R. Foster
@TaiwanBirding

Plumbeous Redstart
鉛色水鶇

ASTR 660 / Class 9

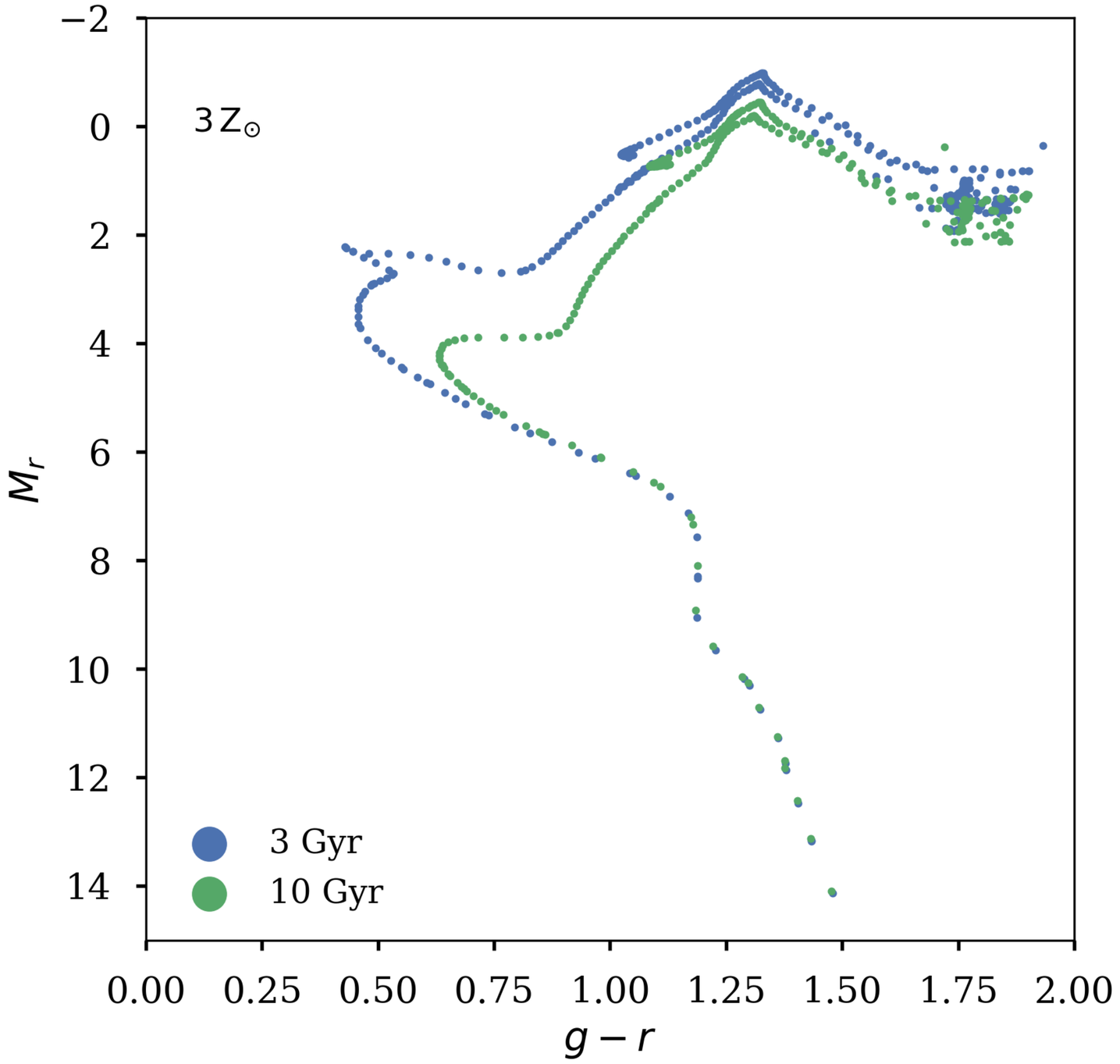
Interpolation

Interpolation

Very often, we end up with a table of data sampled at some particular points. For example, an isochrone from a stellar evolution model is a table of the properties of stars of a given initial mass, for a fixed age and metallicity:

http://stev.oapd.inaf.it/cmd/

Zini	MH	logAge	Mini	int_IMF	Mass	logL	logTe	logg	McoreTP	C_O	Mloss	Z	mbolmag	umag	gmag	rmag	imag	zmag	Ymag
0.0439	0.52753	6.63	0.1039686427	1.608425	0.104	-1.652	3.3943	3.636	0.0	0.545	-1.33E-14	0.04459	8.901	14.22	13.615	12.128	10.075	8.872	8.337
0.0439	0.52753	6.63	0.1130571142	1.690024	0.113	-1.62	3.3973	3.652	0.0	0.545	-1.35E-14	0.04459	8.819	14.142	13.536	12.049	9.994	8.79	8.255
0.0439	0.52753	6.63	0.1271817535	1.804008	0.127	-1.571	3.4021	3.674	0.0	0.545	-1.39E-14	0.04459	8.698	14.025	13.418	11.931	9.872	8.668	8.133
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
0.0439	0.52753	10.12995	0.9980760217	3.140275	0.657	3.474	3.3753	-0.765	0.53	0.471	-4.05E-06	0.0442	-3.915	2.728	3.371	1.596	-1.09	-2.636	-3.32

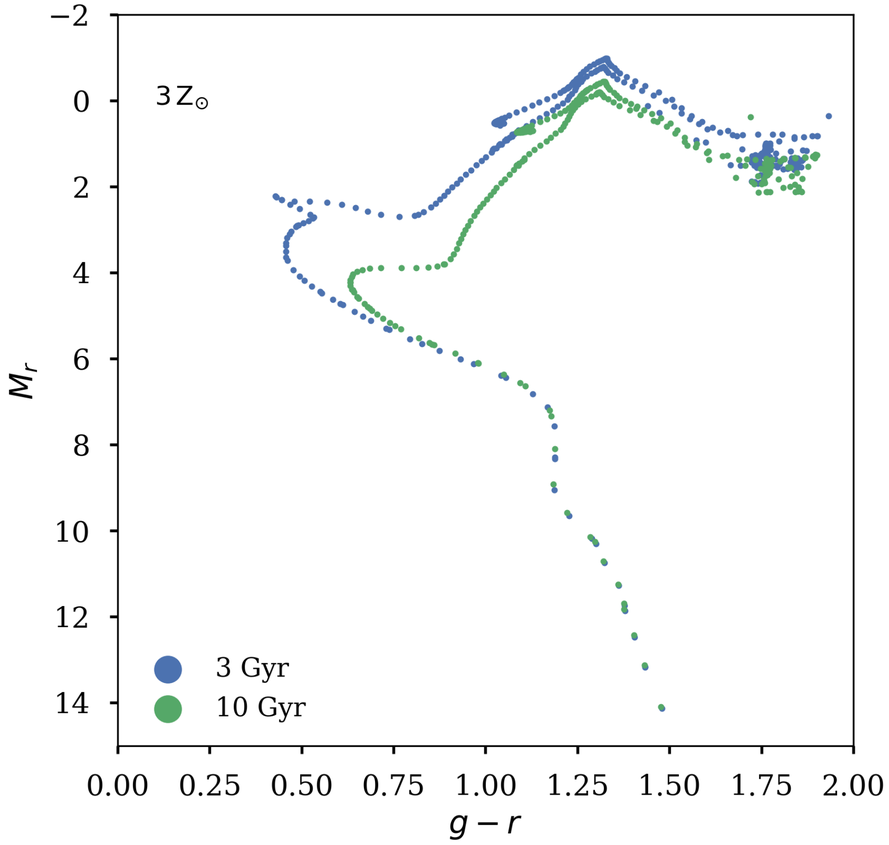

3 Z⊙
3 Gyr
10 Gyr
M_r
$g-r$
−2
0
2
4
6
8
10
12
14
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

Interpolation

What is the effective temperature, log g, or some other property, for a star of a given age, metallicity and initial mass?

This requires (at least) **interpolating** the initial mass along the isochrone, between the calculated grid points.

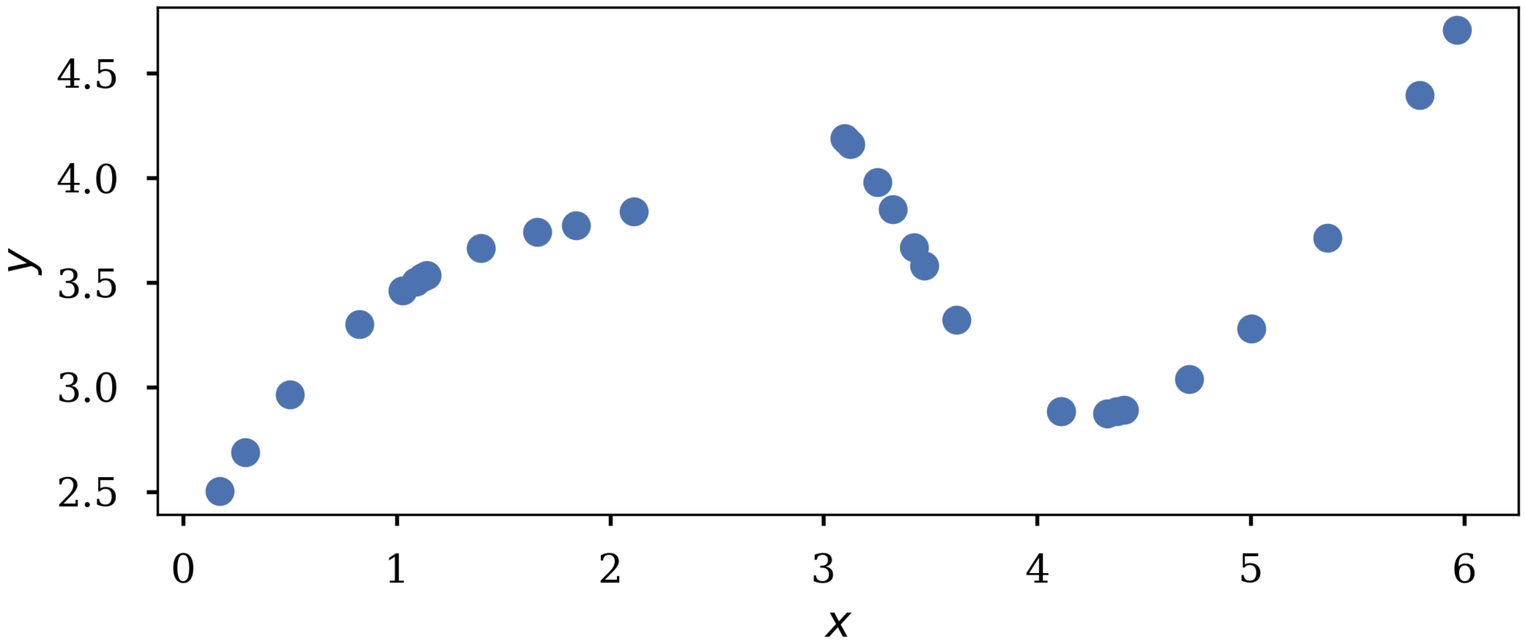
Choosing the values at the closest grid point is also a (simple) form of interpolation.



Interpolation

If we want the value of a property at a mass between two points in the table, and we can't re-run the original calculation, then we have to estimate a value, based on what we know.

There are other uses for a smooth, simple, continuous function that goes through discrete points; for example, finding derivatives and integrals of more complicated functions.



Interpolation

The general idea is that we take a complicated function (or sparsely sampled data) and approximate with an “easy” function (e.g. a polynomial).

Both scipy and numpy include almost equivalent basic interpolation routines. The numpy routines are slightly faster. Their interface is to **pass arrays in and get arrays out**.

