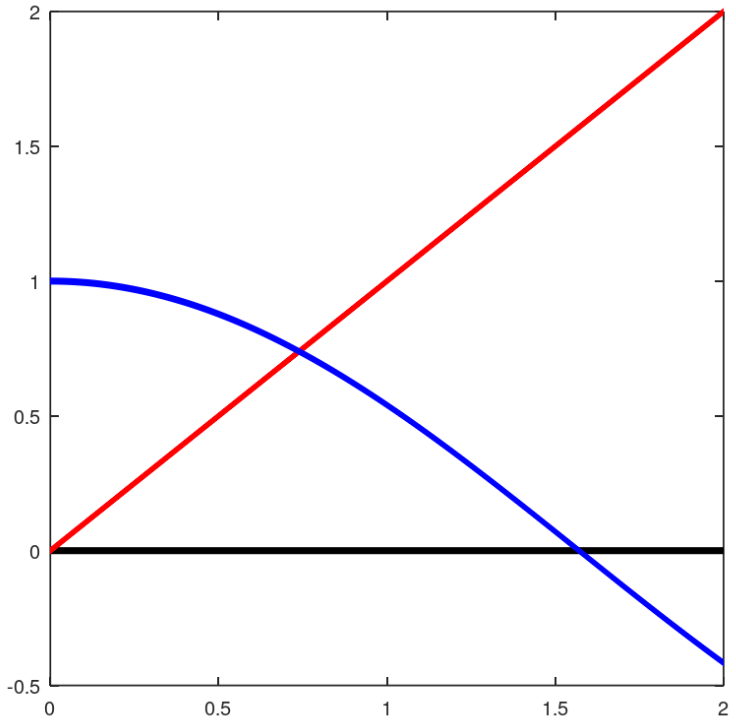
Nonlinear Equations

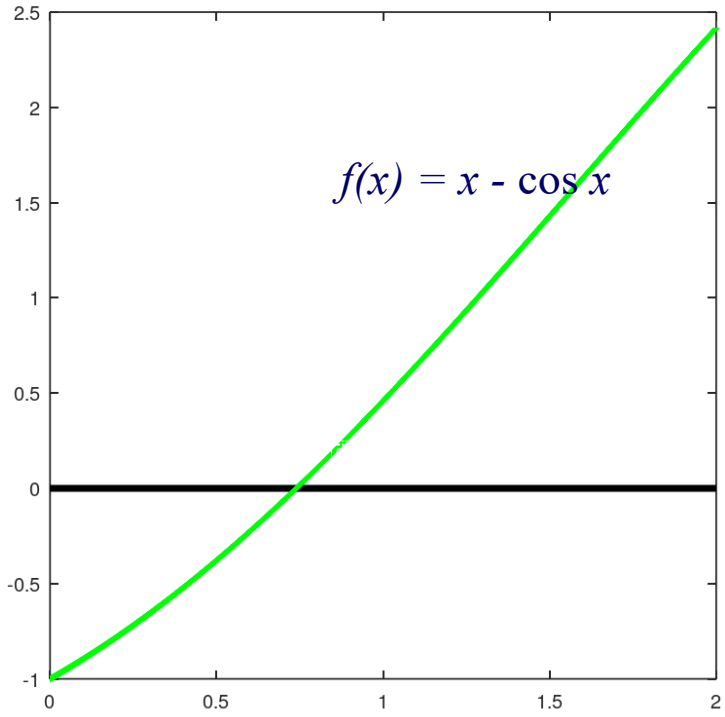
Outline:

- Nonlinear Equations
- Numerical Methods in One Dimension
- Methods for Systems of Nonlinear Equations

Nonlinear Equation Example

- Is there an x where $x = \cos(x)$?
  (Graphing is usually a good idea, if possible.)





- Rewrite as $f(x) = x - \cos x$ and find x^* such that $f(x^*) = 0$.

Nonlinear Equations

- Given function f, we seek value x (sometimes x^*) for which

$$f(x) = 0$$

- Solution is *root* of equation, or *zero* of f

- So problem is known as *root finding* or *zero finding*

Nonlinear Equations

- Single nonlinear equation in one unknown, where

$$f : \mathbb{R} \longrightarrow \mathbb{R}$$

  Solution is scalar x for which $f(x) = 0$

- System of n *coupled* nonlinear equations in n unknowns, where in one unknown, where

$$\mathbf{f} : \mathbb{R}^n \longrightarrow \mathbb{R}^n$$

  Solution is vector $\mathbf{x}$ for which all components are zero *simultaneously*, $\mathbf{f}(\mathbf{x}) = \mathbf{0}$

Existence and Uniqueness

- Existence and uniqueness of solutions are more complicated for nonlinear equations than for linear equations

- For function $f : \mathbb{R} \longrightarrow \mathbb{R}$, *bracket* is interval $[a, b]$ for which sign of f differs at endpoints

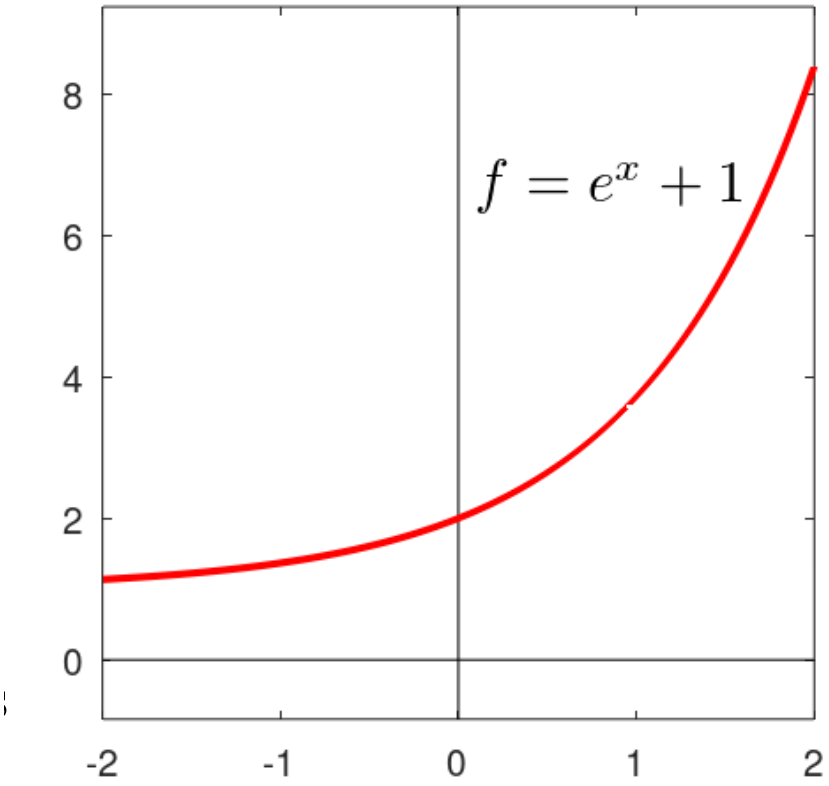
- If f is continuous and sign$(f(a)) \neq$sign$(f(b))$, then Intermediate Value Theorem implies there is $x^{\in}[a, b]$ such that $f(x^*) = 0$

