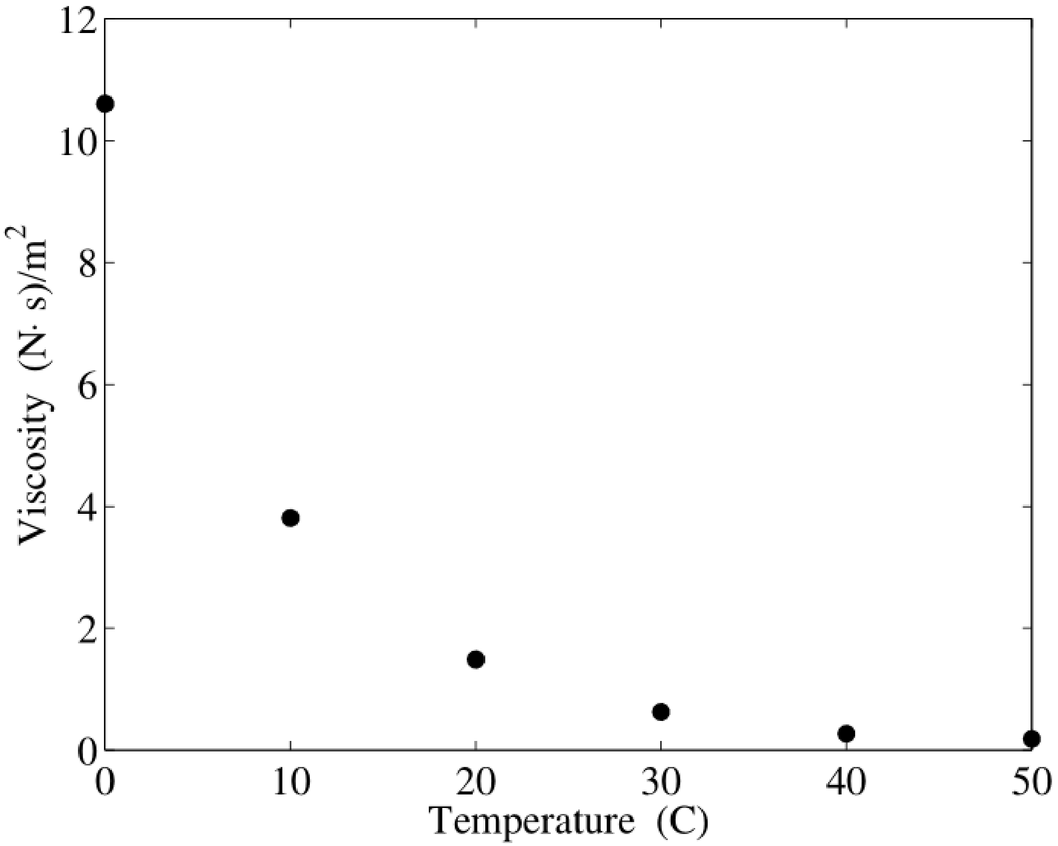
PHYS 351
Interpolation and Fitting

Dr. Daugherity
Abilene Christian University

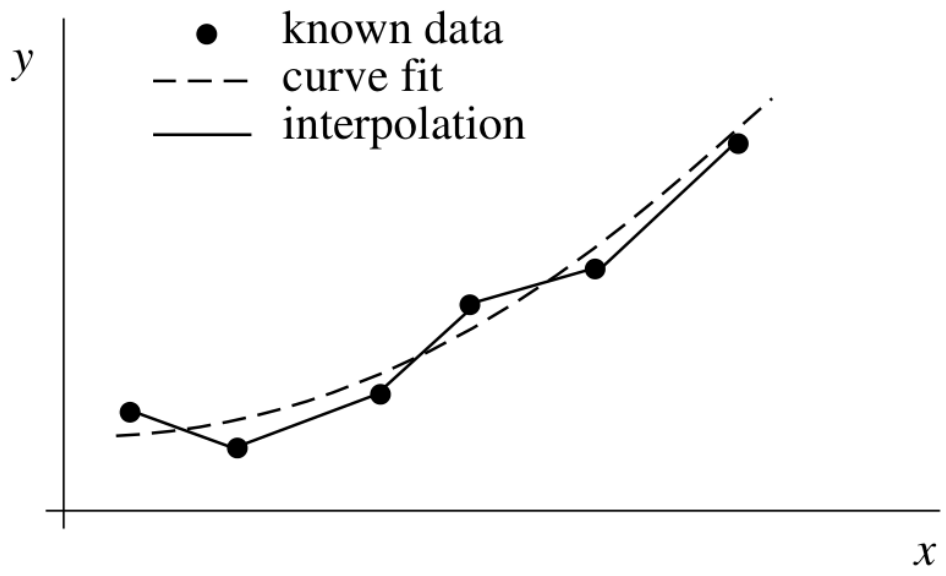
Problem



Make a good guess about what happens in between the data points.

What is V at T=25 C?

Approaches



Interpolation: connect the dots with piecewise equation

Fitting: find a single function which approximates the data

IMPORTANT – Interpolation goes through every point, while fitting will “average between” them.