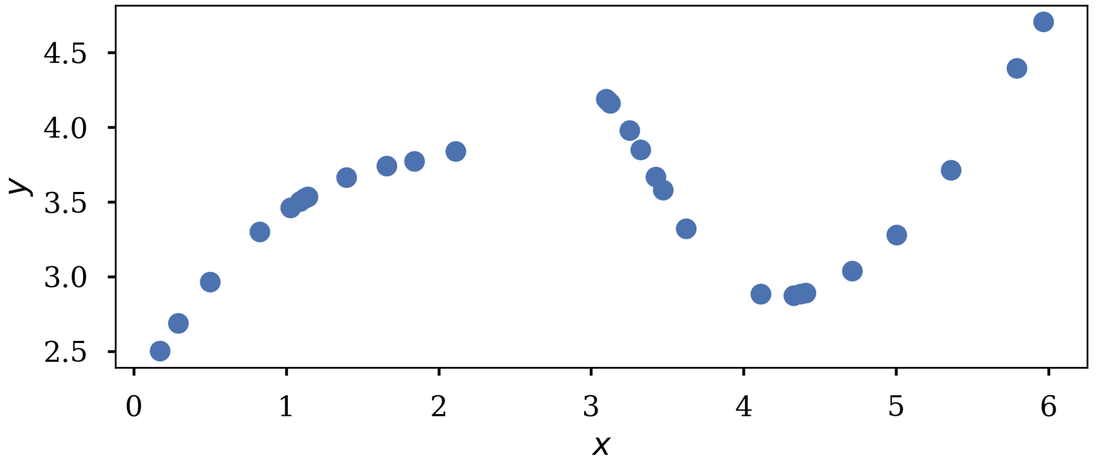
The scipy routines have a few more features. Their interface involves **passing arrays in** and returning **interpolating functions**, which can be evaluated at any point.

Interpolation in numpy

```
[111]: np.interp?
```

```
Signature:       np.interp(x, xp, fp, left=None, right=None, period=None)
Call signature:  np.interp(*args, **kwargs)
Type:            _ArrayFunctionDispatcher
String form:     <function interp at 0x10c8c9f80>
File:            /opt/miniconda3/envs/astr660/lib/python3.11/site-packages/numpy/lib/function_base.py
Docstring:
One-dimensional linear interpolation for monotonically increasing sample points.

Returns the one-dimensional piecewise linear interpolant to a function
with given discrete data points (`xp`, `fp`), evaluated at `x`.

Parameters
----------
x : array_like
    The x-coordinates at which to evaluate the interpolated values.

xp : 1-D sequence of floats
    The x-coordinates of the data points, must be increasing if argument
    `period` is not specified. Otherwise, `xp` is internally sorted after
    normalizing the periodic boundaries with ``xp = xp % period``.

fp : 1-D sequence of float or complex
    The y-coordinates of the data points, same length as `xp`.

left : optional float or complex corresponding to fp
    Value to return for `x < xp[0]`, default is `fp[0]`.

right : optional float or complex corresponding to fp
    Value to return for `x > xp[-1]`, default is `fp[-1]`.
```

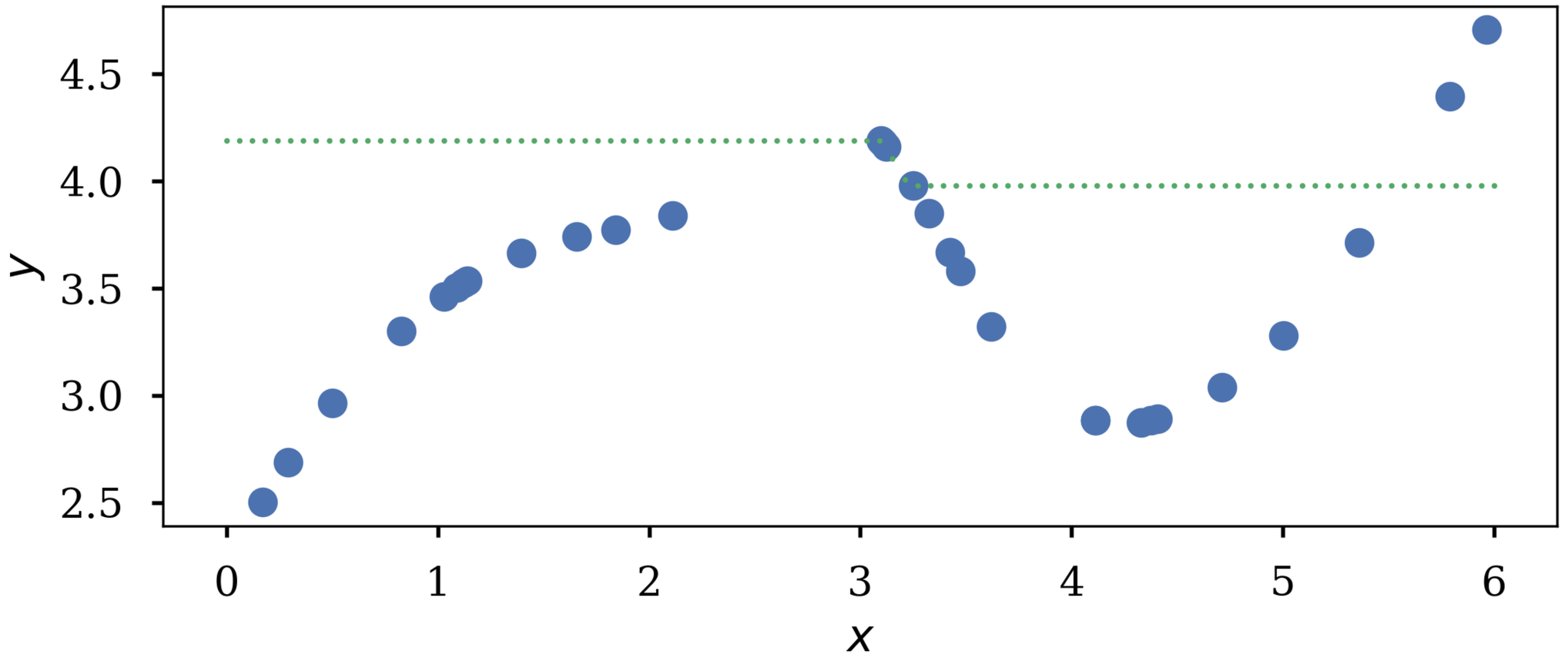
Interpolation in numpy



Let's try this basic numpy interpolation routine:

```
np.random.seed(101)
x = np.random.random(30)*6
y = 2+np.sin(x)-0.5*x+0.1*x**2+np.sqrt(x) + np.exp((-(x-3)**2)/0.4)

xgrid = np.linspace(0,6,100)
y_interp = np.interp(xgrid,x,y)
```

Interpolation in numpy

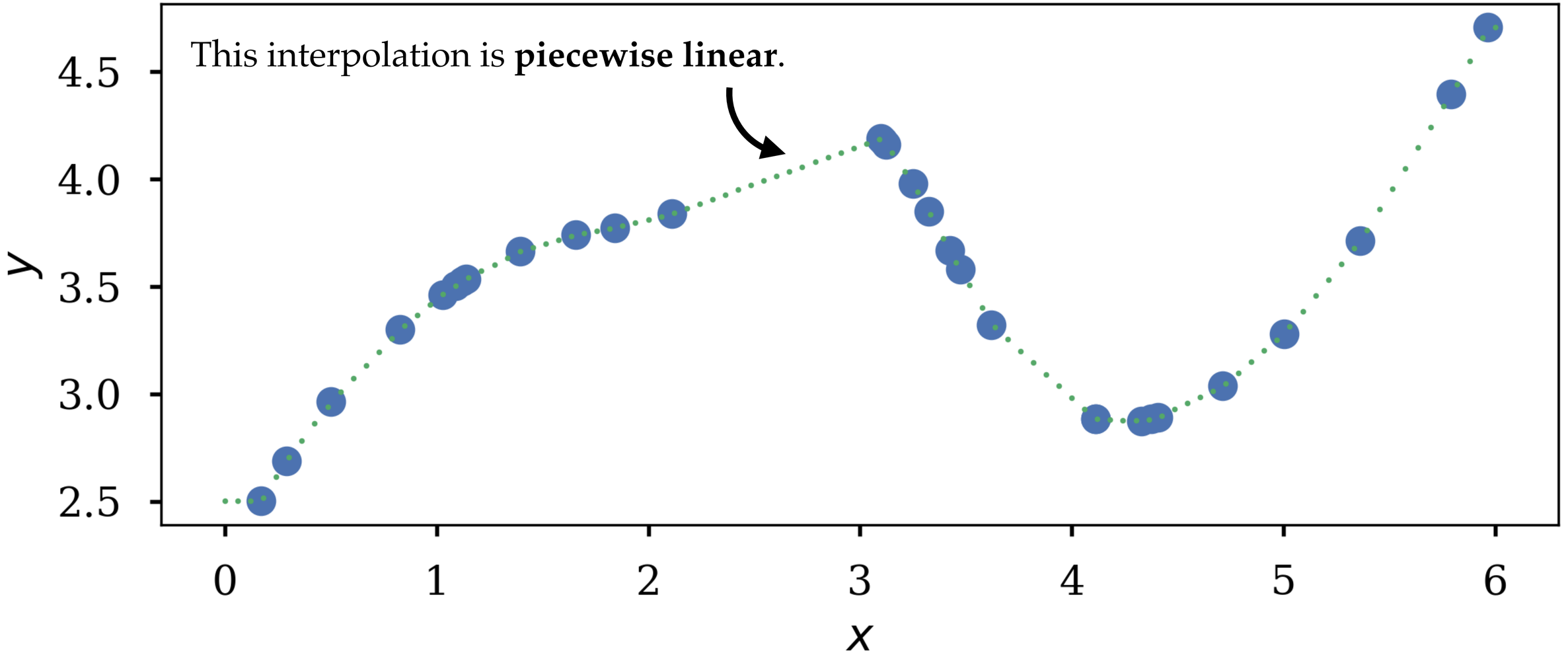


X-coordinates must be monotonic

We need to sort the input values into **monotonically increasing** order of x:

```
np.random.seed(101)
x = np.random.random(30)*6
y = 2+np.sin(x)-0.5*x+0.1*x**2+np.sqrt(x) + np.exp((-(x-3)**2)/0.4)

s = np.argsort(x)

xgrid = np.linspace(0,6,100)
y_interp = np.interp(xgrid,x[s],y[s])
```

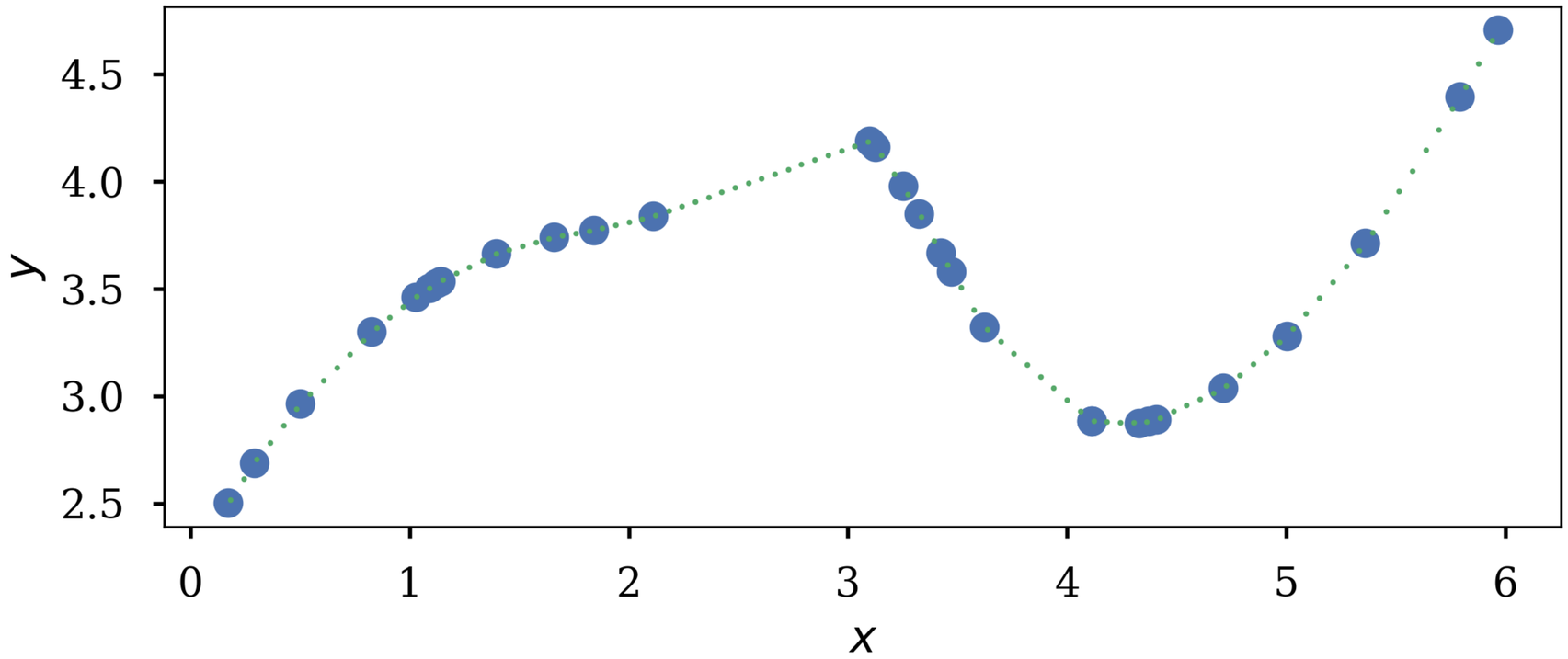
Interpolation in numpy



Interpolation in scipy

```
import scipy.interpolate as spinterp

# Returns an interpolator *function*
interpolator = spinterp.interp1d(x,y,kind='linear')
```

