# Minimization and Curve Fitting with SciPy

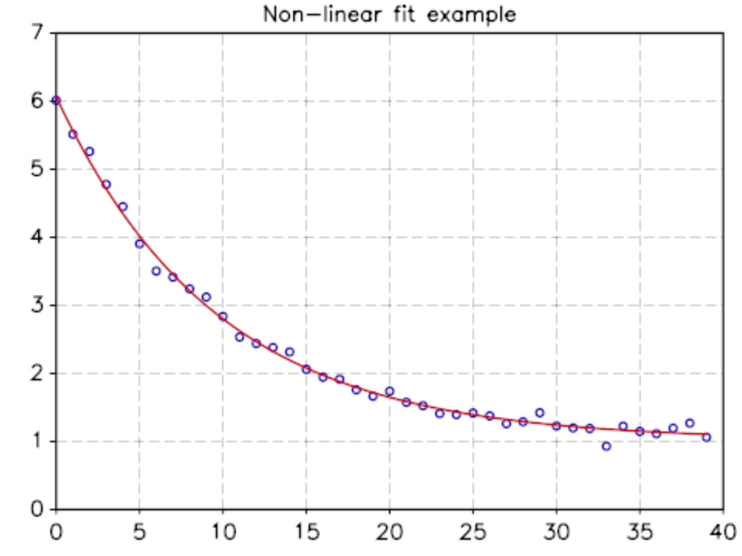
Thomas Schwarz, SJ

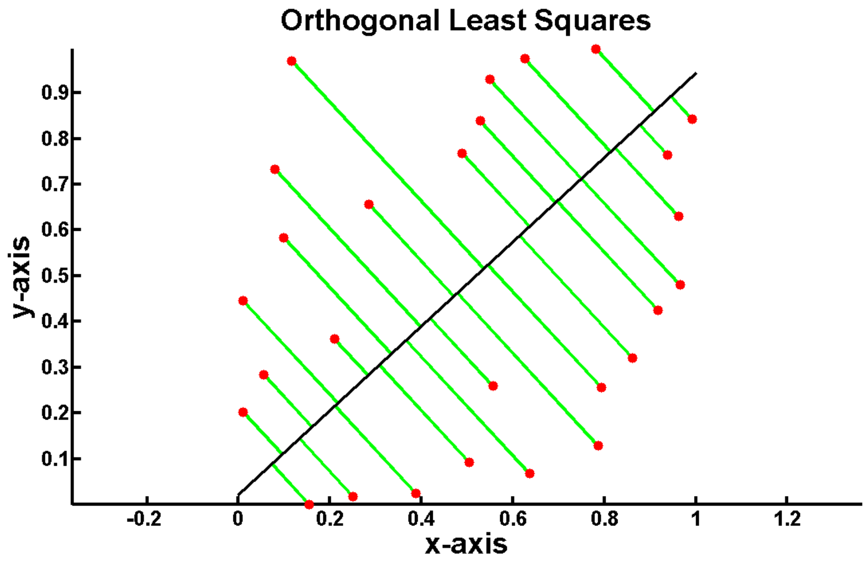
# Curve Fitting

- Want to construct a curve (mathematical function) that best fits a series of data points
  - First, need to select a model: what type of curve?
  - Then, need to determine how we measure fit
    - Examples:
      - y-values:

$$L(y, \hat{y}) = \sum_{\nu=1}^{n} (y_i - \hat{y}(i))^2 \rightarrow \min$$

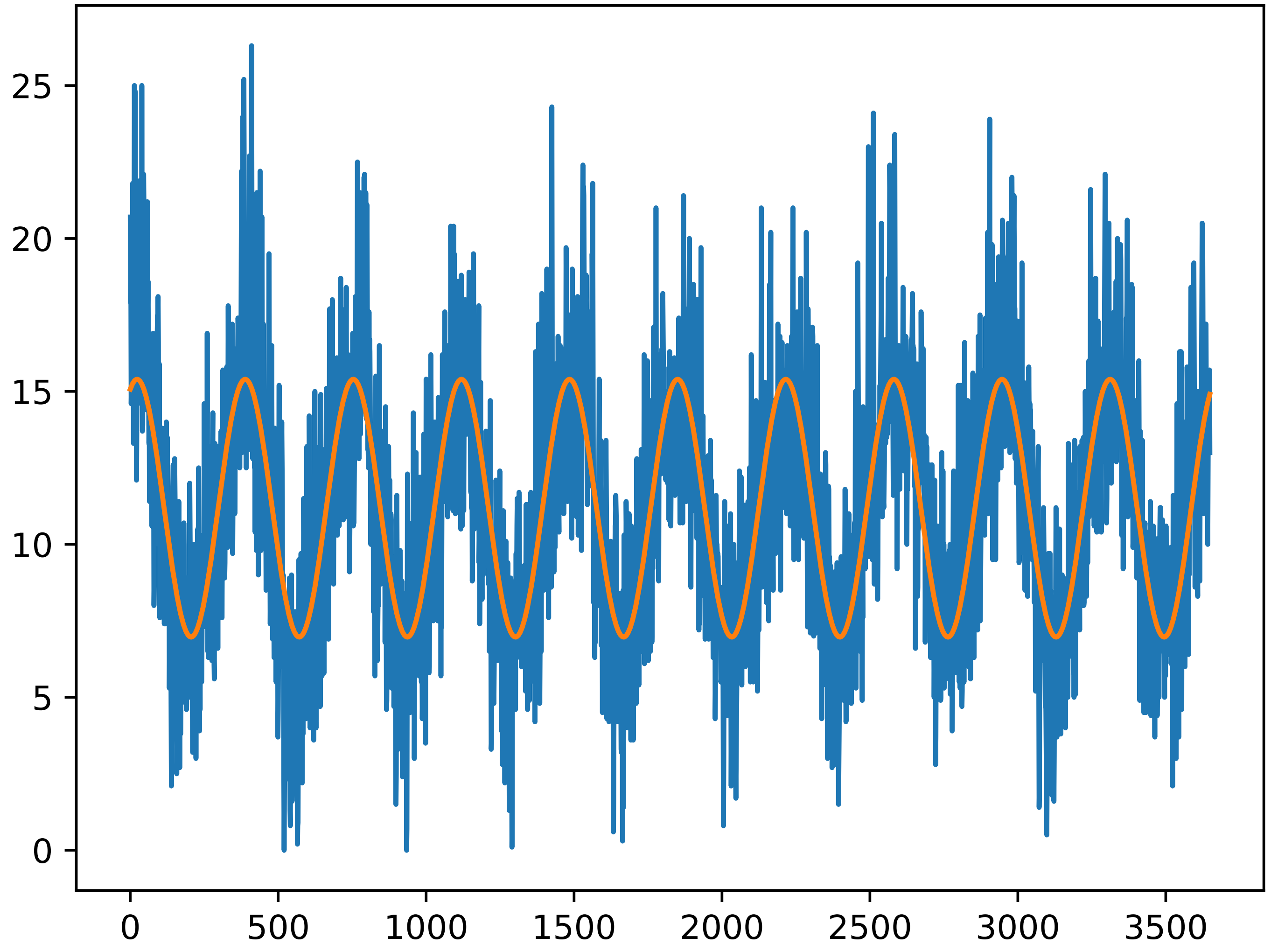
      - orthogonal least squares



# Curve Fitting

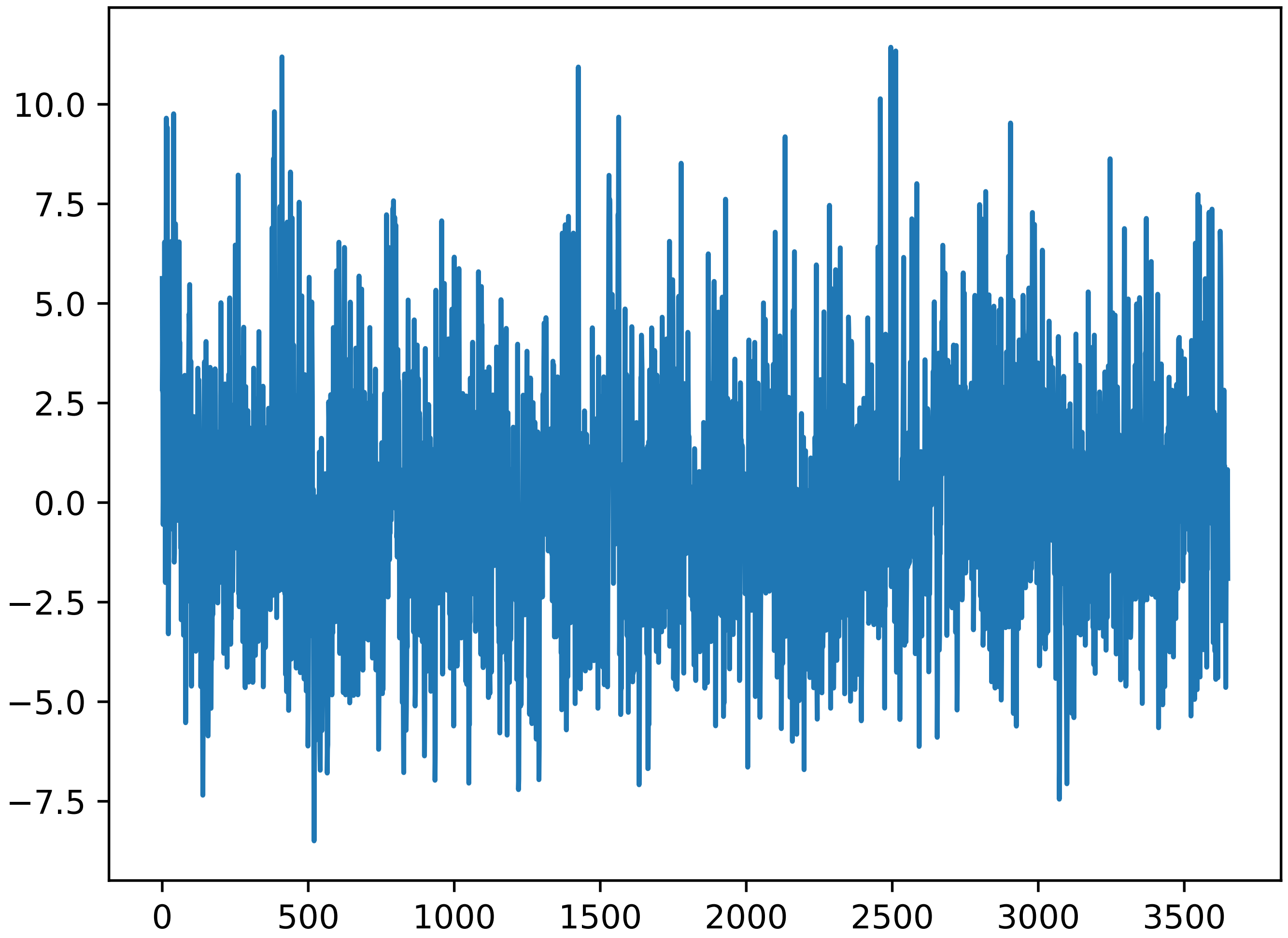
- Example:  Fit a sine curve to meteorological data
  - Minimum daily temperatures in Melbourne




