Review: numerical linear algebra

Solving systems of linear equations:

Gauss elimination (with and without pivoting) – eliminating unknowns one by one until system becomes triangular (elimination phase) + solving for each in sequence (back substitution phase).

LU decomposition:

Every matrix can be split in a product of lower and upper triangular matrices. Split is not unique. Useful e.g. when solving the same system of equations multiple times with different RHS.

Ill-conditioned matrices:

Difficult to handle. Require special methods as standard ones fail.

Banded (esp. triangular) matrices:

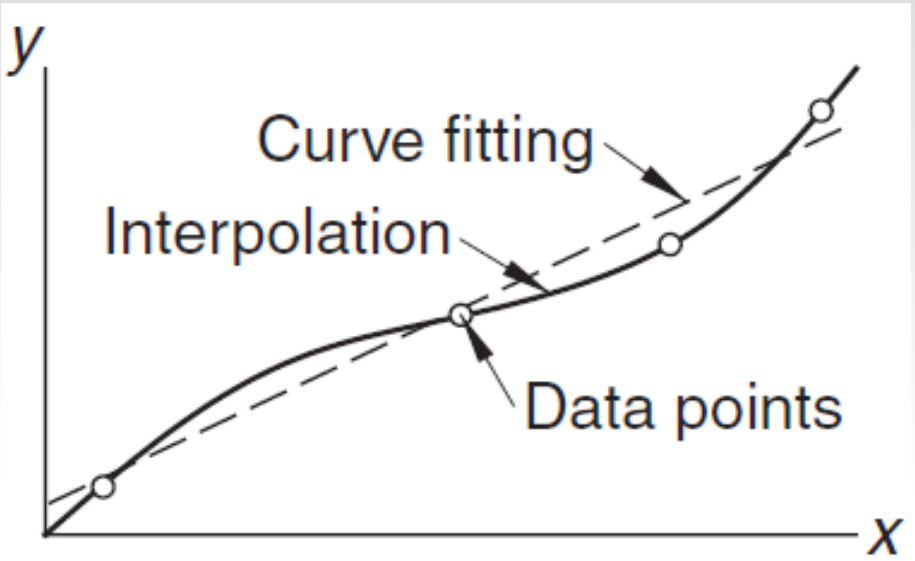
Occur in many applications. Such systems of equations are cheaper to solve than generic ones.

Iterative methods:

Slower than direct methods, but often have advantages – can do larger systems and are self-correcting.

Interpolation and Curve Fitting

Goal: given a series of data points, find a curve that passes through all points (interpolation) or best fits scatter points with noise (curve fitting), and that allows evaluation everywhere within the interval defined by the data (otherwise it is extrapolation) OR



Replace a function with a simpler one (e.g. one easier to integrate/evaluate)

x_0	x_1	x_2	$\cdots$	x_n
y_0	y_1	y_2	$\cdots$	y_n

We will look at two basic interpolation techniques (many more exist):

- polynomial interpolation
- splines (local low-order polynomials + continuity)

and Least Square Fitting for curve fitting.

Polynomial interpolation

Why use polynomials?

easy (fast and accurate) to evaluate, easy to integrate

can approximate any continuous function to any desired precision (Weierstrass theorem – Karl Weierstrass, 1885)

Side (but important) remark: efficient evaluation of a polynomial (Horners rule): take the polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

we can evaluate this with just n multiplications and n additions:

$$p(x) = \{[(a_n x + a_{n-1})x + a_{n-2}]x + \ldots + a_1\}x + a_0$$

Q: Why is this much better than the brute-force approach?

Lagrange interpolation polynomials

How do we construct such an interpolating polynomial?

In general, we need N+1 coefficients, $a_0, a_1, \ldots, a_N$ to uniquely specify a polynomial of order N.

We want a polynomial p(x) for which $p(x_i)=y_i$, i = 0,...,N (i.e. it goes through all the data points y_i) → a system of N+1 linear equations with N+1 unknowns.

This means that we can interpolate N+1 points with a polynomial of order N and this polynomial is unique.