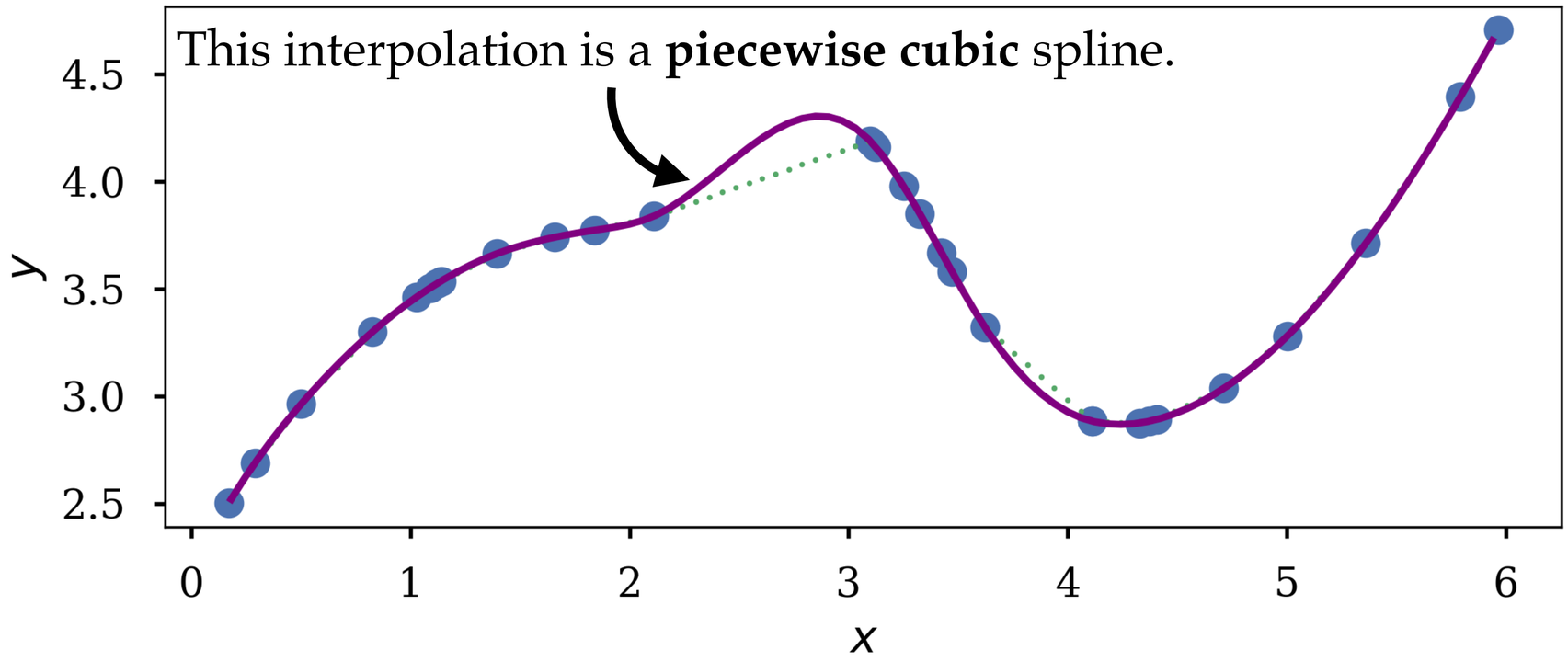
```
pl.scatter(x,y)
pl.scatter(xgrid,interpolator(xgrid),s=2)
```



Interpolation in scipy

```
cubic_interpolator = spinterp.interp1d(x,y,kind='cubic',bounds_error=False,fill_value=np.nan)
```

The interpolating functions have methods to report their coefficients, derivatives, roots etc.



Splines

Drafting spline with weights

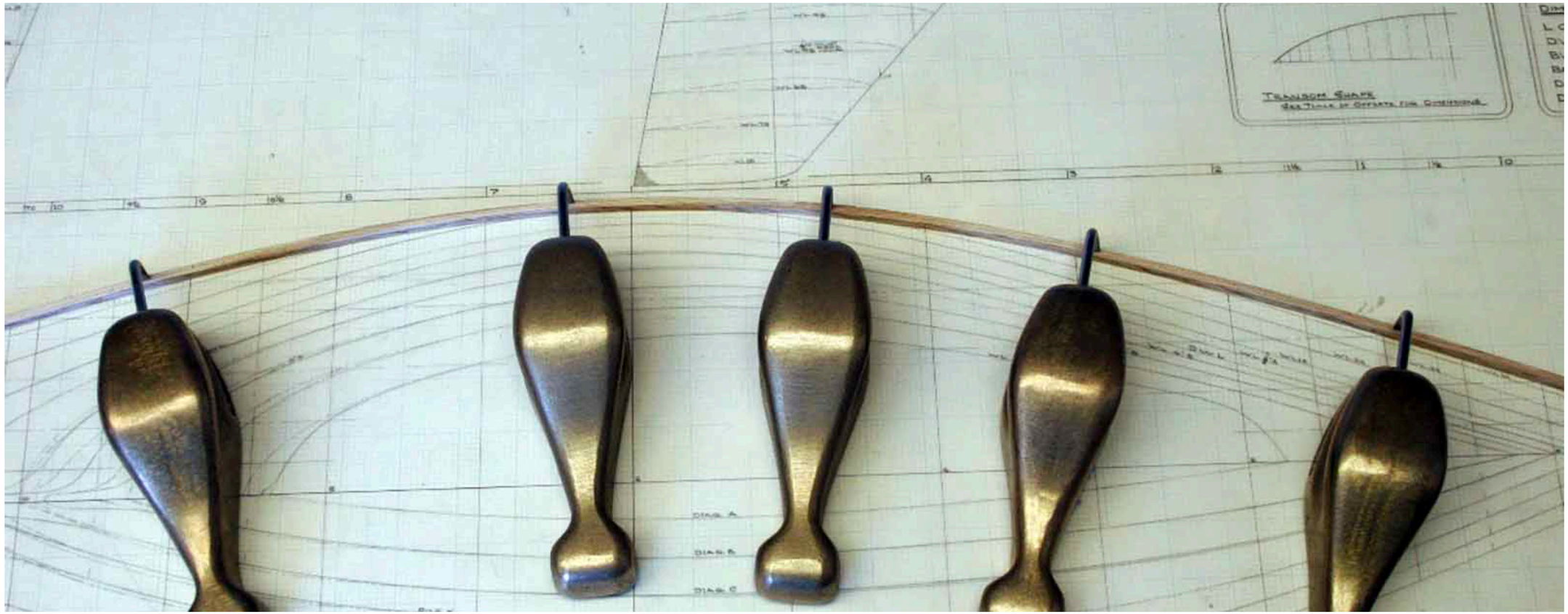


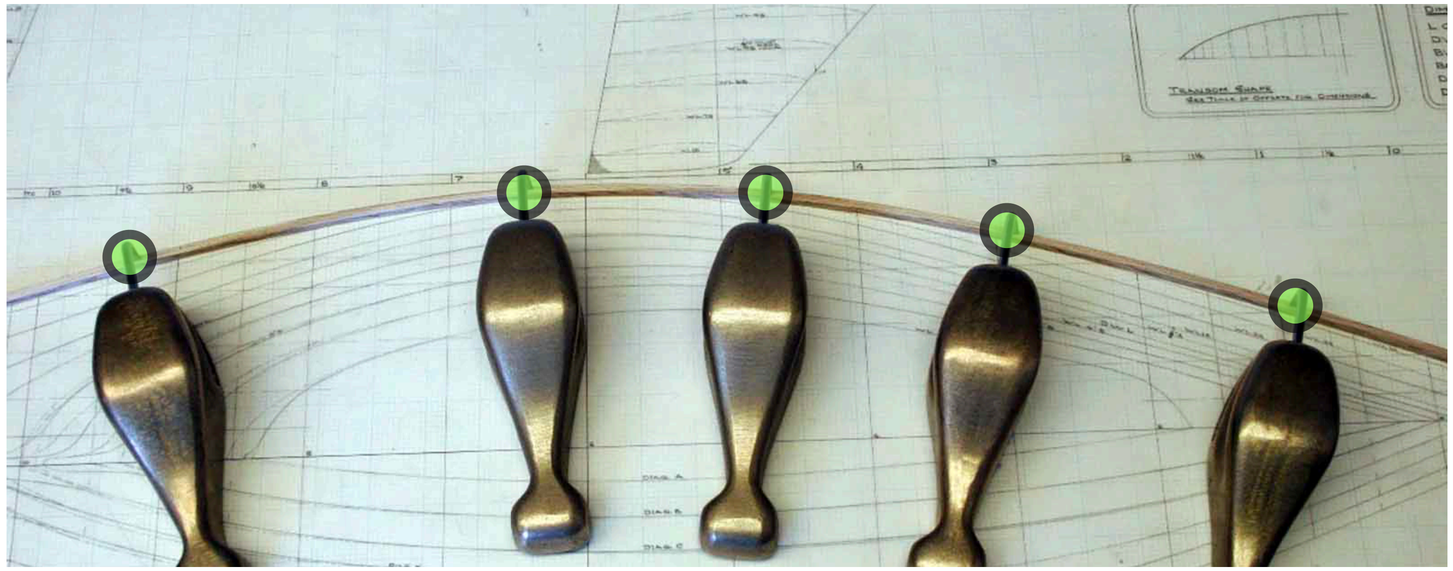
Image credit: unknown!

https://www.core77.com/posts/55368/When-Splines-Were-Physical-Objects

Splines

Think of a spline as a piecewise polynomial, with some restrictions: e.g. continuity of the first and second derivatives at the joining points (*knots*).

An interpolating (as opposed to fitting) spline goes through all the knots.



Interpolation vs. fitting

When interpolating, we are trying to find the function that "goes through" the points we have — we are just describing what we see.

When "fitting" data, we have the idea that our points are generated by some underlying **function**, convolved with **noise**.

In a fit, we want to account for uncertainty (due to the noise) in our judgement about the "best" estimate of the underlying function that generated the data.

With noise, we don't expect the "best" function to go through all the data points exactly; it should probably be smoother.

A typical approach to fitting is to minimise some "average distance" between the fit and the data (e.g. the root-mean-square residuals).

Interpolation vs. fitting

These slides follow Ch. 7 in Heath, Scientific Computing: An introductory Survey (2nd ed., 2005)

A fitting function typically has many fewer parameters than data points (i.e. more equations than unknowns).

