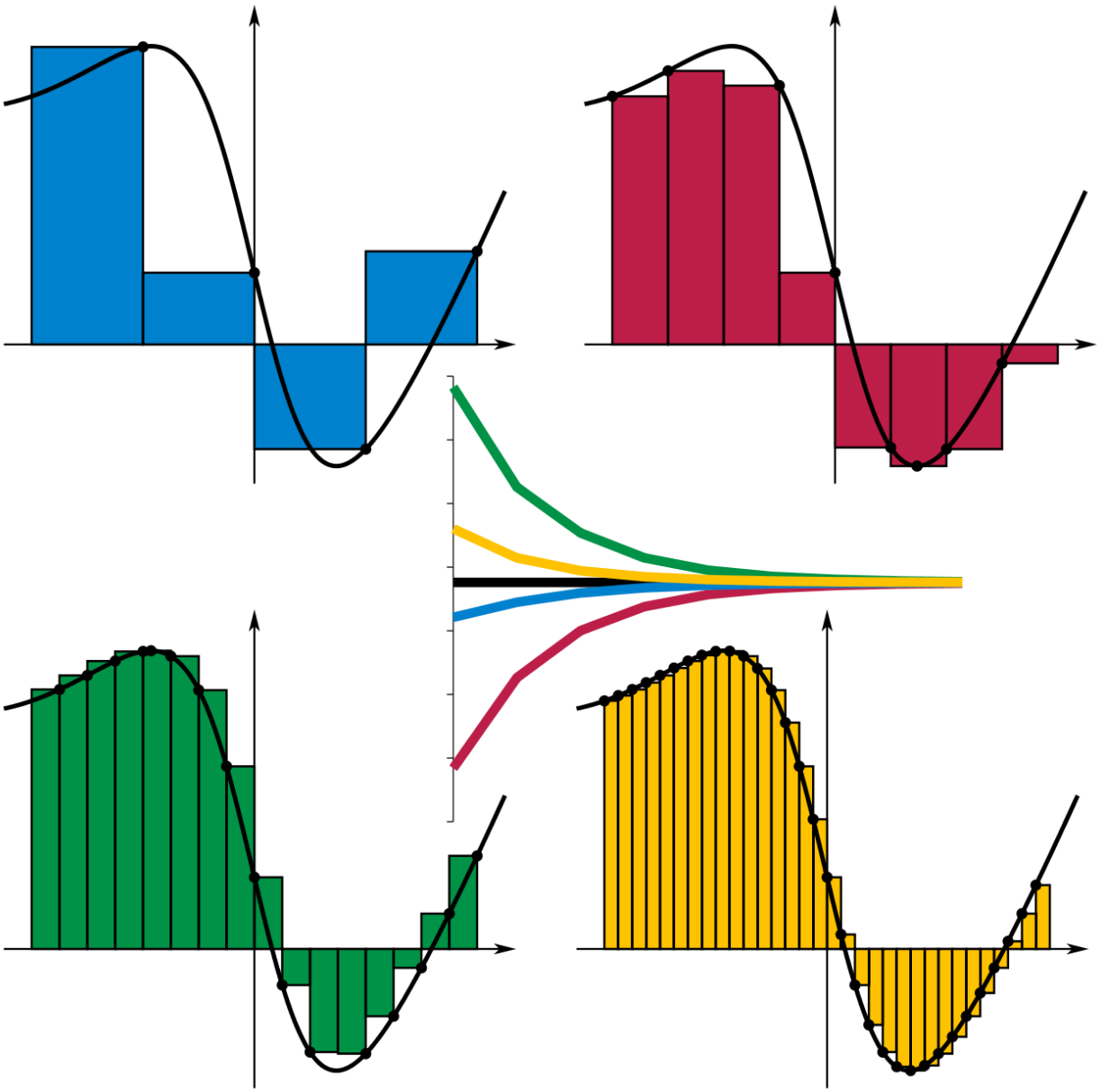
2.1. Midpoint rule (1 node)

The midpoint rule is the simplest Newton-Cotes formula and uses only one node or evaluation of the function. It is simply a Riemann sum approximation over the domain of the function $g(x)$. The midpoint formula simply evaluates the function at the midpoint of the domain of $x = \frac{a+b}{2}$ and assumes that the function is a constant at that level over the entire domain of $x \in [a, b]$.

$$\int_a^b g(x)dx \approx (b-a)g\left(\frac{a+b}{2}\right)$$

In [1]:
```
import requests
from IPython.display import Image

# Download and save the data file Riemann_sum_convergence.png
url = ('https://raw.githubusercontent.com/rickecon/Notebooks/' +
       'master/NumIntegr/images/Riemann_sum_convergence.png')
image_file = requests.get(url, allow_redirects=True)
open('images/Riemann_sum_convergence.png', 'wb').write(image_file.cont
ent)
Image('images/Riemann_sum_convergence.png')
```

Out[1]:

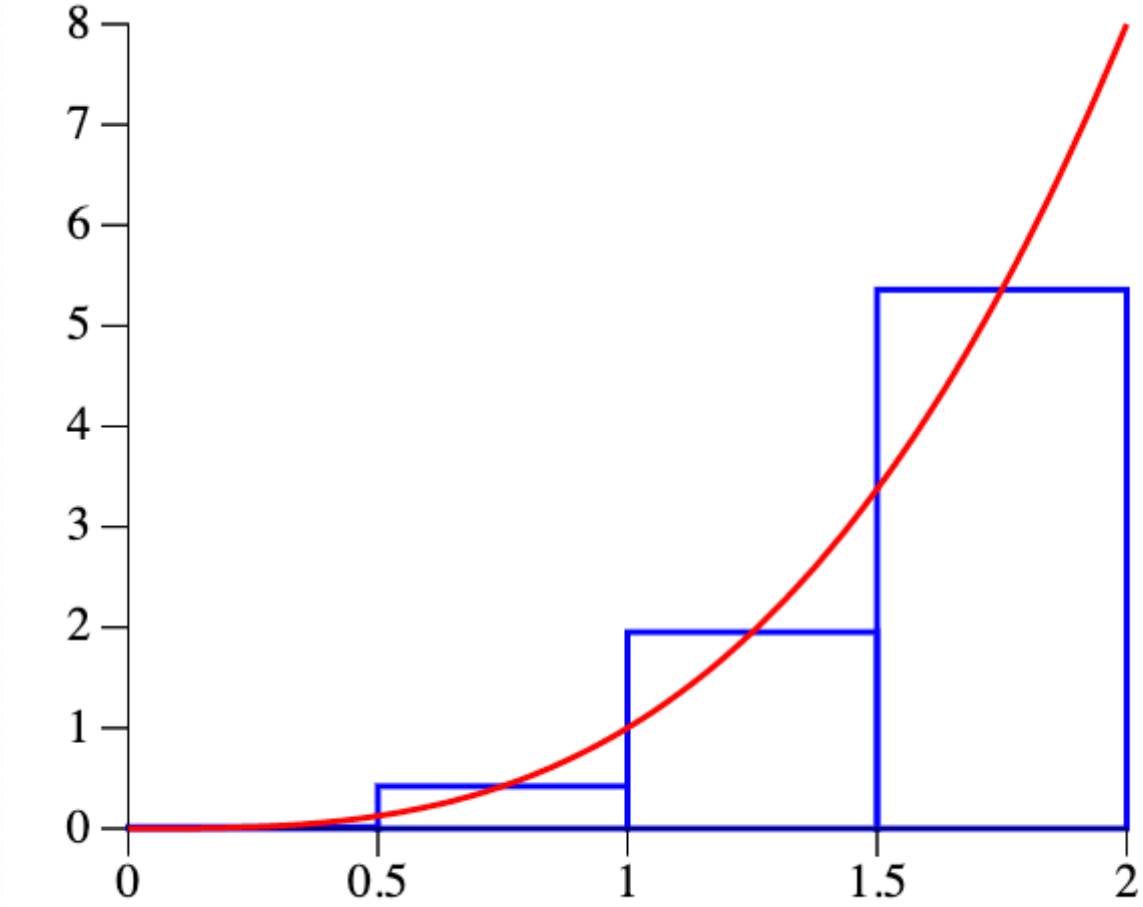


A more sophisticated midpoint rule is the composite midpoint rule, which breaks up the domain of the function $g(x)$ into N intervals and applies the midpoint rule to each interval. For nodes $x_0, x_1, \ldots, x_{N-1}$ with $x_i = a + \frac{(2i+1)(b-a)}{2N}$, the composite midpoint rule is given by

$$\int_a^b g(x)dx \approx \frac{b-a}{N} \sum_{i=0}^{N-1} g(x_i)$$

In [2]:
```
# Download and save the data file MidRiemann2.png
url = ('https://raw.githubusercontent.com/rickecon/Notebooks/' +
       'master/NumIntegr/images/MidRiemann2.png')
image_file = requests.get(url, allow_redirects=True)
open('images/MidRiemann2.png', 'wb').write(image_file.content)
Image('images/MidRiemann2.png')
```

Out[2]:



2.2. Trapezoid rule (2 nodes)

The trapezoid rule estimates the integral as the area under a line that connects the function $g(x)$ at the two endpoints a and b.

