Week 8: Representing functions: interpolation, integration, differentiation

Polynomial bases for colocation, numerical quadrature and finite differencing

Dr K Clough, Topics in Scientific computing, Autumn term 2025

Plan for today

1. What are we doing and why?

2. Interpolation of functions - given some points, give me a function that roughly fits the true one at all points in the interval

3. Integration of functions - given some points, estimate the area under the curve (also used to integrate in time as in the Euler method we have seen)

   Next week:

4. Differentiation of functions - given some points, estimate the local derivative of the function (we will do this next week, but have it in mind now!)

What are we doing and why?

- Most physical systems are described in the language of ODEs and PDEs, not as closed form expressions
- ODEs are described by the value of a variable (or variables) versus time
- PDEs are described by the value of a function (or functions) versus time

  e.g. the heat equation

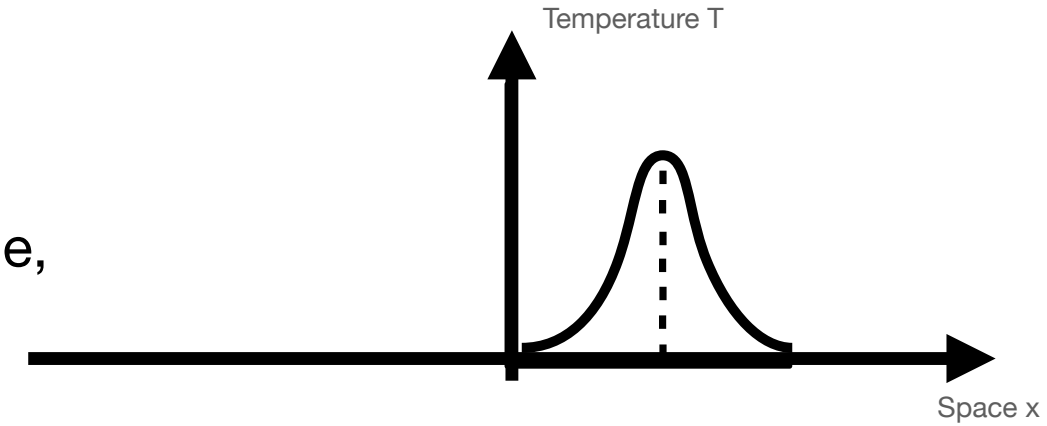
$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

What are we doing and why?

- e.g. the heat equation

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

- Tell me your initial temperature profile, and I can tell you how it changes
- Cannot simply write down T = f(x,t) except in very simple cases



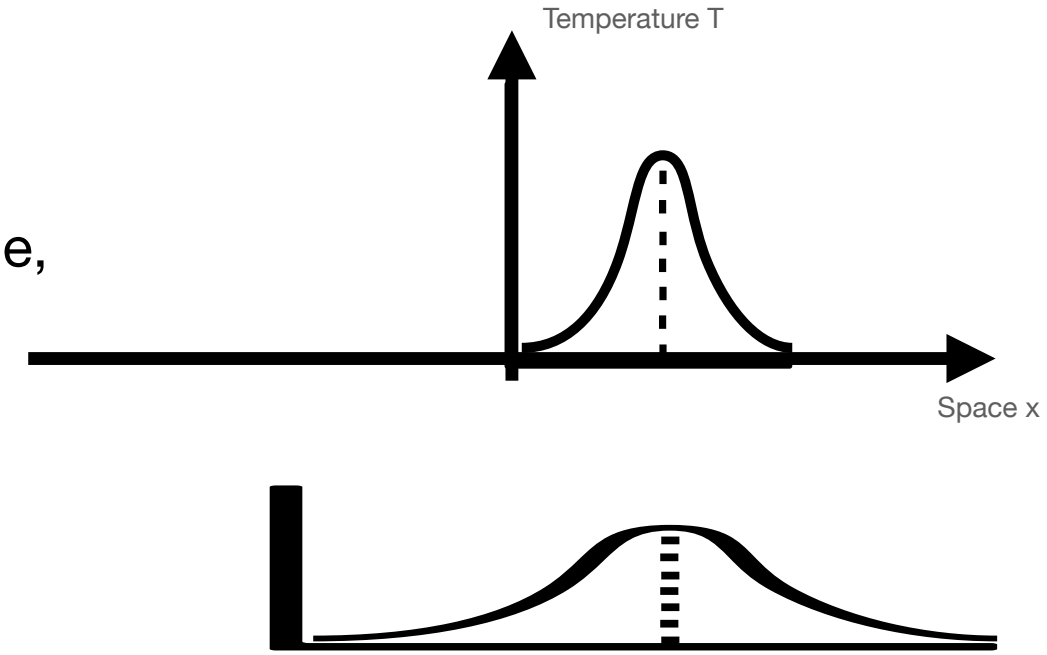
How does T change over time?

What are we doing and why?

- e.g. the heat equation

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

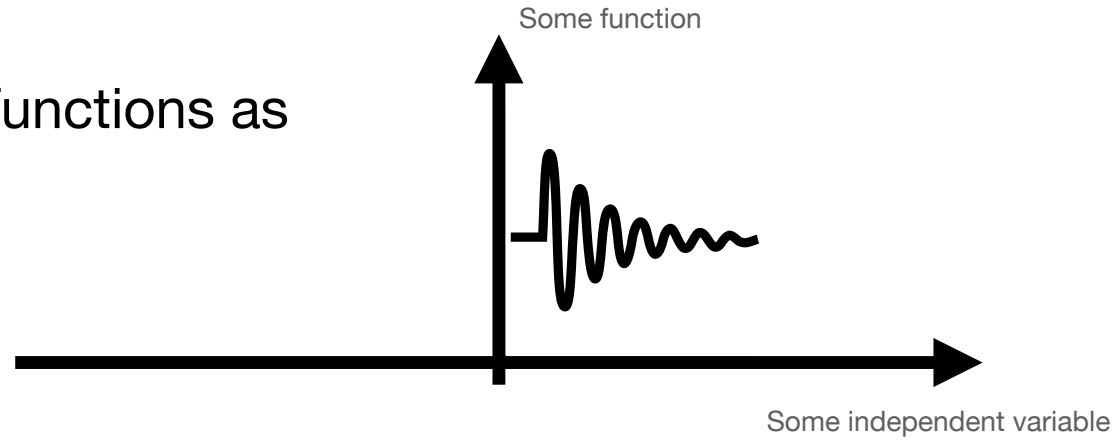
- Tell me your initial temperature profile, and I can tell you how it changes
- Cannot simply write down T = f(x,t) except in very simple cases



The temperature profile spreads out - the value decreases at a maximum where the second derivative is negative

What are we doing and why?

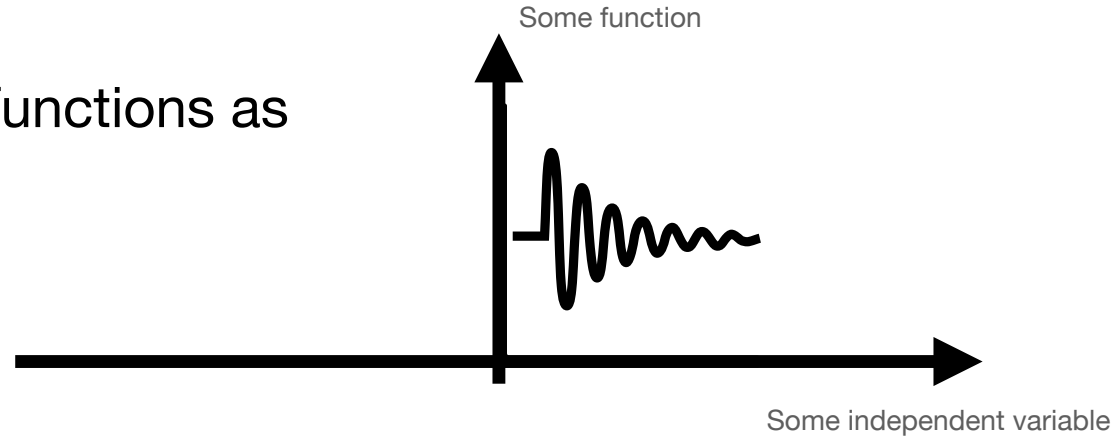
- **It is difficult to store (irregular) continuous functions in finite memory on a computer**
- In general we will want to represent functions as some collection of discrete values



How could I most efficiently store this function?

What are we doing and why?

- **It is difficult to store (irregular) continuous functions in finite memory on a computer**
- In general we will want to represent functions as some collection of discrete values



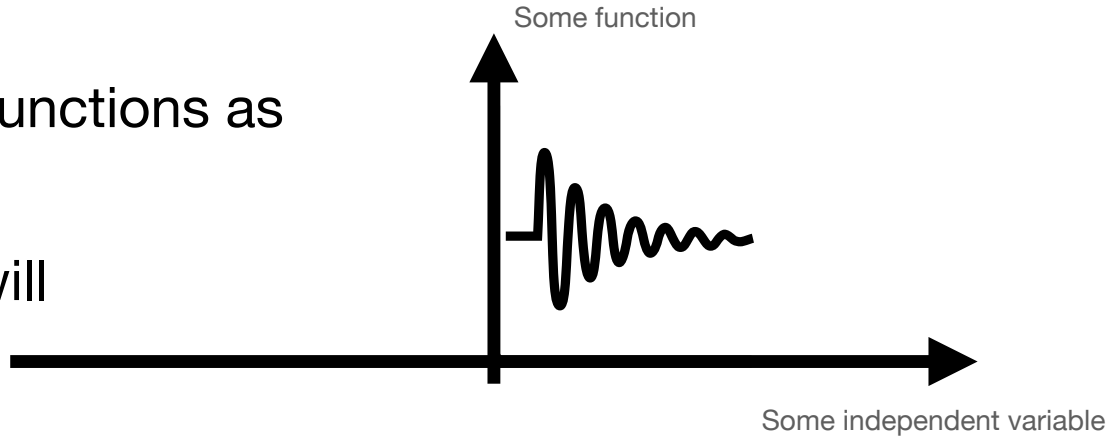
This looks like a damped sinusoid, so I could store it as 4 numbers, the amplitude, frequency, offset and the decay rate, that is I could fit A, B, C and D in the expression:

$f(x) = Ae^{-Bt}\sin(Cx)$ + D. This is a spectral method.

