Python Cheatsheet

Pablo Angulo and Fabricio Macià for ESTIN@UPM.ES (layout by wzchen)

Errors

Types of error

Measurement error Noise, imprecision of measuring instrument, etc

Model error Our model is a simplification of the real word

Truncation error Replace a complicated or unknown function by a polynomial, etc

Rounding error Represent real numbers with finite precision, and perform computations with the approximations instead of the original numbers.

Big O

$f(x) = O(g(x))$ when $x \to a$

$f(x) = O(g(x))$ when $x \to a \Leftrightarrow |f(x)| \leq M|g(x)|$ when $|x-a| < \delta$, for some $\delta, M > 0$. If $a = 0$, Δx is small, and:

$$f(\Delta x) = p(\Delta x) + O(\Delta x^n)$$
$$g(\Delta x) = q(\Delta x) + O(\Delta x^m)$$
$$r = \min(n, m)$$

then

- $f + g = p + q + O(\Delta x^r)$
- $f \cdot g = p \cdot q + p \cdot O(\Delta x^m) + q \cdot O(\Delta x^n) + O(\Delta x^{n+m}) = p \cdot q + O(\Delta x^r)$

$f(x) = O(g(x))$ when $x \to \infty$

$f(x) = O(g(x))$ when $x \to \infty \Leftrightarrow |f(x)| \leq M|g(x)|$ when $x > K$, for some $K, M > 0$. If

$$f(x) = p(x) + O(x^n)$$
$$g(x) = q(x) + O(x^m)$$
$$k = \max(n, m)$$

- $f + g = p + q + O(x^k)$
- $f \cdot g = p \cdot q + pO(x^m) + qO(x^n) + O(x^{n+m}) = p \cdot q + O(x^{n+m})$

Taylor theorem

$$f(x) = \sum_{n=0}^{N} \frac{f^{(n)}(x_0) \cdot (x - x_0)^n}{n!} + O(x^{n+1})$$

Horner's nested evaluation

In order to evaluate $f(x) = a_0 + a_1\,x + \cdots + a_n\,x^n$ at $x = x_0$, place parenthesis like this:

$$f(x) = a_0 + x \cdot (a_1 + x \cdot (a_2 + \cdots + x \cdot a_n))$$

This reduces computing time and rounding error.

```
# coefs = [a0,a1,a2,...,an]
def Horner(x0, coefs):
    r = coefs[-1]
    for a in reversed(coefs[:-1]):
        r = r*x0 + a
    return r
```

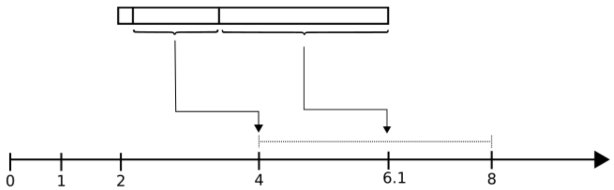
Floating point numbers

32-bit floating point

- One bit for the sign
- 8 bits for exponent
- 23 bits for mantissa

Intuition:

- The exponent chooses a window between two consecutive powers of 2: $[2^s, 2^{s+1}]$.
- The mantissa choose one of 2^{23} points regularly spaced in the interval $[2^s, 2^{s+1}]$. [1]



Root finding with `scipy.optimize`

Goal: Given $f : \mathbb{R} \to \mathbb{R}$, find $c \in \mathbb{R}$ st $f(c) = 0$.