- There is no simple analog for n dimensions

Examples: One Dimension

Nonlinear equations can have any number of solutions

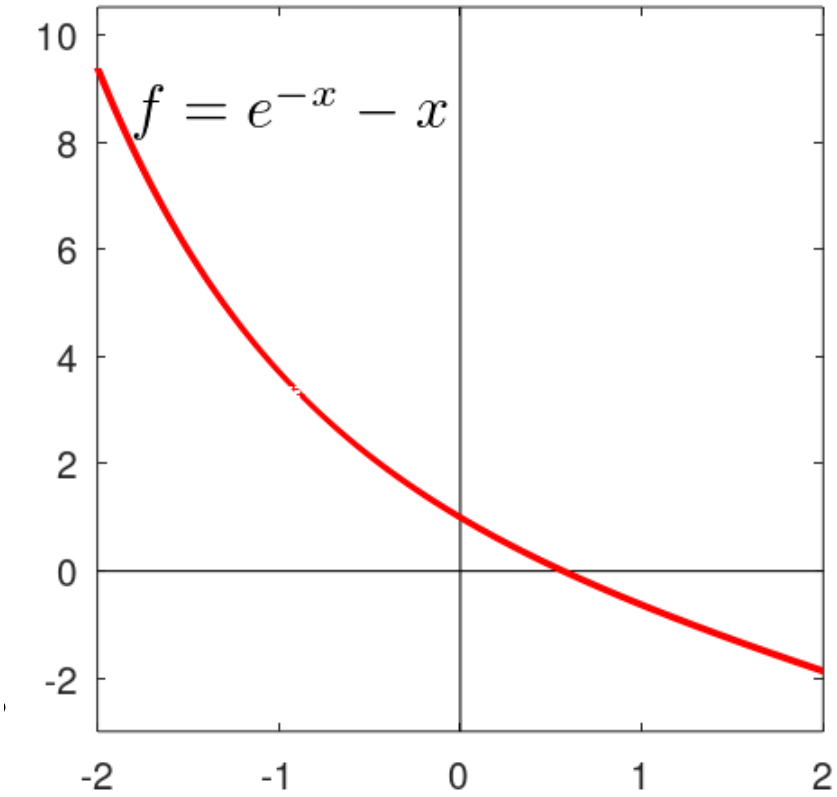
- $e^x + 1 = 0$ has no solution



Examples: One Dimension

Nonlinear equations can have any number of solutions

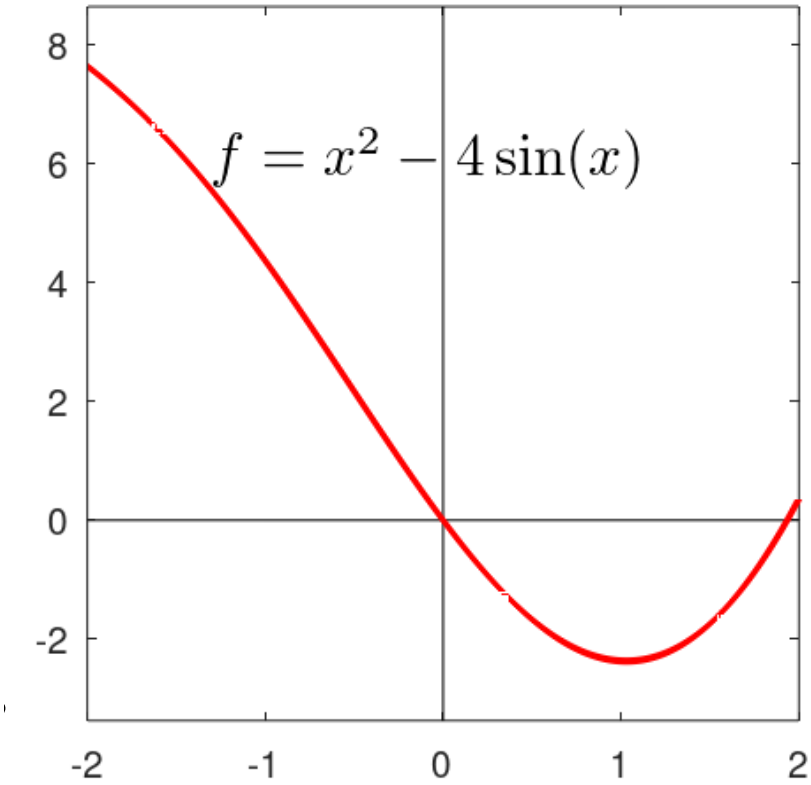
- $e^x + 1 = 0$ has no solution
- $e^{-x} - x = 0$ has one solution



Examples: One Dimension

Nonlinear equations can have any number of solutions

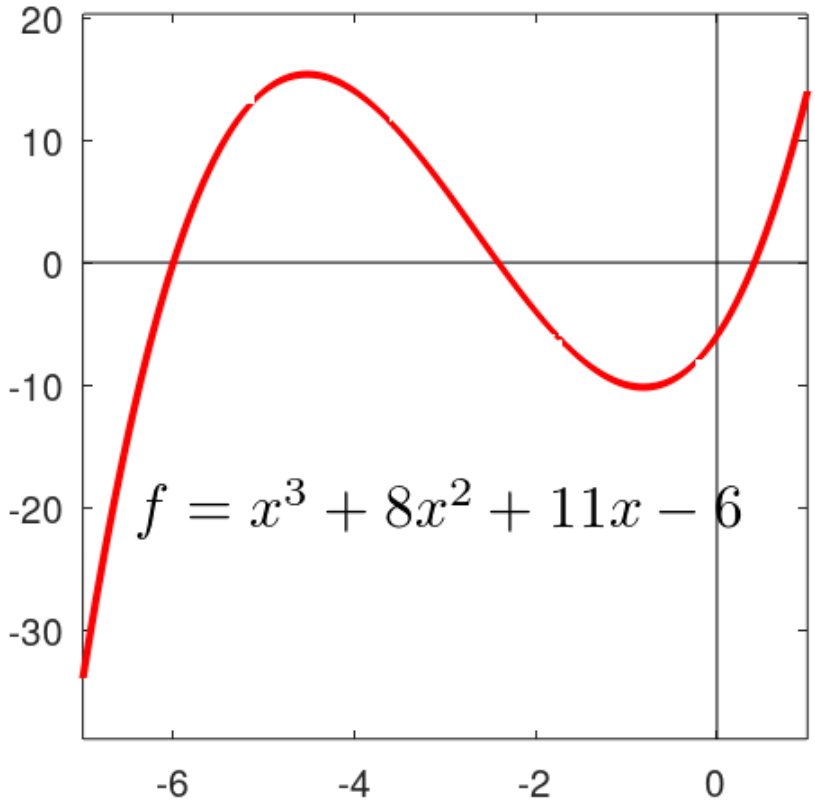
- $e^x + 1 = 0$ has no solution
- $e^{-x} - x = 0$ has one solution
- $x^2 - 4\sin(x) = 0$ has two solutions



Examples: One Dimension

Nonlinear equations can have any number of solutions

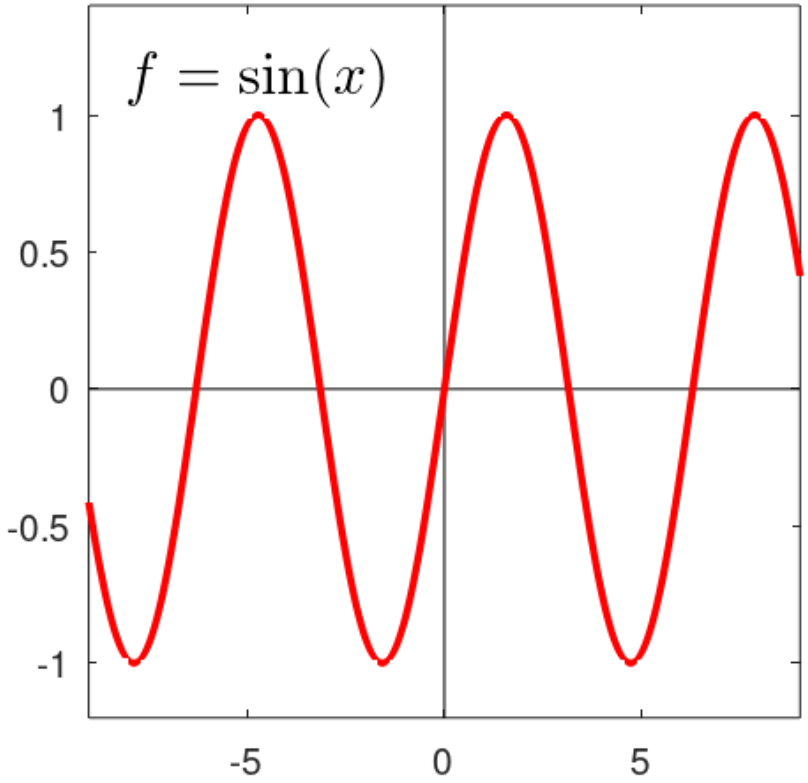
- $e^x + 1 = 0$ has no solution
- $e^{-x} - x = 0$ has one solution
- $x^2 - 4\sin(x) = 0$ has two solutions
- $x^3 + 8x^2 + 11x - 6 = 0$ has three solutions



