Week 7 Lecture 2: Numerical Integration

Some functions for illustration purposes:

```
import numpy as np
import matplotlib.pyplot as plt

# You can ignore these functions.
# They are just for visualizing integration schemes.
def plot_lhr(f, a, b, dx):
    x = np.arange(a, b + dx, dx)
    y = f(x)
    n = x.size

    xplot = np.arange(a, b + dx/100, dx/100)
    yplot = f(xplot)
    plt.plot(xplot, yplot, 'k')

    for k in range(n-1):
        x_rect = [x[k], x[k], x[k + 1], x[k + 1], x[k]]
        y_rect = [0, y[k], y[k], 0, 0]
        plt.plot(x_rect, y_rect, 'b')

def plot_rhr(f, a, b, dx):
    x = np.arange(a, b + dx, dx)
    y = f(x)
    n = x.size

    xplot = np.arange(a, b + dx/100, dx/100)
    yplot = f(xplot)
    plt.plot(xplot, yplot, 'k')

    for k in range(n-1):
```

```
        x_rect = [x[k], x[k], x[k + 1], x[k + 1], x[k]]
        y_rect = [0, y[k + 1], y[k + 1], 0, 0]
        plt.plot(x_rect, y_rect, 'b')

def plot_trap(f, a, b, dx):
    x = np.arange(a, b + dx, dx)
    y = f(x)
    n = x.size

    xplot = np.arange(a, b + dx/100, dx/100)
    yplot = f(xplot)
    plt.plot(xplot, yplot, 'k')

    for k in range(n-1):
        x_rect = [x[k], x[k], x[k + 1], x[k + 1], x[k]]
        y_rect = [0, y[k], y[k + 1], 0, 0]
        plt.plot(x_rect, y_rect, 'b')
```

Numerical Integration

In this lecture, we will discuss several methods for numerical integration. The general setup will be exactly the same as for differentiation. We will either be given a function $f(x)$ and asked to compute

$$\int_a^b f(x)\,\mathrm{d}x, \qquad (1)$$

or we will be given a set of points $(x_0, y_0), (x_1, y_1), \ldots, (x_N, y_N)$, where $y_k = f(x_k)$ for some (usually unknown) function $f(x)$ and then asked to compute

$$\int_a^b f(x)\,\mathrm{d}x.$$

By convention, we will always assume that $a = x_0 < x_1 < \cdots < x_N = b$ and that the points are all evenly spaced, so $x_k - x_{k-1} = \Delta x$ is the same for all k. That is, we will assume that $x_k = a + k\Delta x$ for all $0 \leq k \leq N$. The assumption that the points are in order is really no restriction at all, since we can always re-organize any data we are given. The assumption that these points are evenly spaced is more restrictive. In practice, it is very common to have evenly spaced data (perhaps because you are making measurements at regular times or at locations on a regular grid), but it is not strictly necessary. It turns out that none of the methods we will discuss actually require this even spacing, but the formulas become much messier when Δx is not constant. It is also worth noting that there are $N + 1$ points here, not just N, because we start counting at x_0.

As in the previous two lectures, it is easy to switch back and forth between these two settings. If we are given a function $f(x)$, then we can simply plug in all of our x values and then forget about the function. If we are given a set of (x_k, y_k) points, then we can (at least on paper) pretend that $y_k = f(x_k)$ and just never use a formula for $f(x)$. When working on paper, it is generally much more convenient to think in terms of a function f, but in real world applications we often only have a set of data points, so it is important to be comfortable with both sets of notation.

Remember that the integral in (1) can be thought of as the area under the curve $f(x)$ between $x = a$ and $x = b$. All of our methods will be based upon the following idea: We can chop this area up into many smaller regions, then calculate the area of each smaller region and add them all up. As long as the smaller regions approximately cover the whole area, then we will get a good approximation to the total integral.

It turns out that integration methods are generally much more difficult to analyze than difference schemes, so we will only fully analyze the simplest methods. You have almost certainly seen these simple methods in a basic calculus class.

