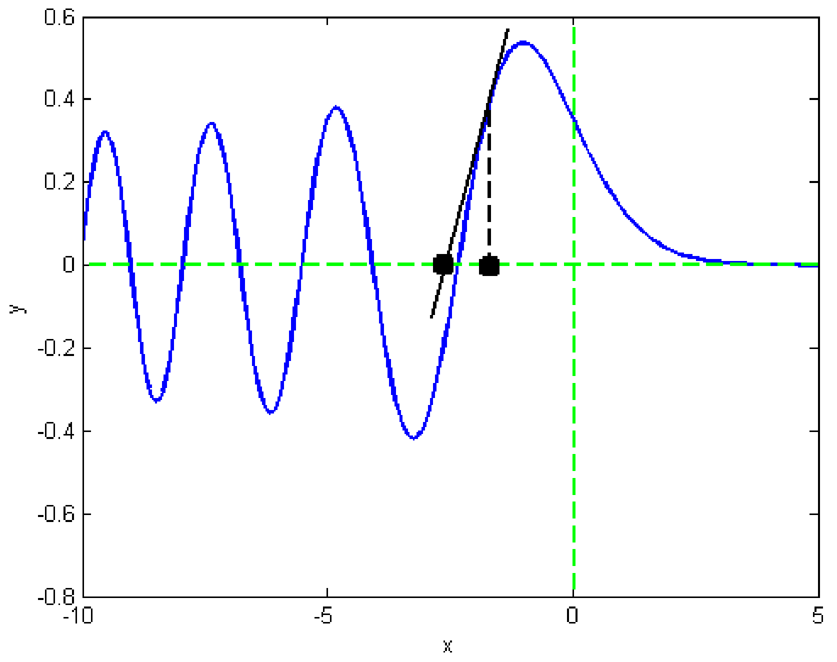
Chapter 4
Rootfinding



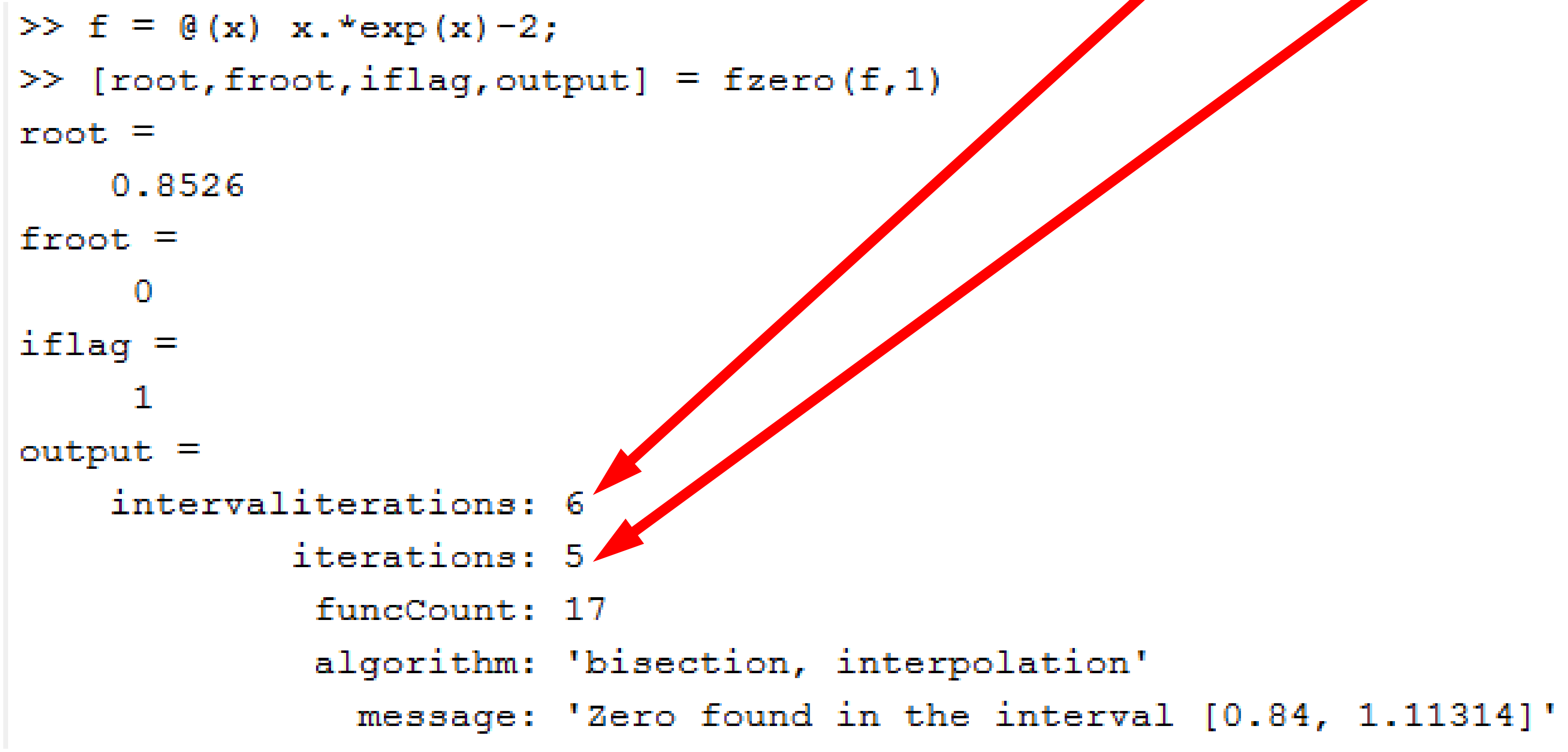
Return to fzero: how it works

- Fzero uses a bracketing method for a few steps
- Method of bisection finds an interval containing the root, and may narrow it
- Then, inverse quadratic interpolation is used to converge to the root
- This has the advantage of a slow but robust method (bisection) getting close to the root so the fast method (IQI) takes over
- Let’s see details of `fzero`…

Return to fzero: how it works

- Use $f(x) = xe^{x} - 2$

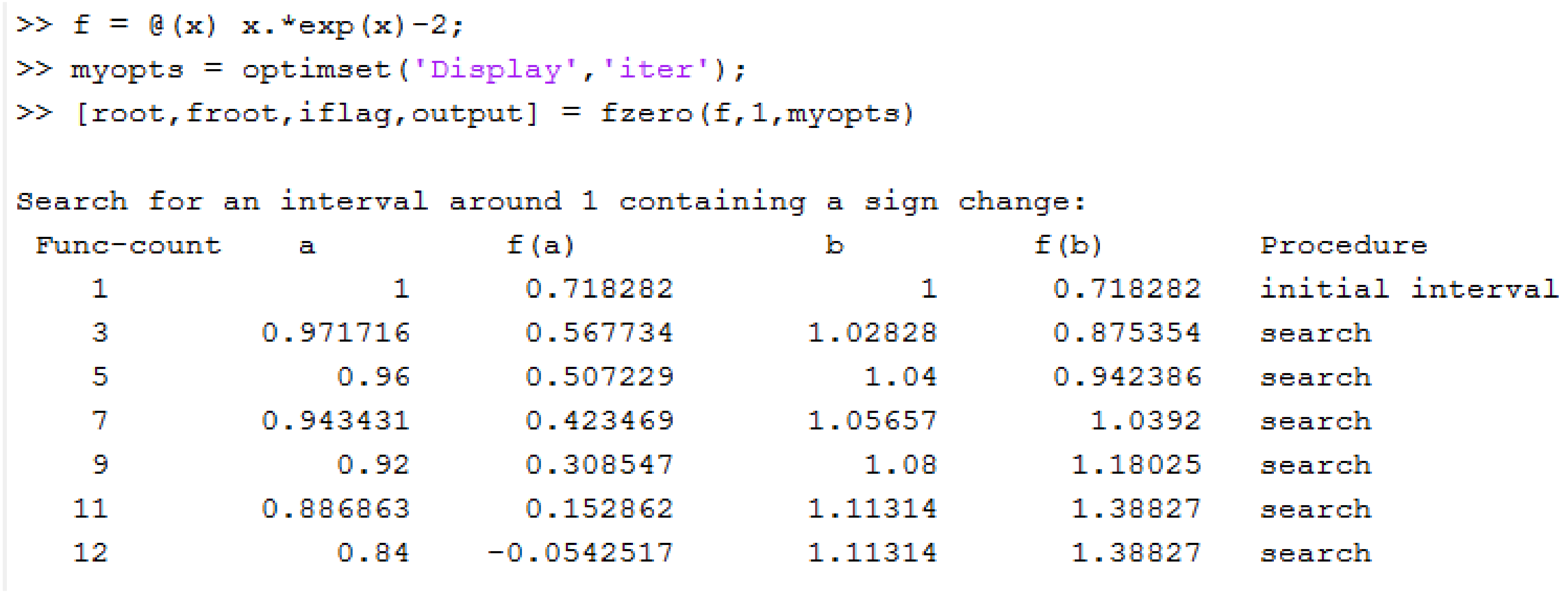
bisection, then IQI



```
>> f = @(x) x.*exp(x)-2;
>> [root,froot,iflag,output] = fzero(f,1)
root =
    0.8526
froot =
     0
iflag =
     1
output =
    intervaliterations: 6
            iterations: 5
             funcCount: 17
             algorithm: 'bisection, interpolation'
               message: 'Zero found in the interval [0.84, 1.11314]'
```

