Chapter 3: Approximation $\checkmark$

3.1 Interpolation

- functions in chapter 1:
  - evaluation of function values $f(x)$ and Ax
  - approximation of real numbers and arithmetic expressions
- functions in chapter 2:
  - computing zeros x^* of functions, i.e., solution of equations $Ax = b$ and $f(x) = 0$
  - using functions $F(x)$ for iterative methods $x^{(k+1)} = F(x^{(k)})$
  - approximation of zeros x^*
- chapter 3:
  - approximation of functions $u(x)$ by simpler functions, in particular polynomials

Functions in scientific computing

- functions are not only arithmetic expressions
- they may solve complicated equations and usually are not known explicitely
  - they need to be approximated
  - these approximations are then used for predictions, diagnosis and decisions

$f(x) \quad x \in \mathbb{R}$

- some functions are univariate, and for example depend on time
  - average temperature, blood pressure
- other functions vary spatially
  - hyper-spectrum of pasture or forest
  - flow speeds of water in ocean
- many functions also depend on various parameters
  - flow through soil and rocks depends on density and other paramters
- some functions are random

$\mathbb{R}^d$ $\mathbb{R}$

- fundamentally, a function is a mapping $u : X \to Y$ with domain X and range Y
- here we will mostly consider $X = \mathbb{R}^d$ and $Y = \mathbb{R}$ $d = 1$
- there are now a variety of ways on how to determine a function.
  - they may be specified by a formula like $u(x) = \exp(-2x)$.
  - they may be defined implicitely, as the solution of some partial differential equation like $\Delta u = f$
  - one may only have partial and indirect information (measurements) of a function
- some functions satisfy equations with unknown parameters which may be determined from observations

Functions in Python

- one-liners using lambda

```
u = lambda x : x*x
```

$u(x) = x^2$

- Python procedures

```
def u(x):
  y = x*x
  return y
```

- imported from Python modules

```
from math import exp
```

3.1.1 Polynomial evaluation

Polynomials, their representation and evaluation

- mathematical form of polynomial

$$p_n(x) = a_0 + a_1 x + \cdots + a_n x^n$$

- simple python code

```
def pn(x,a):
  y = a[0]
  for k in range(1,len(a)):
    y += a[k]*x**k
  return y
```

$y = y + a[k] \cdot x^k$

```
# timing polynomial evaluation
def pn(x,a):
    y = a[0]
    for k in range(1,len(a)):
        y  += a[k]*x**k
    return y
```

same as on next page

cell

```
%%timeit
from numpy.random import random, seed;
n= 200; x = random();a = random(n)
y = pn(x,a) # timing polynomial

1000 loops, best of 3: 278 µs per loop
```

Using Cython to be faster

Cython video tutorial → SciPy meeting

14.7

(3, 7, 2)

```
%%cython
import cython
def pnc(double x, a, int n):
    cdef int k
    cdef double y
    y = a[0]
    for k in range(1,n):
        y  += a[k]*x**k
    return y
```

magic!

degree = n-1

C

R

cython cell

```
%%timeit
from numpy.random import random, seed; import cython;
n= 200; x = random(); a = random(n)
y = pnc(x,a,n) # timing polynomial
```

10000 loops, best of 3: 188 µs per loop

how to get faster code

- computational hardware costs substantially reduced in recent years
- code transformations used by compilers to get faster code
- faster code often by exploiting the *distributive law*

$$(a + b)c = ac + bc$$

- application to polynomial evaluation: *Horner's rule*

$$p_n(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots)))$$

$a_n \to a_n x + a_{n-1}$

$a_0 + a_1 x + a_2 x^2 + \cdots$ . .

```
# fast polynomial evaluation with Horner's method

def pnh(x,a):
    n = len(a)-1
    y = a[n]
    for k in range(n-1,-1,-1):
        y  = x*y + a[k]
    return y

%%timeit
from numpy.random import random, seed;
n= 200; x = random(); a = random(n)
y = pnh(x,a)

10000 loops, best of 3: 165 µs per loop
```

```
%%cython
import cython
def pnc(double x, a, int n):
    cdef int k
    cdef double y
    y = a[n-1]
    for k in range(n-1,-1,-1):
        y  = x*y + a[k]
    return y
```

```
%%timeit
from numpy.random import random, seed; import cython;
n= 200; x = random(); a = random(n)
y = pnc(x,a,n) # timing polynomial
```

```
10000 loops, best of 3: 100 µs per loop
```

Polynomial approximation and the Taylor polynomial

Weierstrass' theorem

Every continuous function over a finite interval can be approximated arbitrarily well by a polynomial of sufficiently high degree.