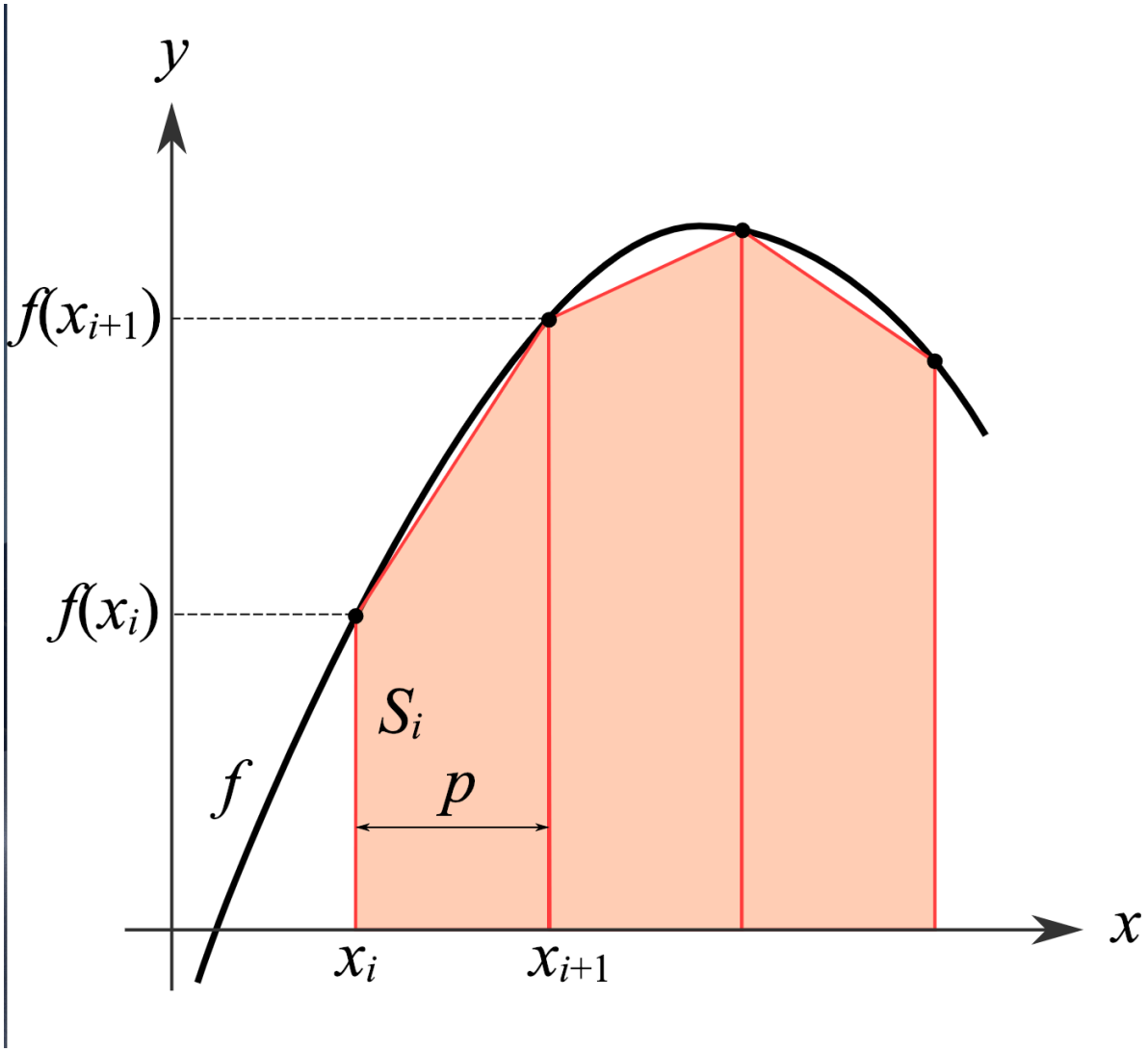
$$\int_a^b g(x)dx \approx \frac{b-a}{2}\Big[g(a)+g(b)\Big]$$

A more sophisticated trapezoid rule is the composite trapezoid rule, which breaks up the domain of the function $g(x)$ into N intervals and applies the trapezoid rule to each interval. For nodes $x_0, x_1, \ldots, x_N$ with $x_i = a + i(b-a)/N$, the composite trapezoid rule is given by

$$\int_a^b g(x)dx \approx \frac{b-a}{2N}\left[g(x_0) + 2\sum_{i=1}^{N-1} g(x_i) + g(x_N)\right]$$

In [3]:
```
# Download and save the data file Integration_num_trapezes_notation.pn
g
url = ('https://raw.githubusercontent.com/rickecon/Notebooks/' +
       'master/NumIntegr/images/Integration_num_trapezes_notation.png'
)
image_file = requests.get(url, allow_redirects=True)
open('images/Integration_num_trapezes_notation.png', 'wb').write(image
_file.content)
Image('images/Integration_num_trapezes_notation.png')
```

Out[3]:



2.3. Simpson's rule (3 nodes)

Simpson's rule offers a smooth nonlinear (quadratic) alternative the linear approximations of the midpoint and trapezoid rules. Simpson's rule finds the unique quadratic function in x that passes through the end points and the midpoint of the function $g(a)$, $g\left(\frac{a+b}{2}\right)$, and $g(b)$, which produces the following weights and values.

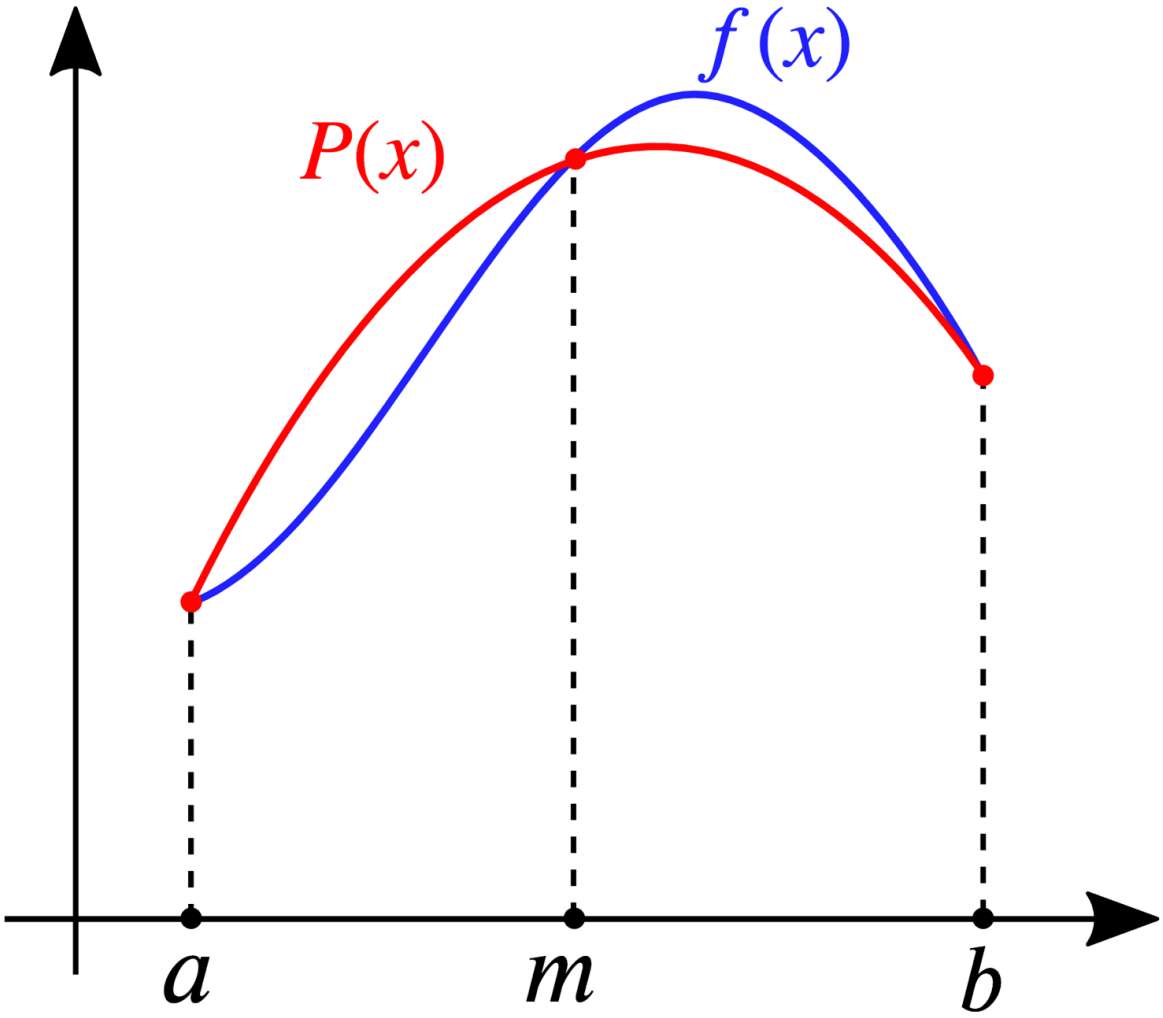
$$\int_a^b g(x)dx \approx \frac{b-a}{6}\left[g(a) + 4g\left(\frac{a+b}{2}\right) + g(b)\right]$$

Again, a more sophisticated Simpson's rule is the composite Simpson's rule, which breaks up the domain of the function $g(x)$ into $2N$ intervals and applies the Simpson's rule to each interval. For nodes $x_0, x_1, \ldots, x_{2N}$ with $x_i = a + i(b-a)/(2N)$, the composite Simpson's rule is given by

$$\int_a^b g(x)dx \approx \frac{b-a}{6N}\left[g(x_0) + 4\sum_{i=1,3,\ldots}^{2N-1} g(x_i) + 2\sum_{i=2,4,\ldots}^{2N-2} g(x_i) + g(x_{2N})\right]$$

In [4]:
```
# Download and save the data file Simpsons_method_illustration.png
url = ('https://raw.githubusercontent.com/rickecon/Notebooks/' +
       'master/NumIntegr/images/Simpsons_method_illustration.png')
image_file = requests.get(url, allow_redirects=True)
open('images/Simpsons_method_illustration.png', 'wb').write(image_file
.content)
Image('images/Simpsons_method_illustration.png')
```

Out[4]:



2.4. Exercises

Exercise 2.1. You can verify that the analytical solution to the integral of the function

$$g(x) = 0.1x^4 - 1.5x^3 + 0.53x^2 + 2x + 1$$

between $x = -10$ and $x = 10$ is $\int_{-10}^{10} g(x)dx = 4,373.3\bar{3}$. Write a Python function that will take as arguments an anonymous function that the user specifies representing $g(x)$, integration bounds a and b, the number of intervals N, and

```
method = {'midpoint', 'trapezoid', 'Simpsons'}
```

Using the composite methods, evaluate the numerical approximations of the integral $\int_a^b g(x)dx$ using all three Newton-Cotes methods in your function and compare the difference between the values of these integrals to the true analytical value of the integral.

a) Plot the absolute error of the difference between the approximated integral and the analytic integral for each of the three methods for different values of N (number of bars). That is, create three different line plots--one for each Newton-Cotes rule--for which the x-axis is different numbers of nodes and the y-axis is the absolute approximation error for each number of nodes. Plot each of these lines for $N = [20, 21, 22, \ldots 200]$. You can create this vector with the code `Nvec = np.arange(20, 201, 1)`.

b) Why are the methods ranked the way they are? [Hint: It has to do with the original function g(x).] You might want to plot the function $g(x)$ between -10 and 10 to see what it looks like.