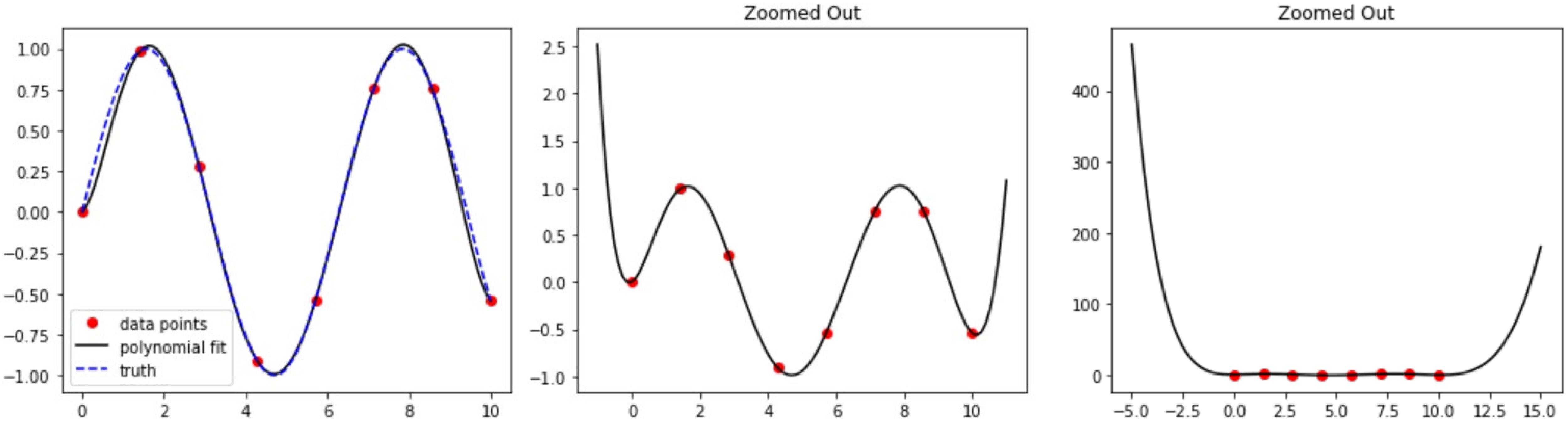
Approaches

	GOOD	BAD
Interpolation	• fast and easy (connect-the-dots) • standard approaches that nearly always work	• does not reduce noise • does not reduce size of data
Fitting	• reduces noise • data reduction • easy to evaluate **after** the fit	• we usually don’t know what function to use • fitting parameters can be extremely hard

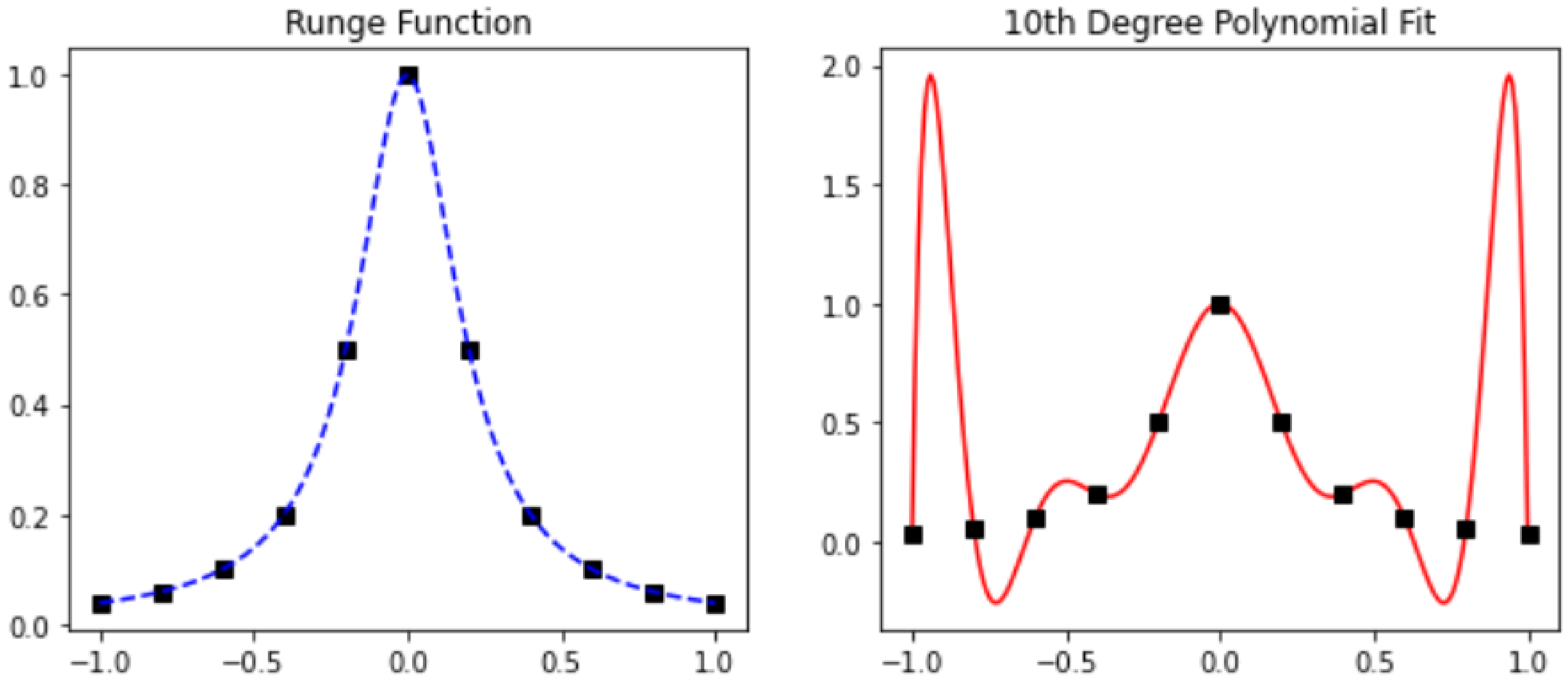
What Doesn't Work

THEOREM: For any set of n points, there is one unique polynomial of degree n-1 that goes through them.