Return to fzero: how it works

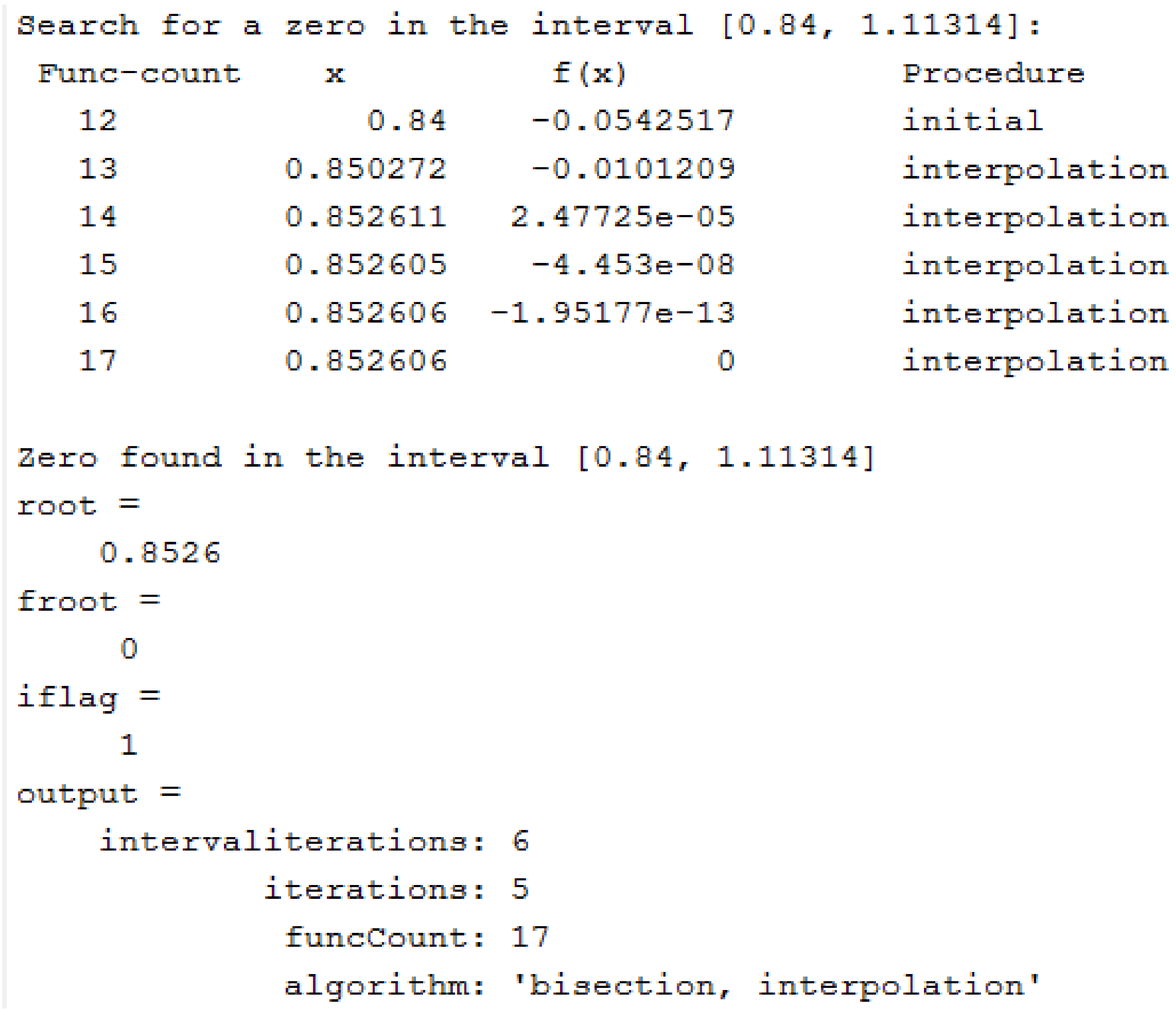
- Now set ‘Display’ to ‘iter’ to see details
- Fzero makes an interval until a sign change found



```
>> f = @(x) x.*exp(x)-2;
>> myopts = optimset('Display','iter');
>> [root,froot,iflag,output] = fzero(f,1,myopts)

Search for an interval around 1 containing a sign change:
 Func-count    a          f(a)             b             f(b)        Procedure
    1               1       0.718282             1       0.718282   initial interval
    3        0.971716       0.567734       1.02828       0.875354   search
    5            0.96       0.507229          1.04       0.942386   search
    7        0.943431       0.423469       1.05657         1.0392   search
    9            0.92       0.308547          1.08        1.18025   search
   11        0.886863       0.152862       1.11314        1.38827   search
   12            0.84     -0.0542517       1.11314        1.38827   search
```

Return to fzero: how it works

- Endpoints where sign change happened plus initial point become three points for starting IQI
- Then iterate using IQI



```
Search for a zero in the interval [0.84, 1.11314]:
 Func-count    x          f(x)             Procedure
   12              0.84    -0.0542517        initial
   13          0.850272    -0.0101209        interpolation
   14          0.852611   2.47725e-05        interpolation
   15          0.852605    -4.453e-08        interpolation
   16          0.852606  -1.95177e-13        interpolation
   17          0.852606             0        interpolation

Zero found in the interval [0.84, 1.11314]
root =
    0.8526
froot =
     0
iflag =
     1
output =
    intervaliterations: 6
            iterations: 5
             funcCount: 17
             algorithm: 'bisection, interpolation'
```

Root finding: Newton’s method

- You have no doubt seen this method somewhere, but we will analyze it in a bit more depth
- We seek $f(p) = 0$ for $x = p$.
- We want to use Taylor’s theorem to linearize the problem near p.
- If we Taylor expand about x near p, we obtain
$$f(p) = f(x) + \frac{f'(x)}{1!}(p - x) + \frac{f''(\xi(p))}{2!}(p - x)^2$$
- The number $\xi(p)$ makes the formula exact.
- To solve the problem approximately, we neglect the quadratic term, which may be expected to work if $|p - x| \ll 1$

Root finding: Newton’s method

- Also use $f(p) = 0$ to obtain

$$0 \approx f(x) + \frac{f'(x)}{1!}(p - x)$$

- This is the equation for a line tangent at x, which crosses the x-axis near p, but not at it (if things work right)
- Solving for p,

$$p \approx x - \frac{f(x)}{f'(x)}$$

- Because we aren’t at the root, we turn this into an iteration:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \qquad n = 0,1, \ldots$$

Newton's method for systems

- We want to turn the previous approach into something for nonlinear systems.
- We want to solve a system of the form for the x_i where
$$f_1(x_1, x_2) = 0, \qquad f_2(x_1, x_2) = 0$$
- We need both f_i to be zero at the same locations $\boldsymbol{x} = \boldsymbol{r}$, with
$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad \boldsymbol{p} = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix}$$
- Let
$$\boldsymbol{F} = \begin{bmatrix} f_1(\boldsymbol{x}) \\ f_2(\boldsymbol{x}) \end{bmatrix} = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix}$$
- At the roots we have
$$\boldsymbol{F} = \begin{bmatrix} f_1(\boldsymbol{p}) \\ f_2(\boldsymbol{p}) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \boldsymbol{0}$$

Newton's method for systems

- Taylor expand through the linear terms for each function
- Expand $\boldsymbol{F}$ about $\boldsymbol{x}^{(0)} = \left[x_1^{(0)} \quad x_2^{(0)}\right]^T$ for $\boldsymbol{F}(\boldsymbol{p})$ with $||\boldsymbol{p} - \boldsymbol{x}^{(0)}|| \ll 1$
- Then

$$f_1(\boldsymbol{p}) = f_1\left(\boldsymbol{x}^{(0)}\right) + f_{1,x_1}\left(\boldsymbol{x}^{(0)}\right)\left(p_1 - x_1^{(0)}\right) + f_{1,x_2}\left(\boldsymbol{x}^{(0)}\right)\left(p_2 - x_2^{(0)}\right)$$

$$f_2(\boldsymbol{p}) \ = f_2(\boldsymbol{x}^{(0)}) + f_{2,x_1}\left(\boldsymbol{x}^{(0)}\right)\left(p_1 - x_1^{(0)}\right) + f_{2,x_2}\left(\boldsymbol{x}^{(0)}\right)\left(p_2 - x_2^{(0)}\right)$$

- We have truncate the expansion, and $f_i(\boldsymbol{p}) = 0$ by definition. Then

$$\boldsymbol{F}(\boldsymbol{p}) = \boldsymbol{0} \ \approx \boldsymbol{F}\left(\boldsymbol{x}^{(0)}\right) + \boldsymbol{J}(\boldsymbol{x}^{(0)})(\boldsymbol{p}-\boldsymbol{x}^{(0)})$$

- Here $\boldsymbol{J}(\boldsymbol{x}^{(0)})$ is the Jacobian matrix

$$\boldsymbol{J}(\boldsymbol{x}^{(0)}) = \begin{bmatrix} f_{1,x_1}\left(\boldsymbol{x}^{(0)}\right) & f_{1,x_2}\left(\boldsymbol{x}^{(0)}\right) \\ f_{2,x_1}\left(\boldsymbol{x}^{(0)}\right) & f_{2,x_2}\left(\boldsymbol{x}^{(0)}\right) \end{bmatrix}$$

Newton’s method for systems

- Here $\boldsymbol{J}(\boldsymbol{x}^{(0)})$ is the Jacobian matrix

$$\boldsymbol{J}(\boldsymbol{x}^{(0)}) = \begin{bmatrix} f_{1,x_1}\left(\boldsymbol{x}^{(0)}\right) & f_{1,x_2}\left(\boldsymbol{x}^{(0)}\right) \\ f_{2,x_1}\left(\boldsymbol{x}^{(0)}\right) & f_{2,x_2}\left(\boldsymbol{x}^{(0)}\right) \end{bmatrix}$$

- The elements are the partial derivatives of the f_i with respect to x_i and evaluated at $\boldsymbol{x}^{(0)}$
- This linearized system only approximates $\boldsymbol{p}$

$$\boldsymbol{0} \approx \boldsymbol{F}\left(\boldsymbol{x}^{(0)}\right) + \boldsymbol{J}(\boldsymbol{x}^{(0)})(\boldsymbol{p}-\boldsymbol{x}^{(0)})$$

- Solving gives

$$\boldsymbol{p} \approx \boldsymbol{x}^{(0)} + \boldsymbol{J}^{-\boldsymbol{1}}(\boldsymbol{x}^{(0)})\boldsymbol{F}\left(\boldsymbol{x}^{(0)}\right)$$

- Analogous with scalar version

$$p \approx x - \frac{f(x)}{f'(x)}$$

Newton’s method for systems

- Turn this into an iteration; theoretically
$$\boldsymbol{x}^{(k+1)} = \boldsymbol{x}^{(k)} + \boldsymbol{J}^{-\boldsymbol{1}}(\boldsymbol{x}^{(k)})\boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right), k = 0,1, \dots$$
- Analogous with scalar version
$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \qquad k = 0,1, \dots$$
- The Jacobian matrix
$$\boldsymbol{J}(\boldsymbol{x}^{(k)}) = \begin{bmatrix} f_{1,x_1}\left(\boldsymbol{x}^{(k)}\right) & f_{1,x_2}\left(\boldsymbol{x}^{(k)}\right) \\ f_{2,x_1}\left(\boldsymbol{x}^{(k)}\right) & f_{2,x_2}\left(\boldsymbol{x}^{(k)}\right) \end{bmatrix}$$
- The function $\boldsymbol{F}$ must also be updated every iteration
$$\boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right) = \begin{bmatrix} f_1(\boldsymbol{x}^{(k)}) \\ f_2(\boldsymbol{x}^{(k)}) \end{bmatrix}$$

Newton's method for systems

- This is a ***theoretical*** iteration; we ***don't compute*** with this:

$$\boldsymbol{x}^{(k+1)}=\boldsymbol{x}^{(k)}+\boldsymbol{J}^{-\mathbf{1}}(\boldsymbol{x}^{(k)})\boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right), k=0,1,\ldots$$

