Interpolation and approximation

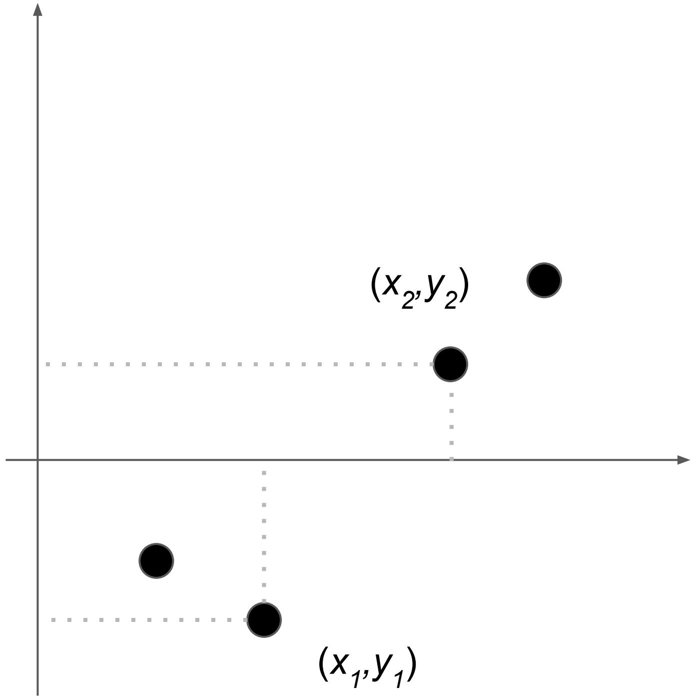
Kristina Kislyakova
University of Vienna - 29.10.2025

Outline

- Concepts and common uses of interpolation and approximation schemes
- Linear interpolation methods
- Lagrange, Hermite and Taylor polynomial interpolation
- Cubic splines
- Multi-dimensional applications: Bilinear interpolation, Kriging
- Weighted approximation schemes, robust statistics, outlier filtering, GPR
- Cookbook for what to use, when and what to be aware of.

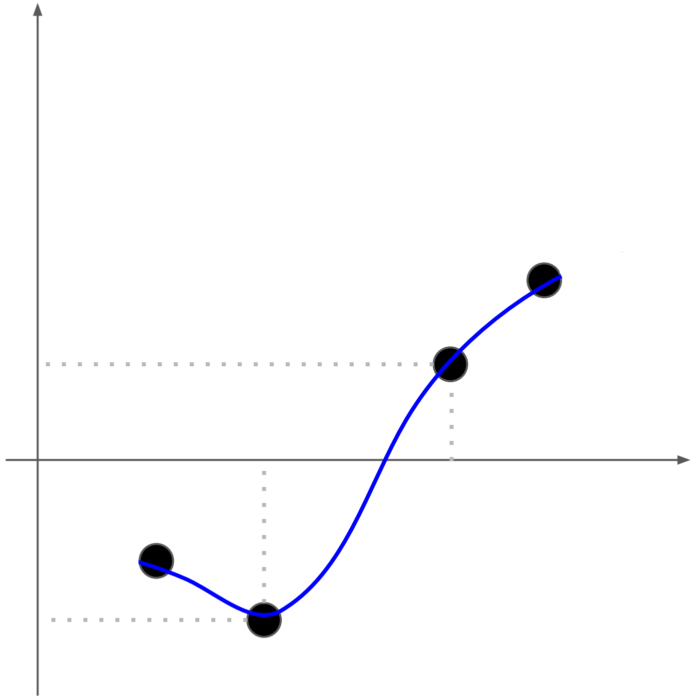
Discrete data representations

- Often have discrete data points for which you require an estimate at an intermediate value of (x,y)



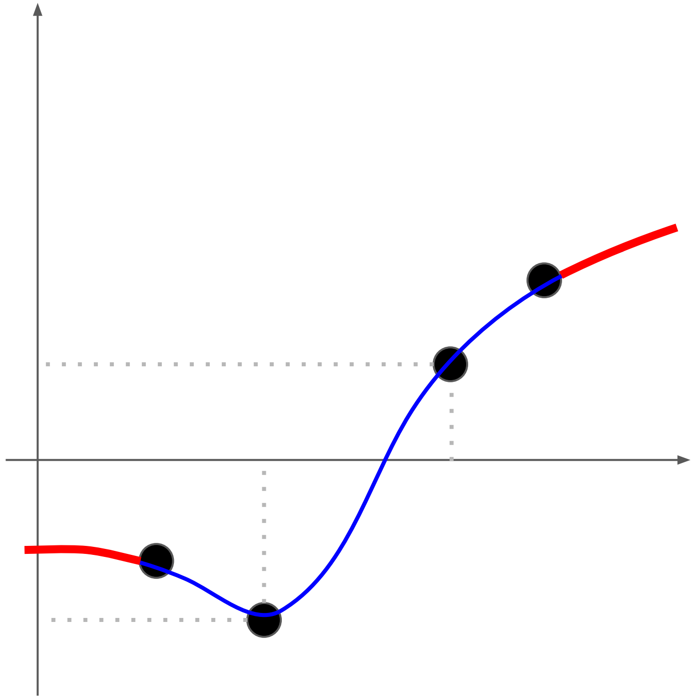
Discrete data representations

- Often have discrete data points for which you require an estimate at an intermediate value of (x,y)
- Functional representation of $y_i(x_i)$ allows sub-sampling/re-binning to intermediate values (interpolation)



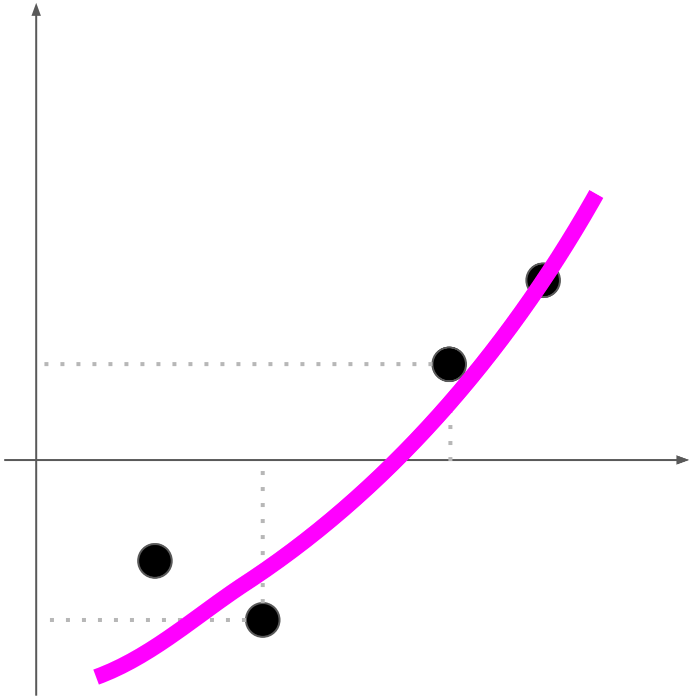
Discrete data representations

- Often have discrete data points for which you require an estimate at an intermediate value of (x,y)
- Functional representation of $y_i(x_i)$ allows sub-sampling/re-binning to intermediate values (interpolation)
- Sometimes extrapolation beyond $[x_{min}, x_{max}]$ is desired as well



Discrete data representations

- Often have discrete data points for which you require an estimate at an intermediate value of (x,y)
- Functional representation of $y_i(x_i)$ allows sub-sampling/re-binning to intermediate values (interpolation)
- Sometimes extrapolation beyond $[x_{min}, x_{max}]$ is desired as well
- Functions that do not pass through irregular points are approximations



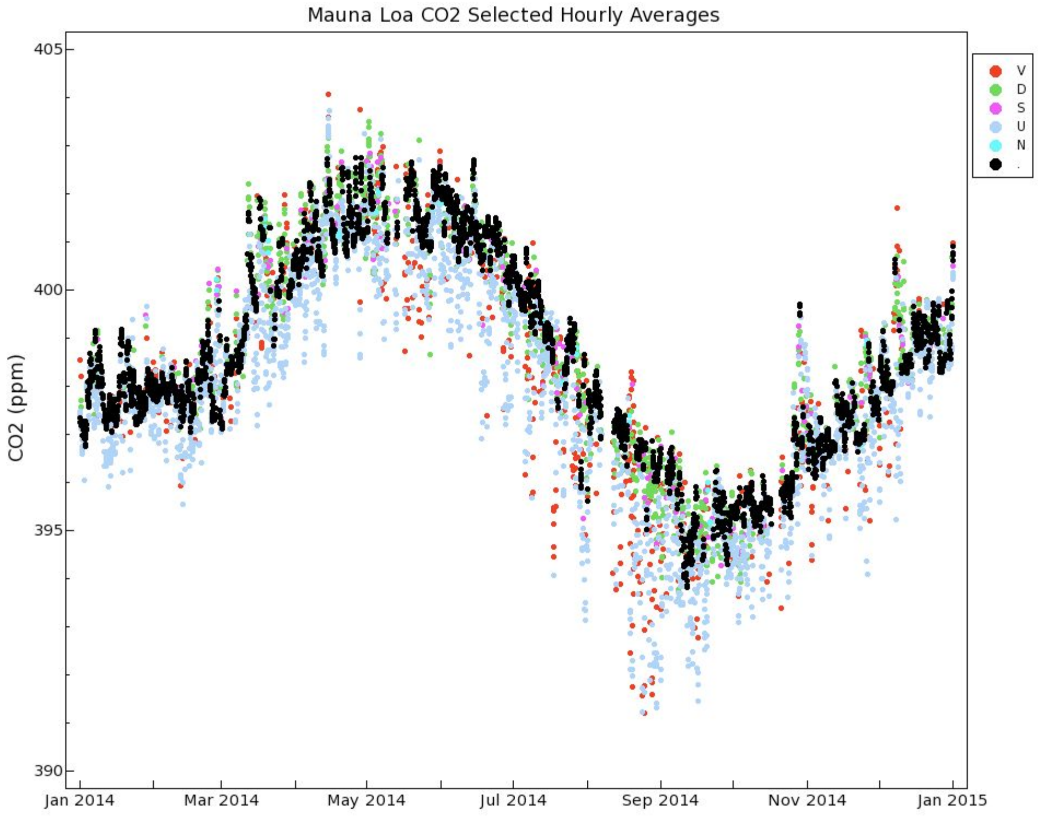
When is interpolation applicable? Useful?

- Missing data. For example, sparse data series in….

When is interpolation applicable? Useful?

- Missing data. For example, sparse data series in….

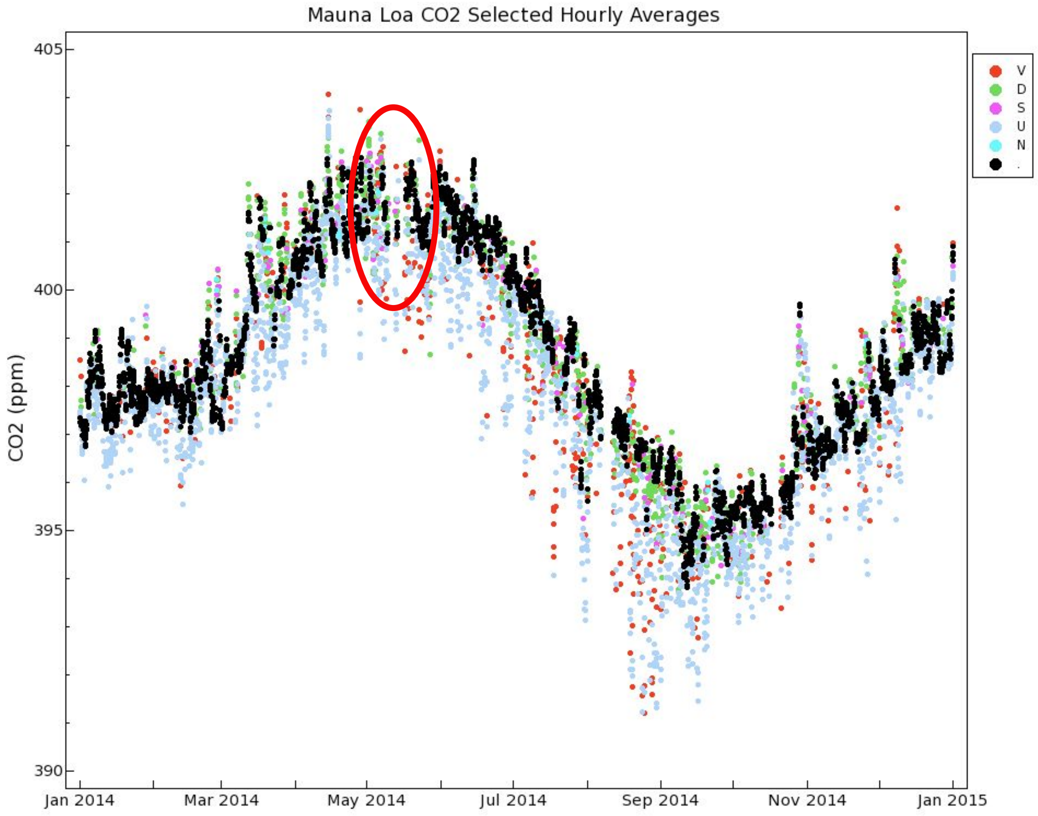
- Historic climate records
- Could often have missing, or corrupted data to interpolate through



When is interpolation applicable? Useful?

- Missing data. For example, sparse data series in….

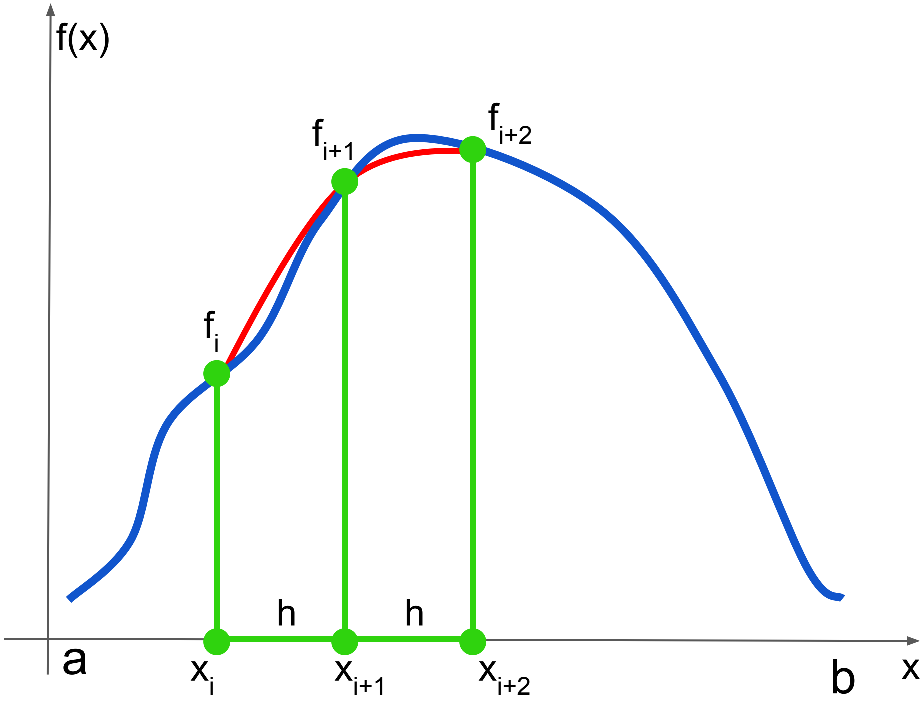
- Historic climate records
- Could often have missing, or corrupted data to interpolate through



When is interpolation applicable? Useful?

- Numerical integration…

- We will see in the next lectures how some interpolation schemes naturally allow for efficient numerical integration



Pitfalls in interpolation?

- There are subjective decisions in the type of adopted interpolation schemes.
- Need to consider your application, known biases in observations and how the error in interpolating some underlying behaviour affects your end goal

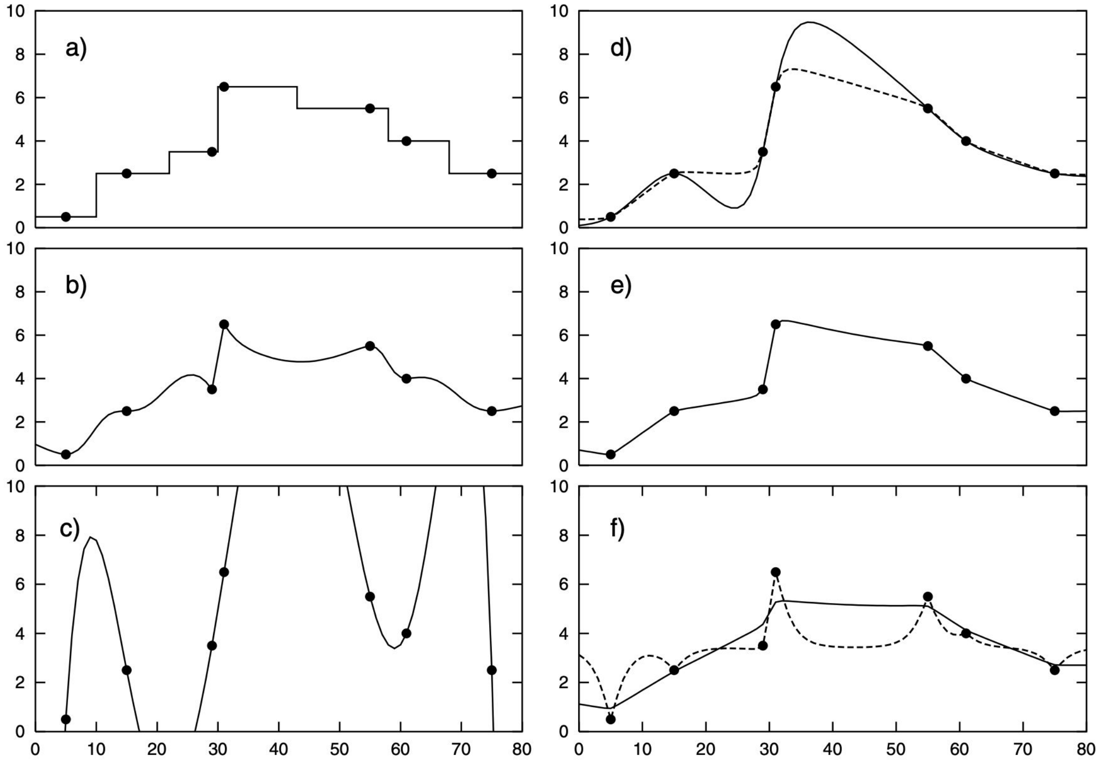
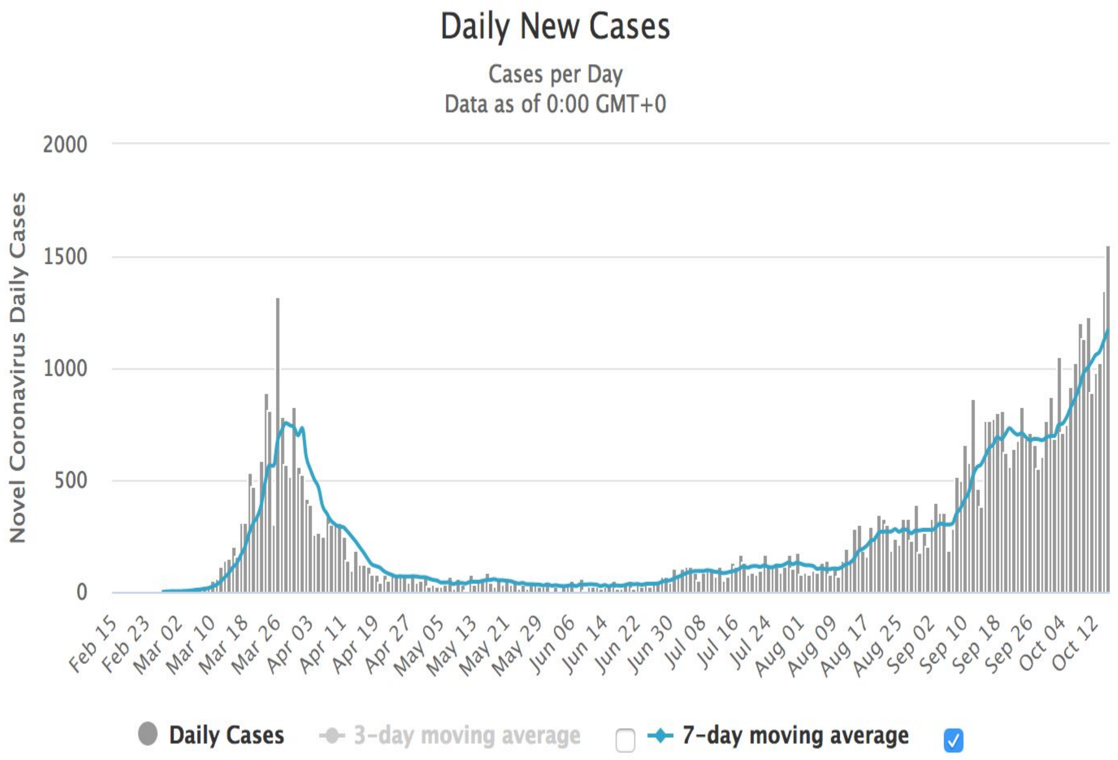


Figure 2.4. Comparison of interpolation methods for a one-dimensional example: (a) Thiessen method; (b) inverse distance squared; (c) example of overfitting using a sixth-order polynomial; (d) thin plate splines with different tension parameters; (e) kriging (zero nugget, large range); (f) kriging (solid line: large nugget, large range; dashed line: zero nugget, very short range).

What about approximations?

- Running averages, confidence intervals can give us a sense of the general trend without reproducing exact values
- For example daily case rates of infections
- Force of infection often specified in terms of per-capita infection rate over a 7 day window



Distinction from other tools

- Important to keep in mind that interpolation and approximations are **non-physical**, and subject to some arbitrary choices

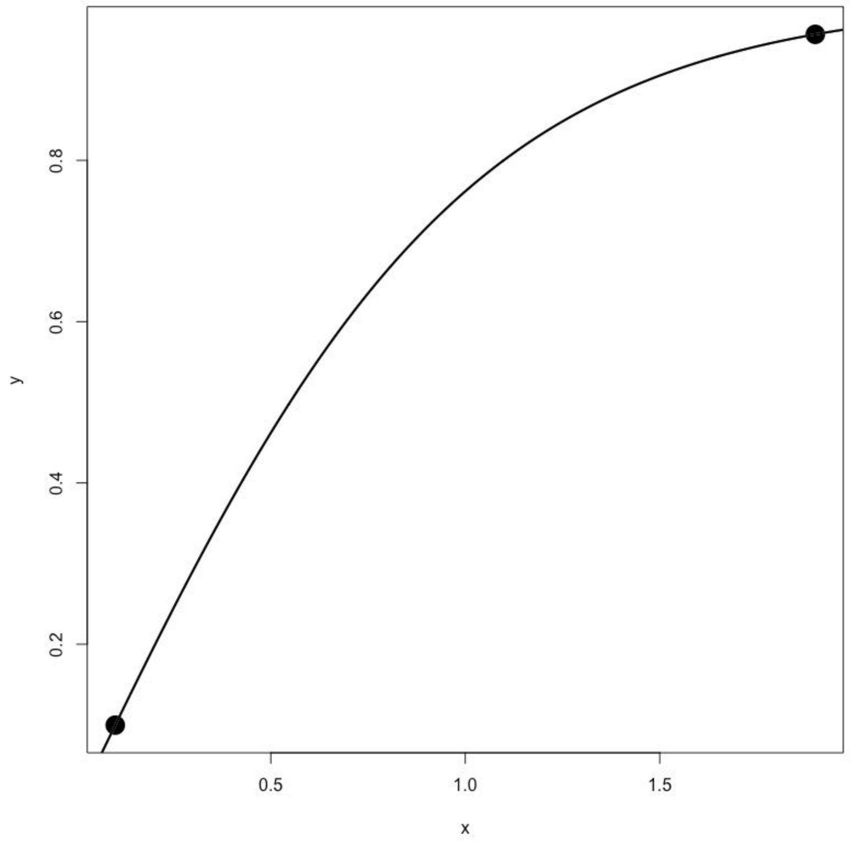
- They are fundamentally **different than fitting models** (analytic, numerical) to data (later lectures will cover this)

- In those cases you are using physical models to infer something from the dataset, and try to ascertain which model parameters are reproducing the data well.

- The tools we will discuss today are either numerically representing the data in places where your observations/simulations don't exist, or refining model grids in a way that can help you further analyze or characterize a problem.

Interpolation Methods:
Linear interpolation

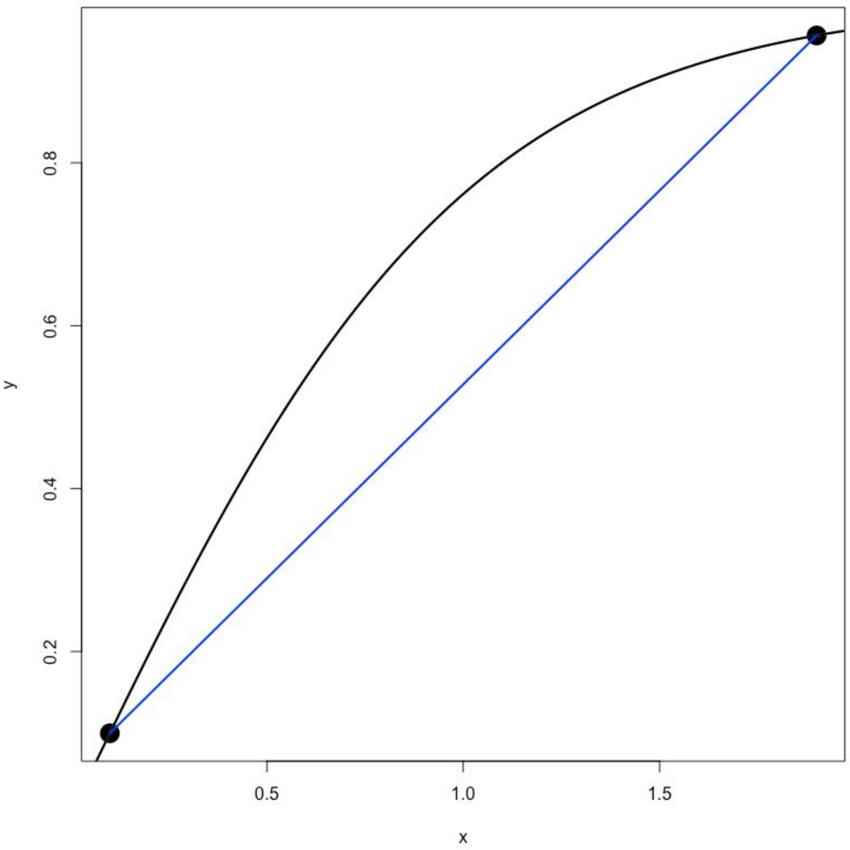
- Simplest form to interpolate a function between two end points will be to create a linear fit.



Interpolation Methods:
Linear interpolation

- Simplest form to interpolate a function between two end points will be to create a linear fit.
- Value at arbitrary *x* is:

$$y \equiv f(x) = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1$$

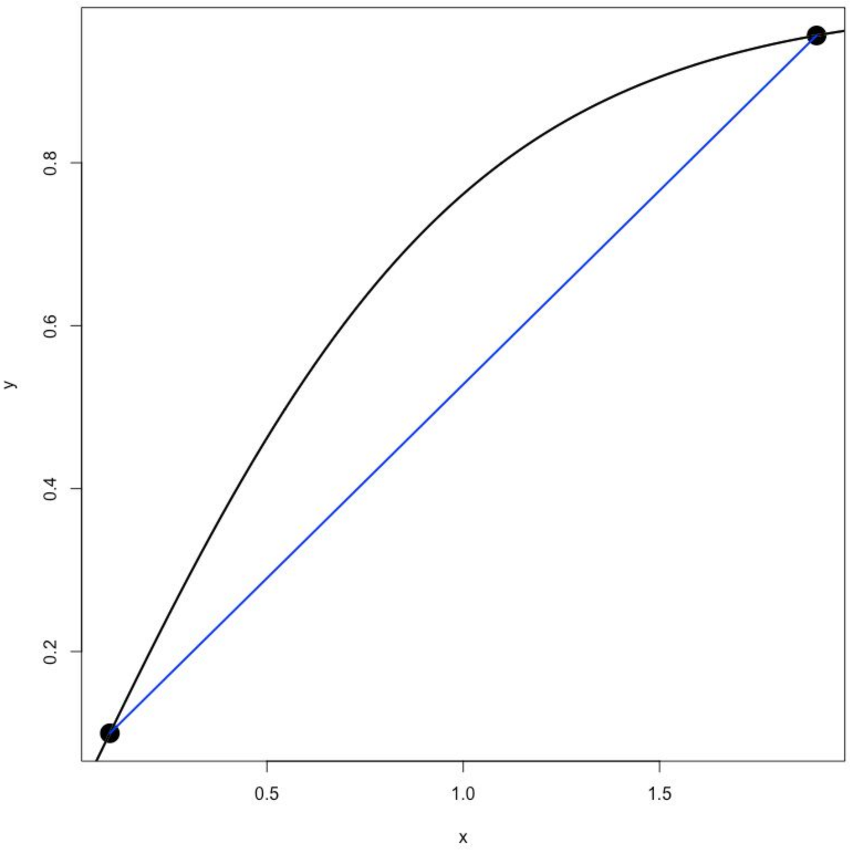


Interpolation Methods:
Linear interpolation

- Simplest form to interpolate a function between two end points will be to create a linear fit.
- Value at arbitrary *x* is:

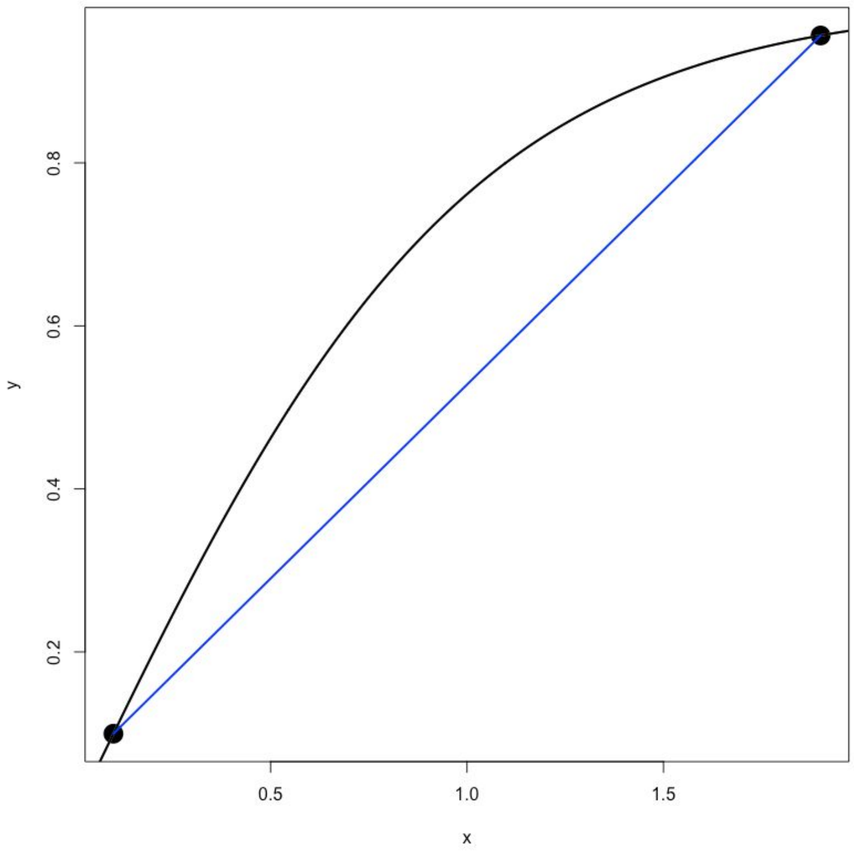
$$y \equiv f(x) = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1$$

- Regardless of the form of *f(x)*, the error will approximately scale of order $(x_{i+1}-x_i)^2$



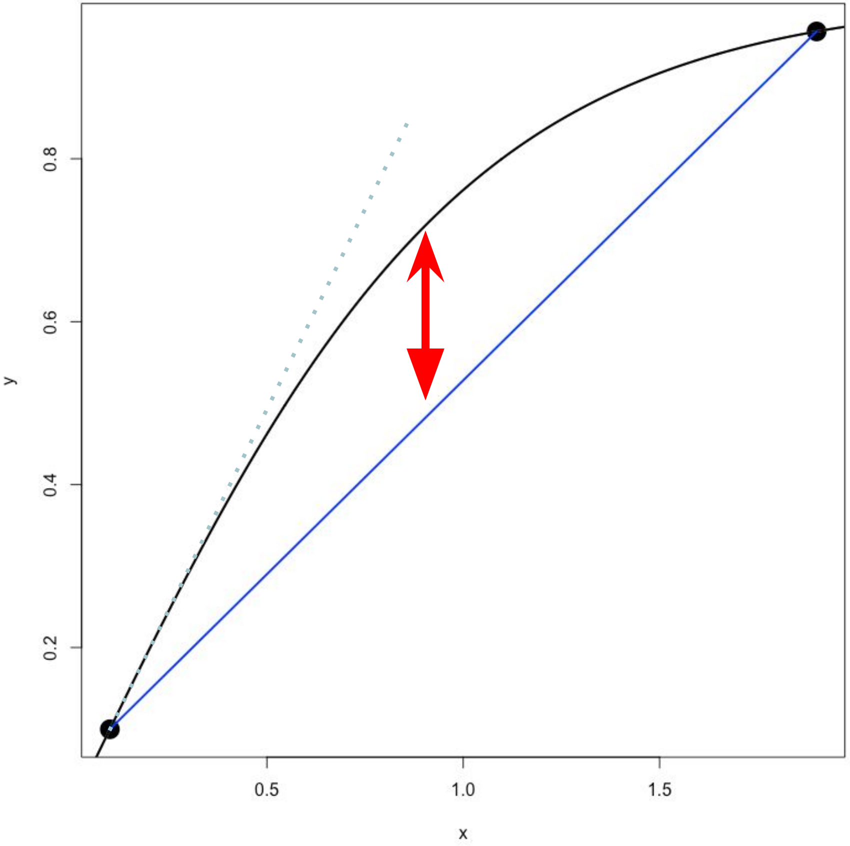
Interpolation Methods:
Polynomial interpolation

- Higher accuracy than linear, but requires a small interval.



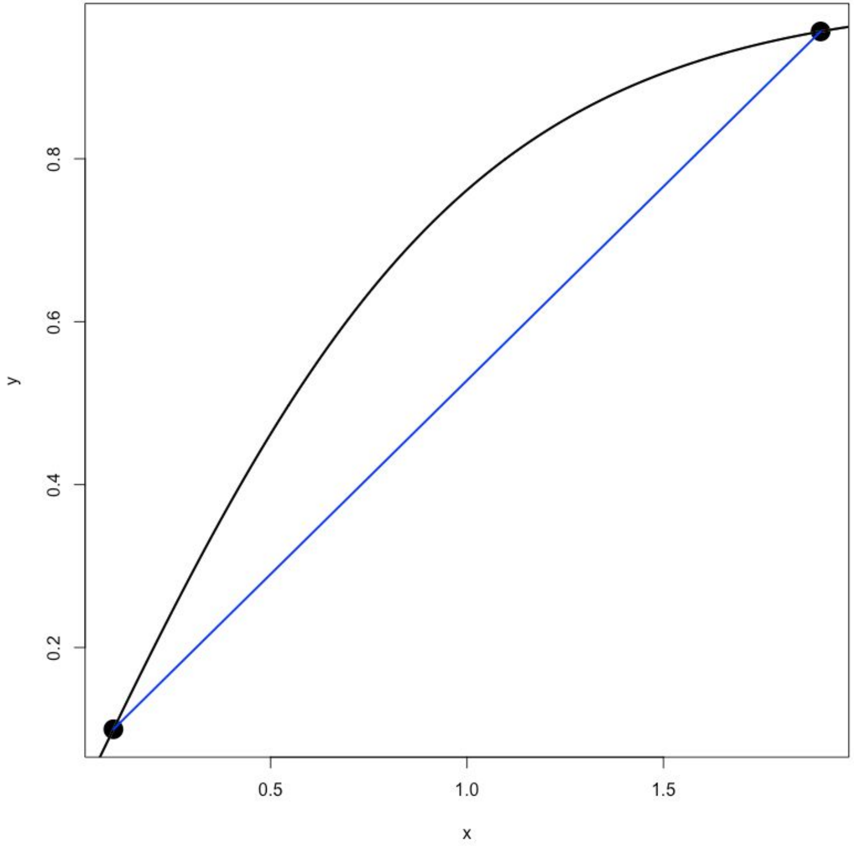
Interpolation Methods:
Polynomial interpolation

- Higher accuracy than linear, but requires a small interval
- This can potentially grow computation time



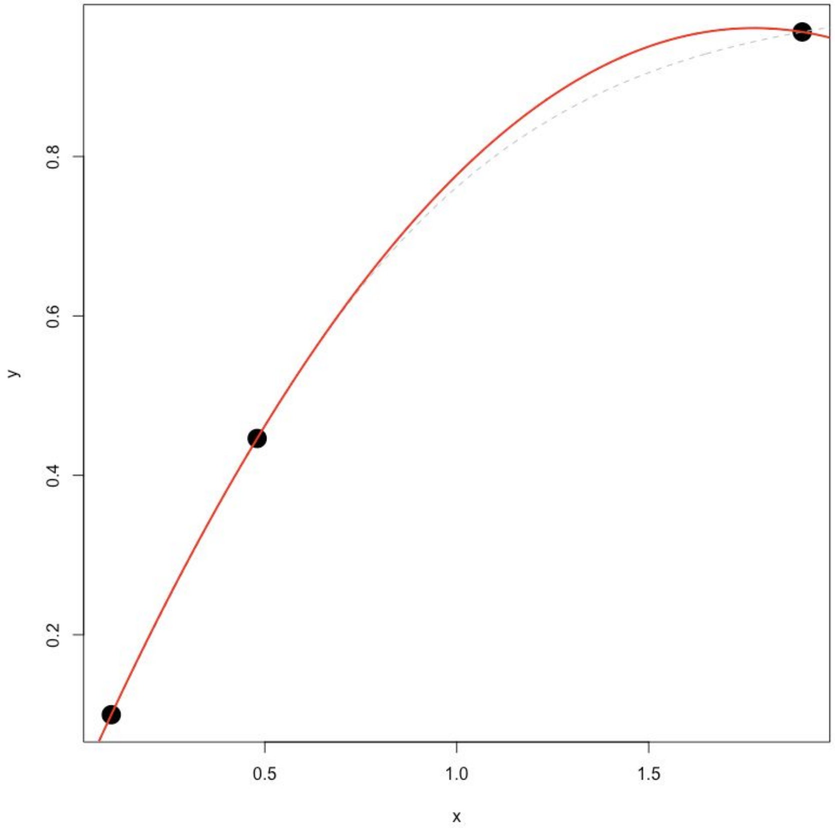
Interpolation Methods:
Polynomial interpolation

- Higher accuracy than linear, but requires a small interval
- This can potentially grow computation time
- Consider instead a polynomial function which passes through several points.



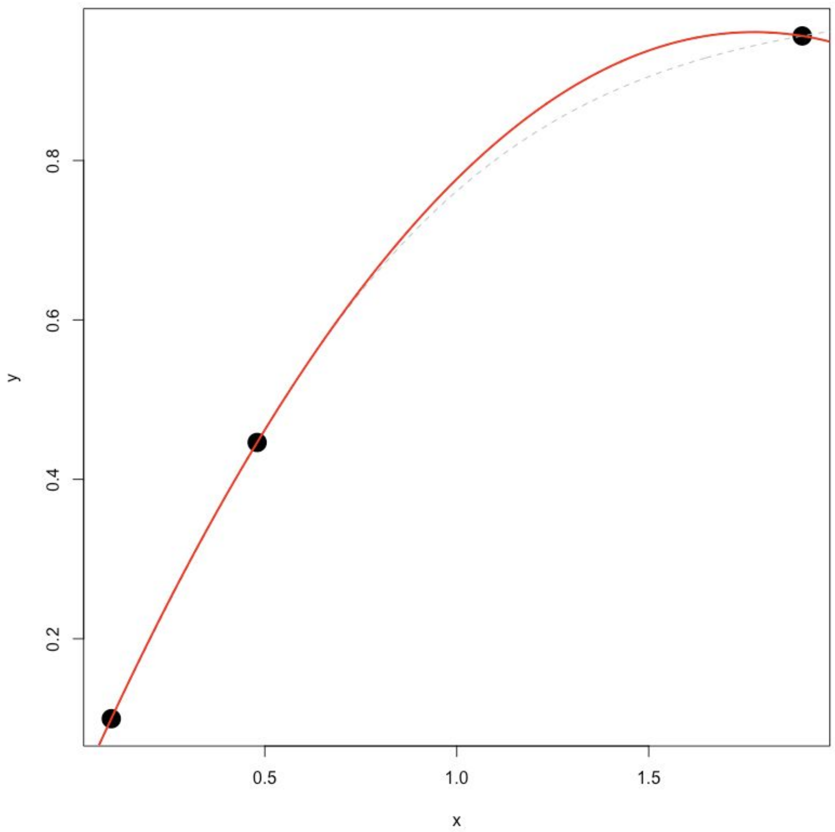
Interpolation Methods:
Polynomial interpolation

- Simplest extension would be quadratic, however let us look at general forms for polynomial interpolation.



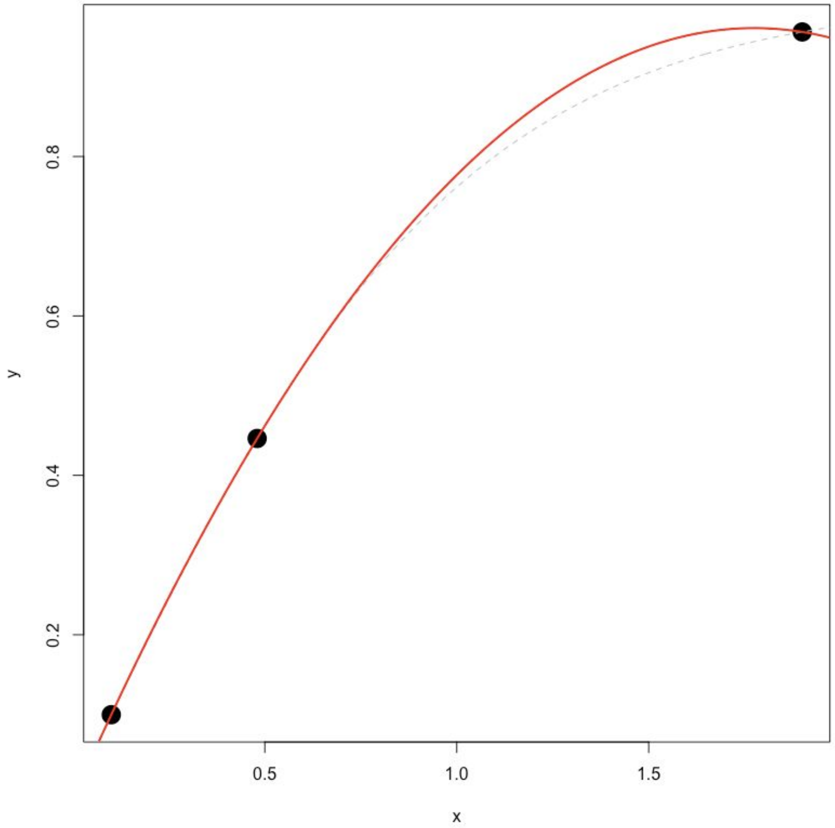
Interpolation Methods:
Polynomial interpolation

- Simplest extension would be quadratic, however let us look at general forms for polynomial interpolation.
- Allows you to interpolate generally, for arbitrary number of data points (x_i, y_i)



Interpolation Methods:
Polynomial interpolation

- Simplest extension would be quadratic, however let us look at general forms for polynomial interpolation.
- Allows you to interpolate generally, for arbitrary number of data points (x_i, y_i)
- Does not require evenly spaced data.

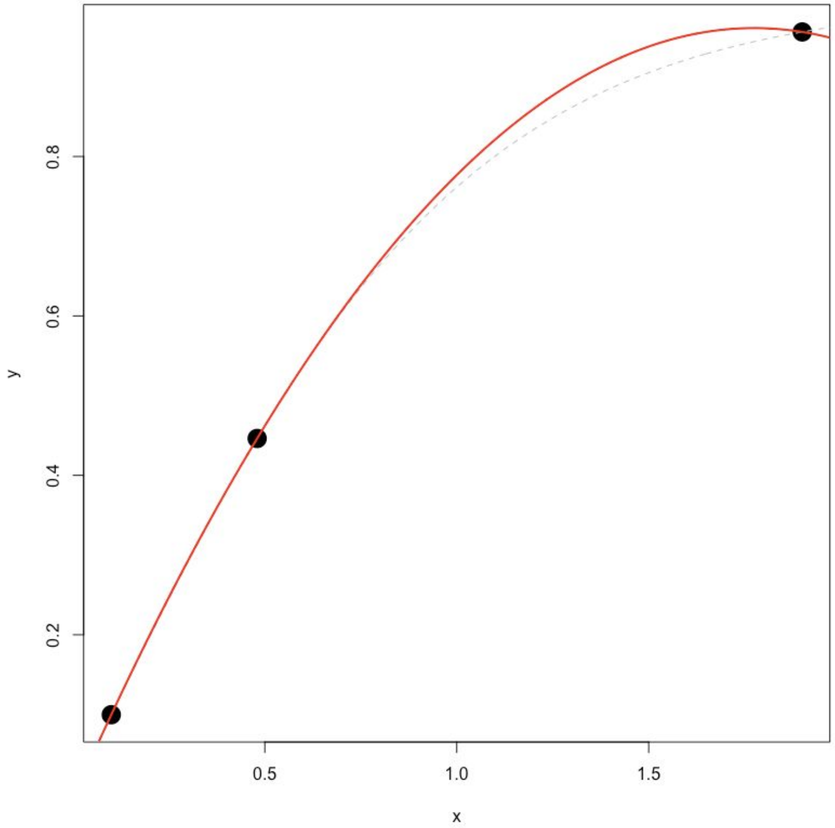


Interpolation Methods:
Polynomial interpolation

- A common and flexible implementation is **Lagrange interpolation**
- Basic idea: with *n* data points, can construct polynomial of order *n-1*

$$y \equiv f(x) = c_0 x^2 + c_1 x + c_2$$

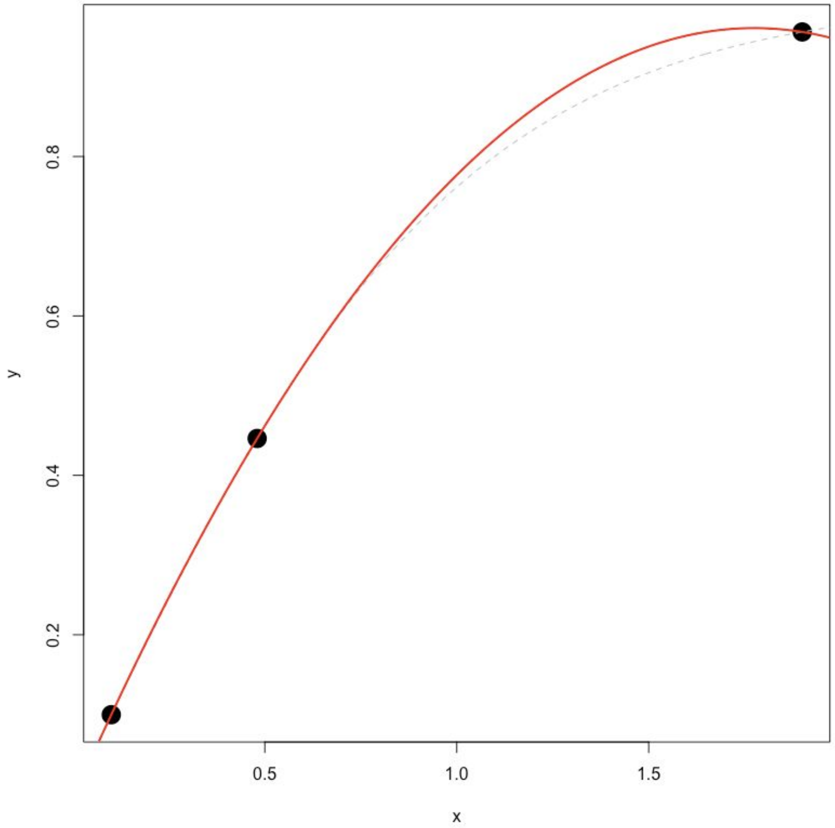
- System of equations can be solved for the *coefficients* c_i



Interpolation Methods:
Polynomial interpolation

- Lagrange generalized a form of solutions so that extension to larger *n* is not unwieldy
- Basis functions L_i represent the *coefficients* in terms of the data x_i

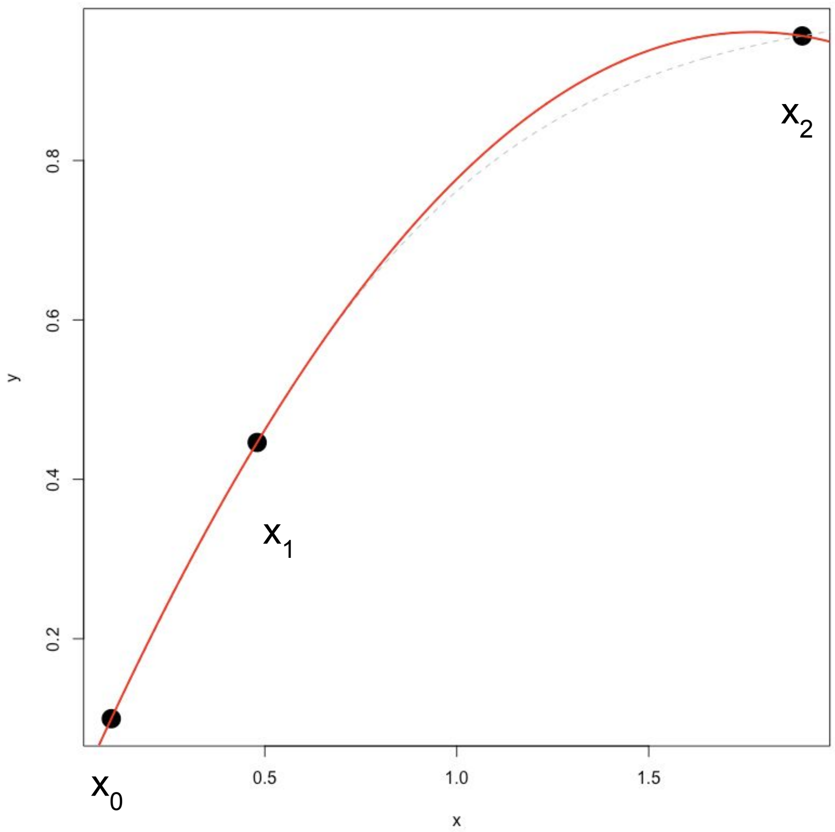
$$L_i(x) = \prod_{i=0, j \neq i}^{n-1} \frac{x - x_j}{x_i - x_j}$$



Interpolation Methods:
Polynomial interpolation

- For example with *n*=3 data points:

$$L_i(x) = \prod_{i=0, j \neq i}^{n-1} \frac{x - x_j}{x_i - x_j}$$



Interpolation Methods:
Polynomial interpolation

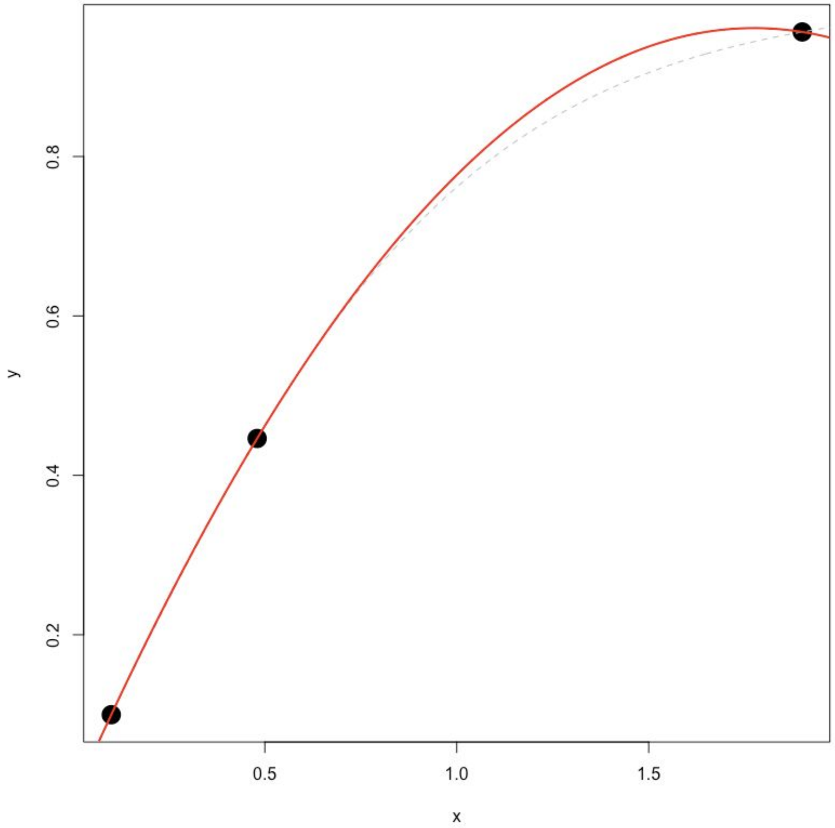
- For example with *n*=3 data points:

$$L_i(x) = \prod_{i=0, j\neq i}^{n-1} \frac{x - x_j}{x_i - x_j}$$

(x-x1)/(x0-x1)*(x-x2)/(x0-x2)

(x-x0)/(x1-x0)*(x-x2)/(x1-x2)

(x-x0)/(x2-x0)*(x-x1)/(x2-x1)



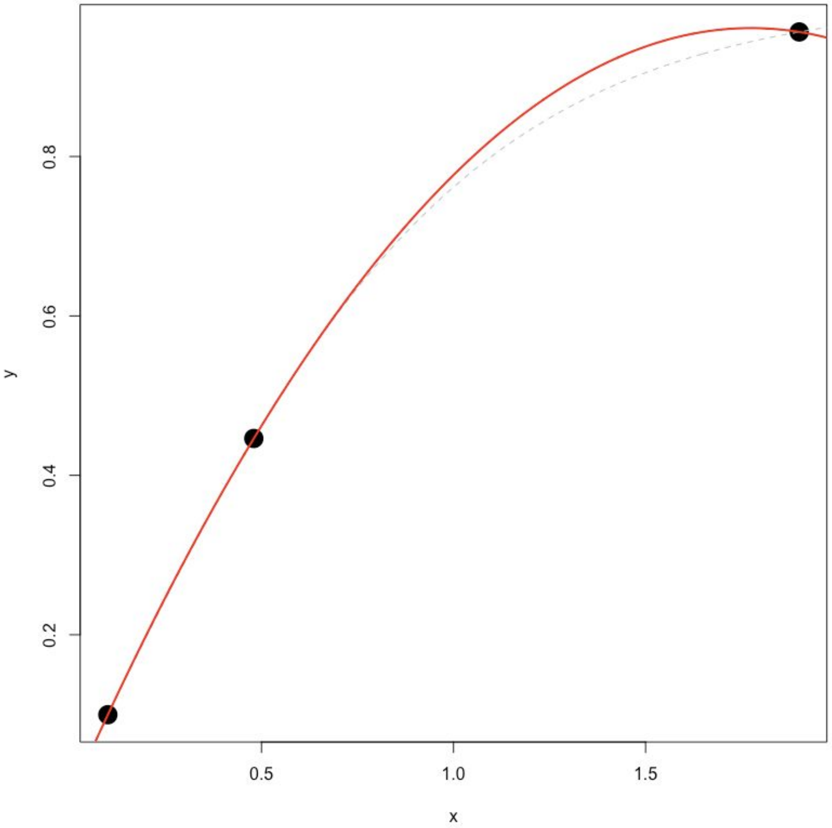
Interpolation Methods:
Polynomial interpolation

- For example with *n*=3 data points:

$$L_0(x) = \frac{x - x_1}{x_0 - x_1} \times \frac{x - x_2}{x_0 - x_2}$$

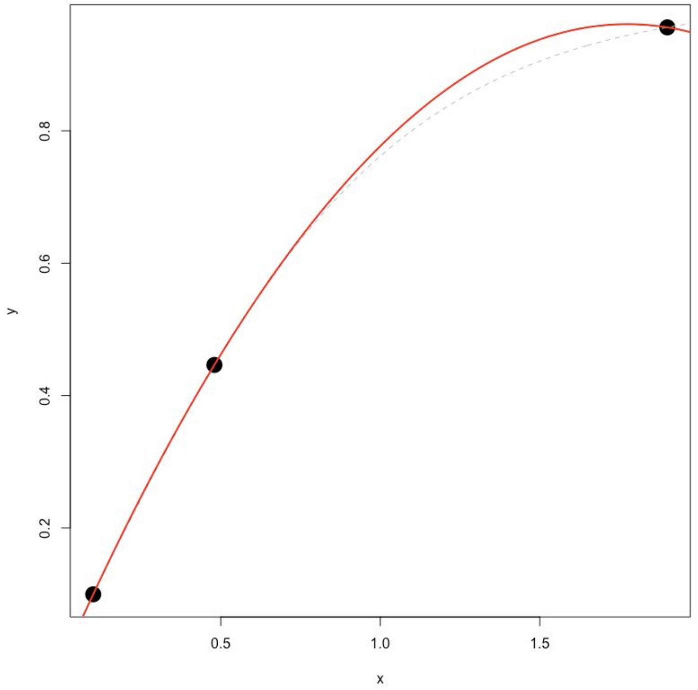
$$L_1(x) = \frac{x - x_0}{x_1 - x_0} \times \frac{x - x_2}{x_1 - x_2}$$

$$L_2(x) = \frac{x - x_0}{x_2 - x_0} \times \frac{x - x_1}{x_2 - x_1}$$



Interpolation Methods:
Polynomial interpolation

$$p(x) = \sum_{i=0}^{n-1} L_i(x) f_i(x)$$

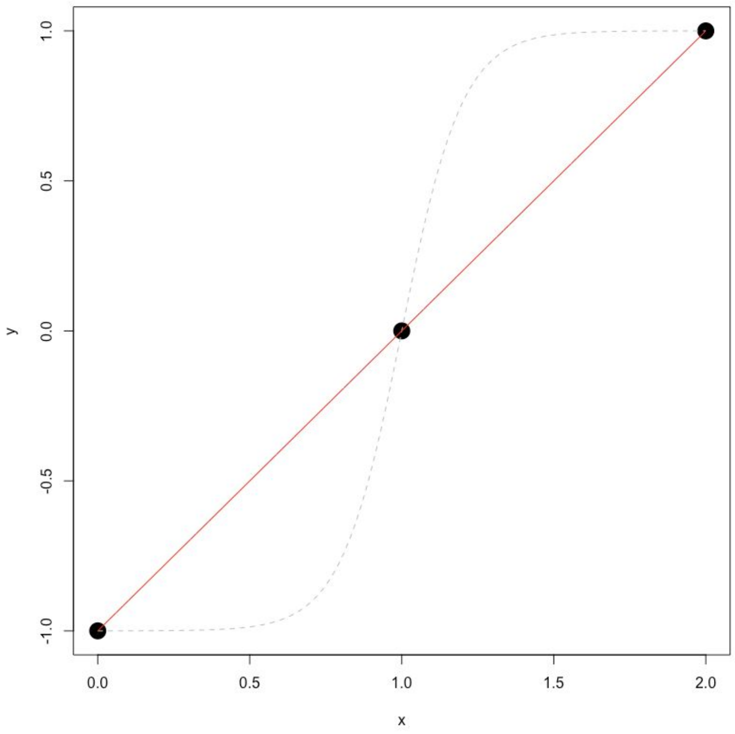


If equal spacings of Δx then this simplifies to:

$$p(x) = \frac{(x-x_1)(x-x_2)}{2\Delta x^2} f_0 - \frac{(x-x_0)(x-x_2)}{\Delta x^2} f_1 + \frac{(x-x_0)(x-x_1)}{2\Delta x^2} f_2$$

Interpolation Methods:
Polynomial interpolation

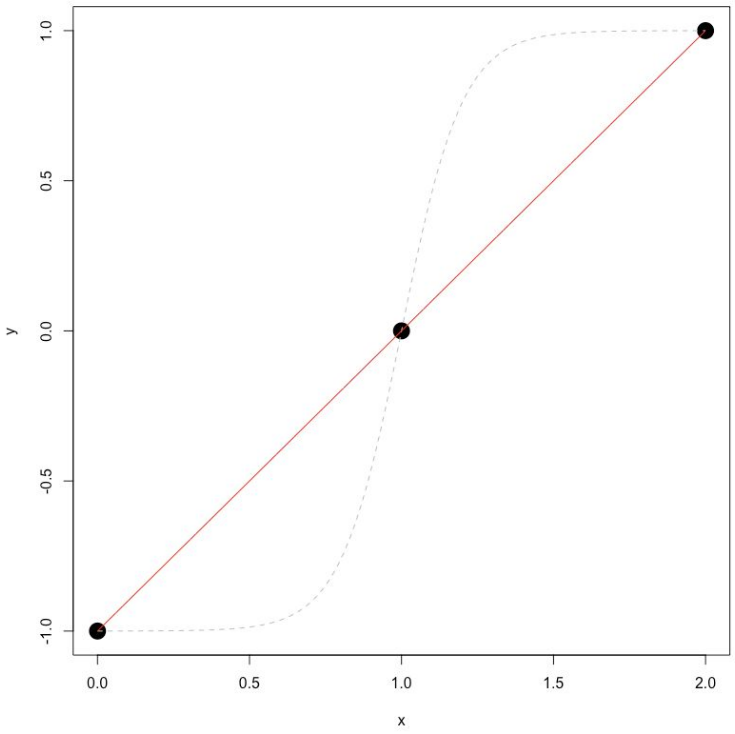
- **Pros**:
  - Easy to code and/or built-in to most languages (`R, python, IDL`)
  - Always finds a solution



Interpolation Methods:
Polynomial interpolation

- **Pros**:
  - Easy to code and/or built-in to most languages (`R, python, IDL`)
  - Always finds a solution

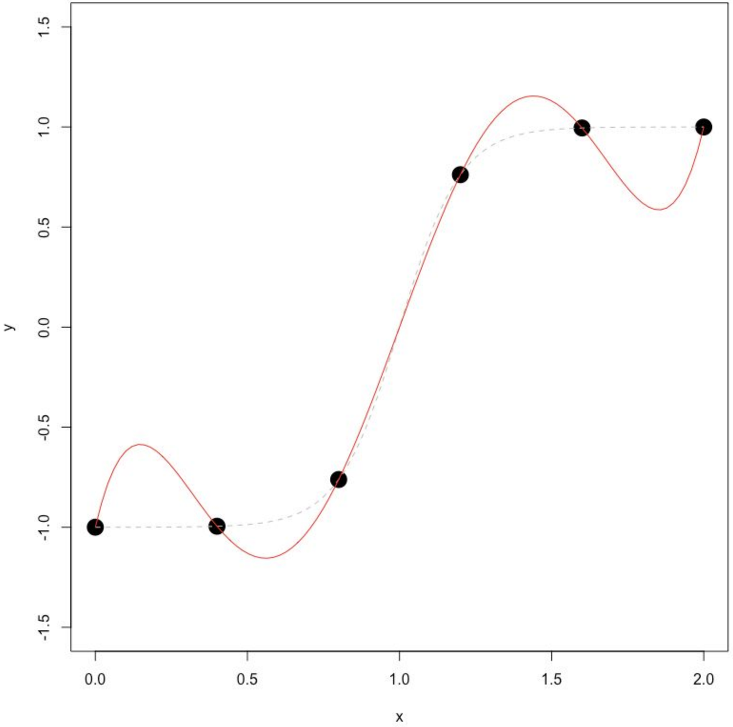
- **Cons**:
  - As the number of points rise, some spacings will produce oscillations with large deviations



Interpolation Methods:
Polynomial interpolation

- **Pros**:
  - Easy to code and/or built-in to most languages (`R, python, IDL`)
  - Always finds a solution

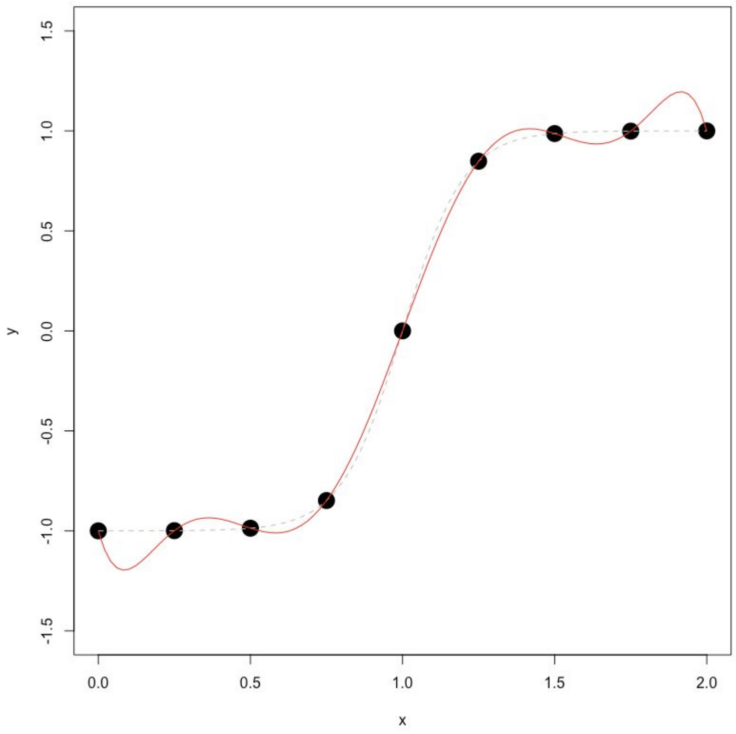
- **Cons**:
  - As the number of points rise, some spacings will produce oscillations with large deviations



Interpolation Methods:
Polynomial interpolation

- **Pros**:
  - Easy to code and/or built-in to most languages (`R, python, IDL`)
  - Always finds a solution

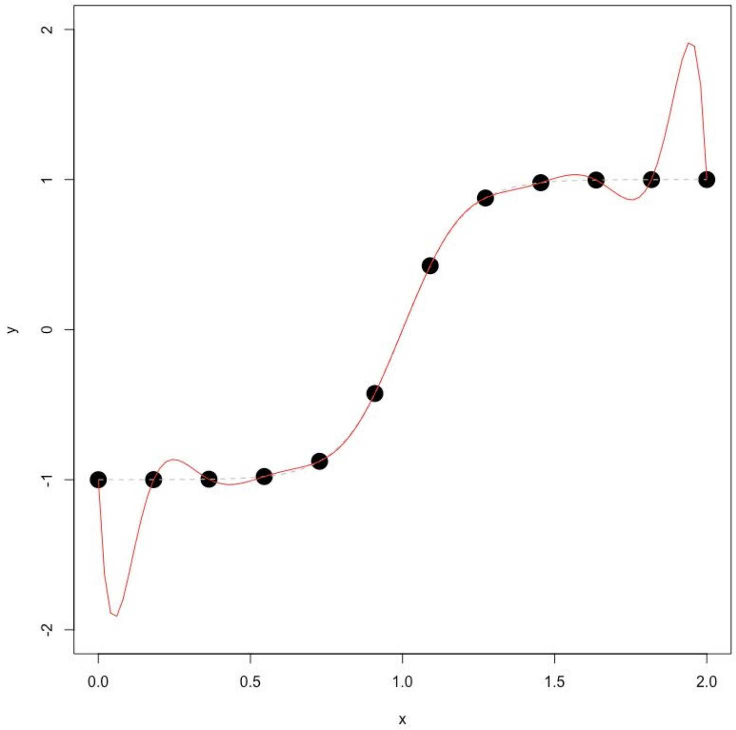
- **Cons**:
  - As the number of points rise, some spacings will produce oscillations with large deviations



Interpolation Methods:
Polynomial interpolation

- **Pros**:
  - Easy to code up, built-in to most languages (`R, python, IDL`)
  - Always finds a solution

- **Cons**:
  - As the number of points rise, some spacings will produce oscillations with large deviations
- Can be overcome if you can tailor the spacing of the points (*Chebyshev polynomial spacing*))

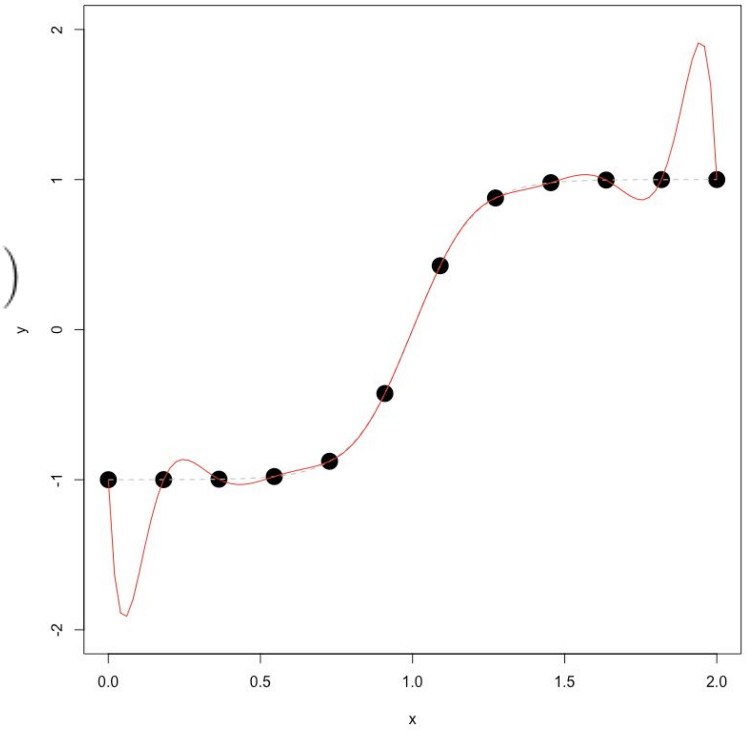


Interpolation Methods:
Aside: Assessing errors

- Formally consider the error of the interpolating polynomial bounded by:

$$f(x) - p(x) = \frac{f^{(n+1)}(u)}{(n+1)!}(x - x_0)\ldots.(x - x_n)$$

- Numerator is $(n+1)^{th}$ derivative of f at some point u within the interval of interest [at most over, $(a,b) = (x_0,x_n)$]



Interpolation Methods:
Aside: Assessing errors

- Often care about the maximum over an interval, so can consider:

$$E(x) = |f(x) - p(x)| \leq \max_{[a,b]} \left| \frac{f^{(n+1)}(u)}{(n+1)!} \right| \max_{[a,b]} \left| \prod_{i=0}^{n} (x - x_i) \right|$$

- Can revisit the statement about why linear approximation error goes as order ~ $(\Delta x)^2$

Interpolation Methods:
Aside: Assessing errors

- Often care about the maximum over an interval, so can consider:

$$E(x) = |f(x) - p(x)| \leq \max_{[a,b]} \left| \frac{f^{(n+1)}(u)}{(n+1)!} \right| \max_{[a,b]} \left| \prod_{i=0}^{n} (x - x_i) \right|$$

- In linear approximation case, for an unknown true function *f*, the approximation between x_i and x_{i+1} depends on the maximal curvature (f'') and the product of $|(x-x_0)(x-x_1)|$

Interpolation Methods:
Aside: Assessing errors

- Often care about the maximum over an interval, so can consider:

$$E(x) = |f(x) - p(x)| \leq \max_{[a,b]} \left| \frac{f^{(n+1)}(u)}{(n+1)!} \right| \max_{[a,b]} \left| \prod_{i=0}^{n} (x - x_i) \right|$$

- **For <u>generic</u> interpolating functions we practically** would want to then:

Interpolation Methods:
Aside: Assessing errors

- Often care about the maximum over an interval, so can consider:

$$E(x) = |f(x) - p(x)| \leq \max_{[a,b]} \left| \frac{f^{(n+1)}(u)}{(n+1)!} \right| \max_{[a,b]} \left| \prod_{i=0}^{n} (x - x_i) \right|$$

- **For <u>generic</u> interpolating functions we practically** would want to then:
- - if possible estimate maximum of $f^{(n+1)}(a<u<b)$

Interpolation Methods:
Aside: Assessing errors

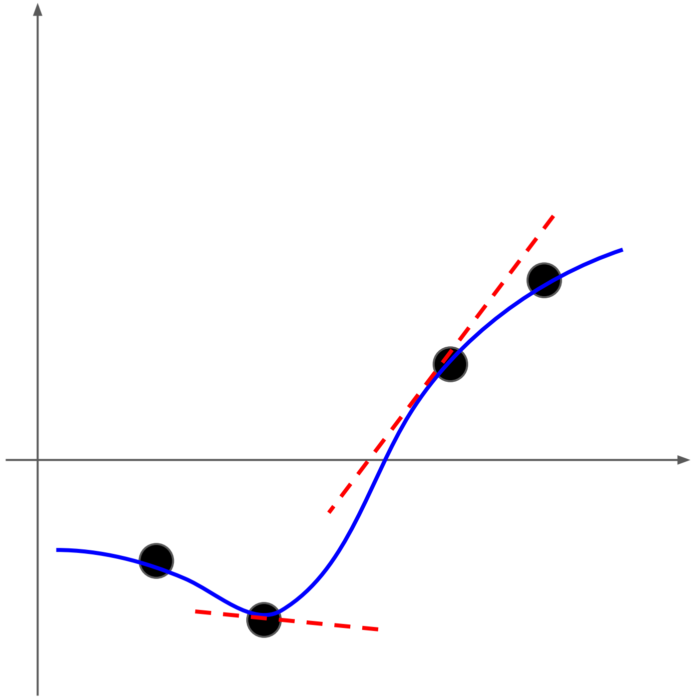
- Often care about the maximum over an interval, so can consider:

$$E(x) = |f(x) - p(x)| \leq \max_{[a,b]} \left| \frac{f^{(n+1)}(u)}{(n+1)!} \right| \max_{[a,b]} \left| \prod_{i=0}^{n} (x - x_i) \right|$$

- **For generic interpolating functions we practically** would want to then:
- - if possible estimate maximum of $f^{(n+1)}(a<u<b)$
- - compute maximum of $(x-x_0)(x-x_1)\ldots(x-x_n)$ over all $(a<x<b)$

Interpolation Methods:
Hermite interpolation

- Can also include information on local derivative to improve accuracy.

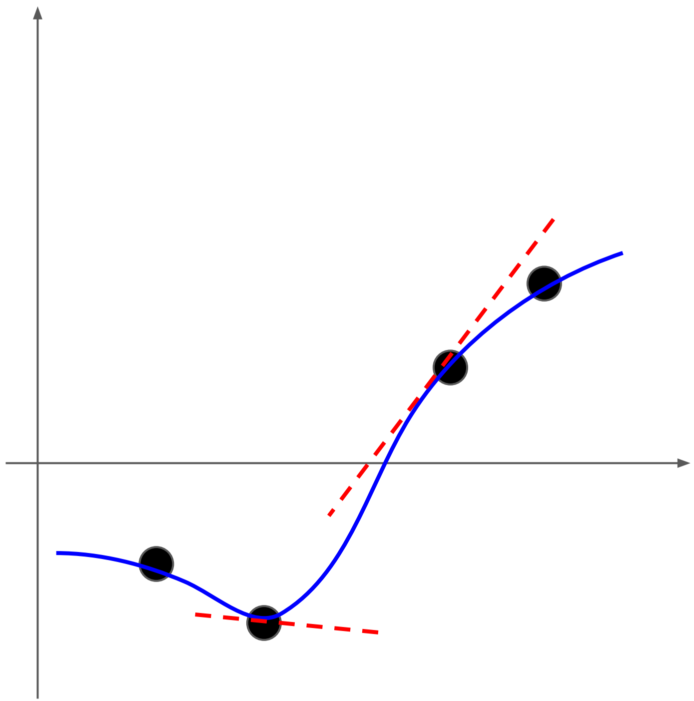


Interpolation Methods:
Hermite interpolation

- Can also include information on local derivative to improve accuracy.
- Polynomial of order *n* is:

$$f(x) = \sum_{i=0}^{n-1} A_i(x) f_i + \sum_{i=0}^{n-1} B_i(x) f_i'$$

- Computes new coefficients to use with local derivatives and derivatives of Lagrange coefficients



Interpolation Methods:
Hermite interpolation

- Can also include information on local derivative to improve accuracy.
- Polynomial of order *n* is:

$$f(x) = \sum_{i=0}^{n-1} A_i(x) f_i + \sum_{i=0}^{n-1} B_i(x) f_i'$$

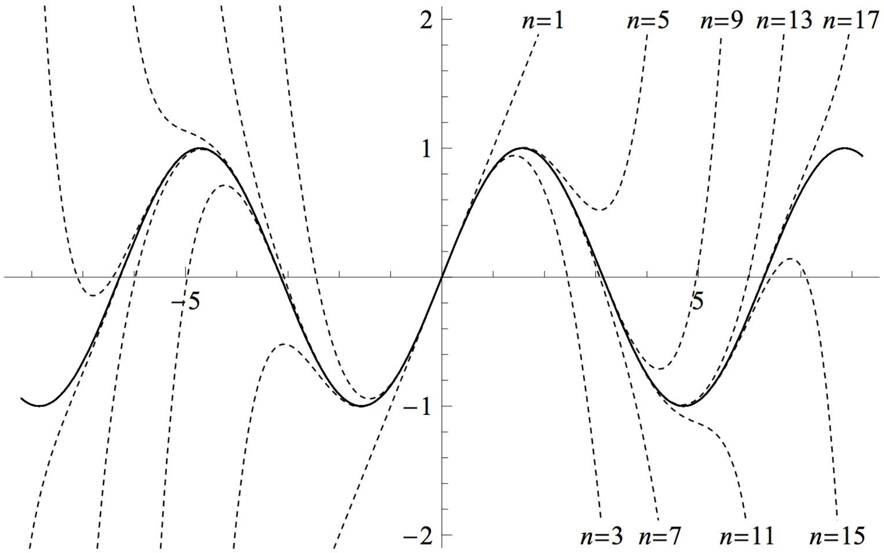
$$A_i(x) = \Big(1 - 2(x - x_i)L_i'(x_i)\Big) \times L_i^2(x)$$

$$B_i(x) = (x - x_i)L_i(x)^2$$

- Computes new coefficients to use with local derivative and derivative of Lagrange coefficients

Interpolation Methods:
Taylor interpolation

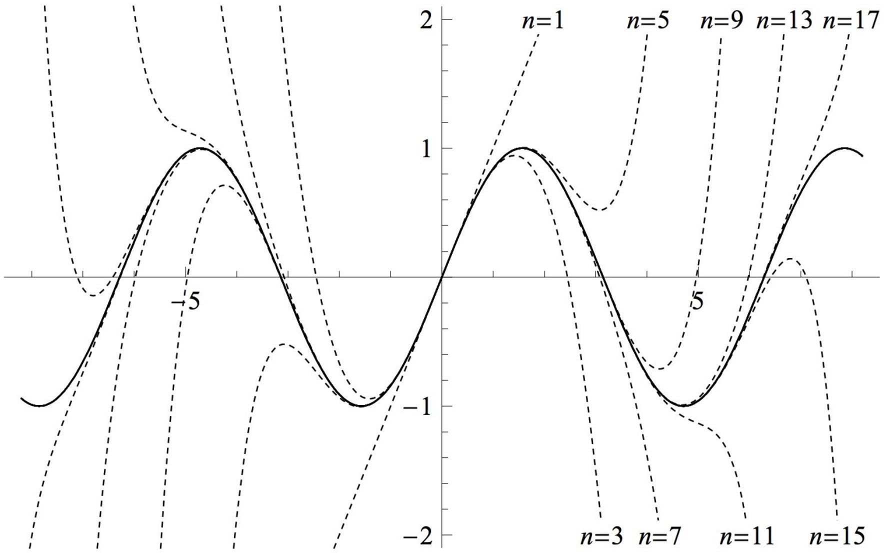
- Hermite polynomials match the function and its first derivative at a set of particular points



Taylor polynomials for different orders at the origin of y = sin(x)

Interpolation Methods:
Taylor interpolation

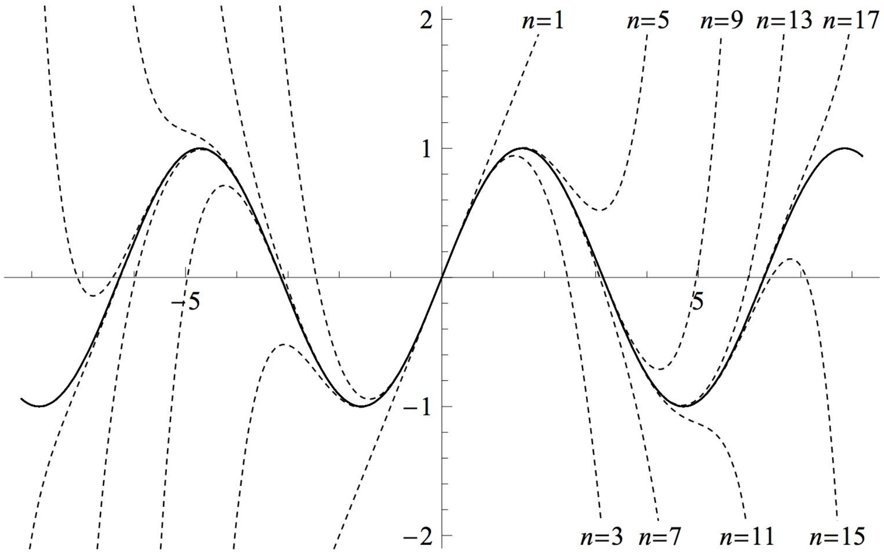
- Hermite polynomials match the function and its first derivative at a set of particular points
- Taylor polynomial methods extend this to make the j^{th} derivative match at the same point x_i
  [e.g., $p^{(j)}(x_i) = f^{(j)}(x_i)$]



Taylor polynomials for different orders at the origin of y = sin(x)

Interpolation Methods:
Taylor interpolation

$$p_j(f;a)(x) = \sum_{j=0}^{m} \frac{f^{(j)}(a)}{j!}(x-a)^j$$

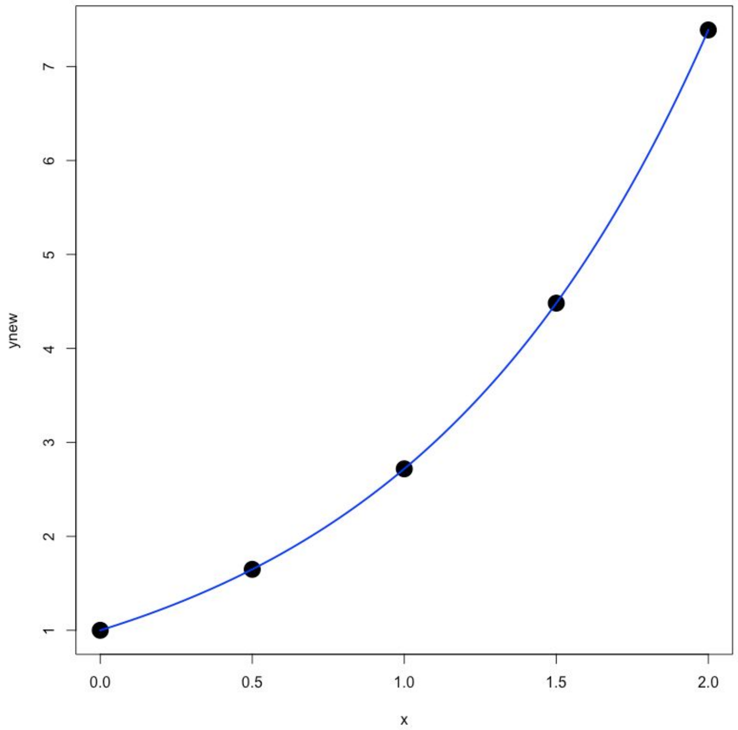


Taylor polynomials for different orders at the origin of y = sin(x)

Interpolation Methods:
Taylor interpolation

- Simple Example: Approximate
  $y = e^x$ expanding about $a=0$

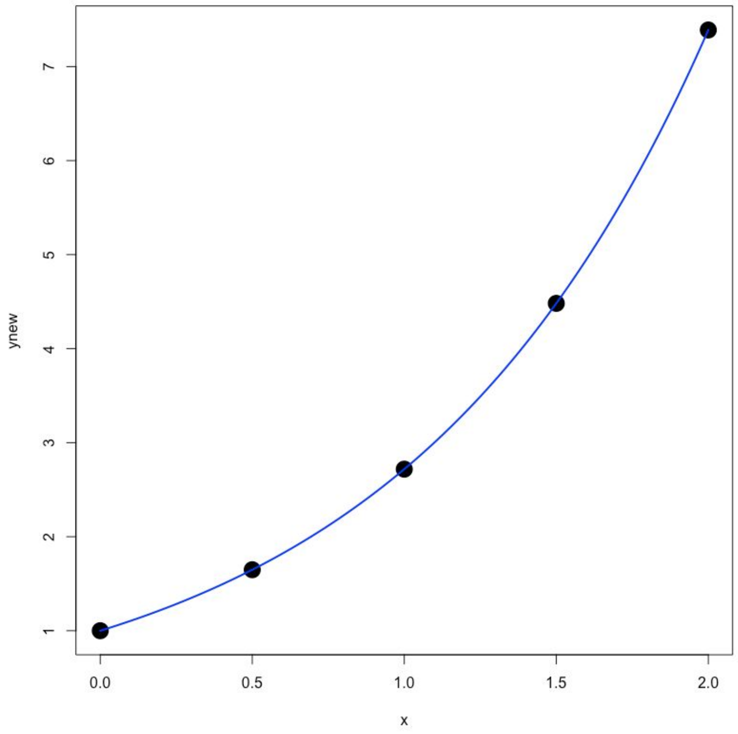
$$p_j(f;a)(x) = \sum_{j=0}^{m} \frac{f^{(j)}(a)}{j!}(x-a)^j$$



Interpolation Methods:
Taylor interpolation

- Error bound in Taylor polynomial typically cares about what order (*j*) you are using for the approximation

$$E(x) = p(x) - f(x) = \frac{(x-a)^{j+1}}{(j+1)!} f^{(j+1)}(u)$$

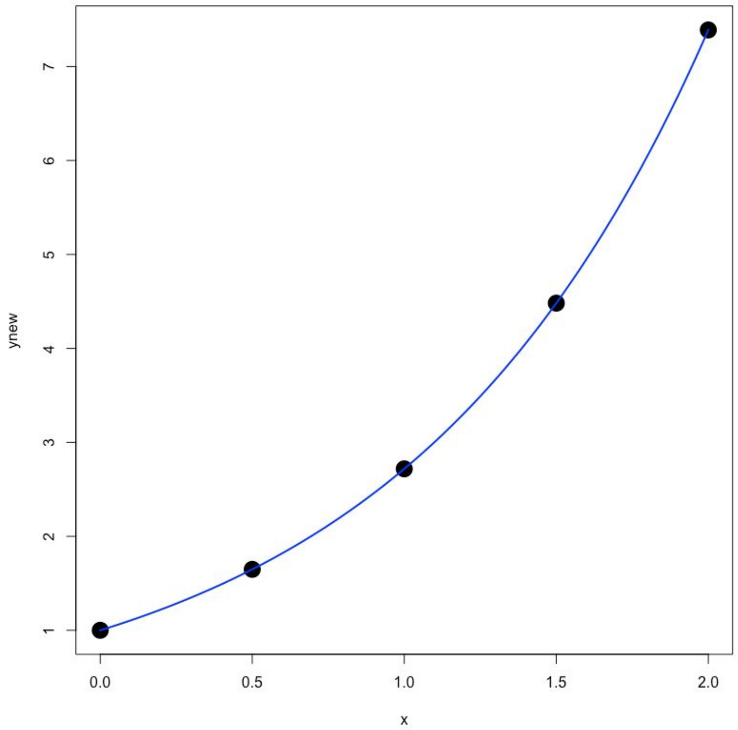


Interpolation Methods:
Taylor interpolation

- Error bound in Taylor polynomial typically cares about what order (*j*) you are using for the approximation

$$E(x) = p(x) - f(x) = \frac{(x-a)^{j+1}}{(j+1)!} f^{(j+1)}(u)$$

- Continuing our example - how precise do we approximate $f = e^x$ at $x=1$ with $p_5(f;a=0)(x)$?

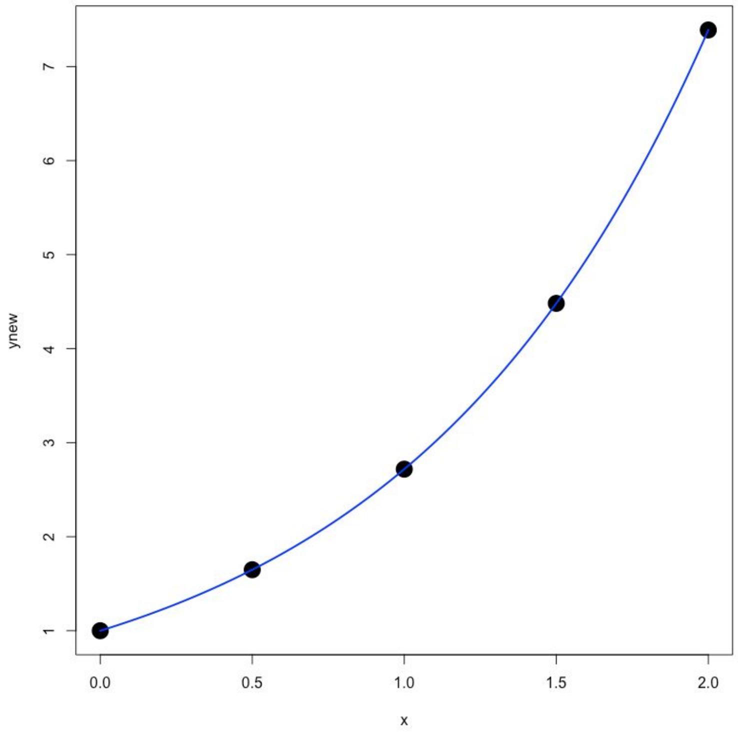


Interpolation Methods:
Taylor interpolation

- First term reduces to 1/(6!)

(for *a=0, x=1, j=5*)

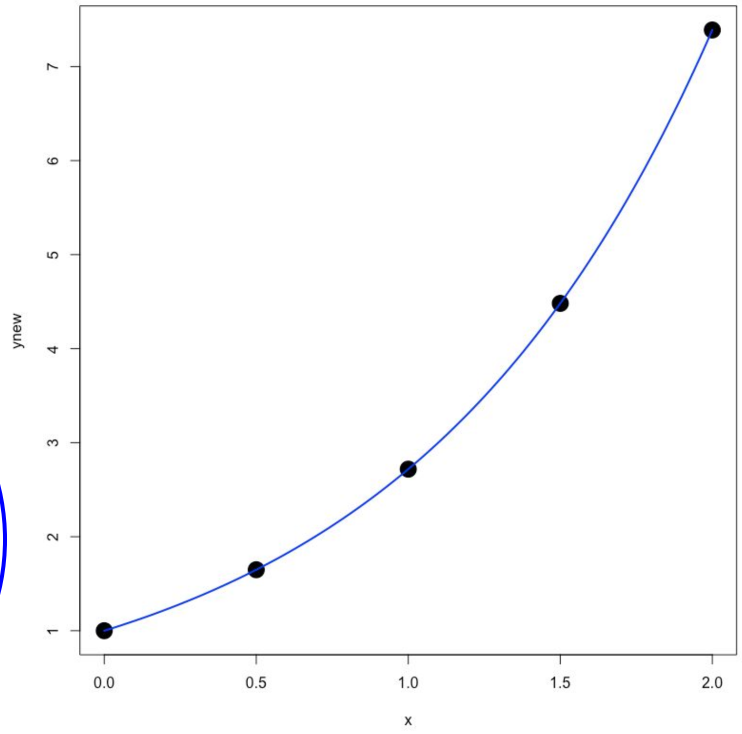
$$E(x) = \frac{(x-a)^{j+1}}{(j+1)!} f^{(j+1)}(u)$$



Interpolation Methods:
Taylor interpolation

- Second term is asking what the *j*th derivative of the function is at some value *u*.

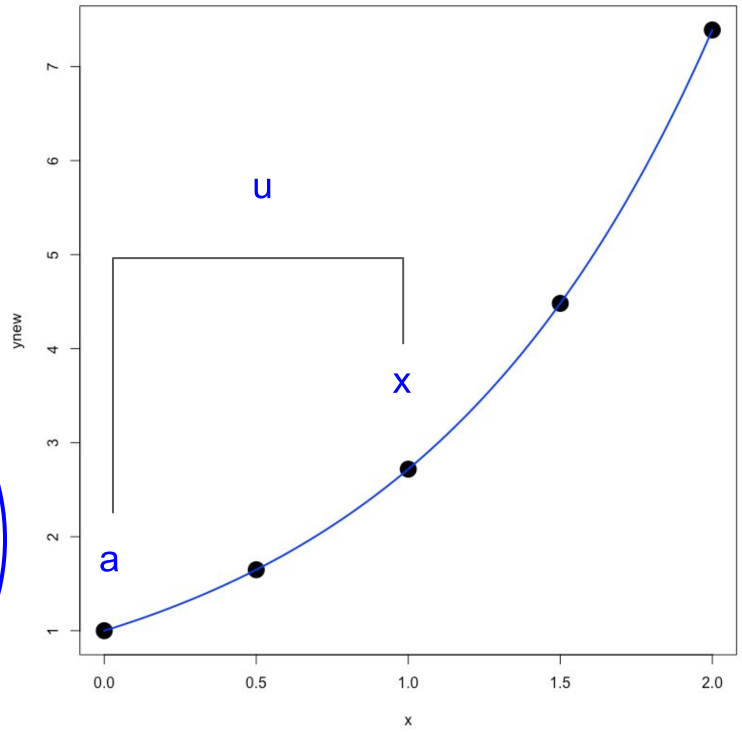
$$E(x) = \frac{(x-a)^{j+1}}{(j+1)!} f^{(j+1)}(u)$$



Interpolation Methods:
Taylor interpolation

- Second term is asking what the j^{th} derivative of the function is at some value u.
- We know that $a < u < x$ in this example

$$E(x) = \frac{(x-a)^{j+1}}{(j+1)!} f^{(j+1)}(u)$$



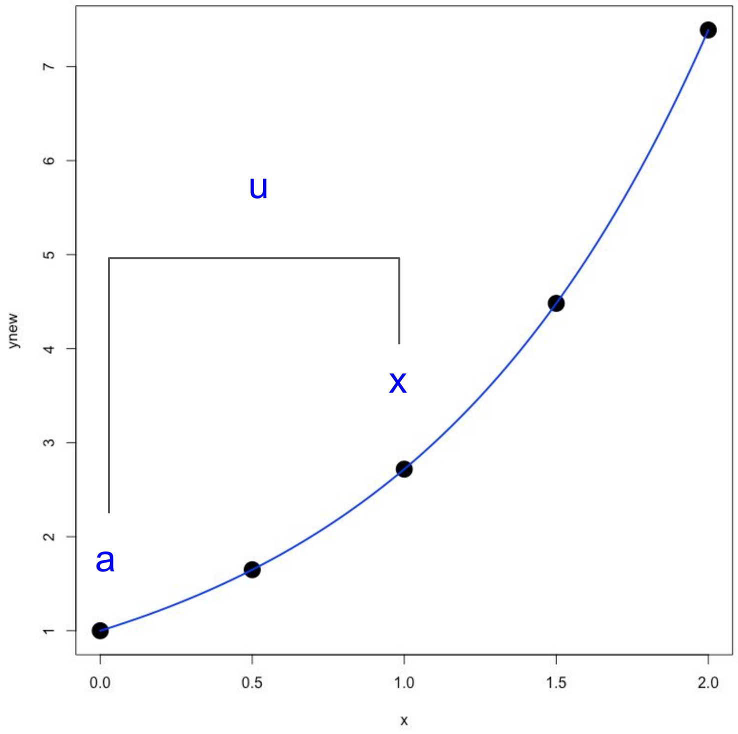
Interpolation Methods:
Taylor interpolation

- Expected error reduces to

$$E_5(x) = \frac{1}{6!} f^6(u \in [0,1])$$

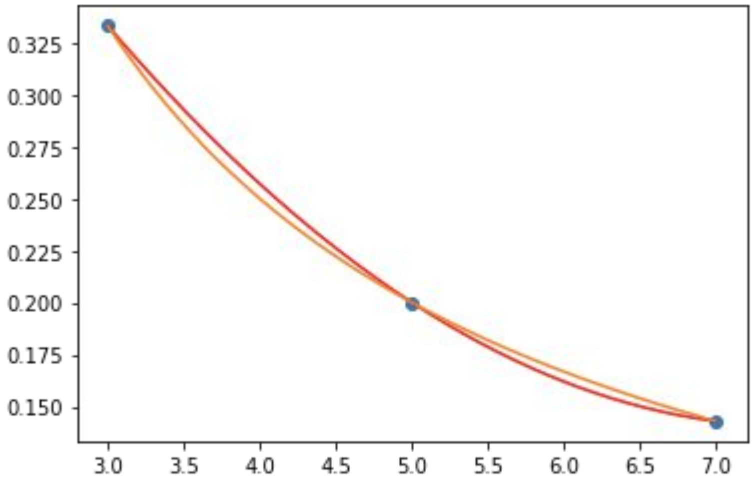
- Given that $f^i(x) = e^x$ and $u<1$ *we* can bound maximum absolute error:

$$E_5(x) = \frac{1}{6!} f^6(u) \leq \frac{e^u}{6!} \leq \frac{e^1}{6!} \sim 0.00378$$



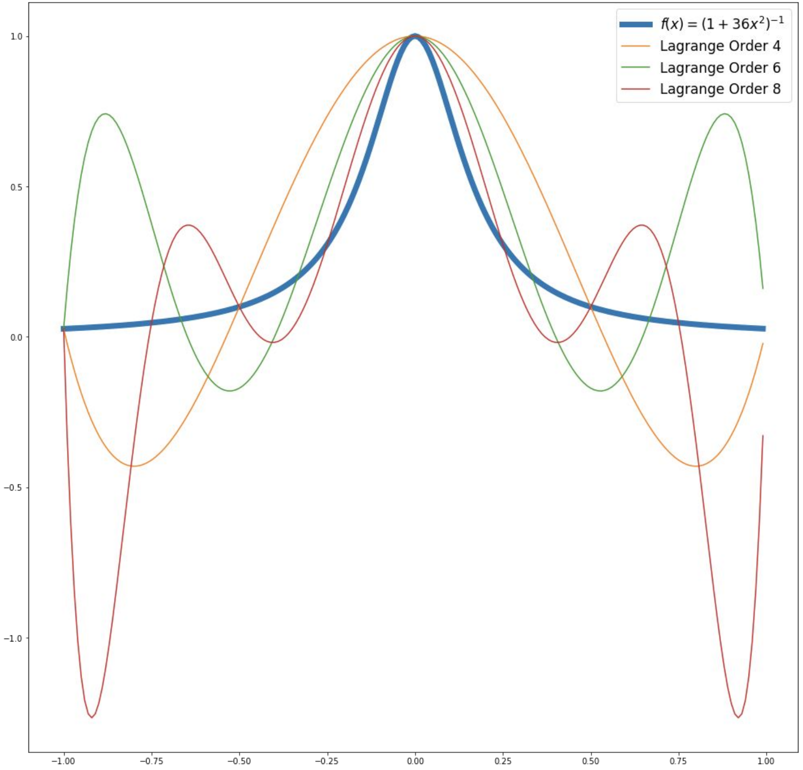
Interpolation Methods:
More ill-behaved examples

- What is guaranteed in these simple interpolation schemes, is that there **is a polynomial of *some order n*** which will pass through the points**, and reproduce the function to within a maximal interpolation error, $E_{n,max}$**
- It is ***not guaranteed*** that the maximum error $E_{max} \rightarrow 0$ as n $\rightarrow$ becomes large!



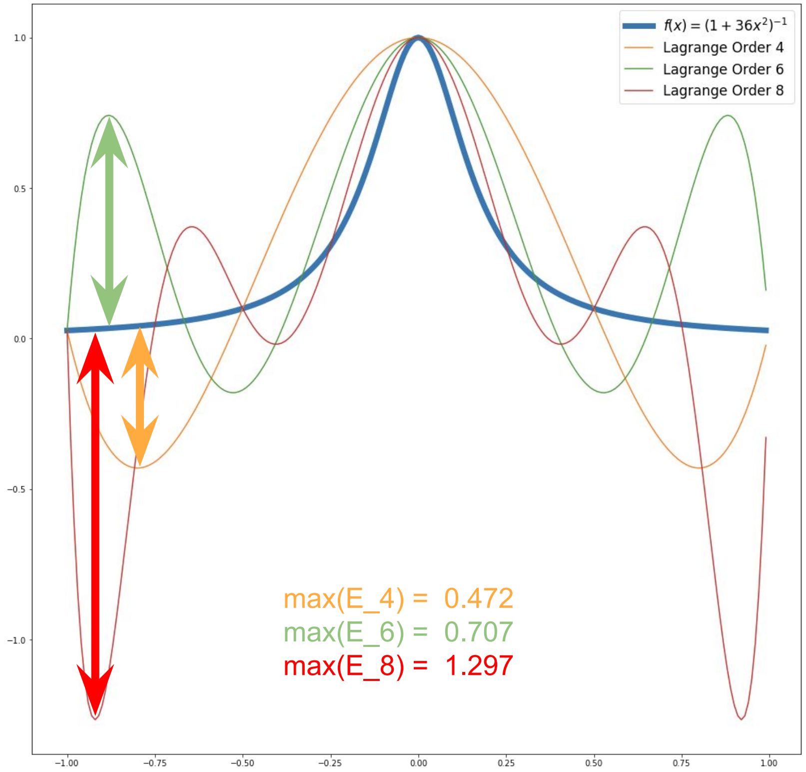
Interpolation Methods:
More ill-behaved examples

- What is guaranteed in these simple interpolation schemes, is that there **is a polynomial of *some order n*** which will pass through the points**, and reproduce the function to within a maximal interpolation error, $E_{n,max}$**
- It is ***not guaranteed*** that the maximum error $E_{max} \rightarrow 0$ as $n \rightarrow$ becomes large!



Interpolation Methods:
More ill-behaved examples

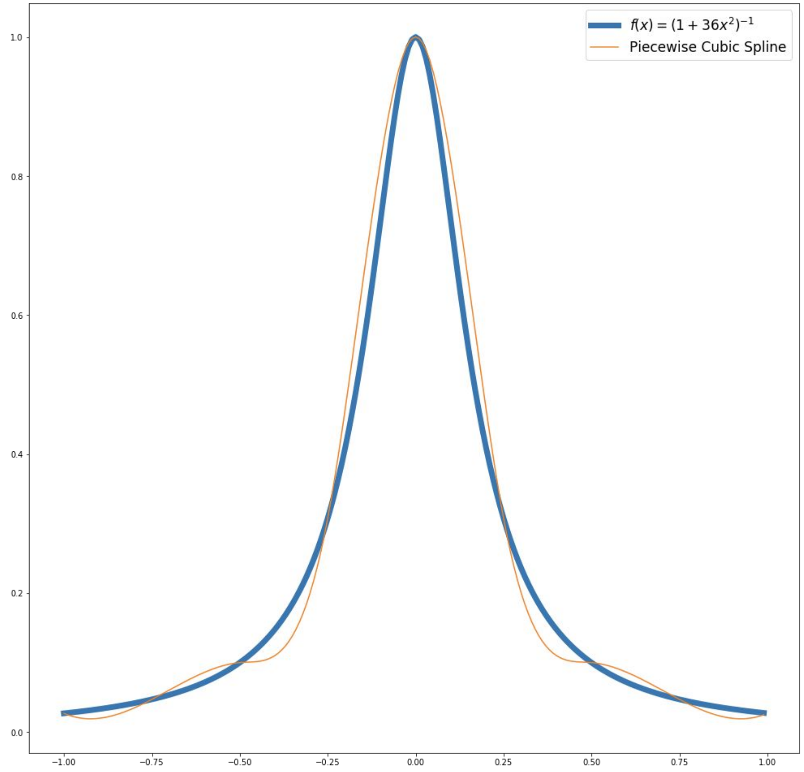
- What is guaranteed in these simple interpolation schemes, is that there **is a polynomial of *some order n*** which will pass through the points, **and reproduce the function to within a maximal interpolation error, $E_{n,max}$**
- It is ***not guaranteed*** that the maximum error $E_{max} \rightarrow 0$ as $n \rightarrow$ becomes large!



Interpolation Methods:
More ill-behaved examples

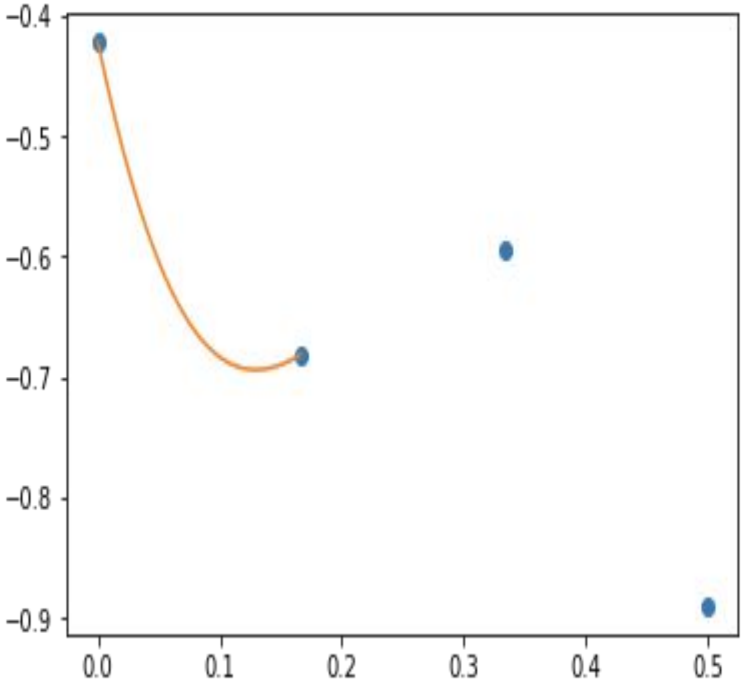
- We can try to improve this behaviour by considering ***piecewise*** interpolating functions.

- These divide the function into segments and impose boundary conditions for how interpolants must behave and inter-connect.



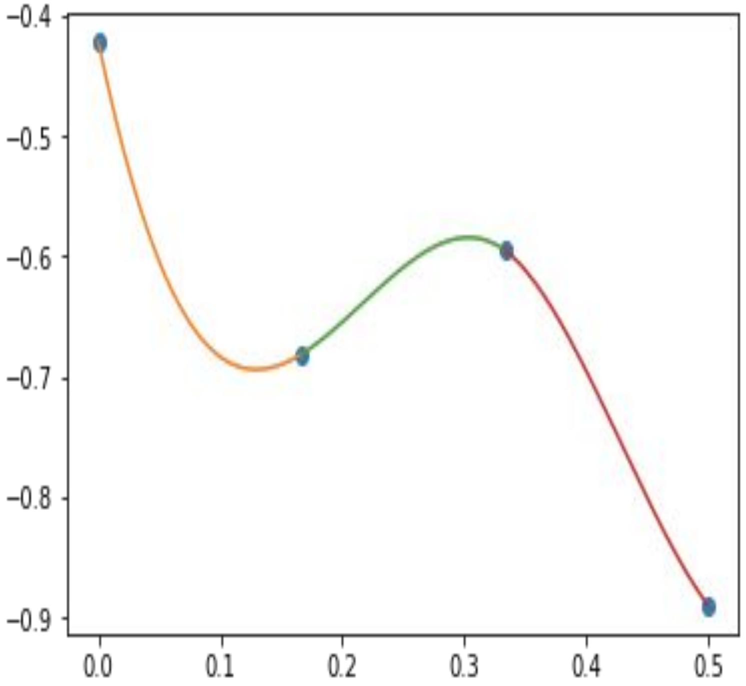
Interpolation Methods:
Spline interpolation

- To avoid errors with large number of points, we can break apart the data set into piecewise segments.



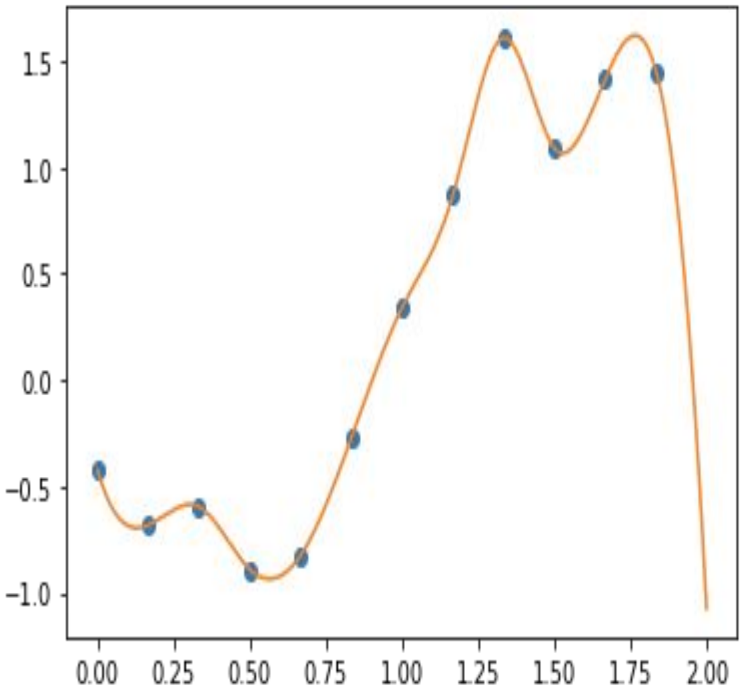
Interpolation Methods:
Spline interpolation

- To avoid errors with large number of points, we can break apart the data set into piecewise segments.
- Find flexible polynomials which reproduce continuity (slope and curvature) with the next segment



Interpolation Methods:
Spline interpolation

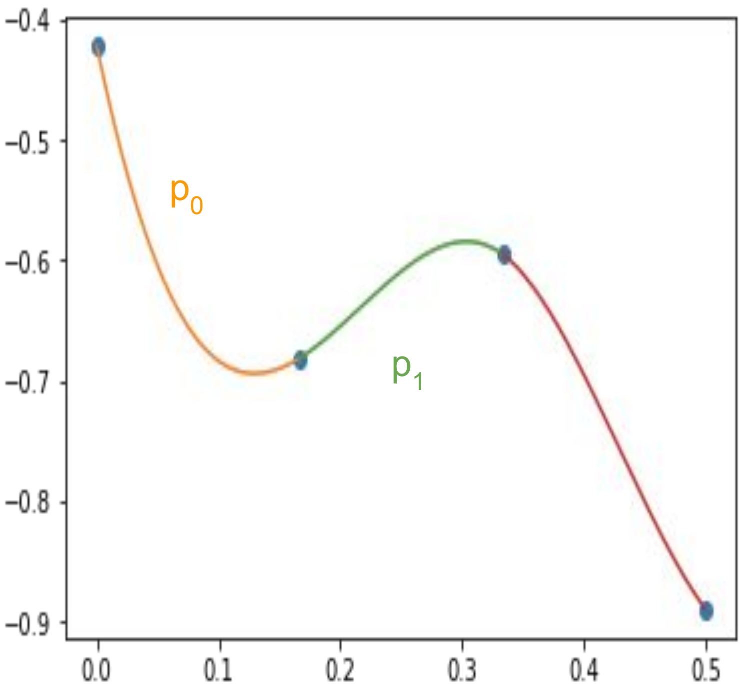
- To avoid errors with large number of points, we can break apart the data set into piecewise segments.
- Find flexible polynomials which reproduce continuity (slope and curvature) with the next segment
- Formally we require the first and second derivatives of each endpoint to match for each cubic polynomial constructed for a segment.



Interpolation Methods:
Spline interpolation

- In brief, for cubic splines (m=3), for each interval you have a polynomial which reproduces those points

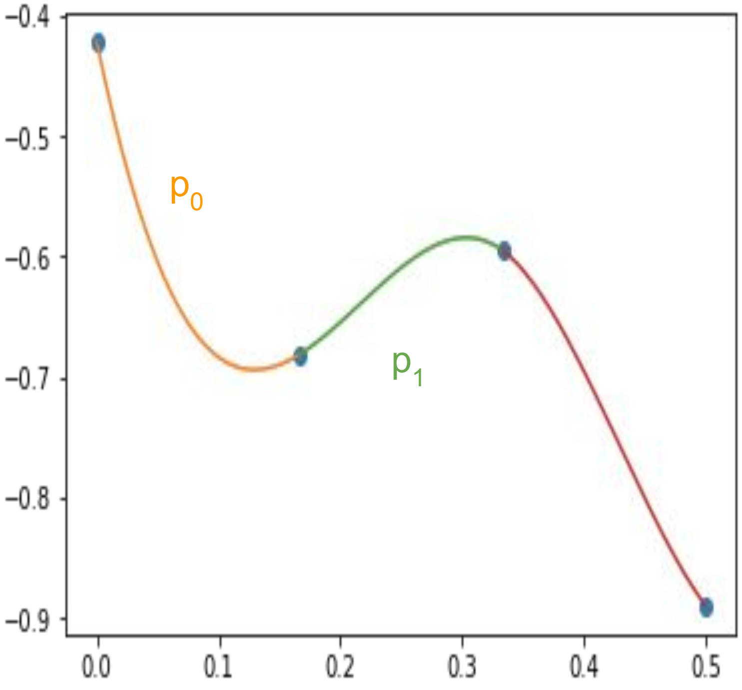
$$p_i(x) = \sum_{k=0}^{k=[m=3]} c_{i,k} x^k$$



Interpolation Methods:
Spline interpolation

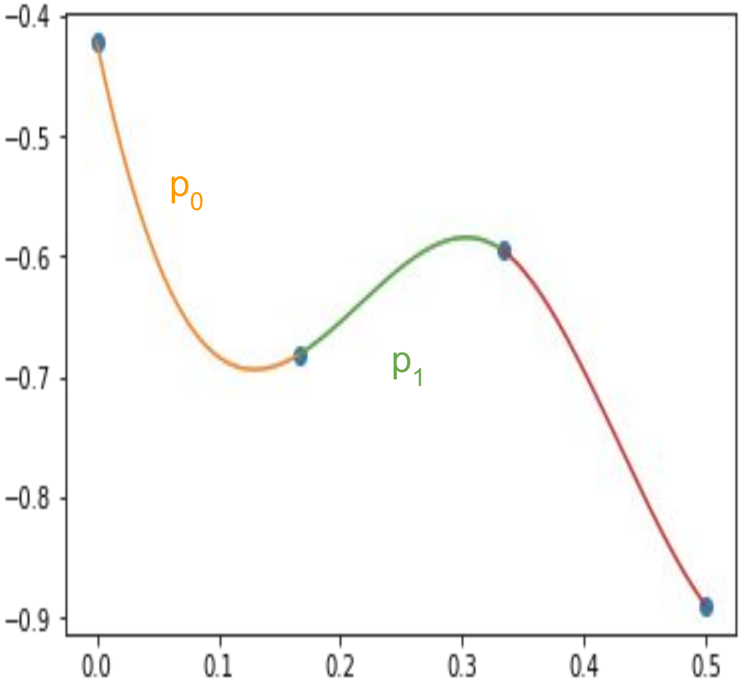
- Also require *j=0,...m-1* derivatives to match at interval transition points

$$p_i^j(x_{i+1}) = p_{i+1}^j(x_{i+1})$$



Interpolation Methods:
Spline interpolation

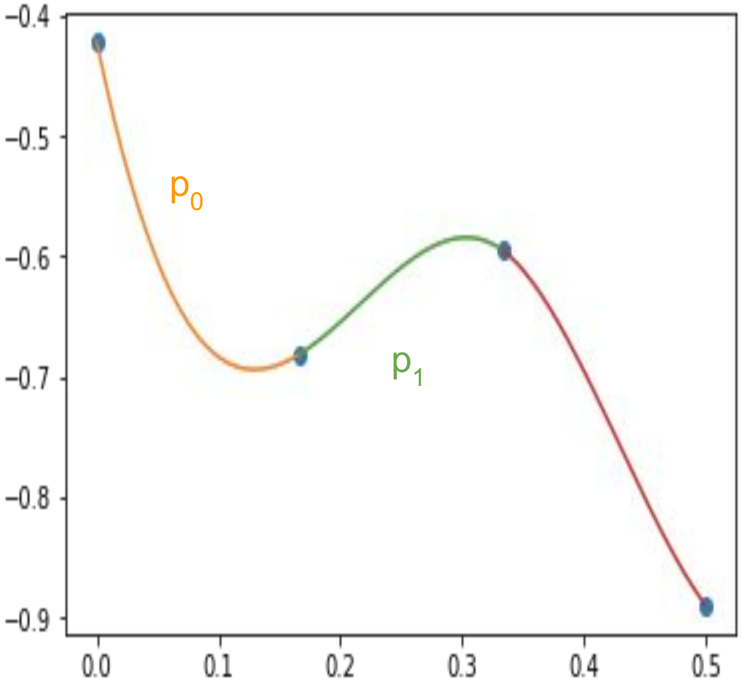
- If you have n data points, there are ***n-1*** intervals of $[x_i, x_{i+1}]$



Interpolation Methods:
Spline interpolation

- If you have n data points, there are ***n-1*** intervals of $[x_i,x_{i+1}]$

- In each interval you have a polynomial that has order *j* with ***j+1*** coefficients

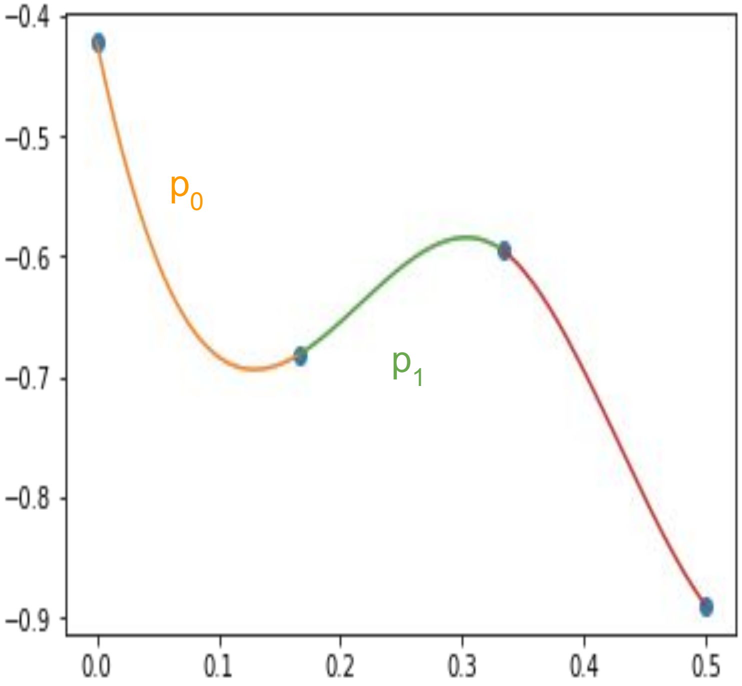


Interpolation Methods:
Spline interpolation

- If you have n data points, there are ***n-1*** intervals of $[x_i, x_{i+1}]$

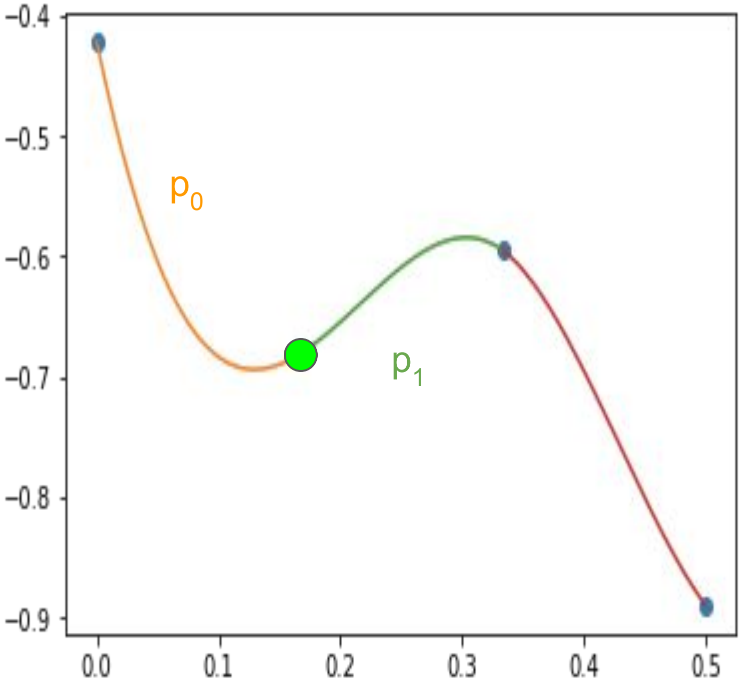
- In each interval you have a polynomial that has order *j* with ***j+1*** coefficients

- Need to jointly solve for coefficients of all polynomials together → Need ***(j+1)(n-1)*** *(= 12 constraints, in this example*)



Interpolation Methods:
Spline interpolation

- For all middle points you get continuity constraints between $p_i^j(x_i) = p_{i+1}^j(x_i)$ up to $j=m-1$

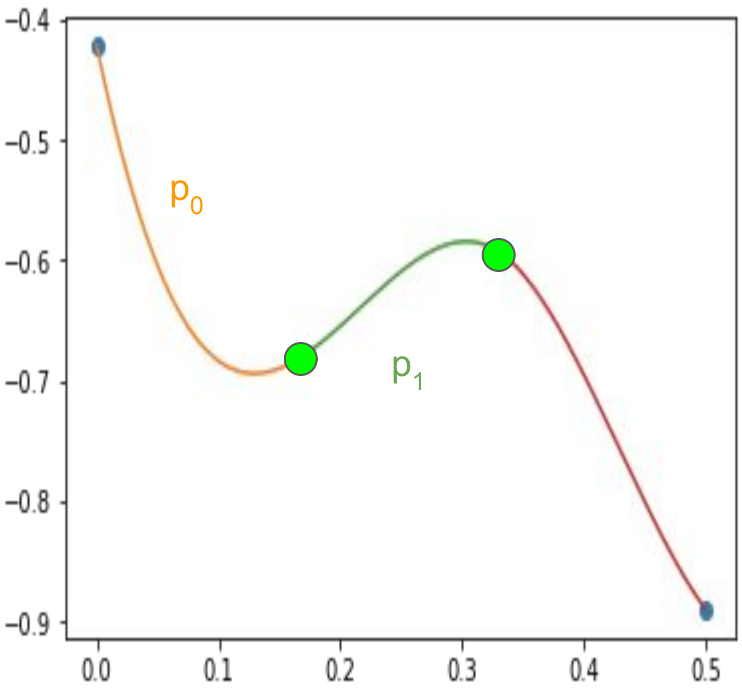


Interpolation Methods:
Spline interpolation

- For all middle points you get continuity constraints between $p_i^j(x_i) = p_{i+1}^j(x_i)$ up to $j=m-1$
- E.g.,

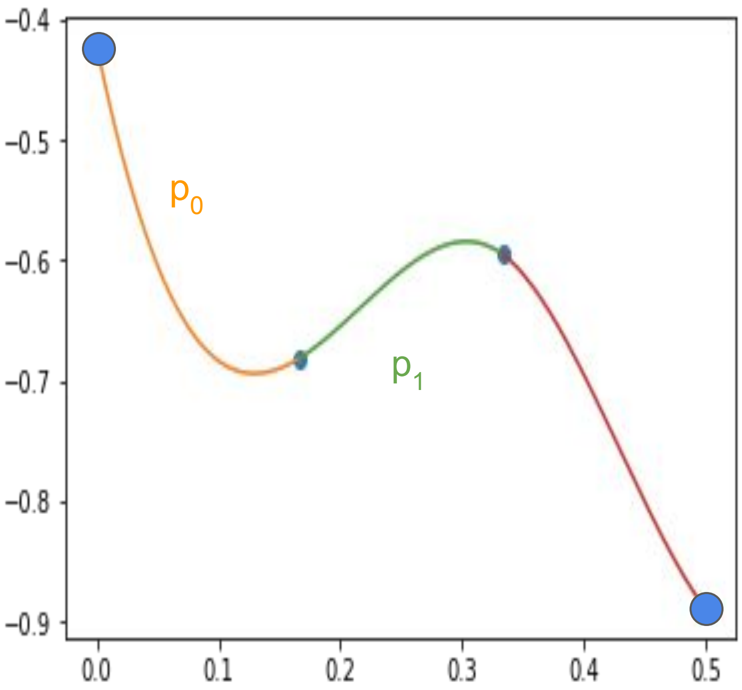
$$p_0(x_1) = p_1(x1) = y_1$$
$$p_0'(x_1) = p_1'(x_1)$$
$$p_0''(x_1) = p_1''(x_1)$$

- Same for other middle point… so **8** constraints so far



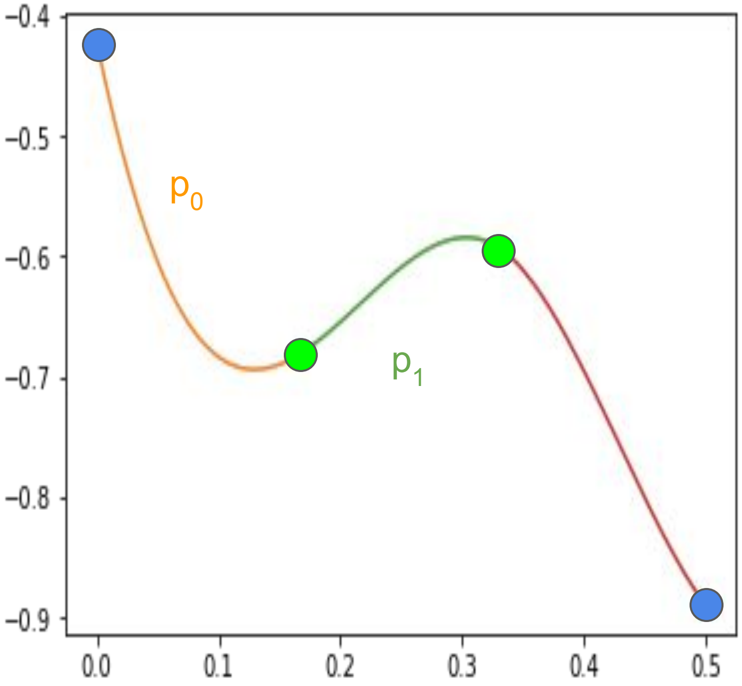
Interpolation Methods:
Spline interpolation

- First and last point must trivially satisfy the value of the data
  $p_0(x_0) = y_0$; $p_2(x_3) = y_3$



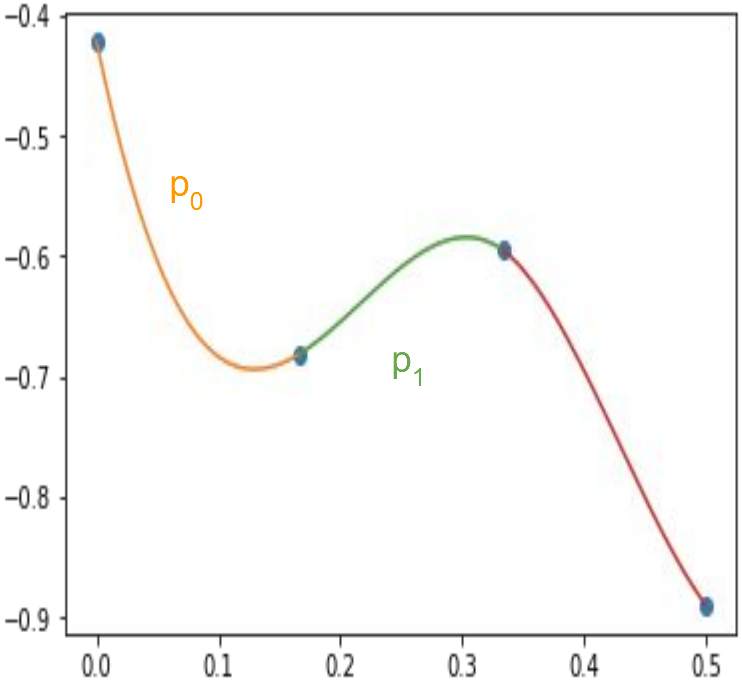
Interpolation Methods:
Spline interpolation

- First and last point must trivially satisfy the value of the data
  $p_0(x_0) = y_0$; $p_2(x_3) = y_3$
- This is now **10** constraints out of **12** in this example.



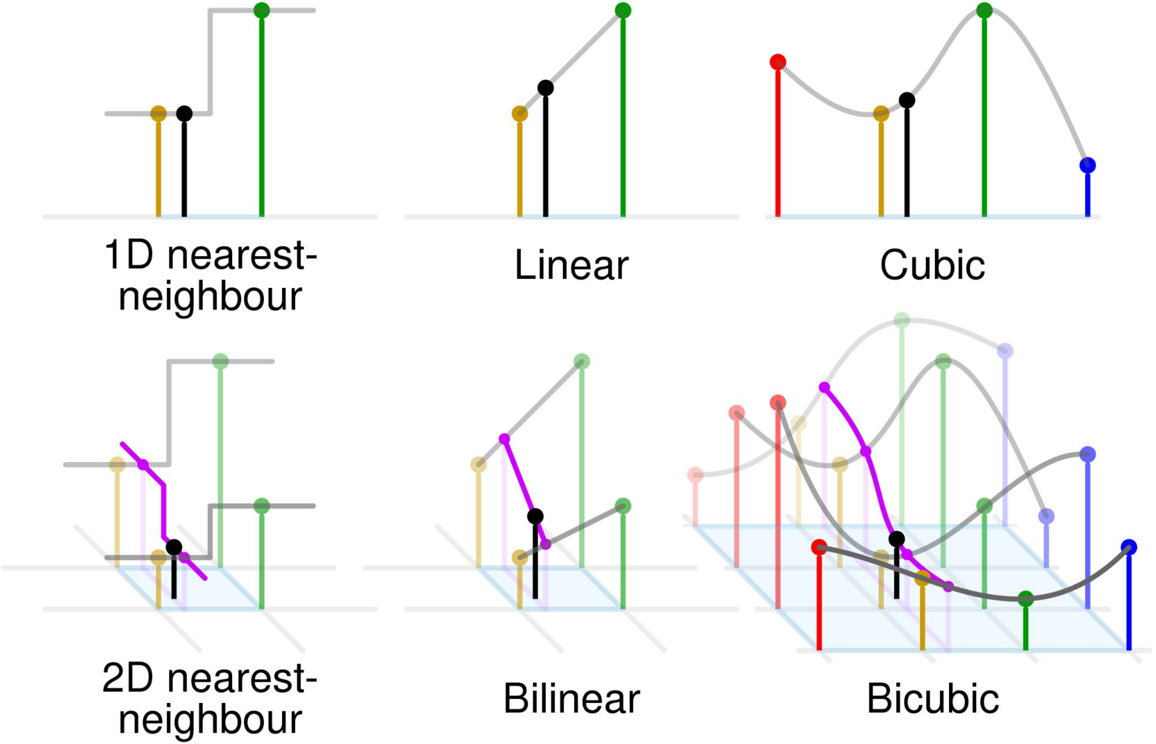
Interpolation Methods:
Spline interpolation

- First and last point must trivially satisfy the value of the data
  $p_0(x_0) = y_0 \quad ; \qquad p_2(x_3) = y_3$
- This is now **10** constraints out of **12** in this example.
- For last two constraints often set second derivative of the first and last point to zero, or linearly extrapolate and set continuity to that line - or any other arbitrary slope.



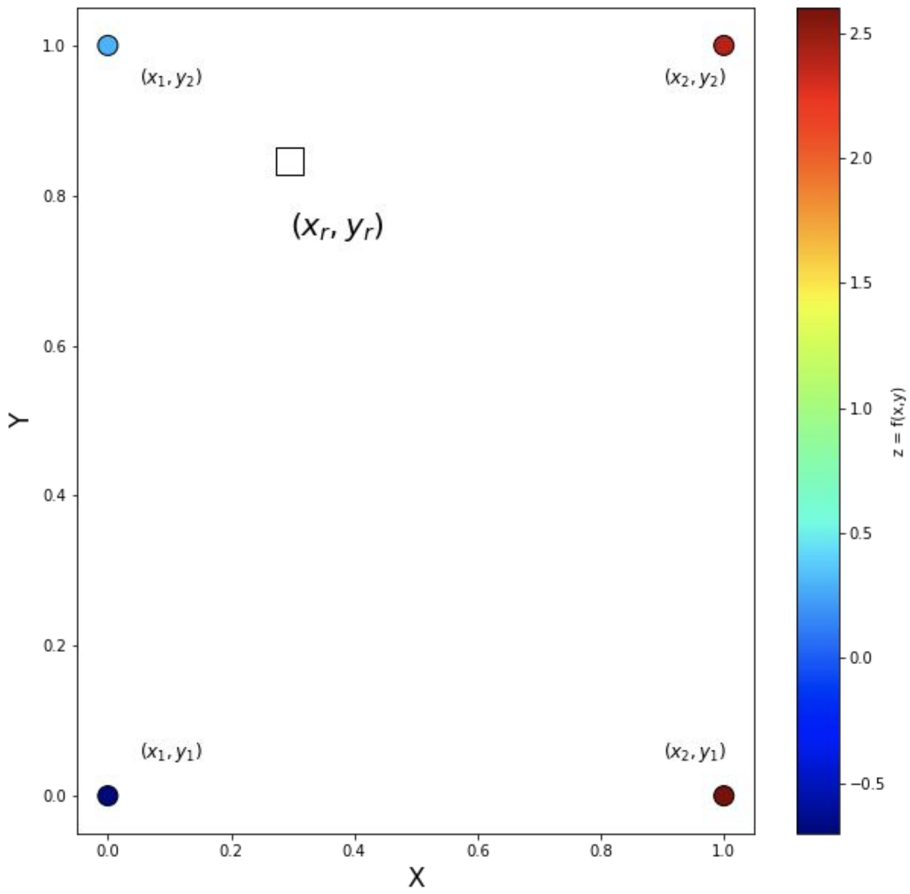
Interpolation Methods:
2D interpolation

- We can extend some of these interpolation schemes easily to higher dimensions.
- Typically you will want to use some built-in routines to do this, but bi-linear or bi-cubic interpolation are conceptually and algorithmically easy to understand



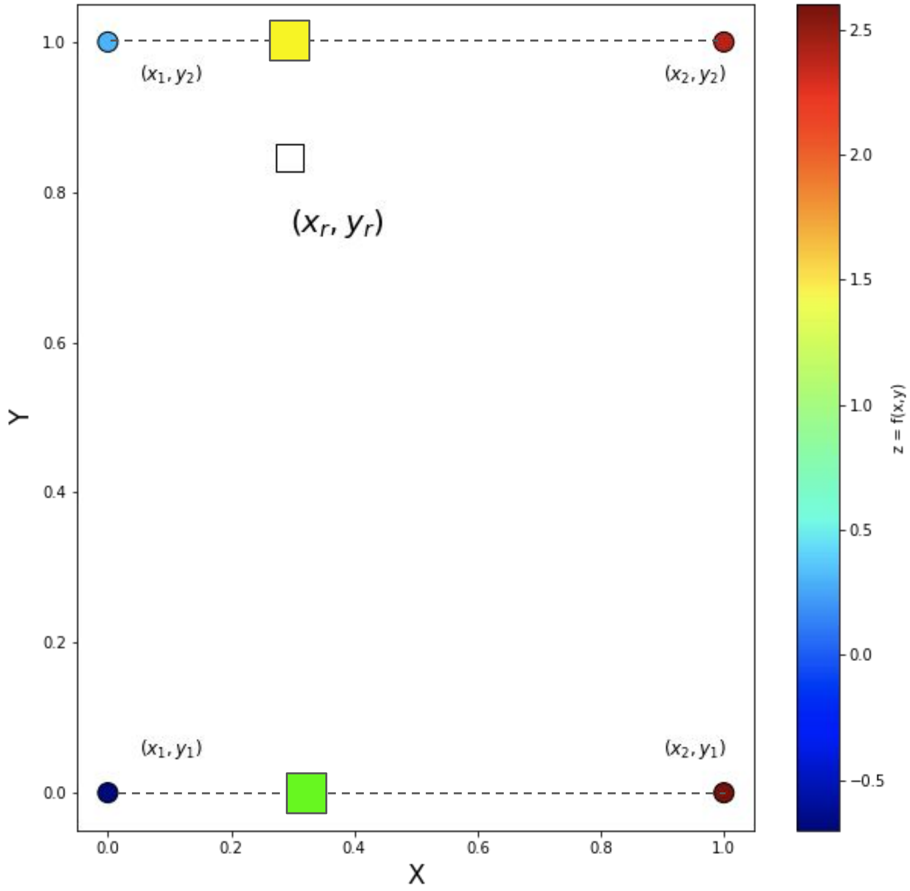
Interpolation Methods:
2D interpolation

- Finding f(x,y) at an arbitrary location based on known surrounding data is not a linear problem in general.



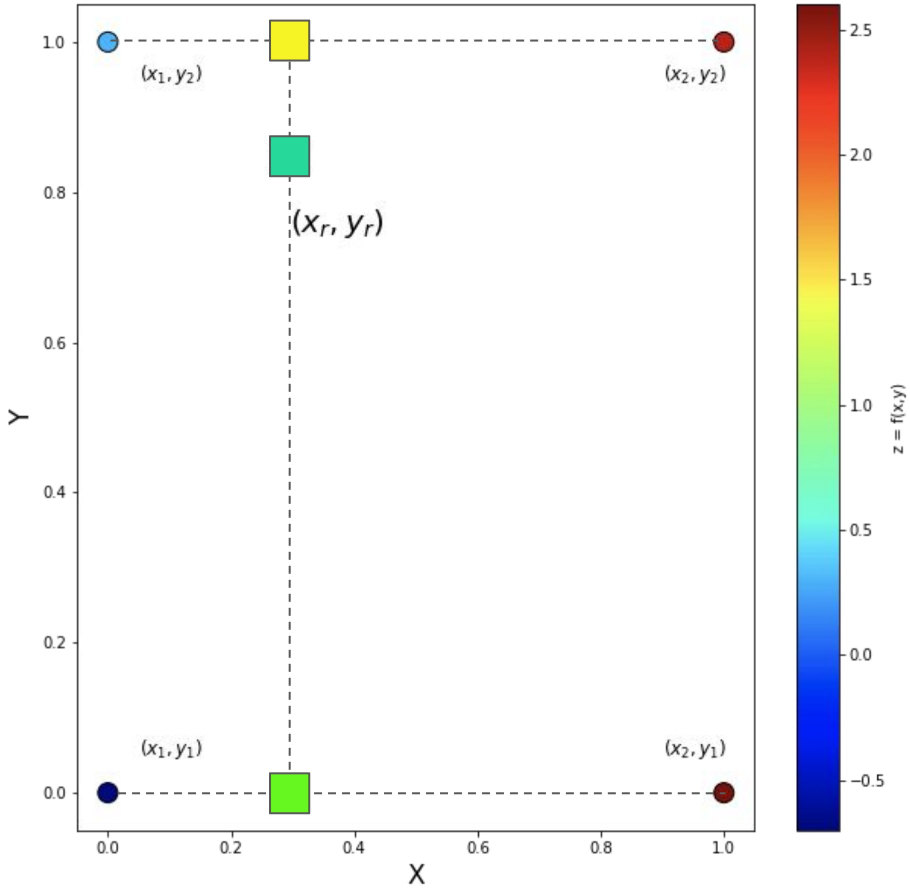
Interpolation Methods:
2D interpolation

- Finding f(x,y) at an arbitrary location based on known surrounding data is not a linear problem in general.
- However a direct approach is to approximate the parameter space as linear in x and y separately.



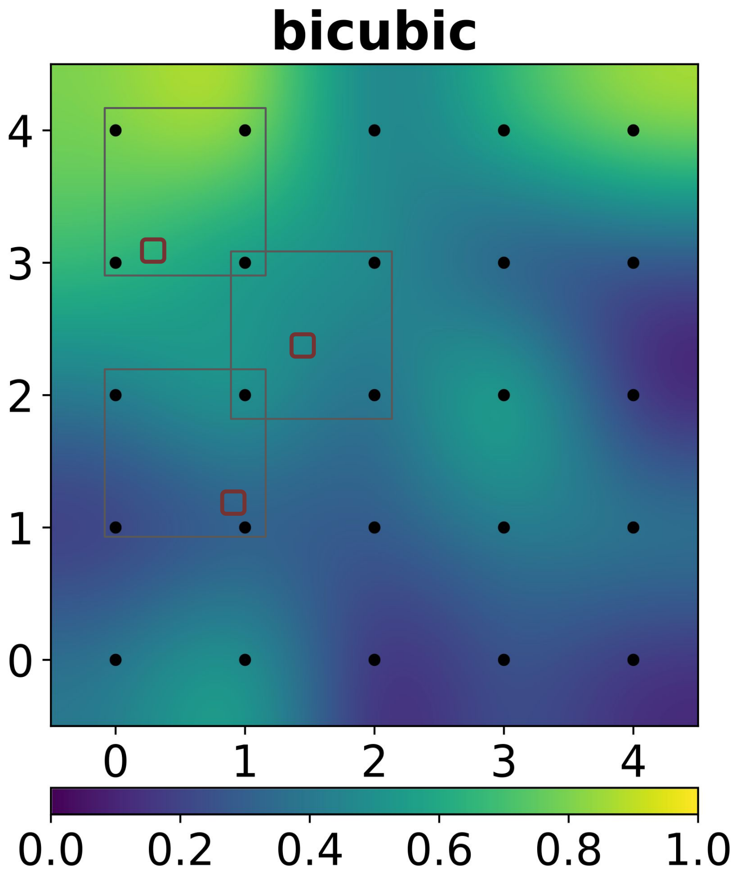
Interpolation Methods:
2D interpolation

- Finding f(x,y) at an arbitrary location based on known surrounding data is not a linear problem in general.
- However a direct approach is to approximate the parameter space as linear in x and y separately.



Interpolation Methods:
2D interpolation

- Whether bi-linear, or bi-cubic or 2D spline, most algorithms will do this piece-wise in 2D.
- Particularly common for image resampling and compression
- These would still be considered deterministic interpolation methods

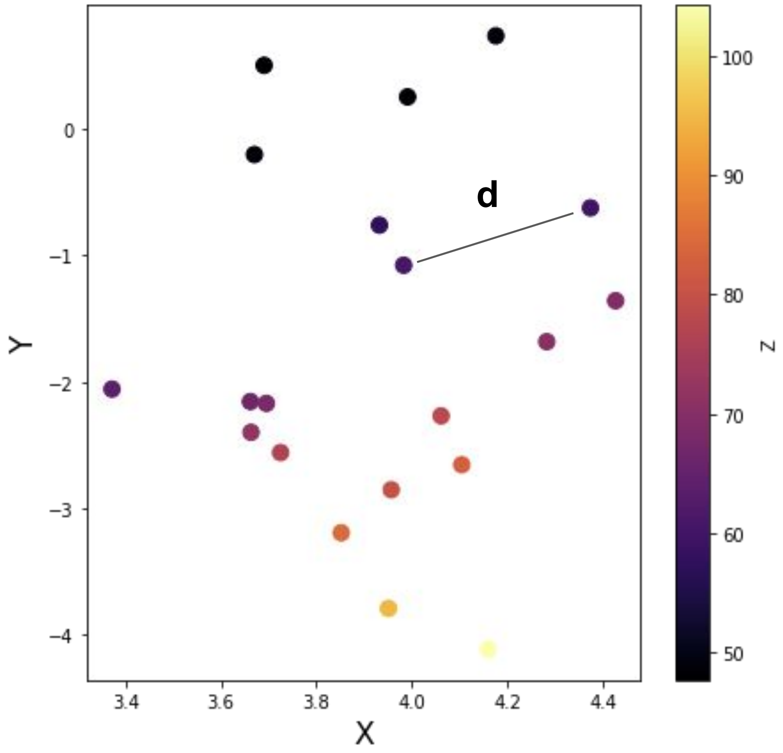


Interpolation Methods:
2D interpolation

- First, one computes a semivariogram - half the variance between data points as a function of their separation

$$\gamma(d) = \frac{1}{2}\mathrm{mean}((z_i - z_j)^2)$$

- Typically bin in pair separation (d) and then compute the quantity γ for all points with roughly that pair separation



Interpolation Methods:
2D interpolation

- Plotting them gives you a sense of how the data are correlated at different spatial scales.
- Typically an analytic function is fit to the semivariogram data
- This helps provide a smooth computation of the weights which go into interpolating a 2D data set

$$z(x_{new}, y_{new}) = \sum_{i}^{N} \lambda_i z(x_i, y_i)$$

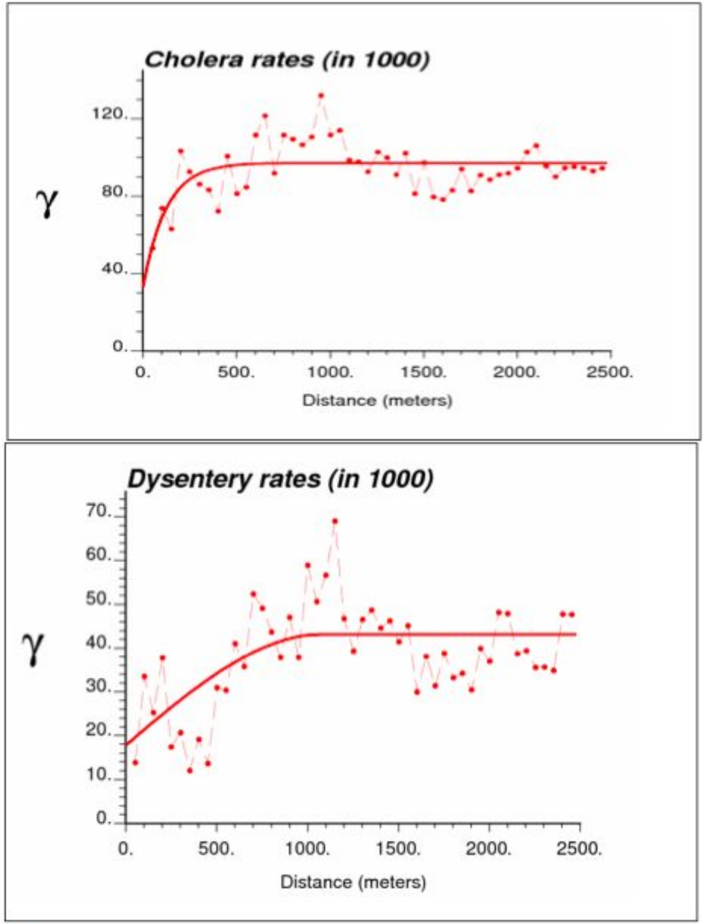


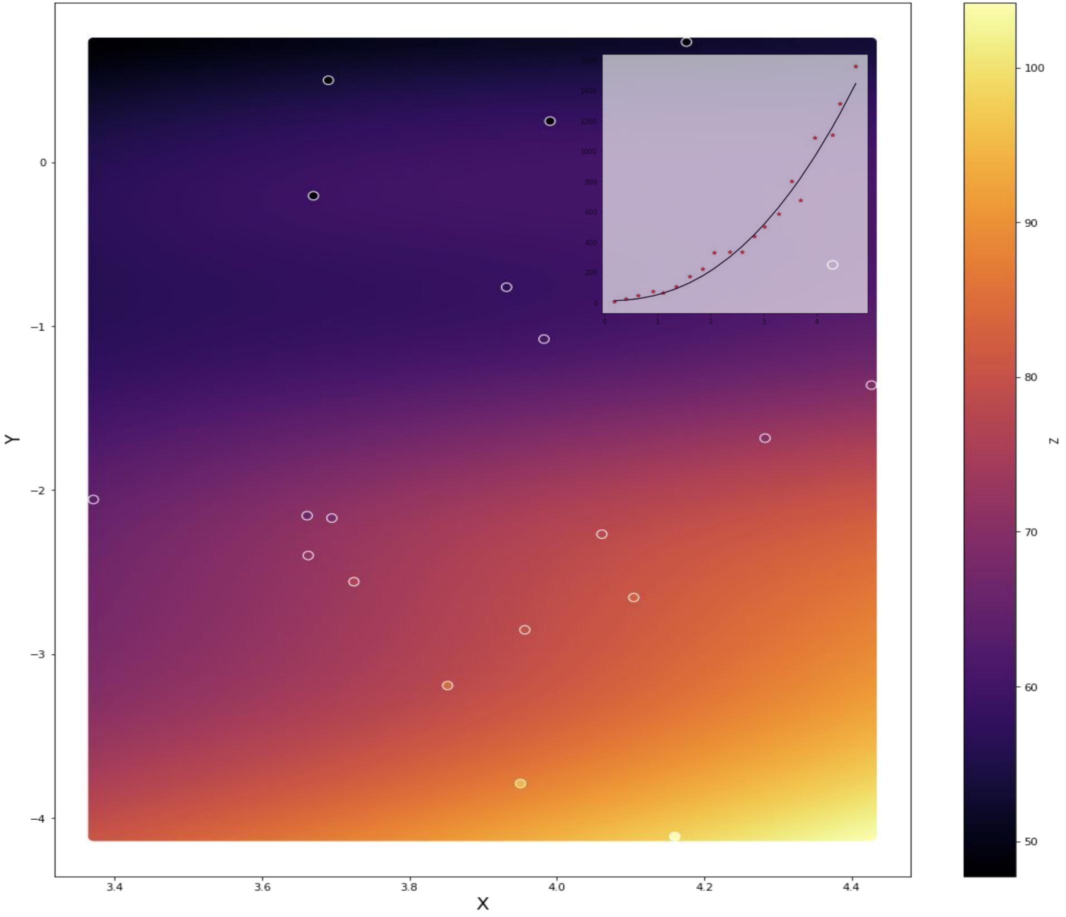
Figure 3
Omnidirectional semivariograms of cholera and dysentery incidence rates, with the model fitted.

Interpolation Methods:

2D interpolation

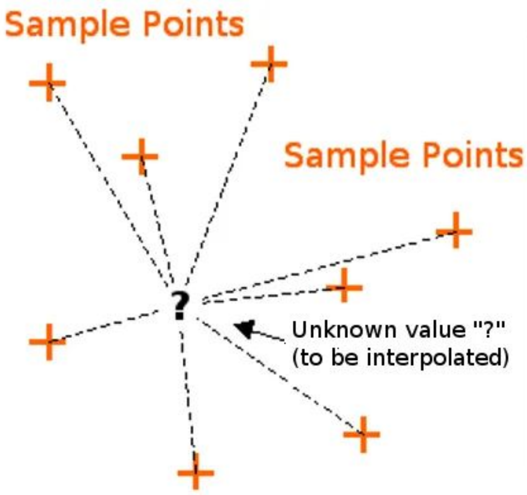
- These weights try to account for some covariance amongst the data during the interpolation.

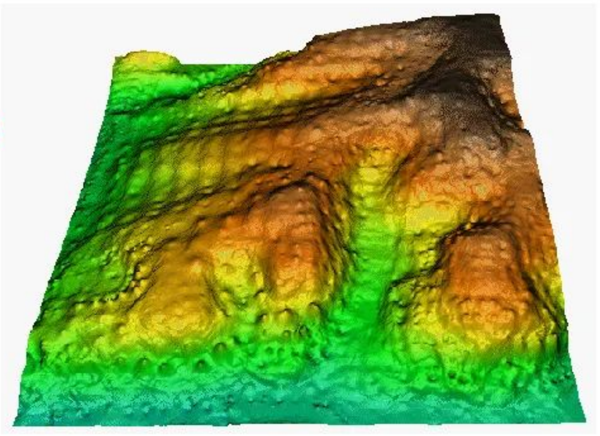
- It can be extremely sensitive to the function fit to the the variogram data



Interpolation Methods:
2D interpolation

- A more stable but less flexible option is **Inverse Distance Weighting (IDW)**
- Here the weighting (λ) is of the form: $\lambda = d^{-p}$
- While closer points provide more weight you still have a subjective choice on how quickly this weighting falls off with distance





Interpolation Methods:
2D interpolation

- As with 1D interpolation you still have subjective choices which can significantly alter your final interpolated values
- E.g., why should weighting fall off linearly with distance? Quadratically? Which variogram model best represents the covariance in the data?
- **You can't separate most numerical algorithms from subjective choices**
- **Even things that are trained on data, have biases because of training sets or construction choices for the neural network, interpolation, or optimization routines.**
- **The best thing you can do is TEST parameter variations! Understand limitations and always consider the physical system.**

Implementation in Python

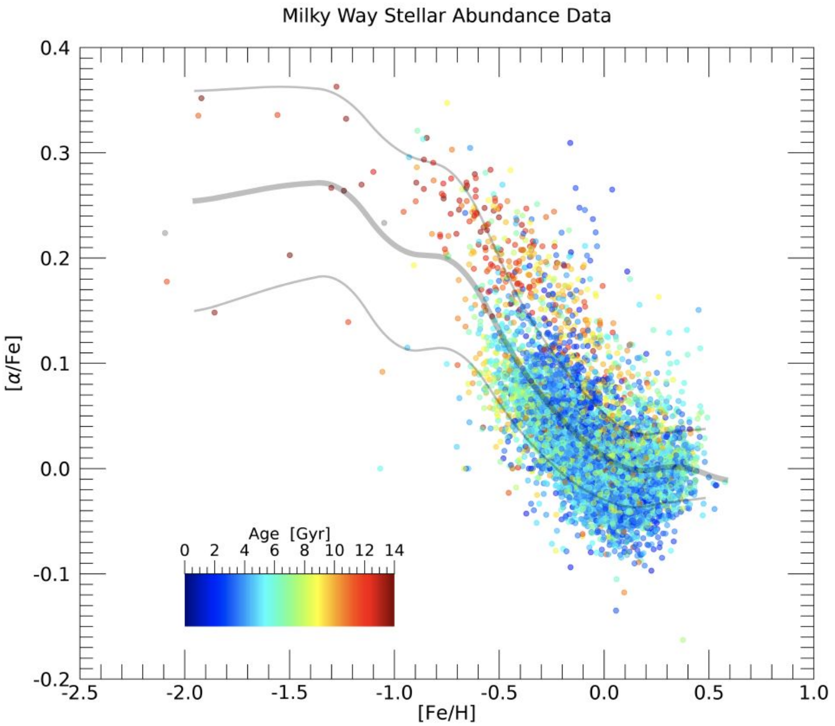
Interpolation Type	Function	Library	Description
1D Linear Interpolation	`interp1d`	`scipy`	Creates a linear interpolant over 1D data. Useful for filling gaps in evenly spaced data.
1D Spline Interpolation	`UnivariateSpline`	`scipy`	Smooth spline interpolation over 1D data, with adjustable smoothing. Ideal for noisy data.
Cubic Spline	`CubicSpline`	`scipy`	Provides cubic spline interpolation, often producing smoother results.
2D Interpolation	`interp2d`	`scipy`	Interpolates over 2D grids using linear, cubic, or quadratic methods.
Multidimensional Spline	`RectBivariateSpline`	`scipy`	Spline-based 2D interpolation for gridded data in two dimensions.
Regular Grid Interpolation	`RegularGridInterpolator`	`scipy`	Interpolates for N-dimensional data on a regular grid. Can handle more than 2 dimensions.

Implementation in Python

Grid Interpolation for Scattered Data	`griddata`	`scipy`	Interpolates unstructured 2D or 3D data using linear, cubic, or nearest neighbor methods.
Nearest Neighbor Interpolation	`NearestNDInterpolator`	`scipy`	Interpolates nearest neighbors for multidimensional data, useful for irregular grids.
Radial Basis Function (RBF)	`Rbf`	`scipy`	Performs interpolation based on radial basis functions for scattered 1D, 2D, or 3D data.
Piecewise Polynomial	`PchipInterpolator`	`scipy`	Monotonic piecewise cubic Hermite interpolator, suitable for non-oscillatory data.
Polynomial Interpolation	`polyfit`	`numpy`	Fits a polynomial to the data, suitable for small datasets or low-degree polynomials.
Lagrange Interpolation	`lagrange`	`scipy`	Uses Lagrange polynomials for interpolation; can be unstable for high-degree polynomials.

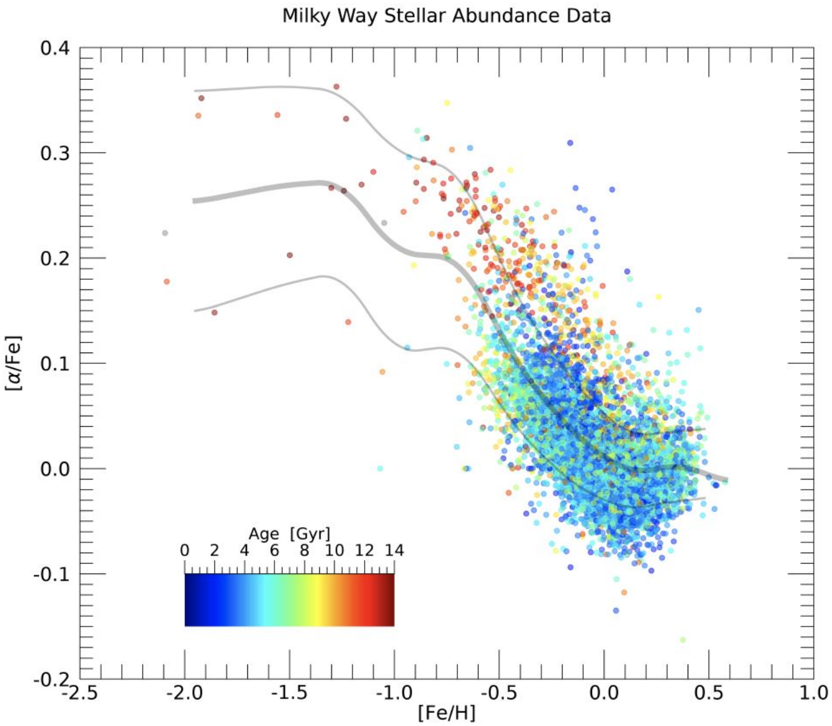
Approximation Methods:
Running averages

- In some cases we may want to highlight the smooth behaviour of a data set, rather than an instantaneous interpolation between two values.



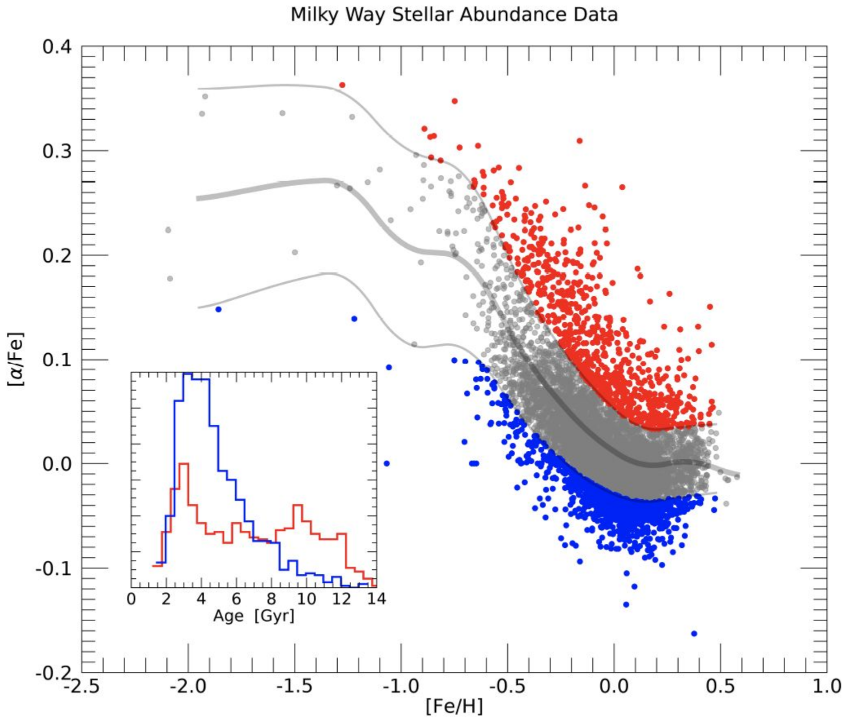
Approximation Methods:
Running averages, confidence intervals

- In some cases we may want to highlight the smooth behaviour of a data set, rather than an instantaneous interpolation between two values.
- This can be sometimes useful for visualization and characterization purposes.



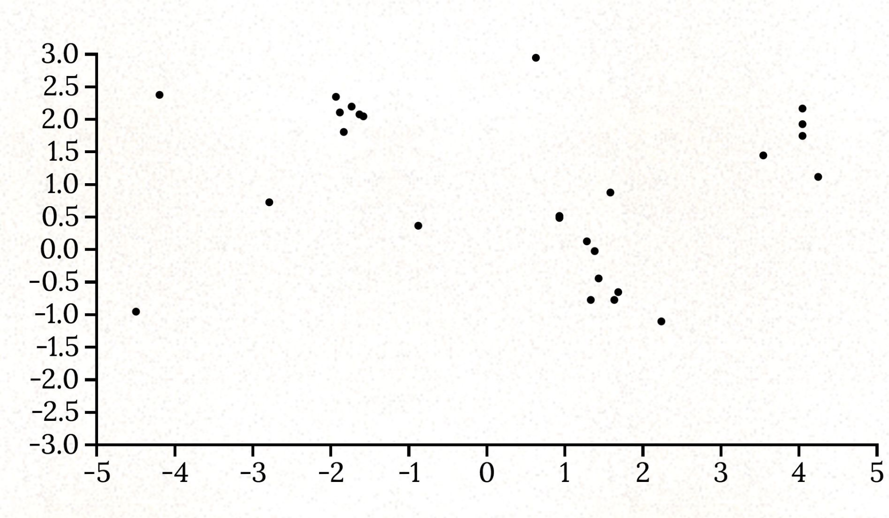
Approximation Methods:
Running averages, confidence intervals

- In some cases we may want to highlight the smooth behaviour of a data set.
- This can be sometimes useful for visualization and characterization purposes.
- More often can help isolate influence/dependence of other variables in multi-dimensional data sets.



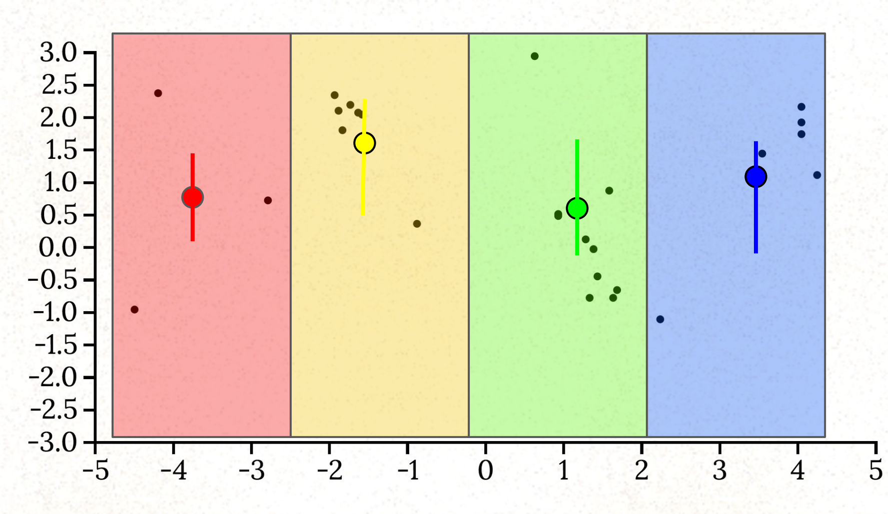
Approximation Methods:
Running averages, confidence intervals

- Simplest boxcar estimates of mean and variance allow for smooth reconstructions at the cost of resolution.



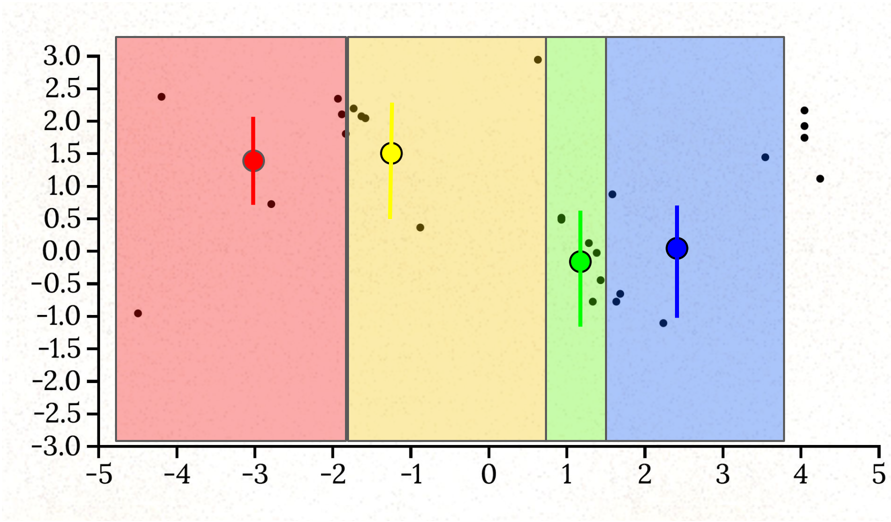
Approximation Methods:
Running averages, confidence intervals

- Simplest boxcar estimates of mean and variance allow for smooth reconstructions at the cost of resolution.
- Divide data $[x_i, x_{i+1}, ..x_n]$ into subsets and compute statistics of interest within this window



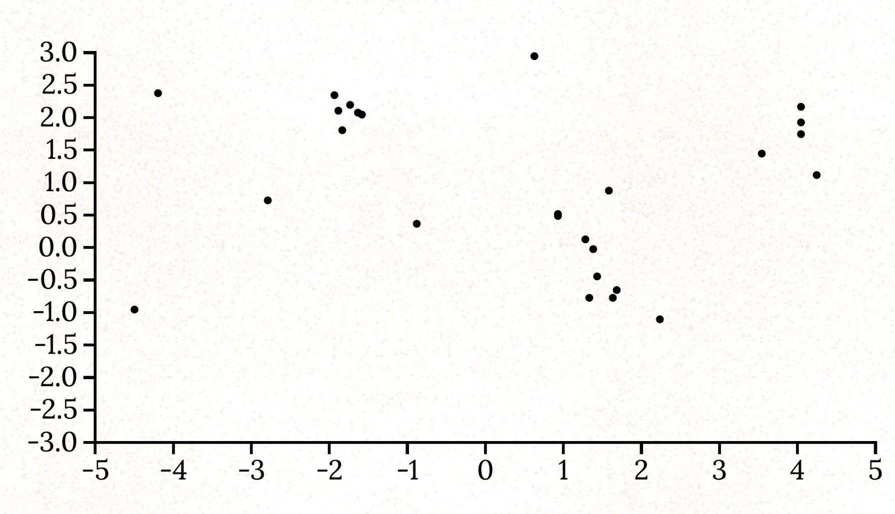
Approximation Methods:
Running averages, confidence intervals

- Simplest boxcar estimates of mean and variance allow for smooth reconstructions at the cost of resolution.
- Divide data $[x_i, x_{i+1}, ..x_n]$ into subsets and compute statistics of interest within this window
- Often want to preserve ~equal error in average estimates; can bin by i.e., equal number of points



Approximation Methods:
Running averages, confidence intervals

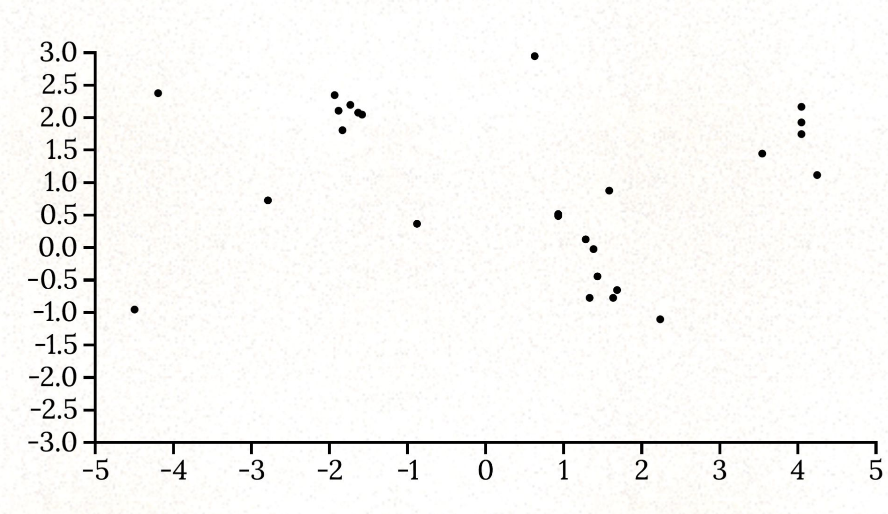
- Moving/running averages instead update the data set by replacing the last element of the window with the average from within the window



Approximation Methods:

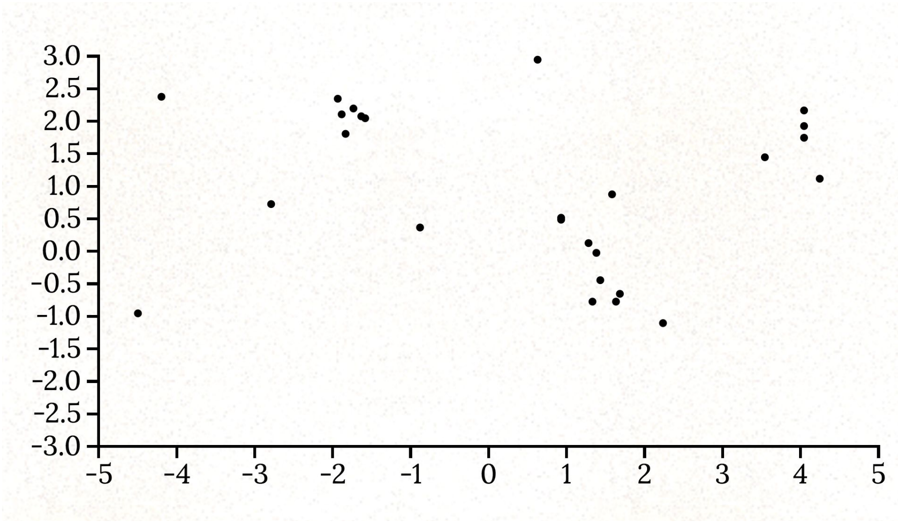
Running averages, confidence intervals

- Moving/running averages instead update the data set by replacing the last element of the window with the average from within the window
- E.g., $\boldsymbol{x_{is} = sum(x_i, x_{i+1} \ldots x_j)/j}$ for window size of *j*, over all data points $i<(n-j)$



Approximation Methods:
Running averages, confidence intervals

- Moving/running averages instead update the data set by replacing the last element of the window with the average from within the window
- E.g., $\boldsymbol{x_{is} = sum(x_i, x_{i+1} \ldots x_j)/j}$ for window size of *j*, over all data points *i<(n-j)*
- Has benefit of preserving resolution, but results qualitatively depend on window choice, and optionally, weighting choices



Summary on Interpolation/Approximation

- If you want smooth re-binning of simple data, representation of missing data for e.g., numerical integrals - **interpolation** can be helpful.
- If you want coarse trends of the data (including confidence intervals) in order to understand higher dimensional dependencies or role of outliers - **approximations/reconstructions** can be helpful
- Finally, remember that both interpolation and approximation are non-physical, can produce extrapolations or behaviours which are not necessarily the ‘truth’, and that subjective choices in e.g., order number or window size can change your interpretation.
- They are helpful methods - but **always**, consider the physical constraints of your system/question/data, and be aware that they all (even GPR) have subjective parameters which you should marginalize over.

Implementation in Python

Approximation Type	Function	Library	Description
Polynomial Approximation	`polyfit`	`numpy`	Fits a polynomial of a specified degree to data points; suitable for simple data or trends in smaller datasets.
Spline Approximation	`UnivariateSpline`, `LSQUnivariateSpline`, `CubicSpline`	`scipy`	Smooth curve fitting with splines, allowing for adjustable smoothing parameters. Great for 1D data.
Radial Basis Functions (RBF)	`Rbf`	`scipy`	Approximates scattered data in higher dimensions using radial basis functions, with various kernel options like Gaussian and Multiquadric.

Implementation in Python

Least Squares Fitting	`curve_fit`	`scipy`	Fits arbitrary functions to data by minimizing the squared error; often used for nonlinear curve fitting.
Piecewise Linear Approximation	`interp1d` with `kind='linear'`	`scipy`	Provides a piecewise linear approximation, essentially connecting data points with straight lines.
Piecewise Polynomial (PCHIP)	`PchipInterpolator`	`scipy`	Monotonic, piecewise polynomial fit that prevents oscillations, ideal for strictly increasing or decreasing data.
Gaussian Process Regression (GPR)	`GaussianProcessRegressor`	`sklearn`	Probabilistic model useful for high-dimensional function approximation, which provides uncertainty estimates.

Implementation in Python

Chebyshev Polynomial Approximation	`Chebyshev.fit`	`numpy.polynomial`	Fits Chebyshev polynomials, which minimize approximation error in the least-squares sense and are useful for function approximation over intervals.
Fourier Series Approximation	`fft` or `irfft`	`numpy`	Approximates periodic data by decomposing it into a sum of sinusoids, useful for analyzing frequency components.
Linear Regression	`LinearRegression`	`sklearn`	Fits a linear model to data, often used as a baseline approximation for continuous or noisy data with a linear relationship.

Implementation in Python

Polynomial Regression	`PolynomialFeatures` + `LinearRegression`	`sklearn`	Transforms data to include polynomial terms before applying linear regression, creating polynomial approximation.
k-Nearest Neighbors Regression	`KNeighborsRegressor`	`sklearn`	A non-parametric approximation that averages values from the nearest neighbors, useful for irregular data.
Support Vector Regression (SVR)	`SVR`	`sklearn`	Uses support vector machines for regression tasks, suitable for nonlinear relationships and higher-dimensional data.

Implementation in Python

Neural Network Approximation	`MLPRegressor`	`sklearn`	Multilayer Perceptron model for function approximation, suitable for complex and high-dimensional data patterns.
Local Polynomial Regression (LOESS)	`lowess`	`statsmodels`	Locally weighted regression that fits simple models around each data point, suitable for smooth approximation in 1D data with trends or seasonality.