An interpolating function has as many free parameters as there are data points, or more. If there system is overdetermined, the extra degrees of freedom can be used to impose other constraints (e.g. smoothness).

Interpolation with polynomials

Some polynomials:

$$P(x) = 1 + x^2 + 3x^4$$

$$Q(x) = 2x^2 + 5x^8$$

$$R(x) = P(x) + Q(x) = 1 + 3x^2 + 3x^4 + 5x^8$$

These polynomials are expressed as set of coefficients that multiply basis functions (in this case, *monomials* $x^0, x^1, x^2, x^3, \ldots, x^n$).

We can add and multiply polynomials; they comprise a vector space of dimension $k + 1$ where k is the highest order we include (so $k = 0$ has dimension 1; a constant).

Any function can be approximated to arbitrary accuracy by a polynomial (*Weierstrass theorem*)
This doesn't constrain the highest order you need to use.

Interpolation with polynomials

How do we find the polynomial that goes through some data points?

*Note: here we are talking about just **one** polynomial that goes through as many of the points as possible, not a piecewise function like a spline. We will come back to splines later.*

If we have n points and we want as many equations as unknowns, then we need to take functions from a space with dimension $k = n - 1$.

For example, if we have one point, all we know is its value (i.e. we need a in ax^0); if we have two points, we can put them both on the same line ($ax^0 + bx^1$), and so on.

Vandermonde matrix

This comes down to solving the linear problem $\mathbf{Ax} = \mathbf{y}$ where $\mathbf{x}$ (perhaps confusingly given previous notation) are the **coefficients** of the polynomial and $\mathbf{y}$ are the datapoints.

The Vandermonde matrix $\mathbf{A}$ contains all the basis functions that we use to assemble a polynomial that goes through one datapoint; in an exactly determined problem it has as many rows as there are datapoints.

$$\mathbf{A} = \begin{bmatrix} 1 & t_1 & \dots & t_1^{n-1} \\ 1 & t_2 & \dots & t_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & t_2 & \dots & t_n^{n-1} \end{bmatrix}$$

The (unique) polynomial we want is $p_{n-1}(t) = x_1 + x_2 t + x_3 t^2 + \dots x_n t^{n-1}$.

Interpolation with polynomials

For example, to fit a quadratic to three points, we solve a system of three equations for the three unknown coefficients:

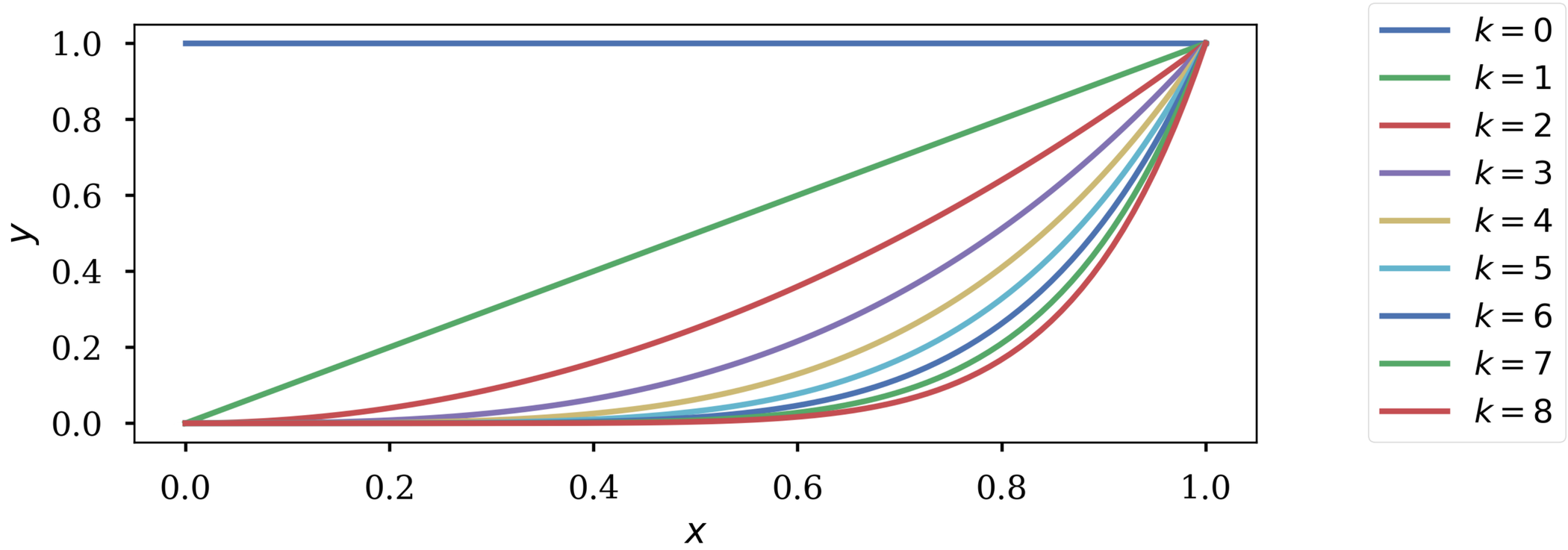
$$\mathbf{A}\mathbf{x} = \begin{bmatrix} 1 & t_1 & t_1^2 \\ 1 & t_2 & t_2^2 \\ 1 & t_3 & t_3^2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \mathbf{y}$$

Doing this with linear algebra using the monomial basis has various computational issues which you can read about elsewhere.

Interpolation with polynomials

The Vandermonde matrix is never singular in theory: there is always a unique solution.

However, if you add a lot of high orders (i.e. interpolate many points), it becomes *effectively* singular, because of finite numerical precision. You can get the same answer using different combinations of high-order terms; they all "look the same".



Interpolation with polynomials

Starting from the idea that you can represent any function as some combination of basis polynomials, the natural question is, which basis?

This is a very rich and historically important mathematical topic, not least as a means of solving differential equations.

The monomial basis is nice in many ways: it's straightforward to do algebra, find integrals and derivatives and so on. It's usually fast enough to compute with.

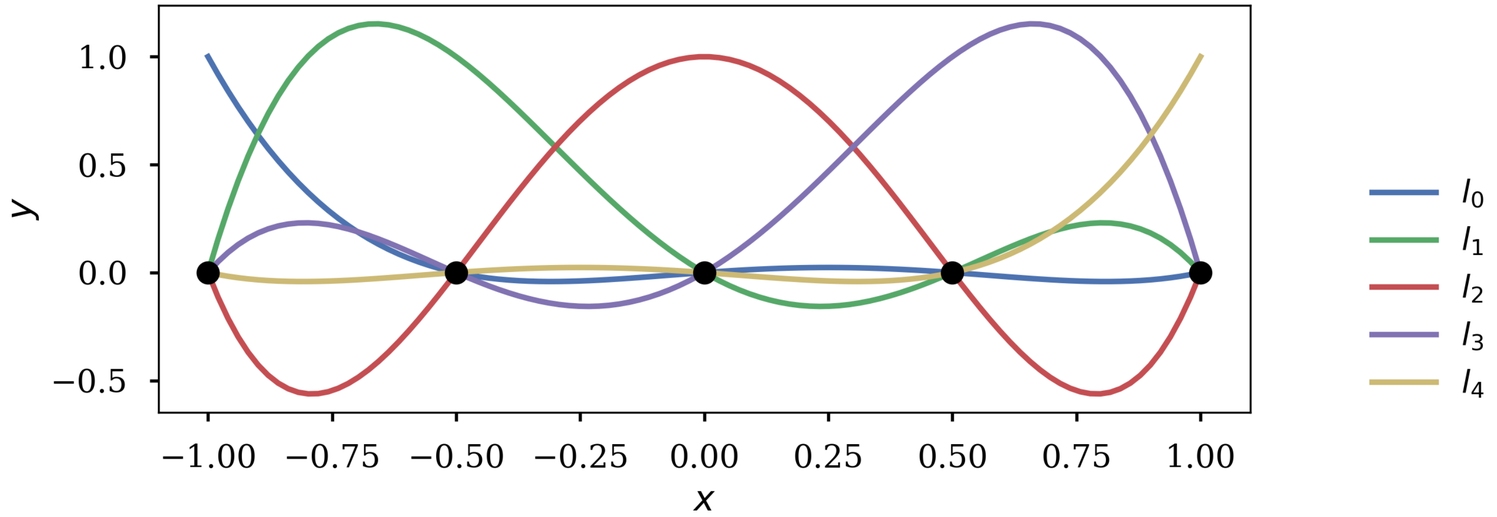
However, there are advantages to other choice of basis, including avoiding the problem raised on the previous slides.

Other polynomial basis functions

One alternative choice is the Lagrange polynomials (you can read about those elsewhere); another is the Legendre polynomials. There are more!

A good choice should be easy and efficient to work with and not prone to problems in the linear algebra involved in fitting/interpolation.

Lagrange

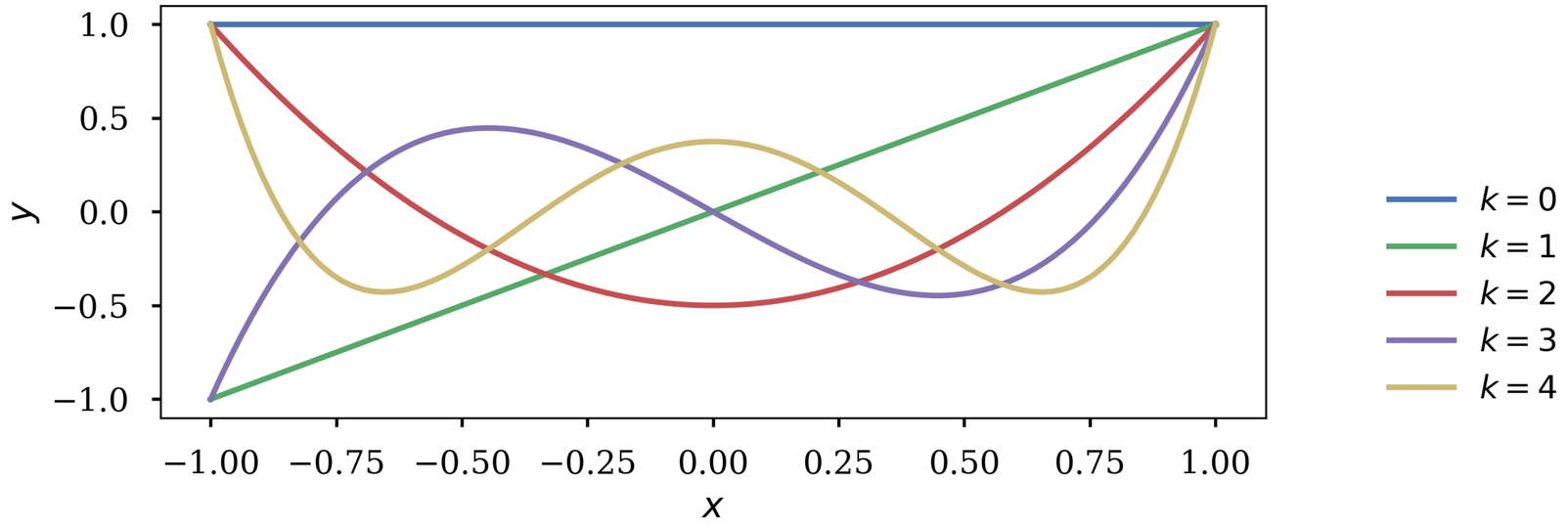



```
def lagrange_basis(t, X, j):
    """
    https://stackoverflow.com/questions/4003794/lagrange-interpolation-in-python
    """
    n = len(X)
    b = [(t - X[k]) / (X[j] - X[k]) for m in range(n) if k != j]
    return np.prod(b, axis=0)

x = np.linspace(-1, 1, 5)
xx = np.linspace(-1,1,100)

pl.plot(xx,lagrange_basis(xx,x,0),label='$l_{0}$')
pl.plot(xx,lagrange_basis(xx,x,1),label='$l_{1}$')
pl.plot(xx,lagrange_basis(xx,x,2),label='$l_{2}$')
pl.plot(xx,lagrange_basis(xx,x,3),label='$l_{3}$')
pl.plot(xx,lagrange_basis(xx,x,4),label='$l_{4}$')
```

$$l_j(t) = \frac{\prod_{k=1,k\neq j}^{n}(t - t_k)}{\prod_{k=1,k\neq j}^{n}(t_j - t_k)},\ j = 1 \ldots n$$