So why not just do that?



EXTRAPOLATION – what happens beyond the data points?



Runge Function
1.0
0.8
0.6
0.4
0.2
0.0
−1.0
−0.5
0.0
0.5
1.0
10th Degree Polynomial Fit
2.0
1.5
1.0
0.5
0.0
−1.0
−0.5
0.0
0.5
1.0

Fitting

Least Squares

https://phet.colorado.edu/sims/html/curve-fitting/latest/curve-fitting_en.html

- mathematical definition of “best” fit
- a **solved** problem for any polynomial

numpy.polyfit

`numpy.polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)`

Least squares polynomial fit.

numpy.polyval

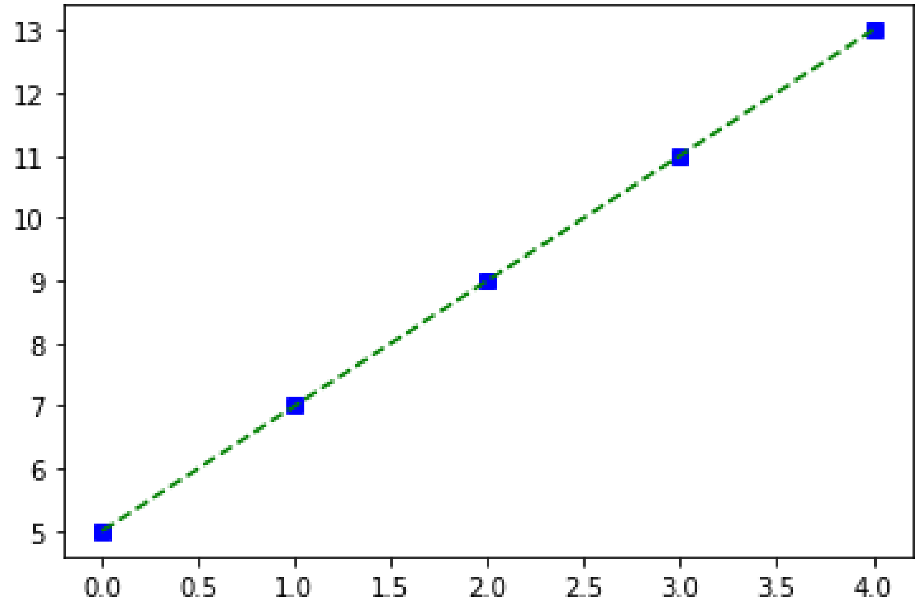
`numpy.polyval(p, x)`

Evaluate a polynomial at specific values.

- For simple polynomial fits use `polyfit`
- Use the fit results in `polyval`

```
xdata = np.arange(0,5)
m = 2
b = 5
ydata = m*xdata + b
p = np.polyfit(xdata,ydata,1)
print('Actual:\t',m,b)
print('Fit:\t',p)

x = np.linspace(0,4)
y = np.polyval(p, x)

plt.plot(xdata,ydata,'bs')
plt.plot(x,y,'g--')
plt.show()
```

```
Actual:   2 5
Fit:      [2. 5.]
```



Use `curve_fit` to fit data to an arbitrary function

Be careful! It will only find a **local** minima from the initial guess

`scipy.optimize.least_squares(fun, x0, jac='2-point', bounds=(- inf, inf), method='trf', ftol=1e-08, xtol=1e-08, gtol=1e-08, x_scale=1.0, loss='linear', f_scale=1.0, diff_step=None, tr_solver=None, tr_options={}, jac_sparsity=None, max_nfev=None, verbose=0, args=(), kwargs={})` [source]

Solve a nonlinear least-squares problem with bounds on the variables.

Given the residuals f(x) (an m-D real function of n real variables) and the loss function rho(s) (a scalar function), `least_squares` finds a local minimum of the cost function F(x):

```
minimize F(x) = 0.5 * sum(rho(f_i(x)**2), i = 0, ..., m - 1)
subject to lb <= x <= ub
```

The purpose of the loss function rho(s) is to reduce the influence of outliers on the solution.

Parameters: **fun** : *callable*

Function which computes the vector of residuals, with the signature `fun(x, *args, **kwargs)`, i.e., the minimization proceeds with respect to its first argument. The argument `x` passed to this function is an ndarray of shape (n,) (never a scalar, even for n=1). It must allocate and return a 1-D array_like of shape (m,) or a scalar. If the argument `x` is complex or the function `fun` returns complex residuals, it must be wrapped in a real function of real arguments, as shown at the end of the Examples section.

`scipy.optimize.curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False, check_finite=True, bounds=(- inf, inf), method=None, jac=None, *, full_output=False, **kwargs)` [source]

Use non-linear least squares to fit a function, f, to data.

Assumes `ydata = f(xdata, *params) + eps`.