- We instead solve the linear system and then update the iterate.
- Define

$$\boldsymbol{x}^{(k+1)}=\boldsymbol{x}^{(k)}+\Delta\boldsymbol{x}^{(k)}$$

- Then, rewrite the top equation as

$$\boldsymbol{J}\left(\boldsymbol{x}^{(k)}\right)\left(\boldsymbol{x}^{(k+1)}-\boldsymbol{x}^{(k)}\right)=-\boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right)$$

- Or,

$$\boldsymbol{J}\left(\boldsymbol{x}^{(k)}\right)\left(\Delta\boldsymbol{x}^{(k)}\right)=-\boldsymbol{F}\left(\boldsymbol{x}^{(k)}\right)$$

- This system is solved each iteration, then compute the updated iterate from $\boldsymbol{x}^{(k+1)}=\boldsymbol{x}^{(k)}+\Delta\boldsymbol{x}^{(k)}$

Newton's method for systems

- So, the approach is as follows:
- Write the equations as $f_i(x_1, \dots x_n) = 0, \quad i = 1,2,\dots,n.$
- Create

$$\boldsymbol{F}(\boldsymbol{x}_k) = \begin{bmatrix} f_1(\boldsymbol{x}_k) \\ \vdots \\ f_n(\boldsymbol{x}_k) \end{bmatrix}$$

- Solve the system

$$\boldsymbol{J}(\boldsymbol{x}_k)(\Delta\boldsymbol{x}_k) = -\boldsymbol{F}(\boldsymbol{x}_k)$$

- Then compute the updated iterate from

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta\boldsymbol{x}_k$$

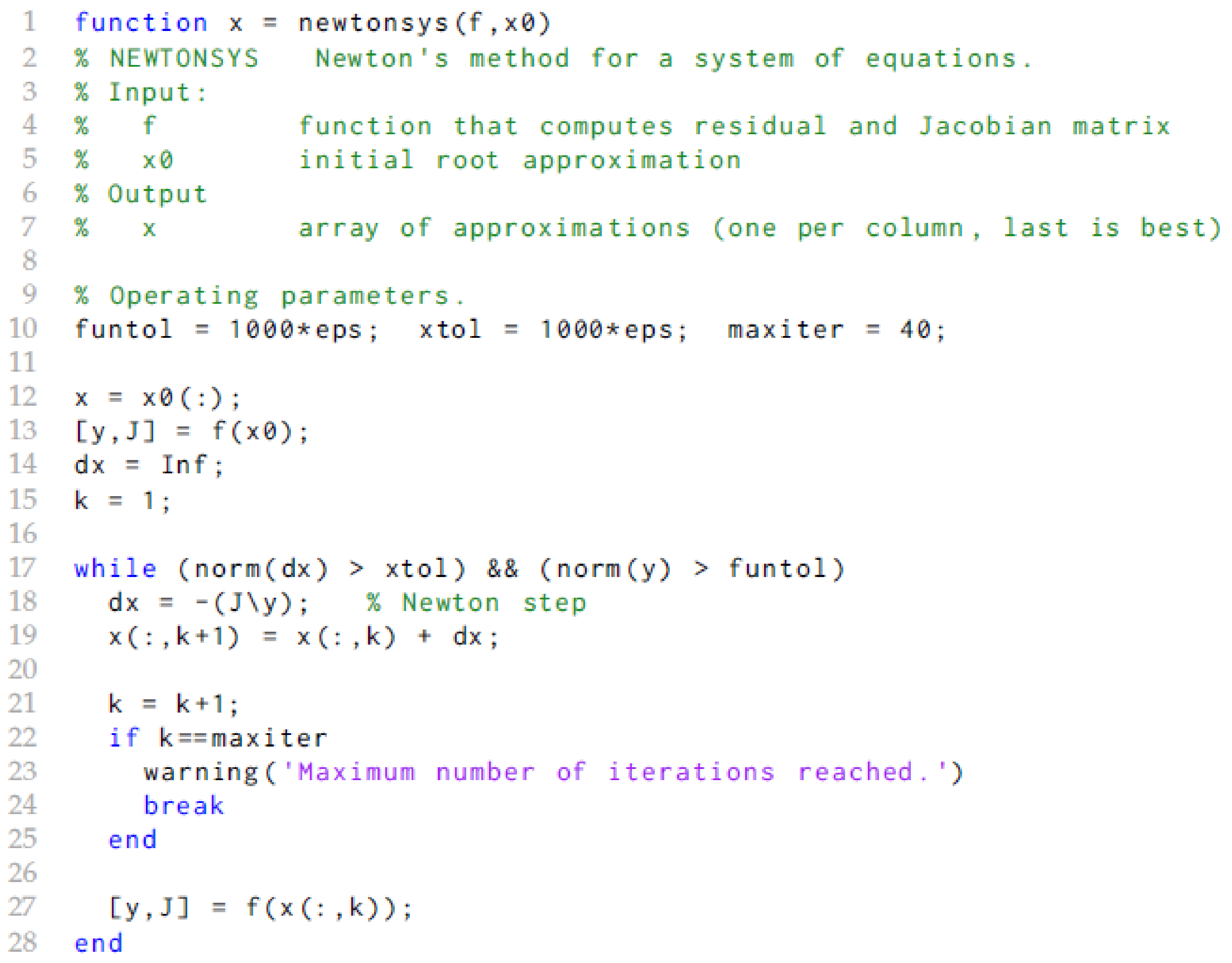
- Repeat until $\|\Delta\boldsymbol{x}_{k+\mathbf{1}}\|$ and $\|\boldsymbol{F}(\boldsymbol{x}_{k+\mathbf{1}})\|$ are small enough

Newton’s method for systems

- Example: 2x2 system

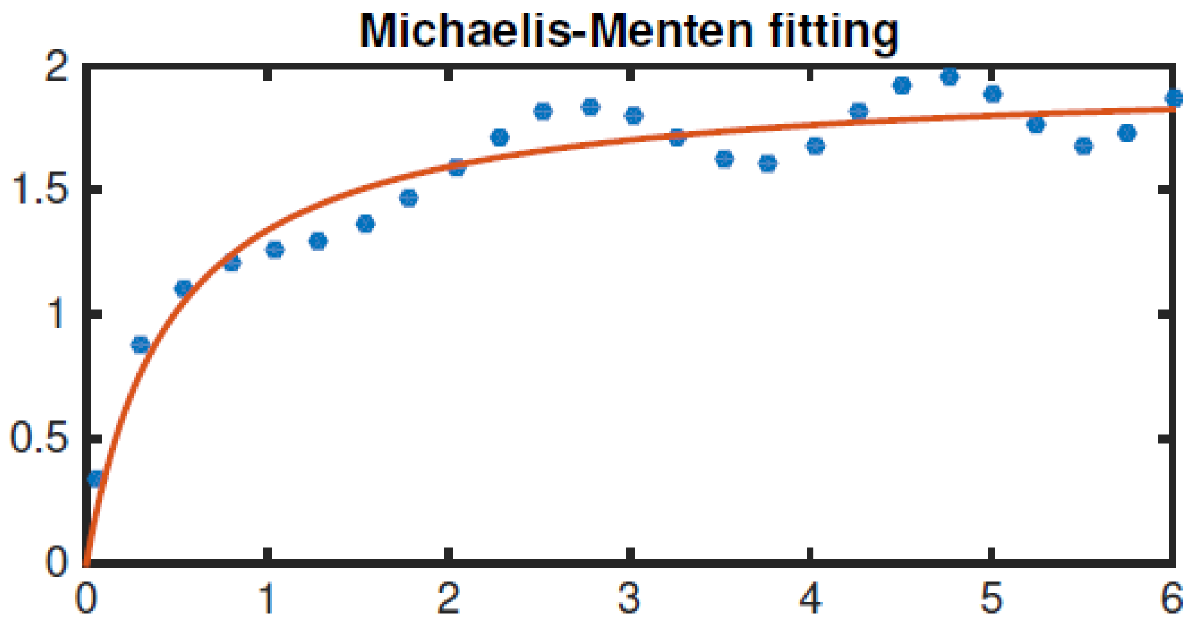
Newton’s method for systems: book code

- For function, we need f, and an initial guess
- F calculates both the function and the Jacobian matrix
- Note bigger tolerances in both $\boldsymbol{dx}$ and $\boldsymbol{f(x)}$ are set to $1000\epsilon_M$ (uses two-norm)
- Iterate in while loop until tolerances aren’t satisfied



```matlab
function x = newtonsys(f,x0)
% NEWTONSYS   Newton's method for a system of equations.
% Input:
%   f       function that computes residual and Jacobian matrix
%   x0      initial root approximation
% Output
%   x       array of approximations (one per column, last is best)

% Operating parameters.
funtol = 1000*eps;  xtol = 1000*eps;  maxiter = 40;

x = x0(:);
[y,J] = f(x0);
dx = Inf;
k = 1;

while (norm(dx) > xtol) && (norm(y) > funtol)
  dx = -(J\y);   % Newton step
  x(:,k+1) = x(:,k) + dx;

  k = k+1;
  if k==maxiter
    warning('Maximum number of iterations reached.')
    break
  end

  [y,J] = f(x(:,k));
end
```

Nonlinear least squares fitting



Nonlinear least squares fitting

- Overdetermined *nonlinear* systems to find fits to m data pts for n parameters, with $m > n$
- Before, we sought linear combos of $f_i(t)$, finding unknown c_i for

$$f(t) = c_1 f_1(t) + \cdots + c_n f_n(t)$$

- But now, we have

$$\boldsymbol{f}(\boldsymbol{t}, \boldsymbol{y}; \boldsymbol{c}) \approx \boldsymbol{0}$$

- Now we don't have a linear function, but we put each of the data $\boldsymbol{t}, \boldsymbol{y} \in \mathbb{R}^m$ and relatively few parameters $\boldsymbol{c} \in \mathbb{R}^n$ into the function f so that we have the vector output $\boldsymbol{f} \in \mathbb{R}^m$

Nonlinear least squares fits

- For convenience think of the problem as finding the parameters $\boldsymbol{x} \in \mathbb{R}^n$ in the vector function $\boldsymbol{f}(\boldsymbol{x}) \in \mathbb{R}^m$ so that we minimize the residual:

$$\text{Find } \mathrm{x} \in \mathbb{R}^n \text{ such that } \|\mathrm{f}(\mathrm{x})\|_2 \text{ is minimized.}$$

- We can also think of this as minimizing $\boldsymbol{f}^{\boldsymbol{T}}(\boldsymbol{x})\boldsymbol{f}(\boldsymbol{x})$
- To solve the problem, we proceed by linearization again
- Define the linearization about $\boldsymbol{x}_k$ as

$$\boldsymbol{q}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}_k) + \boldsymbol{J}(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k)$$