- we do not know in advance how high the degree has to be
- polynomial approximation works well for very smooth functions
- no quantitative error bound
- several proofs, including one using probability theory!

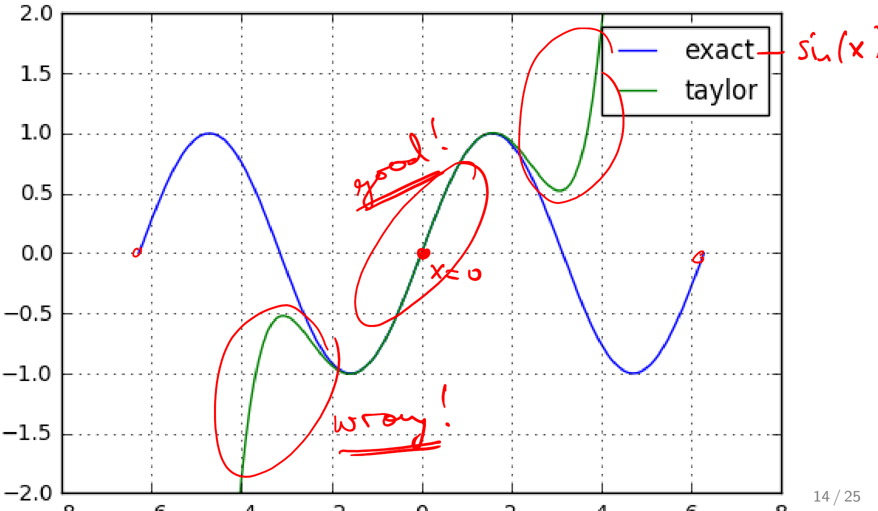
Taylor remainder theorem *If $u(x)$ is $n+1$ times continuously differentiable in $[a, b]$ then for all $x \in [a, b]$ there exists a $\xi \in [a, b]$ such that*

$$u(x) = u(a) + u'(a)(x-a) + \frac{u''(a)}{2}(x-a)^2 + \cdots + \frac{u^{(n)}(a)}{n!}(x-a)^n + \boxed{\frac{u^{(n+1)}(\xi)}{(n+1)!}}$$

- if $|u^{(n+1)}(x)| \leq C$ for all $x \in [a, b]$ then error of Taylor polynomial is bounded by $\cdot (x-a)^{n+1}$

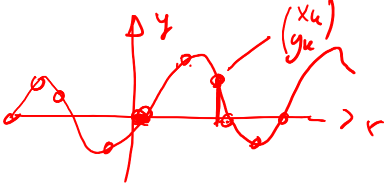
$$\frac{C(b-a)^{n+1}}{(n+1)!}$$

```
plt.grid('on')
plt.axis(ymin = -2, ymax = 2)
plt.plot(xg,yg, label="exact")
plt.plot(xg, ygt, label="taylor")
plt.legend();
```



3.1.2 Polynomial Interpolation

Collocation



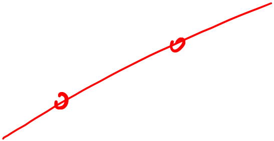
Proposition
There is exactly one polynomial p_n of degree n which satisfies the interpolation conditions

$$p_n(x_k) = y_k, \quad k = 0, \ldots, n$$

collocation

if all x_k are distinct

Proof by construction, will give 3 different approaches below which choose three different sets of basis functions for the linear space of polynomials of degree n



Approach 1: power basis x^k

$1 \quad x \quad x^n$

- if $p_n(x) = a_0 + a_1 x + \cdots + a_n x^n$, then the interpolation conditions lead to a linear system of equations for the a_k:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}$$

- the matrix X of this system is a *Vandermonde matrix*
- **Proposition:** if no two x_k are the same then X is invertible

Example

- $p_2(x) = a_0 + a_1 x + a_2 x^2$
- collocation points

i	0	1	2
x_i	0	0.5	2
y_i	0.2	0.6	-1.0

- system of equations for a_k:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0.5 & 0.25 \\ 1 & 2 & 4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.6 \\ -1 \end{bmatrix}$$

