07 – Integration (Part 1)

Phys 281 – Class 6

Grant Wilson

Exercises

```
#E6.1 - various interpolations and their error plots
#fetch the data
data_file = 'interpolate_me.npz'
npz = np.load(data_file)
x1 = npz['x1']
y1 = npz['y1']
```

Fetch the support points

```
#the x values of the interpolation
x = np.linspace(1,9,100)
```

create the x-values of the interpolated data

```
#the interpolation functions
linear = interpolate.interp1d(x1,y1,'linear')
quad = interpolate.interp1d(x1,y1,'quadratic')
cubic = interpolate.interp1d(x1,y1,'cubic')
```

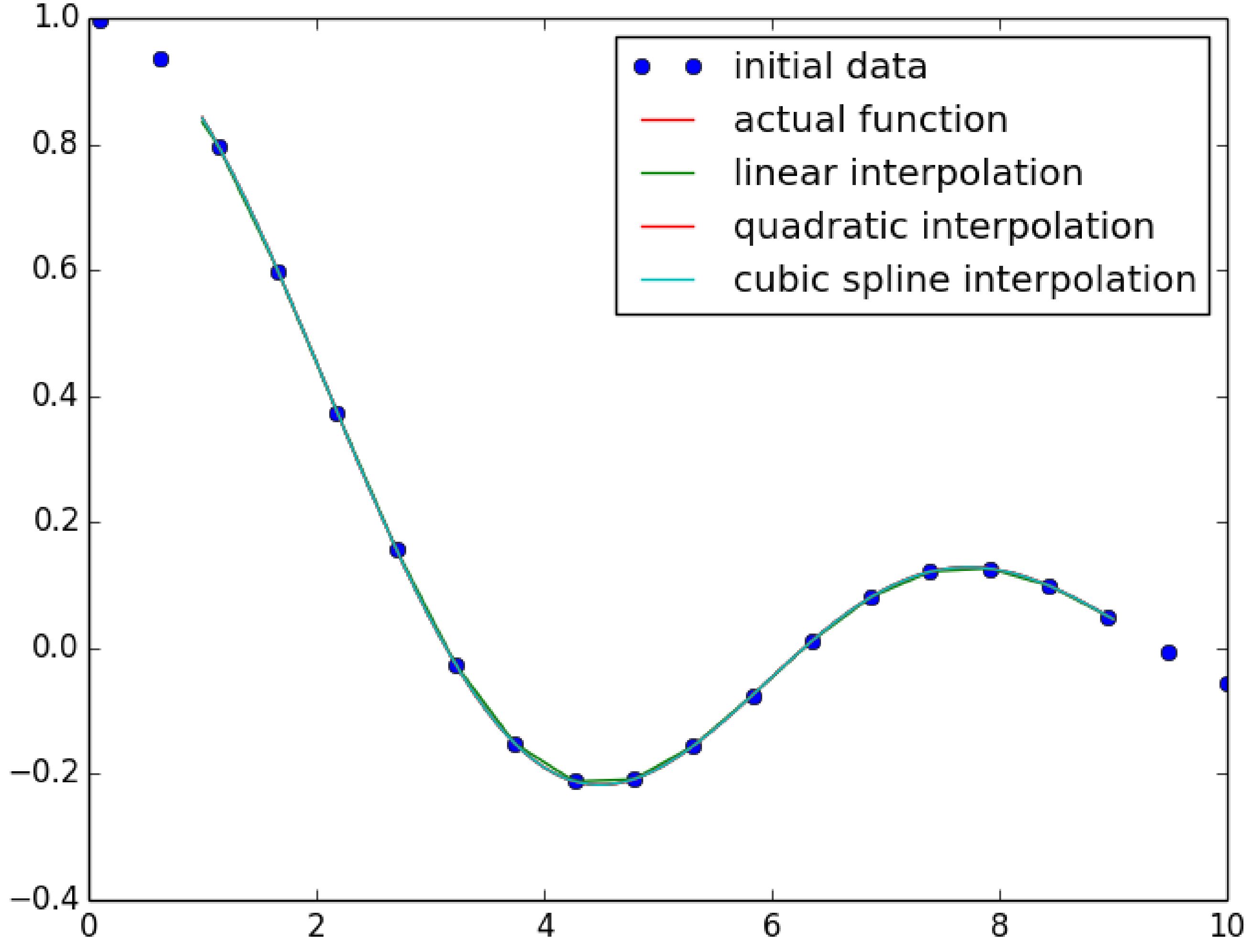
create the interpolation functions

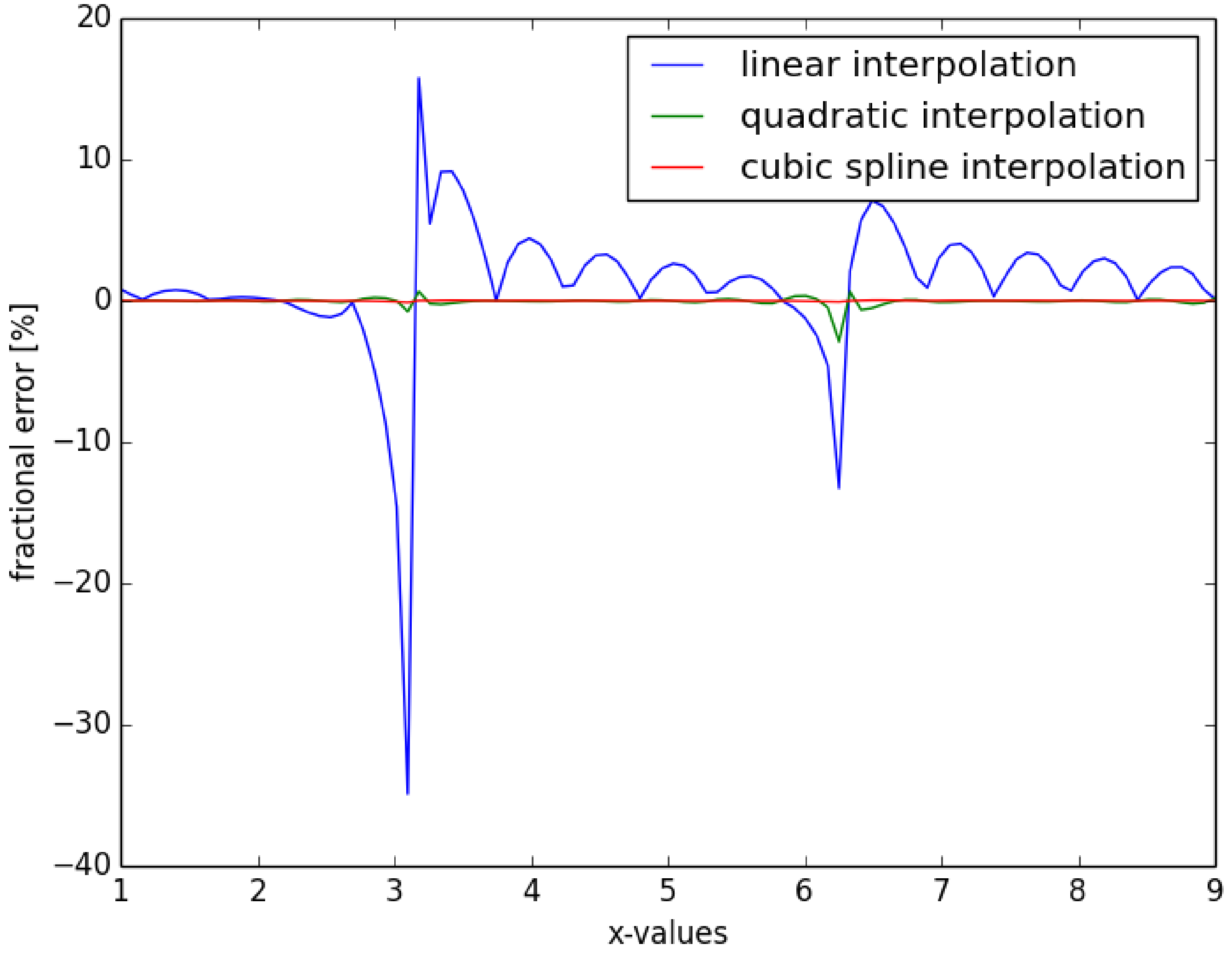
```
#the interpolated y values to go with the x-values
ylin = linear(x)
yquad = quad(x)
ycubic = cubic(x)
```