$$f_{\alpha,\beta,\gamma,\delta}(t) = \alpha + \beta \sin(\gamma t + \delta)$$

# Curve Fitting

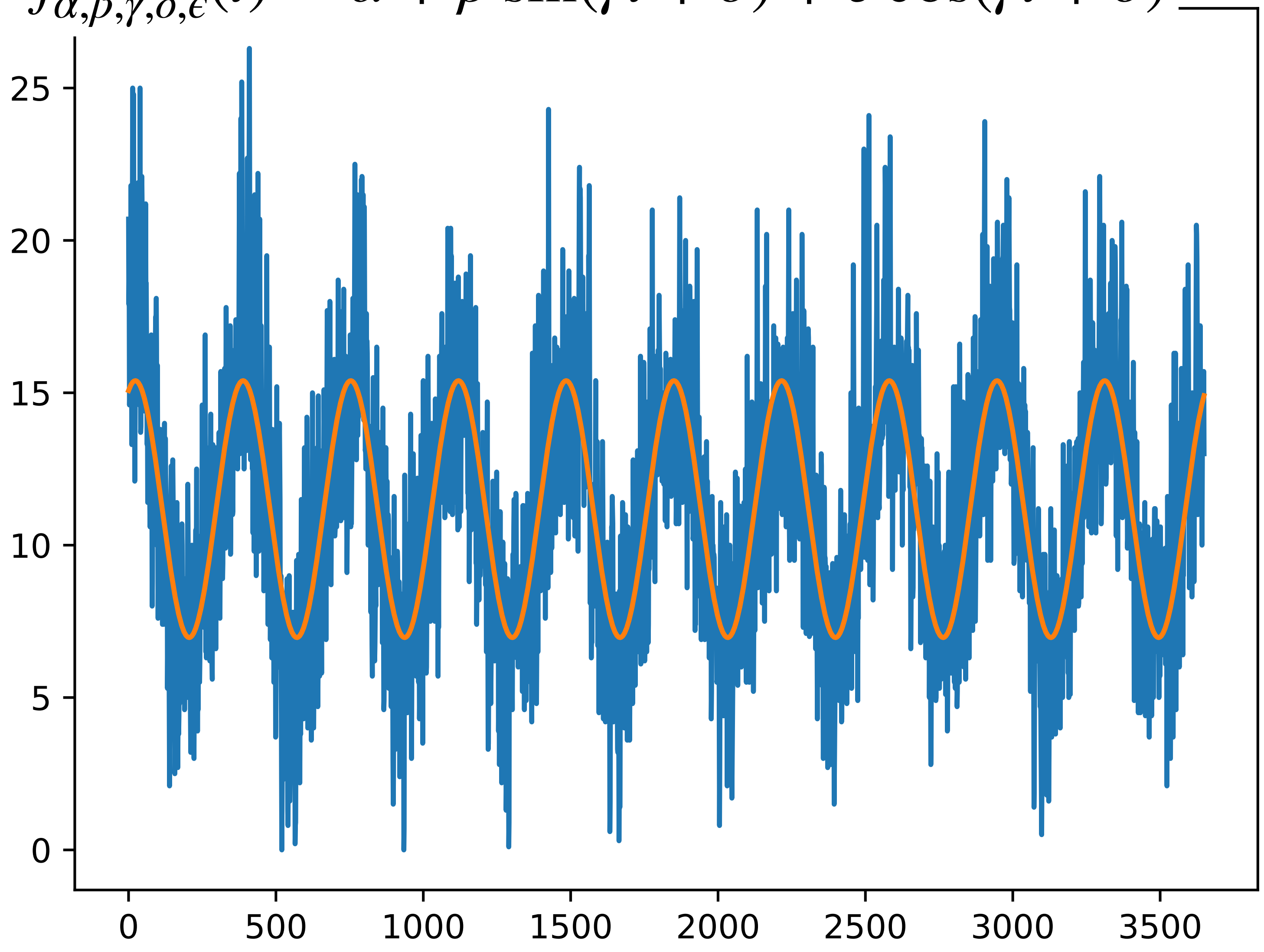
- The data after removing the sine curve shows a seemingly random time series with just a little bit of seasonality



# Curve Fitting

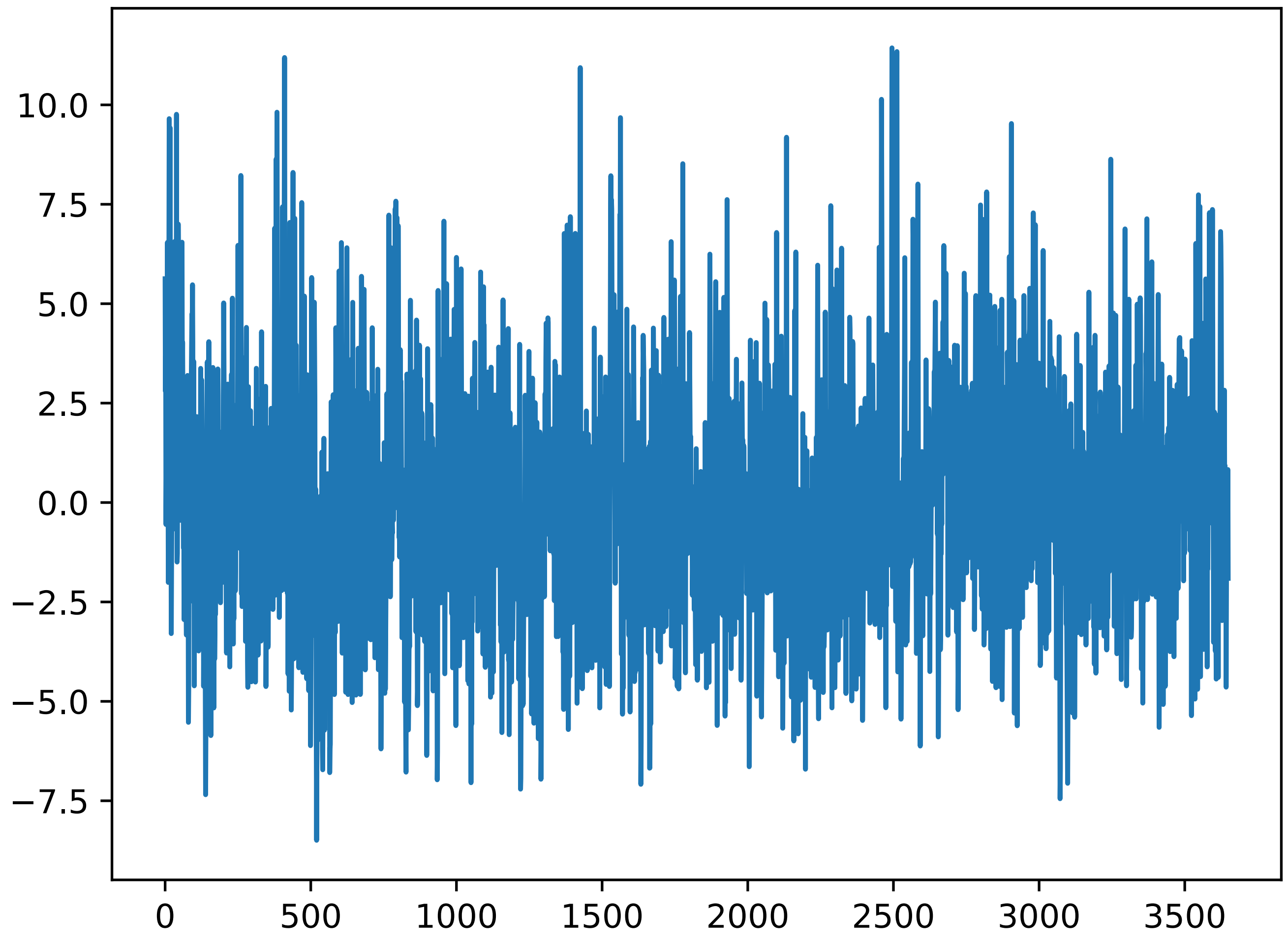
- We can do better by including a cos

$$f_{\alpha,\beta,\gamma,\delta,\epsilon}(t) = \alpha + \beta \sin(\gamma t + \delta) + \epsilon \cos(\gamma t + \delta)$$



# Curve Fitting

- Residual:
  - Looks slightly better?