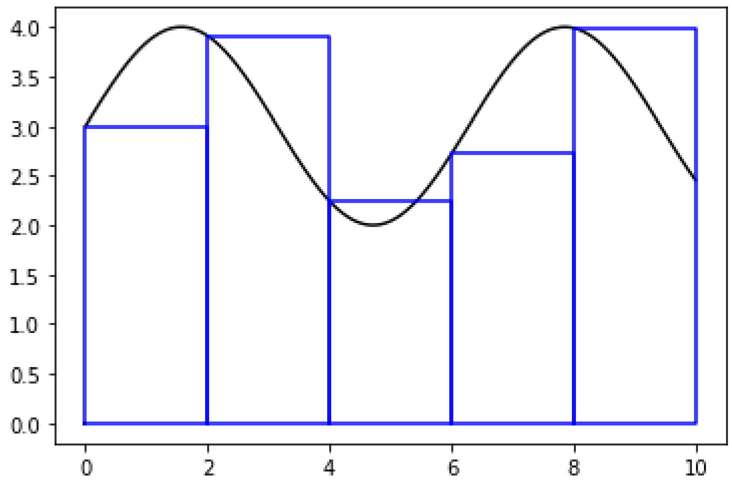
Left Hand Rule

The basic idea in our first approach is to chop the area under the curve $f(x)$ into N different rectangles. The base of the first rectangle will be the interval $[x_0, x_1]$, then the base of the second rectangle will be $[x_1, x_2]$, and so on, all the way to the Nth rectangle, whose base will be $[x_{N-1}, x_N]$. In general, the base of the kth rectangle will be the interval $[x_{k-1}, x_k]$. Because we assumed that the x values were all evenly spaced, the width of each of these intervals is just Δx. We will determine the height by evaluating the function $f(x)$ at some point in the base. The only real choice involved here is which point we use to determine the height.

The first approximation method we will discuss is called the *left hand rule* or *left rectangle rule*. In this method, we choose the height of the kth rectangle to be $f(x_{k-1})$. (The name comes from the fact that x_{k-1} is the leftmost point in the base of the rectangle.)

We can visualize this with the function `plot_lhr`, which is included at the top of the notebook. For example, here are the rectangles for the function $f(x) = \sin(x) + 3$ with $a = 0$, $b = 10$ and $\Delta x = 2$.

```
f = lambda x: np.sin(x) + 3
a = 0
b = 10
dx = 2
plot_lhr(f, a, b, dx)
```

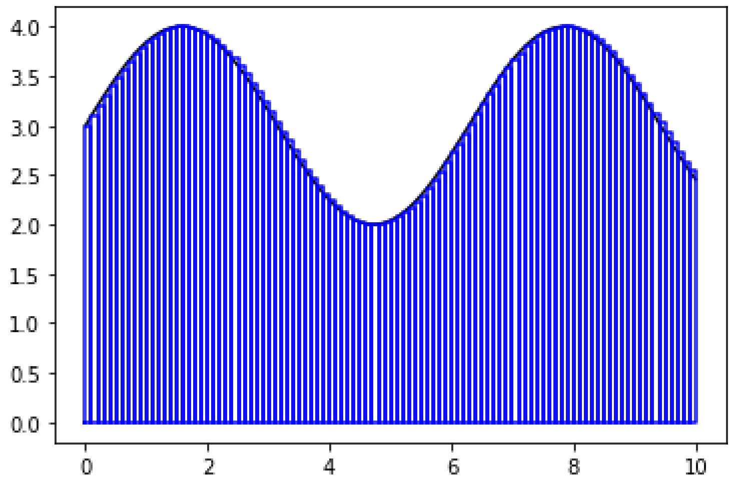


If we plot the same function with a smaller Δx, we get

```
dx = 1
plot_lhr(f, a, b, dx)
```

If we shrink Δx even further, we get

```
dx = 0.1
plot_lhr(f, a, b, dx)
```



The area of the kth rectangle is just the width times the height, or $\Delta x f(x_{k-1})$. To find the area under the whole curve, we just add up the area of all N rectangles, so we get

$$\int_a^b f(x)\,\mathrm{d}x \approx \Delta x f(x_0) + \Delta x f(x_1) + \cdots + \Delta x f(x_{N-1}) = \sum_{k=0}^{N-1} \Delta x f(x_k) = \Delta x \sum_{k=0}^{N-1} f(x_k).$$