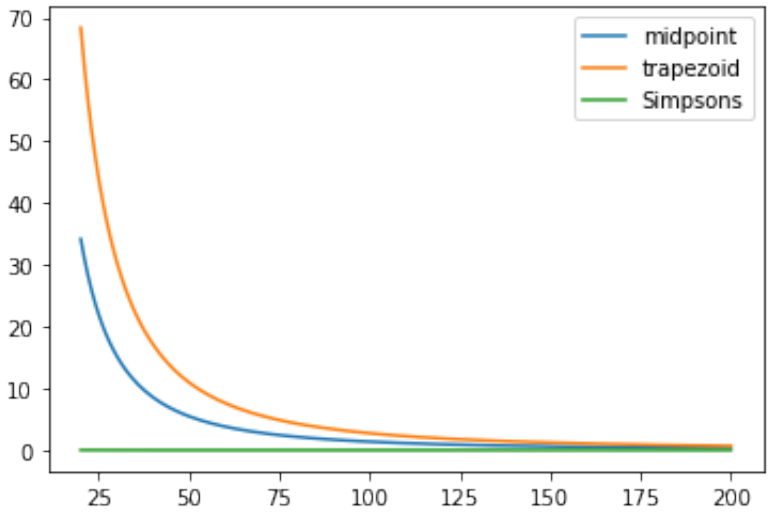
In [3]:
```
import sympy as sy

x = sy.symbols('x')
y = 0.1*x**4 - 1.5*x**3 + 0.53*x**2 + 1
f = sy.lambdify(x, y)
```

In [4]:
```
def intergration(f, a, b, N, method = "midpoint"):
    if method == "midpoint":
        sum = 0
        for i in range(N):
            sum = sum + f(a + (2*i+1)* (b-a)/ (2*N)) * (b-a)/N
        return sum
    elif method == "trapezoid":
        sum = 0
        for i in range(N):
            area = (f(a+i*(b-a)/N) + f(a+(i+1)*(b-a)/N)) * (b-a)/(2*N)
            sum = sum + area
        return sum
    elif method == "Simpsons":
        sum1 = 0
        sum2 = 0
        for i in range(1, 2*N, 2):
            sum1 = sum1 + f(a+ i* (b-a)/(2*N))
        for i in range(2, 2*N-1, 2):
            sum2 =sum2 + f(a+ i* (b-a)/(2*N))
        sum = (f(a) + 4* sum1 + 2* sum2 + f(b))*(b-a)/(6*N)
        return sum

value = 4373 +1 /3
mid_value = intergration(f, -10, 10, 10000, method = "midpoint")
print("The difference of midpoint method is {}".format(abs(mid_value-v
alue)))
trape_value = intergration(f, -10, 10, 10000, method = "trapezoid")
print("The difference of trapezoid method is {}".format(abs(trape_valu
e-value)))
simpsons_value = intergration(f, -10, 10, 10000, method = "Simpsons")
print("The difference of simpsons method is {}".format(abs(simpsons_va
lue-value)))
```

```
The difference of midpoint method is 0.00013686665897694184
The difference of trapezoid method is 0.00027373331795388367
The difference of simpsons method is 2.091837814077735e-11
```

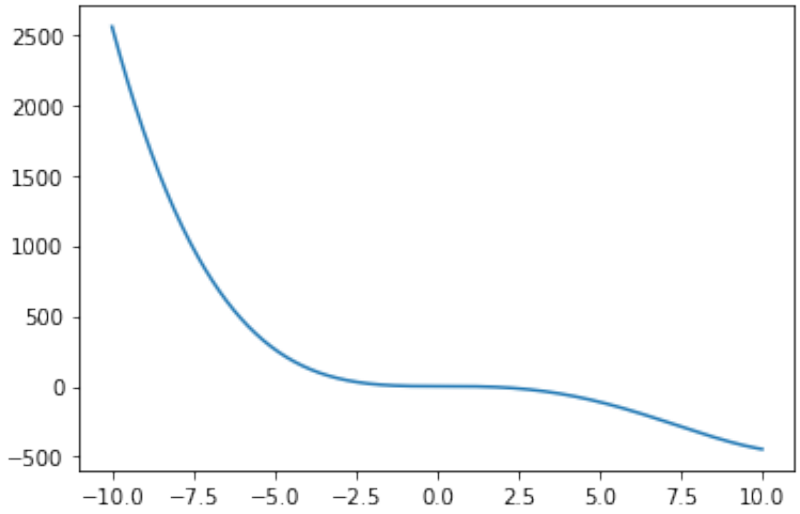
In [5]:
```
from matplotlib import pyplot as plt
import numpy as np
Nvec = np.arange(20, 201, 1)
midpoint = []
trapezoid = []
simpsons = []
for N in Nvec:
    midpoint.append(abs(intergration(f, -10, 10, N, method = "midpoint
") - 4373 - 1/3))
    trapezoid.append(abs(intergration(f, -10, 10, N, method = "trapezo
id") - 4373 - 1/3))
    simpsons.append(abs(intergration(f, -10, 10, N, method = "Simpsons
") - 4373 - 1/3))
plt.plot(Nvec, np.array(midpoint), label = "midpoint")
plt.plot(Nvec, np.array(trapezoid), label = "trapezoid")
plt.plot(Nvec, np.array(simpsons), label = "Simpsons")
plt.legend()
```

Out[5]: <matplotlib.legend.Legend at 0x8227c6610>



In [6]:
```
points = np.linspace(-10, 10, 1000)
plt.plot(points, f(points))
```

Out[6]:
```
[<matplotlib.lines.Line2D at 0x822968050>]
```



When N is small, domain will be divided into a small number of intervals. As we can see from the g(x) graph, if we use trapezoid area to approximate the integration in each interval, it can differ a lot; instead, it will be better to use midpoint rectangale area. And it's very close if we use the integration of quadratic functions to approximate.

As N gets larger, the errors of the former two methods decrease greatly, Because both kind of areas will be closer to the real integration area.

Exercise 2.2. Write a Python function that makes a Newton-Cotes discrete approximation of the distribution of the normally distributed variable $Z \sim N(\mu, \sigma)$. Let this function take as arguments the mean μ, the standard deviation σ, the number of equally spaced nodes N to estimate the distribution, and the number of standard deviations k away from μ to make the furthest nodes on either side of μ. Use the `scipy.stats.norm.cdf` (http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html#scipy.stats.norm) command for the cdf of the normal distribution to compute the weights ω_n for the nodes x_n. Have this function return a vector of nodes of $[Z_1, Z_2, \ldots Z_N]$ and a vector of weights $[\omega_1, \omega_2, \ldots \omega_N]$ such that ω_i is given by the integral under the normal distribution between the midpoints of the two closest nodes. Define $f(Z; \mu, \sigma)$ as the pdf of the normal distribution and $F(Z; \mu, \sigma)$ as the cdf.

$$\omega_i = \begin{cases} F\left(\frac{Z_1+Z_2}{2}; \mu, \sigma\right) & \text{if} \quad i = 1 \\ \int_{Z_{min}}^{Z_{max}} f(Z; \mu, \sigma) dZ & \text{if} \quad 1 < i < N \\ 1 - F\left(\frac{Z_{N-1}+Z_N}{2}; \mu, \sigma\right) & \text{if} \quad i = N \end{cases}$$

$$\text{where} \quad Z_{min} = \frac{Z_{i-1} + Z_i}{2} \quad \text{and} \quad Z_{max} = \frac{Z_i + Z_{i+1}}{2}$$

What are the weights and nodes $\{\omega_n, Z_n\}_{n=1}^{N}$ for $N = 11$?

In [25]:
```
from scipy.stats import norm
import pandas as pd
def normal(mean, sd, N, k):
    Z = np.linspace(mean-k*sd, mean+k*sd, N)
    w = np.zeros(N)
    w[0] = norm.cdf((Z[0]+Z[1])/2, loc = mean, scale = sd)
    w[N-1] = 1 - norm.cdf((Z[N-2]+Z[N-1])/2, loc = mean, scale = sd)
    for i in range(1, N-1):
        w[i] = norm.cdf((Z[i]+Z[i+1])/2, loc = mean, scale = sd) - nor
m.cdf((Z[i-1]+Z[i])/2, loc = mean, scale = sd)
    return Z, w
```

In [29]:
```
Z, w = normal(0, 1, 11, 3)
data = {'Nodes' : Z, 'Weights' : w}
df = pd.DataFrame(data)
df
```

Out[29]:

	Nodes	Weights
0	-3.0	0.003467
1	-2.4	0.014397
2	-1.8	0.048943
3	-1.2	0.117253
4	-0.6	0.198028
5	0.0	0.235823
6	0.6	0.198028
7	1.2	0.117253
8	1.8	0.048943
9	2.4	0.014397
10	3.0	0.003467

Exercise 2.3. If $Z \sim N(\mu, \sigma)$, then $A \equiv e^Z \sim LN(\mu, \sigma)$ is distributed lognormally and $\log(A) \sim N(\mu, \sigma)$. Use your knowledge that $A \equiv e^Z$, $\log(A) \sim N(\mu, \sigma)$, and your function from Exercise 2.2 to write a function that gives a discrete approximation to the lognormal distribution. Note: You will not end up with evenly spaced nodes $[A_1, A_2, \ldots A_N]$, but your weights should be the same as in Exercise 2.2.