# Curve Fitting

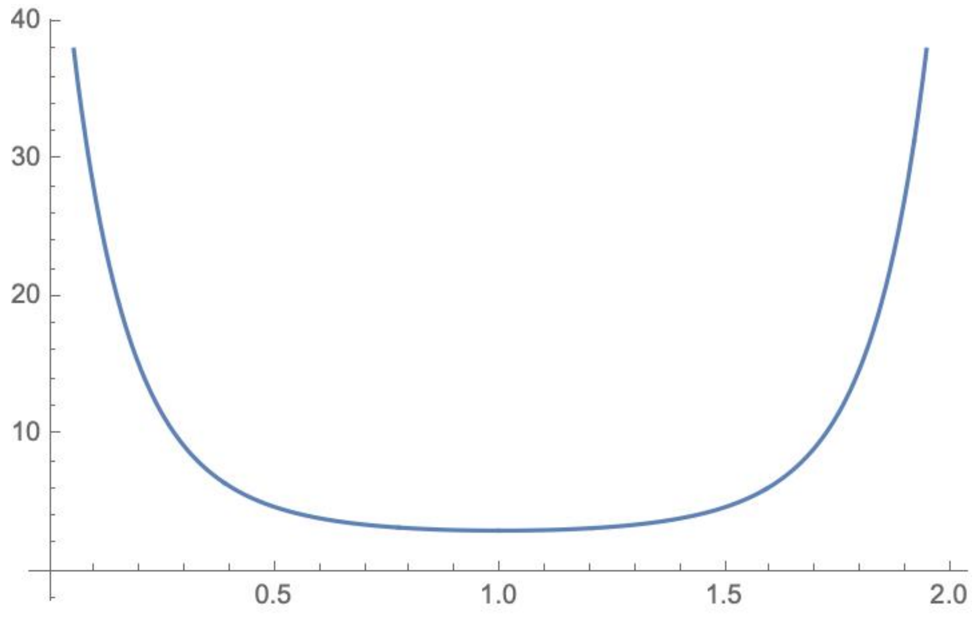
- Find the parameters that minimize the squared difference between function and model
  - This is a minimization problem
- Too general a model:
  - Optimization can be very difficult and lengthy
  - Overfit: The result matches the test set, but not the future
- Not general a model
  - Fit is not good, therefore no strong predictions either

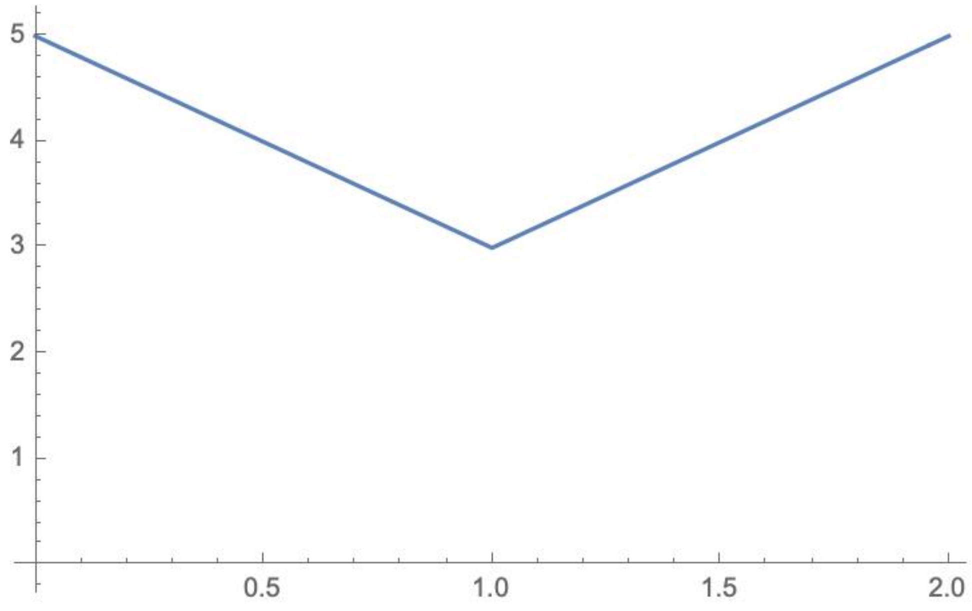
# Program

- Need to learn about minimization
  - One dimensional methods: Minimization along a line
  - Gradient Descent Methods
  - Minimization for Sums of Squares
  - Curve-fitting

# Minimization

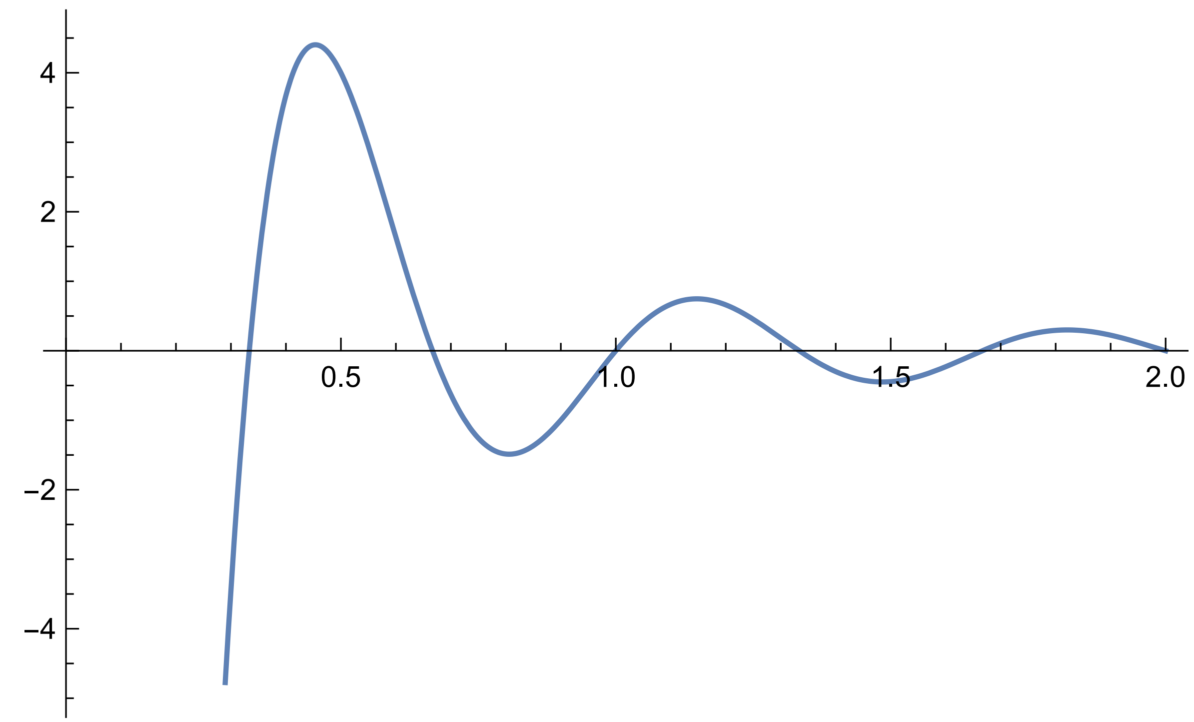
- Functions can be smooth and non-smooth



# Minimization

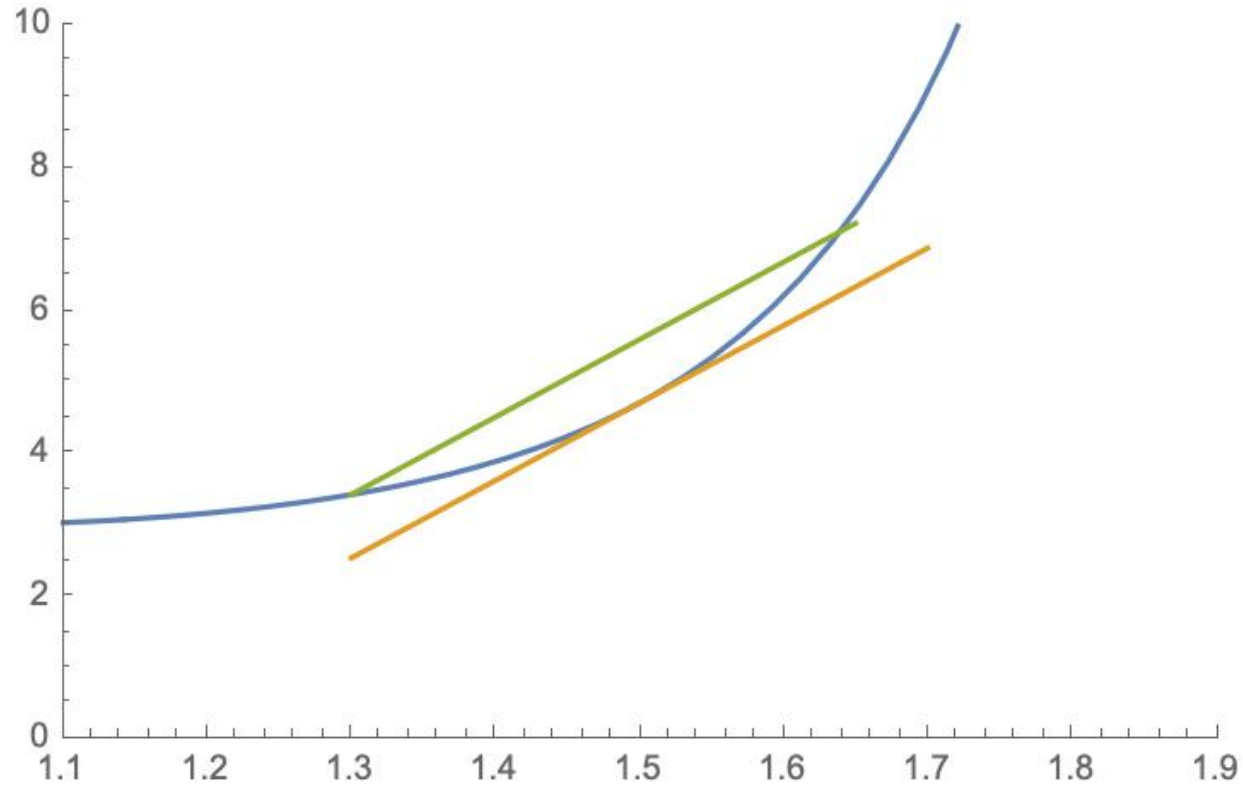
- Given a function $\mathscr{R}^n \rightarrow \mathscr{R}$
  - Find a minimum