Legendre polynomials

```
from numpy.polynomial import Legendre

C1 = Legendre((1))
C2 = Legendre((0,1))
C3 = Legendre((0,0,1))
C4 = Legendre((0,0,0,1))
C5 = Legendre((0,0,0,0,1))
C5 = Legendre((0,0,0,0,0,1))

pl.figure(figsize=(8,3))
x = np.linspace(-1, 1, 100)

for i in range(0,5):
    coeffs = np.zeros(i+1,dtype=np.int32)
    coeffs[i] = 1
    pl.plot(x,Legendre(coeffs)(x))
```

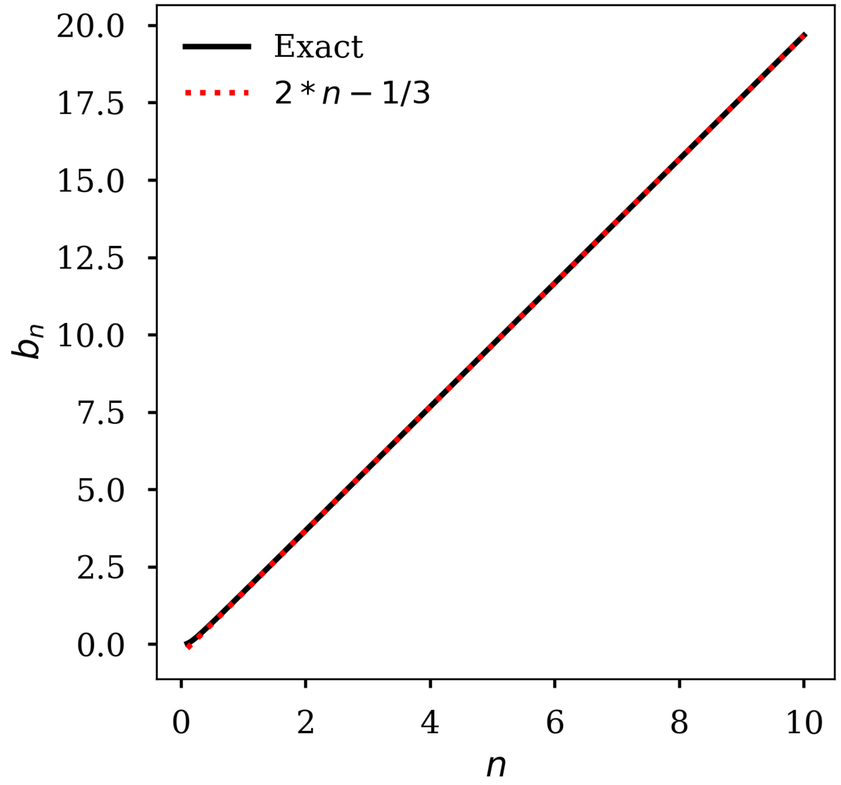


Polynomial approximation example

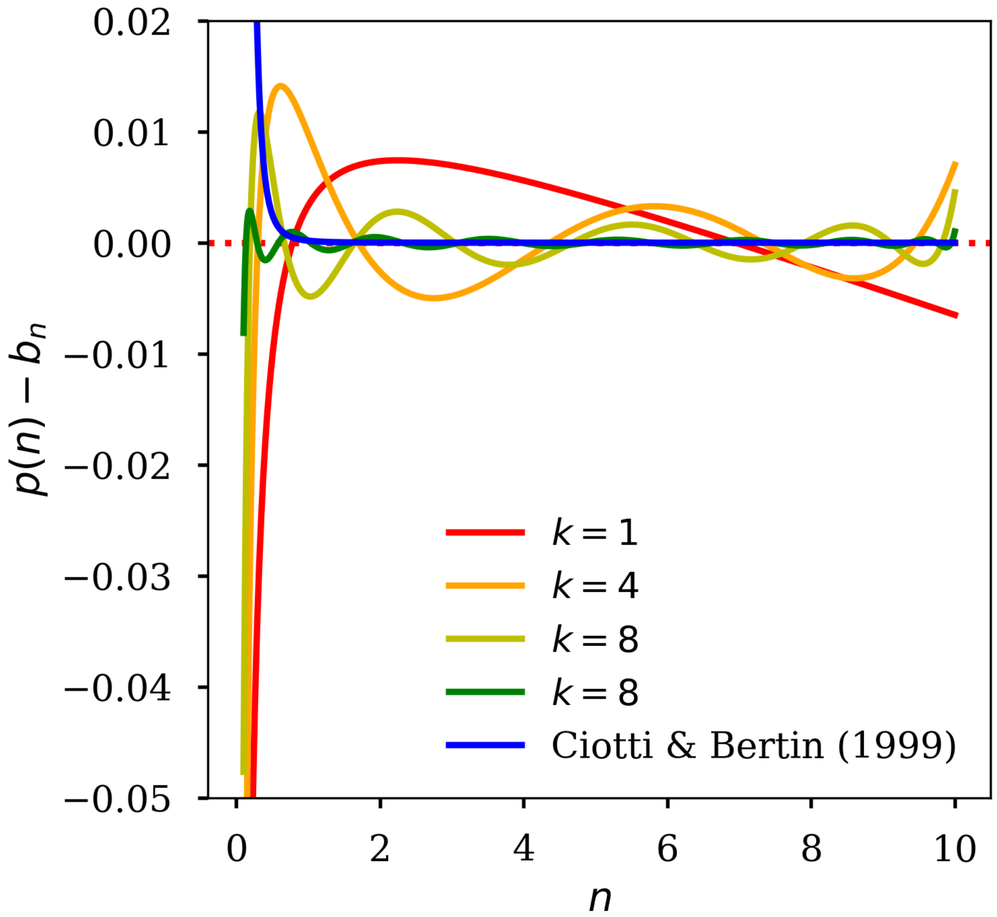
Sersic b_n

```
from scipy.special import gammaincinv
n = np.linspace(0.1,10,10000)
b = gammaincinv(2*n, 0.5)
```

C&B 99: $b_n \approx 2n - \frac{1}{3} + An^{-1} + Bn^{-2} + Cn^{-3} + Dn^{-4}$

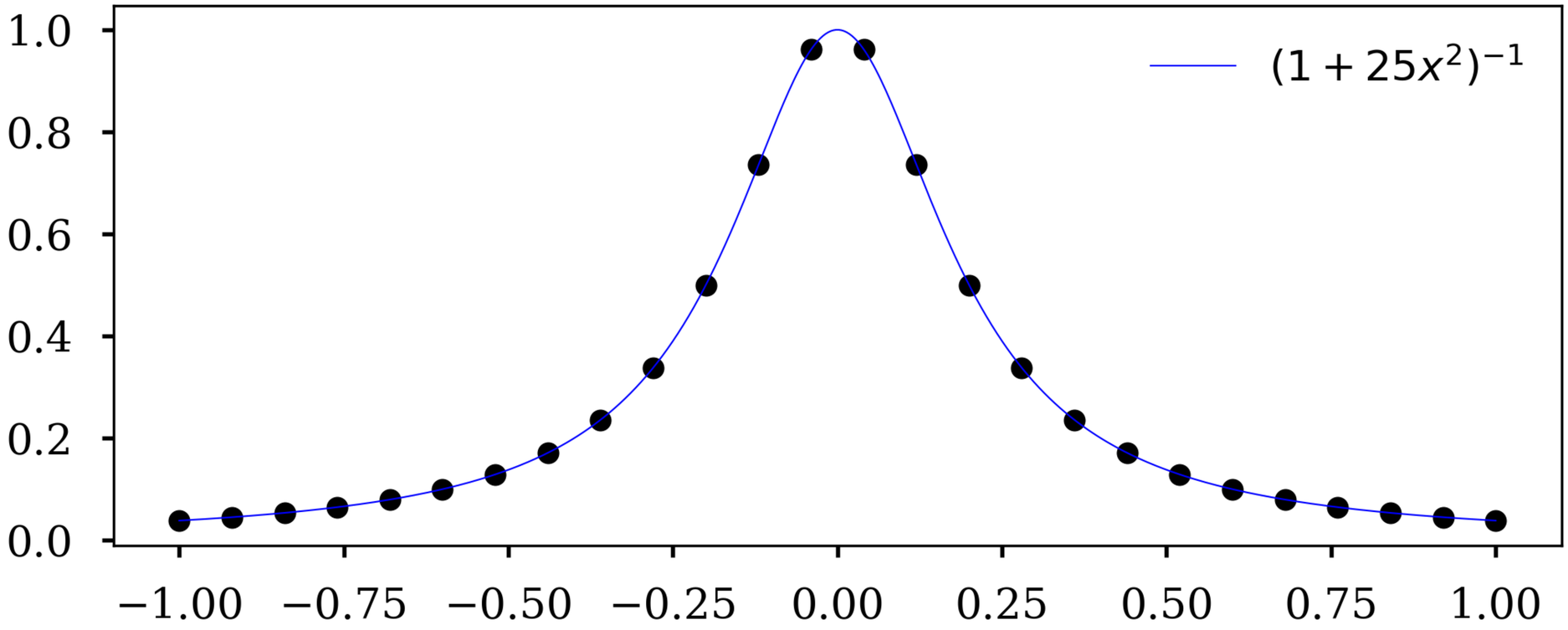



```
poly_b = np.polynomial.Polynomial.fit(n,b,4)
pl.plot(n,(poly_b(n)-b),c='orange',label='$k=4$')
```



Runge's function

$$f_{\text{Runge}}(x) = \frac{1}{1+25x^2}, \quad \text{defined on } [-1,1]$$



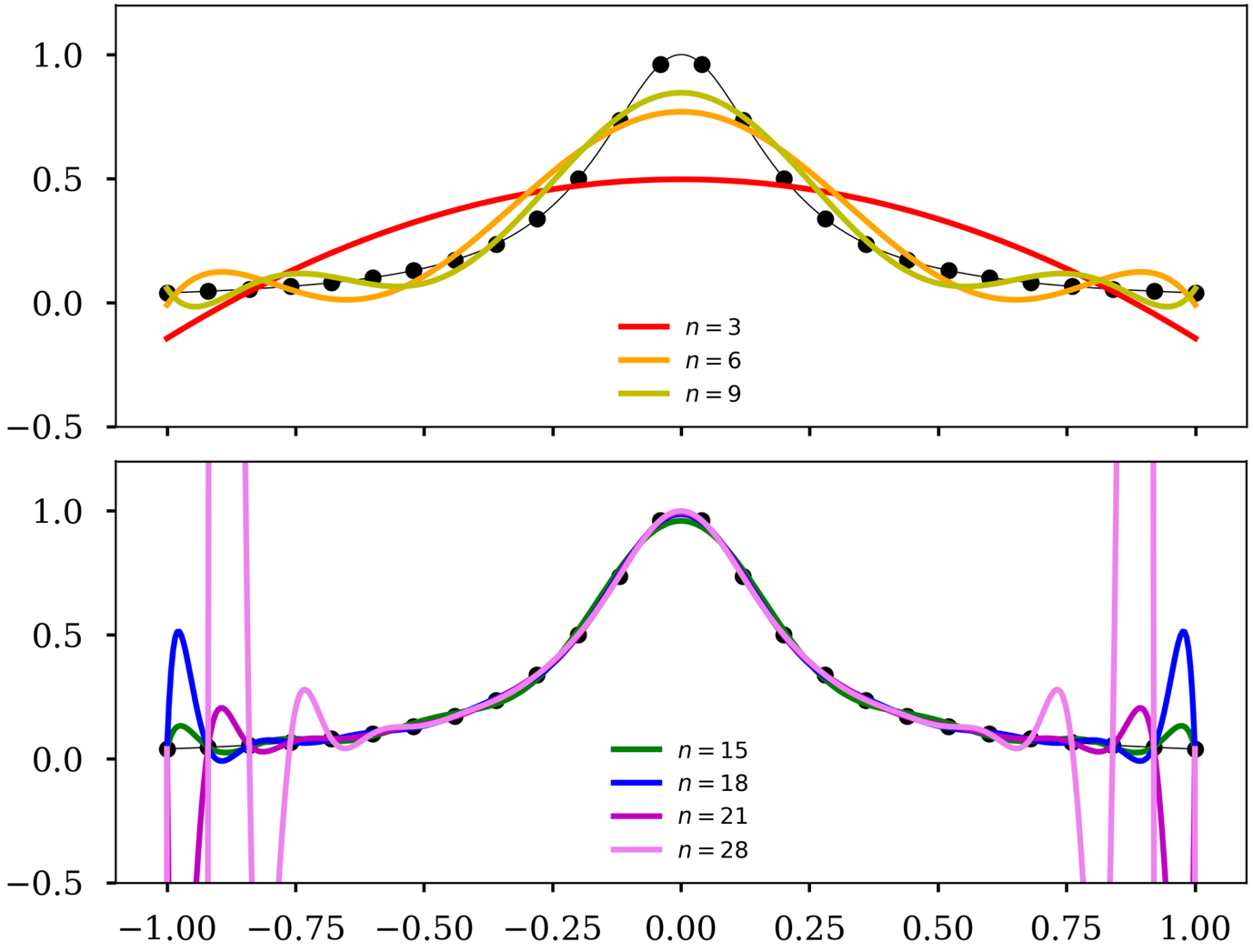
Runge's phenomenon

```
P = np.polynomial.Polynomial.fit(x,y,3)
plot(xx, P(xx))
```