In [32]:
```
def lognormal(mean, sd, N, k):
    Z, w = normal(mean, sd, N, k)
    A = np.zeros(N)
    for i in range(N):
        A[i] = np.exp(Z[i])
    return A, w
```

In [33]:
```
A, w = lognormal(0, 1, 11, 3)
data = {'Nodes' : A, 'Weights' : w}
df = pd.DataFrame(data)
df
```

Out[33]:

	Nodes	Weights
0	0.049787	0.003467
1	0.090718	0.014397
2	0.165299	0.048943
3	0.301194	0.117253
4	0.548812	0.198028
5	1.000000	0.235823
6	1.822119	0.198028
7	3.320117	0.117253
8	6.049647	0.048943
9	11.023176	0.014397
10	20.085537	0.003467

Exercise 2.4. Let Y_i represent the income of individual i in the United States for all individuals i. Assume that income Y_i is lognormally distributed in the U.S. according to $Y_i \sim LN(\mu, \sigma)$, where the mean of log income is $\mu = 10.5$ and the standard deviation of log income is $\sigma = 0.8$. Use your function from Exercise 2.3 to compute an approximation of the expected value of income or average income in the U.S. How does your approximation compare to the exact expected value of $E[Y] = e^{\mu + \frac{\sigma^2}{2}}$?

In [52]:
```
A, w = lognormal(10.5, 0.8, 1000, 9)
estimation =np.sum(np.multiply(A, w))
value = np.exp(10.5+0.5*(0.8**2))
print("The difference is", abs(estimation - value))
```

```
The difference is 0.4329624057209003
```

3. Gaussian Quadrature

3.1. Gaussian Quadrature

In Newton-Cotes quadrature, the nodes are uniformly spaced. Gaussian quadrature formulas for approximating an integral take the same approximation form $\int_a^b g(x)dx \approx \sum_{n=1}^{N} \omega_n g(x_n)$ and optimally choose the weights ω_n and unevenly spaced nodes x_n given the total number of nodes N and some approximating polynomial class $h_i(x)$. The N weights and nodes are chosen to make an *exact integration* relationship hold. That is, for polynomials of order $2N-1$ or less, the N weights and nodes must exactly satisfy

$$\int_a^b h_i(x)dx = \sum_{n=1}^{N} \omega_n h_i(x_n) \quad \text{for} \quad i = 0, 1, \ldots, 2N-1$$

where $h_i(x)$ is an i-order polynomial in x. If the $h_i(x)$ form a basis, this means that every polynomial of degree less than or equal to $2N-1$ will be computed exactly using the N weights and N nodes.

As a simple example, suppose we want to approximate an arbitrary function $g(x)$ with Gaussian quadrature using a simple class of polynomials $h_i(x) = x^i$ and only $N = 2$ weights and nodes. The Gaussian quadrature definition equation above implies a system of four equations used to determine the four variables $(\omega_1, \omega_2, x_1, x_2)$ to approximate the integral $\int_a^b g(x)dx \approx \sum_{n=1}^{N} \omega_n g(x_n)$.

$$\int_a^b dx = \omega_1 + \omega_2$$

$$\int_a^b x dx = \omega_1 x_1 + \omega_2 x_2$$

$$\int_a^b x^2 dx = \omega_1 x_1^2 + \omega_2 x_2^2$$

$$\int_a^b x^3 dx = \omega_1 x_1^3 + \omega_2 x_2^3$$

For $N = 2$, the optimal weights and nodes that solve the system above are $(\omega_1, \omega_2, x_1, x_2) = (1, 1, -0.578, 0.578)$. The Python code to solve this nonlinear system could be a simple root finder such as `scipy.optimize.root` (http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.root.html#scipy.optimize.root) or one of the constrained minimizers in `scipy.optimize.minimize` (http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html#scipy.optimize.minimize). In general, the spacing of the nodes will not be uniform. These nodes and weights will exactly solve the integral of a polynomial of order 3 ($2N-1$).

Some Gaussian quadrature principles:

- The nodes x_n and weights ω_n for Gaussian quadrature are fixed for a given N and basis function $h_i(x)$
- Gaussian quadrature with N nodes is exact for polynomial functions of order up to $2N-1$.
- For a general function, gaussian quadrature approximation error is in proportion to the degree that the function is approximated by a polynomial function.
- The accuracy of the Gaussian quadrature approximation of the integral $\int_a^b g(x)dx$ increases in the number of nodes N.

- The accuracy of the approximation of the integral can also be improved by the choice of polynomial family $h_i(x)$. In particular, the families of orthonormal polynomials have multiple desirable properties. Because of the orthogonality of their coefficients, the system above is easier to solve due to the lack of collinearity. Also, the weights ω_n turn out to be the zeros of the orthogonal polynomial family. Lastly, these orthogonal families of polynomials can give very accurate solutions to integrals of the form $\int_a^b w(x)g(x)dx$, where $w(x)$ is the weighting function of an orthonormal family of polynomials.

For finite integration limits, `scipy.integrate.quad` (http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html#scipy.integrate.quad) uses a Clenshaw-Curtis method which uses Chebyshev orthogonal polynomials as basis functions and uses corresponding Chebyshev moments. If one of the integration limits is infinite, `scipy.integrate.quad` (http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html#scipy.integrate.quad) uses Fourier basis functions with the corresponding Fourier moments.

For a more detailed discussion of the theory behind Gaussian quadrature, see Judd (1998, pp. 257-265) and Heer and Maussner (2009, pp. 599-601). The general applicability of Gaussian quadrature and its accuracy and efficiency advantage over Newton-Cotes formulas is summarized by Judd (1998, p. 265).

> "Even when the asymptotic rate of convergence for Gaussian quadrature is no better than the comparable Newton-Cotes formula, experience shows that Gaussian formulas often outperform the alternative Newton-Cotes formula [in terms of accuracy]."

3.2. Exercises

Exercise 3.1. Approximate the integral of the function in Exercise 2.1 using Gaussian quadrature with $N = 3, (\omega_1, \omega_2, \omega_3, x_1, x_2, x_3)$. Use the class of polynomials $h_i(x) = x^i$. How does the accuracy of your approximated integral compare to the approximations from Exercise 2.1 and the true known value of the integral?

In [57]:
```
import sympy as sy
import numpy as np
from scipy import integrate
from scipy import optimize

x = sy.symbols('x')
y = 0.1*x**4 - 1.5*x**3 + 0.53*x**2 + 1
f = sy.lambdify(x, y)

a = -10
b = 10
N = 3
def func(x):
    func = []
    for i in range(2*N):
        w, n = x[: N], x[N :]
        fi = integrate.quad(lambda t: t**i, a, b)[0]
        for j in range(N):
            fi = fi - w[j] * (n[j]**i)
        func.append(fi)
    return func


def Gaussian(f, a, b, N):
    w0 = [1/N for k in range(N)]
    x0 = [0 for k in range(N)]
    value = w0+x0
    result = optimize.root(func, value).x
    w = result[: N]
    x = result[N :]
    sum = 0
    for i in range(N):
        sum += w[i] * f(x[i])
    return sum
```

In [58]:
```
Gaussian = Gaussian(f , -10, 10, 3)
print(Gaussian)
print("The difference for Gaussian approxiamation is", abs(Gaussian - 4373 -1/3))
```

```
4373.333331273335
The difference for Gaussian approxiamation is 2.059998526704465e-06
```

As we can see, the accuracy of Guassian approxiamation is better than midpoint and trapezoid methods even if the number of nodes for the latter two methods is much larger. When we select enough large number of nodes, the accuracy of simpsons method can outperform Guassian. It's amazing that the method we use most often(midpoint) doesn't have very impressive accuracy.

Exercise 3.2. Use the Python Gaussian quadrature command `scipy.integrate.quad` (http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html#scipy.integrate.quad) to numerically approximate the integral from Exercise 2.1.

$$\int_{-10}^{10} g(x)dx \quad \text{where} \quad g(x) = 0.1x^4 - 1.5x^3 + 0.53x^2 + 2x + 1$$

How does the approximated integral using the `scipy.integrate.quad` (http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html#scipy.integrate.quad) command compare to the known analytical value of the integral?

In [63]:
```
result, error = integrate.quad(f, a, b)
print("The difference is", abs(result - 4373-1/3))
print("The error reported by python is", error)
```

```
The difference is 6.063483048990292e-13
The error reported by python is 8.328471862836829e-11
```

4. Monte Carlo Integration

High-dimensional integration is highly inefficient using the standard one-dimensional methods of Newton-Cotes and Gaussian quadrature. The method of choice in high-dimensional settings is known as Monte Carlo Integration.

In this section, we detail two types of Monte Carlo integration. The standard Monte Carlo simulation approach in Section 4.1 uses pseudo-randomly generated draws from a uniform distribution over the domain of the function in order to approximate the weights and nodes for integration. The quasi-Monte Carlo approach in Section 4.2 uses elements of low-discrepancy sequences over the domain of the function in order to approximate the integral. Both methods have benefits and drawbacks.