Parameters: **f** : *callable*

The model function, f(x, …). It must take the independent variable as the first argument and the parameters to fit as separate remaining arguments.

xdata : *array_like or object*

The independent variable where the data is measured. Should usually be an M-length sequence or an (k,M)-shaped array for functions with k predictors, but can actually be any object.

ydata : *array_like*

The dependent data, a length M array - nominally `f(xdata, ...)`.

p0 : *array_like, optional*

Initial guess for the parameters (length N). If None, then the initial values will all be 1 (if the number of parameters for the function can be determined using introspection, otherwise a ValueError is raised).

sigma : *None or M-length sequence or MxM array, optional*

Determines the uncertainty in *ydata*. If we define residuals as `r = ydata - f(xdata, *popt)`, then the interpretation of *sigma* depends on its number of dimensions:

- A 1-D *sigma* should contain values of standard deviations of errors in *ydata*. In this case, the optimized function is `chisq = sum((r / sigma) ** 2)`.
- A 2-D *sigma* should contain the covariance matrix of errors in *ydata*. In this case, the optimized function is `chisq = r.T @ inv(sigma) @ r`.

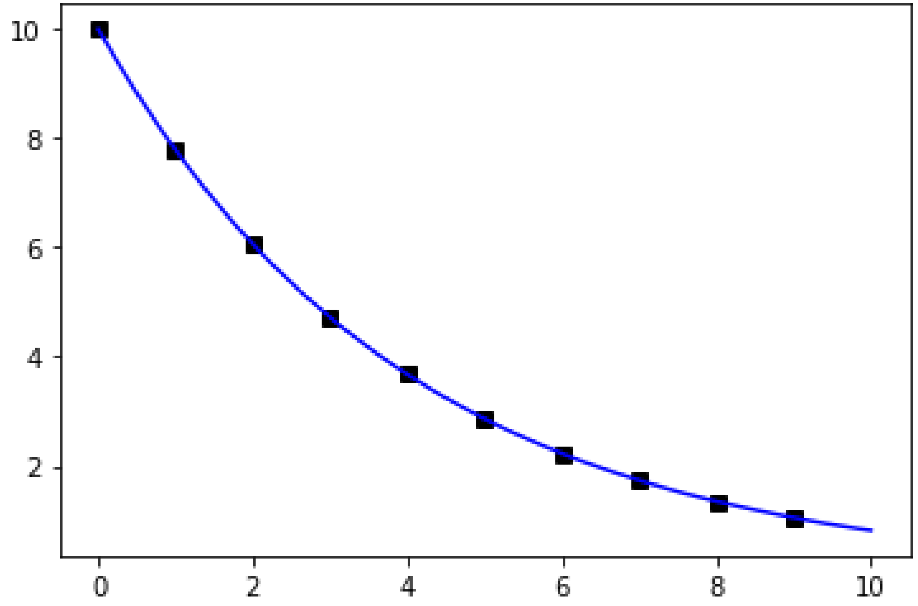
```
scipy.optimize.curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False,
check_finite=True, bounds=(- inf, inf), method=None, jac=None, *, full_output=False,
**kwargs)
```

[source]

```python
from scipy.optimize import curve_fit
```

```python
def fitfun(x,a,b):
    return a*np.exp(b*x)

popt, pcov = curve_fit(fitfun,xdata,ydata)
print(popt)

xfit = np.linspace(0,NUM)
yfit = fitfun(xfit, *popt) # *popt unpacks array into function args

plt.plot(xdata,ydata,'ks')
plt.plot(xfit,yfit,'b-')
```

```
[10.00313449 -0.25024117]
[<matplotlib.lines.Line2D at 0x7fd78ca56650>]
```




```python
NUM = 10
xdata = np.arange(0,NUM)
ydata = 10*np.exp(-xdata*0.25) + 0.005*np.random.randn(NUM)
```

Advanced Fitting

Global Optimization

https://docs.scipy.org/doc/scipy/reference/optimize.html#global-optimization

Global optimization

		Notes
basinhopping(func, x0[, niter, T, stepsize, ...])	Find the global minimum of a function using the basin-hopping algorithm.	needs initial guess and “temperature”
brute(func, ranges[, args, Ns, full_output, ...])	Minimize a function over a given range by brute force.	give ranges, will “polish” final result
differential_evolution(func, bounds[, args, ...])	Finds the global minimum of a multivariate function.	great! Needs bounds
shgo(func, bounds[, args, constraints, n, ...])	Finds the global minimum of a function using SHG optimization.	(haven’t used this much)
dual_annealing(func, bounds[, args, ...])	Find the global minimum of a function using Dual Annealing.	(haven’t used this much)
direct(func, bounds, *[, args, eps, maxfun, ...])	Finds the global minimum of a function using the DIRECT algorithm.	(haven’t used this much)

curve_fit is a local optimizer from your initial guess. You can always try a global optimizer instead!

iminuit

- Yes, `curve_fit` is easy to use but a little dumb
- Need an industrial-strength fitter? Try iminuit
- https://iminuit.readthedocs.io/en/stable/index.html