Examples: One Dimension

Nonlinear equations can have any number of solutions

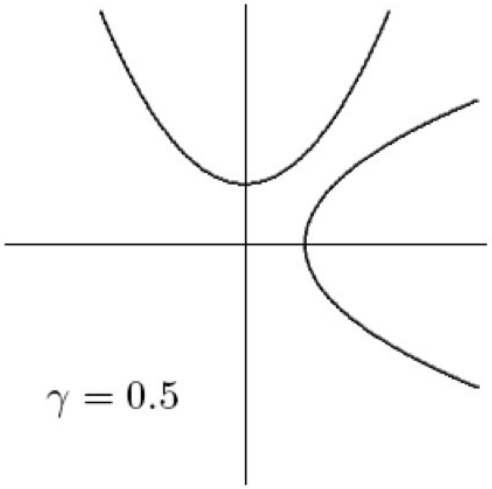
- $e^x + 1 = 0$ has no solution
- $e^{-x} - x = 0$ has one solution
- $x^2 - 4\sin(x) = 0$ has two solutions
- $x^3 + 8x^2 + 11x - 6 = 0$ has three solutions
- $\sin(x) = 0$ has infinitely many solutions

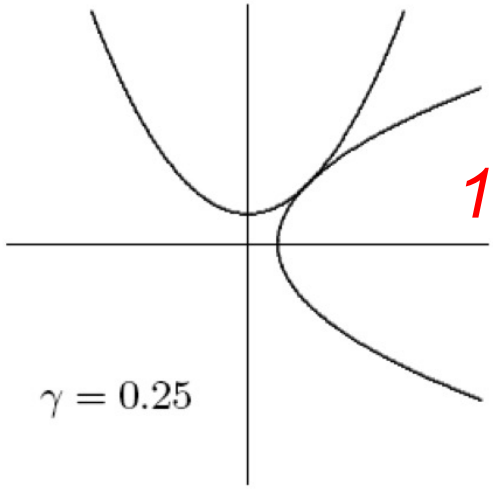


Examples: Systems in Two Dimensions

$$\left.\begin{aligned} x_1^2 - x_2 + \gamma &= 0 \\ -x_1 + x_2^2 + \gamma &= 0 \end{aligned}\right\}$$ *A pair of parabolas*

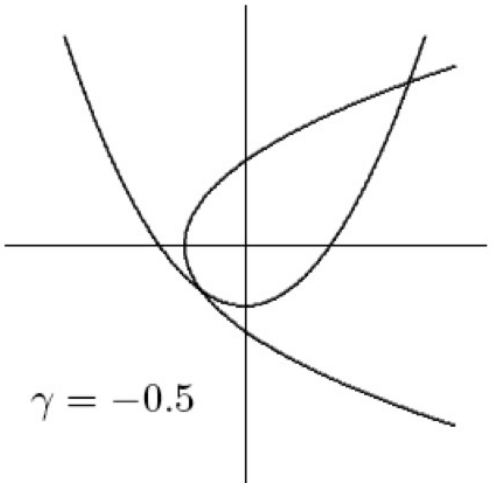
No solution

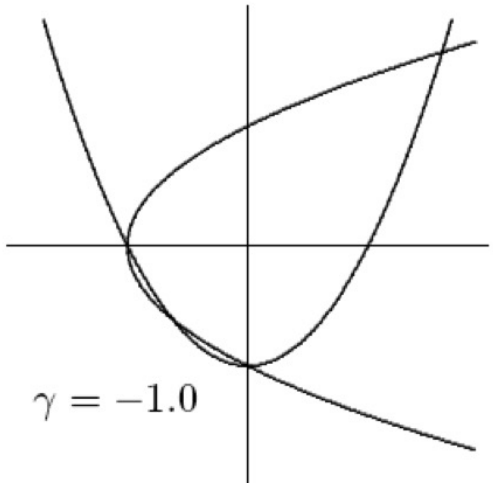




1 solution

2 solutions



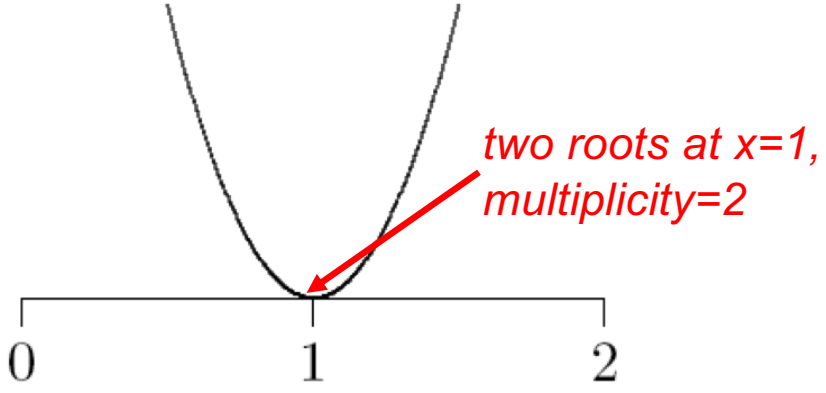


4 solutions

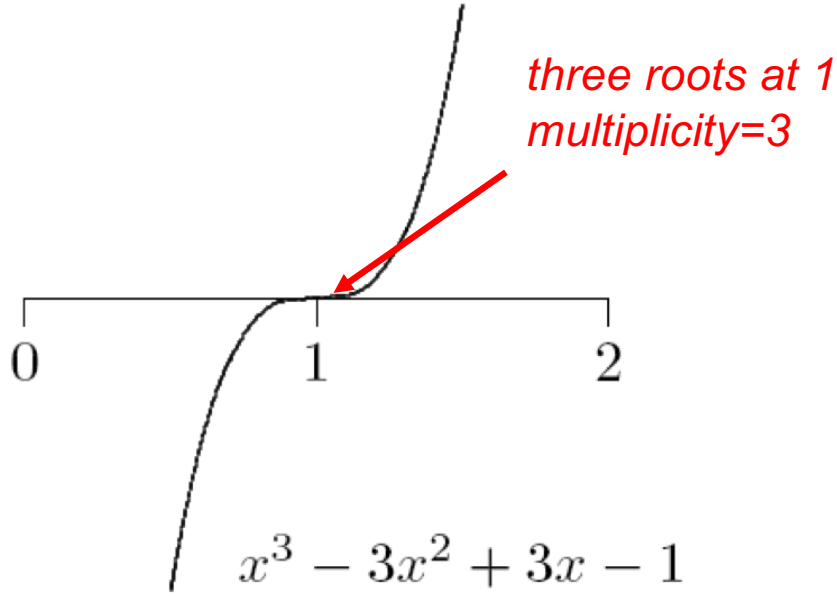
Multiplicity

- If $f(x^*) = f'(x^*) = f''(x^*) = \cdots = f^{(m-1)}(x^*) = 0$, but $f^{(m)}(x^*) \neq 0$ (i.e., mth derivative is lowest nonvanishing derivate at x^*), then root x^* has *multiplicity m*

- If $m = 1$ ($f(x^*) = 0$ and $f'(x^*) \neq 0$), then x^* is *simple* root



$x^2 - 2x + 1$



$x^3 - 3x^2 + 3x - 1$

Sensitivity and Conditioning

- Conditioning of root finding problem is opposite to that for evaluating function (steeper is better!)