Plot a function and find a root

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import bisect
def f(x):
    return x**3 - 4*np.sin(x) - 1
a, b = -2, 2
x0 = bisect(f, a, b)
xs = np.linspace(a-1,b+1,100) # regularly spaced points
plt.plot(xs,f(xs))            # draw f
plt.axhline(color='k')        # draw x axis
plt.plot([a, b], [0,0], 'o')  # initial interval
plt.plot([x0], [0], 'o')      # root
```

Bisection method

- A real-valued continuous function f defined on an interval $[a, b]$
- The signs of $f(a)$ and $f(b)$ are different $\Rightarrow$ by Bolzano theorem there is a root $c \in [a, b]$ st $f(c) = 0$.
- Guaranteed to find one root, but may pick any if there is more than one.
- Guaranteed to achieve precision $\frac{b-a}{2^n}$ after n iterations.

INIT Start with `interval = [a,b]`

REPEAT `c=(a+b)/2`. If `sign(f(a))!=sign(f(c))`, then set `interval = [a,c]`, otherwise it must hold that `sign(f(c))!=sign(f(b))`, and we set `interval = [c,b]`.

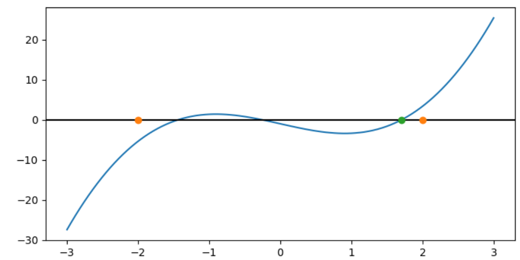
UNTIL Repeat until length of `interval` is smaller than `xtol`.

- Find an approximation to the root

```
x0 = bisect(f, a, b)
```

- Outputs information about convergence

```
x0, extra = bisect(f, a, b, full_output=True)
```



Secant method

- A function f with a simple root x_0 (e.g. $f'(x_0) \neq 0$).
- Convergence is faster than the bisection method.
- Generalizes to higher dimensions (*Broyden's method*).
- Needs a good initial approximation.
- Does not need the first derivative of f.

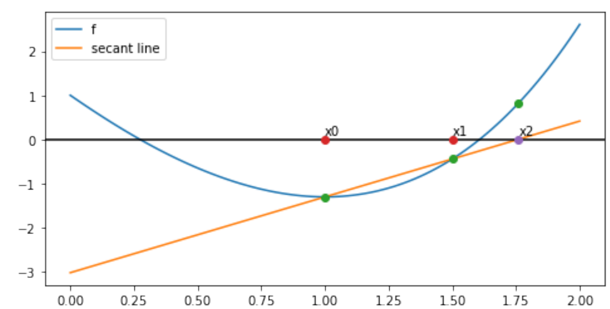
INIT Start with two approximations to the root `x0,x1`

REPEAT Compute a new point `x2 = x1 - f(x1)*(x1-x0)/(f(x1)-f(x0))` (root of the linear function through $(x_0, f(x_0))$ and $(x_1, f(x_1))$). Then advance the indices `x0,x1=x1,x2`.

UNTIL Two stop criterion are common (choose one):

x-tolerance Repeat until `np.abs(x1-x0)` is smaller than `xtol`.

y-tolerance Repeat until `np.abs(f(x1))` is smaller than `ytol`.



- Find an approximation to the root

```
from scipy.optimize import newton
xroot = newton(f, x0=xfirst, x1=xsecond)
```

- Find an approximation to the root, makes up xsecond if not provided

```
xroot = newton(f, x0=xfirst)
```

- Outputs information about convergence

```
xroot, extra = newton(f, x0=xfirst, x1=xsecond,
    full_output=True)
```

Newton method

- A function f with a simple root x_0 (e.g. $f'(x_0) \neq 0$).
- Convergence is quadratic.
- Generalizes to higher dimensions.
- Needs a good initial approximation.
- Needs the first derivative of f.

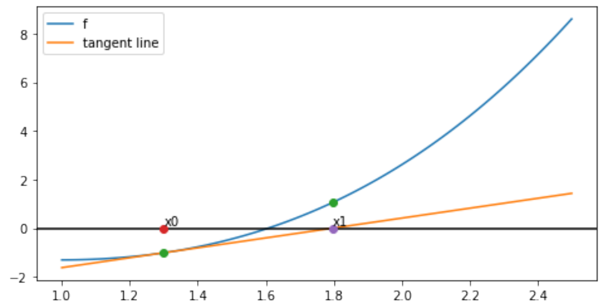
INIT Start with one approximations to the root `x0`

REPEAT Compute a new point `x1 = x0 - f(x0)/f'(x1)` (root of the linear function through $(x_0, f(x_0))$ with slope $f'(x_1)$. Then advance the indices `x1=x0`.

UNTIL Two stop criterion are common (choose one):

x-tolerance Repeat until `np.abs(x1-x0)` is smaller than `xtol`.

y-tolerance Repeat until `np.abs(f(x1))` is smaller than `ytol`.



- Find an approximation to the root, `fprime` was computed by hand :-/