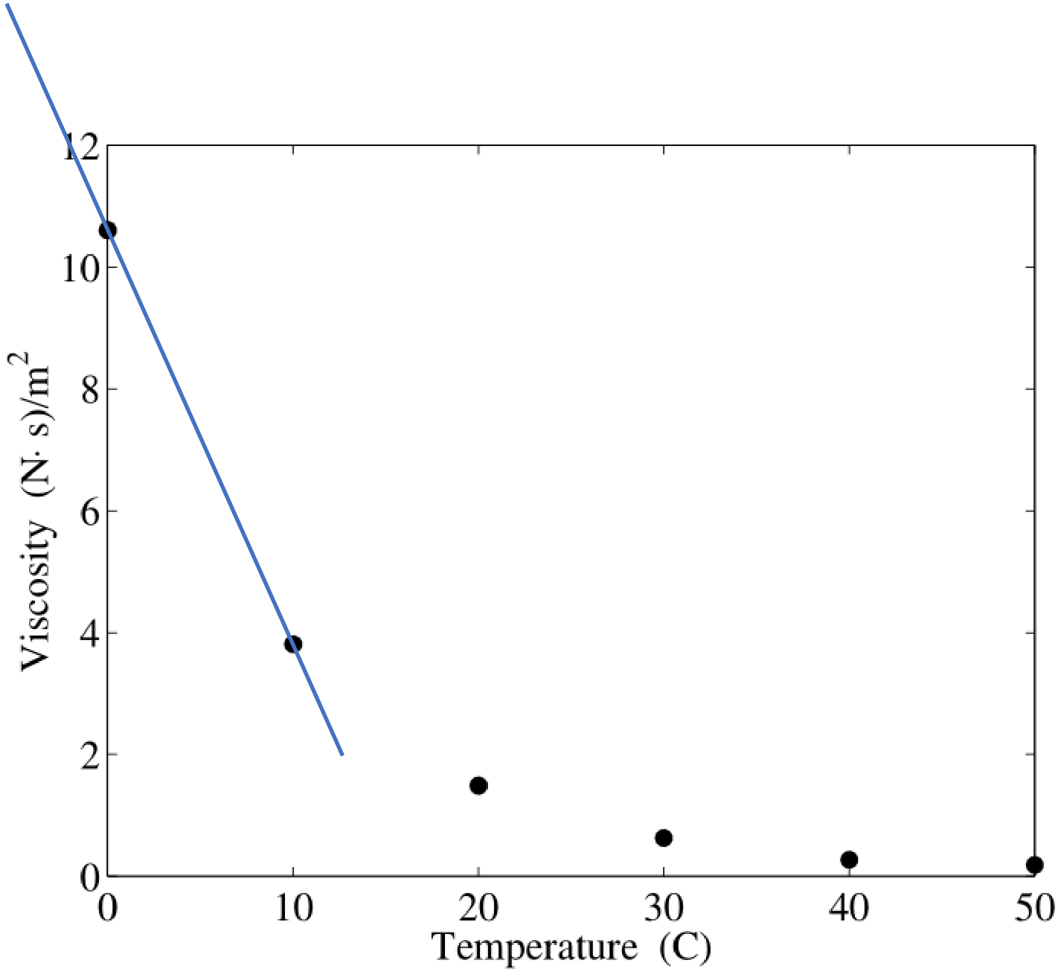
Fitting Notes

- easy for simple polynomials
- do NOT use high order polynomials
- you may need to transform your data first
  - example: if y(x) is exponential, try fitting the log of y

- General purpose fitting to an arbitrary function is hard, but tools do exist…

Interpolation

Problem



Do a linear interpolation by hand

Given:

x1 = 0,   y1 = 10.5

x2 = 10, y2 = 4

Find y at x=5

Linear

https://numpy.org/doc/stable/reference/generated/numpy.interp.html

numpy.interp

numpy.interp(*x, xp, fp, left=None, right=None, period=None*) [source]

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*

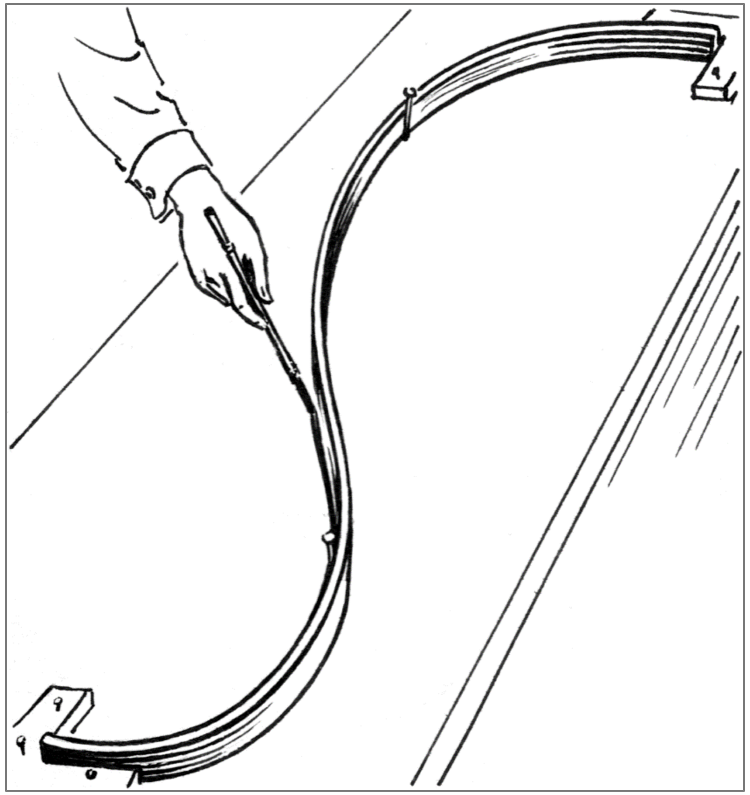
The y-coordinates of the data points, same length as *xp*.