- Absolute condition number of root finding problem is root x^* of f is $1/|f'(x^*)|$

- Root is ill-conditioned if tangent line is nearly horizontal

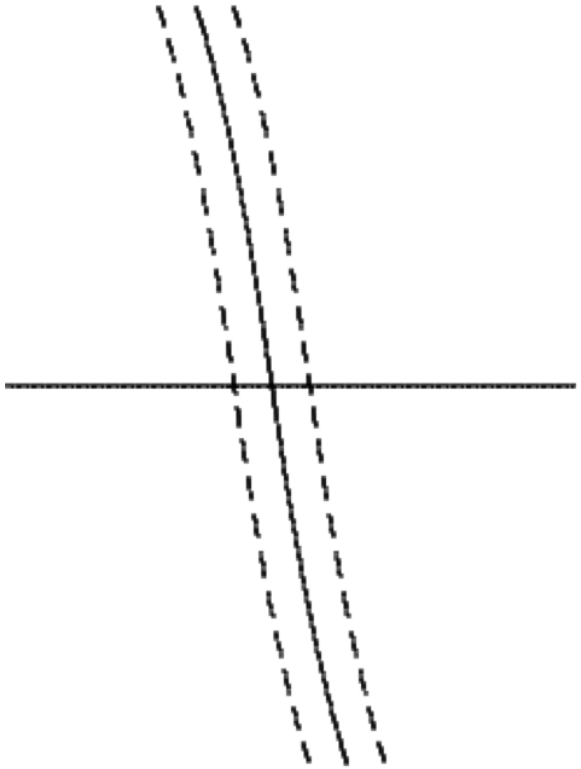
- In particular, multiple root ($m > 1$) is ill-conditioned

- Absolute condition number of root finding problem for system of equations at root $\mathbf{x}^*$ is $\|\mathbf{J}_f^{-1}(\mathbf{x}^*)\|$, where $\mathbf{J}_f$ is *Jacobian matrix* of $\mathbf{f}$:

$$\{\mathbf{J}_f(\mathbf{x})\}_{ij} = \frac{\partial f_i(\mathbf{x})}{\partial x_j}$$

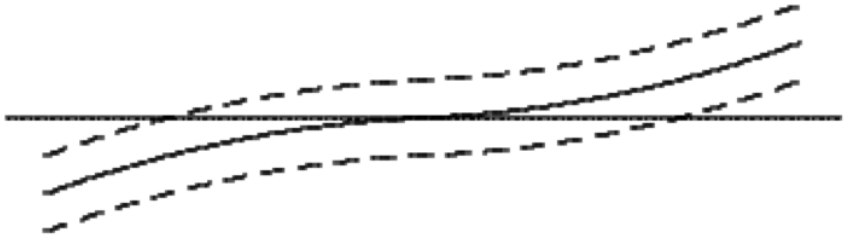
- Root is ill-conditioned if $\mathbf{J}_f(\mathbf{x}^*)$ is nearly singular

Sensitivity and Conditioning



f is near zero for large range of x in neighborhood of x^*.

Difficult to find x^* to significant precision.



- well-conditioned
- ill-conditioned

Sensitivity and Conditioning

- What do we mean by approximate solution $\hat{\mathbf{x}}$ to nonlinear system,

$$\|\mathbf{f}(\hat{\mathbf{x}})\| \approx 0 \quad \text{or} \quad \|\hat{\mathbf{x}} - \mathbf{x}^*\| \approx 0\,?$$

- First corresponds to "small residual," second measures closeness to (usually unknown) true solution $\mathbf{x}^*$

- These solution criteria are not necessarily "small" simultaneously

- Small residual implies accurate solution only if problem is well-conditioned

High multiplicity at x^* implies that values of x near x^* will yield "small" values of $f(x) \sim C|x-x^*|^m$

Convergence Rate

- For general iterative methods, define error at iteration k by

$$\mathbf{e}_k = \mathbf{x}_k - \mathbf{x}^*$$

  where $\mathbf{x}_k$ is approximate solution and $\mathbf{x}^*$ is true solution

- For methods that maintain interval known to contain solution, rather than specific approximate value for solution, take error to be length of interval containing solution

- Sequence converges with rate r if

$$\lim_{k \longrightarrow \infty} \frac{\|\mathbf{e}_{k+1}\|}{\|\mathbf{e}_k\|^r} = C$$

  for some finite nonzero constant C

Important definition for this chapter…

A central theme throughout the chapter (and the course) is to build methods that have a high rate of convergence.

Convergence Rate, continued

Some particular cases of interest

- $r = 1$: *linear* $(C < 1)$
- $r > 1$: *superlinear*
- $r = 2$: *quadratic*

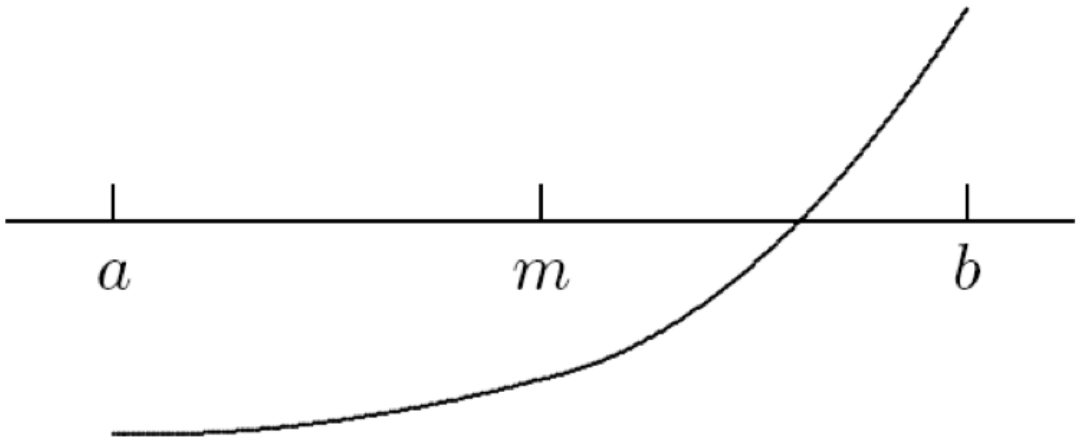
Convergence Rate	Digits gained per iteration
linear	constant
superlinear	increasing
quadratic	double

Methods for One-Dimensional Problems

Interval Bisection Method

Bisection method begins with initial bracket and repeatedly halves the bracket length until solution is isolated as accurately as desired

while $((b - a) > tol)$ do
  $m = a + (b - a)/2$
  if $\text{sign}(f(a)) = \text{sign}(f(m))$ **then**
    $a = m$
  else
    $b = m$
  end
end