- For convenience, define

$$\boldsymbol{h}_k = \boldsymbol{x} - \boldsymbol{x}_k, \qquad \boldsymbol{f}_k = \boldsymbol{f}(\boldsymbol{x}_k), \qquad \boldsymbol{J}_k = \boldsymbol{J}(\boldsymbol{x}_k)$$

Nonlinear least squares fits

- The problem from the linearized version of $\boldsymbol{f}$ then becomes solving for the iterate update $\boldsymbol{h}$ such that
$$\min_{\boldsymbol{h}} \|\boldsymbol{f}_k + \boldsymbol{J}_k \boldsymbol{h}\|_2$$
- This is minimized if $\boldsymbol{J}_k \boldsymbol{h} = -\boldsymbol{f}_k$
- This linear rectangular system is solved
- Each time we update via $\boldsymbol{x}_{\boldsymbol{k+1}} = \boldsymbol{x}_k + \boldsymbol{h}$
- We then update $\boldsymbol{f}_k$ and $\boldsymbol{J}_k$, and solve the system again
- This is exactly what we did for Newton’s method, and this time it is called Gauss-Newton iteration

Nonlinear least squares fits – Gauss-Newton

- So, here is our recipe:

1. Begin with initial guess
2. Evaluate $\boldsymbol{f}_k$ and $\boldsymbol{J}_k$
3. Solve $\boldsymbol{J}_k\boldsymbol{h} = -\boldsymbol{f}_k$ for $\boldsymbol{h}$, $k = 1,2, \ldots$
4. Each time we update via $\boldsymbol{x}_{\boldsymbol{k+1}} = \boldsymbol{x}_k + \boldsymbol{h}$
5. Repeat previous three steps until $\boldsymbol{h}$ is small enough

- How to solve the linear least squares problem for $\boldsymbol{h}$?

Nonlinear least squares fits – Gauss-Newton

- How to solve the linear least squares problem for $\boldsymbol{h}$?
- We can use Matlab's backslash; it will find the least squares solution automatically
- Because of this, we can use `newtonsys.m`!!!
- It takes $\boldsymbol{f}$ and $\boldsymbol{J}$ as input, and returns the result of iterating on the linearized system!
- We usually have to relax the tolerances since we likely can't make the residual equal zero

Example: Michaelis-Menten kinetics

- In production of proteins in cells, it is often observed that protein production may be very low at low concentration and saturates at high concentrations
- A sigmoidal function is often used to fit this observation is

$$v(x) = \frac{V_{max} x}{K_m + x}.$$

- We want to find the best production rate $v(x)$ that fits the data for different concentrations x
- To do this, we need to find the two parameters $V_{\max}$ and K_m that best fit the given data

Example: Michaelis-Menten kinetics

- The function to be minimized is

$$f(x) = v(x) - y = \frac{V_{max}\, x}{K_m + x} - y$$

- The input data are $(x_i, y_i), i = 1,2, \dots, m$
- Plugging in each data point gives the *i*-th component of $\boldsymbol{f}$
- We also need the Jacobian matrix, which is rectangular and comes from dwrt the parameters V_{max} and K_m (each entry is a column)

$$\boldsymbol{J}(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial V_{max}} & \frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial K_m} \end{bmatrix} = \begin{bmatrix} \frac{\boldsymbol{x}}{K_m + \boldsymbol{x}} & \frac{-V_{\mathrm{max}}\boldsymbol{x}}{(K_m + \boldsymbol{x})^2} \end{bmatrix}$$

Example: Michaelis-Menten kinetics

- We also need some data
- We cook up some here:

$$v_{ideal}(x) = \frac{2\,x}{0.5 + x}$$

- And we add some "noise" that makes the data less than perfect:

$$y(x) = v_{ideal}(x) + 0.15\cos(2xe^{x/16})$$

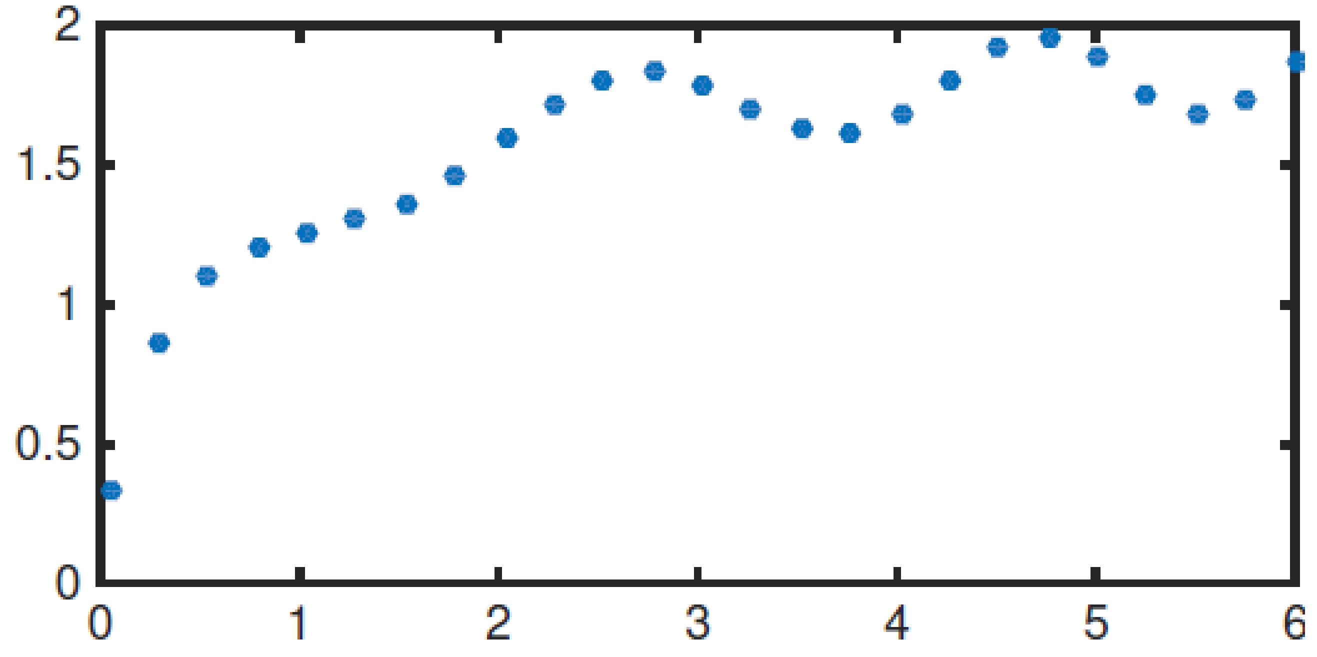
- Now we have the data, and we expect that the fit should be close to $v_{ideal}(x)$
- Let's go to Matlab and solve the problem…

Example: Michaelis-Menten kinetics

- Create the data:

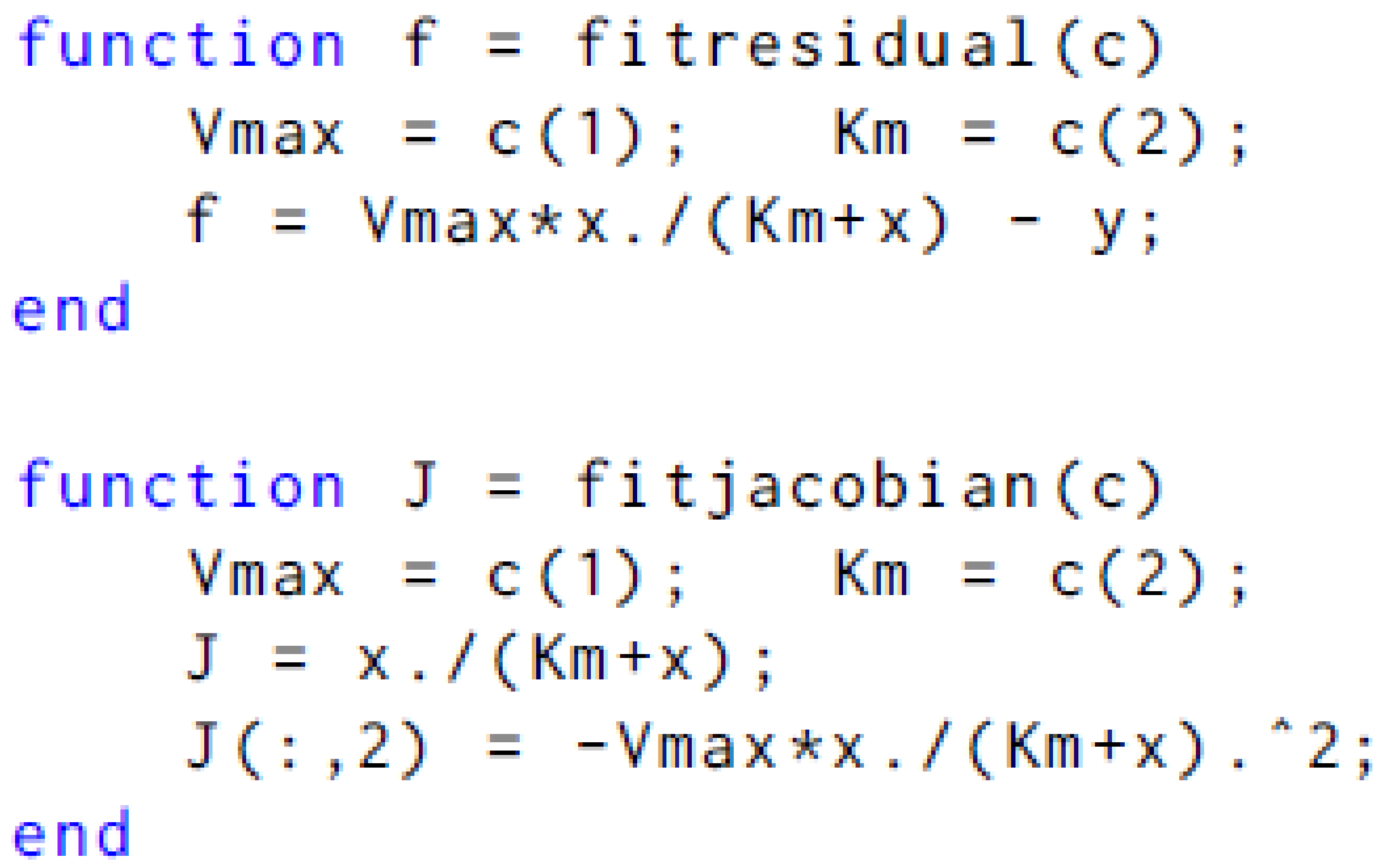
```
m = 25;
x = linspace(0.05,6,m)';
y = 2*x./(0.5+x);
y = y + 0.15*cos(2*exp(x/16).*x);
plot(x,y,'.')
```