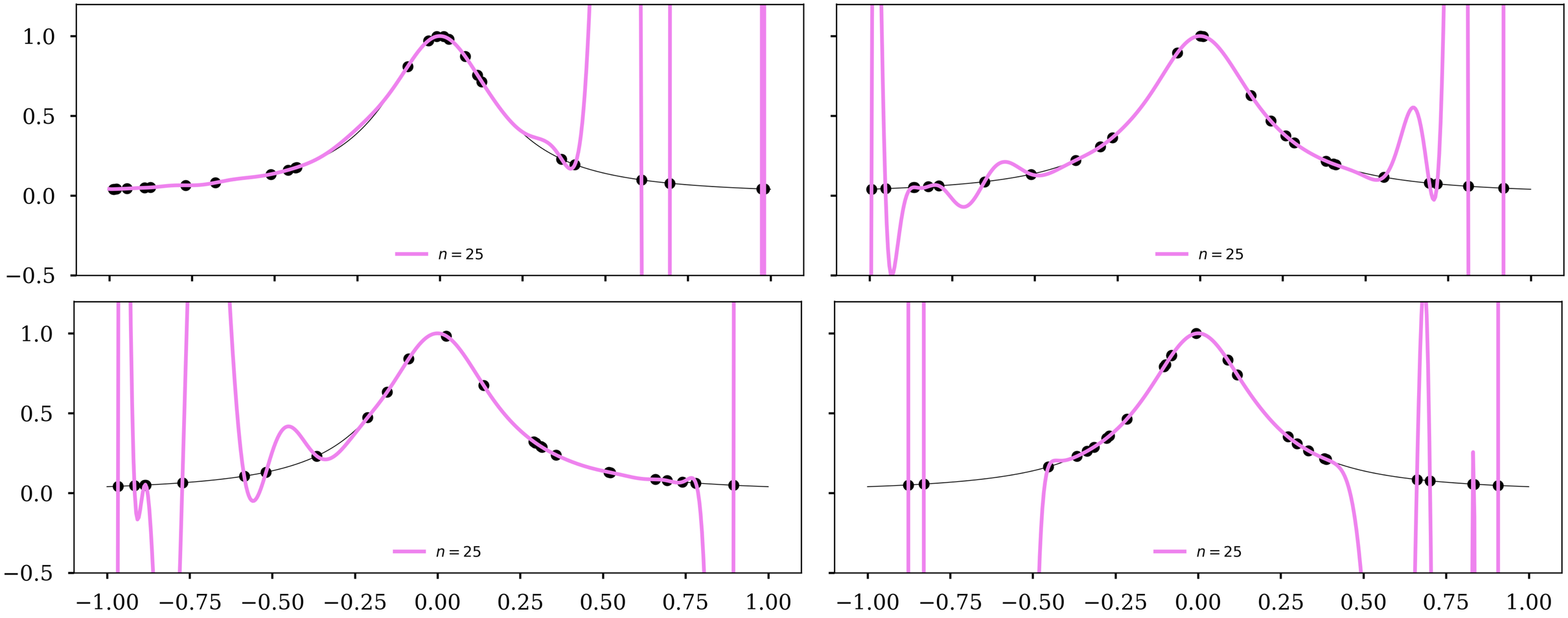
This is doing least-squares fitting; there are 26 points, so this becomes an interpolation at order $k = 25$.

Higher orders k give better approximations at $x = 0$, but "**ring**" at the edges of the interval. More orders, stronger oscillations.

Polynomial fits always "wiggle" more than the data, but these wiggles are catastrophic!



Runge’s phenomenon

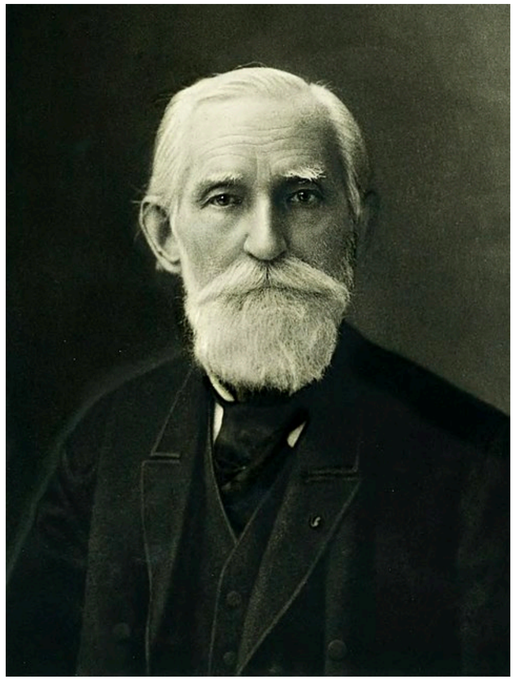


Chebyshev points

Pafnuty Lvovich Chebyshev (Russian: Пафну́тий Льво́вич Чебышёв, IPA: [pɐfˈnutʲɪj ˈlʲvovʲɪtɕ tɕɪbɨˈʂof]) (16 May [O.S. 4 May] 1821 – 8 December [O.S. 26 November] 1894)[2] was a Russian mathematician and considered to be the founding father of Russian mathematics.

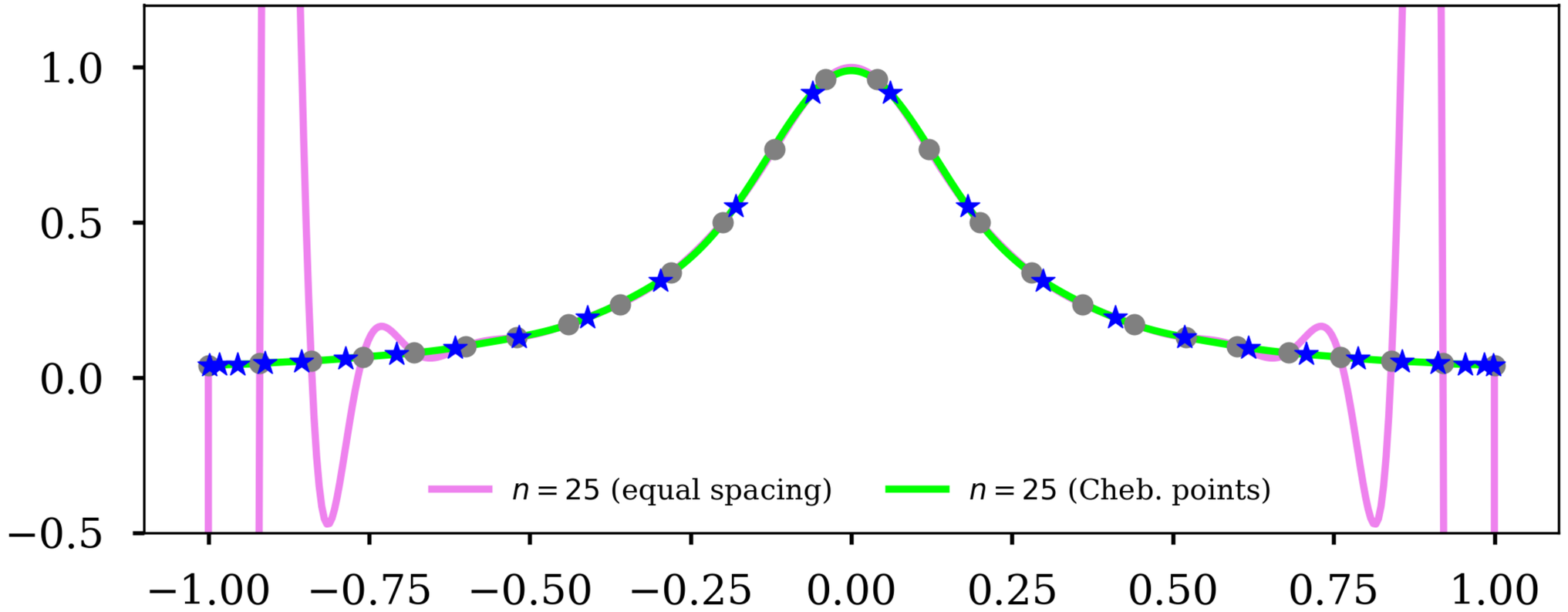
Transcription [edit]

The surname Chebyshev has been transliterated in several different ways, like Tchebichef, Tchebychev, Tchebycheff, Tschebyschev, Tschebyschef, Tschebyscheff, Čebyčev, Čebyšev, Chebysheff, Chebychov, Chebyshov (according to native Russian speakers, this one provides the closest pronunciation in English to the correct pronunciation in old Russian), and Chebychev, a mixture between English and French transliterations considered erroneous. It is one of the most well known data-retrieval nightmares in mathematical literature. Currently, the English transliteration *Chebyshev* has gained widespread acceptance, except by the French, who prefer *Tchebychev.* The correct transliteration according to ISO 9 is *Čebyšëv*. The American Mathematical Society adopted the transcription *Chebyshev* in its Mathematical Reviews.[3]



Chebyshev points

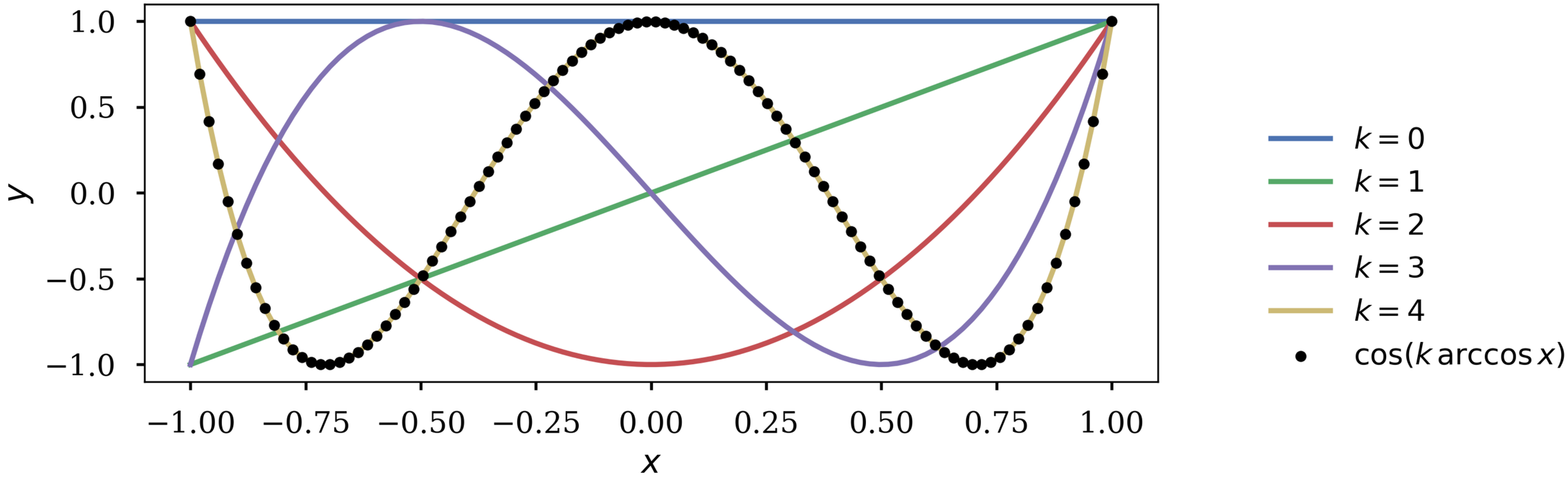
`polynomial.chebyshev.chebpts1(npts)`



Chebyshev polynomials

$$T_k(t) = \cos(k \arccos t), k = 0,1,2,3,...$$

$$T_0(t) = 1,\ T_1(t) = t,\ T_2(t) = 2t^2 - 1,\ T_3(t) = 4t^3 - 3t,\ T_4(t) = 8t^4 - 8t^2 + 1,\ \ldots$$



Back to splines

In practice, splines are a very useful brute-force alternative to the challenge of finding a single unique polynomial going through all the points. They are mathematically more tedious, but we have computers do the hard work.