Bisection Method, continued

- Bisection method makes no use of magnitudes of function values, only their signs
- Bisection is certain to converge, but does so slowly
- At each iteration, length of interval containing solution is reduced by half, so convergence rate is *linear*, with $(r = 1)$ and $C = 0.5$
- One bit of accuracy is gained in approximate solution fo reach iteration of bisection
- Given starting interval $[a, b]$, length of interval after k iterations is $(b-a)/2^k$, so achieving error tolerance tol reqires

$$\left\lceil \log_2 \left(\frac{b-a}{tol} \right) \right\rceil$$

iterations, regardless of function f involved

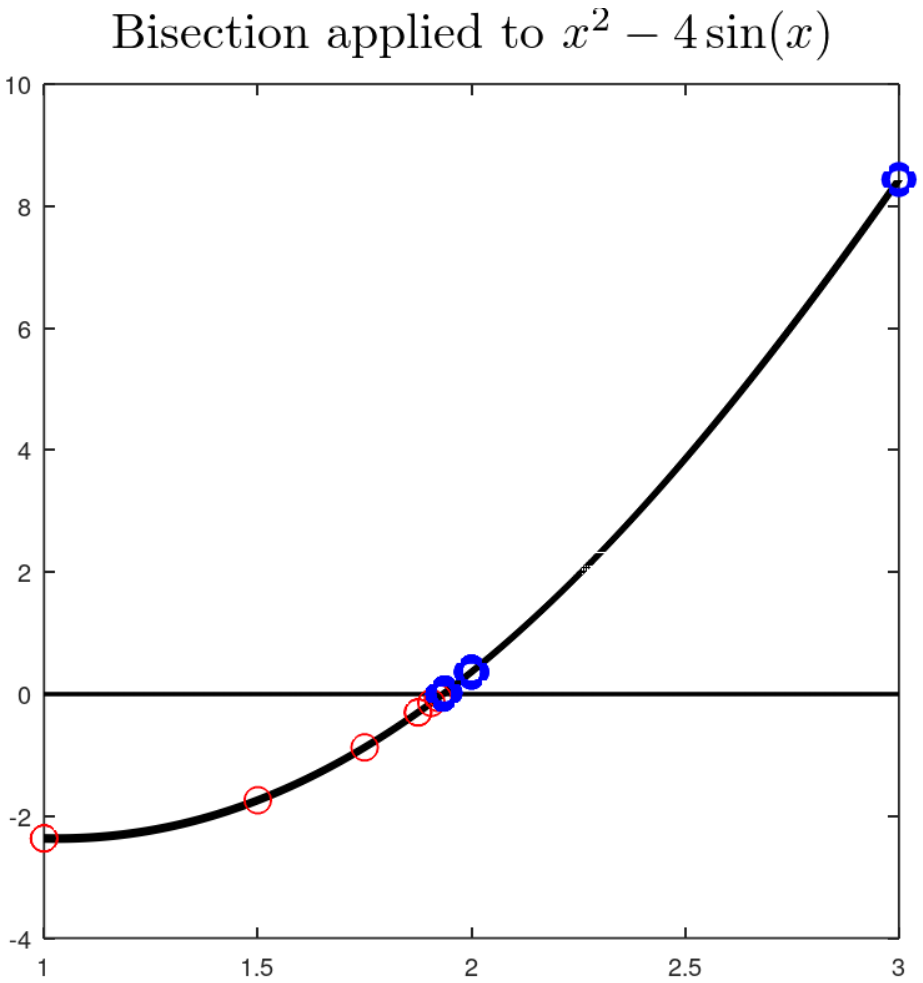
Example: Bisection Method

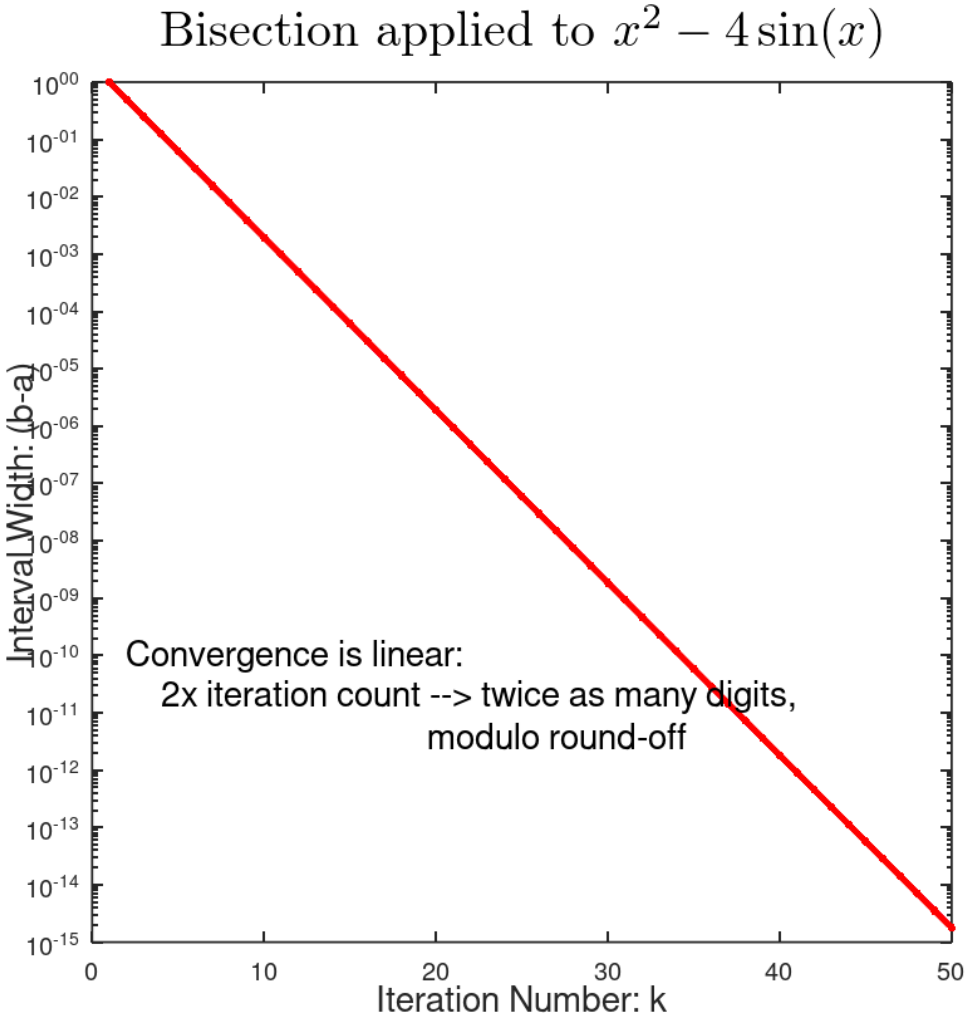
bisect.m

```
a=1; b=3;
x=a; fa=x*x-4*sin(x);
x=b; fb=x*x-4*sin(x);

for k=1:50;
   m=a+(b-a)/2; fm=m*m-4*sin(m);
   if sign(fa)==sign(fm);
      a=m; fa=fm;
   else
      b=m; fb=fm;
   end;
   plot(a,fa,'ro',b,fb,'bo',lw,2);
   disp([k a fa b fb])
   wk(k)=b-a; kk(k)=k;
   pause

end;
```

k	a	f(a)	b	f(b)	b-a
1.0000e+00	1.0000e+00	-2.3659e+00	2.0000e+00	3.6281e-01	1.0000e+00
2.0000e+00	1.5000e+00	-1.7400e+00	2.0000e+00	3.6281e-01	5.0000e-01
3.0000e+00	1.7500e+00	-8.7344e-01	2.0000e+00	3.6281e-01	2.5000e-01
4.0000e+00	1.8750e+00	-3.0072e-01	2.0000e+00	3.6281e-01	1.2500e-01
5.0000e+00	1.8750e+00	-3.0072e-01	1.9375e+00	1.9849e-02	6.2500e-02
6.0000e+00	1.9062e+00	-1.4326e-01	1.9375e+00	1.9849e-02	3.1250e-02
7.0000e+00	1.9219e+00	-6.2406e-02	1.9375e+00	1.9849e-02	1.5625e-02
8.0000e+00	1.9297e+00	-2.1454e-02	1.9375e+00	1.9849e-02	7.8125e-03
9.0000e+00	1.9336e+00	-8.4602e-04	1.9375e+00	1.9849e-02	3.9062e-03
1.0000e+01	1.9336e+00	-8.4602e-04	1.9355e+00	9.4906e-03	1.9531e-03
1.1000e+01	1.9336e+00	-8.4602e-04	1.9346e+00	4.3196e-03	9.7656e-04
1.2000e+01	1.9336e+00	-8.4602e-04	1.9341e+00	1.7361e-03	4.8828e-04
1.3000e+01	1.9336e+00	-8.4602e-04	1.9338e+00	4.4486e-04	2.4414e-04
1.4000e+01	1.9337e+00	-2.0062e-04	1.9338e+00	4.4486e-04	1.2207e-04