actually do the interpolating

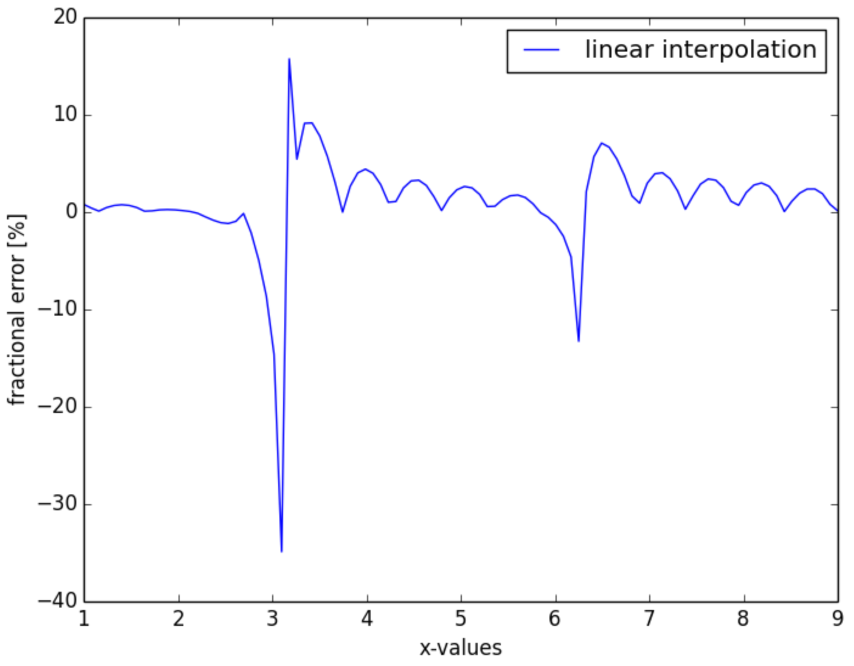
```
#The actual values of y that go with the x-values
y = np.sin(x)/x

#The fractional errors
ylin_err = (y - ylin)/y
yquad_err = (y-yquad)/y
ycubic_err = (y-ycubic)/y
```

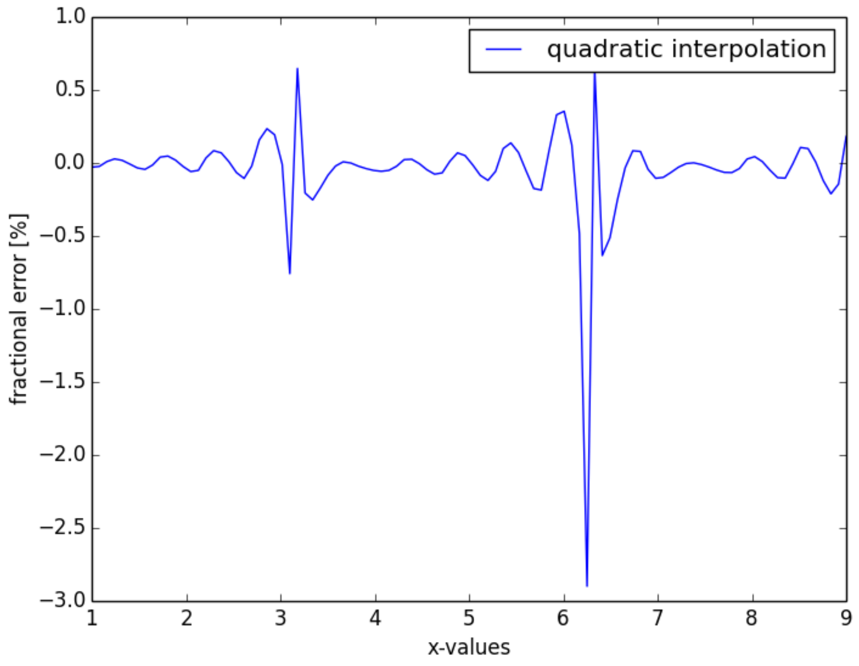
since we know the underlying function, calculate it and the fractional errors

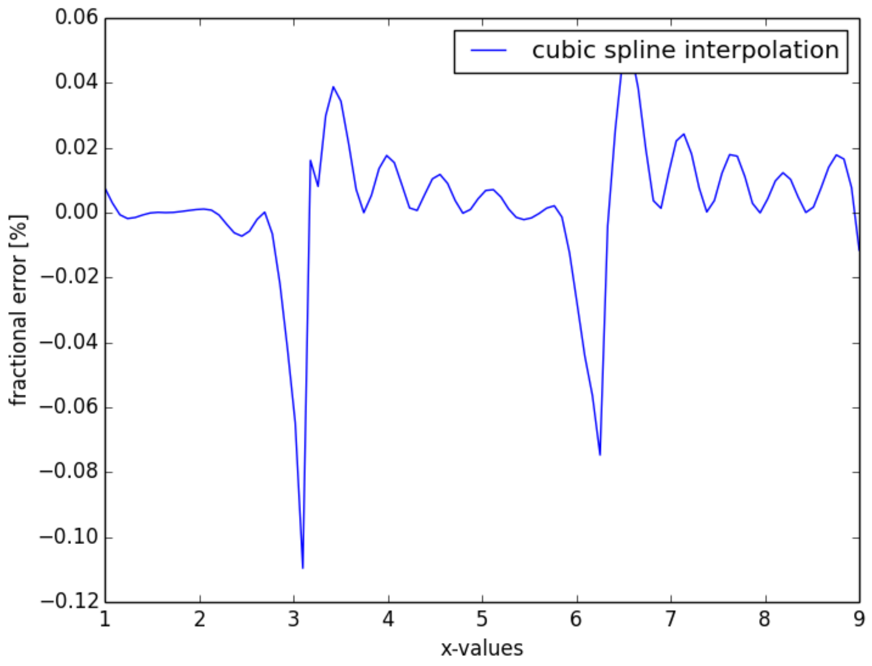

1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0
2
4
6
8
10
initial data
actual function
linear interpolation
quadratic interpolation
cubic spline interpolation