Every spline is a low-order polynomial constrained at two points. For example, linear spline interpolation joins every pair of points with a line.

Since there are lots of cubics (for example) that go through the same two points, we have to make some choices about we want the spline to look like (for example, how smooth it should be / what the derivatives should be like).

Let's look at the scipy routines for spline interpolation and fitting.

Spline routines

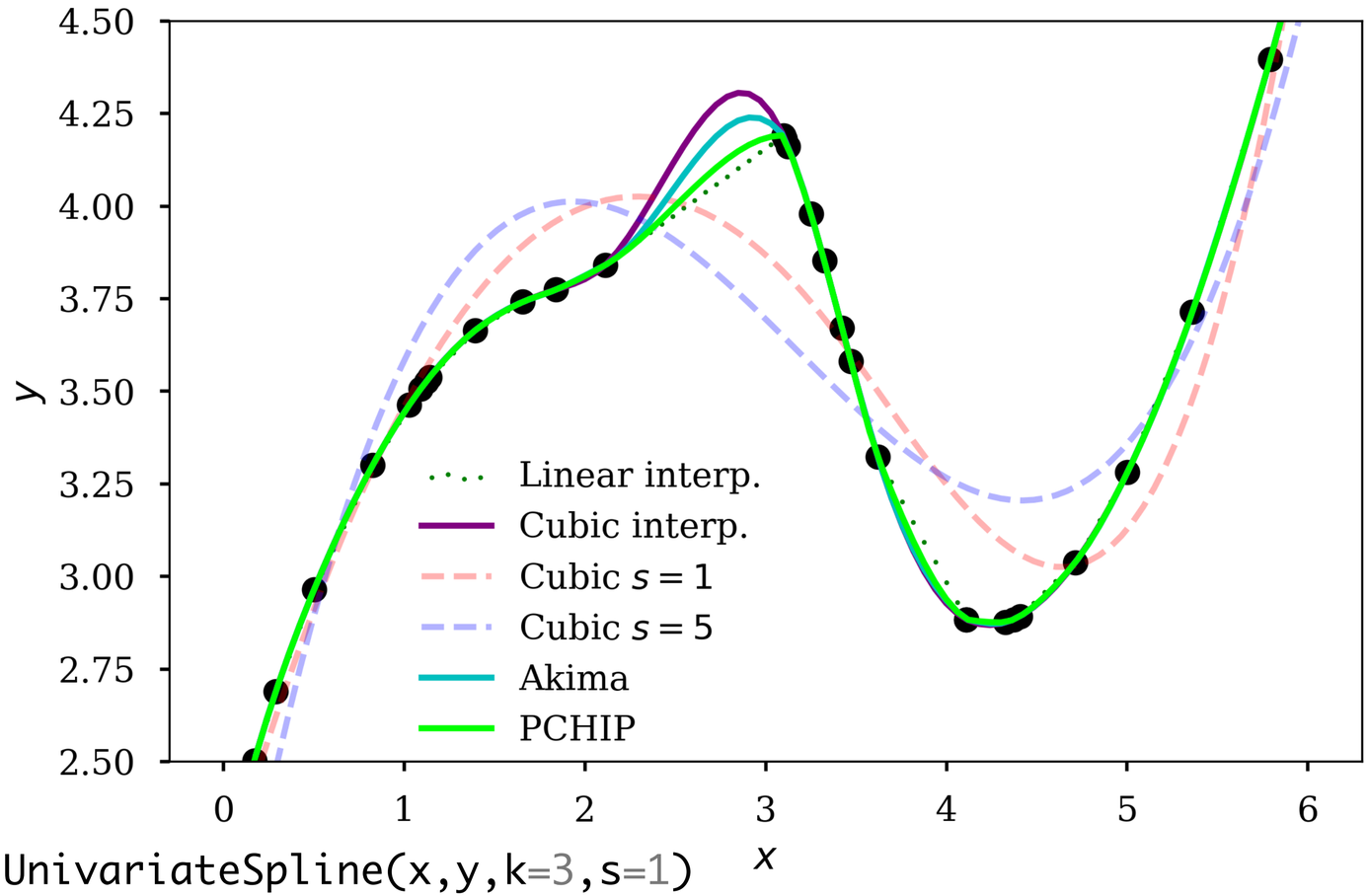
```
import scipy.interpolate
help(scipy.interpolate)
```

```
Univariate interpolation
========================

   interp1d
   BarycentricInterpolator
   KroghInterpolator
   barycentric_interpolate
   krogh_interpolate
   pchip_interpolate
   CubicHermiteSpline
   PchipInterpolator
   Akima1DInterpolator
   CubicSpline
   PPoly
   BPoly
```

```
1-D Splines
===========

   BSpline
   make_interp_spline
   make_lsq_spline
   make_smoothing_spline
```

Interpolation in scipy

```
cubic_interpolator = spinterp.interp1d(x,y,kind='cubic',bounds_error=False,fill_value=np.nan)
```




```
uvspl1 = spinterp.UnivariateSpline(x,y,k=3,s=1)
akima  = spinterp.Akima1DInterpolator(x,y)
pchip  = spinterp.PchipInterpolator(x,y)
```

Interpolation summary

Whenever possible, make a plot.

In the first instance use piecewise linear interpolation, then cubic splines. If it really matters, be aware of the implicit choices in these methods.

Consider the difference between fitting and smoothing. For example, when using look-up tables, you usually want to interpolate. When “interpolating” real data, you usually want to fit.

Polynomial approximations are nice if they work, but watch out for ringing.

Be extremely cautious about extrapolation.

R. Foster
@TaiwanBirding

Black-naped Monarch
黑枕藍鶲

ASTR 660 / Class 9

Random Numbers

Random numbers

We haven't talked about random numbers yet!

```
rng = np.random.default_rng(seed=101)
rng.random(5)
```

```
array([0.94353251, 0.35942103, 0.78480541, 0.59127819, 0.29432856])
```

```
rng = np.random.default_rng(seed=101)
rng.random((3,3))
```

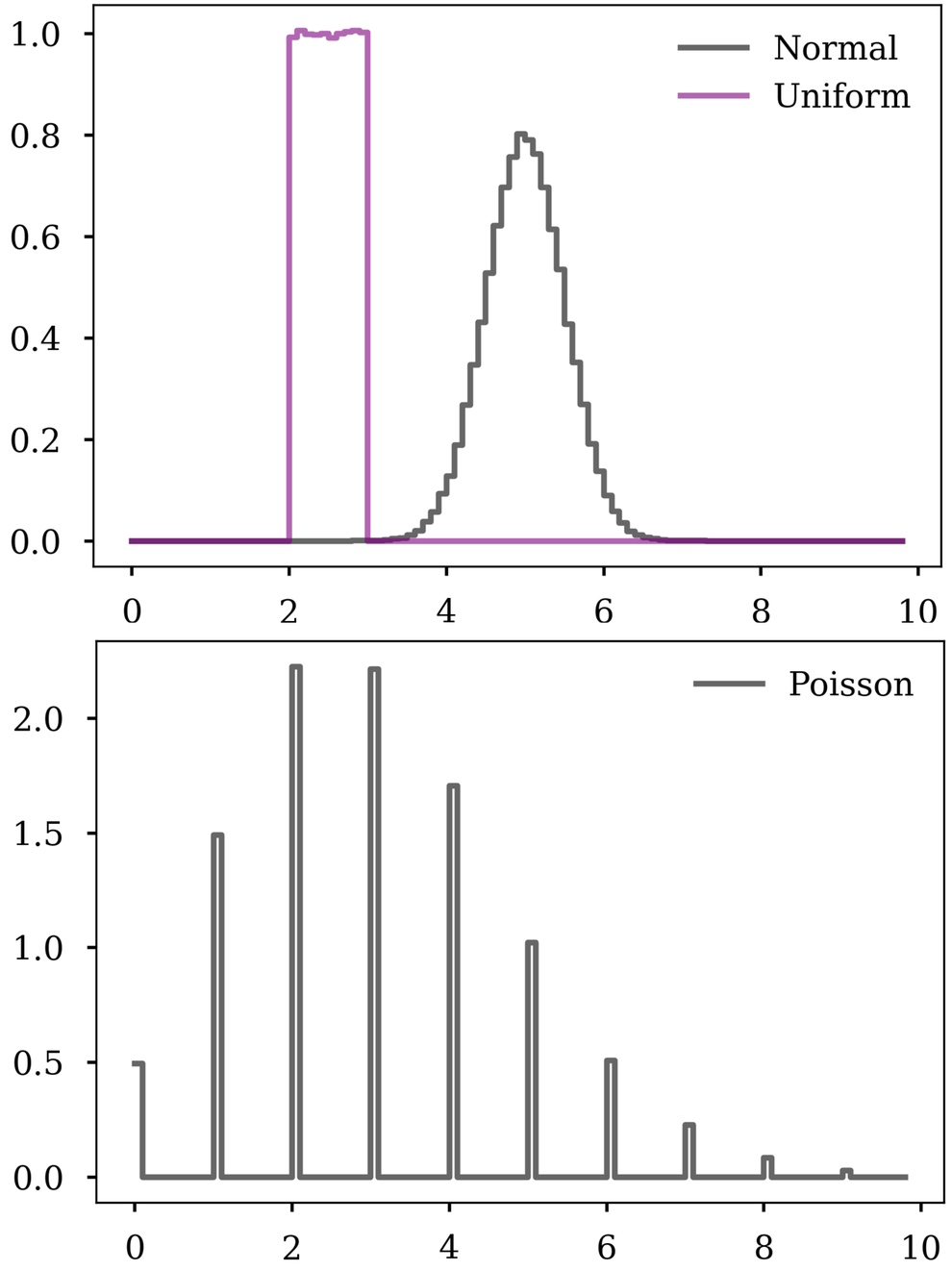
```
array([[0.94353251, 0.35942103, 0.78480541],
       [0.59127819, 0.29432856, 0.92272569],
       [0.86933154, 0.36413843, 0.97317681]])
```

```
rng = np.random.default_rng(seed=101)
rng.random(5)
rng.random((3,3))
```

```
array([[0.92272569, 0.86933154, 0.36413843],
       [0.97317681, 0.22452433, 0.80549587],
       [0.68089623, 0.47106052, 0.03080547]])
```

Random numbers

```
rng = np.random.default_rng(seed=101)

n = rng.normal(loc=5,scale=0.5,size=100_000)
h,b = np.histogram(n,bins=np.arange(0,10,0.1),density=True)
pl.plot(b[:-1],h,drawstyle='steps-post',c='k')

rng = np.random.default_rng(seed=101)

n = rng.uniform(2,3,size=100_000)
h,b = np.histogram(n,bins=np.arange(0,10,0.1),density=True)
pl.plot(b[:-1],h,drawstyle=Ôsteps-post',c='purple')

rng = np.random.default_rng(seed=101)

n = rng.poisson(3,size=100_000)
h,b = np.histogram(n,bins=np.arange(0,10,0.1),density=True)
pl.plot(b[:-1],h,drawstyle='steps-post',c='k');
```



Random numbers in Fortran

```
program random
  implicit none
  real :: r(10)

  ! Return a vector of 10 random numbers
  call random_number(r)
end program random
```

In Fortran, if you want a random uniform or normal variate etc., you make it yourself.