bisect.m



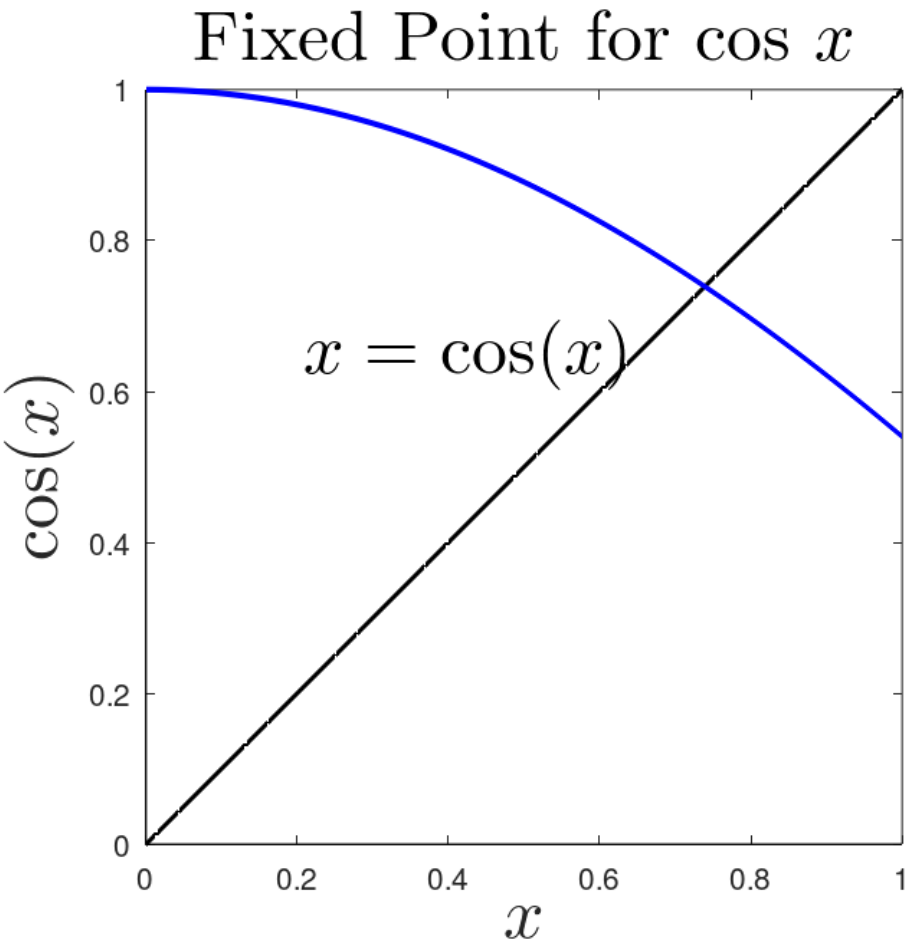


Q: What happens if we go beyond 50 iterations?

Fixed-Point Problems

- *Fixed point* of given function $g : \mathbb{R} \longrightarrow \mathbb{R}$ is value x such that

$$x = g(x)$$



Fixed-Point Problems

- *Fixed point* of given function $g : \mathbb{R} \longrightarrow \mathbb{R}$ is value x such that

$$x = g(x)$$

- Many iterative methods for solving nonlinear equations use *fixed-point iteration* of the form

$$x_{k+1} = g(x_k),$$

- Also called *functional iteration*, since g is applied repeatedly to the output, starting with initial value x_0

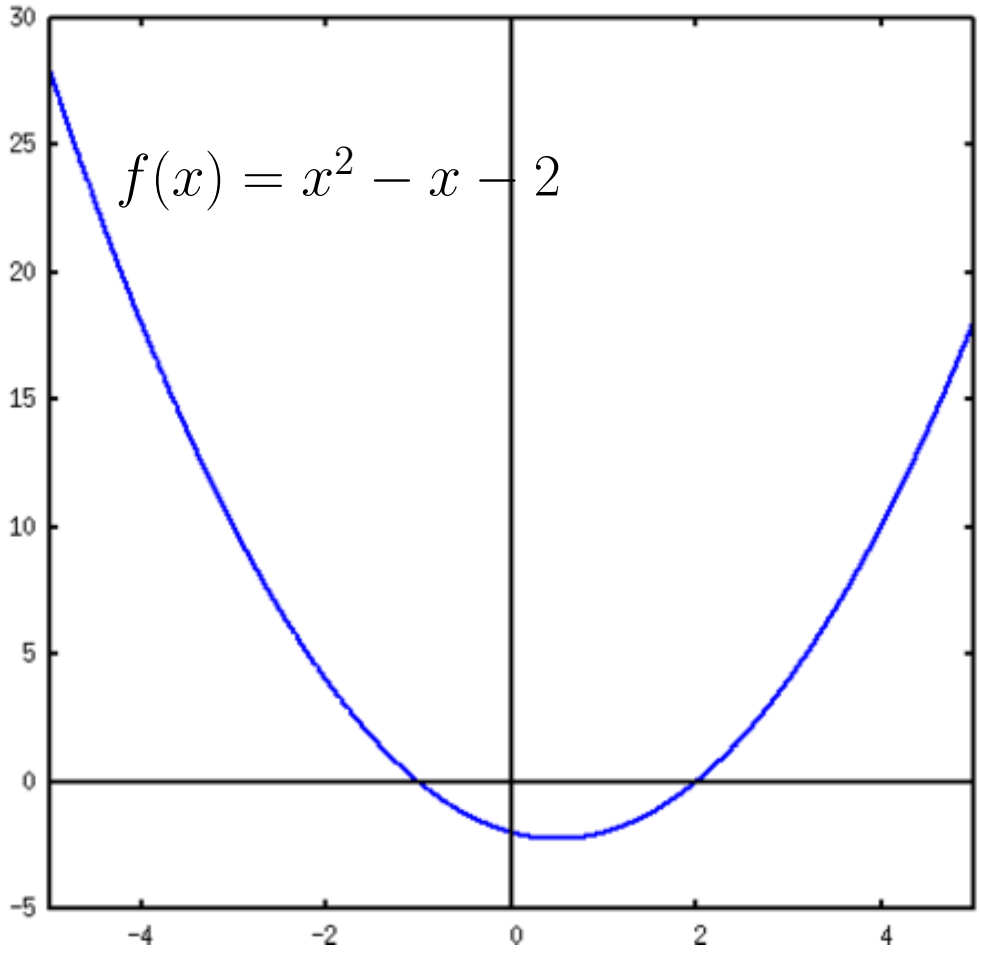
- For given equation $f(x) = 0$, there may be many equivalent fixed-point problems with different choices for g and different rates of convergence