I could also just store the function values at discrete points, which would require less knowledge about the form of the function.

What are we doing and why?

- **It is difficult to store (irregular) continuous functions in finite memory on a computer**
- In general we will want to represent functions as some collection of discrete values
- Physical quantities or the evolution will be related to the integral or derivative of the function, so we need ways to represent and calculate these from the discrete values we store, e.g. for

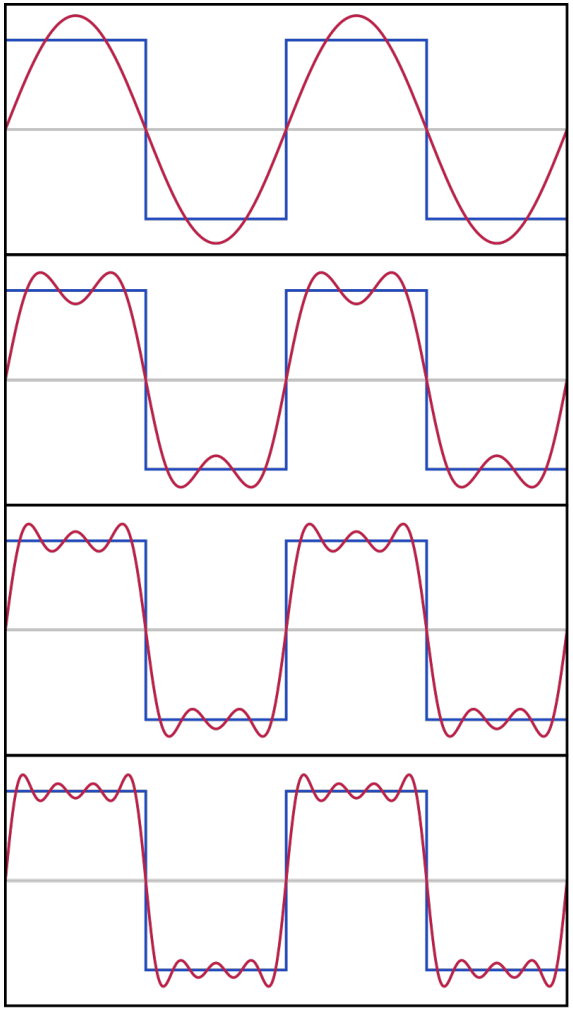


$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

What are we doing and why?

- **All methods rely on some basis of functions**
- We express the true function as a weighted sum of some basis functions
- Usually we require the basis functions to be **orthonormal**
- e.g. Fourier series representation of periodic functions

$$f(x) \sim A_0 + \sum_{n=1}^{\infty} \left(A_n \cos(2\pi n x / P) + B_n \sin(2\pi n x / P) \right)$$



Plan for today

1. ~~What are we doing and why?~~

2. Interpolation of functions - given some points, give me a function that roughly fits the true one at all points in the interval

3. Integration of functions - given some points, estimate the area under the curve (also used to integrate in time as in the Euler method we have seen)

   Next week:

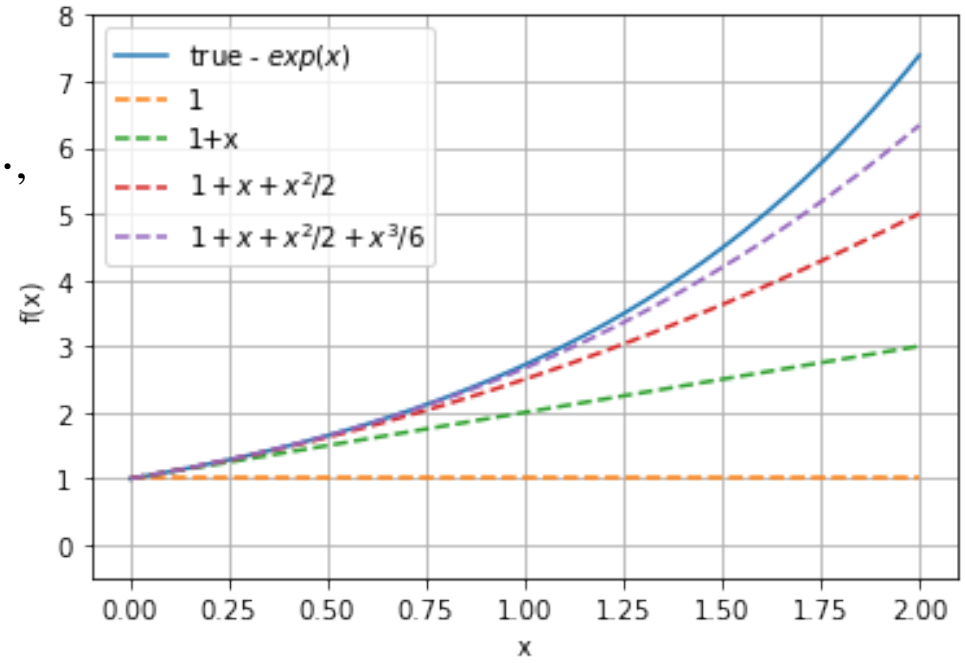
4. Differentiation of functions - given some points, estimate the local derivative of the function (we will do this next week, but have it in mind now!)

How to choose a basis

- Choice of basis function is problem dependent, but some are more robust than others
- Consider Taylor polynomials

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots,$$

Are the Taylor polynomials a good basis for the exponential function near x=0?

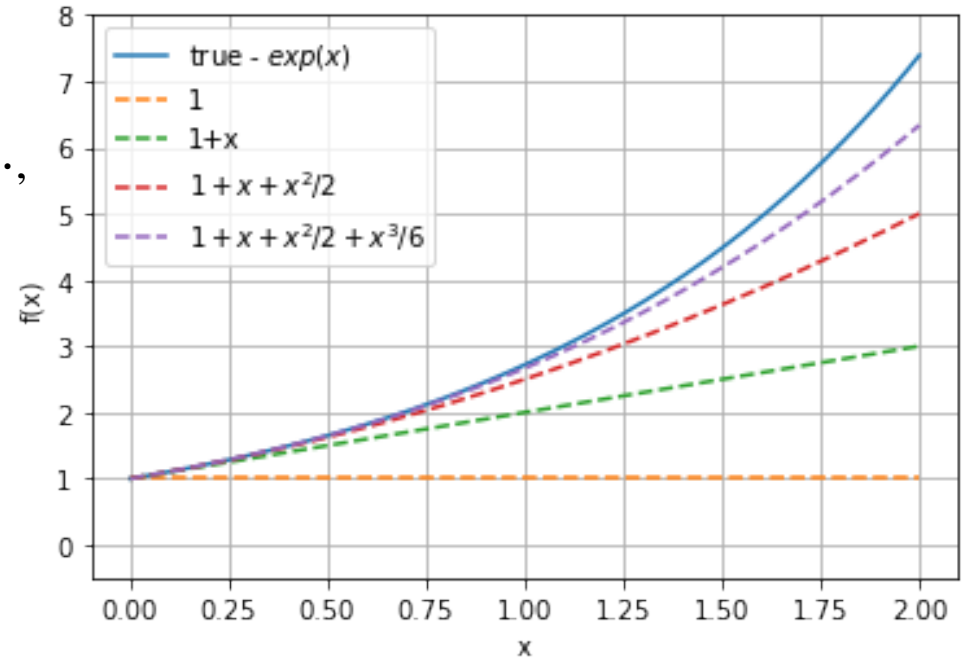


How to choose a basis

- Choice of basis function is problem dependent, but some are more robust than others
- Consider Taylor polynomials

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots,$$

Yes, they converge to the true solution as the order increases

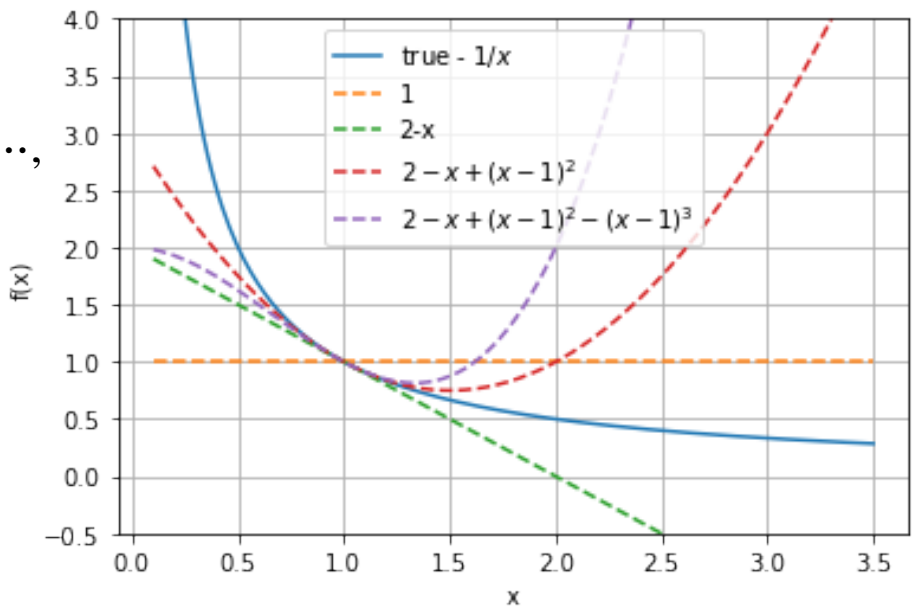


How to choose a basis

- Choice of basis function is problem dependent, but some are more robust than others
- Consider Taylor polynomials