- Potential problems:
  - Minimum might not exist
  - Minimum might be local

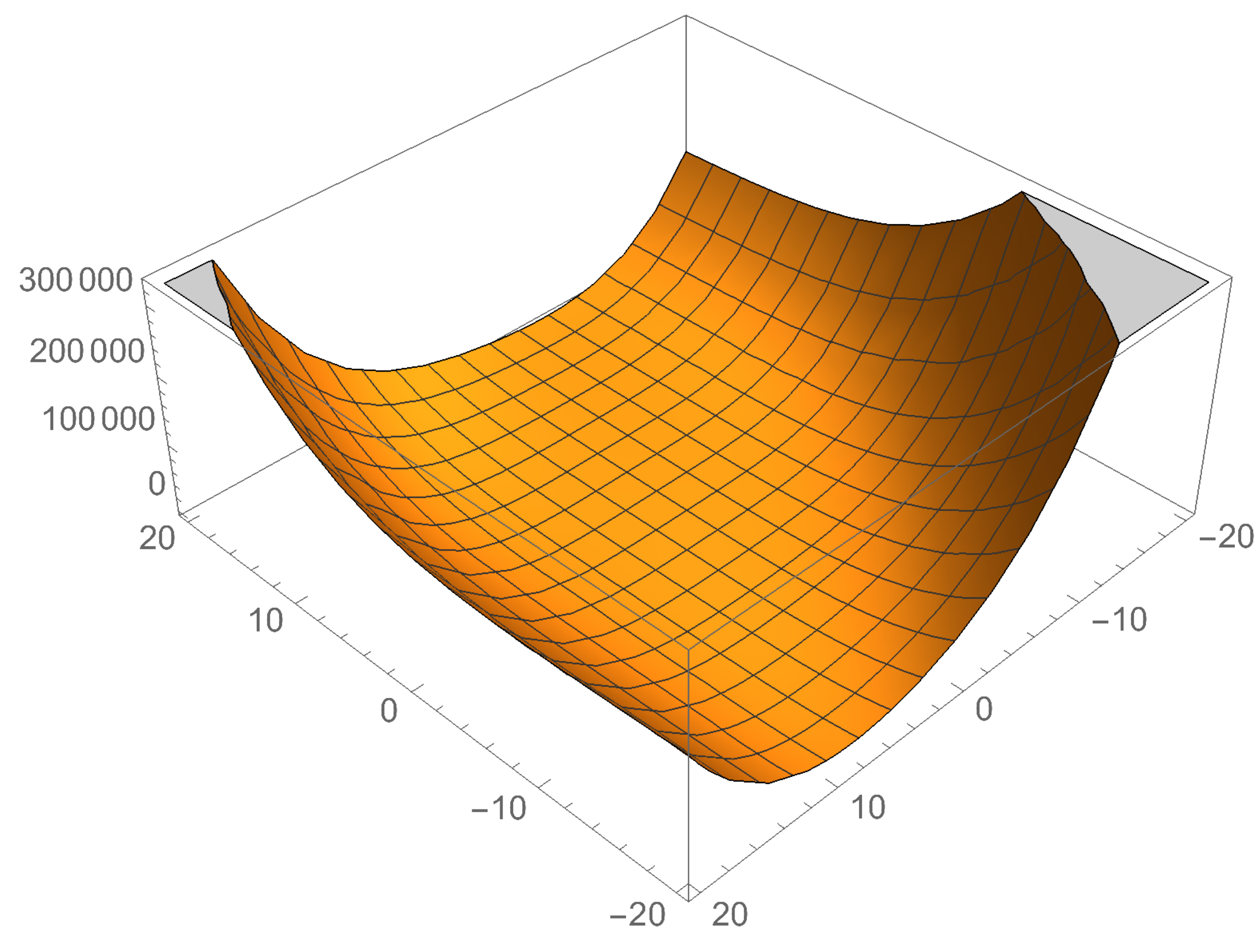
# Minimization

- Convex functions: For $t \in [0,1]$ :
  - $f(\overrightarrow{a} + t(\overrightarrow{b} - \overrightarrow{a})) \leq f(\overrightarrow{a}) + t(f(\overrightarrow{b}) - f(\overrightarrow{a}))$
  - Tends to be easy
  - Relative minimum is unique



# Minimization

- Minimization is easier for convex functions



# Minimization

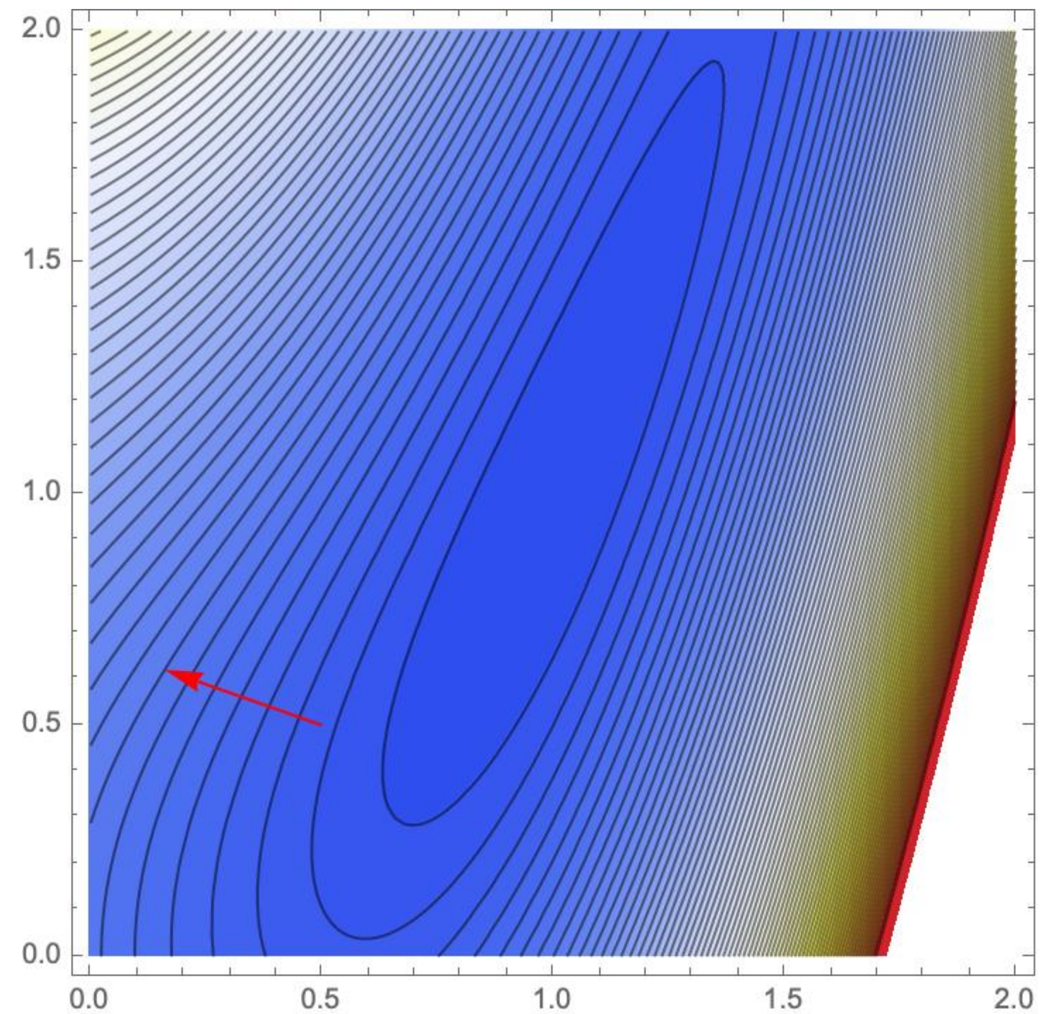
- Smooth functions are (usually) easier than non-smooth functions
  - Exception: Linear systems with constraints —> Linear Programming

# Minimization

- Gradient $\nabla f = (\frac{df}{dx_1}, \frac{df}{dx_2}, \frac{df}{dx_3}, \ldots, \frac{df}{dx_n})$ is always in the direction of greatest increase of a function

# Minimization

- Example: Rosenbrock Function
  - $f(x, y) = 1.2(y - x^2)^2 + 1.1(1 - x)^2$
  - Gradient is $(-2.2(1 - x) - 4.8x(-x^2 + y), 2.4(-x^2 + y))$
  - Contour graph is



# Minimization

- Descent Methods:

1. Choose a starting point $x_0 \in \mathscr{R}^n$
2. If $\|\nabla f(x_k)\| < \epsilon$ declare victory and return $x_k$
3. Pick a search direction $d_k \in \mathscr{R}^n$ s.t. $\nabla f(x_k) \cdot d_k < 0$
4. Choose a step size $\alpha_k > 0$ s.t. $f(x_k + \alpha_k d_k) < f(x_k)$
5. Set $x_{k+1} = x_k + \alpha_k d_k$. Go to 2

# Minimization

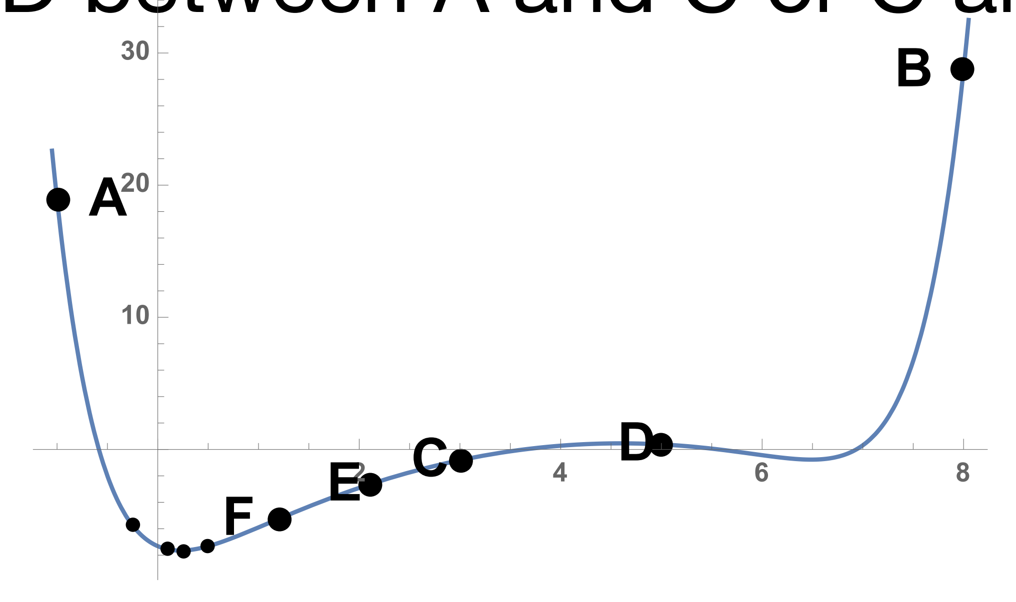
- This algorithm leaves two things open:
  - Selecting the step length $\alpha_k$
  - Selecting the search direction $d_k$

# Minimization

- Finding minimum along line:
    - Finding minimum of function $t \mapsto f(x_k + td_k)$
    - Use derivative is usually dangerous:
        - Often function too flat
        - Better bracketing