E.g.: polynomial through 2 pts. (x_1,y_1) and (x_2,y_2)

Solving this system of equations usually is not the best way to go, though … Can we be smarter about it?

Lagrange interpolation polynomials

Imagine we had N polynomials $\ell_j(x)$ such that $\ell_j(x_k) = \delta_{jk}$.

Then,

$$p(x) = \sum_{j=0}^{N} y_j \ell_j(x)$$

is the interpolating polynomial - it clearly is a polynomial, and $p(x_k) = y_k$, for k=0,...,N.

How do we construct the Lagrange basis polynomials ℓ_j?

Lagrange basis polynomials

Remember $\ell_j(x_k) = \delta_{jk}$, so $\ell_j(x)$ needs to be zero for all $x=x_k$ except for $k=j$.

$$\ell_j(x) \propto \prod_{k \neq j} (x - x_k) = c(x - x_0) \cdots (x - x_{j-1})(x - x_{j+1}) \cdots (x - x_N)$$

the proportionality constant c is chosen to ensure that $\ell_j(x_j) = 1$, $c=1/P(x_j-x_k)$:

$$\ell_j(x) = \prod_{k \neq j} \frac{(x - x_k)}{(x_j - x_k)} = \frac{(x - x_0) \cdots (x - x_{j-1})(x - x_{j+1}) \cdots (x - x_N)}{(x_j - x_0) \cdots (x_j - x_{j-1})(x_j - x_{j+1}) \cdots (x_j - x_N)}$$

or,

$$P(x) = \sum_k \left(\prod_{j \neq k} \frac{x - x_j}{x_k - x_j} \right) y_k.$$

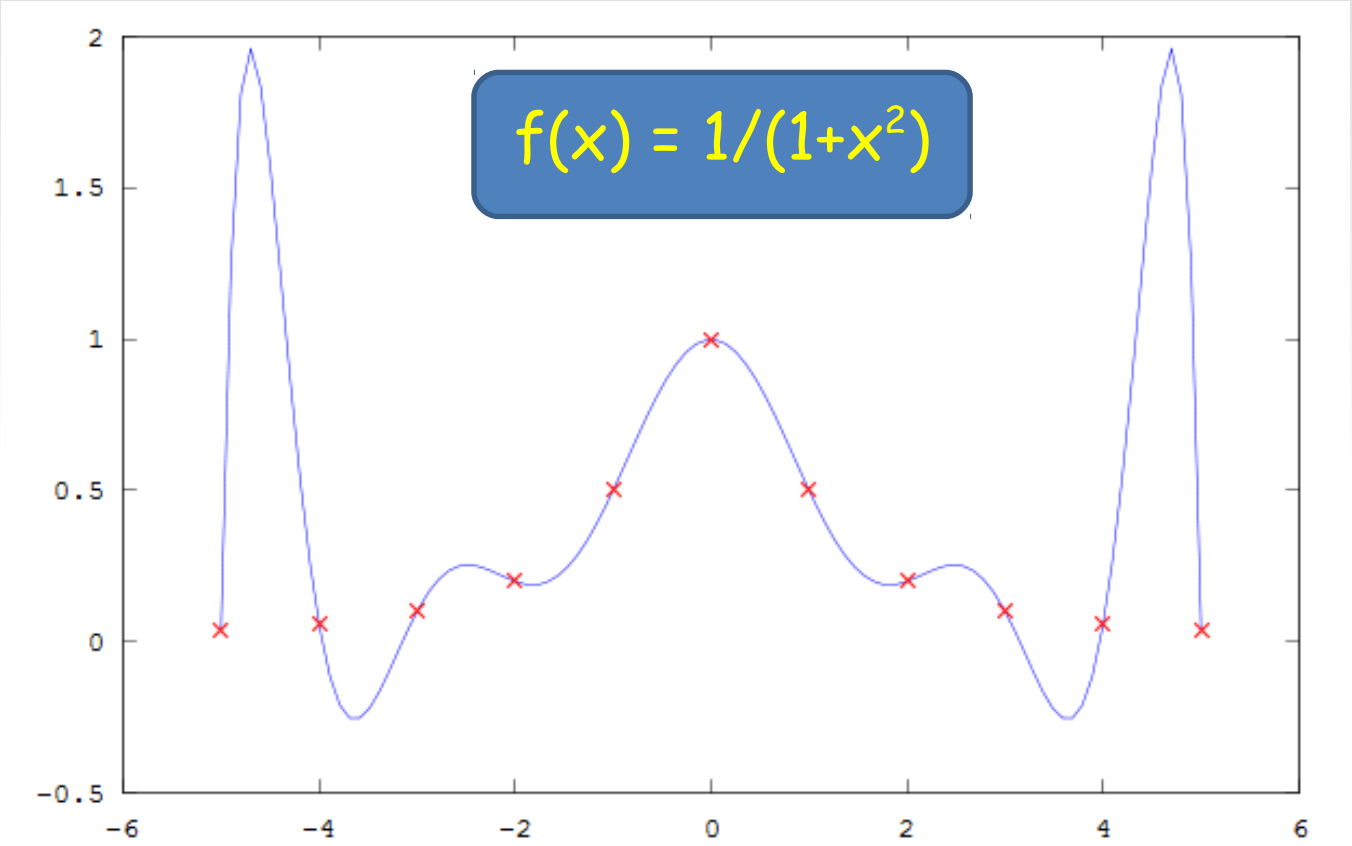
Python in-build polynomial interpolation