Example: Fixed-Point Problems

If $f(x) = x^2 - x - 2$, then fixed points of each of the functions

- $g(x) = x^2 - 2$
- $g(x) = \sqrt{x + 2}$
- $g(x) = 1 + 2/x$
- $g(x) = \frac{x^2+2}{2x-1}$

are solutions to equation
$f(x) = 0$ at $x^* = 2$



Essential Questions for Fixed-Point Iteration

$$x^* = g(x^*) \qquad \textit{fixed-point}$$

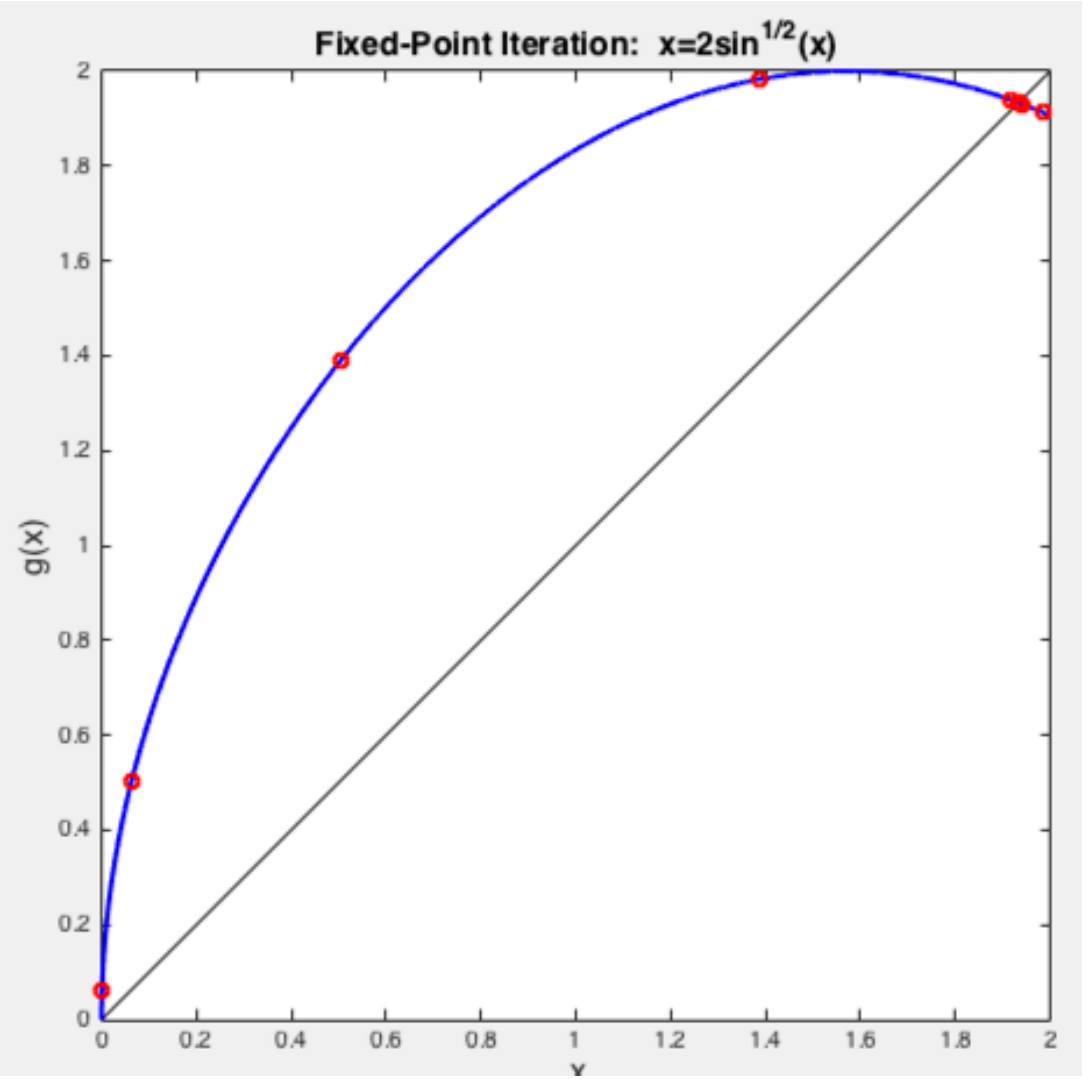
$$x_k = g(x_{k-1}) \qquad \textit{fixed-point iteration}$$

$$x_k \longrightarrow x^* \qquad \textit{convergent?}$$

- Is the convergence linear? Superlinear?
- Fixed-point iteration is often superlinear.
- Fixed-point iteration extends to multiple dimensions.
- Newton's method is a fixed-point iteration.

fixpt_demo.m

```
x=.001;
for k=1:30;
    g=sqrt(4*sin(x));
    disp([k x g abs(x-g)]);
    plot(x,g,'ro',lw,2); pause
    x=g;
end;
```



Q: Why does the iteration diverge near x=0, but converge to x^* near 1.9 ?

Convergence of Fixed-Point Iteration

- Define error at kth iterate: $e_k = x_k - x^* \iff x_k = x^* + e_k$

- Apply fixed-point iteration and subtract fixed-point solution,

$$\begin{aligned} x_{k+1} &= g(x_k) \\ - \quad x^* &= g(x^*) \\ \hline e_{k+1} &= g(x_k) - g(x^*) \end{aligned}$$

- Use Taylor series expansion about x^*

$$\begin{aligned} e_{k+1} &= g(x^* + e_k) - g(x^*) \\ &= [g(x^*) + e_k\, g'(x^*) + \frac{e_k^2}{2} g''(x^*) + \cdots] - g(x^*) \\ &= e_k\, g'(x^*) + \frac{e_k^2}{2} g''(x^*) + \text{higher-order terms} \end{aligned}$$

- Therefore, as $k \longrightarrow \infty$,

$$\frac{|e_{k+1}|}{|e_k|} \sim |g'(x^*)| \text{ if } g'(x^*) \neq 0 \text{ (linear)}$$

$$\frac{|e_{k+1}|}{|e_k|^2} \sim \frac{1}{2} |g''(x^*)| \text{ if } g'(x^*) = 0 \text{ and } g''(x^*) \neq 0 \text{ (quadratic)}$$

Convergence of Fixed-Point Iteration

- If $x^* = g(x^*)$ and $|g'(x^*)| < 1$, then there is an interval containing x^* such that the iteration

$$x_{k+1} \;=\; g(x_k),$$

  converges to x^* if started within that interval

- If $|g'(x^*)| > 1$, then the iterative scheme diverges

- Asymptotic convergence rate of fixed-point iteration is usually linear, with constant $C = |g'(x^*)|$