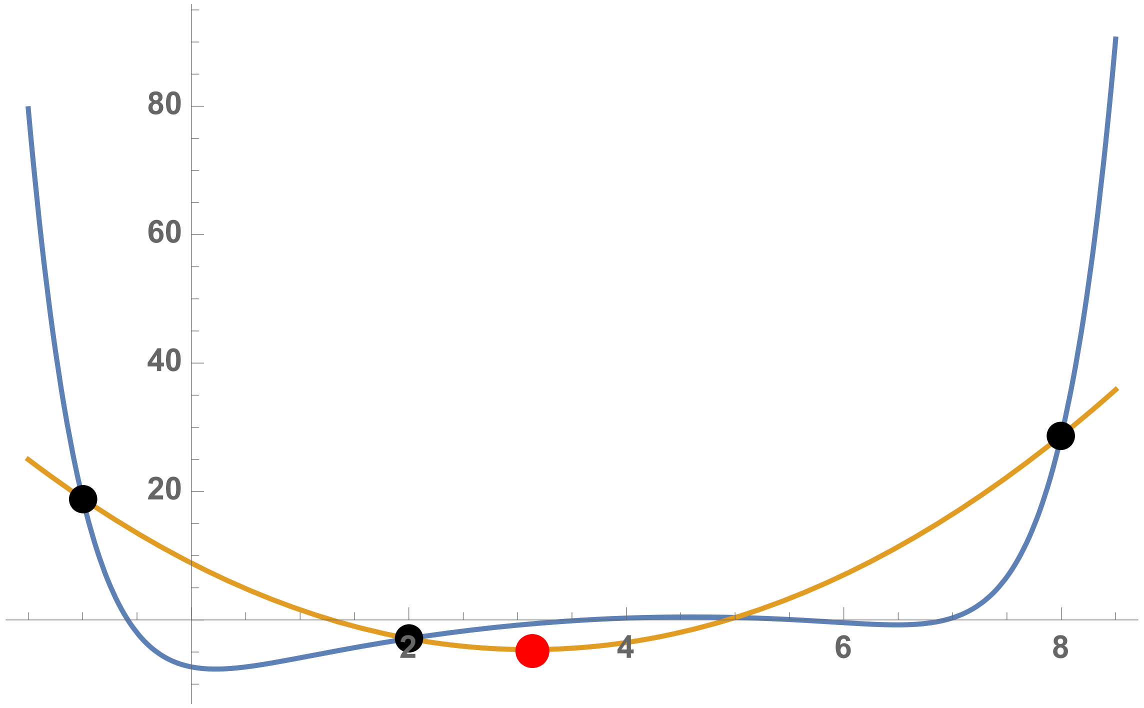
# Minimization

- Can use bracketing
  - Three points A < C < B such that $f(A) > f(C) < f(B)$
    - Thus, minimum guaranteed to exist
    - Now find another point D between A and C or C and B
    - Get a new bracket



# Minimization

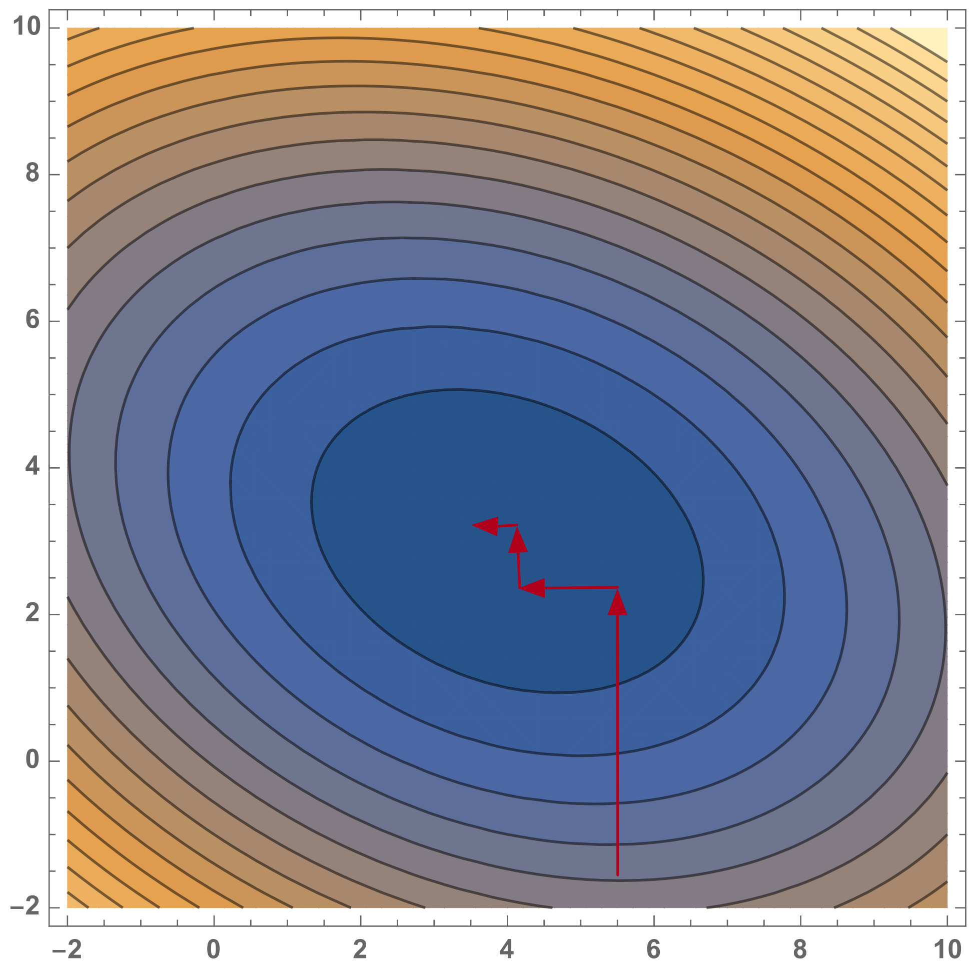
- One possibility: golden ratio: $\dfrac{|A-C|}{|A-B|} = \dfrac{|B-C|}{|A-C|}$
- Other possibility: parabolic approximation
- Or a combination of both

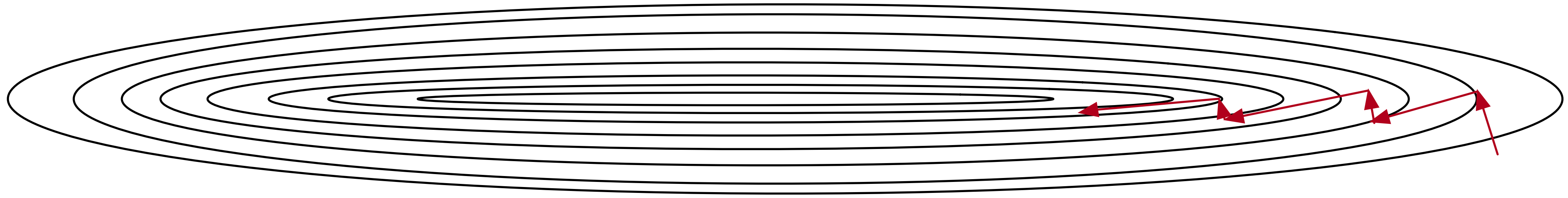


# Minimization

- Determining direction:
  - Can use coordinates

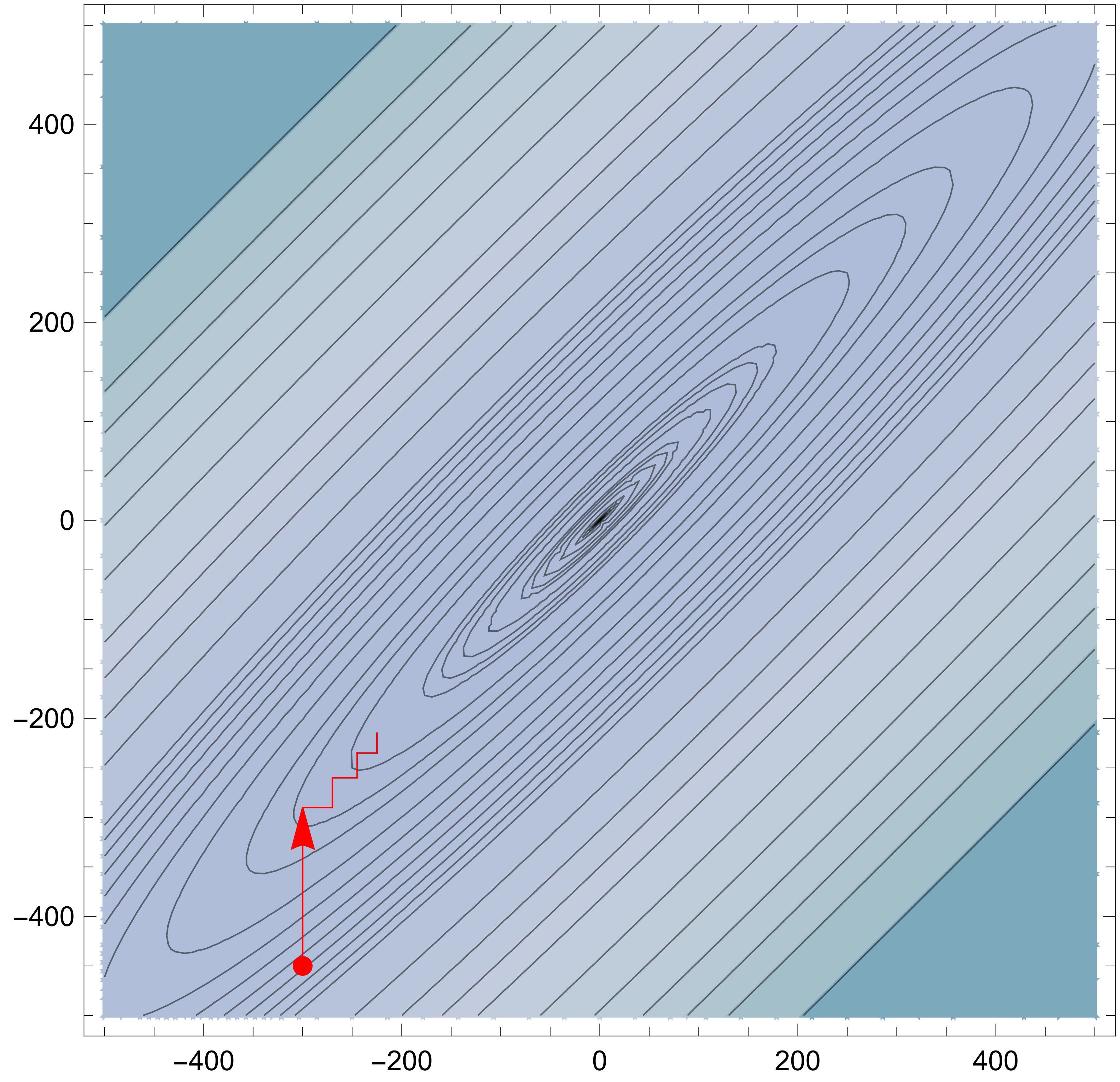
# Minimization



# Minimization

- Using coordinates
  - Make little progress per iteration



# Minimization

- Better use orthogonal directions:
  - otherwise we partially undo the previous steps
- Possibilities
  - Canonical Directions
  - Steepest Descent (Cauchy)
    - badly affected by round-off errors and subject to zigzagging
  - Powell: Change set of directions every so often