4.1. Standard Monte Carlo integration

In the Newton-Cotes quadrature methods of approximating an integral, nodes and weights for the approximation $\sum_{n=1}^{N} \omega_n g(x_n)$ are chosen without much attention to the effect of the placement of these nodes or the levels of the weights on the accuracy of the approximation. Newton-Cotes methods are computationally fast, but lack in accuracy. Gaussian quadrature methods spend more computational time choosing "optimal" weights and nodes, but this gives an accuracy payoff over Newton-Cotes formulas. Monte Carlo integration methods use the computationally fast method of drawing uniformly from the support of the variable of integration. These methods depend on a large number of draws to get high accuracy. Judd (1998, pp. 309-311) spends significant time explaining that these methods are more correctly called "pseudo-Monte Carlo methods" because they make use of pseudorandom number generators, the use of which cannot invoke the law of large numbers or the central limit theorem. However, the biases introduced by pseudorandom number generators are rarely significant in practice. Although Monte Carlo integration methods do not converge as quickly as Gaussian quadrature methods for functions of one variable, they are especially valuable when integrating over functions of multiple variables.

Let $\Omega \subset \mathbb{R}^m$ be the domain of integration. Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ be N uniform random draws from Ω. Then we can write the following approximation of the integral.

$$\int_{\Omega} g(\mathbf{x}) d\mathbf{x} \approx V \frac{1}{N} \sum_{n=1}^{N} g(\mathbf{x}_n) \quad \text{where} \quad V = \int_{\Omega} d\mathbf{x}$$

The equation above says we can approximate the integral of a function $g(\mathbf{x})$ on a domain Ω by taking the average of the evaluations of the function g at N random draws of the vector $\mathbf{x}_n$ multiplied by the volume of the domain.

An easy example of a univariate integral is $\int_0^1 x\, dx$ (here $g(\mathbf{x}) = x$). The Monte Carlo approximation formula for this integral is the following.

$$\int_0^1 x\, dx \approx V \frac{1}{N} \sum_{n=1}^{N} x_n = \frac{1}{N} \sum_{n=1}^{N} x_n$$

It is easy to see that the answer to the exact integral on the left-hand-side of this integral is 1/2. In the approximation on the right-hand-side of this equation, V is the volume of the domain of $x \in [0, 1]$, which is 1. It is straightforward to see that the average of N draws from a uniform distribution between 0 and 1 will converge quickly to 1/2.

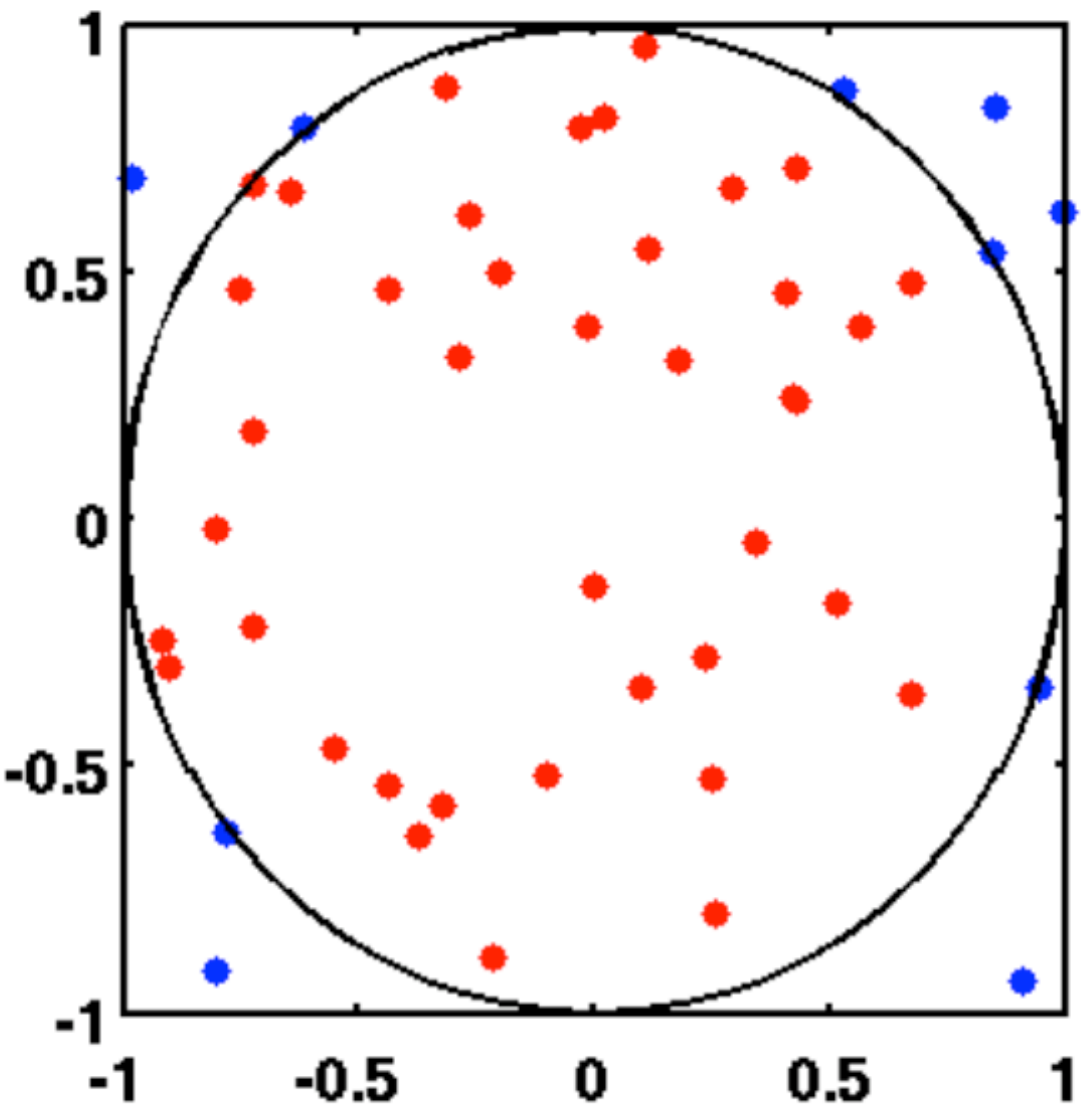
In [2]:
```
# integrate f(x) = x between 0 and 2

import numpy as np
import scipy.stats as sts
import matplotlib.pyplot as plt

# # Approximate integral int_0^1 x dx by Monte Carlo integration
np.random.seed(seed=25)

N = 10
mc_draws = 2 * sts.uniform.rvs(size=N)
print(mc_draws)
approx_int = 2 * (1 / N) * mc_draws.sum()
print('For N=', N, ', MC approx integral =', approx_int)
```

```
[1.74024827 1.16455386 0.55767788 0.37182246 0.82220026 0.23475109
 1.36993749 0.87522212 1.11245865 0.73416064]
For N= 10 , MC approx integral = 1.796606545590371
```

Exercise 4.1 lets you try your hand at coding a classic Monte Carlo integration approximation of the integral of a function of two variables to approximate the value of π. The area of a circle with radius $r = 1$ is π (πr^2). A way to visualize the Monte Carlo approximation of the area of that circle, or π, is to enclose the circle in a square with sides of length 2, in which the x-axis goes from -1 to 1 and the y-axis goes from -1 to 1. The points in the Figure are the uniformly distributed random draws from $(x, y) \in [-1, 1] \times [-1, 1]$. Intuitively, the area of the circle is the fraction of the dots (red dots divided by total dots) that are inside the circle or on the boundary of the circle, multiplied by the area or volume of the square in which the circle lies.

In [8]:
```
# Download and save the data file MonteCarloCircle.png
url = ('https://raw.githubusercontent.com/rickecon/Notebooks/' +
       'master/NumIntegr/images/MonteCarloCircle.png')
image_file = requests.get(url, allow_redirects=True)
open('images/MonteCarloCircle.png', 'wb').write(image_file.content)
Image('images/MonteCarloCircle.png')
```

Out[8]:



Following the intuition of the previous paragraph and of the figure above, the exact area of the circle can be written as an integral of the indicator function of coordinate variables x and y in the following way.

$$\text{Area} = \int_{\Omega} g(x,y)dx\,dy = \pi$$

$$\text{where} \quad g(x,y) = \begin{cases} 1 & \text{if} \quad x^2+y^2 \leq 1 \\ 0 & \text{else} \end{cases} \quad \text{and} \quad \Omega = [-1,1]\times[-1,1]$$

The exact integral for the area of a unit radius circle can be Monte Carlo approximated using the form above resulting in the following function.

$$\int_{\Omega} g(x,y)dx\,dy \approx 4\frac{1}{N}\sum_{n=1}^{N} g\left(x_n, y_n\right)$$

4.2. Quasi-Monte Carlo integration

It is important to realize what Monte Carlo methods really are in practice. Due to the impracticality of generating truly "random" sequences, Monte Carlo methods utilize pseudorandom sequences. Any sequence generated using a pseudorandom number generator will have a small amount of artificial correlation, and this problem is compounded in higher dimensions.

Quasi-Monte Carlo methods dispense with the attempt to create deterministic samples that mimic random samples, and instead embrace their deterministic character. Judd (1998, ch. 9) defines quasi-Monte Carlo methods as sampling methods that do not rely on probabilistic ideas and pseudorandom sequences for constructing the sample and analyzing the estimate. Quasi-Monte Carlo methods use the same approximating function $\sum_{n=1}^{N} \omega_n g(x_n)$, but draw on number theory and Fourier analytic methods to create low-discrepancy sequences that are used as sample points. However, other than the selcetion of sample points, quasi-Monte Carlo integration proceeds in exactly the same way as standard Monte Carlo integration as detailed in the previous section.

