NUMERICAL INTEGRATION IN PYTHON

You are learning different techniques for finding antiderivatives to apply the Fundamental Theorem of Calculus to compute definite integrals. But not every function has a (straightforward) antiderivative! In this lab, we will look at techniques for approximating definite integrals when the Fundamental Theorem cannot be easily applied.

NOTE: The example here (and in future overviews) is NOT a copy/paste to solve the problems in lab. However, it will USE many of the features you will use to solve your problems, such as (in this case) differentiating functions, finding slopes and equations of tangent lines, solving equations, and plotting.

EXAMPLE:

Given the function f(x) = e^(-x^2), approximate the integral of f from x=0 to x=2 using right endpoint rectangles (recall 5.1-5.2 in 151):

a) using 4 subintervals

b) using 8 subintervals

c) using 50 subintervals

Refer to Theorem 4 of Section 5.2 (p380 in the Stewart text). If we drop the limit, we obtain the following approximation:

integral is approximately equal to the sum of f(x_i) *dx, where dx = (b-a)/n and x_i = a + i* dx.

NOTE: This lab uses several commands from different packages. Therefore, we will import everything from the numpy package, and use a shorthand notation for the other packages (you may already be familiar with this using **matplotlib.pyplot** for numerical plotting in other courses). So our introductory lines will be the following:

In [1]:
```
from numpy import *
import sympy as sp
```

Let's start by trying to symbolically integrate f(x). Every symbolic command must be prefaced by "sp":

In [3]:
```
x=sp.symbols('x')
f=sp.exp(-x**2)
sp.integrate(f,(x,0,2))
```

Out[3]:
```
sqrt(pi)*erf(2)/2
```

We got an answer, so what's the point of approximating? The function "erf" is the special name for a function based on "the integral of e^(-x^2)". So Python is basically saying that the integral of e^(-x^2) is a function based on the integral of e^(-x^2). Not very useful.

So to approximate this integral, the general strategy is to calculate dx, create a list of x-values (easily done with the arange command), compute the y-values, add them up, and multiply by dx, which is the same for every rectangle.

In [2]:
```
# Part a: n=4
# Compute dx
dx=(2-0)/4
# Create the list of x-values.  We will call it "x_i" since we used x as a
 symbolic variable earlier.
# Remember the arange(a,b,dx) creates a list from a INCLUSIVE to b EXCLUSIV
E.  Since we want to start at x1 = a + 1*dx and end at xn = b, we adjust th
e arguments accordingly:
x_i=arange(0+dx,2+dx,dx)
print(x_i)
```

```
[0.5 1.  1.5 2. ]
```

In [3]:
```
# Compute the y-values.  For the numerical work, you can use either sp.E**
() or exp() in the numpy package
y_i=exp(-x_i**2)
# Sum the y-values and multiply by dx
R4=sum(y_i)*dx
print('The approximation with 4 right-endpoint rectangles is',R4)
```

```
The approximation with 4 right-endpoint rectangles is 0.6351975438467229
```

In [4]:
```
# Part b: all we have to do is change the n value!
dx=(2-0)/8
x_i=arange(0+dx,2+dx,dx)
y_i=exp(-x_i**2)
R8=sum(y_i)*dx
print('The approximation with 8 right-endpoint rectangles is',R8)
```

```
The approximation with 8 right-endpoint rectangles is 0.7589932461932254
```

In [10]:
```
# Part c: same idea
dx=(2-0)/50
x_i=arange(0+dx,2+dx,dx)
y_i=exp(-x_i**2)
R50=sum(y_i)*dx
print('The approximation with 50 right-endpoint rectangles is',R50)
```

```
The approximation with 50 right-endpoint rectangles is 0.862437937804386
```

Finally, we will illustrate another approximation technique called Simpson's Rule. Simpson's Rule approximates the curve along its subintervals with parabolas. This command is located in the scipy.integrate package. Since this is the only command we need, we will just import it.

Use Simpson's Rule to approximate the integral of 1-x^3 from x=0 to x=1 with n=10 subintervals. (Yes, you can integrate this-you'll see why we chose this below)

In [12]:
```
from scipy.integrate import simps
dx=(1-0)/10
# Our x list now must go from a INCLUSIVE to b INCLUSIVE (so we go one past
it in the arange command)
x_i=arange(0,1+dx,dx)
y_i=1-x_i**3
S10=simps(y_i,x_i) #Notice the arguments: y-values, THEN x-values
print("The approximation with Simpson's Rule for 10 subintervals is",S10)
```

```
The approximation with Simpson's Rule for 10 subintervals is 0.75
```

NOTICE: even though we use parabolas to approximate the curve, the approximation for this cubic (and ANY cubic) is EXACT!

In [ ]:
```
```