# Minimization

- Selecting the step length
  - Finding the best step length is laborious
  - Often do better by guessing
  - Many machine learning algorithms use a steadily declining $\alpha$
    - Trying out several guesses

# Minimization

- Newton Methods:
  - Repeatedly replace condition $\nabla f(x) = 0$ by a sequence of linear problems
  - Newton-Raphson:
    - apply **exact** Newton steps
- possibly does not converge
- works best for convex functions
  - Use linear-search descent, then switch to Newton

# Minimization

- Numerical minimization of a cost function of the parameters
  - Various minimization methods
    - Line methods
      - Minimize along a particular line

# Minimization

- Instead of line searches: **Trust Region Methods**
  - **Idea:** for each iteration: replace $f$ with a **quadratic model function**
    - Quadratic model function approximates $f$ in the "trust region"
    - And quadratic model functions are easy to minimize!
    - The proposed solution can or cannot have a smaller value for $f$
    - Many different ways of defining the model

# Minimization with SciPy

- Can use a number of method for finding a local minimum
  - Some need the Jacobian and some need in addition the Hessian
  - Can be calculated numerically but results are better with exact functions
  - Jacobian $\mathbf{J}(f) = \frac{df}{dx} = (\frac{\delta f}{\delta x_1}, \frac{\delta f}{\delta x_2}, \ldots, \frac{\delta f}{\delta x_n})$
  - Hessian $\mathbf{H}(f) = \left( \frac{\delta^2 f}{\delta x_i \delta x_j} \right)_{i,j}$

# Minimization with SciPy

- Use scipy.optimize.minimize
  - Needs a function that is a one-dimensional np.array
  - Specify a starting point, options, and method

# Minimization with SciPy

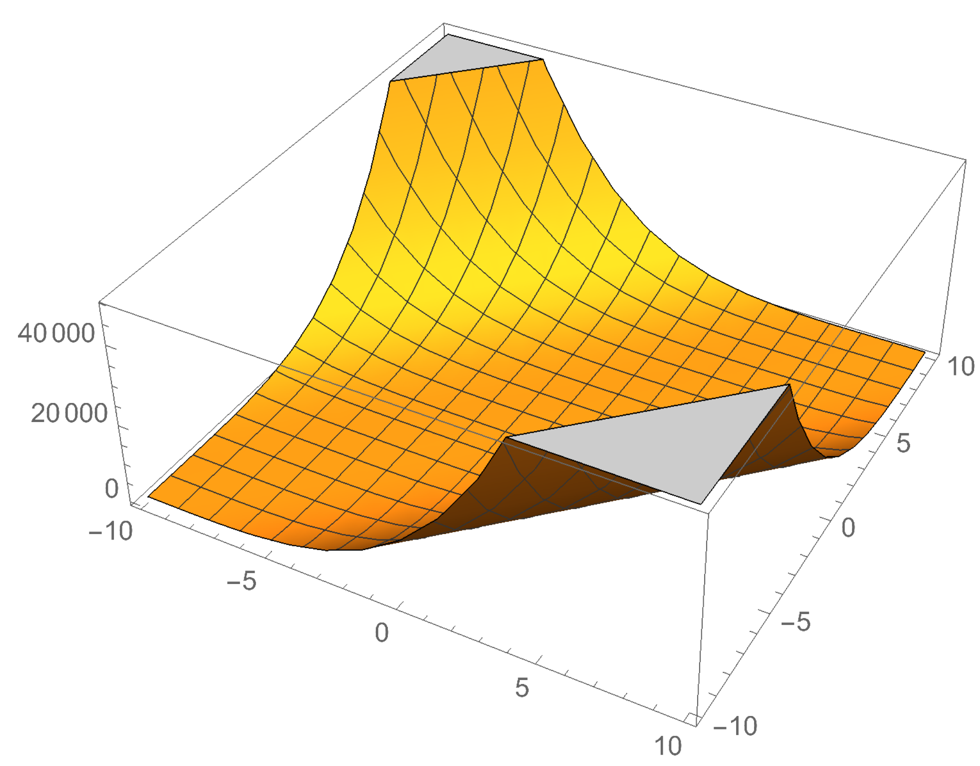
- Importing the optimizer:

```
import numpy as np
from scipy.optimize import minimize
```

# Minimization with SciPy

- Defining a function to be minimized
  - Needs to be in "standard form", i.e. numpy array of one dimension

```
def func(x):
    return np.sin(x[0]*x[1])+(x[0]+x[1]-1)**2+(x[0]-x[1]+1)**4
```



# Minimization with SciPy

- Sometimes need to give Jacobian
-
```
def jacob(x):
    return np.array(
      (4*(1+x[0]-x[1])**3 +
       2*(x[0]+x[1]-1)+
       x[1]*np.cos(x[0]*x[1]),
       -4*(1+x[0]-x[1])**3 +
       2*(x[0]+x[1]-1)+
       x[0]*np.cos(x[0]*x[1])) )
```

# Minimization with SciPy

- We pick (5,5) as the starting point

```
res = minimize(func,
               [5,5],
               method = 'nelder-mead',
               options = {'xatol': 1e-9, 'disp':True}
               )

print(res.x)
```

# Minimization with SciPy

- Success: (but with lots of function evaluations)

```
Optimization terminated successfully.
         Current function value: -0.295490
         Iterations: 84
         Function evaluations: 164
[-0.37249737  1.18821832]
```

# Minimization with SciPy

```
res = minimize(func,
               [5,5],
               method = 'Newton-CG',
               jac = jacob,
               options = {'disp':True}
               )
print(res.x)
```

# Minimization with SciPy

```
Optimization terminated successfully.
         Current function value: -0.295490
         Iterations: 10
         Function evaluations: 12
         Gradient evaluations: 47
         Hessian evaluations: 0
[-0.37249737  1.18821832]
```

# Minimization with SciPy

```
res = minimize(func,
               [5,5],
               method = 'Powell',
               options = {'disp':True}
               )
print(res.x)
```

# Minimization with SciPy

```
Optimization terminated successfully.
         Current function value: -0.295490
         Iterations: 5
         Function evaluations: 139
[-0.37249969  1.18821518]
```

# Minimization with SciPy

```
res = minimize(func,
               [5,5],
               method = 'BFGS',
               options = {'disp':True}
               )
print(res.x)
```

# Minimization with SciPy

```
Optimization terminated successfully.
         Current function value: -0.295490
         Iterations: 15
         Function evaluations: 84
         Gradient evaluations: 21
[-0.37249848  1.18821848]
```

# Least Square Optimization

- Want to minimize a sum of squares
  - $f : \mathscr{R}^n \rightarrow \mathscr{R}^m$

$$f(\overrightarrow{x}) = \frac{1}{2} \sum_{j=1}^{m} r_j^2(\overrightarrow{x})$$

  - (Factor of 1/2 to make derivatives look nicer)

# Least Square Optimization

- With this special form, we can calculate the Jacobian of $(r_1(\overrightarrow{x}), r_2(\overrightarrow{x}), \ldots, r_m(\overrightarrow{x}))^T$ more easily

$$\mathbf{J}^T = \begin{pmatrix} \frac{\delta r_1}{\delta x_1} & \frac{\delta r_2}{\delta x_1} & \cdots & \frac{\delta r_m}{\delta x_1} \\ \frac{\delta r_1}{\delta x_2} & \frac{\delta r_2}{\delta x_2} & \cdots & \frac{\delta r_m}{\delta x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta r_1}{\delta x_n} & \frac{\delta r_2}{\delta x_n} & \cdots & \frac{\delta r_m}{\delta x_n} \end{pmatrix} = \left( \frac{\delta r_j}{\delta x_i} \right)$$

# Least Square Optimization

- Then

$$\nabla f(\overrightarrow{x}) = \mathbf{J}(\overrightarrow{x})^T \mathbf{r}(x)$$

$$\nabla^2 f(\overrightarrow{x}) = \mathbf{J}(\overrightarrow{x})^T \mathbf{J}(\overrightarrow{x}) + \sum_{j=1}^{m} r_j(\overrightarrow{x}) \nabla^2 r_j(\overrightarrow{x})$$

# Least Square Optimization

- Now we assume that $||\mathbf{r}(\overrightarrow{x})||$ is linear
  - Then $\mathbf{J}$ is a constant
  - $\nabla^2(r_j)(\overrightarrow{x}) = 0$
  - Taylor expansion is

$$f(\overrightarrow{x}) = f(x_0) + \mathbf{J}(f(x))|_{\overrightarrow{x}_0}(\overrightarrow{x} - \overrightarrow{x}_0) + \frac{1}{2}(\overrightarrow{x} - \overrightarrow{x}_0)^T\mathbf{H}(f(\overrightarrow{x})|_{\overrightarrow{x}_0}(\overrightarrow{x} - \overrightarrow{x}_0) + \dots$$

- Taking derivatives gives
- $\nabla f(\overrightarrow{x}) = \mathbf{J}^T(\mathbf{J}\overrightarrow{x} + \vec{r}) = 0$
- at a minimum

# Least Square Optimization

- This means we can solve for the minimum since then $\mathbf{J}^T\mathbf{J}\overrightarrow{x} = -\mathbf{J}^T\mathbf{r}(\overrightarrow{x})$
- and so we could solve $\overrightarrow{x} = (\mathbf{J}^T\mathbf{J})^{-1}\mathbf{J}^T\mathbf{r}$

# Least Square Optimization

- However, calculating the inverse is
  - computationally expensive
  - numerically unstable
  -