What’s wrong with this plot?


linear interpolation
fractional error [%]
x-values
20
10
0
−10
−20
−30
−40
1
2
3
4
5
6
7
8
9


quadratic interpolation
fractional error [%]
x-values
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
1
2
3
4
5
6
7
8
9


cubic spline interpolation
fractional error [%]
x-values
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06
−0.08
−0.10
−0.12
1
2
3
4
5
6
7
8
9

E6.2

- Write your own function to linearly interpolate a set of data. You may not use the Python interpolation code for this - you need to write it from scratch. However, you should use interp1d to check that your function does the right thing.

E6.2

Always start a new coding problem by grabbing a piece of paper and writing down your strategy.

E6.2

Always start a new coding problem by grabbing a piece of paper and writing down your strategy.

#strategy:

1. for given x, find bracketing support points
2. build linear function between bracketing support points
3. evaluate function at x and return the result

E6.2

Always start a new coding problem by grabbing a piece of paper and writing down your strategy.

#strategy:

1. check that the input x is between the min and max of the x-values of the support points
2. for given x, find bracketing support points
3. build linear function between bracketing support points
4. evaluate function at x and return the result

E6.2

```
def my_painful_linear_interpolator(x1,y1,x):
    #check the input x value to guard against extrapolation
    if((x < x1.min()) | (x > x1.max())):
        print 'requested x is outside min and max of x1'
        return

    #find the nearest support points
    min_dx = x1[0]-x
    max_dx = x1[len(x1)-1] - x
    ilow = -1
    ihigh = len(x1)+1
    for i in range(len(x1)):
        if(x1[i]-x < 0):
            if(x1[i]-x > min_dx):
                min_dx = x1[i]-x
                ilow = i
        if(x1[i]-x > 0):
            if(x1[i]-x < max_dx):
                max_dx = x1[i]-x
                ihigh = i
        if(x1[i]-x == 0):
            return y1[i]

    #print out the x1 values to make sure we bracketed the x value
    #print x1[ilow:ihigh+1], '; x=', x

    #calculate the answer
    ans = y1[ilow]+ (y1[ihigh]-y1[ilow])/(x1[ihigh]-x1[ilow]) * (x-x1[ilow])
    return ans
```

E6.2

```
def my_painful_linear_interpolator(x1,y1,x):
   #check the input x value to guard against extrapolation
   if((x < x1.min()) | (x > x1.max())):
      print 'requested x is outside min and max of x1'
      return

   #find the nearest support points
   min_dx = x1[0]-x
   max_dx = x1[len(x1)-1] - x
   ilow = -1
   ihigh = len(x1)+1
   for i in range(len(x1)):
      if(x1[i]-x < 0):
         if(x1[i]-x > min_dx):
            min_dx = x1[i]-x
            ilow = i
      if(x1[i]-x > 0):
         if(x1[i]-x < max_dx):
            max_dx = x1[i]-x
            ihigh = i
      if(x1[i]-x == 0):
         return y1[i]

   #print out the x1 values to make sure we bracketed the x value
   #print x1[ilow:ihigh+1], '; x=', x

   #calculate the answer
   ans = y1[ilow]+ (y1[ihigh]-y1[ilow])/(x1[ihigh]-x1[ilow]) * (x-x1[ilow])
   return ans
```

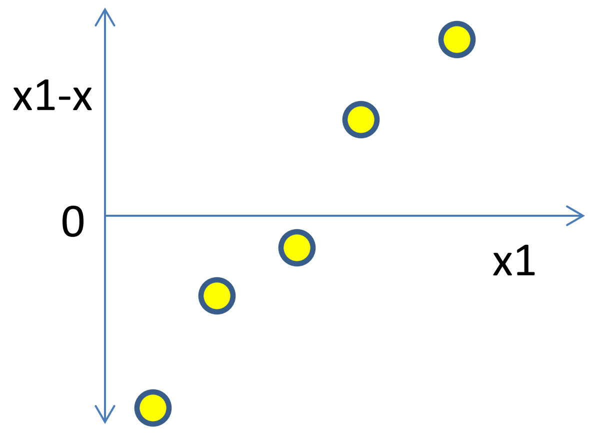
check the input to guard against extrapolation

E6.2

```
def my_painful_linear_interpolator(x1,y1,x):
    #check the input x value to guard against extrapolation
    if((x < x1.min()) | (x > x1.max())):
        print 'requested x is outside min and max of x1'
        return

    #find the nearest support points
    min_dx = x1[0]-x
    max_dx = x1[len(x1)-1] - x
    ilow = -1
    ihigh = len(x1)+1
    for i in range(len(x1)):
        if(x1[i]-x < 0):
            if(x1[i]-x > min_dx):
                min_dx = x1[i]-x
                ilow = i
        if(x1[i]-x > 0):
            if(x1[i]-x < max_dx):
                max_dx = x1[i]-x
                ihigh = i
        if(x1[i]-x == 0):
            return y1[i]

    #print out the x1 values to make sure we bracketed the x value
    #print x1[ilow:ihigh+1], '; x=', x

    #calculate the answer
    ans = y1[ilow]+ (y1[ihigh]-y1[ilow])/(x1[ihigh]-x1[ilow]) * (x-x1[ilow])
    return ans
```

check the input to guard against extrapolation

find the bracketing support points



E6.2

```
#a more concise linear interpolator – x1 must be monotonically increasing
def my_lin_interp(x1,y1,x):
        if((x < x1.min()) | (x > x1.max())):
                print 'requested x is outside min and max of x1'
                return
        ilow = np.max(np.where(x1-x < 0))
        ihigh = np.min(np.where(x1-x >= 0))
        ans = y1[ilow]+ (y1[ihigh]-y1[ilow])/(x1[ihigh]-x1[ilow]) * (x-x1[ilow])
        return ans
```