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots,$$

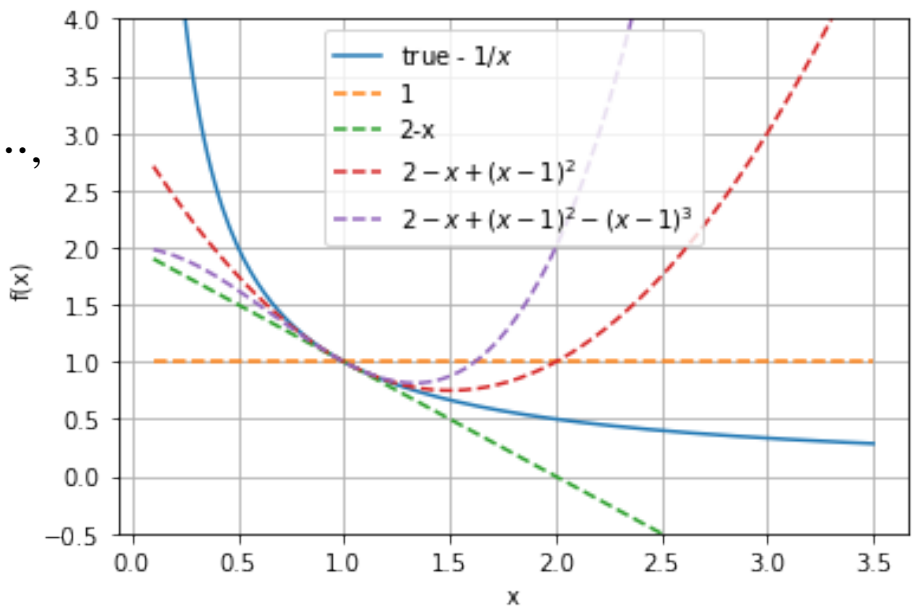
Are the Taylor polynomials a good basis for the function 1/x near x=1?



How to choose a basis

- Choice of basis function is problem dependent, but some are more robust than others
- Consider Taylor polynomials

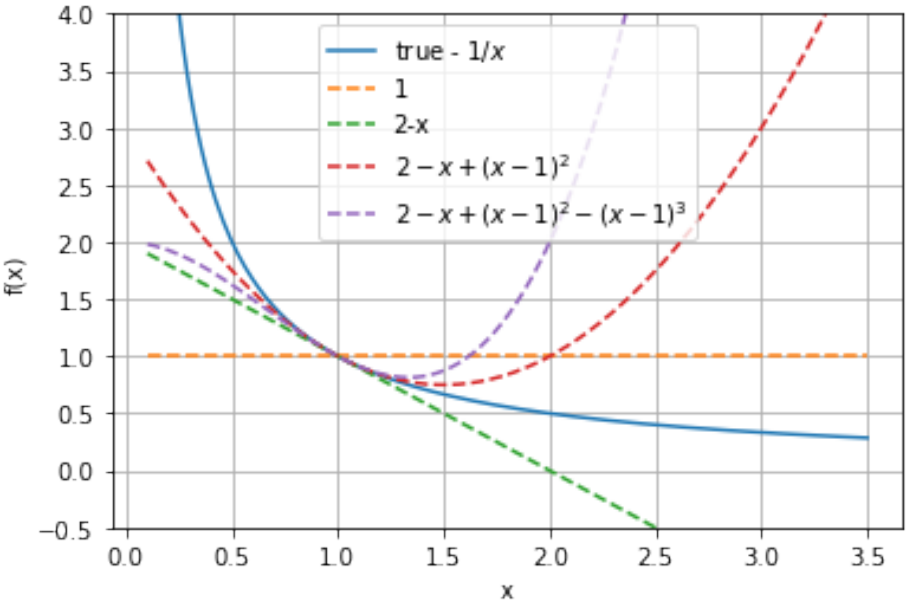
$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots,$$



No! They do not converge to the true solution as the order increases, in fact they diverge rapidly away from x=a, and get worse at higher order!

Taylor polynomials are too local

- Taylor polynomials use the value of the function and its derivatives at a single point
- In general we will get better fits by using more global (or at least a bit less local) information
- This will constrain the behaviour in some interval, rather than at a single point
- Another way to think about this is that we are always *extrapolating* with a Taylor polynomial, and not *interpolating,* and we know this is usually a bad idea



Lagrange polynomials use colocation at points

- Degree n Lagrange polynomials agree exactly with a function f(x) at n+1 distinct points, $(x_0, f(x_0)),\ (x_1, f(x_1)) \ldots (x_{n+1}, f(x_{n+1}))$

How can I construct an order n polynomial that is zero at all the given points except x_k, and has a value of f(x_k) at that point?

Lagrange polynomials use colocation at points

- Degree n Lagrange polynomials agree exactly with a function f(x) at n+1 distinct points, $(x_0, f(x_0)),\ (x_1, f(x_1)) \ldots (x_{n+1}, f(x_{n+1}))$
- First we construct the basis functions

$$L_k(x) = \prod_{i=0, i \neq k}^{n} \frac{(x - x_i)}{(x_k - x_i)}$$

What does the numerator and denominator achieve here?

- Then their weights are the values of the functions at each point, so that the Lagrange interpolant is:

$$P_n(x) = \sum_{k=0}^{n} L_k(x)\, f(x_k)$$

Lagrange polynomials use colocation at points

- Degree n Lagrange polynomials agree exactly with a function f(x) at n+1 distinct points, $(x_0, f(x_0)),\ (x_1, f(x_1)) \ldots (x_{n+1}, f(x_{n+1}))$
- First we construct the basis functions

$$L_k(x) = \prod_{i=0, i \neq k}^{n} \frac{(x - x_i)}{(x_k - x_i)}$$

Numerator - functions to be zero at all of the points other than x_k - denominator - function is normalised so it has value 1 at x_k

- Then their weights are the values of the functions at each point, so that the Lagrange interpolant is:

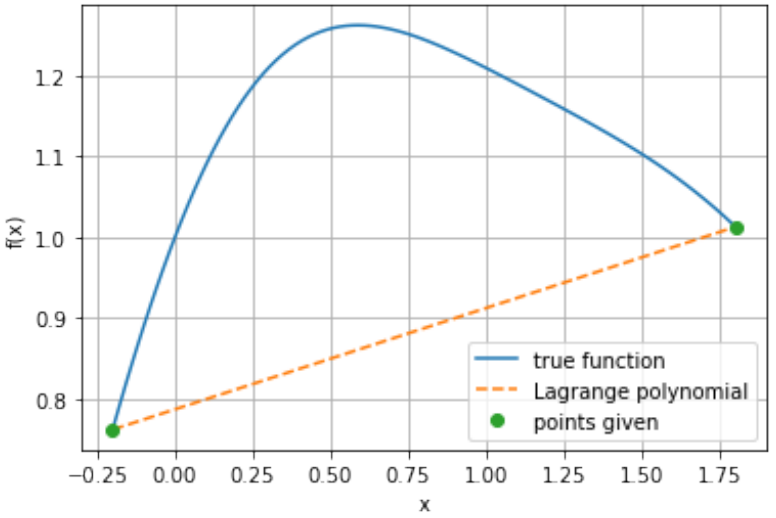
$$P_n(x) = \sum_{k=0}^{n} L_k(x)\ f(x_k)$$

Lagrange polynomials use colocation at points

- Degree n Lagrange polynomials agree exactly with a function f(x) at n+1 distinct points, $(x_0, f(x_0)),\ (x_1, f(x_1)) \ldots (x_{n+1}, f(x_{n+1}))$
- Easiest to see an example:

```
x_points = np.array([-0.2, 1.8])
y_points = get_y_test_function(x_points)

# Construct the polynomial using sympy
[x1,x2] = x_points
[f1,f2] = y_points

x = symbols('x')
L1 = (x - x2 ) / (x1 - x2)
L2 = (x - x1 ) / (x2 - x1)
P = f1 * L1 + f2 * L2
print("Using sympy we get", simplify(P))
```

```
Using sympy we get 0.12544570880996*x + 0.787209250119254
```