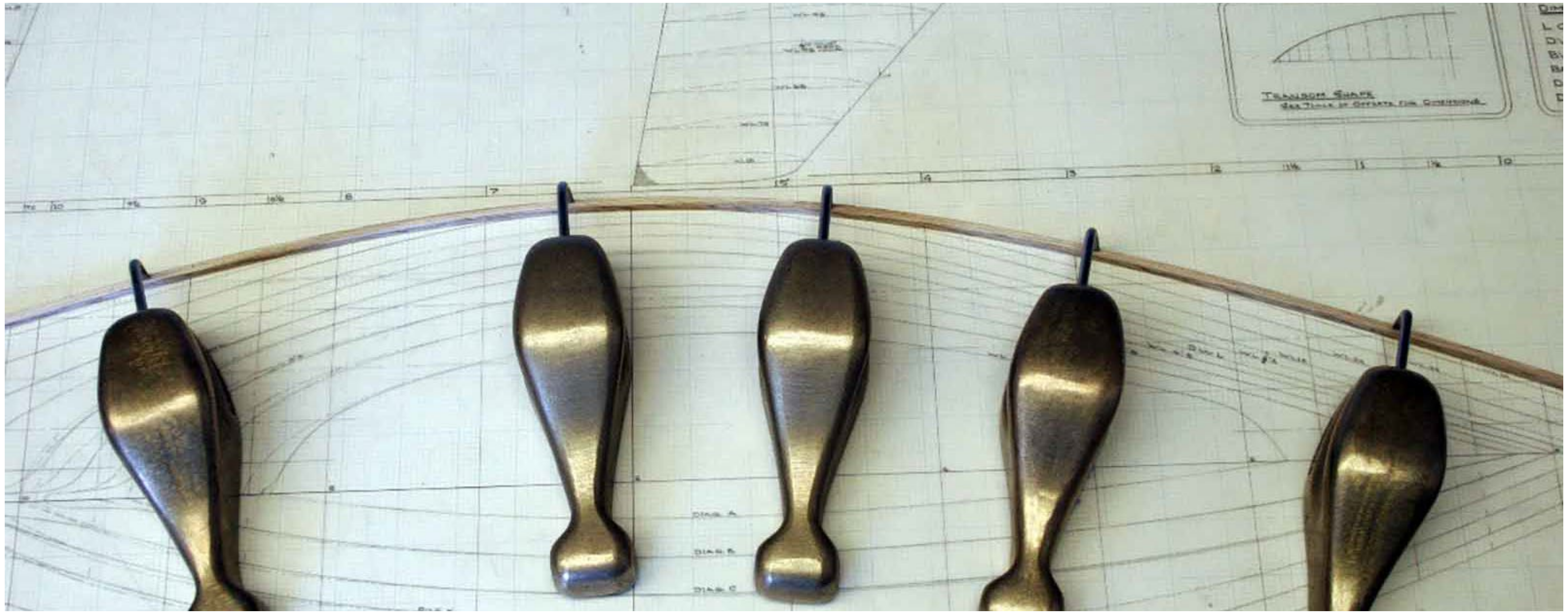
```
xp = [1, 2, 3]
fp = [3, 2, 0]
np.interp(2.5, xp, fp)
```

```
1.0
```

- Use **np.interp** for linear interpolation
- WARNING: make sure your data points are sorted in increasing order

Splines!





Cubic Splines

We gets lots of advantages skipping from linear to cubic!

Given N data points we construct N-1 cubic polynomials with **4(N-1)** unknown coefficients

Constraint	**Number**
Each (N-1) poly goes through data points on both ends	2(N-1)
First derivatives match at interior points $P'_{i-1}(x_i) = P'_i(x_i)$	N-2
Second derivatives match at interior points $P''_{i-1}(x_i) = P''_i(x_i)$	N-2
TOTAL	**4N-6**

Need 2 more constraints! Use boundary conditions at endpoints

Cubic Splines

Common choices for constraints:

- **fixed-slope:** user specifies slope at endpoints
- **clamped:** first derivative at endpoints is zero
- **natural:** second derivative at endpoints is zero
- **not-a-knot** (*default*): if we require $P_1'''(x_2) = P_2'''(x_2)$ the third derivative matches at the first interior point then the first two segments P_1 and P_2 have the same coefficients. The "knots" are the data points, so now x_2 is no longer a true knot. Use this option if you don't have any information about the endpoint slopes

Cubic

https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.CubicSpline.html#scipy.interpolate.CubicSpline

scipy.interpolate.CubicSpline

```
class CubicSpline(x, y, axis=0, bc_type='not-a-knot', extrapolate=None)
```
[source]

Cubic spline data interpolator.

Interpolate data with a piecewise cubic polynomial which is twice continuously differentiable [1]. The result is represented as a `PPoly` instance with breakpoints matching the given data.