# Least Square Optimization

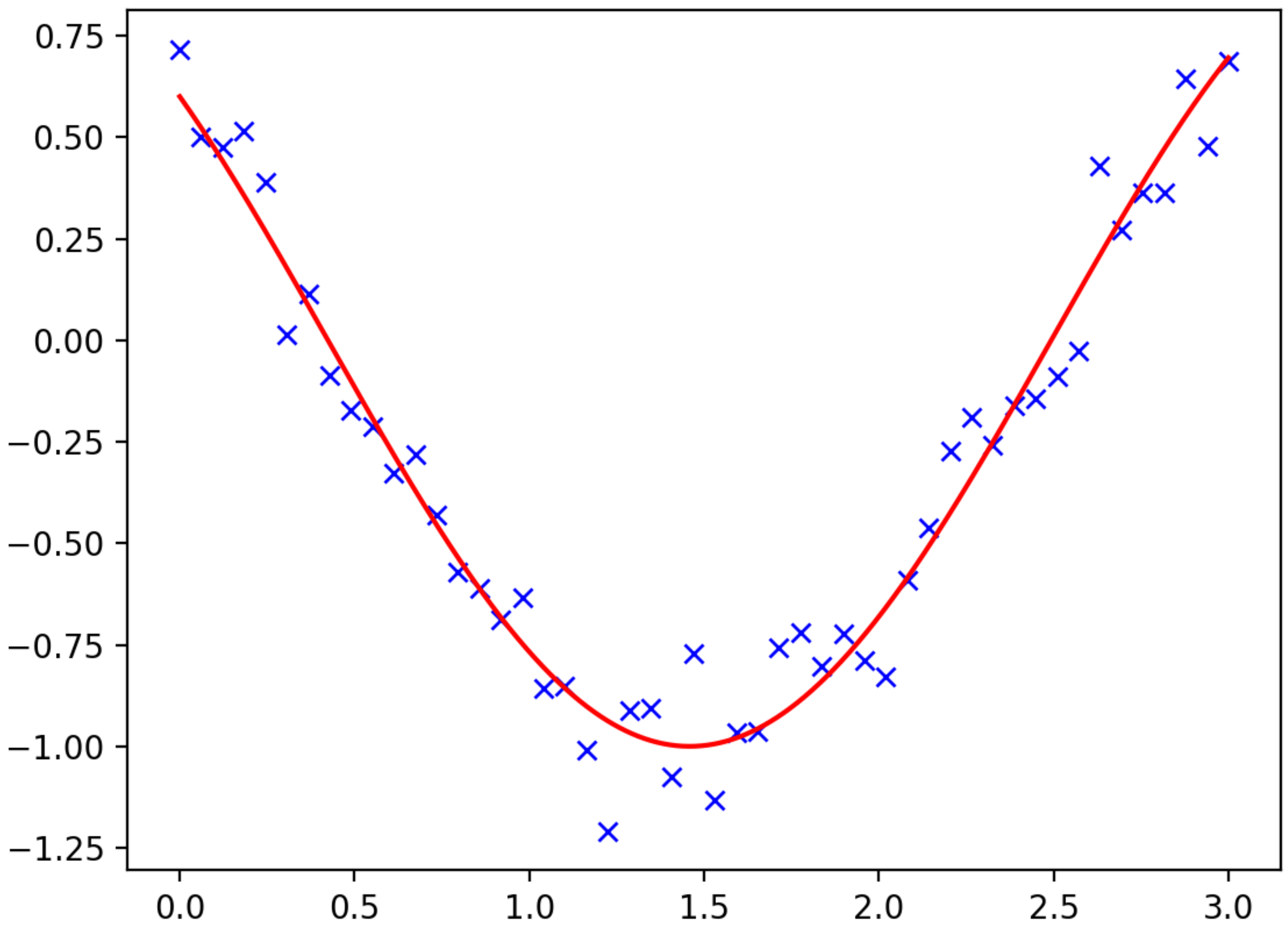
- Can use
  - Cholesky factorization
  - QR factorisation
  - Singular value decomposition of
- $\mathbf{J}^T\mathbf{J}$
  - which are all implemented in np.linalg

# Least Square Optimization

- Levenberg Marquardt algorithm
  - Even if $\mathbf{r}$ is not linear:
    - Assume that it is approximately
    - Use the above method as an approximator
    - Get results

# Curve Fitting

- Number of numerical methods for minimization problems
- Curve fitting:
  - Given a number of points, find a smooth curve going through it



# Curve Fitting

- Use a cosine as the test function

```
def f(t, omega, phi):
    return np.cos(omega * t + phi)
```

# Fitting with scipy

- Create sample data

```
x = np.linspace(0, 3, 50)
y = f(x, 1.5, 1) + .1*np.random.normal(size=50)
```

# Fitting with scipy

- Now fit using scipy.optimize

```
params, params_cov = optimize.curve_fit(f, x, y)
print(params)
```

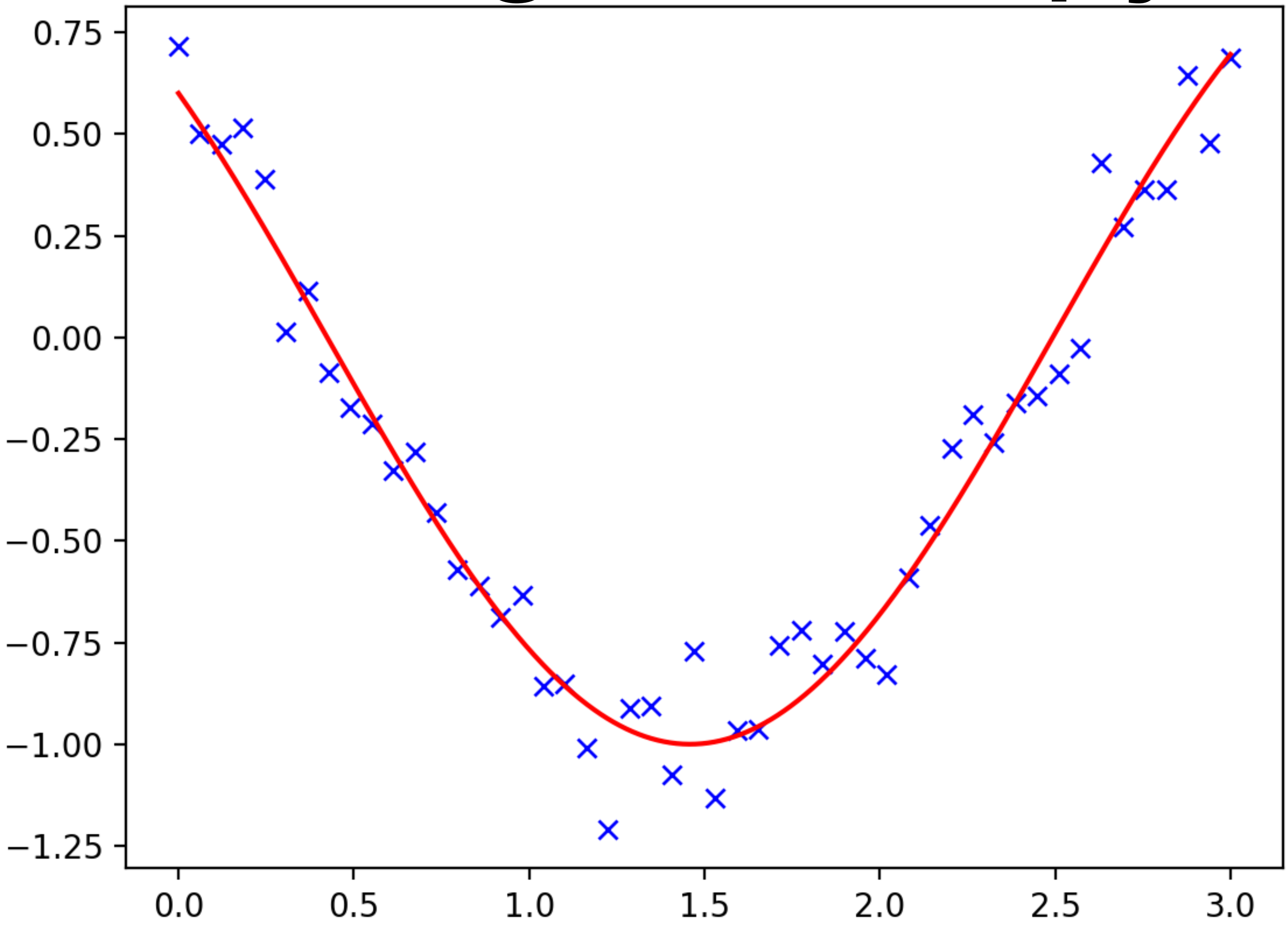
# Fitting with scipy

- Can almost recover parameters

```
y = f(x, 1.5, 1) + .1*np.random.normal(size=50)
```

```
[1.51854577 0.92665541]
```