E6.2

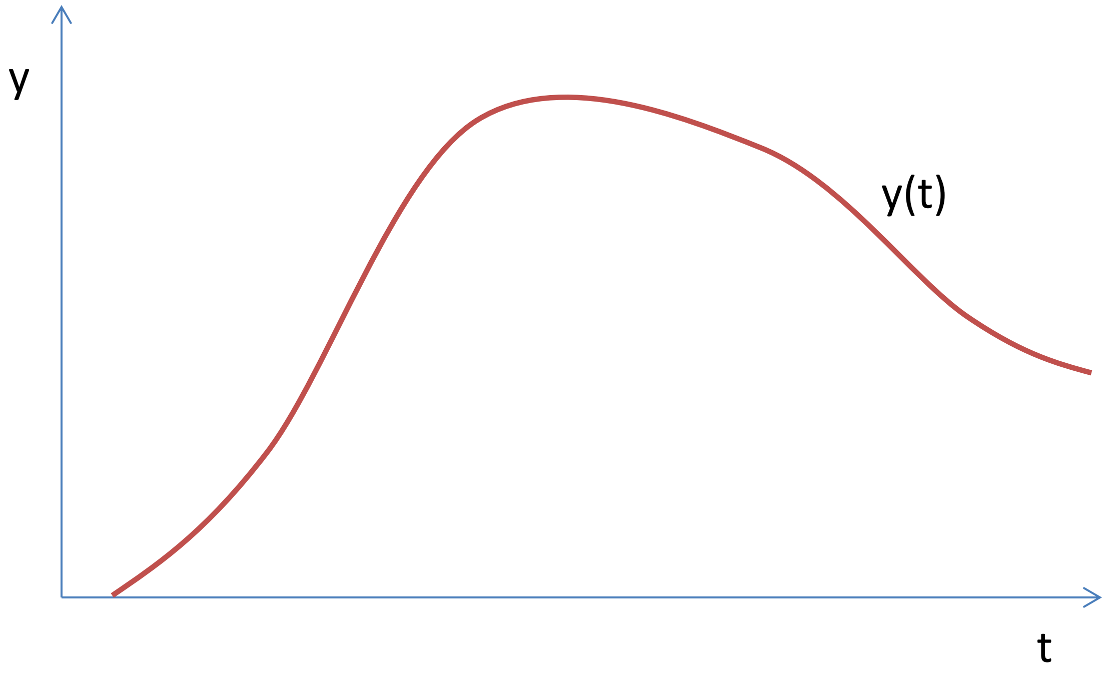
```
#my own linear interpolator – x1 must be monotonically increasing
def my_lin_interp(x1,y1,x):
        if((x < x1.min()) | (x > x1.max())):
                print 'requested x is outside min and max of x1'
                return
        ilow = np.max(np.where(x1-x < 0))
        ihigh = np.min(np.where(x1-x >= 0))
        ans = y1[ilow]+ (y1[ihigh]-y1[ilow])/(x1[ihigh]-x1[ilow]) * (x-x1[ilow])
        return ans
```

07 – Integration (Part 1)