Random seeds in Fortran

```
program random
  implicit none

  integer :: nseed
  integer, allocatable :: seed(:)
  real :: r

  ! Set up the seed vector
  call random_seed(size=nseed)
  allocate(seed(nseed))

  call random_seed(get=seed)
  seed(:) = 2020
  call random_seed(put=seed)

  call random_number(r)

  ! The seed will have been updated now
  call random_seed(get=seed)
end program random
```

In Fortran the state of the RNG is an array.

Get the size of the state array.

Fill the array.

Check the state.

R. Foster
@TaiwanBirding

Swinhoe's White-eye
斯氏繡眼

ASTR 660 / Class 9

Root-finding

Root-finding

A common practical task is to find the points at which $f(x) = 0$ (or any other value, redefining $f'(x) = f(x) - a = 0$): i.e. the **roots** of $f(x)$.

This is obviously the same problem as the point(s) at which two functions are equal to each other, $f(x) - g(x) = 0$; in other words, it is a means of **solving nonlinear equations**.

This is not the same as the problem of finding the minimum of a function, although it is related. Both are types of **optimisation** problem: they involve a *function*, and some *constraints* that have to be satisfied.

Fitting data with a model is also an optimisation problem (e.g. minimizing a function that computes the sum of the square residuals between the data and the model).

Root-finding toolbox

```
import scipy.optimize as spo
help(spo)
```

```
Root finding
============

Scalar functions
----------------
.. autosummary::
   :toctree: generated/

   root_scalar - Unified interface for nonlinear solvers of scalar functions.
   brentq - quadratic interpolation Brent method.
   brenth - Brent method, modified by Harris with hyperbolic extrapolation.
   ridder - Ridder's method.
   bisect - Bisection method.
   newton - Newton's method (also Secant and Halley's methods).
   toms748 - Alefeld, Potra & Shi Algorithm 748.
   RootResults - The root finding result returned by some root finders.
```

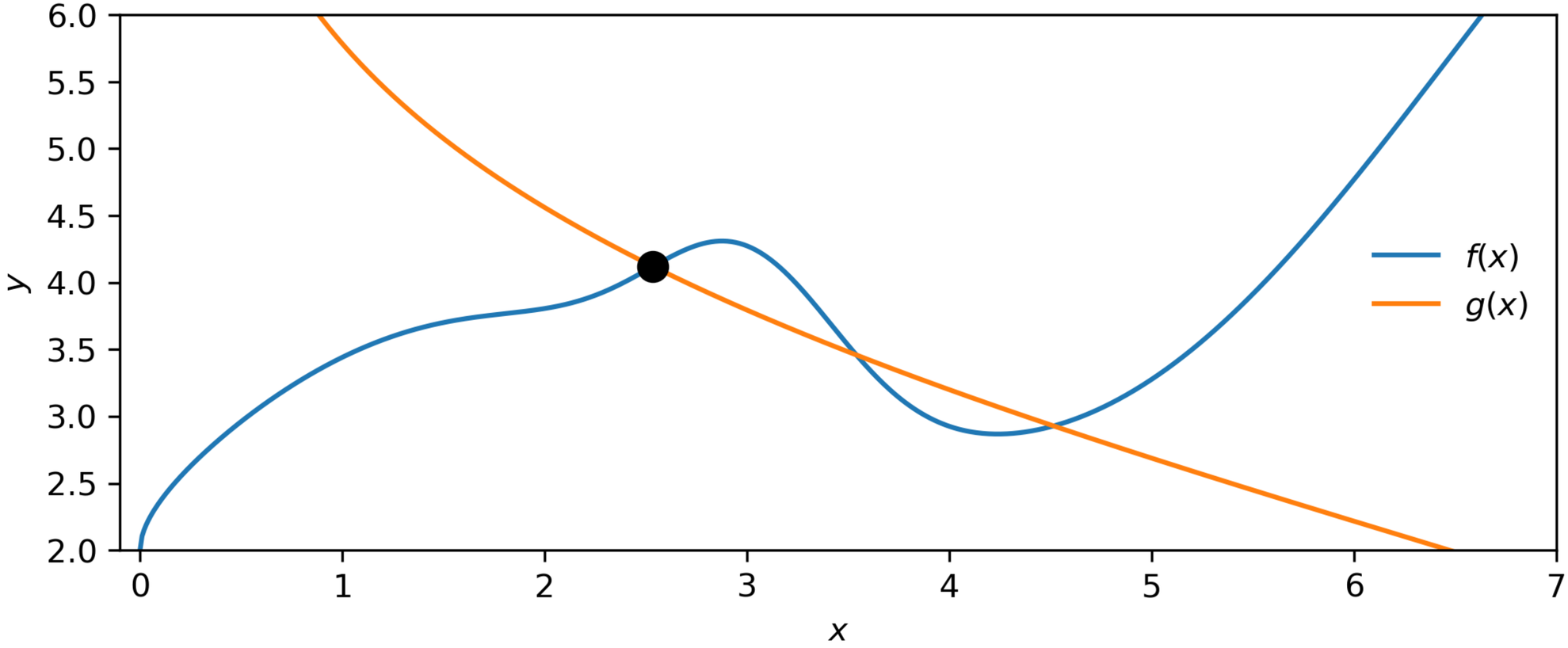
Root-finding toolbox

```
import scipy.optimize as spo
help(spo)
```

```
The table below lists situations and appropriate methods, along with *asymptotic* convergence rates per I iteration (and per
function evaluation) for successful convergence to a simple root(*). Bisection is the slowest of them all, adding one bit of
accuracy for each function evaluation, but is guaranteed to converge. The other bracketing methods all (eventually) increase the
number of accurate bits by about 50% for every function evaluation. The derivative-based methods, all built on `newton`, can
converge quite quickly if the initial value is close to the root.

+-------------+----------+----------+-----------+-------------+-------------+----------------+
| Domain of f | Bracket? |    Derivatives?      | Solvers     |        Convergence           |
+             +          +----------+-----------+             +-------------+----------------+
|             |          | `fprime` | `fprime2` |             | Guaranteed? |  Rate(s)(*)    |
+=============+==========+==========+===========+=============+=============+================+
| `R`         | Yes      | N/A      | N/A       | - bisection | - Yes       | - 1 "Linear"   |
|             |          |          |           | - brentq    | - Yes       | - >=1, <= 1.62 |
|             |          |          |           | - brenth    | - Yes       | - >=1, <= 1.62 |
|             |          |          |           | - ridder    | - Yes       | - 2.0 (1.41)   |
|             |          |          |           | - toms748   | - Yes       | - 2.7 (1.65)   |
+-------------+----------+----------+-----------+-------------+-------------+----------------+
| `R` or `C`  | No       | No       | No        | secant      | No          | 1.62 (1.62)    |
+-------------+----------+----------+-----------+-------------+-------------+----------------+
| `R` or `C`  | No       | Yes      | No        | newton      | No          | 2.00 (1.41)    |
+-------------+----------+----------+-----------+-------------+-------------+----------------+
| `R` or `C`  | No       | Yes      | Yes       | halley      | No          | 3.00 (1.44)    |
+-------------+----------+----------+-----------+-------------+-------------+----------------+
```

Brent

```
import scipy.optimize as spo

f = lambda x: 2+np.sin(x)-0.5*x+0.1*x**2+np.sqrt(x) + np.exp((-(x-3)**2)/0.4)
g = lambda x: 4*np.arccos(np.log10(x)) - 0.5

h  = lambda x: f(x) - g(x)
r1 = spo.brentq(h, 2, 6)
```

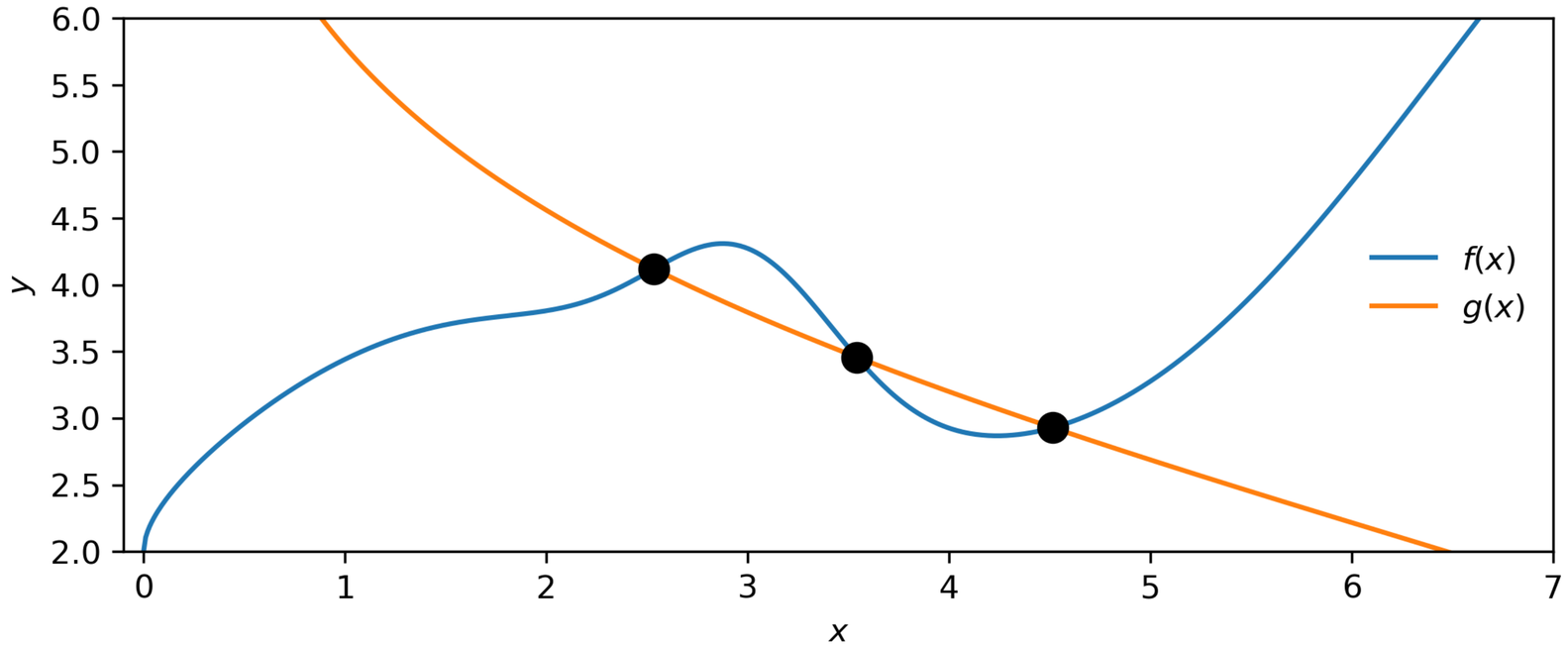


Key point

For one-off problems, just use Brent.

For problems where you need near-maximum speed, and you have a good understanding of the function, consider the Newton or Secant methods (or the even faster alternatives).

Challenge



As far as I know, there is no truly general method for finding all the roots of a function. However, there are some reasonable hacks.

Using only functions/modules from today's slides, write a routine that tries to returns a list of all the roots of an arbitrary function on a given interval of x. For example:

```
def find_all_roots(x, f, method='brentq', eps=1e-10):
```