Polynomial interpolation is implemented as part of the Scipy package scipy.interpolate:

```
>import scipy
>from scipy import interpolate

>y=scipy.interpolate.barycentric_interpolate(xi, yi, x)
# evaluates a polynomial going through the
# points y_i(x_i)  at points given by vector x,
# values are returned in a vector y
```

Problems with polynomial interpolation



High-order polynomials are very susceptible to spurious oscillations (this is called Runge phenomenon). It can be avoided by using a sequence of low-order polynomials called splines instead.

Splines

Splines are polynomials of certain (low) order N (different polynomials in between data points, or “knots”) which are continuous and N-1 times continuously differentiable at all knots.

Examples:

- linear spline: continuous sequence of straight lines connecting the knots – continuous, but not differentiable

- cubic spline: continuous sequence of cubic polynomial curves connecting the knots – twice differentiable

Q: Is the following a (cubic) spline?

$s(x) = x^3+2x$ for x in [-1,0]

$s(x) = 2x+2x^2$ for x in [0,1]

$s(x) = x^3-x^2+5x-1$ for x in [1,3]

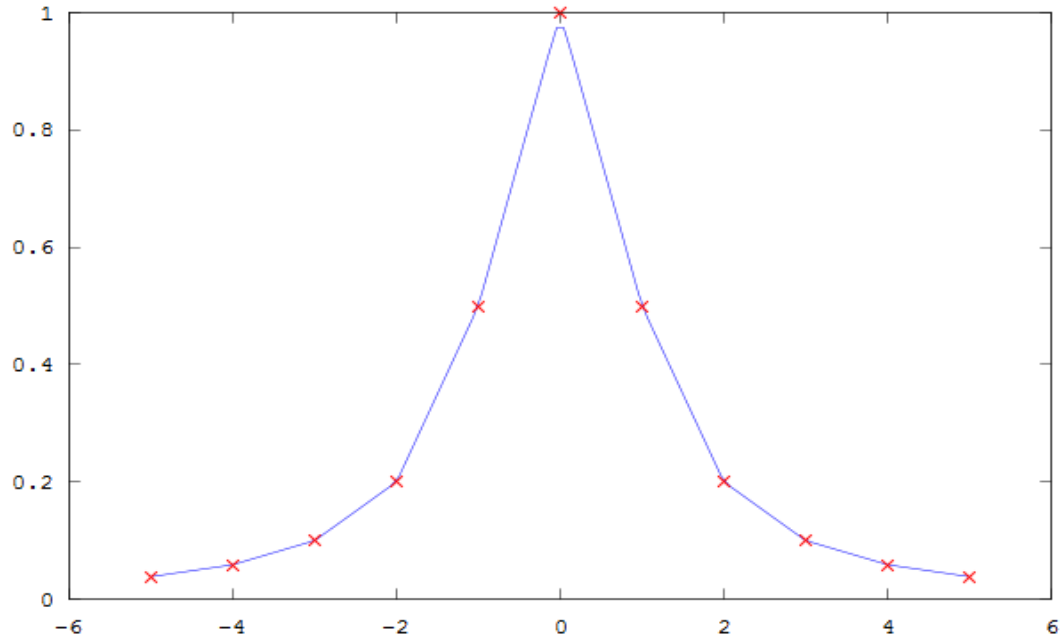
Constructing linear splines

Given a function f at knots $x_0 < x_1 < x_2 < \ldots < x_n$, what are the equations for straight lines which satisfy $s(x_k) = f(x_k)$, $k = 0,\ldots,n$?

We can write $s_k(x) = a_k + b_k(x-x_k)$ for the $[x_k, x_k+1]$ spline, then

$a_k = f(x_k)$ since s has to pass through node at x_k

$$b_k = \frac{f(x_{k+1}) - f(x_k)}{x_{k+1} - x_k}$$ to pass through node at x_{k+1}



Constructing cubic splines

A third-order polynomial has 4 coefficients.

The spline has to pass through both end-points (2 conditions) of each sub-interval and has to have the same 1st and 2nd derivative as the previous spline segment (2 more conditions) → this fully specifies the spline.

The first segment does not have the two conditions on the continuity of the derivatives → in total 2 free coefficients.