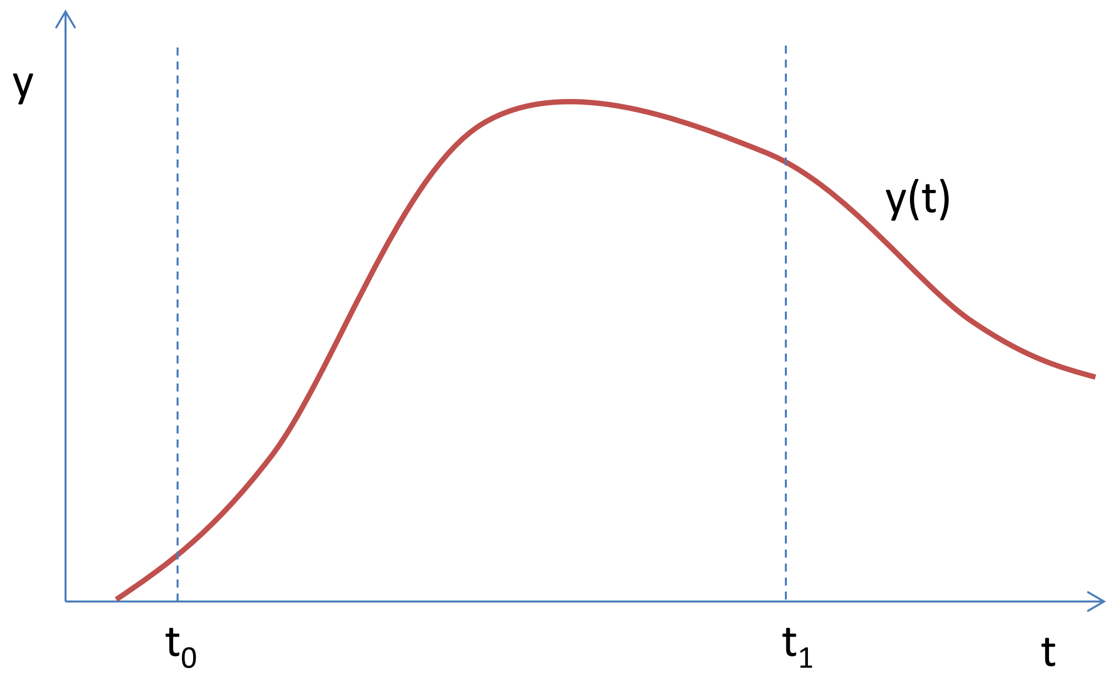
Phys 281 – Class 6

Grant Wilson

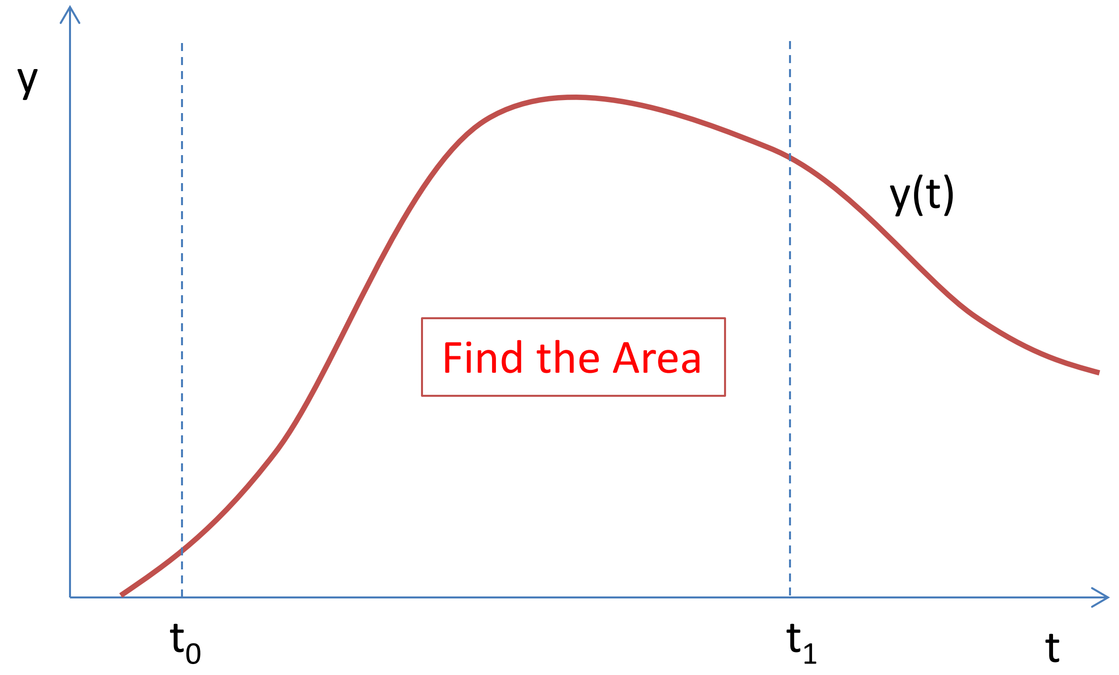
Integration in 1-d



Integration in 1-d



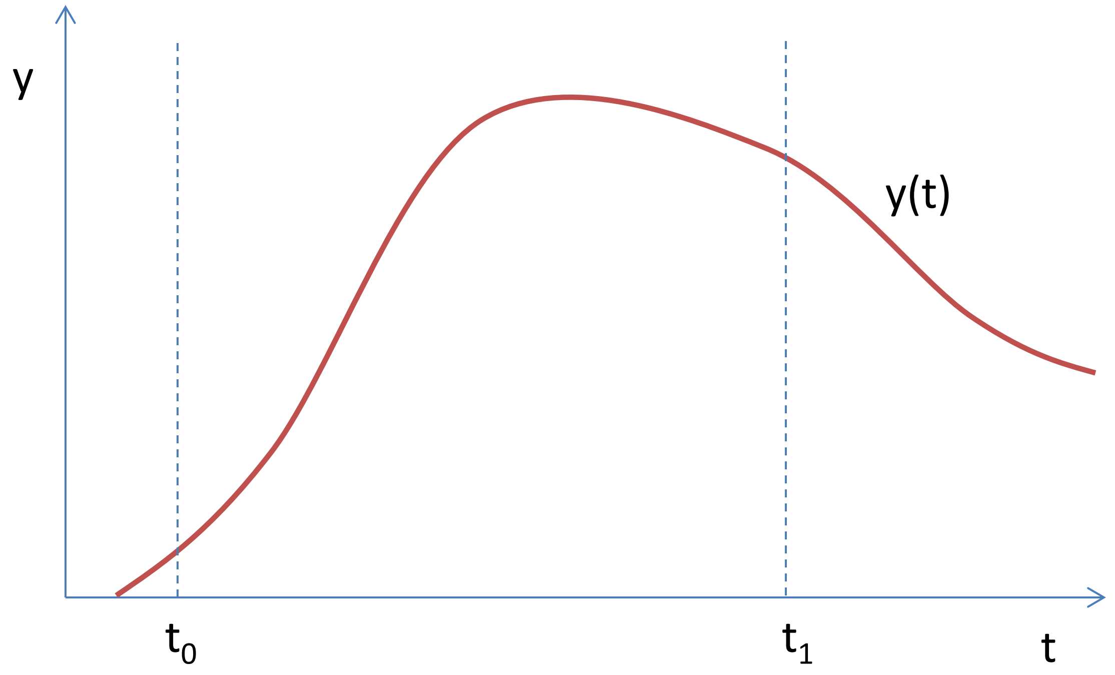
Integration in 1-d



Integration and Interpolation

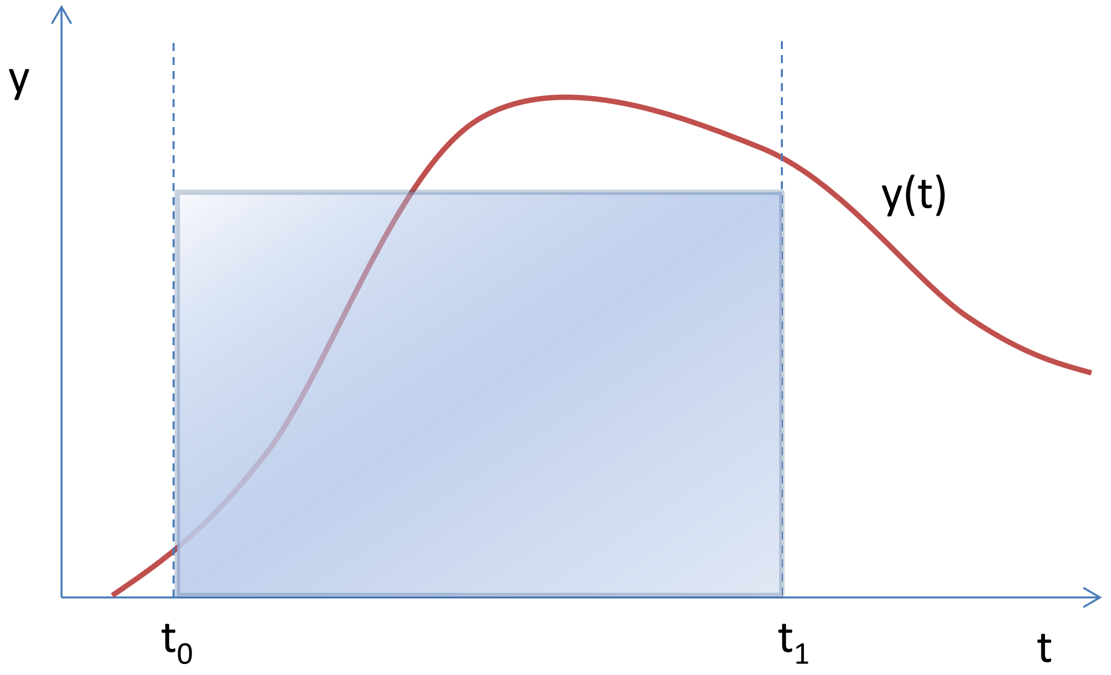
See if you can spot the connection.

Integration in 1-d



Give me the **simplest** strategy that you can possibly think of.