How good an approximation is this? More importantly, how does the quality of this approximation change as we reduce Δx? To answer that question, we will first look at a simpler problem and only try to find the integral over one of the rectangles (i.e., the integral from x_{k-1} to x_k). The left hand rule approximation to this integral is

$$\int_{x_{k-1}}^{x_k} f(x)\,\mathrm{d}x \approx \Delta x f(x_{k-1}).$$

We can also approximate this integral by using a Taylor expansion of f. In particular, we will expand $f(x)$ about the point x_{k-1}. We get

$$\int_{x_{k-1}}^{x_k} f(x)\,\mathrm{d}x = \int_{x_{k-1}}^{x_k} f(x_{k-1}) + (x - x_{k-1})f'(x_{k-1}) + \frac{(x - x_{k-1})^2}{2} f''(x_{k-1}) + \mathcal{O}((x - x_{k-1})^3)\,\mathrm{d}x.$$

This integral looks quite complicated, but we are only integrating with respect to x, not x_{k-1}, so it is really just a polynomial. We therefore get

$$\int_{x_{k-1}}^{x_k} f(x)\,\mathrm{d}x = \left[x f(x_{k-1}) + \frac{(x - x_{k-1})^2}{2} f'(x_{k-1}) + \frac{(x - x_{k-1})^3}{6} f''(x_{k-1}) + \mathcal{O}((x - x_{k-1})^4)\right]_{x=x_{k-1}}^{x_k}.$$

If we plug in the endpoints $x = x_{k-1}$ and $x = x_k$ into this antiderivative, then many of the terms become zero and we are left with

$$\int_{x_{k-1}}^{x_k} f(x)\,\mathrm{d}x = (x_k - x_{k-1})f(x_{k-1}) + \frac{(x_k - x_{k-1})^2}{2} f'(x_{k-1}) + \frac{(x_k - x_{k-1})^3}{6} f''(x_{k-1}) + \mathcal{O}((x_k - x_{k-1})^4)$$

$$= \Delta x f(x_{k-1}) + \tfrac{(\Delta x)^2}{2} f'(x_{k-1}) + \tfrac{(\Delta x)^3}{6} f''(x_{k-1}) + \mathcal{O}(\Delta x^4).$$

Since Δx is supposed to be very small, higher powers of Δx will be even smaller, so we can safely ignore the $\mathcal{O}(\Delta x^3)$ terms. This means that we have

$$\int_{x_{k-1}}^{x_k} f(x)\,\mathrm{d}x \approx \Delta x f(x_{k-1}) + \frac{(\Delta x)^2}{2} f'(x_{k-1}).$$

Notice that the first term matches our approximation from the left hand rule. This means that the left hand rule gives us a good approximation of the integral over this small interval and that the error from the left hand rule is $\frac{(\Delta x)^2}{2} f'(x_{k-1})$. In particular, this error has a Δx^2 in it, so we say that it is second order. We call this quantity the *local error* for the left hand rule, and we say that the left hand rule has second order local error.

However, it is important to remember that this isn't really the problem we wanted to solve. We just found a second order approximation to the integral of $f(x)$ from x_{k-1} to x_k, but we wanted to approximate the integral from a to b. The total integral involves N of these small intervals, so the total error will be roughly $N\frac{(\Delta x)^2}{2} f'(x_{k-1})$. You might be tempted to think that N is simply a constant, so we can ignore it when we use big-oh notation and just say that the total error is $\mathcal{O}(\Delta x^2)$, but this would be incorrect. The problem is that N actually depends on Δx. If we make Δx smaller, then we need more rectangles to cover the whole area. In particular, we have $N = (b - a)/\Delta x$. This means that the error for the entire integral is $\frac{(b-a)\Delta x}{2} f'(x_{k-1})$, which only has a Δx^1. We call this quantity the *global error* of the left hand rule and we say that the left hand rule has first order global error.