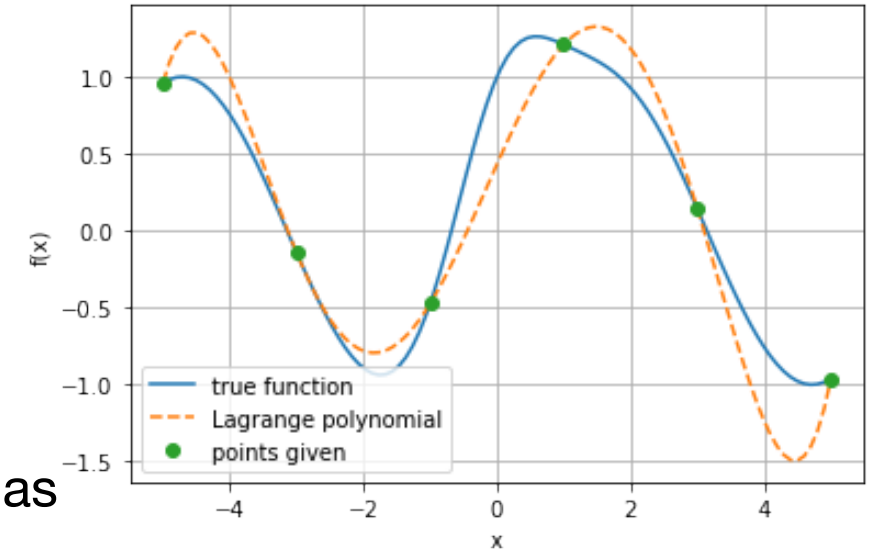


Error theorem for Lagrange polynomials

- Can show that the max error is:

$$E_{max} = \max \left[\left| \frac{f^{n+1}(\zeta)}{(n+1)!} \prod_{i=0}^{n} (x - x_i) \right| \right]$$

where $f^{n+1}(\zeta)$ is the (n+1)th derivative of f and $\zeta \in [a, b]$



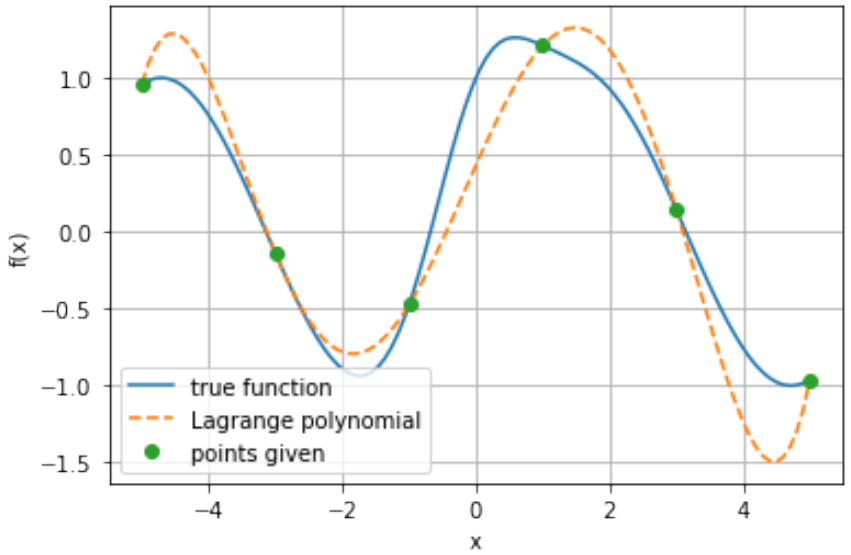
- Can estimate this (if actual function unknown) as

$$E_{max} \sim \max \left[\left| P_{n+1}(x) - P_n(x) \right| \right]$$

Lagrange polynomials use colocation at points

- Can use a python function *scipy.interpolate.lagrange()* to construct using higher number of points

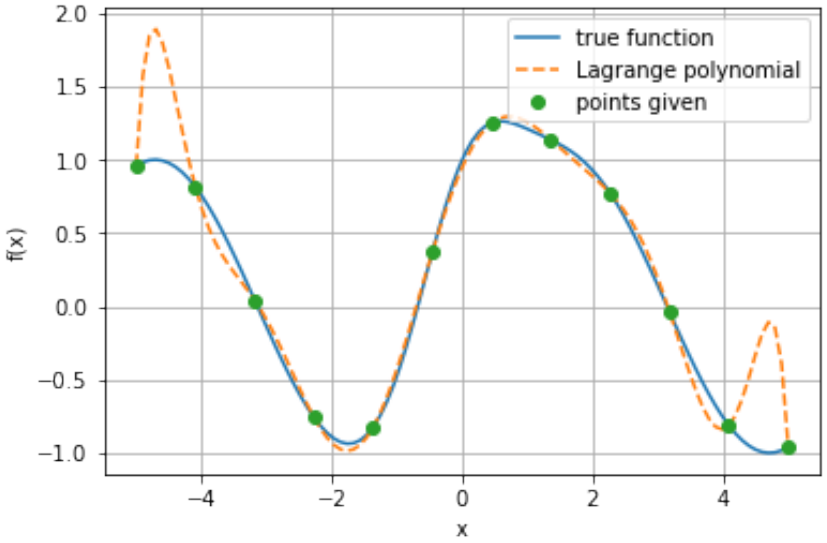
Is a higher number of points always better?



Lagrange polynomials use colocation at points

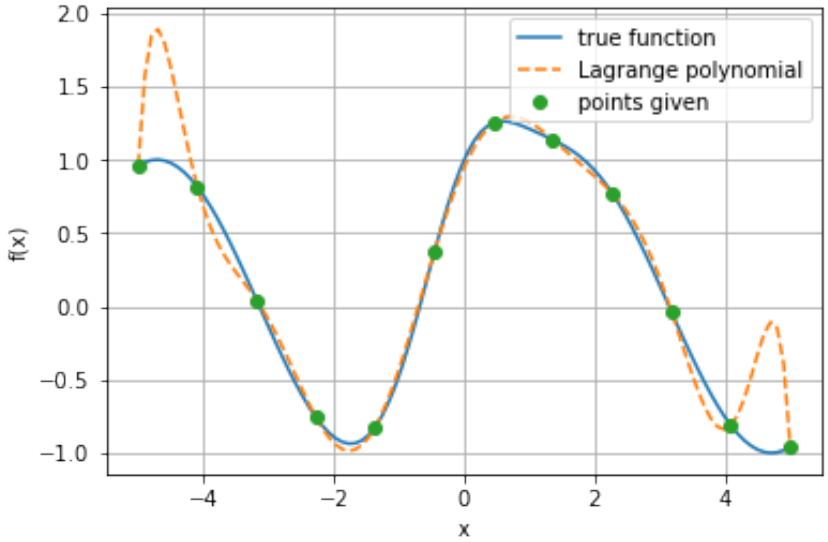
- More points improved the fit at the interior, but with regular intervals it tends to lead to spurious oscillations at the edges of the interval

  -> “Runge’s phenomenon”



Interpolation using colocation

How can we do better?



Interpolation using colocation

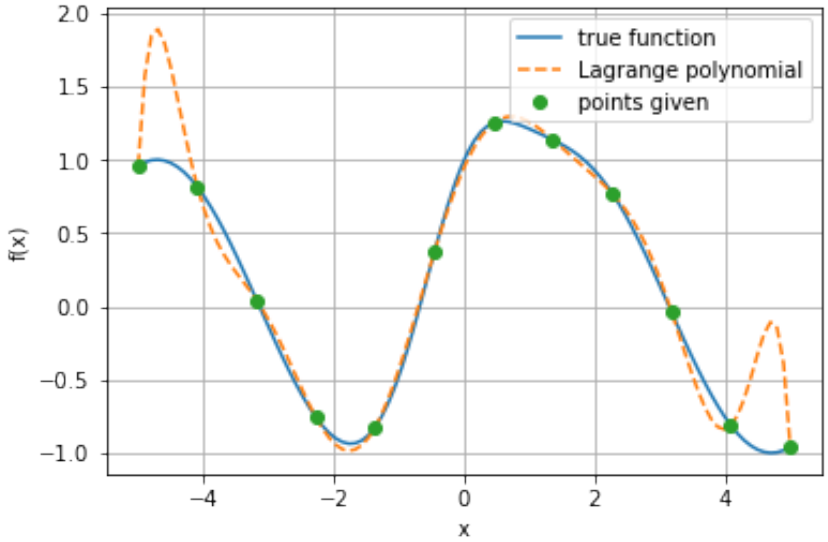
How can we do better?

Some ideas:

1. Split the interval up into sections and use lower order interpolants for each section
   -> “composite collocation"

2. Use more information e.g. about the derivatives at each point.
   -> “osculating polynomials, e.g. Hermite”

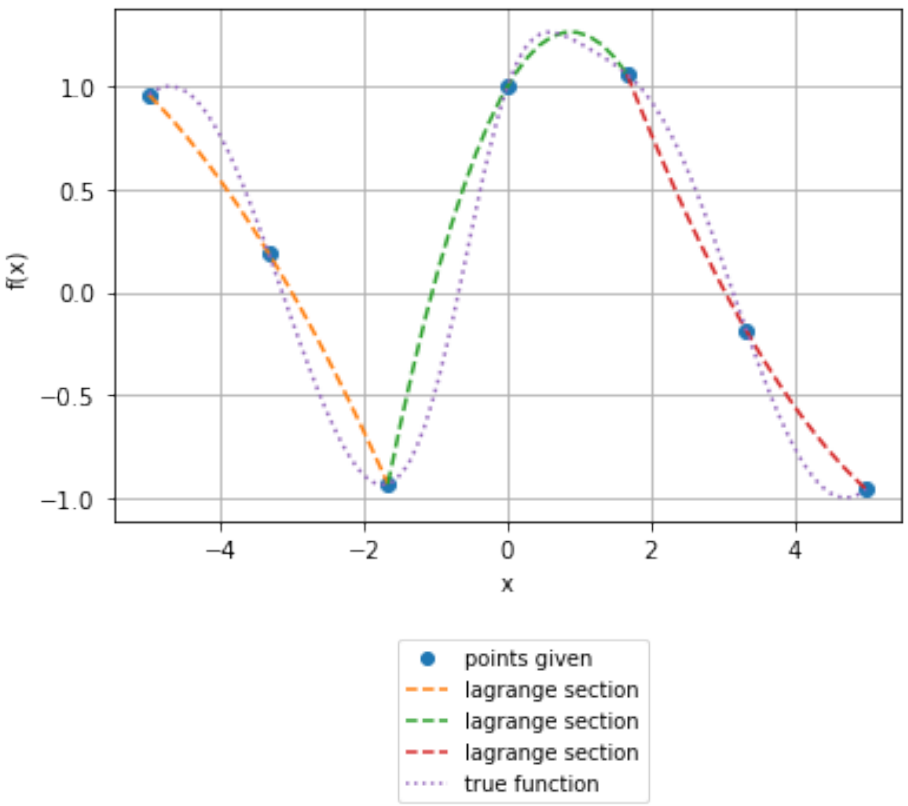
3. Move the points to strategic locations
   -> pseudospectral/Gauss Lobato nodes



How to do better - composite colocation

1. We could try to divide the interval up into smaller sections and fit lower order Lagrange polynomials to each part in turn - this is a **composite colocation** method.