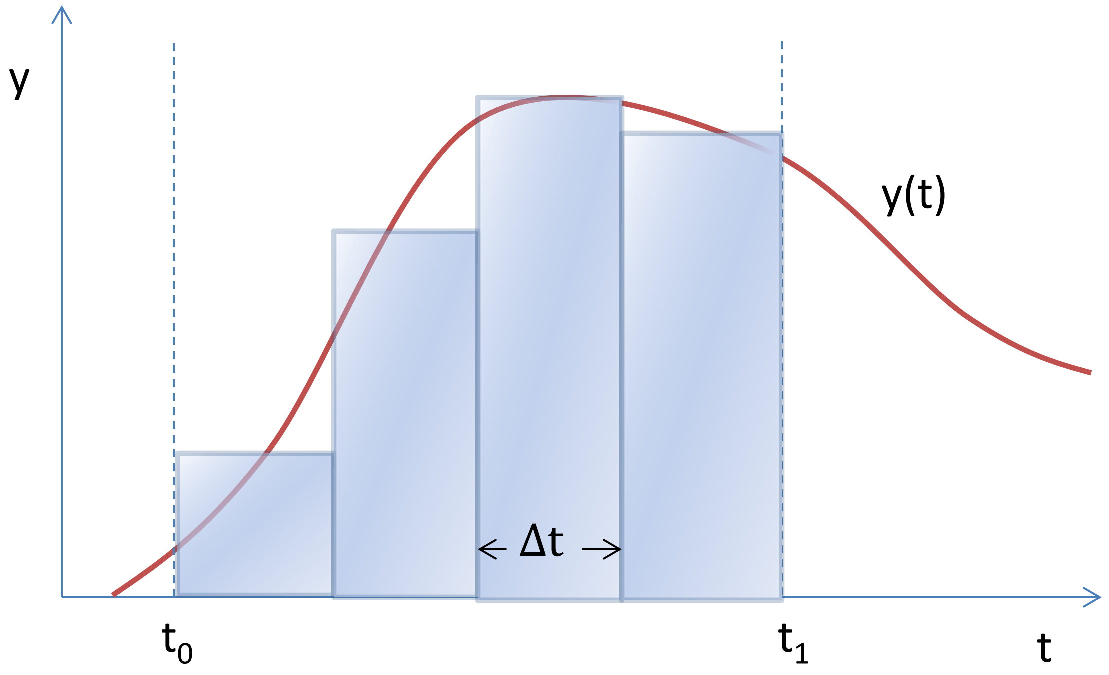
Integration in 1-d



Build a box with width = (t_1-t_0) and height of average(y(t))

$$I = \int_{t_0}^{t_1} y(t)dt \approx \overline{y(t)}\,(t_1 - t_0)$$

Integration in 1-d



Build smaller boxes!

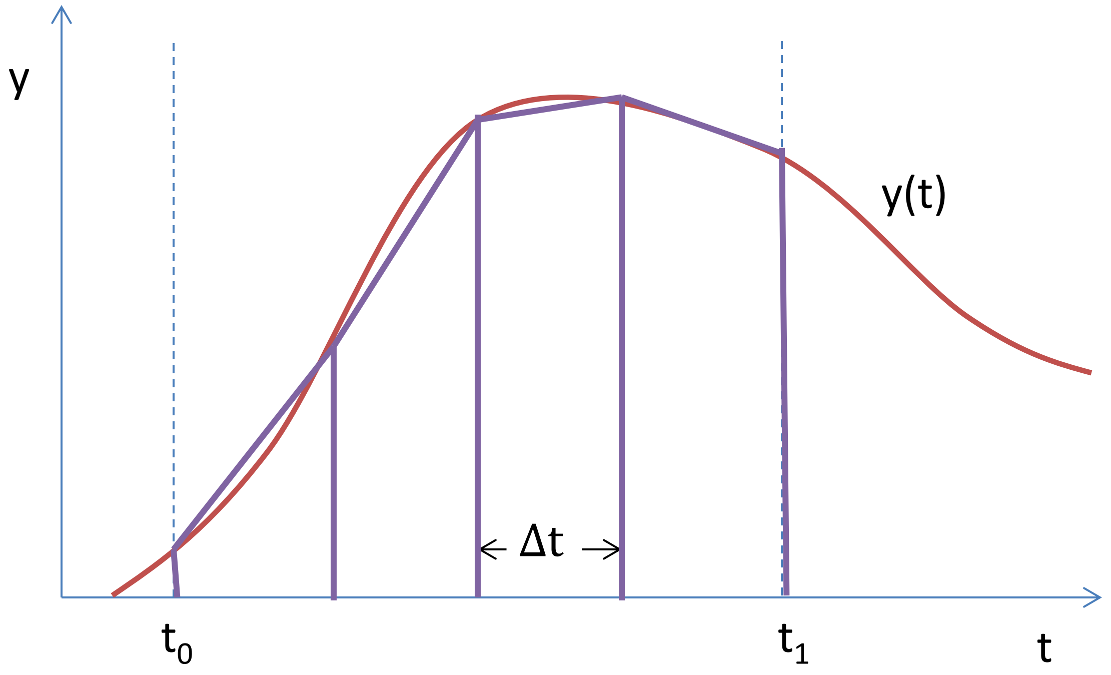
$$I = \int_{t_0}^{t_1} y(t)dt \approx \sum_{i=0}^{N-1} y(i\Delta t)\Delta t$$

- Notice that:
  1. The smaller the boxes, the more accurate the integration estimate.

  2. The smaller the boxes, the more function evaluations we need to do.

By the way, do you see the connection to iteration yet?

Integration in 1-d



Try trapezoids:

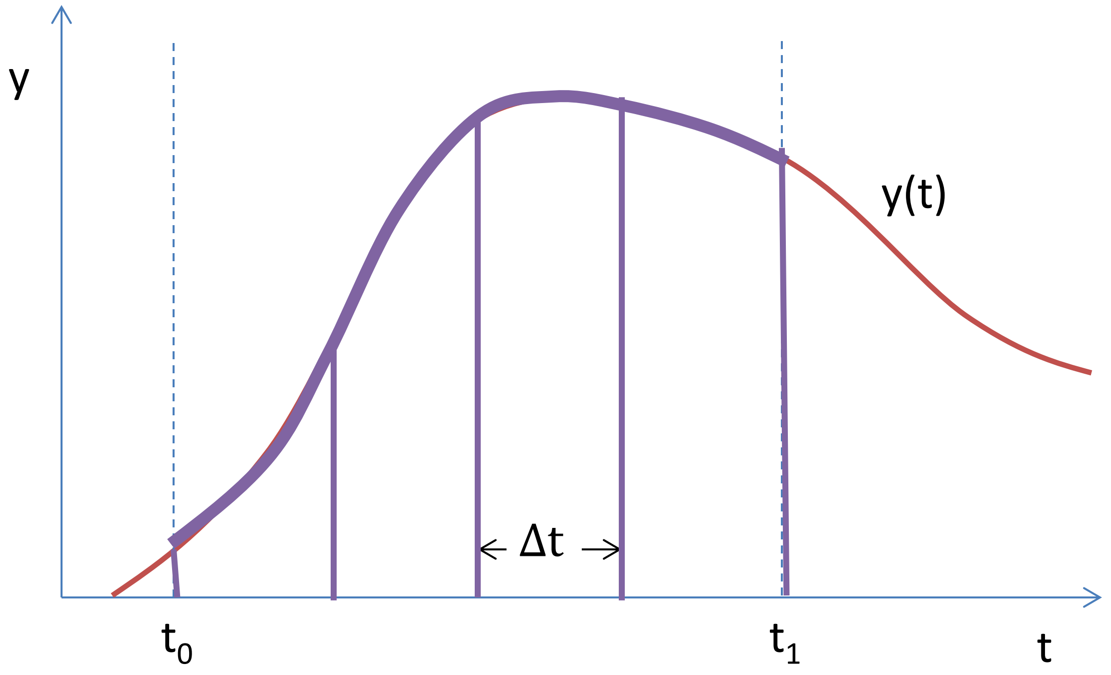
$$I = \int_{t_0}^{t_1} y(t)dt \approx \sum_{i=1}^{N-1} \frac{y(i\Delta t) + y((i-1)\Delta t)}{2} \Delta \mathrm{t}$$

- Notice that:
  1. Again, the more trapezoids we use, the more accurate the integration estimate.