- Here's how it looks:



Example: Michaelis-Menten kinetics

- Create the functions to evaluate $\boldsymbol{f}$ and $\boldsymbol{J}$:
- Note how the derivatives are with respect to the desired parameters
- Each expression in $\boldsymbol{J}$ is a column here



```
function f = fitresidual(c)
    Vmax = c(1);   Km = c(2);
    f = Vmax*x./(Km+x) - y;
end

function J = fitjacobian(c)
    Vmax = c(1);   Km = c(2);
    J = x./(Km+x);
    J(:,2) = -Vmax*x./(Km+x).^2;
end
```

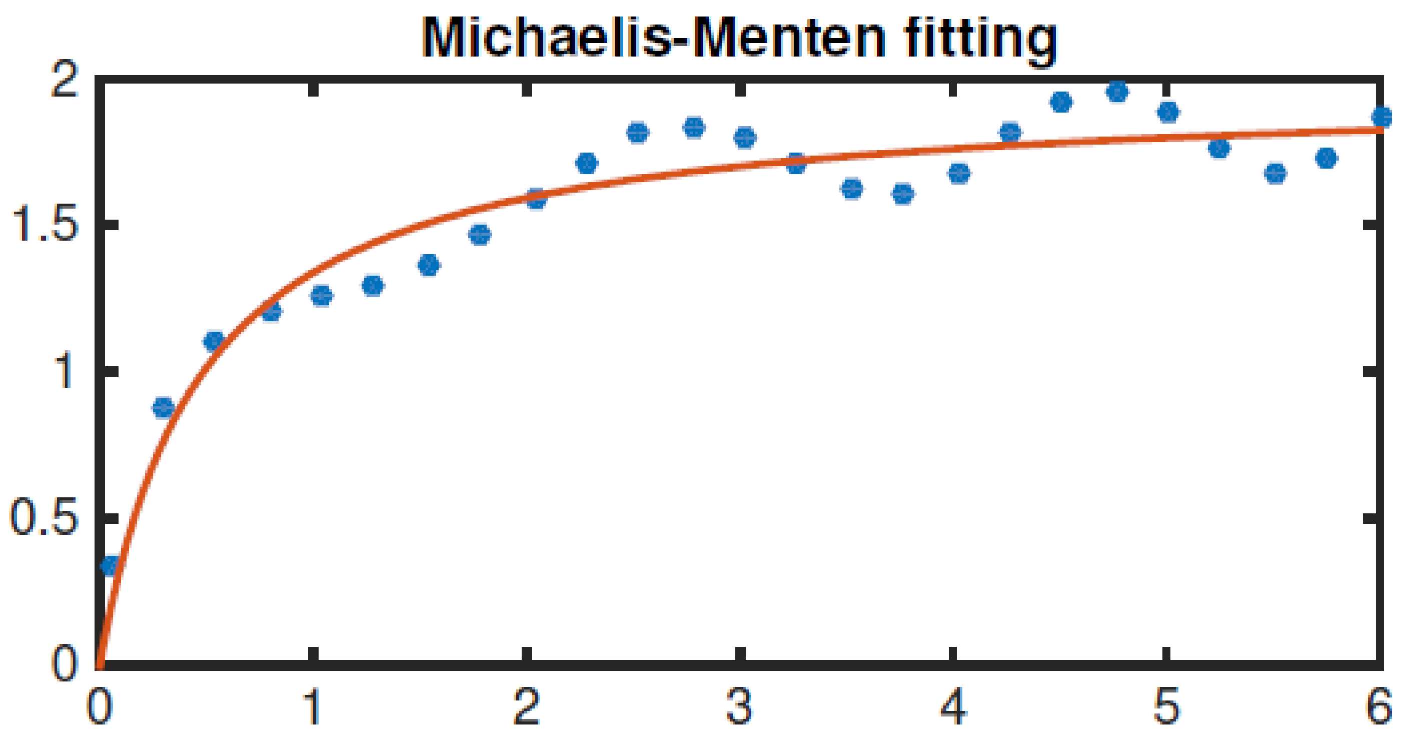
Michaelis-Menten ex

```
c0 = [1; 0.75];
c = newtonsys(@fitresidual,@fitjacobian,c0);
c(:,1:3:end)
Vmax = c(1,end);   Km = c(2,end);
```

- Solve the problem:

```
ans =
     1.0000    1.9134    1.9685    1.9687    1.9687    1.9687
     0.7500    0.3751    0.4691    0.4693    0.4693    0.4693
```

- Here's how it looks:



- The last values of the parameters are close to the (2,0.5) we started with

Example: Michaelis-Menten kinetics

- We can also use a linearized fit function to see how it does.
- The fit is then of the form:

$$\frac{1}{v(x)} = \frac{a}{x} + b$$

- We now need to find $a = K_m/V_{max}$ and $b = 1/V_{max}$
- The data are $(x_i, 1/y_i), i = 1,2,\ldots,m$
- The columns of the Vandermonde matrix are

$$A = [1./\boldsymbol{x} \quad \mathbf{1}]$$

- Let's go to Matlab and solve the new problem…

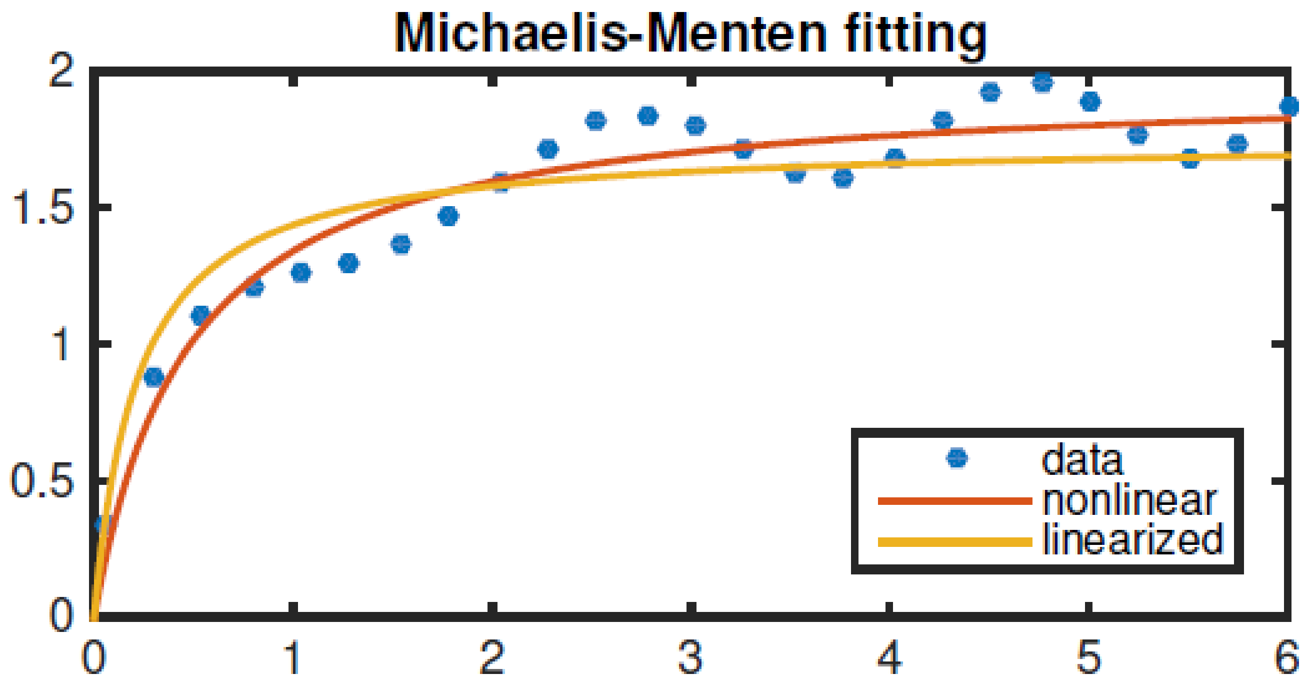
Michaelis-Menten ex

- Solve the problem:

```
c = [ x.^(-1), x.^0 ] \ (1./y);
a = c(1);   b = c(2);
```

```
fplot(@(x) 1./((a./x)+b), [0 6] )
legend('data','nonlinear','linearized','location','southeast')
```

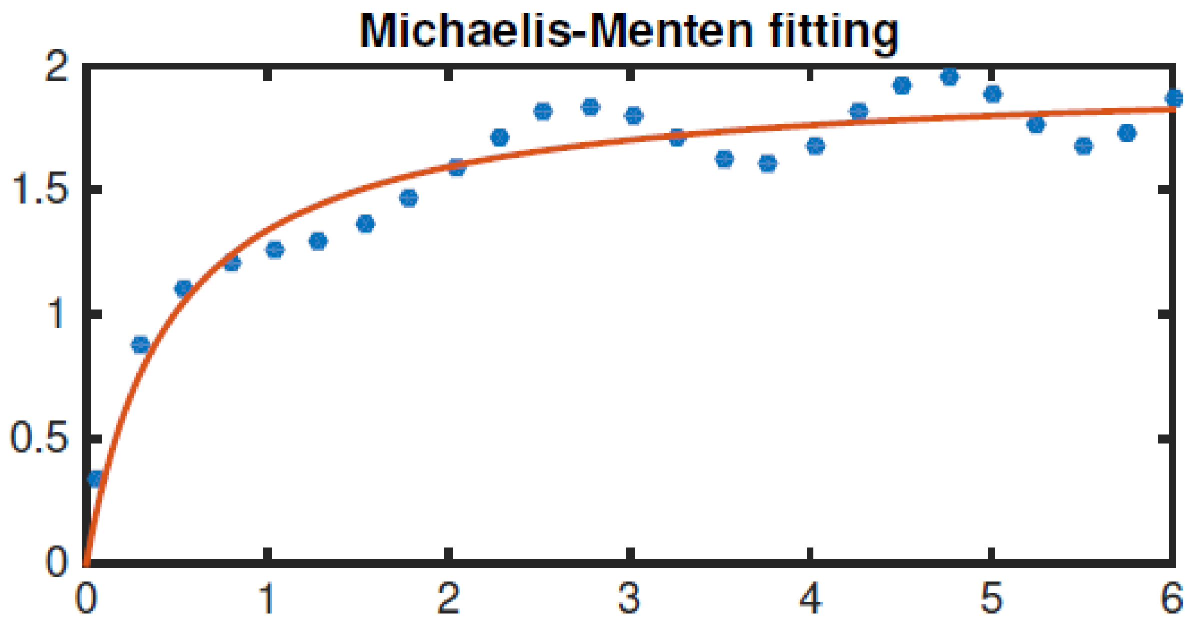
- Here’s how it looks:
- Fit is worse than for NL version
- Residual worse, now 0.7487 (last time 0.5234)



Example: Michaelis-Menten kinetics

- The linearized fit is worse than the nonlinear fit
- The linearized fit minimizes a different quantity than the nonlinear version
- Result is less faithful to original data than nonlinear version
- This is a classic example of this issue, but it is very common.
- Let’s try it out.