Many different deterministic sequences can be used in quasi-Monte Carlo sampling. All of these "quasirandom" sequences strive for uniformity in a general sense. Hence it is useful to have a precise way of measuring the degree to which a point set exhibits uniformity. If we have a uniformly distributed sequence $\mathbf{x}_n$ in the s-dimensional unit cube $I^s = [0,1)^s$, we would intuitively expect that every subset of I^s with the same volume would contain the same number of points. This idea is described precisely by the discrepancy of $\mathbf{x}_n$. We first define local discrepancy, and then define global discrepancy.

For N points $\{\mathbf{x}_n\}_{n=1}^{N}$ in I^s, $s \geq 0$, and $J \subseteq I^s$, the local discrepancy $D(J;N)$ is defined by

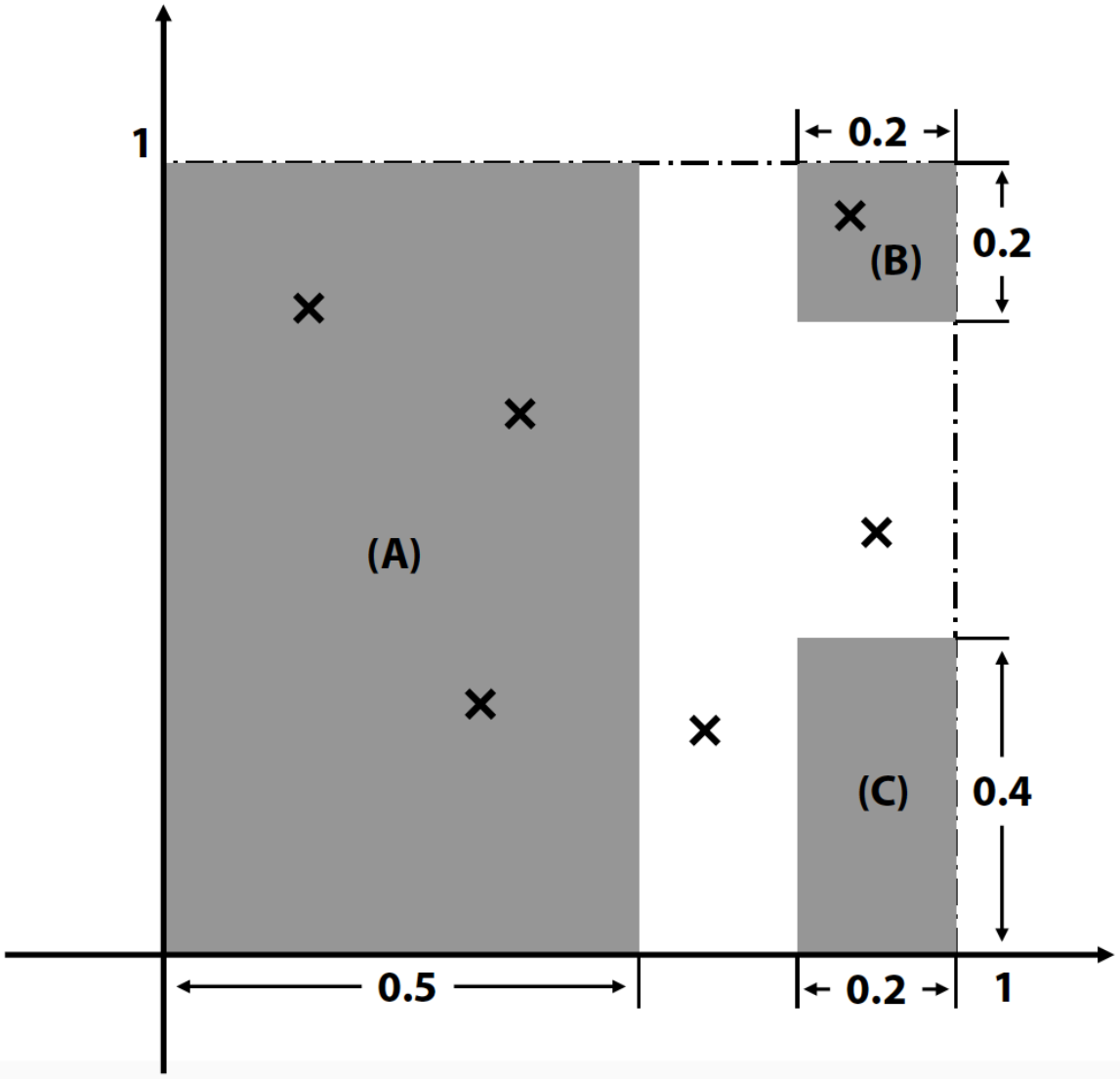
$$D(J;N) = S(J;N) - V(J)N,$$

where $V(J)$ is the volume of the subinterval J and $S(J;N)$ is the number of points from $\{\mathbf{x}_n\}_{n=1}^{N}$ that are in J. If the N points are uniformly distributed, then the local discrepancy should be very small for all J's.

The figure below shows a 2-dimensional unit cube $I^s = [0,1)^2$ with six points. Three sub-intervals A, B, and C are shaded. The local discrepancy of A, which contains 3 points, is calculated from the equation above as $D(A;6) = S(A;6) - V(A)6 = 3 - 6/2 = 0$. Similarly, the local discrepancies of B and C are 0.1267 and 0.8, respectively.

In [9]:
```
# Download and save the data file NumInt_Discrepancy.png
url = ('https://raw.githubusercontent.com/rickecon/Notebooks/' +
       'master/NumIntegr/images/NumInt_Discrepancy.png')
image_file = requests.get(url, allow_redirects=True)
open('images/NumInt_Discrepancy.png', 'wb').write(image_file.content)
Image('images/NumInt_Discrepancy.png')
```

Out[9]:



A global concept of discrepancy is given by the star-discrepancy. We define the star-discrepancy $\Delta(N)$ of N points by,

$$\Delta(N) = \sup_{J} |D(J; N)|,$$

where the supremum is taken over all subsets J of the form $J = \prod_{i=1}^{s} [0, u_i)$. The star-discrepancy can be thought of as the worst-case local discrepancy, looking only at subintervals that have the origin as a corner.

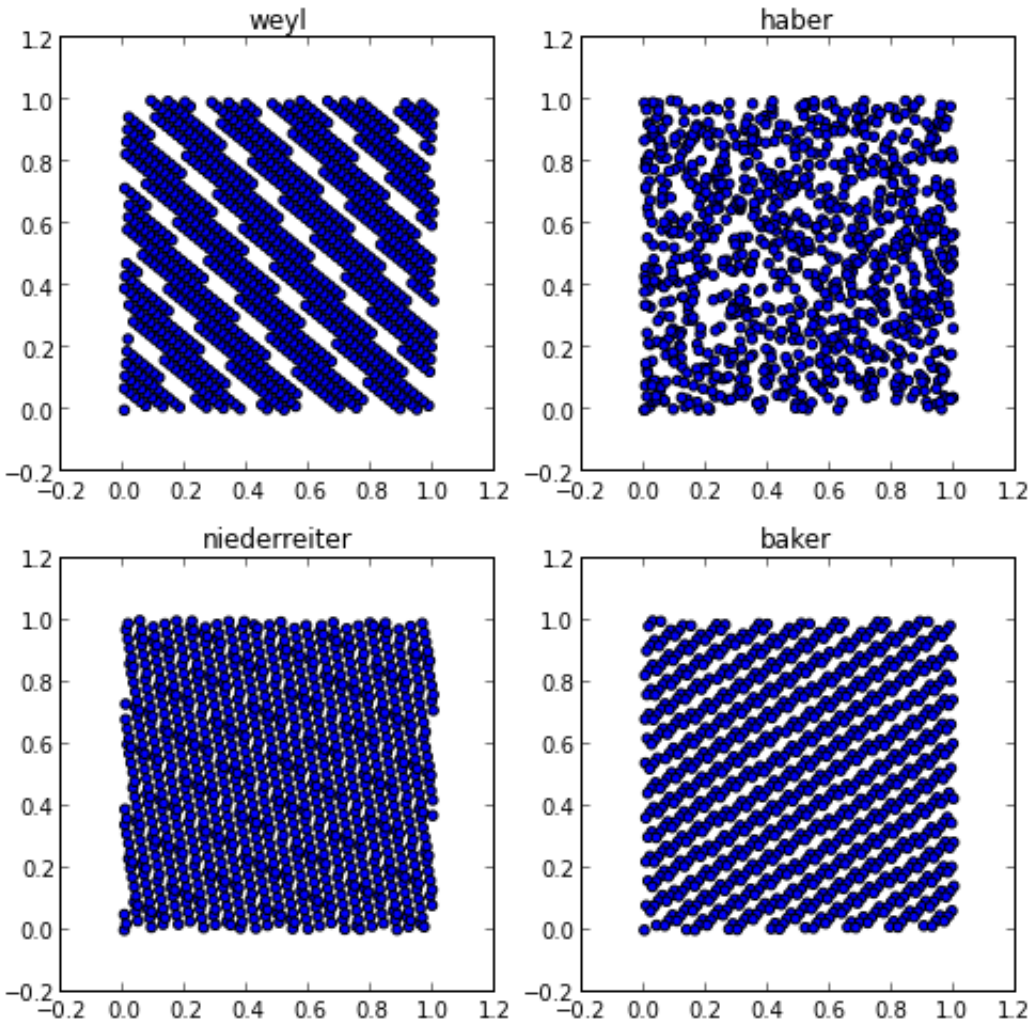
A common class of sequences used in quasi-Monte Carlo sampling is equidistributed sequences. Equidistributed sequences sequences where the star-discrepancy $\Delta(N)$ tends to zero as N tends to infinity. In other words, in the limit the proportion of terms falling in any subinterval is proportional to the length of that interval.