Those can be chosen arbitrarily, but one common and convenient choice, called natural spline picks them so that $s''(x_1) = s''(x_n) = 0$.

Cubic splines: implementation

```
## module cubicSpline
''' k = curvatures(xData,yData).
    Returns the curvatures of cubic spline at its knots.

    y = evalSpline(xData,yData,k,x).
    Evaluates cubic spline at x. The curvatures k can be
    computed with the function 'curvatures'.
'''
from numpy import zeros,ones
from LUdecomp3 import *

def curvatures(xData,yData):
    n = len(xData) - 1
    c = zeros(n)
    d = ones(n+1)
    e = zeros(n)
    k = zeros(n+1)
    c[0:n-1] = xData[0:n-1] - xData[1:n]
    d[1:n] = 2.0*(xData[0:n-1] - xData[2:n+1])
    e[1:n] = xData[1:n] - xData[2:n+1]
    k[1:n] =6.0*(yData[0:n-1] - yData[1:n]) \
                /(xData[0:n-1] - xData[1:n]) \
            -6.0*(yData[1:n] - yData[2:n+1])   \
                /(xData[1:n] - xData[2:n+1])
    LUdecomp3(c,d,e)
    LUsolve3(c,d,e,k)
    return k
```

```
def evalSpline(xData,yData,k,x):

    def findSegment(xData,x):
        iLeft = 0
        iRight = len(xData)- 1
        while 1:
            if (iRight-iLeft) <= 1: return iLeft
            i =(iLeft + iRight)/2
            if x < xData[i]: iRight = i
            else: iLeft = i

    i = findSegment(xData,x)
    h = xData[i] - xData[i+1]
    y = ((x - xData[i+1])**3/h - (x - xData[i+1])*h)*k[i]/6.0 \
      - ((x - xData[i])**3/h - (x - xData[i])*h)*k[i+1]/6.0   \
      + (yData[i]*(x - xData[i+1])                            \
       - yData[i+1]*(x - xData[i]))/h
    return y
```

Python in-build splines

standard 1D interpolation: interp1d(x,y,kind='str',*opt_args*)

different options for 'kind', e.g.:

linear → linear spline (default)

cubic → cubic spline

Usage:

```
>>> from scipy import interpolate
>>> x = np.arange(0, 10)
>>> y = np.exp(-x/3.0)
>>> f = interpolate.interp1d(x, y, kind='cubic')
>>> xnew = np.arange(0,9, 0.1)
>>> ynew = f(xnew)   # use interpolation function returned by `interp1d`
>>> plt.plot(x, y, 'o', xnew, ynew, '-')
>>> plt.show()
```