Quasi-Newton Methods for Systems



Quasi-Newton methods

- In many problems, it can be difficult to implement an exact Jacobian matrix for the problem.
- We want to find an approach like the secant method where we don't need the derivatives
- For the secant method, we replaced $f'(x_k)$ with

$$\frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}.$$

- In the limit as the denominator tends to zero, we get the derivative by definition, but we don't take the limit in numerical methods
- How to use this for the Jacobian?

Quasi-Newton methods

- $\boldsymbol{J}(\boldsymbol{x}^{(0)})$ is the Jacobian matrix,

$$\boldsymbol{J}(\boldsymbol{x}^{(0)}) = \begin{bmatrix} f_{1,x_1}(\boldsymbol{x}^{(0)}) & f_{1,x_2}(\boldsymbol{x}^{(0)}) \\ f_{2,x_1}(\boldsymbol{x}^{(0)}) & f_{2,x_2}(\boldsymbol{x}^{(0)}) \end{bmatrix}$$

in the 2 by 2 case

- More generally, we can write one column of the Jacobian as at right ($\boldsymbol{e}_j$ is the *j*-th column of $I_{n\times n}$)

$$\mathbf{J}(\mathbf{x})\mathbf{e}_j = \begin{bmatrix} \frac{\partial f_1}{x_j}(\mathbf{x}) \\ \frac{\partial f_2}{x_j}(\mathbf{x}) \\ \vdots \\ \frac{\partial f_m}{x_j}(\mathbf{x}) \end{bmatrix}$$

- We can use the finite difference approximation for each element in the Jacobian
- The columns are:

$$\mathbf{J}(\mathbf{x})\mathbf{e}_j \approx \frac{\mathbf{f}(\mathbf{x}+\delta\mathbf{e}_j) - \mathbf{f}(\mathbf{x})}{\delta}, \qquad j = 1, \ldots, n.$$

Quasi-Newton methods

- The approximate Jacobian

$$\mathrm{J}(\mathrm{x})\mathrm{e}_j \approx \frac{\mathrm{f}(\mathrm{x}+\delta \mathrm{e}_j)-\mathrm{f}(\mathrm{x})}{\delta}, \qquad j=1,\ldots,n.$$

- We have the function values $\boldsymbol{f}(\boldsymbol{x})$ already, but we need to evaluate the first term $\boldsymbol{f}(\boldsymbol{x}+\delta \boldsymbol{e}_j)$ to get the Jacobian
- If we expect a noise level of ϵ, then pick $\delta = \sqrt{\epsilon}$
- If only roundoff is around to pollute the computation, then $\delta = \sqrt{\epsilon_M}$ where ϵ_M=eps
- Call approximated Jacobian $\tilde{\boldsymbol{J}}(\boldsymbol{x})$

Quasi-Newton methods

- We could use the Newton method for system with the approximate Jacobian
- Solve the system
$$\tilde{\boldsymbol{J}}(\boldsymbol{x}_k)(\Delta\boldsymbol{x}_k) = -\boldsymbol{F}(\boldsymbol{x}_k)$$
- Then compute the updated iterate from
$$\boldsymbol{x}_{k+1}=\boldsymbol{x}_k + \Delta\boldsymbol{x}_k$$
- Repeat until $\|\Delta\boldsymbol{x}_{k+\mathbf{1}}\|$ and $\|\boldsymbol{F}(\boldsymbol{x}_{k+\mathbf{1}})\|$ are small enough
- But there are additional factors to consider
- Sometimes (often?) the Newton's method gets more sensitive and additional fixes needed

Quasi-Newton methods

- One fix is using *damped iteration* or *line search*
- Instead of the Newton update

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta \boldsymbol{x}_k$$

- We can use

$$\boldsymbol{p}(t) = \boldsymbol{x}_k + t\Delta \boldsymbol{x}_k$$

- For damped Newton iteration, it is often the case that a fixed 0<t<1 is used in the update, and the result is called $\boldsymbol{x}_{k+1}$
- We can also vary t and find a value that results in

$$\left\|\mathbf{f}(\mathbf{p}(t))\right\| < \left\|\mathbf{f}(\mathbf{x}_k)\right\|$$

- This is line search

Quasi-Newton methods

- We can use

$$\boldsymbol{p}(t)=\boldsymbol{x}_k + t\Delta\boldsymbol{x}_k$$

- Use $t = 1$ if it results in $\|\mathbf{f}(\mathbf{p}(t))\| < \|\mathbf{f}(\mathbf{x}_k)\|$
- If it the norm of the residual doesn’t decrease, cut t in half and check again.
- Repeat until $\|\mathbf{f}(\mathbf{p}(t))\| < \|\mathbf{f}(\mathbf{x}_k)\|$ is satisfied or until t is too small and the method fails

Quasi-Newton: Levenberg’s method

- It may also be advantageous to adjust the iteration process by modifying the linear solves and update direction
- This was the Newton’s method equation for the update:
$$\tilde{\boldsymbol{J}}(\boldsymbol{x}_k)(\Delta\boldsymbol{x}_k) = -\boldsymbol{F}(\boldsymbol{x}_k)$$
- A different way to try decrease $\|\boldsymbol{f}(\boldsymbol{x})\|_2$ is to use steepest descent
- In this approach , consider $r = (\|\boldsymbol{f}(\boldsymbol{x})\|_2)^2 = \boldsymbol{f}^{\mathrm{T}}\boldsymbol{f}$
- Recall that from calculus, that the negative of the gradient is the direction of steepest descent of a function
- Then, $\nabla r(\boldsymbol{x}) = \nabla\left(\boldsymbol{f}^{\mathrm{T}}(\boldsymbol{x})\boldsymbol{f}(\boldsymbol{x})\right) = 2\boldsymbol{J}^{T}(\boldsymbol{x})\boldsymbol{f}(\boldsymbol{x})$
- We could try to figure out a step from $\boldsymbol{v}(\boldsymbol{x}) = -s\boldsymbol{J}^{T}(\boldsymbol{x})\boldsymbol{f}(\boldsymbol{x})$ where s is a scalar that must be found (like t in line search)

Quasi-Newton: Levenberg's method

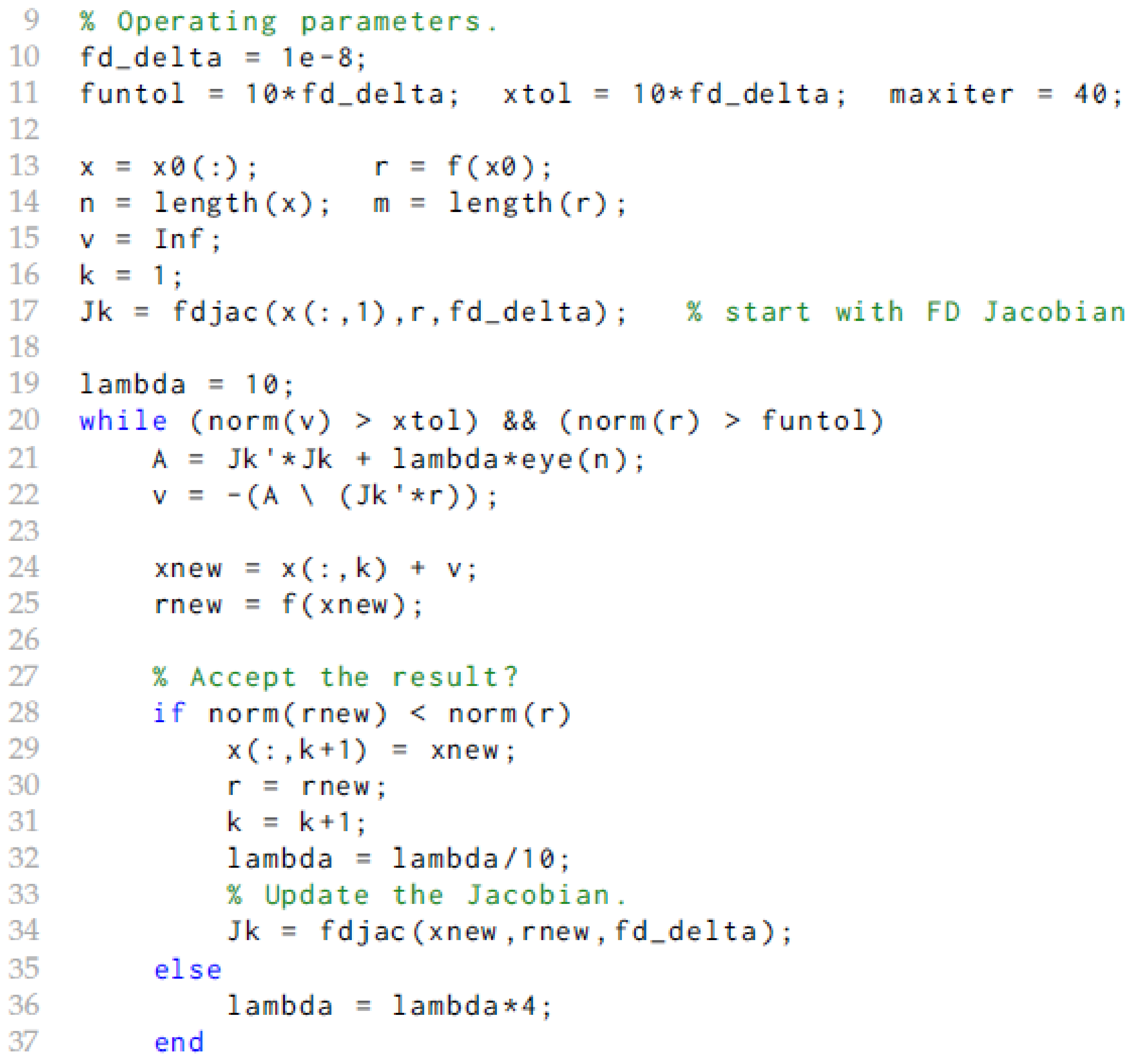
- The steepest descent method can slow down depending on the direction's relation to the minimum
- Newton's method may be difficult to start but converges quickly near the answer
- Get the best of both by combining them: Levenberg
- In this method, one solves:

$$(\boldsymbol{J}^T\boldsymbol{J} + \lambda\boldsymbol{I})\boldsymbol{v} = -\boldsymbol{J}^T\boldsymbol{f}$$