  2. The complication of adding an extra term for the trapezoids is small compared to the gain in the accuracy.

By the way, do you see the connection to iteration yet?

Integration in 1-d

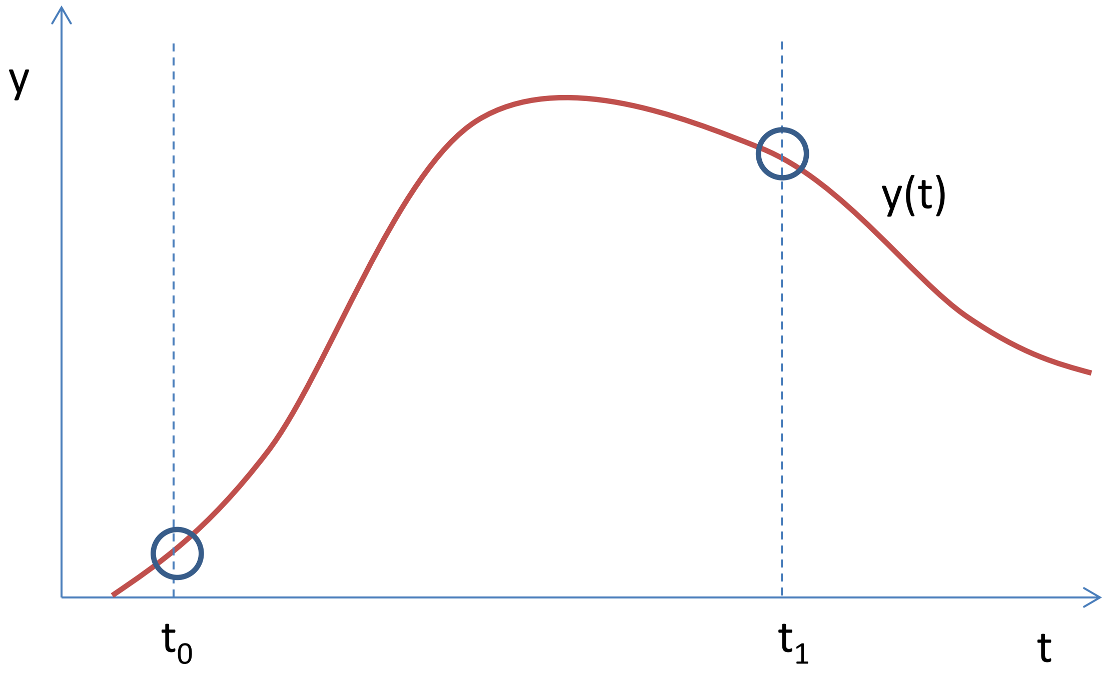


Why stop at trapezoids, use polynomial interpolation!

- Notice that:
  1. Simpson’s rule … interpolate with a quadratic and then integrate the quadratic.

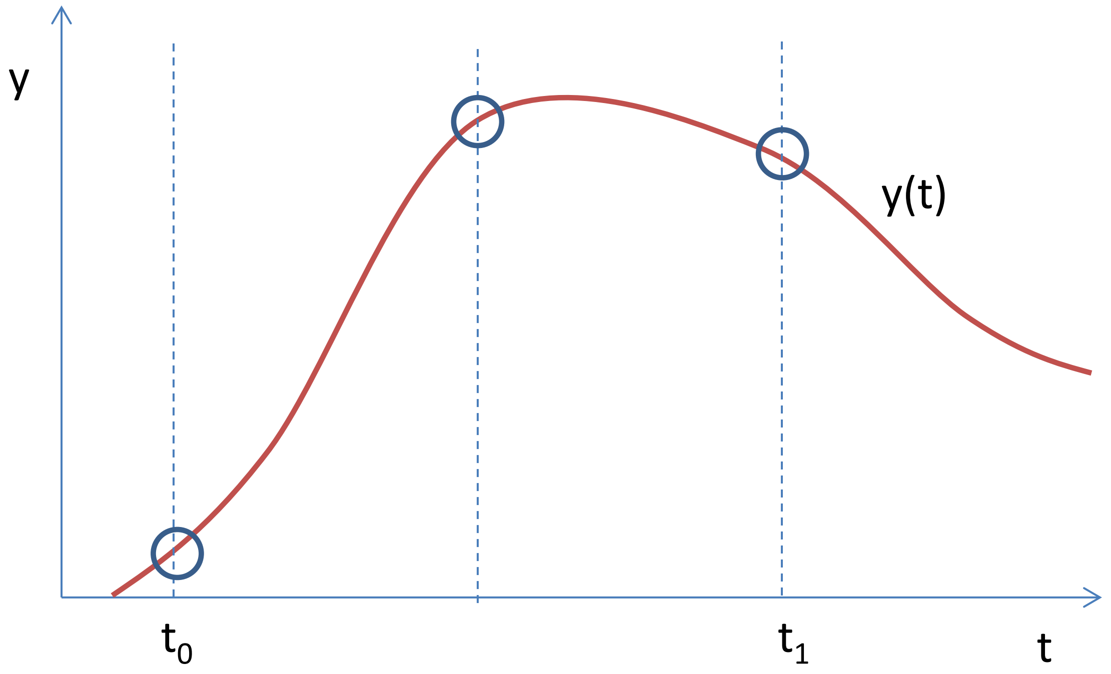
  2. There are two modifications to the approach we can make to improve things even more:
     1. Richardson’s extrapolation – make successively better approximations to I without wasting previous function evaluations.
     2. Use the pattern in the error terms to cancel higher order errors with each new approximation.