Least square fitting

- Data obtained in experiments typically contains noise due to measurement errors.
- Interpolation through points does not make much sense in this case → different approach is needed
- Find the best-fit simple (e.g. polynomial, exponent, …) curve through the cloud of points, so as to reproduce the physical law, but not the noise.
- How do we find this best fit curve? → least square fit

Least square fitting: method

- Let our data points be: $(x_i, y_i), \quad i = 0, 1, \ldots, n$
- We want to fit a function, $f(x) = f(x; a_0, a_1, \ldots, a_m)$

  where a_i are parameters to be determined and m<n
- The least square fit minimizes $S(a_0, a_1, \ldots, a_m) = \sum_{i=0}^{n} [y_i - f(x_i)]^2$
- Minimum is given by $\frac{\partial S}{\partial a_k} = 0, \quad k = 0, 1, \ldots, m$
- Often the fitting function is takes to be a linear combination of simple 'base' functions:

$$f(x) = a_0 f_0(x) + a_1 f_1(x) + \cdots + a_m f_m(x)$$

- How good is the fit is characterized by standard deviation: $\sigma = \sqrt{\frac{S}{n-m}}$

Example: fitting a straight line

- Straight line equation is $f(x)=a+bx$
- We need to find the best-fit coefficients a and b minimizing

$$S(a,b)=\sum_{i=0}^{n}\left[y_i-f(x_i)\right]^2=\sum_{i=0}^{n}\left(y_i-a-bx_i\right)^2$$

- Setting the derivatives to zero and solving for a and b yields

$$b=\frac{\sum y_i(x_i-\bar{x})}{\sum x_i(x_i-\bar{x})} \qquad a=\bar{y}-\bar{x}b$$

where

$$\bar{x}=\frac{1}{n+1}\sum_{i=0}^{n}x_i \qquad \bar{y}=\frac{1}{n+1}\sum_{i=0}^{n}y_i$$

are the means of the x and y measured values.

- Higher-order polynomials (and other functions) are done in a similar way (see book), but equations are more complicated, and in some cases non-linear.

Polynomial least squares fitting: implementation

- For polynomial fits we have $f_j(x)=x^j$

  and $f(x) = \sum_{j=0}^{m} a_j x^j$

- The fitting coefficients a_j are the solution of a linear system of equations (see book for details).

```
## module polyFit
''' c = polyFit(xData,yData,m).
    Returns coefficients of the polynomial
    p(x) = c[0] + c[1]x + c[2]x^2 +...+ c[m]x^m
    that fits the specified data in the least
    squares sense.

    sigma = stdDev(c,xData,yData).
    Computes the std. deviation between p(x)
    and the data.
'''
from numpy import zeros
from math import sqrt
from gaussPivot import *

def polyFit(xData,yData,m):
    a = zeros((m+1,m+1))
    b = zeros(m+1)
    s = zeros(2*m+1)
    for i in range(len(xData)):
        temp = yData[i]
        for j in range(m+1):
            b[j] = b[j] + temp
            temp = temp*xData[i]
        temp = 1.0
        for j in range(2*m+1):
            s[j] = s[j] + temp
            temp = temp*xData[i]
    for i in range(m+1):
        for j in range(m+1):
            a[i,j] = s[i+j]
    return gaussPivot(a,b)
```

In-build Python nonlinear curve fitting: example

```
import numpy as np
from scipy.optimize import curve_fit
def func(x, a, b, c):
    return a*np.exp(-b*x) + c

x = np.linspace(0,4,50)
y = func(x, 2.5, 1.3, 0.5)
yn = y + 0.2*np.random.normal(size=len(x))
popt, pcov = curve_fit(func, x, yn)
```

Uses non-linear least squares to fit a function, f, to data. Check the help for details.

Final remarks

Splines are very widely used in CAD, modelling, etc.

If you don’t know better/have special reasons, use a (natural) cubic spline.

If you can choose the location of your knots, you often can improve the accuracy of the interpolation.

(related to above) Your function may be easier to interpolate in other coordinates (e.g. log vs linear).

Multi-dimensional interpolation can be achieved by using repeatedly 1D interpolation in different dimensions, but it can be difficult to understand what is going on.