```
xroot = newton(f, x0=xfirst, fp=fprime)
```

- Use `sympy` to compute derivative of `f`:

```
import sympy as sym
x = sym.symbols('x') # define a symbol
# y is a symbolic function
y = 1 + (x**3 - 4*x) + sym.log(1+x**2)
# derivative of y with respect to x
yder = sym.diff(y,x)
# lambdify builds a python function that accepts
    numpy arrays
f = sym.lambdify(x, y)
fp = sym.lambdify(x, yder)
xroot = newton(f, x0, fp=fp)
```

- Outputs information about convergence

```
x0, extra = newton(f, x0, fp, full_output=True)
```

Finding roots in higher dimension

```
from scipy.optimize import root
def F(xs):
    x,y=xs
    return y + np.log(x), x-np.sin(y)
output = root(F, [1,1]) # contains root and convergence
    information
F(output['x']) # output['x'] is a root => F(output['x
    ']) is almost zero
```

Interpolation

Interpolating polynomial

Definition

The *interpolating polynomial* of f through points $(x_0, y_0), \dots (x_n, y_n)$, where the x_i *are all different*, is the *unique* polynomial P of degree $\leq n$ such that $P(x_i) = y_i$ for $i = 0, 1, \dots, n$.

Error

When $y_i = f(x_i)$ for a $(N+1)$-times differentiable function:

$$\text{Error} = f(x) - P(x) = \frac{(x - x_0)(x - x_1)\dots(x - x_N)}{(N+1)!} f^{(N+1)}(\xi_x),$$

for some unknown point ξ_x in the interval of the points (x_i).

Vandermonde matrix

The coefficients $\overline{a} = (a_i)_{i=0}^n$ of P satisfy a linear system of equations:

$$\begin{aligned} a_0 + a_1\, x_0 + \dots + a_n\, x_0^n &= y_0 \\ a_0 + a_1\, x_1 + \dots + a_n\, x_1^n &= y_1, \\ &\vdots \\ a_0 + a_1\, x_n + \dots + a_n\, x_n^n &= y_n. \end{aligned}$$

or $V \cdot \overline{a} = \overline{y}$, for the Vandermonde matrix V of the points (x_i).

Lagrange form of the interpolating polynomial

$$P(x) := y_0 \ell_0(x) + y_1 \ell_1(x) + \dots + y_N \ell_N(x).$$

where

$$\ell_j(x) := \prod_{\substack{0 \leq m \leq N \\ m \neq j}} \frac{x - x_m}{x_j - x_m};$$

Newton's form

$$\begin{aligned} P(x) = \quad & f[x_0] + f[x_0, x_1](x - x_0) + \dots \\ & + f[x_0, \dots, x_N](x - x_0)(x - x_1)\dots(x - x_{N-1}). \end{aligned}$$

where $f[x_m, \dots, x_{m+j}]$ are the *divided differences*, defined for $j = 1, \dots, N$ and $m = 0, \dots, N - j$ as:

$$f[x_m, \dots, x_{m+j}] := \frac{f[x_{m+1}, \dots, x_{m+j}] - f[x_m, \dots, x_{m+j-1}]}{x_{m+j} - x_m}.$$

and for $m = 0, \dots, N$:

$$f[x_m] := f(x_m), \qquad m = 0, \dots, N$$

`numpy`'s `polyfit` and `polyval`

`polyfit` Returns the coefficients of the polynomial P of degree k that minimizes squared error between f and P through points `xs` (an approximation polynomial).

```
ys = f(xs)
coefs = np.polyfit(xs, ys, k)
```

`polyfit` If k is the number of points minus 1, then `polyfit` actually computes the interpolating polynomial:

```
coefs = np.polyfit(xs, ys, len(xs)-1)
```

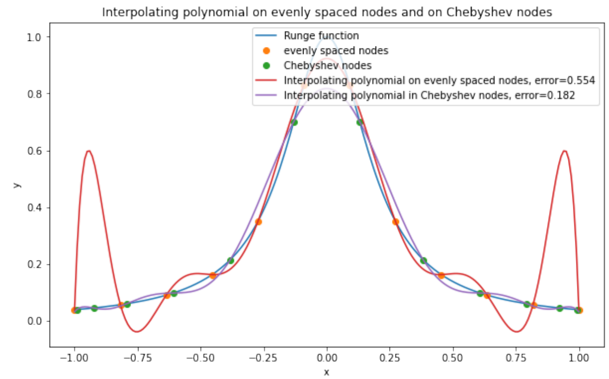
`polyval` Evals a polynomial on a set of points `xeval`, where the polynomial is given by its coefficients `coefs`.