It is very easy to implement the left hand rule in python. As an example, let's try to find the integral of $f(x) = \sin(x) + 3$ between $a = 0$ and $b = 10$. The antiderivative of this function is $F(x) = -\cos(x) + 3x$, and so the actual solution is

$$\int_a^b \sin(x) + 3 \,\mathrm{d}x = (-\cos(b) + 3b) - (-\cos(a) + 3a).$$

```
print((-np.cos(b) + 3 * b) - (-np.cos(a) + 3 * a))
```

```
31.839071529076453
```

To set up our problem, we need to choose a list of x values and calculate the corresponding y values. For example, if we choose $\Delta x = 1$ then we have

```
dx = 1
x = np.arange(a, b + dx, dx)
y = np.sin(x) + 3
```

To calculate the left hand rule, we have to add up $(\Delta x)y_k$ for every y value except the last one. One approach (which we have used several times in this class already) is to add these terms one at a time in a loop. For example,

```
LHR = 0
for k in range(x.size - 1):
    LHR = LHR + dx * y[k]
print(LHR)
```

```
31.95520948210738
```

Another option is to collect all of the necessary y values and then use the numpy function `sum` to add them all up. Since we want to add the first through the second to last entries of y, we can use

```
LHR = dx * np.sum(y[:-1])
print(LHR)
```

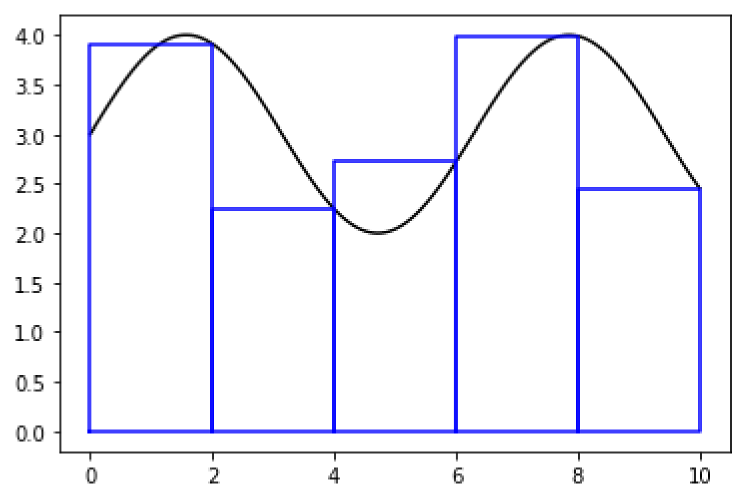
```
31.955209482107378
```

Right Hand Rule

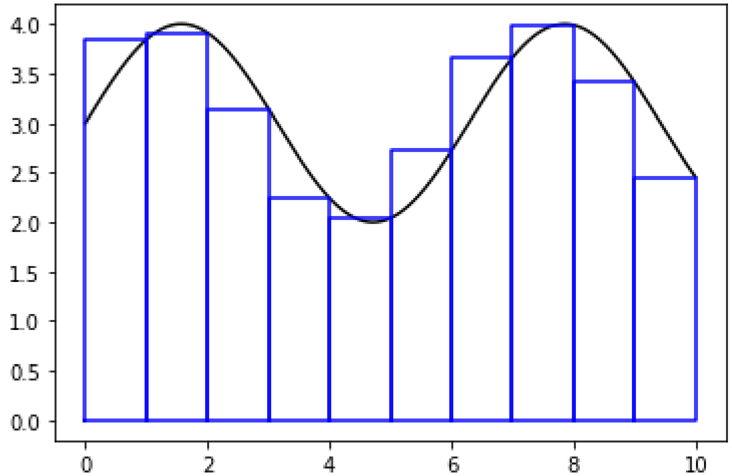
There was no particular reason to choose the left endpoint of each interval in the previous approximation. We could just as easily have chosen the right endpoint (or any point in between, for that matter, but we will stick to the endpoints for now).

Here are some illustrations of the right hand rule for different choices of Δx:

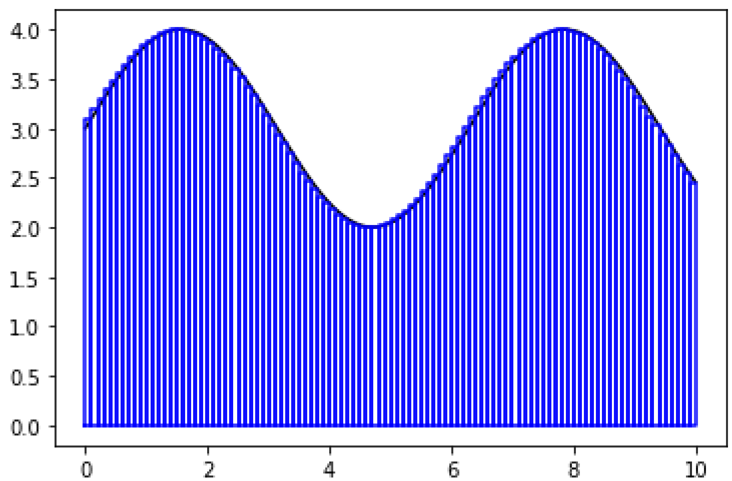
```
dx = 2
plot_rhr(f, a, b, dx)
```




```
dx = 1
plot_rhr(f, a, b, dx)
```




```
dx = 0.1
plot_rhr(f, a, b, dx)
```



The area of the kth rectangle is now $\Delta x f(x_k)$, and so we have the approximation

$$\int_a^b f(x)\,\mathrm{d}x \approx \Delta x f(x_1) + \Delta x f(x_2) + \cdots + \Delta x f(x_N) = \sum_{k=1}^{N} f(x_k)\Delta x = \Delta x \sum_{k=1}^{N} f(x_k).$$

We call this method the *right hand rule* or *right rectangle rule*. Notice that the formula is almost exactly the same as for the left hand rule, but the sum starts at 1 instead of 0 and goes up to N instead of $N-1$. That is, we use every $f(x_k)$ value except for $f(x_0)$, whereas in the left hand rule we used every value except for $f(x_N)$.

We can calculate the error in exactly the same way as we did for the left hand rule (except that we would need to Taylor expand about x_k instead of x_{k-1}). If we do so, we will find that the local error is approximately $-\frac{(\Delta x)^2}{2} f'(x_k)$, and so the right hand rule has second order local error. Likewise, we would find that the global error is approximately $-\frac{(b-a)\Delta x}{2} f'(x_k)$, and so the right hand rule has first order global error.

Like the left hand rule, it is easy to implement this method in python. Let's try the same integral as above:

$$\int_0^{10} \sin(x) + 3 \,\mathrm{d}x = \Big[-\cos(x) + 3x\Big]_{x=0}^{10}.$$

```
dx = 1
a = 0
b = 10
print((-np.cos(b) + 3 * b) - (-np.cos(a) + 3 * a))
```

```
31.839071529076453
```

To calculate the right hand rule, we have to add up $(\Delta x)y_k$ for every y value except the first one. As before, we can either do this with a loop or the built-in `sum` function.

```
RHR = 0
for k in range(1, x.size):
    RHR = RHR + dx * y[k]
print(RHR)
```

```
31.41118837121801
```

```
RHR = dx * np.sum(y[1:])
print(RHR)
```

```
31.41118837121801
```

Trapezoidal Rule

It is worth noting that the leading error terms for the left and right hand rules look very similar, but with opposite signs. This suggests an improvement very similar to our approach with the central difference scheme. We can calculate both the right and left hand approximations and then average the two. We would then obtain the formula

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{1}{2}\left[\sum_{k=0}^{N-1} \Delta x\, f(x_k) + \sum_{k=1}^{N} f(x_k)\Delta x\right].$$

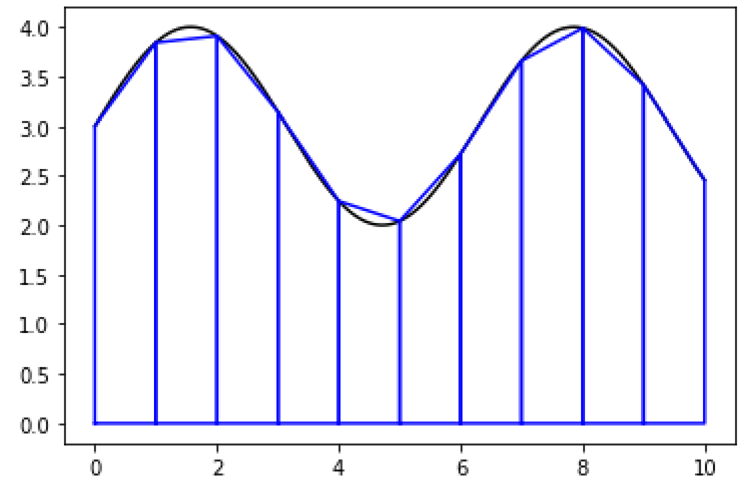
We can actually simplify this even more by noting that most of the terms appear twice, once in each sum. The only two exceptions are the $k = 0$ and $k = N$ terms, which only appear once. We can therefore rewrite this approximation as