- For $\lambda = 0$, we get back to Newton's method
- For $\lambda \to \infty$, we get close to steepest descent
- We implement a method that varies λ so that it starts large and gets reduced as needed

Levenberg’s method

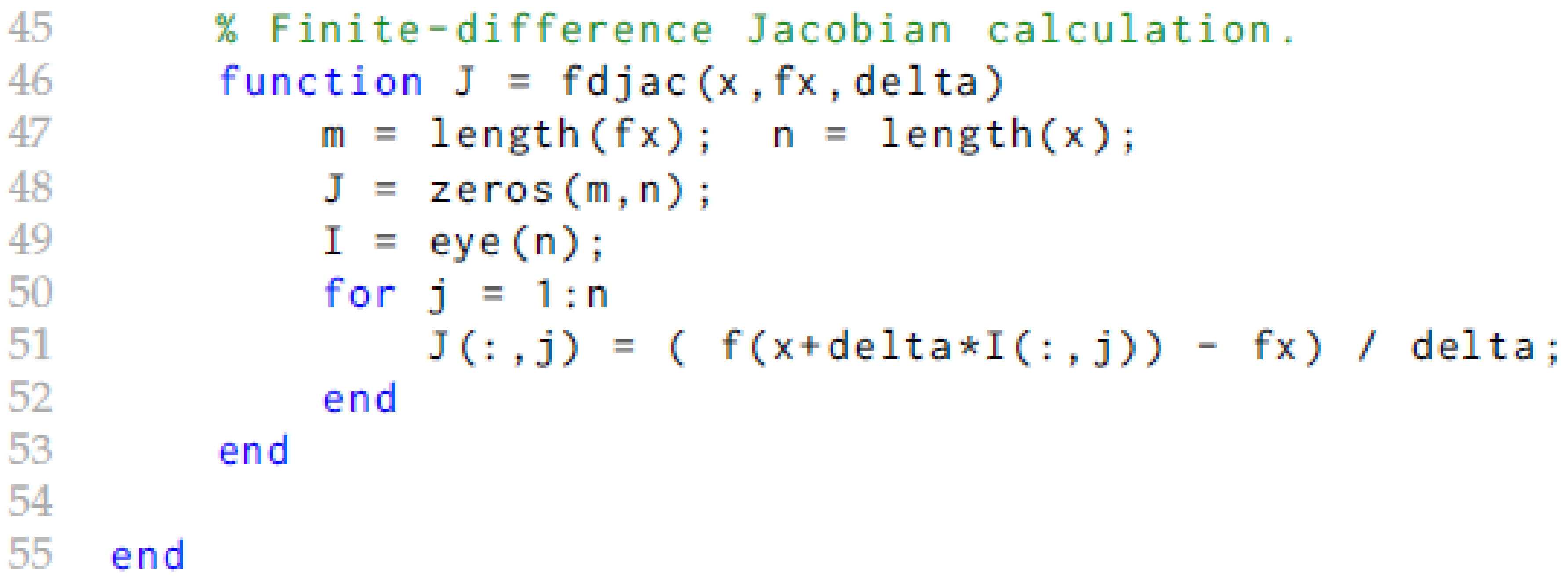
- Start with $\lambda = 10$
- If it works, cut it (more like Newton)
- If it fails, make it bigger (more like steepest descent)
- This code uses approximate Jacobian…



```
% Operating parameters.
fd_delta = 1e-8;
funtol = 10*fd_delta;  xtol = 10*fd_delta;  maxiter = 40;

x = x0(:);      r = f(x0);
n = length(x);  m = length(r);
v = Inf;
k = 1;
Jk = fdjac(x(:,1),r,fd_delta);   % start with FD Jacobian

lambda = 10;
while (norm(v) > xtol) && (norm(r) > funtol)
    A = Jk'*Jk + lambda*eye(n);
    v = -(A \ (Jk'*r));

    xnew = x(:,k) + v;
    rnew = f(xnew);

    % Accept the result?
    if norm(rnew) < norm(r)
        x(:,k+1) = xnew;
        r = rnew;
        k = k+1;
        lambda = lambda/10;
        % Update the Jacobian.
        Jk = fdjac(xnew,rnew,fd_delta);
    else
        lambda = lambda*4;
    end
```

Levenberg’s method

- This code uses approximate Jacobian…



```
    % Finite-difference Jacobian calculation.
    function J = fdjac(x,fx,delta)
        m = length(fx);  n = length(x);
        J = zeros(m,n);
        I = eye(n);
        for j = 1:n
            J(:,j) = ( f(x+delta*I(:,j)) - fx) / delta;
        end
    end

end
```

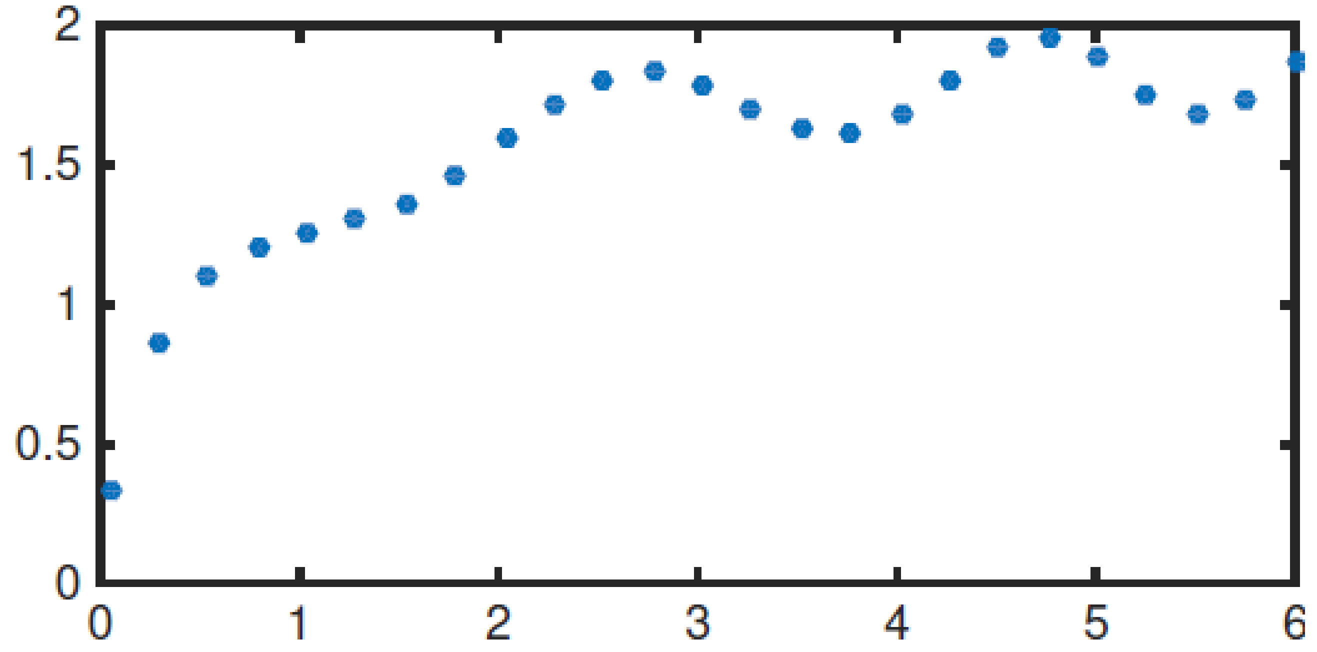
- Let’s do an example

Example: Michaelis-Menten kinetics

- Create the data:

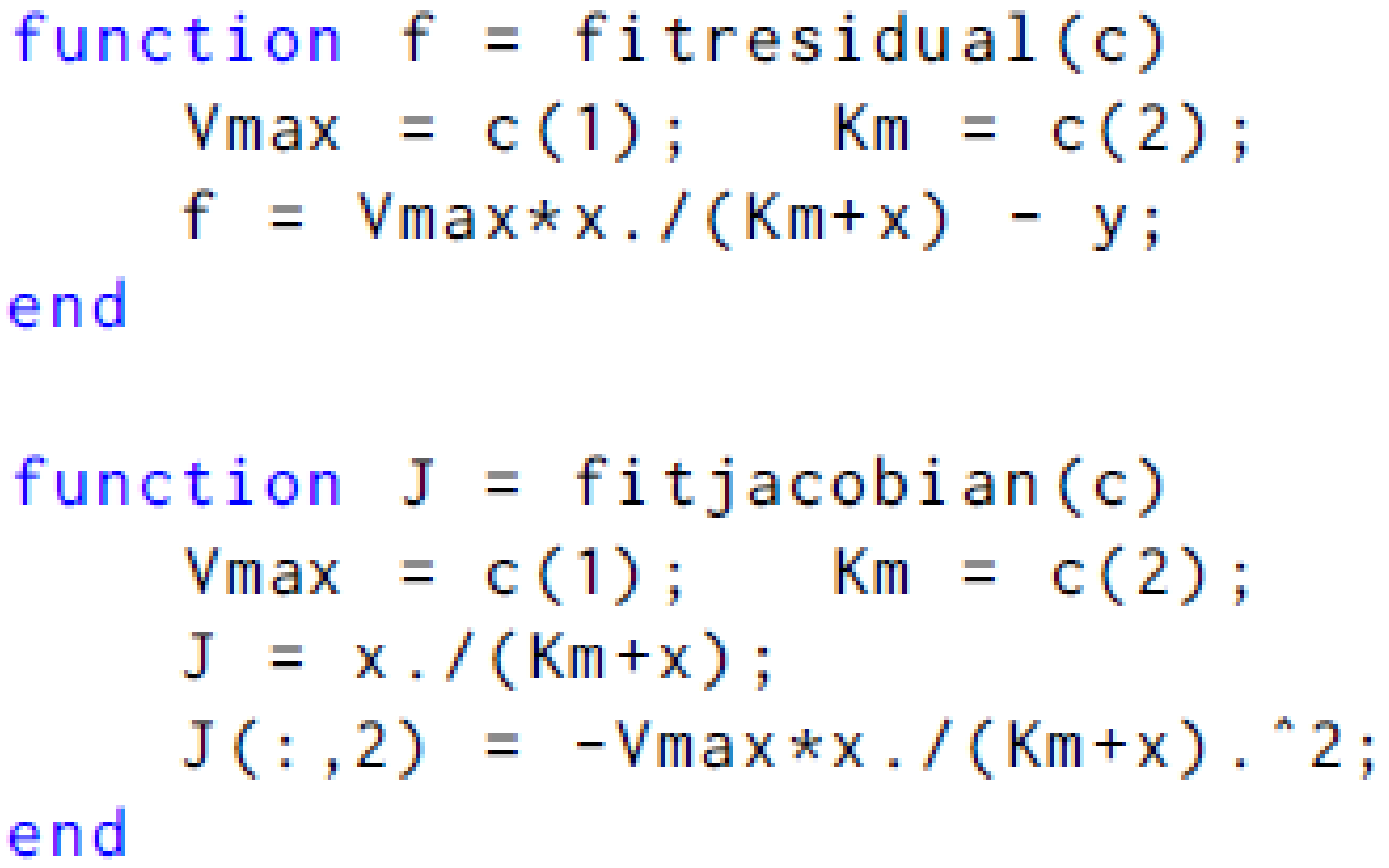
```
m = 25;
x = linspace(0.05,6,m)';
y = 2*x./(0.5+x);
y = y + 0.15*cos(2*exp(x/16).*x);
plot(x,y,'.')
```