There are a number of equidistributed sequences, and here we will provide some examples. By way of notation, let $p_1, p_2, \ldots$ denote the sequence of prime numbers $2, 3, 5, \ldots$, and let $\langle x \rangle$ represent the fractional part of x, that is $\langle x \rangle = x - \lfloor x \rfloor$, where $\lfloor x \rfloor$ is the rounded down integer of x. Thus each element of vector $\langle x \rangle$ is between 0 and 1. The table below contains formulas for a number of s-dimensional equidistributed sequences on $[0, 1)^s$. The figure below shows the first 1,000 points for two-dimensional Weyl, Haber, Niederreiter, and Baker sequences.

Name of Sequence	**Formula for** $(x_1, x_2, \ldots, x_s)_n$
Weyl	$(\langle np_1^{1/2} \rangle, \ldots, \langle np_s^{1/2} \rangle)$
Haber	$\left(\left\langle \frac{n(n+1)}{2} p_1^{1/2} \right\rangle, \ldots, \left\langle \frac{n(n+1)}{2} p_s^{1/2} \right\rangle\right)$
Niederreiter	$\left(\left\langle n \left(2^{1/(s+1)}\right) \right\rangle, \ldots, \left\langle n \left(2^{s/(s+1)}\right) \right\rangle\right)$
Baker	$(\langle ne^{r_1} \rangle, \ldots, \langle ne^{r_s} \rangle)$ r_j rational and distinct

In [10]:
```
# Download and save the data file NumInt_ComparisonWHNB1000.png
url = ('https://raw.githubusercontent.com/rickecon/Notebooks/' +
       'master/NumIntegr/images/NumInt_ComparisonWHNB1000.png')
image_file = requests.get(url, allow_redirects=True)
open('images/NumInt_ComparisonWHNB1000.png', 'wb').write(image_file.co
ntent)
Image('images/NumInt_ComparisonWHNB1000.png')
```

Out[10]:



Because the equidistributed sequences shown in the table and figure above rely on ascending sequences of prime numbers, I include some functions to produce these sequences of primes. The first function `isPrime(n)` tells you whether a number n is a prime or not. The second function `primes_ascend(N)` returns a vector of length N of the first N primes starting from minimum value of 2 (this is the default).

In [1]:
```
import numpy as np

def isPrime(n):
    '''
    ------------------------------------------------------------------
--
    This function returns a boolean indicating whether an integer n is
a
    prime number
    ------------------------------------------------------------------
--
    INPUTS:
    n = scalar, any scalar value

    OTHER FUNCTIONS AND FILES CALLED BY THIS FUNCTION: None

    OBJECTS CREATED WITHIN FUNCTION:
    i = integer in [2, sqrt(n)]

    FILES CREATED BY THIS FUNCTION: None

    RETURN: boolean
    ------------------------------------------------------------------
--
    '''
    for i in range(2, int(np.sqrt(n) + 1)):
        if n % i == 0:
            return False

    return True
```

In [2]:
```
def primes_ascend(N, min_val=2):
    '''
    ------------------------------------------------------------------
--
    This function generates an ordered sequence of N consecutive prime
    numbers, the smallest of which is greater than or equal to 1 using
    the Sieve of Eratosthenes algorithm.
    (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes)
    ------------------------------------------------------------------
--
    INPUTS:
    N       = integer, number of elements in sequence of consecutive
              prime numbers
    min_val = scalar >= 2, the smallest prime number in the consecutiv
e
              sequence must be greater-than-or-equal-to this value

    OTHER FUNCTIONS AND FILES CALLED BY THIS FUNCTION:
```

```
        isPrime()

    OBJECTS CREATED WITHIN FUNCTION:
    primes_vec     = (N,) vector, consecutive prime numbers greater th
an
                     min_val
    MinIsEven      = boolean, =True if min_val is even, =False otherwi
se
    MinIsGrtrThn2  = boolean, =True if min_val is
                     greater-than-or-equal-to 2, =False otherwise
    curr_prime_ind = integer >= 0, running count of prime numbers foun
d

    FILES CREATED BY THIS FUNCTION: None

    RETURN: primes_vec
    ------------------------------------------------------------------
--
    '''
    primes_vec = np.zeros(N, dtype=int)
    MinIsEven = 1 - min_val % 2
    MinIsGrtrThn2 = min_val > 2
    curr_prime_ind = 0
    if not MinIsGrtrThn2:
        i = 2
        curr_prime_ind += 1
        primes_vec[0] = i
    i = min(3, min_val + (MinIsEven * 1))
    while curr_prime_ind < N:
        if isPrime(i):
            curr_prime_ind += 1
            primes_vec[curr_prime_ind - 1] = i
        i += 2

    return primes_vec
```

In [3]:
```
primes_ascend(10)
```

Out[3]:
```
array([ 2,  3,  5,  7, 11, 13, 17, 19, 23, 29])
```

A key distinction between quasirandom sequences and pseudorandom sequences is that quasirandom sequences do not "look like" random numbers. As can be seen in the figures above, they generally display quite obvious patterns. From the outset, quasirandom sequences are chosen so as to have low discrepancy, and are not encumbered by any other requirements of random numbers.

However, equidistribution is a rather weak criterion to express the idea that a sequence is uniform. Even though the discrepancy approaches zero as n approaches infinity, the Weyl sequence in the figure above shows how there will often be large gaps for small n. An alternative approach is to use low-discrepancy sequences. The goal of a low-discrepancy sequence is to minimize the star-discrepancy of every subsequence. In contrast to equidistributed sequences, the algorithms to generate these sequences take into account the total number of points desired so that maximum uniformity is achieved for every subsequence, and not just in the limit.

Halton sequences describe a class of low-discrepancy multidimensional sequences that fill the interval $[0, 1)$. To construct a Halton sequence, begin with a consecutive sequence of positive integers of length N, for example $n = 1, 2, \ldots, N$. Now choose any prime number p and convert each integer n into its representation in the base p number system. For multiple dimensions, repeat the process with a different prime p. Then reflect the base p representation of each integer about the decimal point to obtain a number in the interval $[0, 1)$.

Many other techniques exist for creating low-discrepancy sequences. Faure' sequences are permutations of Halton sequences. Sobol sequences are a reordering of Halton sequences. Other methods include (t, m, s)-Nets and the method of good lattice points. Niederreiter (1978) is a good resource on low-discrepancy sequences in quasi-Monte Carlo methods.

Quasi-Monte Carlo methods do far better asymptotically than any Monte Carlo method for many problems. With N points in s dimensions, quasi-Monte Carlo techniques have a worst-case convergence rate of $O\left(\frac{(\log N)^s}{N}\right)$ as opposed to $O\left(\frac{1}{\sqrt{N}}\right)$ for standard Monte Carlo techniques. However, standard Monte Carlo integration is easier to implement properly and is generally sufficient for most purposes.

4.3. Exercises

Exercise 4.1. Use Monte Carlo integration to approximate the value of π. Define a function in that takes as arguments a function $g(\mathbf{x})$ of a vector of variables $\mathbf{x}$, the domain Ω of $\mathbf{x}$, and the number of random draws N and returns the Monte Carlo approximation of the integral $\int_\Omega g(\mathbf{x})d\mathbf{x}$. Let Ω be a generalized rectangle--width x and height y. In order to approximate π, let the functional form of the anonymous function be $g(x, y)$ from Section 4.1 with domain $\Omega = [-1, 1] \times [-1, 1]$. What is the smallest number of random draws N from Ω that matches the true value of π to the 4th decimal 3.1415? Set the random seed in your uniform random number generator to 25. This will make the correct answer consistent across submissions.

In [11]:
```
import numpy as np
import scipy.stats as sts
import matplotlib.pyplot as plt

np.random.seed(25)
def f(x, y):
    if x**2 + y**2 <= 1:
        return 1
    else:
        return 0

def Monte_Carlo(f, domain, N):
    x_draws = sts.uniform.rvs(loc = domain[0][0], scale = domain[0][1]
- domain[0][0], size=N)
    y_draws = sts.uniform.rvs(loc = domain[1][0], scale = domain[1][1]
- domain[1][0], size=N)
    sum = 0
    for i in range(N):
        sum += f(x_draws[i], y_draws[i])
    size = (domain[0][1]-domain[0][0])*(domain[1][1] - domain[1][0])
    approx_int = sum / N * size
    return approx_int
```

In [13]:
```
Monte_Carlo(f, [[-1,1],[-1,1]], 2884)
```

Out[13]:
```
3.144244105409154
```

In [14]:
```
i = 1
while round(Monte_Carlo(f, [[-1,1], [-1,1]], i), 4) != 3.1415:
    i += 1
else:
    print("The smallest number is", i)
```

```
The smallest number is 1491
```

Exercise 4.2. Define a function in that returns the n-th element of a d-dimensional equidistributed sequence. It should have support for the four sequences in the Table in Section 4.2.

In [4]:
```
def equidistributed(n, d, type):
    prime_seq = primes_ascend(d)
    if type == "Weyl":
        seq = np.sqrt(prime_seq) * n
        seq = seq - np.floor(seq)
        return seq
    elif type == "Haber":
        seq = np.sqrt(prime_seq) * n * (n + 1) / 2
        seq = seq - np.floor(seq)
        return seq
    elif type == "Niederreiter":
        power = [i / (n + 1) for i in range(1, d + 1)]
        seq = n * np.power(2, power)
        seq = seq- np.floor(seq)
        return seq
    elif type =="Baker":
        seq = n * np.exp(1/prime_seq)
        seq = seq- np.floor(seq)
        return seq
```

In [5]:
```
print(equidistributed(10, 3, "Weyl"))
print(equidistributed(10, 3,"Haber"))
print(equidistributed(10, 3,"Niederreiter"))
print(equidistributed(10, 3,"Baker"))
```

```
[0.14213562 0.32050808 0.36067977]
[0.78174593 0.26279442 0.98373876]
[0.65041089 0.34312522 0.08089444]
[0.48721271 0.95612425 0.21402758]
```

Exercise 4.3 Repeat Exercise 4.1 to approximate the value of π, this time using quasi-Monte Carlo integration. You will need to appropriately scale the equidistributed sequences. Compare the rates of convergence. What is the smallest number of random draws N from Ω for the quasi-Monte Carlo integration that matches the true value of π to the 4th decimal 3.1415?. Set the seed in your uniform random number generator to 25. This will make the correct answer consistent across submissions.