What are the disadvantages here?



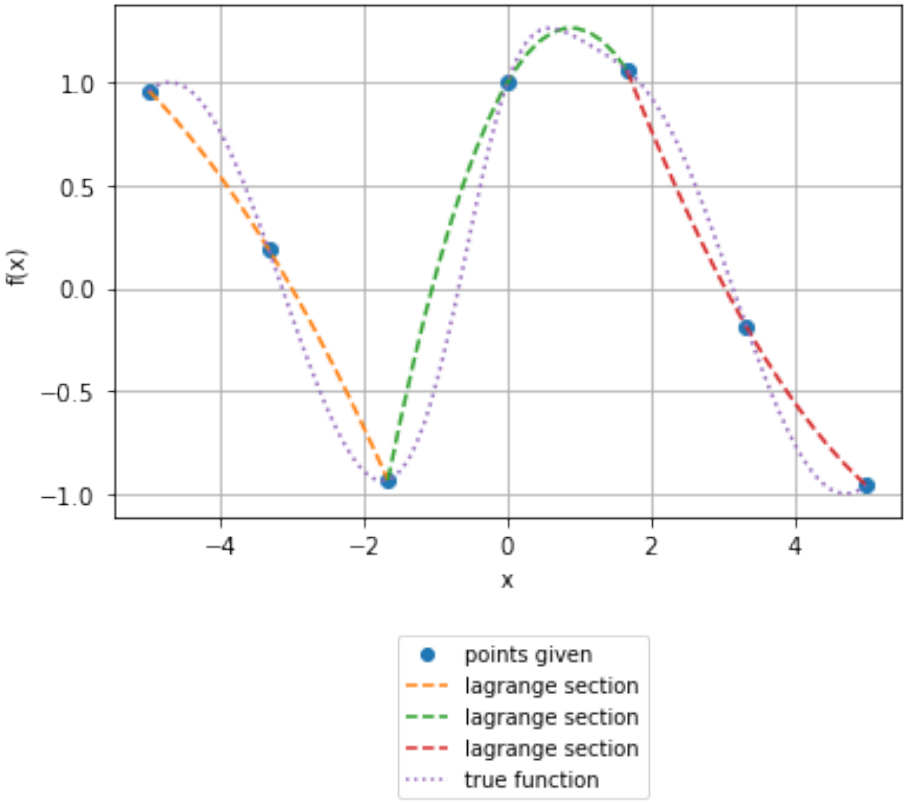
How to do better - composite colocation

1. We could try to divide the interval up into smaller sections and fit lower order Lagrange polynomials to each part in turn - this is a **composite colocation** method.

   Disadvantages:

   - Derivatives don't match between sections so function not smooth

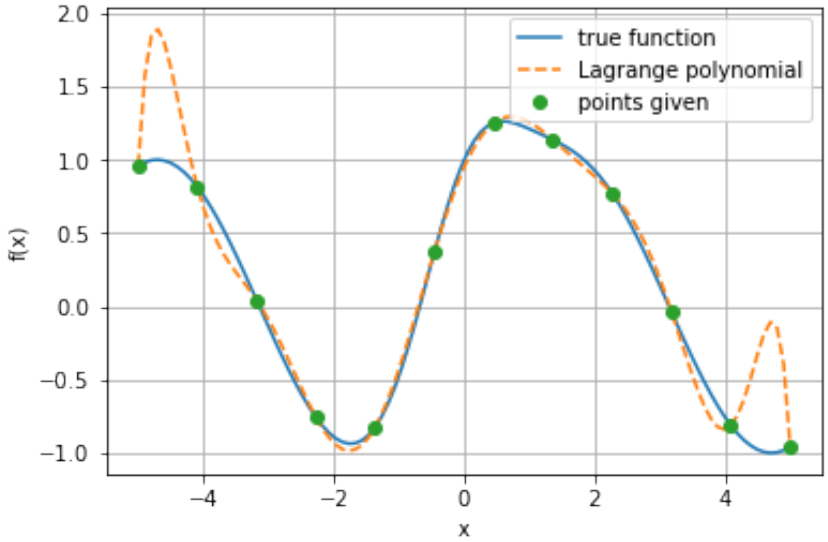
   - We have different polynomials for each section, rather than one overall one, so less "analytic"

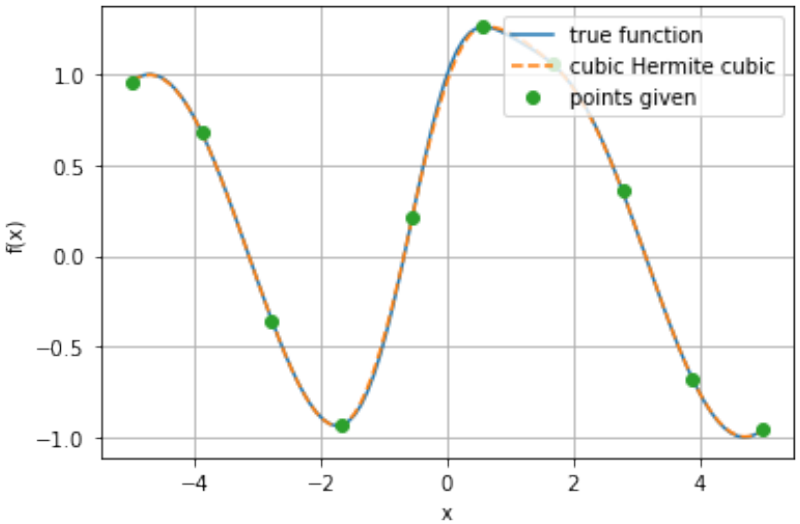


How to do better - osculating polynomials

2. **Osculating polynomials** - match both the values of the function and its derivatives (up to some order) at each point. e.g. **Hermite polynomials** fit the first derivatives only, so now if we use composite sections the (first) derivatives will be smooth.

   Python function *scipy.interpolate.CubicHermiteSpline* can be used for this.



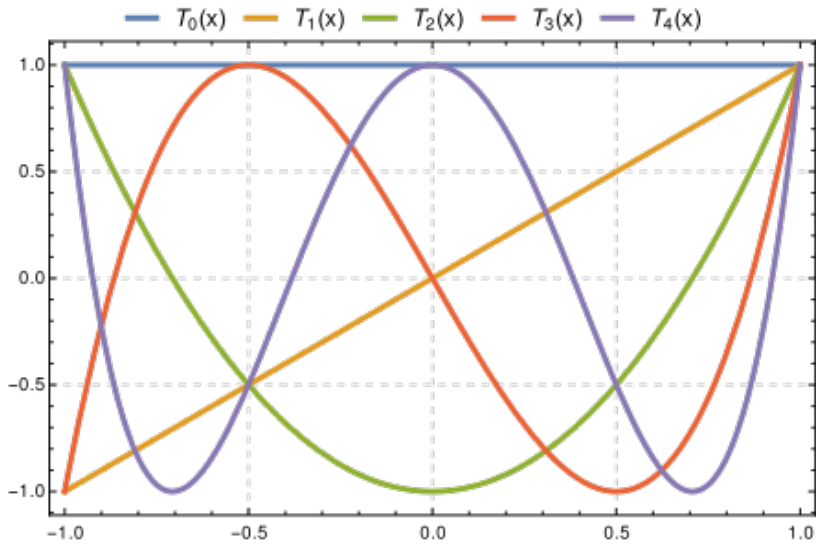


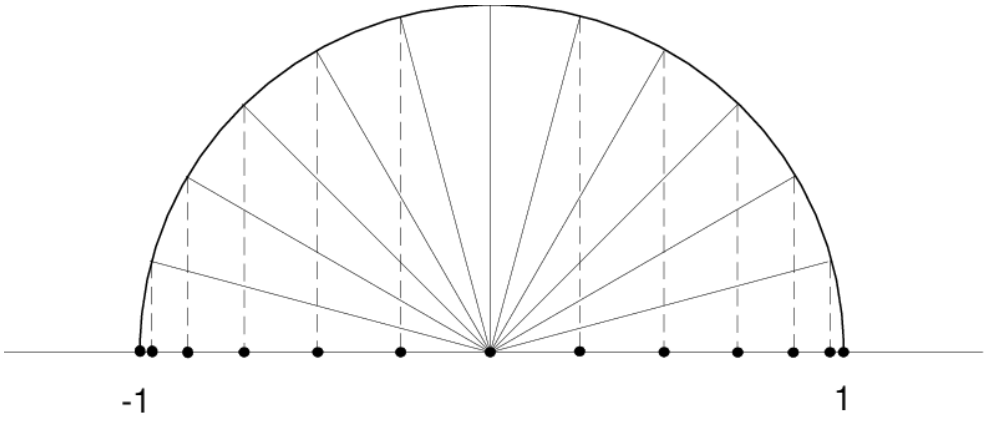
How to do better - Chebyshev Polynomials

3. The **Gauss Lobato nodes** are the zeros of the Chebyshev polynomials defined by $T_n(\cos(x)) = \cos(nx)$. (They are also the projections of equally spaced sections of a unit circle onto the x axis - see below)

   By locating the points at these locations, we effectively force the polynomial basis to be the Chebyshev polynomials.