Parameters:

x : ***array_like, shape (n,)***

1-D array containing values of the independent variable. Values must be real, finite and in strictly increasing order.

y : ***array_like***

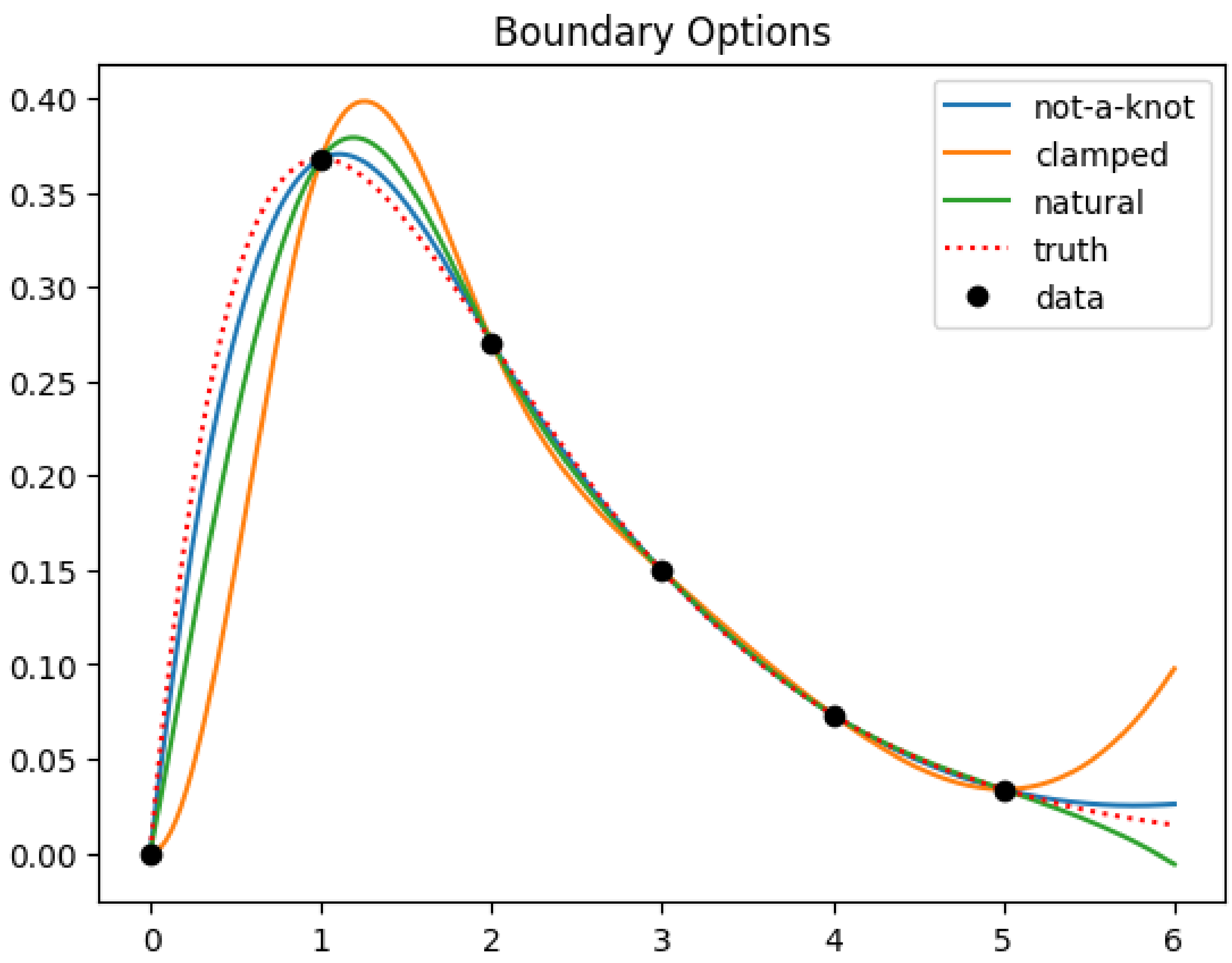
Array containing values of the dependent variable. It can have arbitrary number of dimensions, but the length along `axis` (see below) must match the length of `x`. Values must be finite.

bc_type : ***string or 2-tuple, optional***

Boundary condition type. Two additional equations, given by the boundary conditions, are required to determine all coefficients of polynomials on each segment [2].