- solution $a_0 = 1/5$, $a_1 = 19/15$ and $a_2 = -14/15$
- interpolating polynomial

$$p_2(x) = 1/5 + 19/15x + -14/15x^2$$

```
# polynomial interpolant
```

Approach 2: cardinal basis l_j

- basis functions
$$l_j(x) = \prod_{k \neq j} \frac{x - x_k}{x_j - x_k}$$
- this forms a basis of the linear space of polynomials of degree n if the x_k are all distinct
- collocation matrix is identity
- basis functions satisfy
$$l_j(x_k) = \delta_{j,k}$$
where $\delta_{j,k}$ is Kronecker delta
  - thus they are the solution of a special interpolation problem
- interpolation polynomial
$$p_n(x) = \sum_{j=0}^{n} y_j l_j(x)$$
- no need to solve any equations!
- also called the *Lagrange form* of the interpolation polynomial

Example – cardinal functions

- for the data points

i	0	1	2
x_i	0	0.5	2
y_i	0.2	0.6	-1.0

the cardinal functions are

$$l_0(x) = \frac{(x-0.5)(x-2)}{(0-0.5)(0-2)} = (x-0.5)(x-2),$$

$$l_1(x) = \frac{(x-0)(x-2)}{(0.5-0)(0.5-2)} = -\frac{4}{3}x(x-2),$$

$$l_2(x) = \frac{(x-0)(x-0.5)}{(2-0)(2-0.5)} = \frac{1}{3}x(x-0.5).$$

Example – Lagrangian interpolant

$$p_2(x) = 0.2 * l_0(x) + 0.6 * l_1(x) - l_2(x)$$

- verification:
  1. $p_2(x)$ has degree at most 2
  2. satisfies interpolation conditions

$$\begin{aligned} p_n(x_j) = &\quad y_0 l_0(x_j) + \cdots + y_j l_j(x_j) + \cdots + y_n l_n(x_j) \\ = &\quad y_0 \cdot 0 + \cdots + y_j \cdot 1 + \cdots + y_n \cdot 0 \\ = &\quad y_j \end{aligned}$$

- uniqueness of this interpolant:
  - suppose $p(x)$ and $q(x)$ both satisfy collocation equations
  - then $r(x) = p(x) - q(x)$ is a polynomial of degree at most n
  - and $r(x)$ has $n + 1$ roots $x_0 \dots x_n$
  - thus $r(x)$ must be identically zero, and so $p = q$

```
# Lagrangian or cardinal polynomials lj(x)

npts = 6
```

Approach 3: Newton's basis $n_j(x)$

- basis functions $n_0(x) = 1$ and

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

  - collocation matrix is triangular
- interpolant for points $(x_0, y_0), \ldots, (x_k, y_k)$:

$$p_k(x) = \sum_{j=0}^{k} c_j n_j(x)$$

  NB: the c_j are independent of k!
- first polynomial $p_0(x) = y_0$
- recursion

$$p_{k+1}(x) = p_k(x) + c_{k+1} n_{k+1}(x)$$

  - substituting $x = x_{k+1}$ to get

$$c_{k+1} = \frac{y_{k+1} - p_k(x_{k+1})}{n_{k+1}(x_{k+1})}$$

Example Polynomial Interpolation

- same interpolation points

i	0	1	2
x_i	0	0.5	2
y_i	0.2	0.6	-1.0

- Newton's functions are

$$\begin{aligned} n_0(x) &= 1, \\ n_1(x) &= x, \\ n_2(x) &= x(x - 0.5) \end{aligned}$$

- and so

$$\begin{aligned} p_0(x) &= 0.2, \\ p_1(x) &= 0.2 + 0.8x, \\ p_2(x) &= 0.2 + 0.8x - \frac{14}{15}x(x - 0.5) \end{aligned}$$

 # Evaluation of all Newton polynomials -- -- TO BE CO

Another example

- another illustration of how the same polynomial is represented in three different forms
- Consider the polynomial $p_3(x) = 4x^3 + 35x^2 - 84x - 954$
- Show that the four points with coordinates $(5, 1)$, $(-7, -23)$, $(-6, -54)$ and $(0, -954)$ are on the graph of p_3

Example - Newton's Form

- the Newton functions are then
  $n_0(x) = 1, \quad n_1(x) = x - 5, \quad n_2(x) = (x-5)(x+7), \quad n_3(x) = (x-5)(x+7)(x+6)$
- An application of Newton's interpolation method gives then

$$p_3(x) = n_0(x) + 2n_1(x) + 3n_2(x) + 4n_3(x)$$