   One can show that this makes the error exponentially convergent with the order of the polynomial.



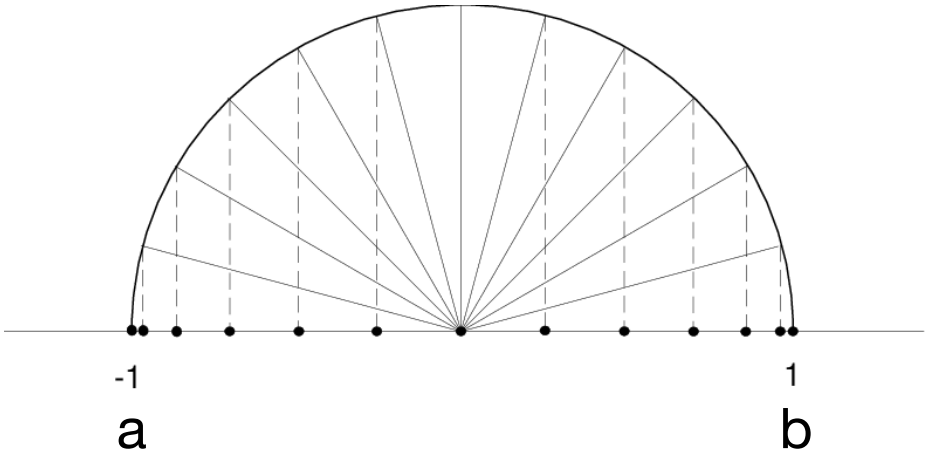


How to do better - Chebyshev Polynomials

3. Note that the Gauss Lobato nodes for the unit circle are at

   $u_i = \cos\left(\frac{\pi i}{N}\right)$ but for a general interval $[a, b]$ their values need to be

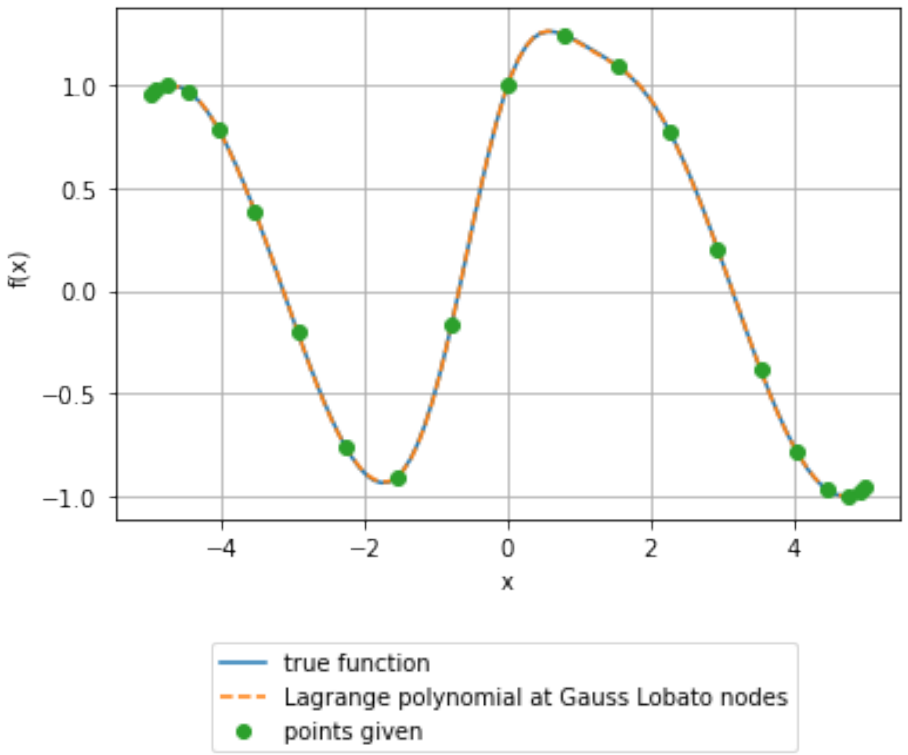
   mapped to $x_i = \frac{1}{2}(b-a)u_i + \frac{1}{2}(a+b)$



How to do better - Chebyshev Polynomials

3. Procedure: locate the points to be fit at the (appropriately mapped) Gauss Lobato nodes, and then fit a Lagrange polynomial of the appropriate order
   (Note: we could also use orthogonality of the Chebychev polynomials to find the coefficients of each $T_i(x)$, but we still require collocation at the points so the polynomial obtained is unique and thus should be the same via either method).

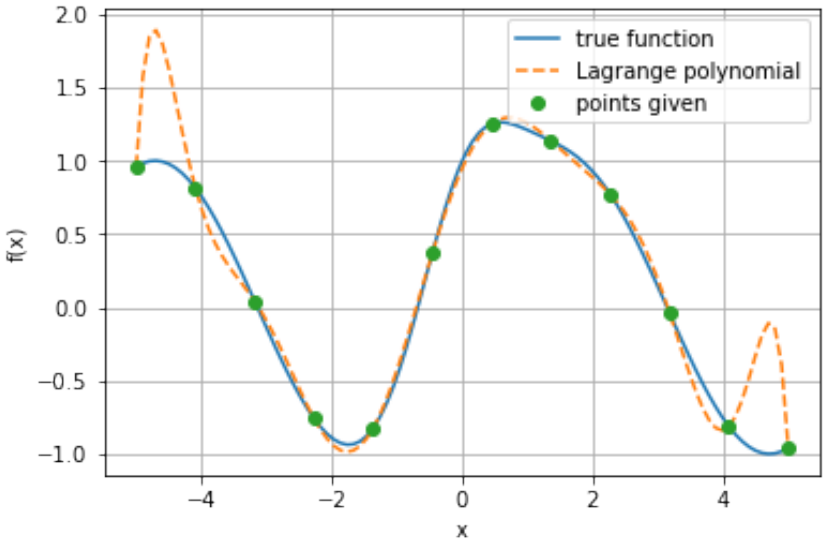
   This procedure stabilises the fit against Runge's phenomenon as the number of points increases. For smooth functions it is a very efficient way to represent the data accurately using the minimum number of points.



In this week's tutorial

We will investigate all of these methods for fitting a function and compare them.

```
x_points = np.array([-0.2, 1.8])
y_points = get_y_test_function(x_points)

# Construct the polynomial using sympy
[x1,x2] = x_points
[f1,f2] = y_points

x = symbols('x')
L1 = (x - x2 ) / (x1 - x2)
L2 = (x - x1 ) / (x2 - x1)
P = f1 * L1 + f2 * L2
print("Using sympy we get", simplify(P))

lagrange_polynomial = lagrange(x_points, y_points)
print("Using the scipy function we get \n",lagrange_polynomial)
print("We can also return the coefficients of the polynomial as an array: ",
      lagrange_polynomial.coef)

x_true = np.linspace(-0.2,1.8,100)
plt.plot(x_true, get_y_test_function(x_true), '-', label="true function")
plt.plot(x_true, lagrange_polynomial(x_true), '--', label="Lagrange polynomial")
plt.plot(x_points, y_points, 'o', label="points given")
plt.xlabel("x")
plt.ylabel("f(x)")
```



Plan for today

1. ~~What are we doing and why?~~

2. ~~Interpolation of functions - given some points, give me a function that roughly fits the true one at all points in the interval~~

3. Integration of functions - given some points, estimate the area under the curve (also used to integrate in time as in the Euler method we have seen)

   Next week:

4. Differentiation of functions - given some points, estimate the local derivative of the function (we will do this next week, but have it in mind now!)

Integration

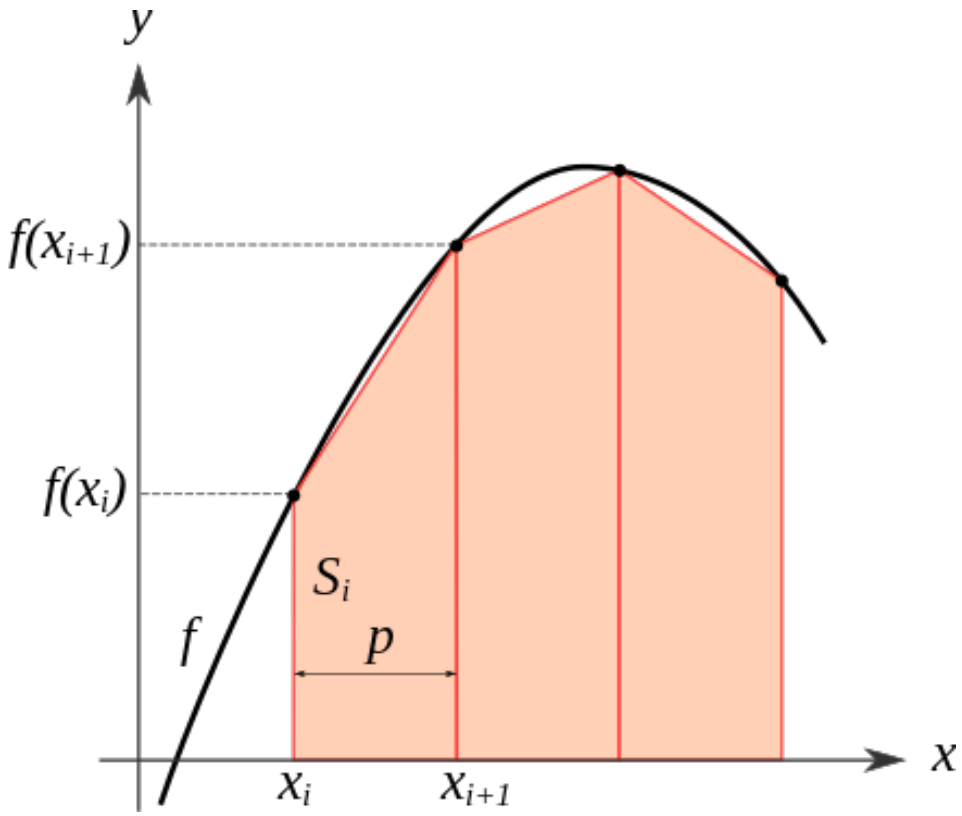
An approximation to the integral of a function as a weighted sum of its values at certain points

$$I[f] = \int_a^b f(x)\,dx = \sum_i w_i f(x_i)$$

is called a **numerical quadrature.**

Methods can be classed as:

- Single step (often used for time integration between steps in an initial value problem)
- Multistep and composite (often used for spatial integration at a single time step)



Integration - single step methods

We will study 2 methods that just use information about the function at the two end points:

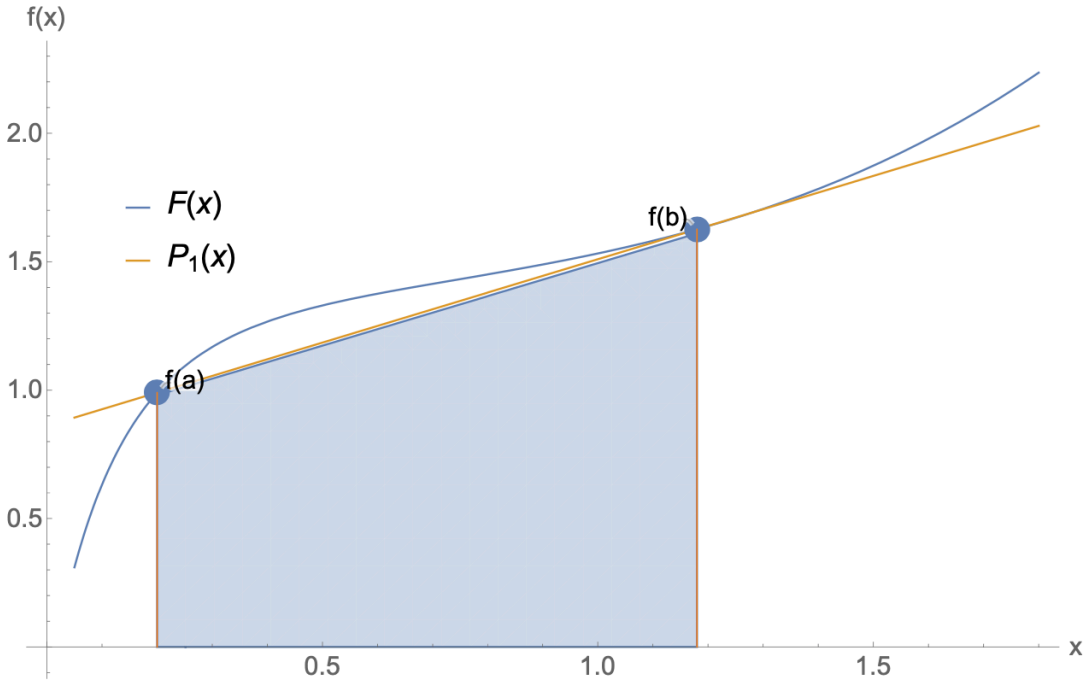
1. Trapezoid rule

   - uses the value of f

2. Hermite rule

   - uses the value of f and dfdx

3. One can go to even higher orders but we will not discuss these (see e.g. the Hermite-Lanczos-Dyche or Euler-Maclaurin formulae)



Trapezoid rule

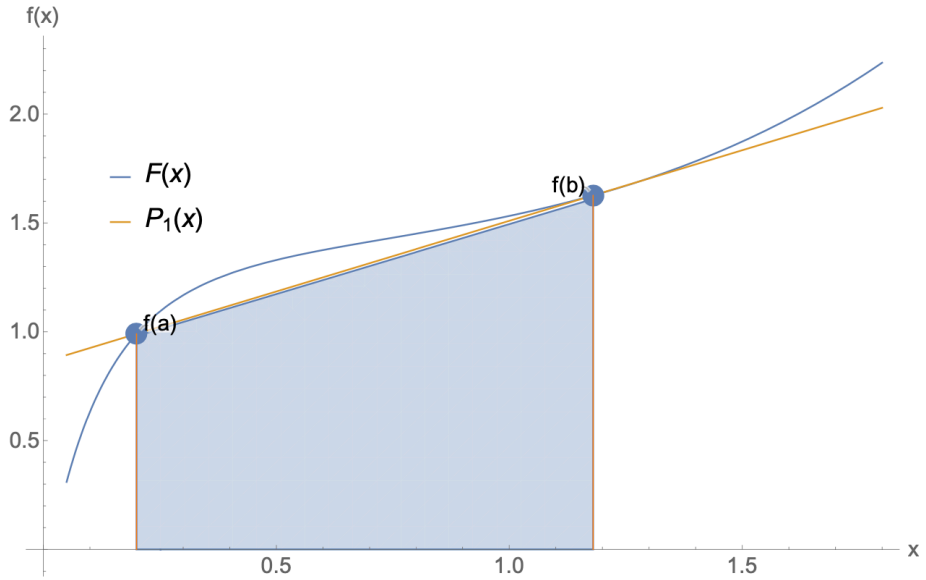
- Fit a Lagrange polynomial of order 1 to the endpoints:

$$P(x) = \frac{(b-x)}{\Delta x} f(a) + \frac{(x-a)}{\Delta x} f(b)$$

- Integrating this between the limits gives the well known formula:

$$\int_a^b f(x)\,dx \approx \frac{f(a)+f(b)}{2}\Delta x + O(\Delta x^3)$$

- Note that the trapezium rule will always over or under estimate the integral



Hermite rule

- Knowing the function $f(x)$ and its derivatives $f'(x)$ at each end of the interval, we can construct a cubic Hermite interpolating polynomial *(homework - derive or look up the derivation of this cubic polynomial).*

- Integrating this cubic function between the limits yields a correction to the trapezium rule:

$$\int_a^b f(x)\,dx \approx \frac{f(a)+f(b)}{2}\Delta x + \frac{f'(a)-f'(b)}{12}\Delta x^2 + O(\Delta x^5)$$

- We see that this method is much more accurate for a given step size, at the cost of needing additional information about the function derivatives.

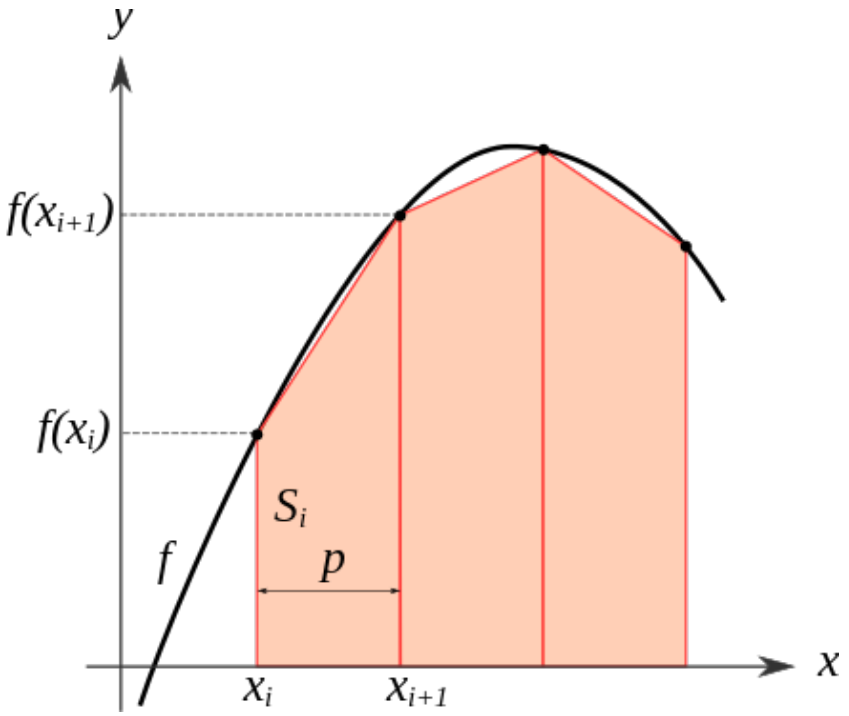
Integration - multistep / composite methods

We will study 2 methods:

1. Composite trapezium rule
2. (Composite) Simpson's 3 point method

   Simpson's method is very accurate
   and you rarely need anything better.

   Unlike in interpolation, making the intervals
   smaller will usually be beneficial, and uniform
   spacing is also not usually problematic.



Another method is Gaussian quadrature - smarter but with additional complexity (not covered in this course).

The composite trapezoid rule

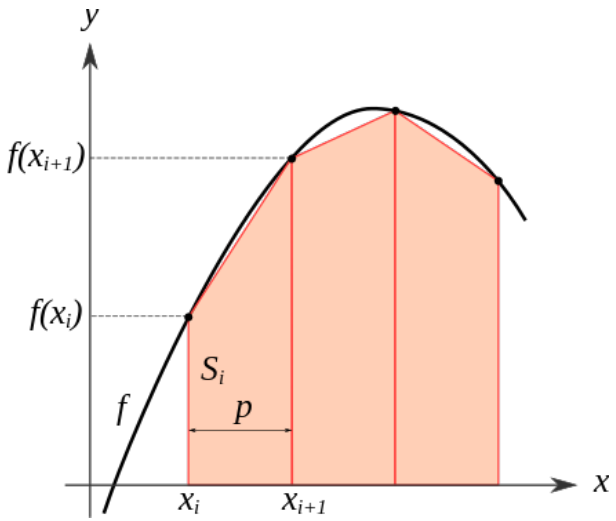
Composite version of the single step rule, error now $\sim O(\Delta x^2)$ because we have $N \sim 1/\Delta x$ sections each with error $\sim O(\Delta x^3)$.