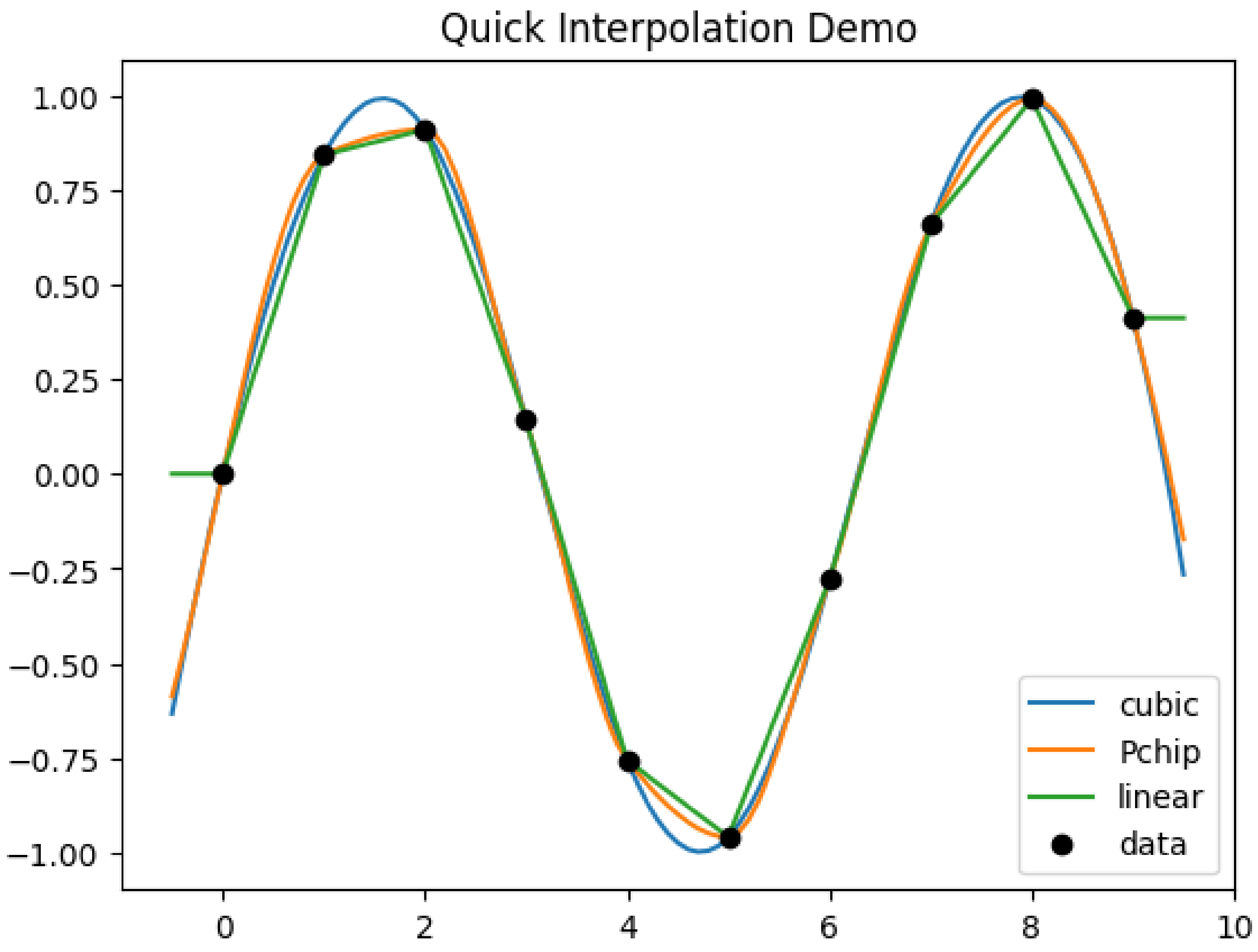
If *bc_type* is a string, then the specified condition will be applied at both ends of a spline. Available conditions are:

- 'not-a-knot' (default): The first and second segment at a curve end are the same polynomial. It is a good default when there is no information on boundary conditions.
- 'periodic': The interpolated functions is assumed to be periodic of period `x[-1] - x[0]`. The first and last value of *y* must be identical: `y[0] == y[-1]`. This boundary condition will result in `y'[0] == y'[-1]` and `y''[0] == y''[-1]`.
- 'clamped': The first derivative at curves ends are zero. Assuming a 1D *y*, `bc_type=((1, 0.0), (1, 0.0))` is the same condition.
- 'natural': The second derivative at curve ends are zero. Assuming a 1D *y*, `bc_type=((2, 0.0), (2, 0.0))` is the same condition.




```
def fun1(x):
    return x*np.exp(-x)

xdata = np.arange(0,6)
ydata = fun1(xdata)

xs = np.linspace(0, 6, 200)

for bc in ['not-a-knot', 'clamped', 'natural']:
    cs = CubicSpline(xdata, ydata, bc_type=bc)
    plt.plot(xs, cs(xs), label=bc)

plt.plot(xs, fun1(xs), 'r:',label='truth')
plt.plot(xdata,ydata,'ko', label='data')
plt.title('Boundary Options')
plt.legend()
plt.show()
```

The last option I'll mention is PCHIP (Piecewise Cubic Hermite Interpolating Polynomial). This is a middle ground between linear and cubic since the function and first derivatives match, but the second derivatives don't. The result are splines that are still smooth but much "flatter" with very little overshoot.




```
xdata = np.arange(10)
ydata = np.sin(xdata)
cs = CubicSpline(xdata, ydata)
csp = PchipInterpolator(xdata, ydata)

xs = np.arange(-0.5, 9.6, 0.1)

plt.plot(xs,cs(xs), label='cubic')
plt.plot(xs,csp(xs), label='Pchip')
plt.plot(xs, np.interp(xs, xdata, ydata),label='linear')
plt.plot(xdata,ydata,'ko', label='data')
plt.title('Quick Interpolation Demo')
plt.legend()
plt.show()
```

Interpolation Summary

- Lots of options: https://docs.scipy.org/doc/scipy/tutorial/interpolate.html
- **np.interp** for linear interpolation
- **scipy.interpolate.CubicSpline** for cubic
- 3rd option: if you really need an option that is smooth but as flat as possible you can use scipy.interpolate.PchipInterpolator

- WARNING: make sure your data points are sorted in increasing order

Things to Know

- When to interpolate and when to fit
- Fitting
  - polyfit and polyval
  - curve_fit
  - advanced tricks like brute or differential_evolution
- Interpolation
  - np.interp
  - scipy.Interpolate.CubicSpline