In [14]:
```
import numpy as np
def smallest_draw(type):
    np.random.seed(25)
    N = 1
    while N >= 0:
        x_draws = [2 * equidistributed(i, 2, type)[0] - 1 for i in ran
ge(N)]
        y_draws = [2 * equidistributed(i, 2, type)[1] - 1 for i in ran
ge(N)]
        pi = 4 * np.mean(list(map(f, x_draws, y_draws)))
        if round(pi, 4) != 3.1415:
            N += 1
        else:
            print("The smallest number of", type, "is", N)
            break
        if N > 3000:
            print("The smallest number of", type, "is larger than 3000
")
            break
```

In [15]:
```
smallest_draw("Weyl")
```

```
The smallest number of Weyl is 1230
```

In [16]:
```
smallest_draw("Haber")
```

```
The smallest number of Haber is 2064
```

In [17]:
```
smallest_draw("Niederreiter")
```

```
The smallest number of Niederreiter is larger than 3000
```

In [18]:
```
smallest_draw("Baker")
```

```
The smallest number of Baker is 1067
```

Baker type solution has the smallest number of random draws, so its covergence rate is best, following by Weyl, Monte-Carlo, Haber, Niederreiter.

5. Sparse Grids

Sparse grid interpolation is a method of approximating functions with many dimensions. A direct way of representing multidimensional functions is to use a full grid, such as in Newton-Cotes quadrature. However, using this method the number of grid points that have to be computed depends exponentially on the number of dimensions. Because of this curse of dimensionality, processing functions of beyond five or six dimensions becomes intractable using the techniques of Sections 2 and 3. While Monte Carlo methods perform reasonably well in multidimensional integration, sparse grids remain the most efficient.

The sparse grid method selects the nodes of integration by a special truncation of the tensor product expansion of a one-dimensional multilevel selection of nodes. To construct a sparse grid, we first select a series of one-dimensional quadrature formulas indexed by l for a univariate function f, and write it as

$$Q_l^{(1)} f = \sum_{i=1}^{N_l} w_{li} f(x_{li}).$$

For example, the nodes and weights of this initial quadrature formula series could be based on the trapezoid rule or Simpson's rule with N_l nodes. Now define the difference formulas by,

$$\Delta_k^{(1)} f = (Q_k^{(1)} - Q_{k-1}^{(1)}) f$$

where $Q_0^{(1)} f = 0$. Note that the differences $\Delta_k^{(1)} f$ are just univariate quadrature formulas.

For a given level $l \in \mathbb{N}$, the sparse gride integration approximation for a d-dimensional function f is given by

$$Q_l^{(d)} = \sum_{||\mathbf{k}|| \leq l+d-1} (\Delta_{k_1}^{(1)} \otimes \ldots \otimes \Delta_{k_d}^{(1)}) f.$$

Using the quadrature rule in the equation above, every possible tensor product of the difference formulas is considered, but only those whose sum of indices is smaller than the constant $l + d - 1$ are used.

The selcetion of nodes in a two-dimensional sparse grid of level $l = 3$ using the trapezoid rule as the intial formula is visualized in the figure below. Along the top and left are the one-dimensional grid points for $l = 1, 2, 3$ in the x- and y-directions. These points are used to create the corresponding product grids $\Delta_{k_1} \otimes \Delta_{k_2}$ for $1 \leq k_1, k_2 \leq 3$. Because the grid points used in the trapezoid rule are nested, many of the grid points are cancelled out, and the only ones that remain are formed from a union of the grids along the diagonal as indicated by the black line. The resulting sparse grid $Q_3^{(2)}$ is shown on the right.

In [ ]:
```
# Download and save the data file NumInt_TrapezoidGrid.png
url = ('https://raw.githubusercontent.com/rickecon/Notebooks/' +
       'master/NumIntegr/images/NumInt_TrapezoidGrid.png')
image_file = requests.get(url, allow_redirects=True)
open('images/NumInt_TrapezoidGrid.png', 'wb').write(image_file.content
)
Image('images/NumInt_TrapezoidGrid.png')
```

6. Discrete Markov Approximation of Continuous AR(1) Process

Suppose you have a random shock z_t in your model that has some persistence according to the following AR(1) process.

$$z_{t+1} = \rho z_t + (1-\rho)\mu + \varepsilon_{t+1} \quad \text{where} \quad \varepsilon_t \sim N(0,\sigma) \quad \text{and} \quad \rho \in (0,1)$$

The expected value of z_{t+1} is conditional on the current realization of the shock $E[z_{t+1}|z_t] = \rho z_t + (1-\rho)\mu$ but the variance of z_{t+1} is unconditional $Var[z_{t+1}] = \sigma^2$. The AR(1) process in this equation generates a variable that fluctuates around its mean μ, and the expected value of the variable tomorrow $E[z_{t+1}|z_t]$ is some convex combination of the variable today z_t and the mean μ.

Typical examples of these types of shocks in economics are shocks to ability, health status, and productivity shocks--all of which exhibit persistence or dependence on recent values. If the shock must be strictly positive, as is the case with productivity shocks, the variable z_t is simply exponentiated.

$$Y_t = A_t K_t^{\alpha} L_t^{1-\alpha} \quad \text{where} \quad A_t = e^{z_t}$$

Notice that the variable A_t is lognormally distributed $A_t \sim LN\Big(\rho z_{t-1} + (1-\rho)\mu, \sigma\Big)$ because $\log(A_t) = z_t$ and $z_t \sim N\Big(\rho z_{t-1} + (1-\rho)\mu, \sigma\Big)$. You made a discretized approximation of the i.i.d. (no persistence) version of this distribution in Exercise 2.3 and estimated average income in the U.S. using it in Exercise 2.4.

Tauchen and Hussey (1991) describe a quadrature-based method for producing efficient nodes and probabilities of a discrete first-order Markov process to approximate a continuous AR(1) random variable. Tauchen (1986) details a simpler non-quadrature based method for producing efficient nodes and probabilities of a discrete first-order Markov process to approximate a continuous AR(1) random variable. A classic example of where this discretization is extremely valuable is in the stochastic intertemporal Euler equation from Section 1.

$$u'(c_t) = \beta E_{z_{t+1}|z_t}\Big[(1 + r_{t+1} - \delta)u'(c_{t+1})\Big]$$

$$\Rightarrow \quad u'(c_t) = \beta \int_a^b \Big(1 + r_{t+1}(z_{t+1}) - \delta\Big)u'\Big(c_{t+1}(z_{t+1})\Big) f(z_{t+1}|z_t) dz_{t+1}$$

The expectation on the right-hand-side of the Euler equation is over z_{t+1} given z_t, where z_{t+1} is the AR(1) process described at the beginning of this section. One of the most common nonlinear solution techniques for the functional equations of the dynamic household decision problem characterized by this Euler equation is value function iteration on the following recursive Bellman equation.

$$V(k,z) = \max_{k'} \; u(k,z,k') + \beta E_{z'|z}\Big[V(k',z')\Big]$$

The expectation on the right-hand-side of the Bellman equation is simply an integral of the form $E_{z'|z}\Big[V(k',z')\Big] = \int_{z'} V(k',z') f(z'|z) dz'$. However, it is difficult to use standard Gaussian quadrature or Monte Carlo integration methods because the value function $V(k',z')$ is often only known at a few points.

One solution to this problem is to interpolate or fit some continuous function $\tilde{V}(k', z')$ to the known points of $V(k', z')$ and then use Gaussian quadrature or Monte Carlo integration to approximate the integral $\int_{z'} \tilde{V}(k', z') f(z'|z) dz'$. Heer and Maussner (2008) and Heer and Maussner (2009, p. 237) find that the errors in the extrapolated values of the interpolated function $\tilde{V}$ beyond the bounds of the known points of V cause the solution to be less accurate than using the Tauchen-Hussey method of approximating $f(z'|z)$ with a discrete first order Markov process.

7. References

- Adda, Jerome and Russell Cooper, *Dynamic Economics: Quantitative Methods and Applications*, MIT Press (2003).
- Armington, Paul S., "A Theory of Demand for Products Distinguished by Place of Production," IMF Staff Papers, 16:1, March (1969).
- Dixit, Avinash K. and Joseph E. Stiglitz, "Monopolistic Competition and Optimum Product Diversity," *American Economic Review*, 67:3, pp. 297-308, June (1977).
- Heer, Burkhard and Alfred Maussner, "Computation of Business Cycle Models: A Comparison of Numerical Methods," *Macroeconomic Dynamics*, 12:5, pp.641-663, November (2008).
- Heer, Burkhard and Alfred Maussner, *Dynamic General Equilibrium Modeling: Computational Methods and Applications*, 2nd edition, Springer (2009).
- Judd, Kenneth L., *Numerical Methods in Economics*, MIT Press, (1998).
- Niederreiter, Harald, "Quasi-Monte Carlo Methods and Pseudo-Random Numbers," *Bulletin of the American Mathematical Society*, 84:6, November (1978).
- Tauchen, George, "Finite State Markov-chain Approximation to Univariate and Vector Autoregression," *Economics Letters*, 20:2, pp. 177-181 (1986).
- Tauchen, George and Robert Hussey, "Quadrature-based Methods for Obtaining Approximate Solutions to Nonlinear Asset Pricing Models," *Econometrica*, 59:2, pp. 371-396, March (1991).

In [ ]: