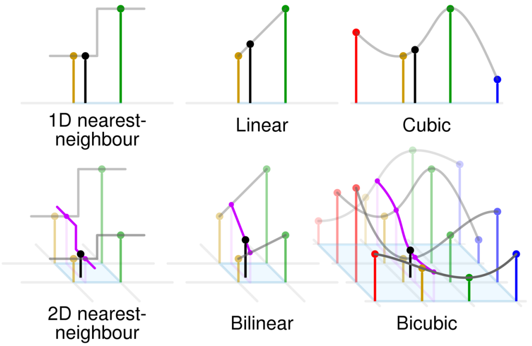
Computational Physics (PHYS6350)

Lecture 3: Interpolation



January 23, 2025

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: https://github.com/vlvovch/PHYS6350-ComputationalPhysics/tree/spring2025

Interpolation

Sometimes we know the value of some function $f(x)$ at a discrete set of points $x_0, x_1, \ldots, x_N$, but we do not know how to (easily) calculate its value at an arbitrary x

Examples:

- Physical measurements
- Long numerical calculations

Interpolation is a mathematical technique used to estimate values between a discrete set of known data points. It is also employed to approximate an unknown function based on its known values over a certain range.

Two steps:

1. Fitting the interpolating function to data points
2. Evaluating the interpolating function at a target point x

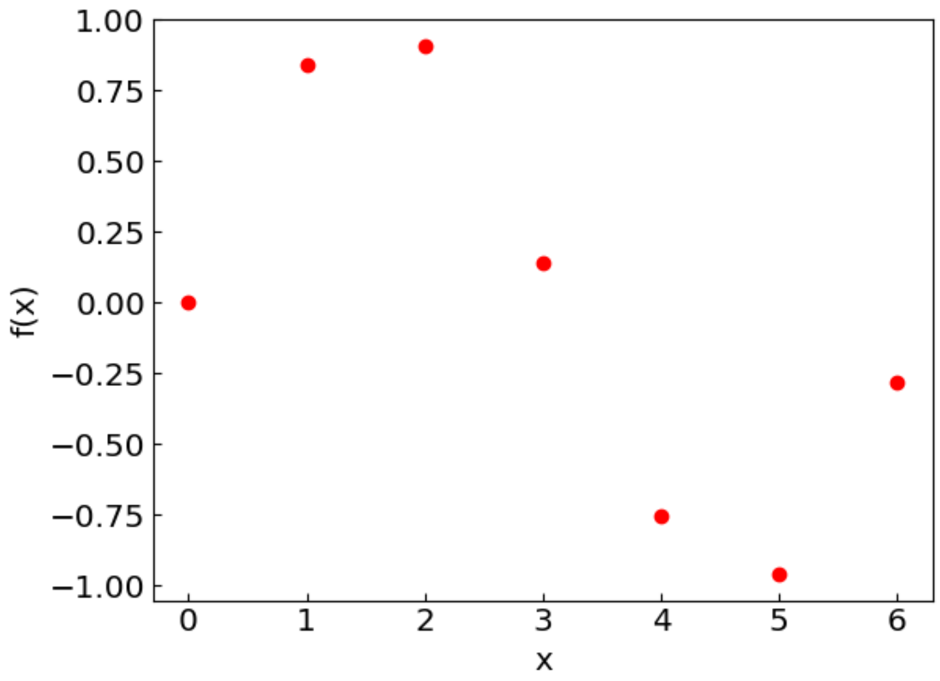
Reference: Chapter 3 of *Numerical Recipes Third Edition* by W.H. Press et al.

Interpolation

Recall the function $f(x) = \sin(x)$

Imagine that we cannot easily compute $\sin(x)$ at arbitrary x and all we are given is its values at some finite number of points

x	sin(x)
0	0.
1	0.841471
2	0.9092974
3	0.14112
4	-0.7568025
5	-0.9589243
6	-0.2794155



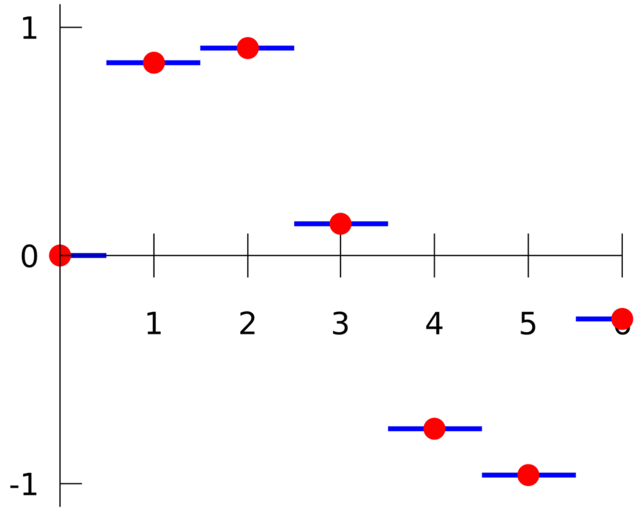
Now let us consider different methods of interpolating the function

Nearest-neighbor interpolation

Simply assign the value of the closest data point to x, i.e.

$f(x) = f_{nn}(x)$ where $f_{nn}(x) = y_i$

Where i is chosen such that $|x - xi|$ is the smallest among all i.

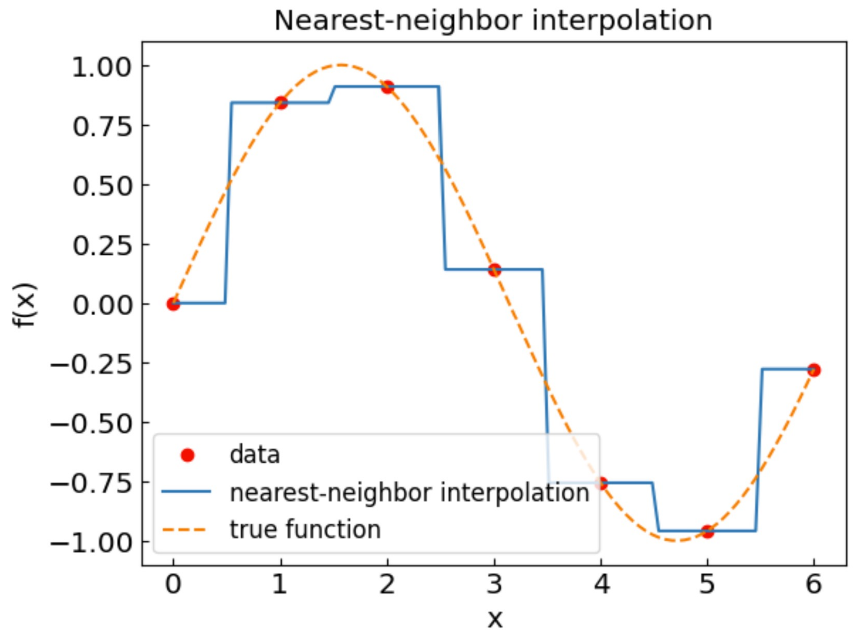


Nearest-neighbor interpolation

In Python:

```python
def f_nearestneighbor_int(x, xdata, fdata):
    """Returns the nearest-neighbor interpolation of a function at point x.
    xdata and ydata are the data points used in interpolation.
    xdata is assumed to be in sorted in ascending order."""
    ind = np.searchsorted(xdata, x) # Search for the interval for point x
    if (ind == 0):
        return xdata[0]
    if (ind == len(xdata)):
        return xdata[-1]
    x0,f0 = xdata[ind-1],fdata[ind-1]
    x1,f1 = xdata[ind],fdata[ind]
    if (abs(x-x0) < abs(x-x1)):
        return f0
    else:
        return f1

xcalc = np.linspace(0,6,100)
fcalc = [f_nearestneighbor_int(xin,xdat,fdat) for xin in xcalc]
```



Advantages:

- Very simple
- Easy to generalize to multiple dimensions

Disadvantages:

- Limited accuracy
- Better & simple options available

Linear interpolation

Let us have the data points (x_0,y_0) and (x_1,y_1)

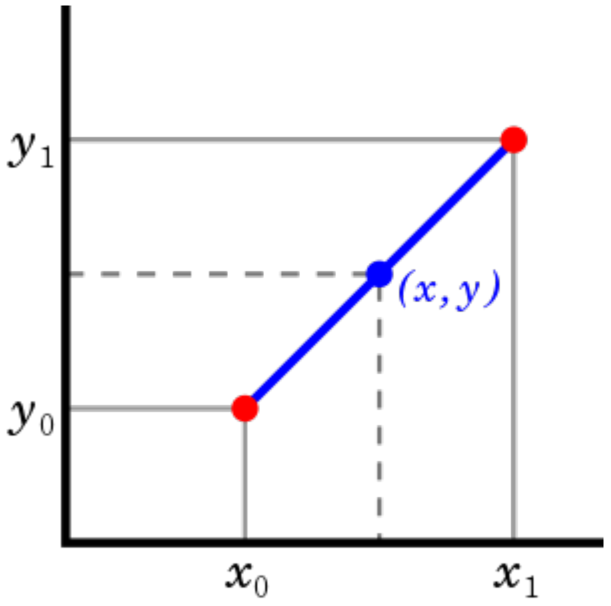
Linear interpolant is a **straight line** between these points

Use it to calculate the function value at any $x \in [x_0, x_1]$

$$f_{\text{lerp}}(x) = y_0 + \frac{x-x_0}{x_1-x_0}(y_1 - y_0)$$

For a larger set of points $x_0 < x_1 < \ldots < x_N$, find the interval (x_i, x_{i+1}) enveloping x and use the linear interpolant formula

$$f_{\text{lerp}}(x) = y_i + \frac{x-x_i}{x_{i+1}-x_i}(y_{i+1} - y_i)$$



Credit: Wikipedia

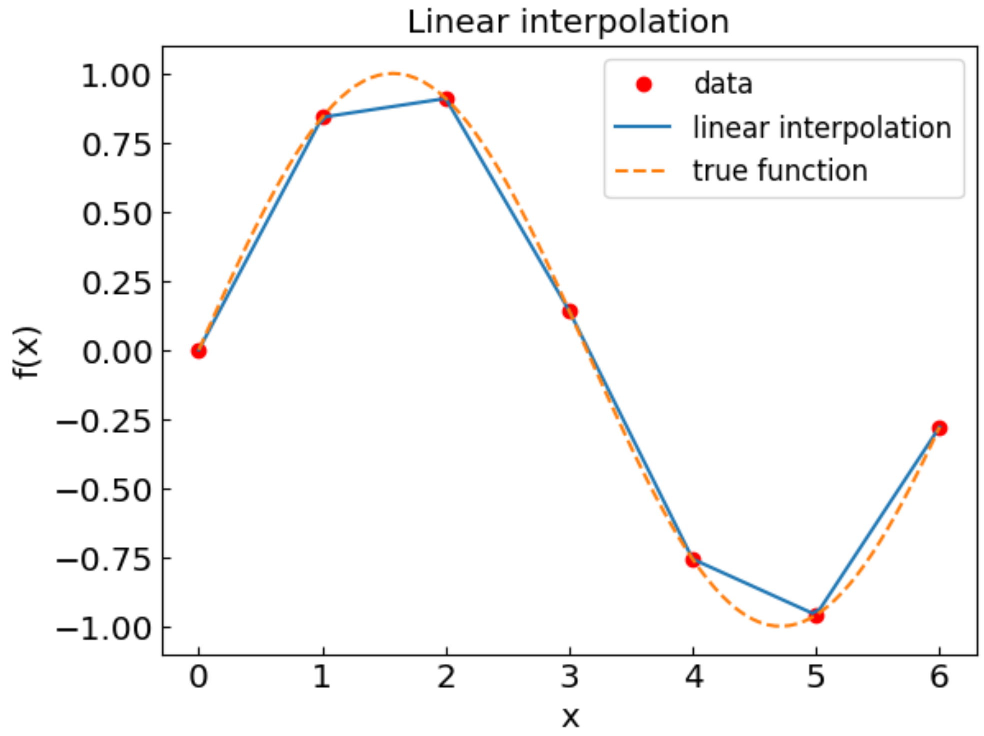
Linear interpolation

In Python:

```
def linear_int(x,x0,f0,x1,f1):
    """Returns the value of a function at point x
    through linear interpolation between points (x0,y0) and (x1,y1)."""
    return f0 + (f1 - f0) * (x-x0) / (x1-x0)


def f_linear_int(x, xdata, fdata):
    """Returns linear interpolation of a function at point x.
    xdata and ydata are the data points used in interpolation.
    xdata is assumed to be in sorted in ascending order."""
    ind = np.searchsorted(xdata, x) # Search the right interval for point x
    if (ind == 0):
        if ((xdata[0] - x) > 1e-12):
            print("x = ", x, " is outside the interpolation range [",xdata[0],",",xdata[-1],"]")
        ind = ind + 1
    if (ind == len(xdata)):
        if ((x - xdata[-1]) > 1e-12):
            print("x = ", x, " is outside the interpolation range [",xdata[0],",",xdata[-1],"]")
        ind = ind - 1
    x0,f0 = xdata[ind-1],fdata[ind-1]
    x1,f1 = xdata[ind],fdata[ind]
    return linear_int(x,x0,f0,x1,f1)

# Calculate the values of f(x) using the linear interpolation
xcalc = np.linspace(0,6,100)
fcalc = [f_linear_int(xin,xdat,fdat) for xin in xcalc]
```



Advantages:

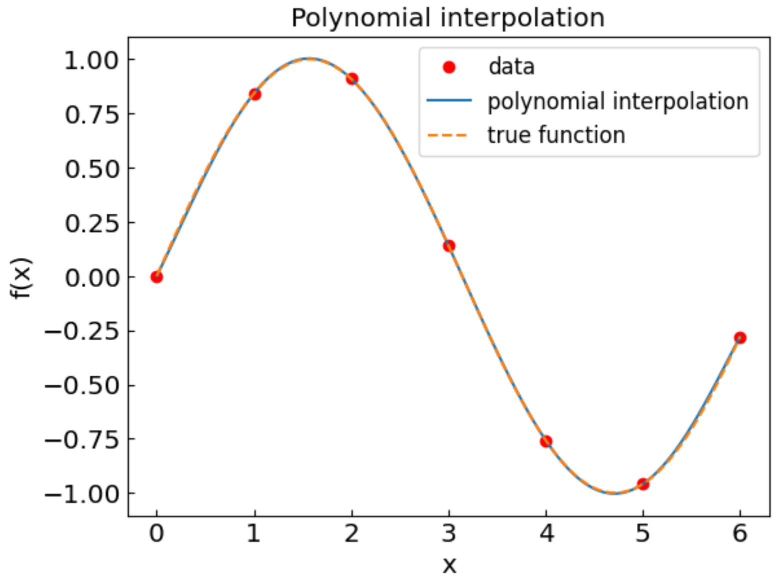
- Simple
- Generalizes to multiple dimensions
- More accurate than nearest-neighbor appr.

Disadvantages:

- Limited accuracy compared to polynomials
- Not good for derivatives

Polynomial interpolation (Lagrange form)

Theorem: There exists a *unique* polynomial of order n that interpolates through $n+1$ data points (x_0,y_0), (x_1,y_1), ..., (x_n,y_n)



How to build such a polynomial?

Consider ***Lagrange basis functions***:

$$L_j(x) = \prod_{k \neq j} \frac{x - x_k}{x_j - x_k}$$

Easy to see that for $x = x_k$ one has $L_j(x_k) = \delta_{kj}$

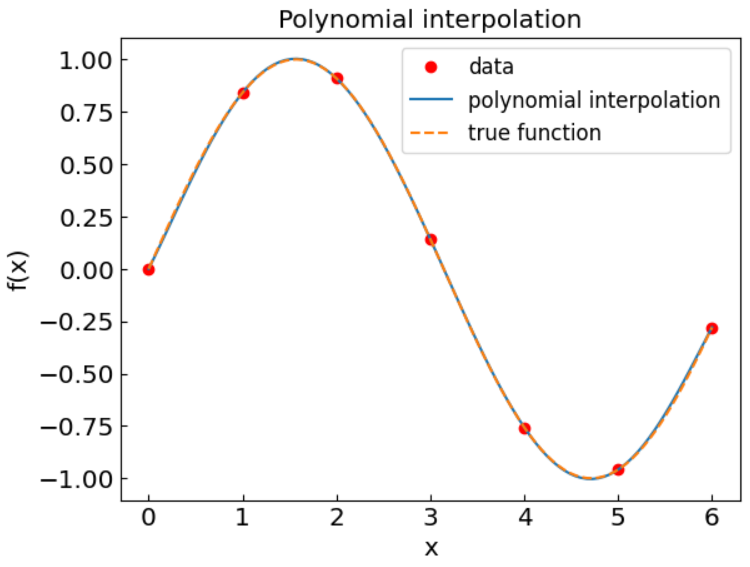
Therefore:

$$f(x) \approx p(x) = \sum_{j=0}^{n} y_j L_j(x)$$

Polynomial interpolation

For our example $f(x) = \sin(x)$

x	sin(x)
0	0.
1	0.841471
2	0.9092974
3	0.14112
4	-0.7568025
5	-0.9589243
6	-0.2794155



one obtains

$$p(x) = \sum_{j=0}^{n} y_j L_j(x)$$

$$= \sum_{j=0}^{6} \sin(x_j) \prod_{j\neq k}^{6} \frac{x - x_j}{x_k - x_j} \qquad \textbf{Lagrange form}$$

$$= -0.0001521x^6 - 0.003130x^5 + 0.07321x^4 - 0.3577x^3 + 0.2255x^2 + 0.9038x. \qquad \textbf{Canonical form}$$

In practice, the Lagrange form is more stable with respect to round-off errors

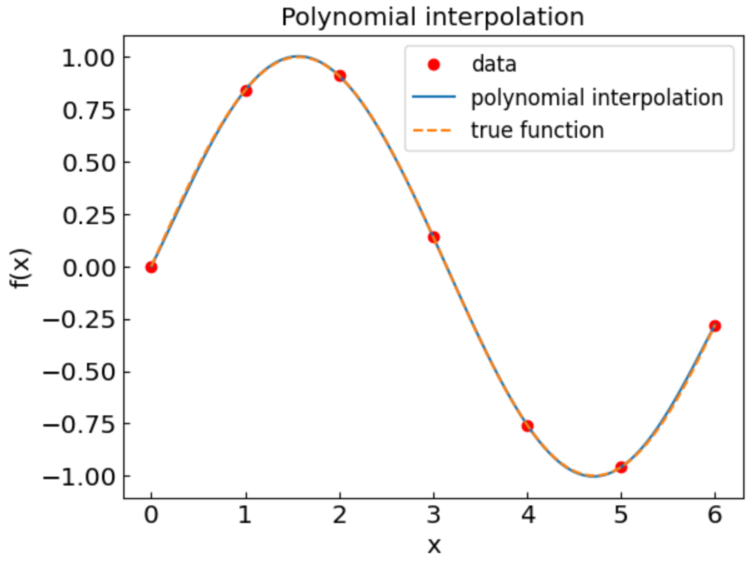
Polynomial interpolation

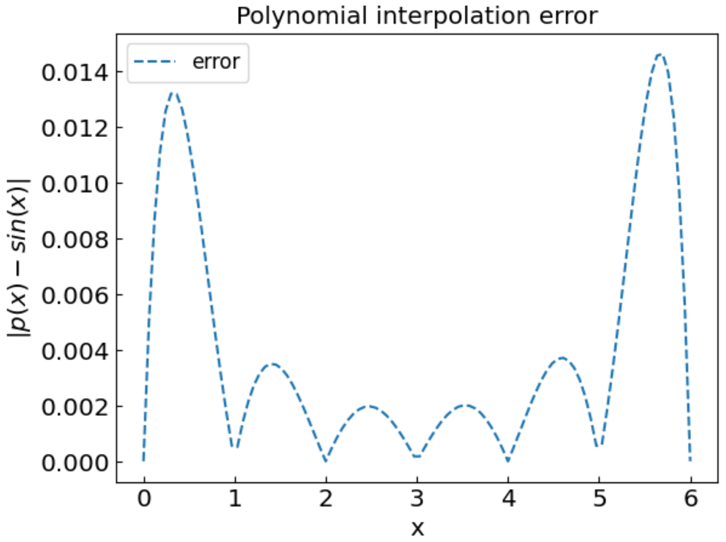
In Python:

```
def Lnj(x,n,j,xdata):
    """Lagrange basis function."""
    ret = 1.
    for k in range(0, len(xdata)):
        if (k != j):
            ret *= (x - xdata[k]) / (xdata[j] - xdata[k])
    return ret


def f_poly_int(x, xdata, fdata):
    """Returns the polynomial interpolation of a function at point x.
    xdata and ydata are the data points used in interpolation."""
    ret = 0.
    n = len(xdata) - 1
    for j in range(0, n+1):
        ret += fdata[j] * Lnj(x,n,j,xdata)
    return ret

xpoly = np.linspace(0,6,100)
fpoly = [f_poly_int(xin,xdat,fdat) for xin in xpoly]
```





Polynomial interpolation: Newton polynomial

Newton interpolating polynomial

$$p(x) = \sum_{j=0}^{n} f[x_0, x_1, \ldots, x_j] N_j(x).$$

Newton basis functions

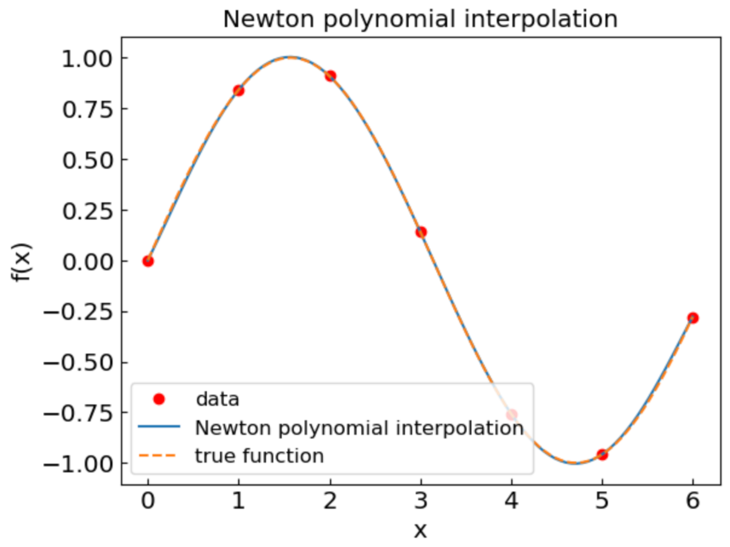
$$N_j(x) = \prod_{k=0}^{j-1} (x - x_k).$$

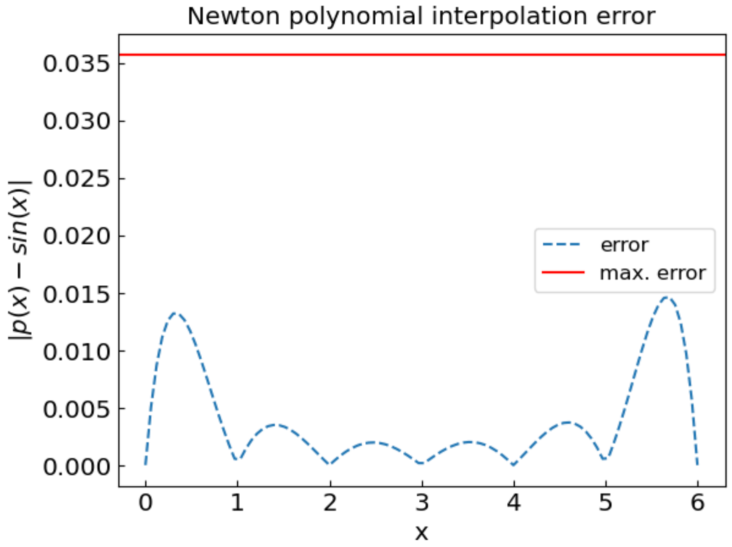
Divided differences

$$f[x_i] = f(x_i), \qquad f[x_i, x_{i+1}, \ldots, x_{i+j}] = \frac{f[x_{i+1}, \ldots, x_{i+j}] - f[x_i, \ldots, x_{i+j-1}]}{x_{i+j} - x_i}.$$

The polynomial itself is the same as Lagrange!

Advantage: Easier to incrementally add data points





Polynomial interpolation: Errors and artefacts

- Truncation errors
  - Lagrange remainder

$$R_n(x) = \underbrace{\frac{f^{(n+1)}(\xi)}{n+1}}_{\textit{derivative factor}} \underbrace{\prod_{i=0}^{n}(x - x_i)}_{\textit{product factor}}$$

- Round-off errors
  - Especially for high-order polynomials

Truncation errors can be a problem if

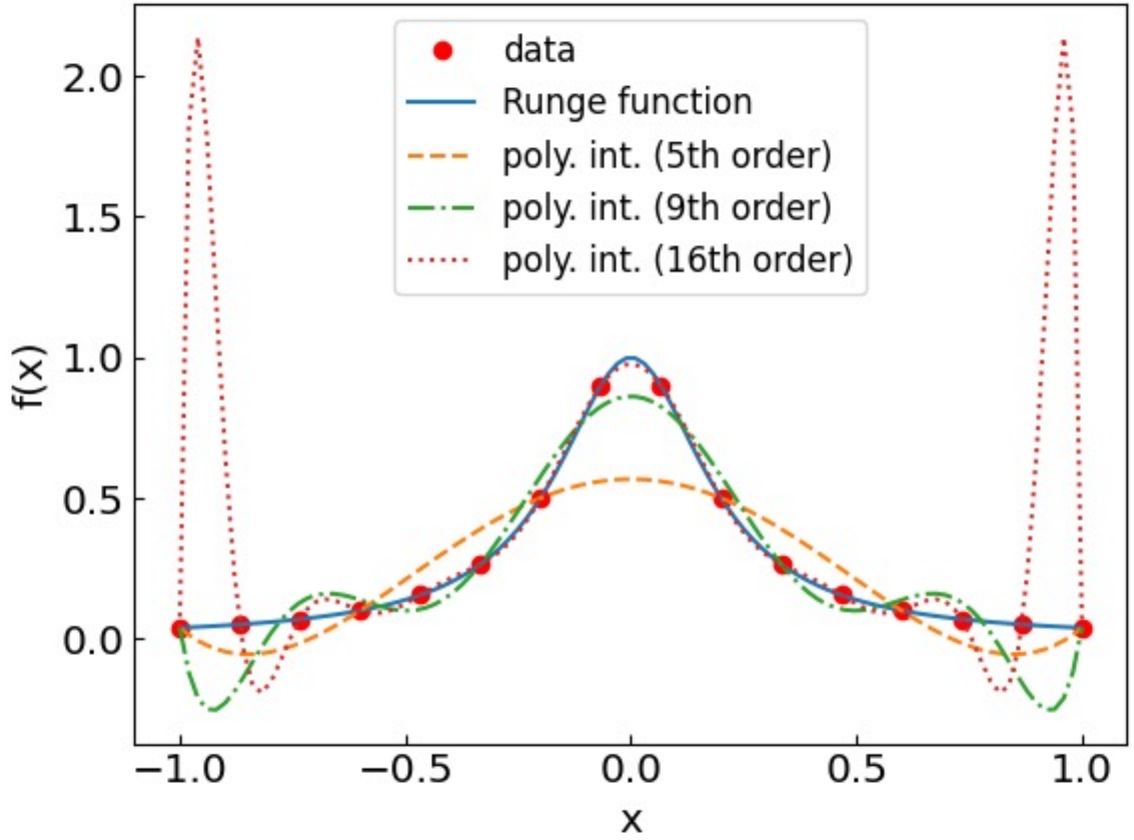
- High-order derivatives $f^{(n+1)}(x)$ of the function are significant
- The choice of nodes leads to a large value of the product factor

Runge phenomenon: Oscillation at the edges of the interval which gets *worse* as the interpolation order is increased

Polynomial interpolation: Runge phenomenon

Consider the Runge function: $$f(x) = \frac{1}{1+25x^2}$$

Let us do polynomial interpolation using equidistant nodes



Polynomial interpolation: Chebyshev nodes

Recall the truncation error

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{n+1} \prod_{i=0}^{n} (x - x_i)$$

So far, we used the equidistant nodes:

$$x_k = a + hk, \qquad k = 0, \dots, n, \qquad h = (b-a)/n$$

Can we choose the nodes x_i differently to minimize the product factor?

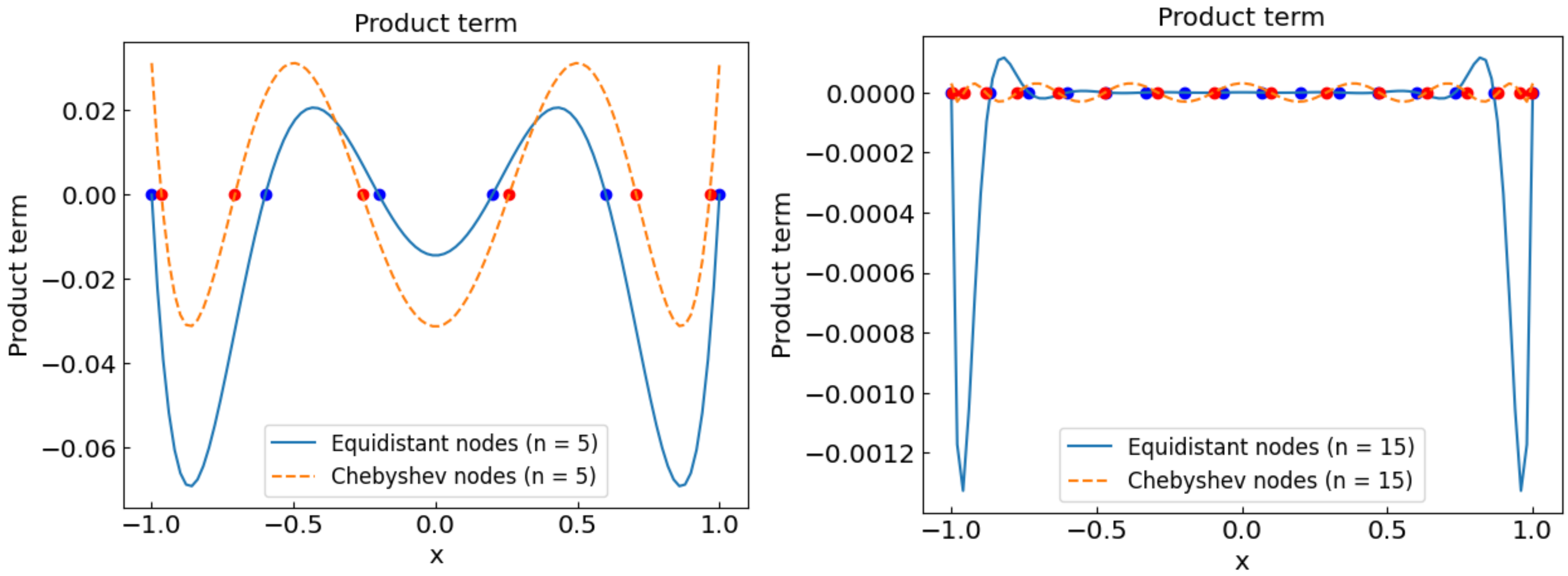
Yes!

Chebyshev nodes:

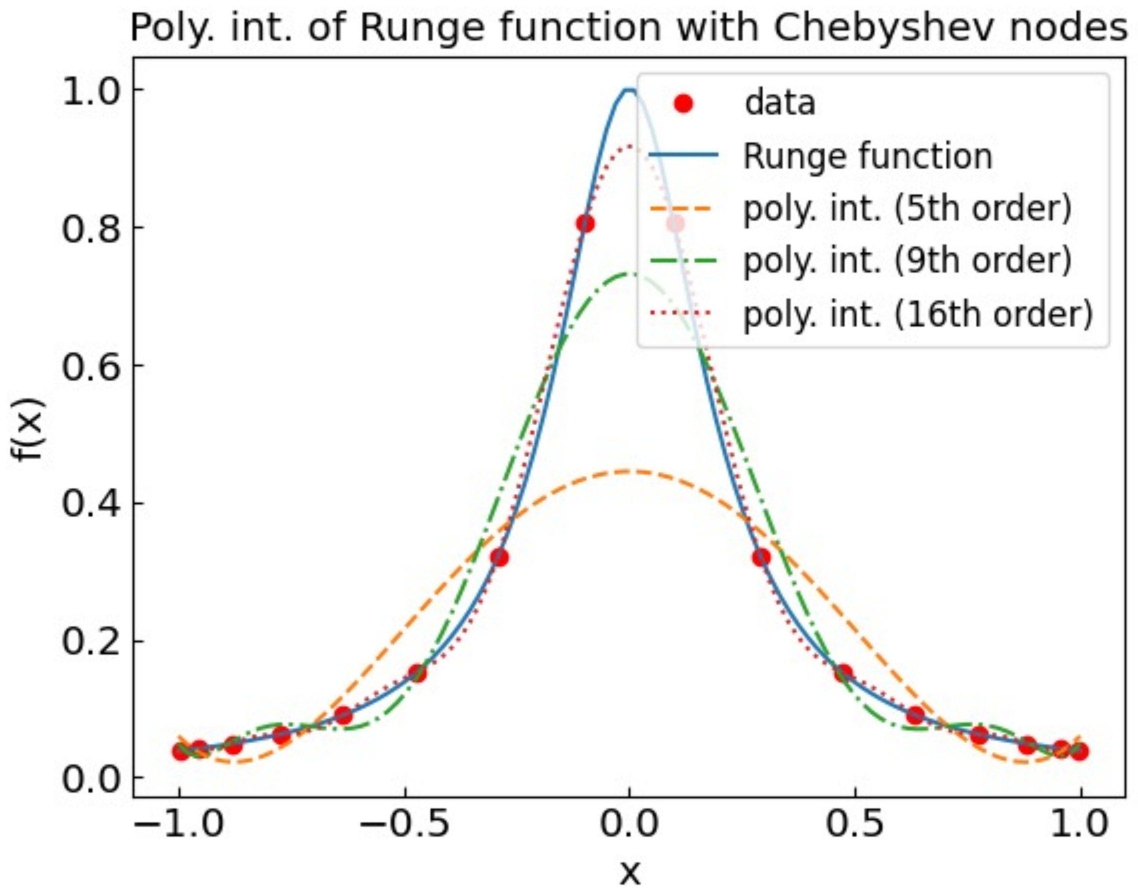
$$x_k = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{2k+1}{2n+2}\pi\right), \qquad k = 0, \dots, n,$$

Equidistant vs Chebyshev nodes

Plot $\prod_{i=0}^{n}(x - x_i)$ as a function of x for different number of nodes n on a $(-1,1)$ interval



Back to the Runge function: Chebyshev nodes



Polynomial interpolation: Summary

Advantages:

- Generally, more accurate than the linear interpolation
- Derivatives are continuous
- Can be used for numerical integration and differential equations

Disadvantages:

- Implementation not so simple
- Artefacts possible (such as large oscillations between nodes)
- Polynomials of large order susceptible to round-off errors
- Not easily generalized to multiple dimensions

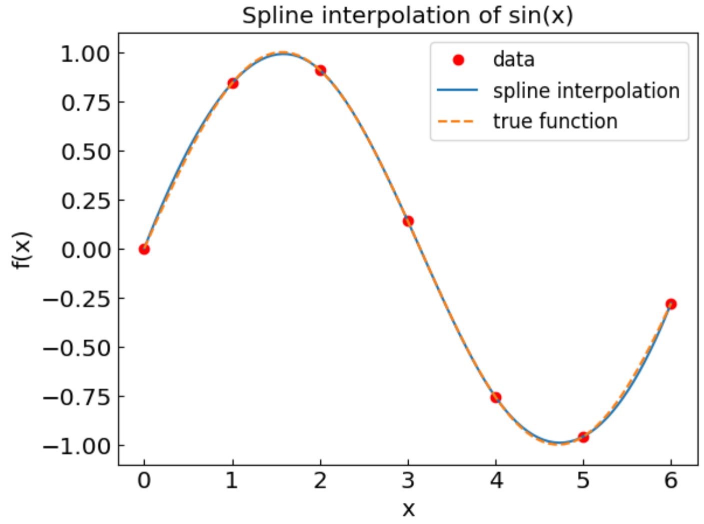
Spline interpolation

Connect each pair of nodes by a cubic polynomial

$$q_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i, \qquad x \in (x_i, x_{i+1})$$

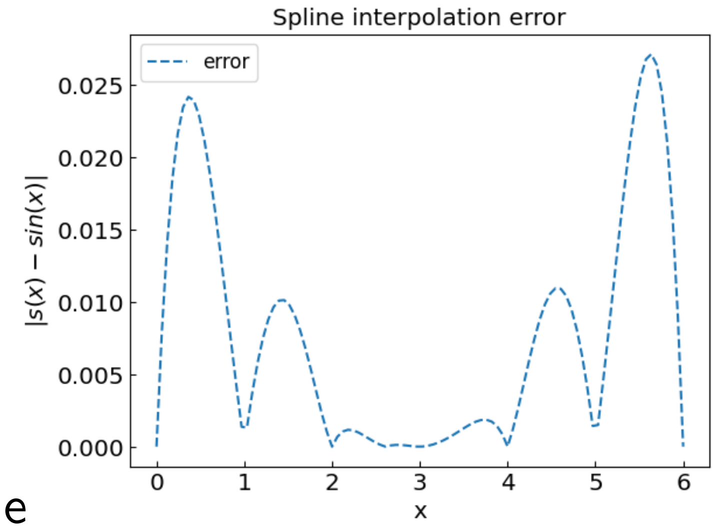
$4n$ coefficients a_i, bi, ci, di determined from

- data points and continuity ($2n$ equations)
- + continuity of 1st and 2nd derivatives ($2n$-2 equations)
- + boundary conditions for first derivative (2 equations)



Advantages:

- More accurate than linear interpolation
- Derivatives are continuous
- Avoids issues with polynomials of high degree



Disadvantages:

- Implementation not so simple
- Artefacts such as large oscillations between nodes are possible

Multiple dimensions

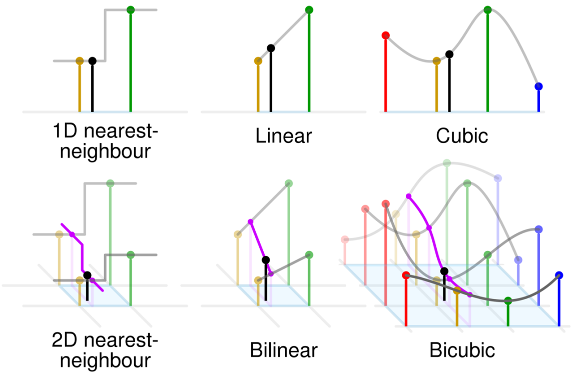
Functions of more than one variable, e.g. $f(x, y) = \sin(x + y)$

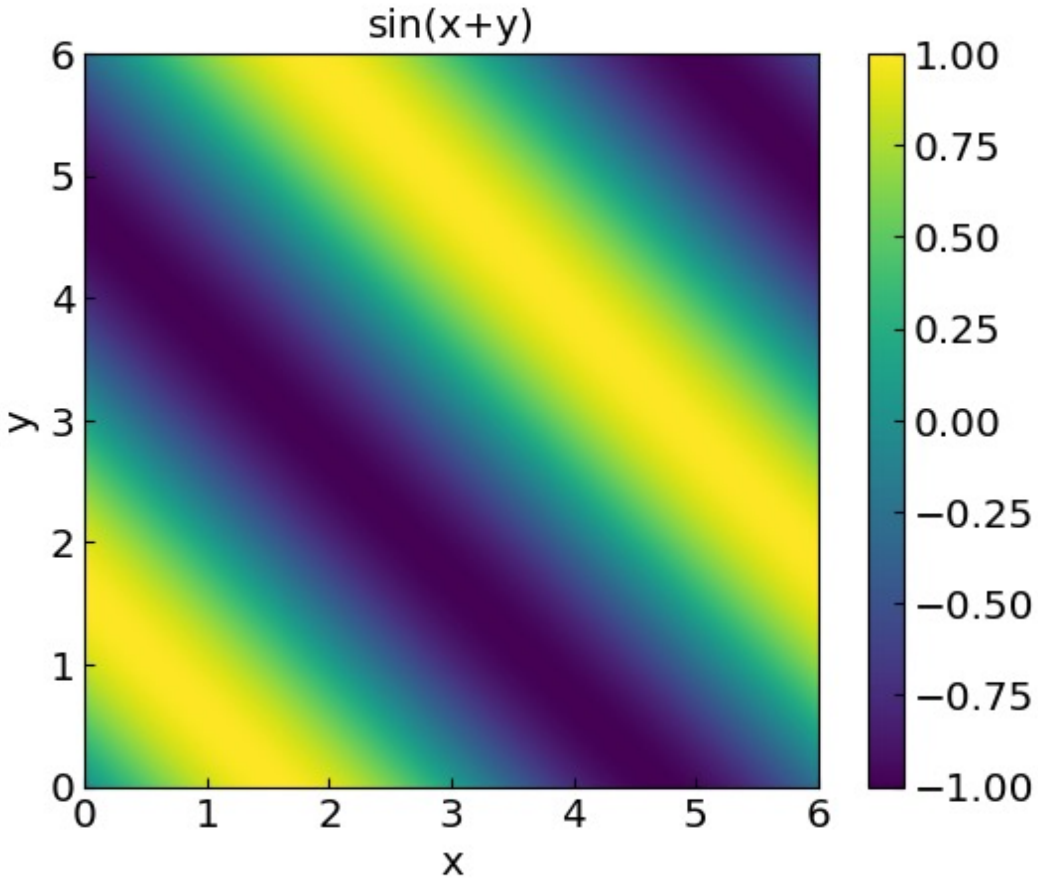
Data points: (x_i, yi, fi)

Main methods:

- Nearest-neighbor

- Successive 1D interpolations



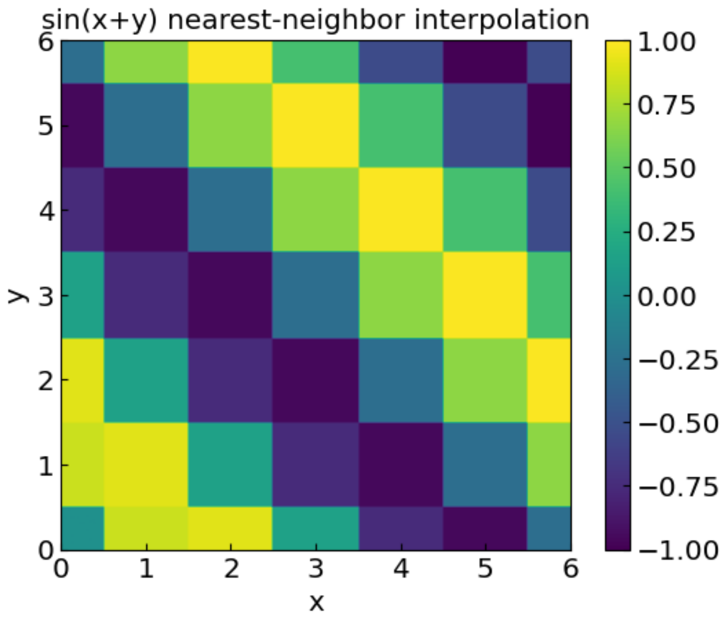


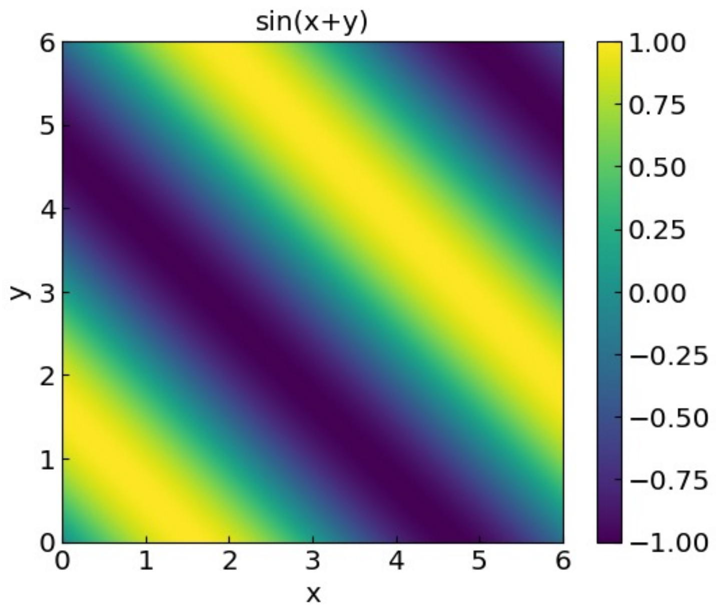
2D nearest-neighbor

2D nearest-neighbor:
Simply assign the value of the closest data point to *(x,y)* in the plane

Consider $f(x,y) = sin(x+y)$

Data points at integer values $x,y=0,1,\ldots 6$ (*regular grid*)

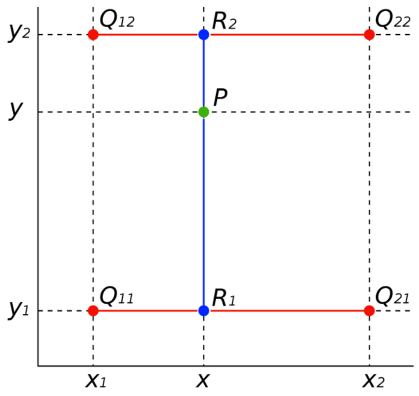


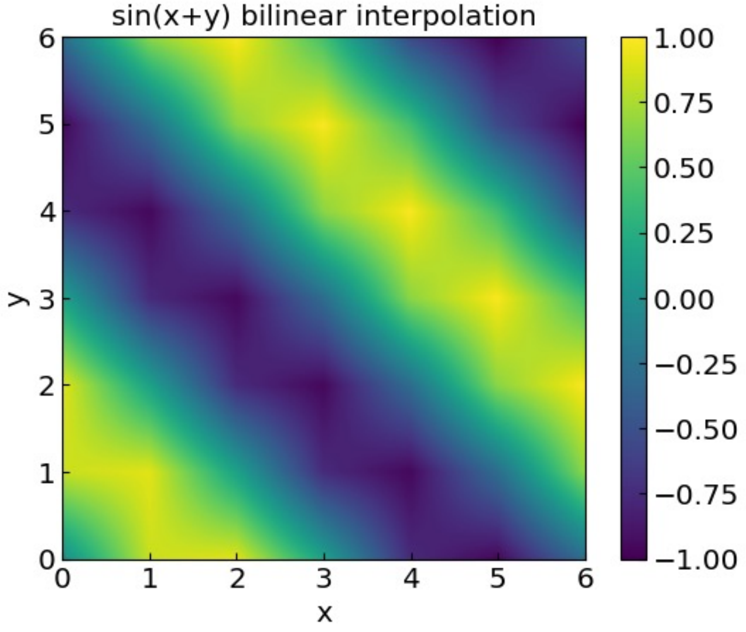


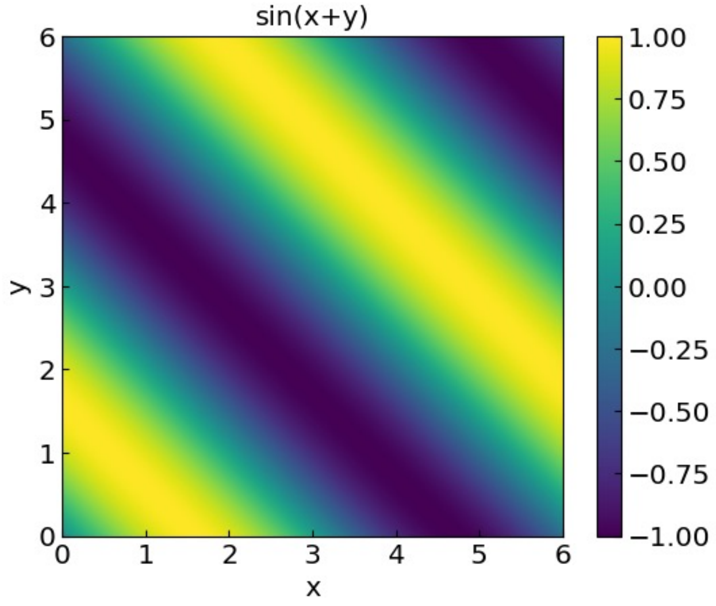
Bilinear interpolation

Bilinear interpolation: apply linear interpolation twice

1. Find (x_1, x_2) and (y_1, y_2) such that $x \in (x_1, x_2)$ and $y \in (y_1, y_2)$
2. Calculate R_1 and R_2 for $y = y_1$ and $y = y_2$, respectively, by applying linear interpolation in x
3. Calculate the interpolated function value at (x, y) by performing linear interpolation in y using the computed values of R_1 and R_2





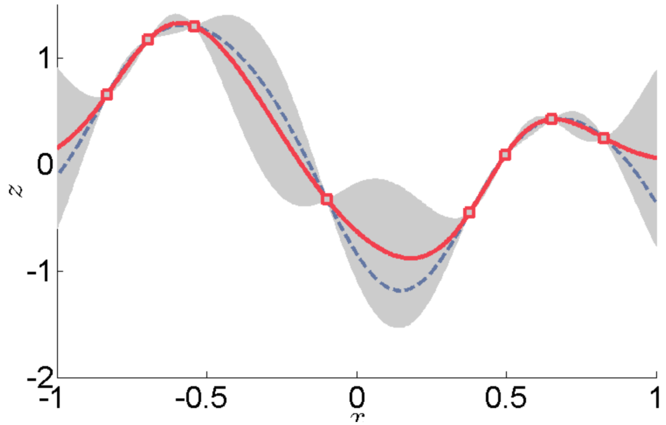


Advanced topics (further reading)

Gaussian process regression (Kriging)

- Uses prior assumption on covariances
- Provides a measure of uncertainty
- Extendable to noisy data and multiple dimensions

final project idea(?)



Hermite interpolation

- Interpolation of both the function values and derivative
- Can be polynomial or splines

final project idea(?)