```
xeval = np.linspace(-1,1,100) # 100 evenly spaced
    points, for plotting
yeval = np.polyval(coefs, xeval)
```

Chebyshev nodes

The interpolating polynomial does not need the points to be evenly spaced. Indeed, *evenly spaced points can lead to large errors* (Runge's phenomenon). This can be avoided if nodes are chosen carefully. For instance, Chebyshev nodes minimize the error. For the interval $[-1, 1]$, the nodes are:

$$x_k = \cos\left(\frac{(2k-1)\pi}{2N}\right) \quad \text{for} \quad k = 1, \dots, N$$



Hermite interpolation

Given $(n+1)$ different points $x_0, \dots, x_n$, we look for a polynomial $P(x)$ of degree $\leq 2n+1$ satisfying:

$$P(x_0) = y_0, \dots, P(x_n) = y_n$$

$$P'(x_0) = z_0, \dots, P'(x_n) = z_n.$$

- Theory is analogous to that for interpolating polynomials.
- It is possible to define Hermite polynomial interpolators that fit derivatives of degree higher than one.

Piecewise linear interpolation (Linear Splines)

The linear spline s_n interpolating through $(x_j, y_j)_{j=0}^n$ is defined piecewise: a linear (degree one polynomial) on each $[x_j, x_{j+1}]$ that interpolates the given values.

- This forces, for x in $[x_j, x_{j+1}]$:

$$s_n(x) = y_j + \frac{y_{j+1} - y_j}{x_{j+1} - x_j}(x - x_j).$$

- When $y_j = f(x_j)$ the error satisfies:

$$|f(x) - s_n(x)| \leq \frac{h^2}{8} \max_{[x_0, x_n]} |f''(x)|,$$

where

$$h = \max_{i=0,\dots,N-1} |x_{i+1} - x_i|.$$

- Create an `UnivariateSpline` object interpolating piecewise linearly points with x coordinates `xs` and y-coordinates `ys`:

```
from scipy.interpolate import UnivariateSpline
s = UnivariateSpline(xs, ys, k=1)
```

Cubic splines

The cubic spline interpolating through $(x_j, y_j)_{j=0}^n$ is defined piecewise: a cubic polynomial S_j in each interval $[x_j, x_{j+1}]$ such that:

- It interpolates the given values (this implies that S is continuous)
$$S_j(x_j) = y_j,\ S_j(x_{j+1}) = y_{j+1}$$
- It has a continuous first derivative. It is sufficient to check at the nodes:
$$S_j'(x_j) = S_{j+1}'(x_j)$$
- It has a continuous second derivative. It is sufficient to check at the nodes:
$$S_j''(x_j) = S_{j+1}''(x_j)$$

The above amounts to $4n-2$ conditions, for a total of $4n$ degrees of freedom (4 for each interval $[x_j, x_{j+1}]$). So there are two missing conditions, and there are several alternatives:

Natural boundary conditions $S_0''(x_0) = 0$, $S_n''(x_n) = 0$

Clamped boundary conditions $S_0'(x_0) = 0$, $S_n'(x_n) = 0$

'Not-a-knot' boundary conditions $S_1'''(x_0) = S_1'''(x_1)$, $S_{n-1}'''(x_{n-1}) = S_n'''(x_{n-1})$.

- Create a `CubicSpline` object interpolating points with x-coordinates `xs` and y-coordinates `ys`:

```
from scipy.interpolate import CubicSpline
cs3 = CubicSpline(xs, ys)
```

- Use different boundary conditions (default is *'not-a-knot'*)

```
from scipy.interpolate import CubicSpline
cs3 = CubicSpline(xs, ys, bc_type='natural')
```

- Eval a `CubicSpline` on a set of points `xeval`.

```
# 200 evenly spaced points, for plotting
xeval = np.linspace(min(xs), max(xs), 200)
# eval the cubic spline on xeval, and plot
plt.plot(xeval, cs3(xeval))
```