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{\Delta x}{2}\left[f(x_0) + 2\sum_{k=1}^{N-1} f(x_k) + f(x_N)\right].$$

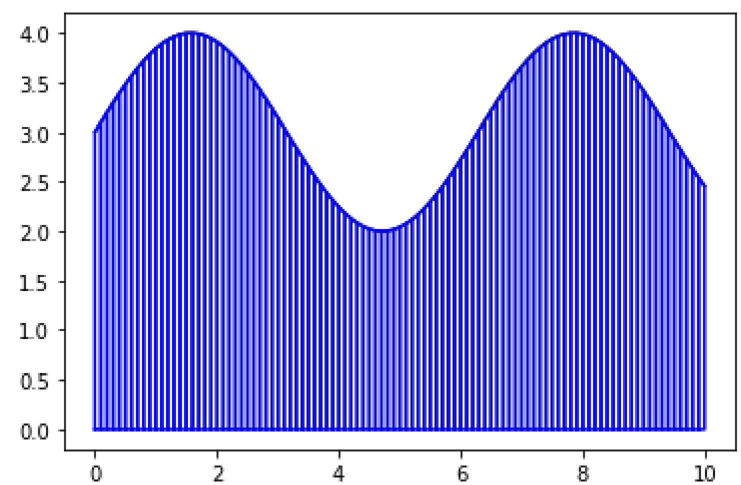
This approximation is called the *trapezoidal rule*. The name comes from the fact that we can also derive this method by dividing the total area up into many small trapezoids instead of rectangles. For example:

```
dx = 2
plot_trap(f, a, b, dx)
```


```
dx = 1
plot_trap(f, a, b, dx)
```




```
dx = 0.1
plot_trap(f, a, b, dx)
```



The analysis for this method is more complicated than before, but it turns out that the trapezoidal rule has third order local error and second order global error.

As an example, let's use the trapezoidal method to approximate the same integral as before:

```
dx = 1
a = 0
b = 10
print((-np.cos(b) + 3 * b) - (-np.cos(a) + 3 * a))
```

```
31.839071529076453
```

We can either find the left and right hand rules first and then average them

```
LHR = dx * np.sum(y[:-1])
RHR = dx * np.sum(y[1:])
trap = (LHR + RHR) / 2
print(trap)
```

```
31.683198926662694
```

or we can use the formula directly:

```
trap = (dx / 2) * (y[0] + 2 * np.sum(y[1:-1]) + y[-1])
print(trap)
```

```
31.683198926662694
```

Builtin Functions

Python has several useful builtin functions for approximating integrals. First, there is the `trapz` function from the numpy package, which implements the trapezoidal rule. The easiest way to use this command is to give it a vector of y values and a vector of x values corresponding to the (x_k, y_k) points from our data set. For example, if we were solving the same problem as above, then we could just write

```
trap = np.trapz(y, x)
print(trap)
```

```
31.683198926662694
```

It is important to keep in mind that `y` and `x` are arrays, not functions. Python figures out the spacing Δx by looking at the array `x`.

Another important built-in function called `quad` can be found in the `integrate` sub-package of `scipy`. The general syntax looks like this:

```
Int, err = scipy.integrate.quad(f, a, b)
```

where `f` is the function (not an array) you want to integrate and `a` and `b` are the bounds of integration. The first return value is the integral and the second is an estimate of the error between this approximation and the true value.

The `quad` function uses substantially more sophisticated algorithms than we have covered here. It is worth noting that this function does not use the same method as the MATLAB function `quad`. Instead, `scipy.integrate.quad` is the equivalent of the MATLAB function `integral`.

```
import scipy.integrate

Int, err = scipy.integrate.quad(f, a, b)
print(Int)
```

```
31.839071529076453
```