- Here's how it looks:



Example: Michaelis-Menten kinetics

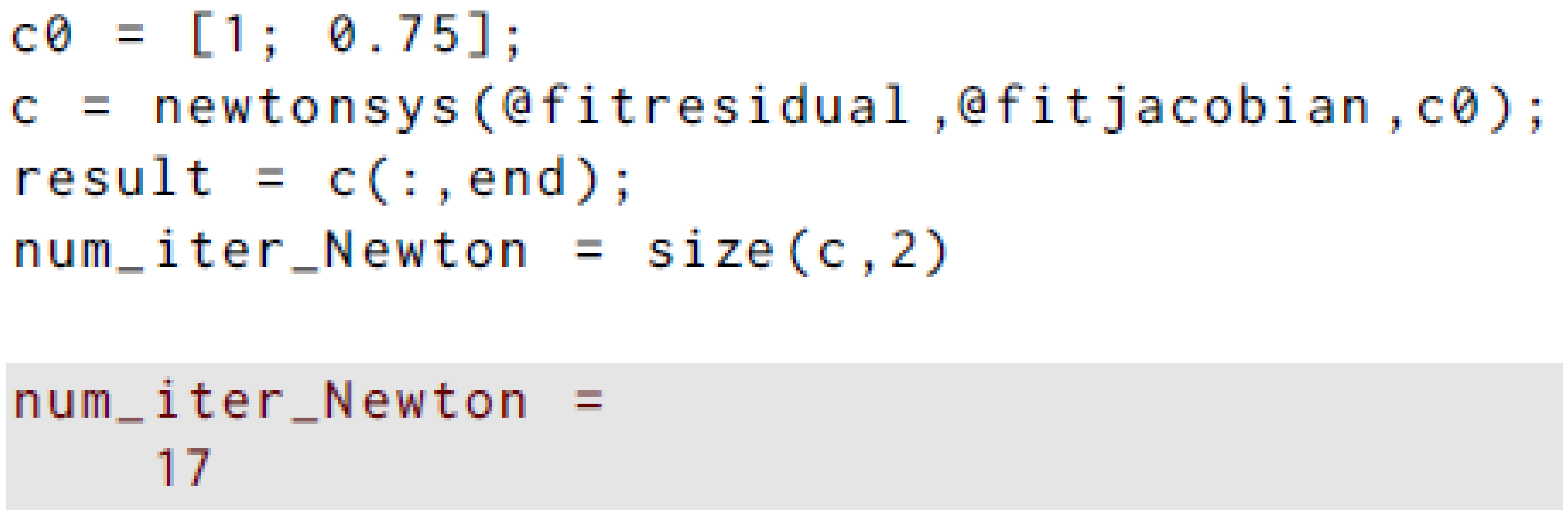
- Create the functions to evaluate $\boldsymbol{f}$ and $\boldsymbol{J}$:
- Note how the derivatives are with respect to the desired parameters
- Each expression in $\boldsymbol{J}$ is a column here



```
function f = fitresidual(c)
    Vmax = c(1);   Km = c(2);
    f = Vmax*x./(Km+x) - y;
end

function J = fitjacobian(c)
    Vmax = c(1);   Km = c(2);
    J = x./(Km+x);
    J(:,2) = -Vmax*x./(Km+x).^2;
end
```

Michaelis-Menten ex

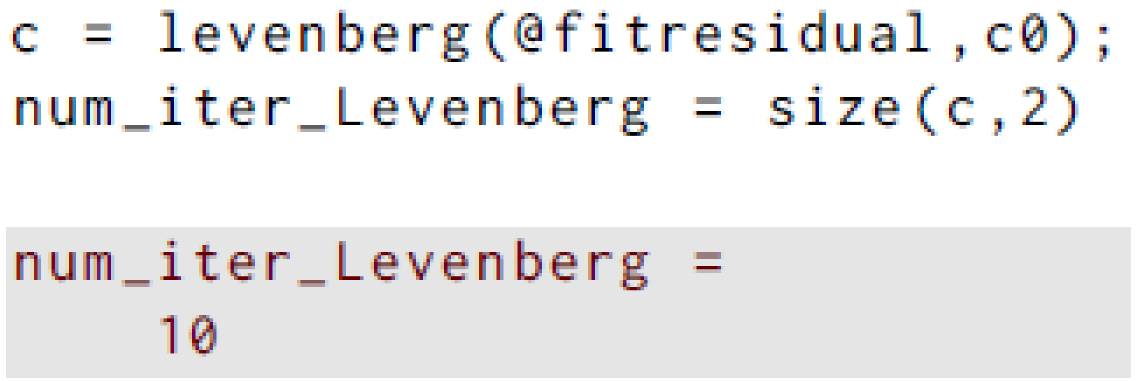
- Solve the problem and note the iterations req’d:



```
c0 = [1; 0.75];
c = newtonsys(@fitresidual,@fitjacobian,c0);
result = c(:,end);
num_iter_Newton = size(c,2)
```

```
num_iter_Newton =
    17
```

- Using Levenberg’s method only takes 10 iterations

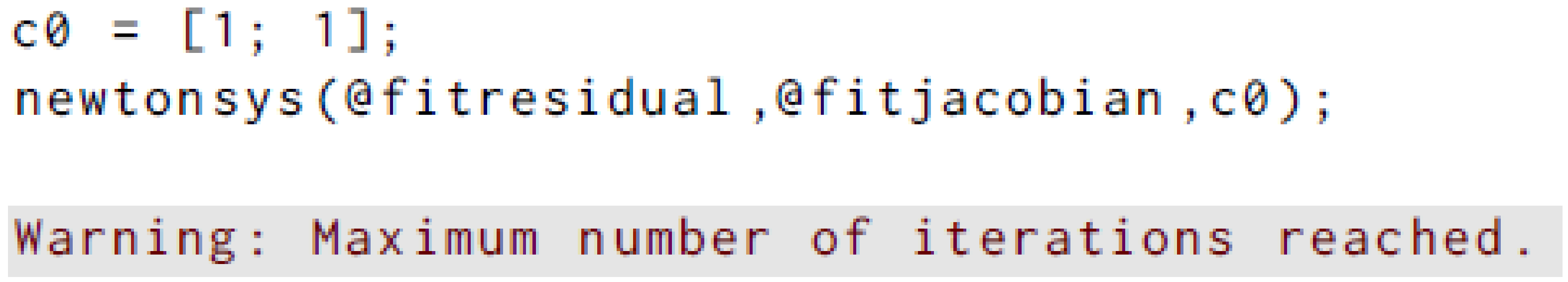


```
c = levenberg(@fitresidual,c0);
num_iter_Levenberg = size(c,2)
```

```
num_iter_Levenberg =
    10
```

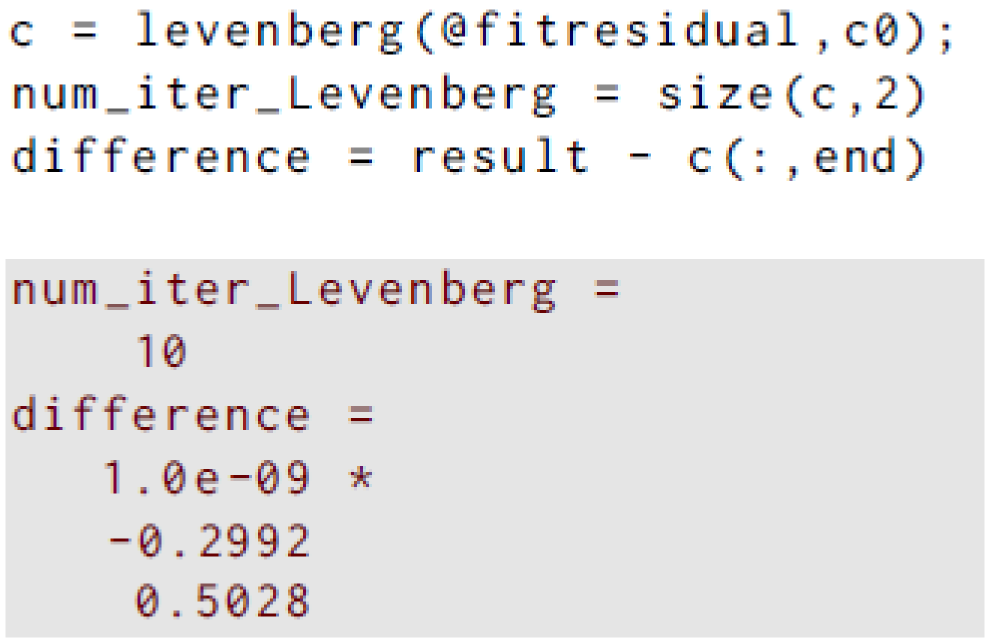
Michaelis-Menten ex

- Solve the problem with a new guess; this time it fails:



```
c0 = [1; 1];
newtonsys(@fitresidual,@fitjacobian,c0);

Warning: Maximum number of iterations reached.
```

- Using Levenberg’s method is barely affected, with about 1e-9 difference in the answer:



```
c = levenberg(@fitresidual,c0);
num_iter_Levenberg = size(c,2)
difference = result - c(:,end)

num_iter_Levenberg =
    10
difference =
   1.0e-09 *
   -0.2992
    0.5028
```

Quasi-Newton methods

- Levenberg’s method (pub’d 1944) was rediscovered and improved a little bit by Donald Marquardt in 1963 while working at DuPont (Wikipdedia names three others that rediscovered it in ‘58 to ‘60)
- Levenberg-Marquardt method:

$$(\boldsymbol{J}^T\boldsymbol{J} + \lambda\mathrm{diag}(\boldsymbol{J}^T\boldsymbol{J}))\boldsymbol{v} = -\boldsymbol{J}^T\boldsymbol{f}$$

- The improvement works a bit better at large λ for some problems
- We implement a method that varies λ so that it starts large and gets reduced as needed
- This method can be chosen as an option in lsqnonlin in Matlab (optimization toolbox)