Richardson Extrapolation



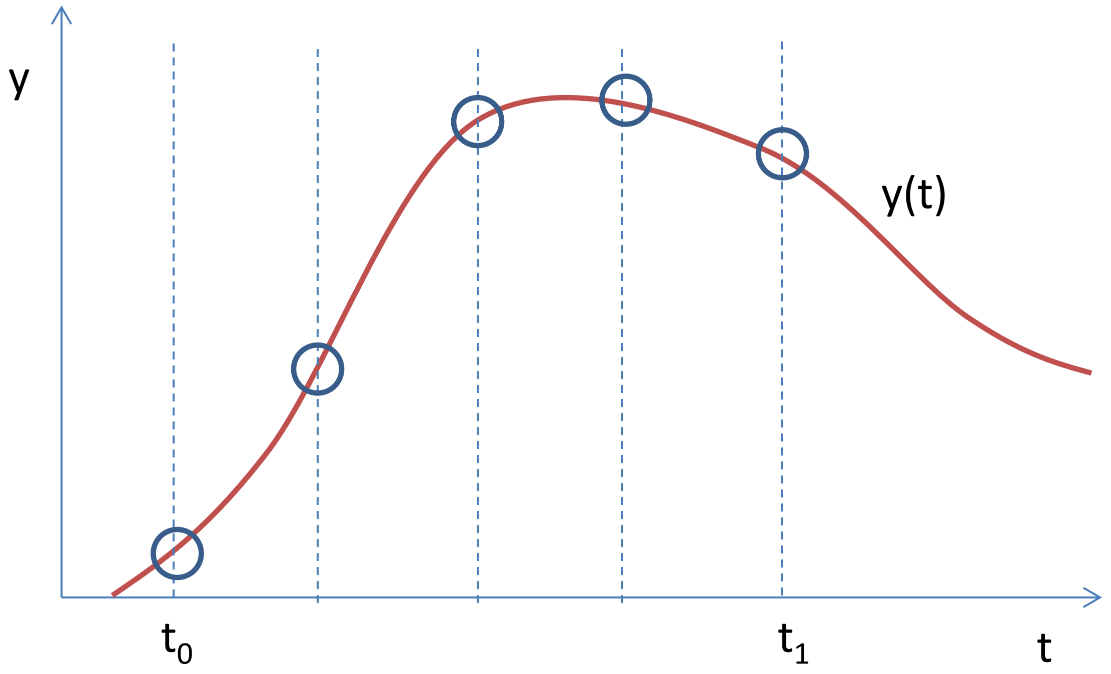
$$I_1 = (t_1 - t_0)\left[\frac{1}{2}y(t_0) + \frac{1}{2}y(t_1)\right]$$

Richardson Extrapolation



$$I_2 = \frac{(t_1 - t_0)}{2}\left[\frac{1}{2}y(t_0) + y\left(\frac{(t_1 + t_0)}{2}\right) + \frac{1}{2}y(t_1)\right]$$

Richardson Extrapolation



$$I_3 = \frac{(t_1 - t_0)}{3}\left[\frac{1}{2}y(t_0) + y\left(\frac{(3t_0 + t_1)}{4}\right) + y\left(\frac{(t_1 + t_0)}{2}\right) + y\left(\frac{(t_0 + 3t_1)}{4}\right) + \frac{1}{2}y(t_1)\right]$$

Richardson Extrapolation Strategy

- Evaluate I_1
- Evaluate I_2
- Evaluate I_3
- …
- stop when

$$\left|(I_{j+1} - I_j)/I_j\right| < your\ tolerance$$

Romberg integration carries this one step further by exploiting a pattern in the errors of the integration to cancel higher and higher order error terms.

Romberg is simple, straightforward, and almost always converges. It is a go-to method!
see scipy.integrate.romberg()

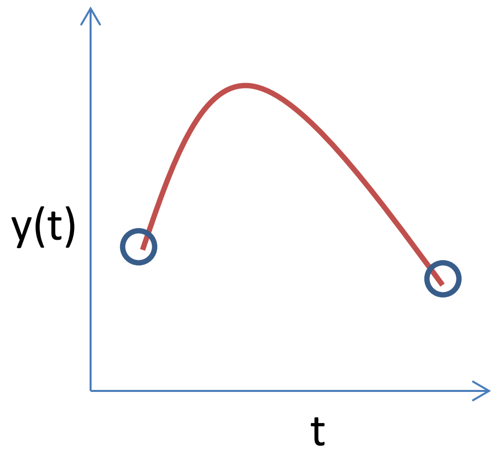
Gaussian Quadrature

- Gaussian Quadrature is a technique you can use to minimize function evaluations.

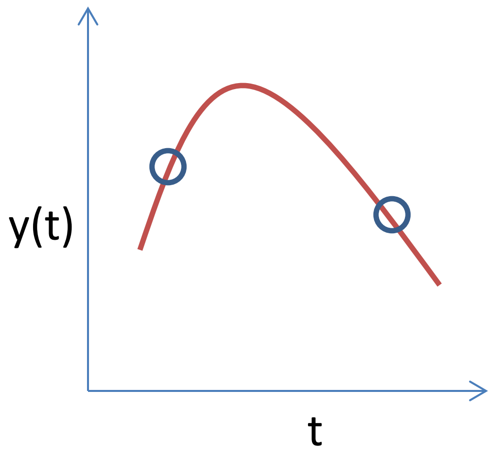
- This requires stepping away from the notion of interpolation. Instead, choose the support points based on the function to be integrated.

- The pre-calculations are more work, but for smooth integrands, Gaussian quadrature converges *exponentially* with the number of function evaluations.

Gaussian Quadrature



Trapezoidal Rule:
exact for constant
and linear functions.



Gaussian Quadrature (m=2):
exact for:
- constant,
- linear,
- quadratic and
- cubic functions.

In Gaussian Quadrature, the location of the support points are taken to be the roots of orthogonal polynomials. See appendix to the notes for technical details.

Gaussian Quadrature vs. Simpson’s Rule

- Benchmark: Evaluate the following integral using Gaussian Quadrature and Simpson’s Rule

$$\int_0^{\pi/2} dx \sin(x)$$

Number of Evaluations	Gaussian Quadrature	Simpson’s Rule
2	0.9984726135	1.0022798775
4	0.9999999770	1.0001345845
6	0.9999999904	1.0000263122
8	1.0000000001	1.0000082955
10	0.9999999902	1.0000033922

When to use which approach

- Always use Gaussian integration for a fixed number of integrand evaluations, for example when:
  - Required accuracy may be determined from the start,
  - a specific weight function is known,
  - non-adaptive behavior is not required,
  - you need to calculate a similar integral many times.

- Use Romberg integration when all you care about is reaching a particular tolerance.

Integrating in Python

- Methods:
  - scipy.integrate.quad() – gaussian quadrature
  - scipy.integrate.romberg() – romberg integration

- In both cases you:
  1. define a function for the integrand
  2. choose your limits of the integration
  3. set a tolerance

- Of course you are going to read the documentation before trying to use them!

Exercise #1

- Integrate the benchmark integral given earlier using Romberg and Gaussian Quadrature

$$\int_0^{\pi/2} dx \sin(x)$$

Exercises

1. Write a python script to integrate the following function

$$y(x) = \begin{cases} 0 & \text{for } x < -\pi \\ \cos(x) & \text{for } -\pi \leq x < 0 \\ \sin(x) & \text{for } 0 \leq x \leq \pi \\ 0 & \text{for } \pi < x \end{cases}$$

The limits of the integral should be from –infinity to infinity.