# Fitting with scipy



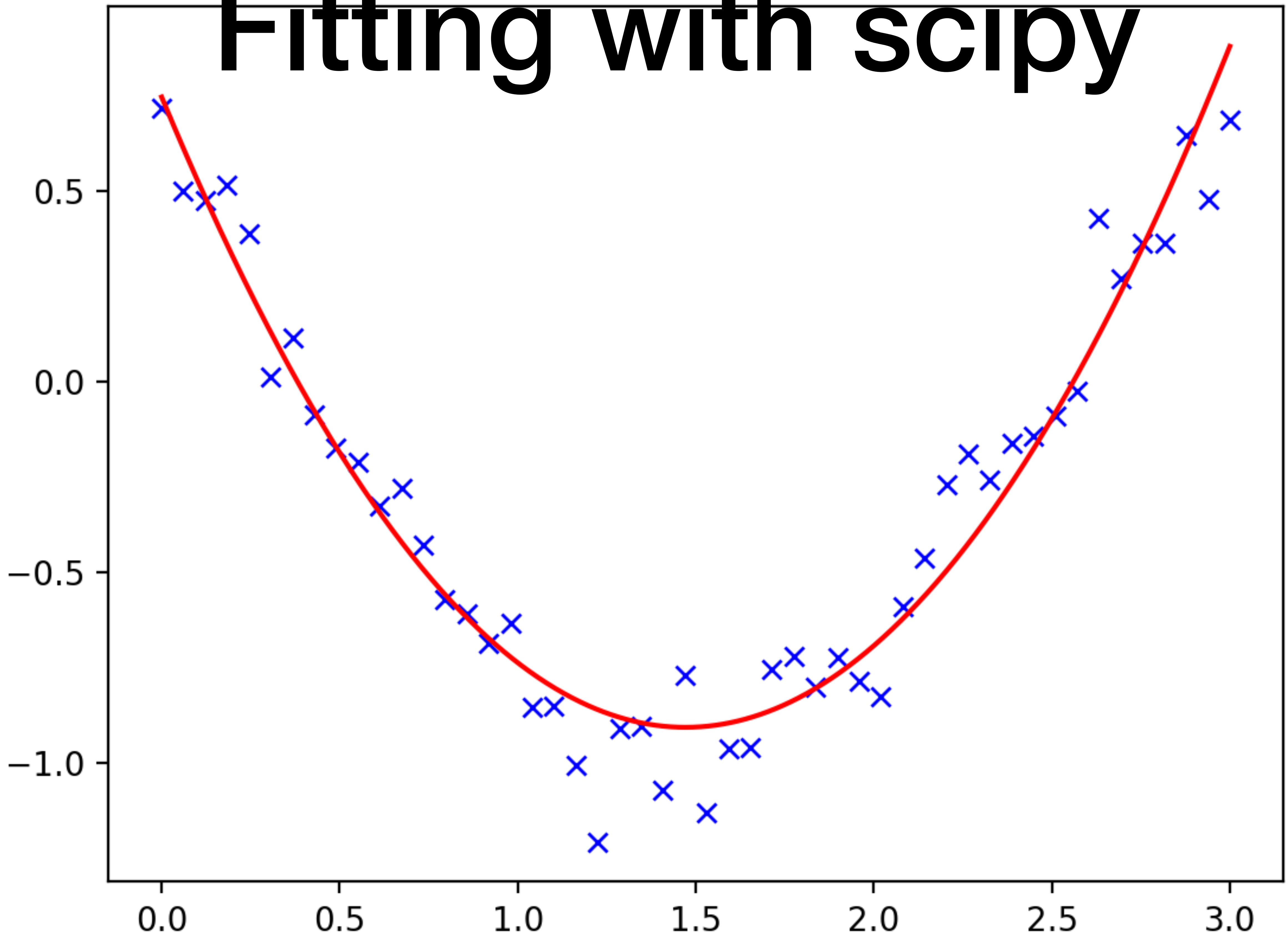
# Fitting with scipy

- Could also fit with a quadratic

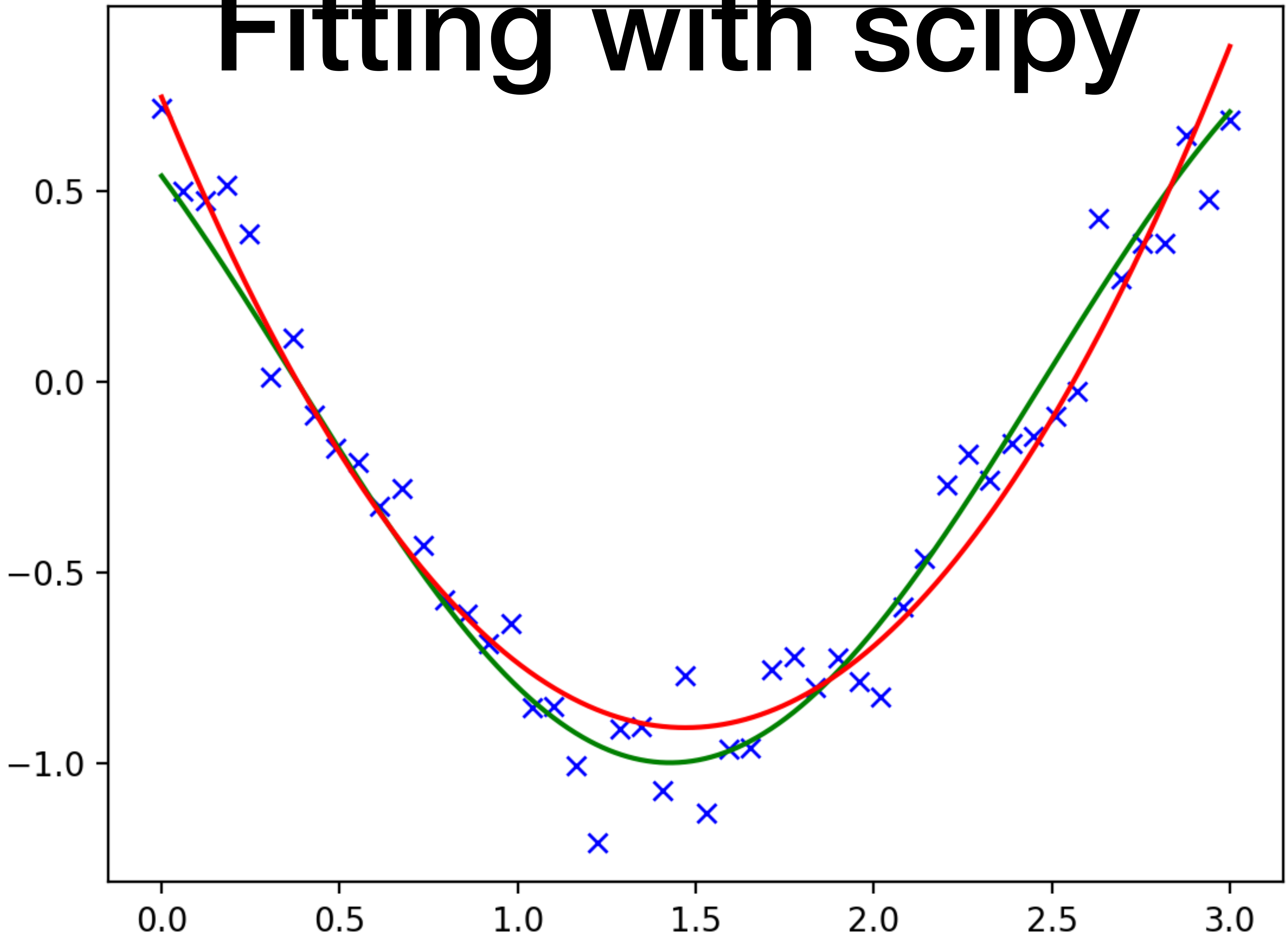
```
def g(t, a, b, c):
    return a*t**2+b*t+c

params, params_cov = optimize.curve_fit(g, x, y)
```

# Fitting with scipy

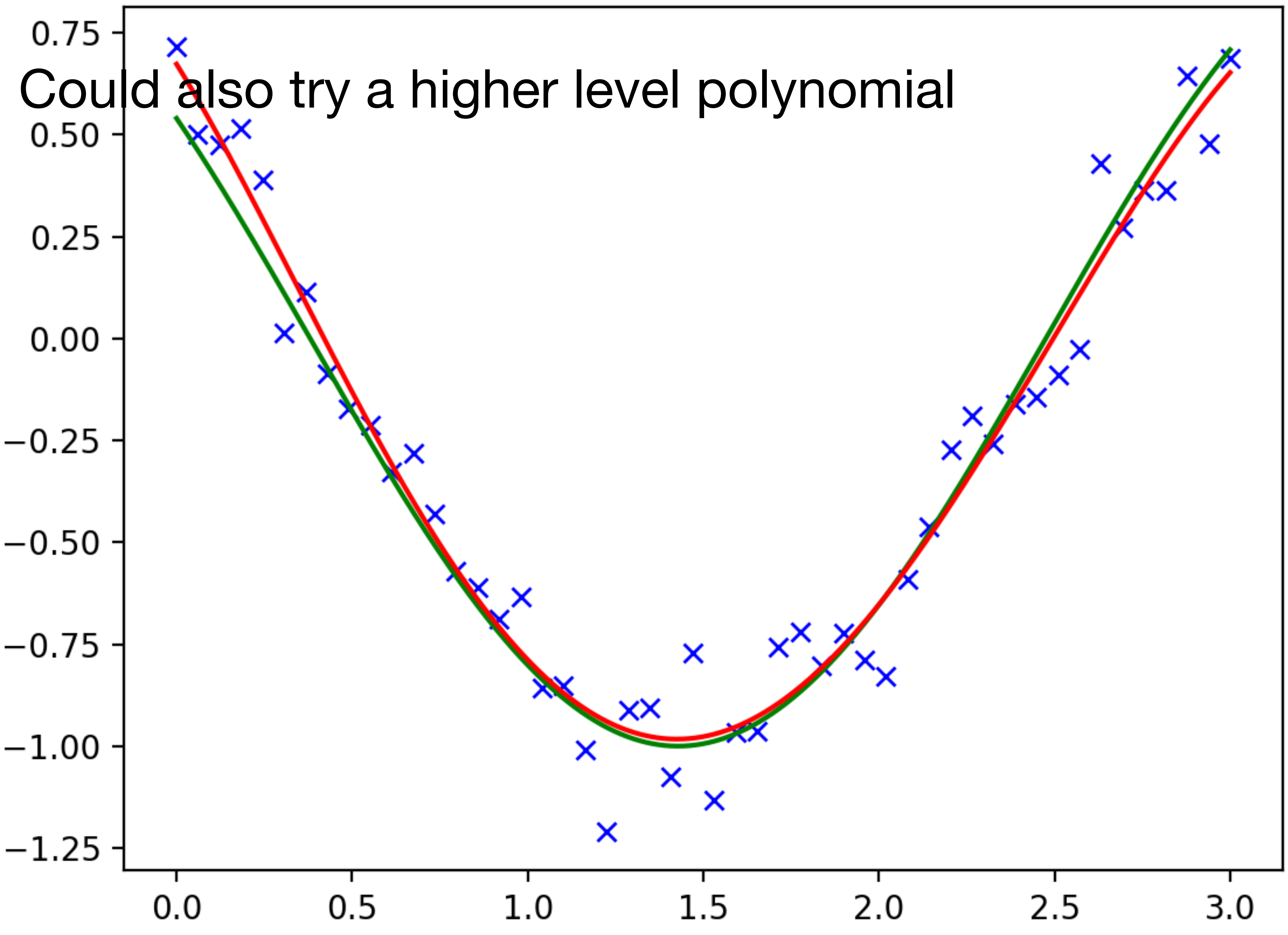


# Fitting with scipy



# Fitting with scipy

- Could also try a higher level polynomial



# Fitting with scipy

- And if we try with a polynomial with as many degrees as there are points, we would get perfect fit
  - And absolutely no insights!

# Optimization

- Global optimizers:
  - Grid search: Start out at a large number of starting positions
  - Try out several methods
    - If possible, calculate the gradient and the Hessian yourself
    - Can use scipy.optimize.check_grad( ) to see whether you calculated correctly

# Curve Fitting

- Need to have a good model:
  - Avoids under- and over-fitting
  - Find a way to measure success
    - E.g. time series: You want to remove trends and have white noise left over

# HELP!!!!

- Get Scipy Lecture notes (for free)
  - www.scipy-lectures.org