- Divide the interval into sections, approximate the area under each section as a trapezium, and add them up:

$$\int_a^b f(x)\,dx \approx \sum_{k=1}^{N} \frac{f(x_{k-1}) + f(x_k)}{2} \Delta x_k,$$

- If the sections are the same width Δx this simplifies to:

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



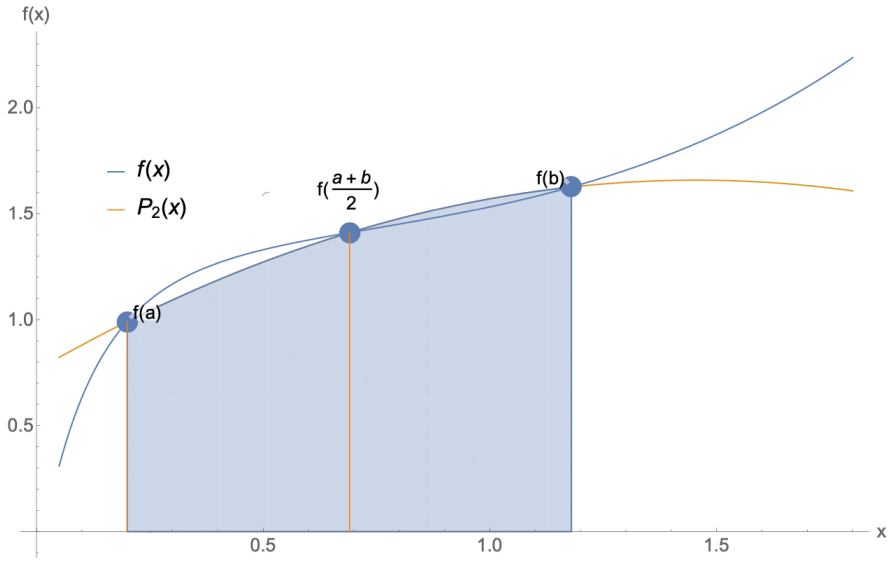
Simpson's 3 point rule

- Fit a Lagrange polynomial of order 2 to the endpoints and a midpoint:

$$P(x) = \frac{(x-x_1)}{x_0-x_1}\frac{(x-x_2)}{x_0-x_2}f(x_0) + \frac{(x-x_0)}{x_1-x_0}\frac{(x-x_2)}{x_1-x_2}f(x_1) + \frac{(x-x_0)}{x_2-x_0}\frac{(x-x_1)}{x_2-x_1}f(x_2)$$

- Integrating this between the limits and assuming equally spaced points $x_2 = x_1 + \Delta x = x_0 + 2\Delta x$ gives the resulting quadrature:

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\left(f(x_0) + 4f(x_1) + f(x_2)\right) + O(\Delta x^5)$$



Composite Simpson's 3 point rule

Composite version of the single step rule, error now $\sim O(\Delta x^4)$ because we have $N \sim 1/\Delta x$ sections each with error $\sim O(\Delta x^5)$.

- Divide the interval into sections, approximate the area under each section using 3 points to define the quadratic fit, then add them up.

- If the sections are the same width Δx this gives

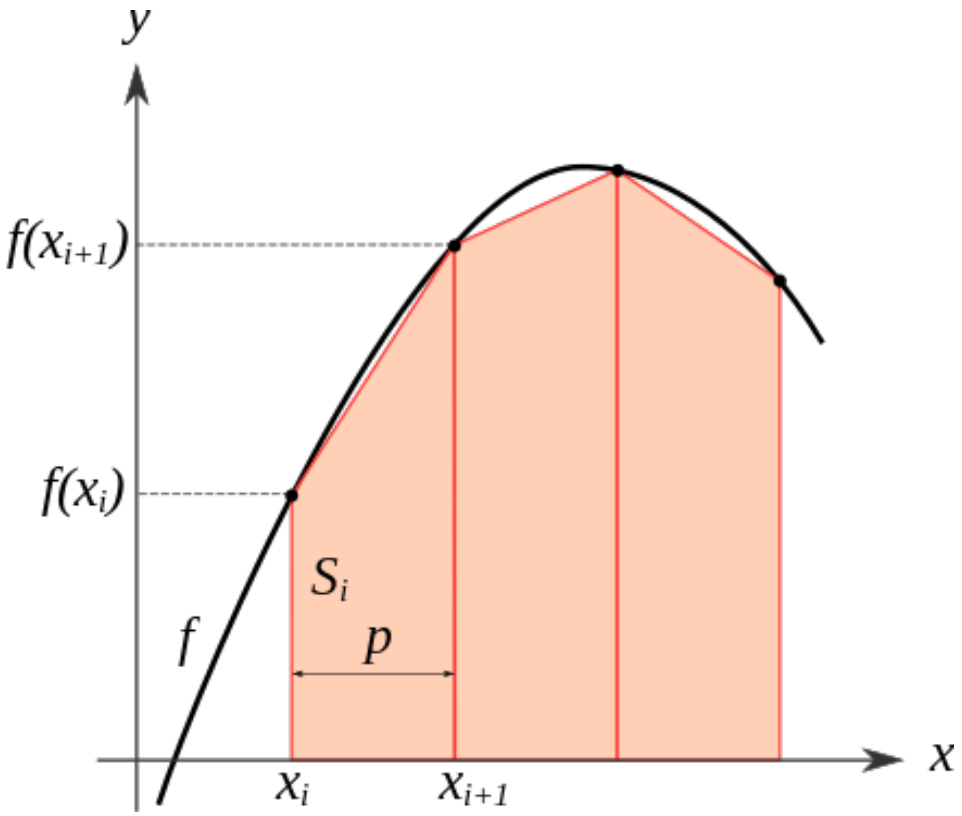
$$\int_a^b f(x)\,dx \approx \frac{1}{3}\Delta x \left[f(x_0) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + 2\sum_{i=1}^{n/2-1} f(x_{2i}) + f(x_n) \right].$$

- Python function exists for this!

Integration - remarks on higher dimensions

The extension of these methods to higher dimensions is reasonably straightforward - we just integrate along one direction at each point in the other (there are simple algorithms for fixed spacings with weights that you can look up in wikipedia).

For dimensions d > 7 these traditional numerical methods become inefficient and we need to use Monte Carlo methods - random sampling of the domain. These methods are commonly required in data analysis.

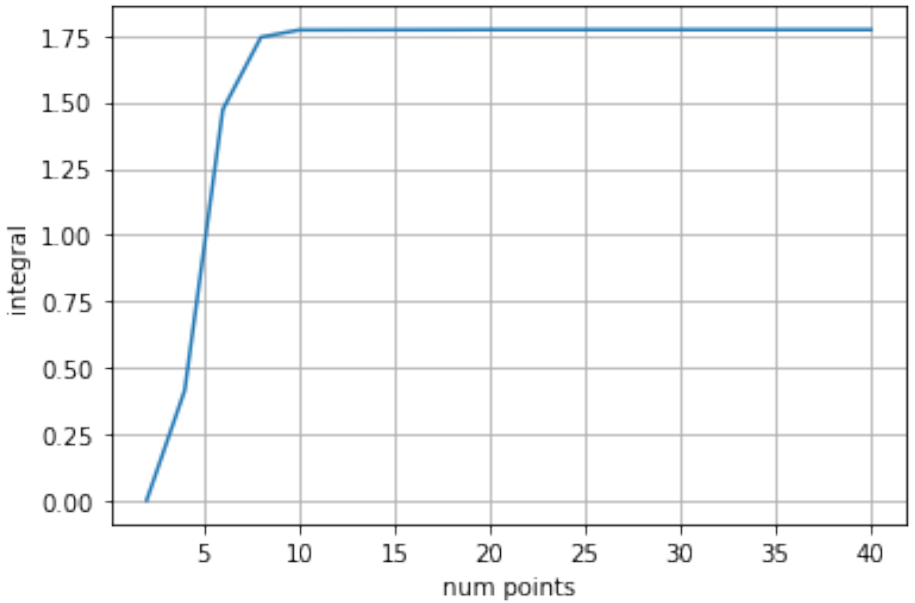


In this week's tutorial

We will investigate all of these methods for integrating a function and compare them.

```
# function to get integral using trapezoid rule
a = -5
b = 5
N = 10

# Implement your own version of the composite trapezoid rule
def get_test_integral(a_num_points) :
    x_integration = np.linspace(a , b,  a_num_points)
    y_integration = get_y_test_function(x_integration)
    dx = (b - a) / (a_num_points - 1)
    integral = (dx/2)*(y_integration[0] + 2 * sum(y_integration[1:a_num_points-1])
                       + y_integration[a_num_points-1])
    return integral

integral = get_test_integral(N)

# Check using the python function
x_integration = np.linspace(a , b,  N)
y_integration = get_y_test_function(x_integration)
integral_scipy = trapezoid(y_integration,x_integration)

print("The integral with ", N, " points is ", integral, integral_scipy)
```

```
The integral with  10  points is  1.7712579642870152 1.7712579642870154
```



Plan for today

1. What are we doing and why?

2. Interpolation of functions - given some points, give me a function that roughly fits the true one at all points in the interval

3. Integration of functions - given some points, estimate the area under the curve (also used to integrate in time as in the Euler method we have seen)

   Next week:

4. Differentiation of functions - given some points, estimate the local derivative of the function (we will do this next week, but have it in mind now!)