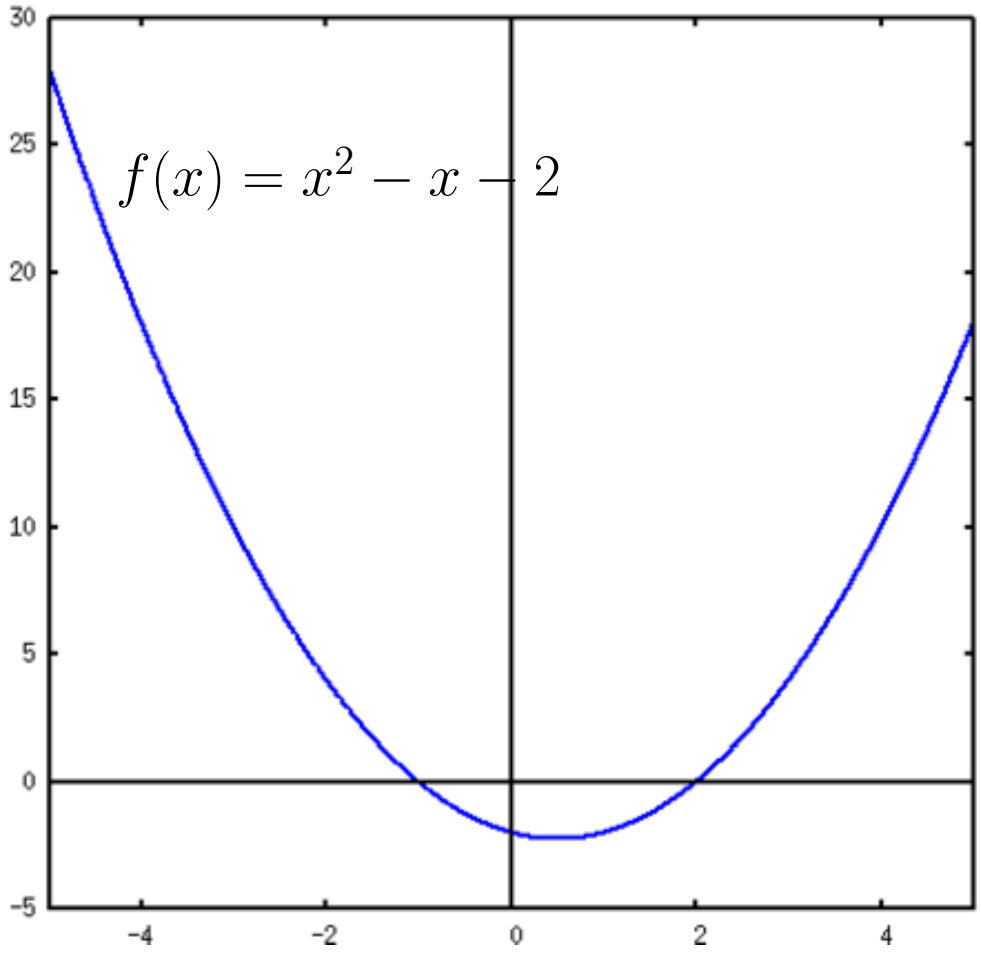
- But if $g'(x^*) = 0$, then convergence rate is at least quadratic

Returning to Our Fixed Point Examples

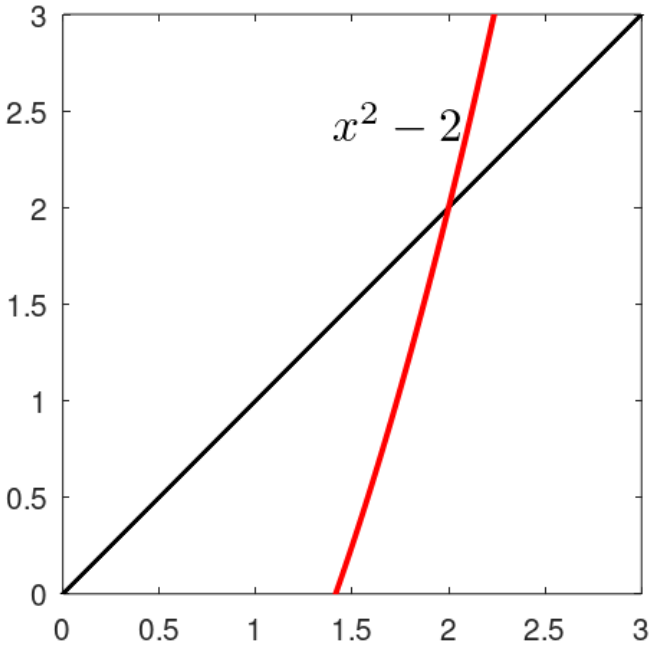
If $f(x) = x^2 - x - 2$, then fixed points of each of the functions

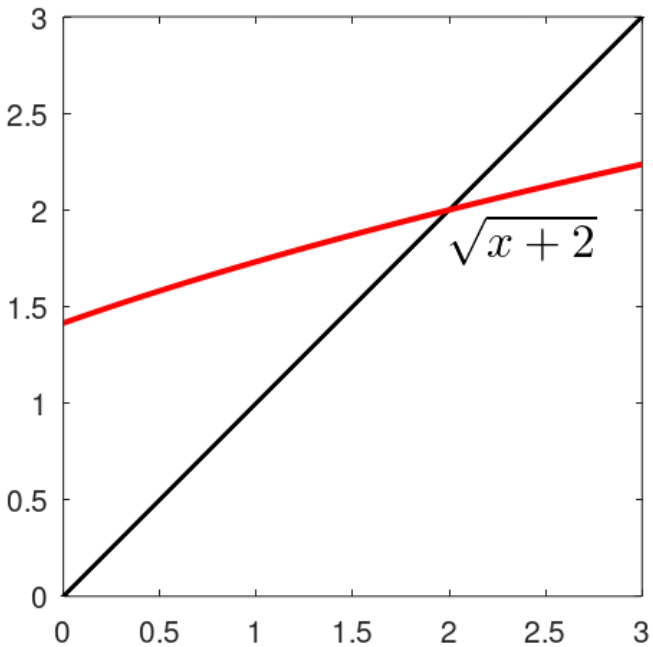
- $g(x) = x^2 - 2$
- $g(x) = \sqrt{x + 2}$
- $g(x) = 1 + 2/x$
- $g(x) = \frac{x^2+2}{2x-1}$

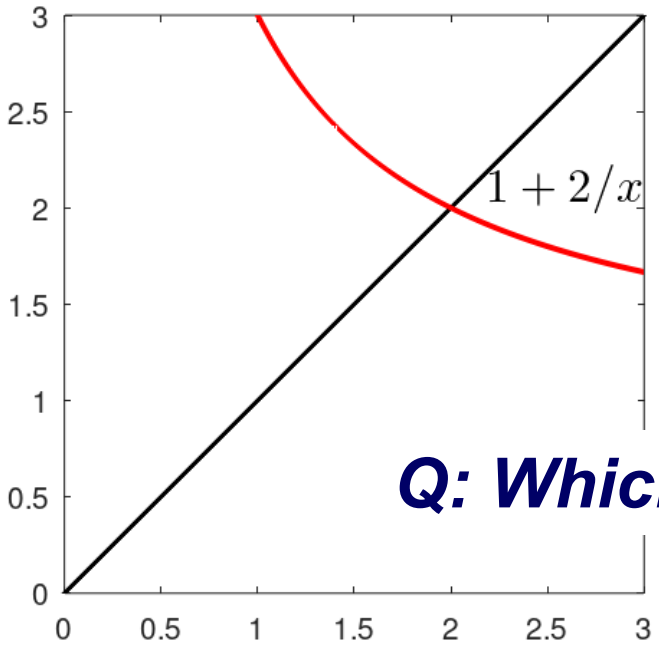
are solutions to equation
$f(x) = 0$ at $x^* = 2$

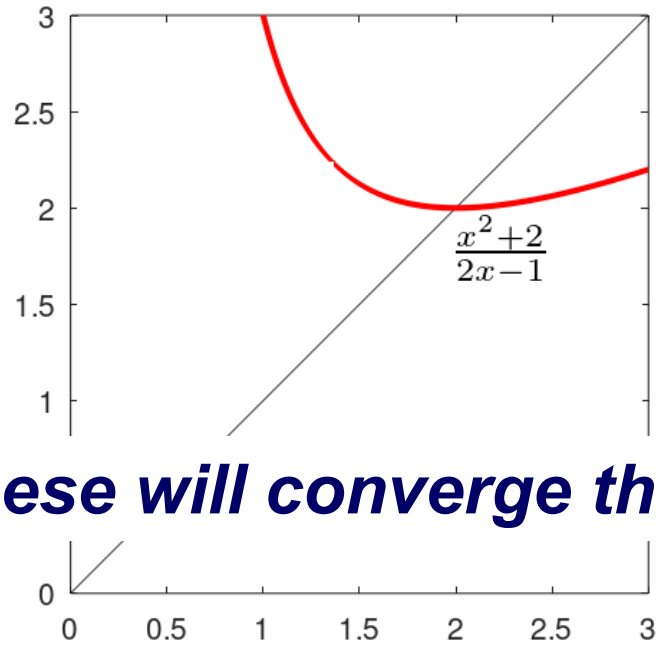


Example: Fixed-Point Problems





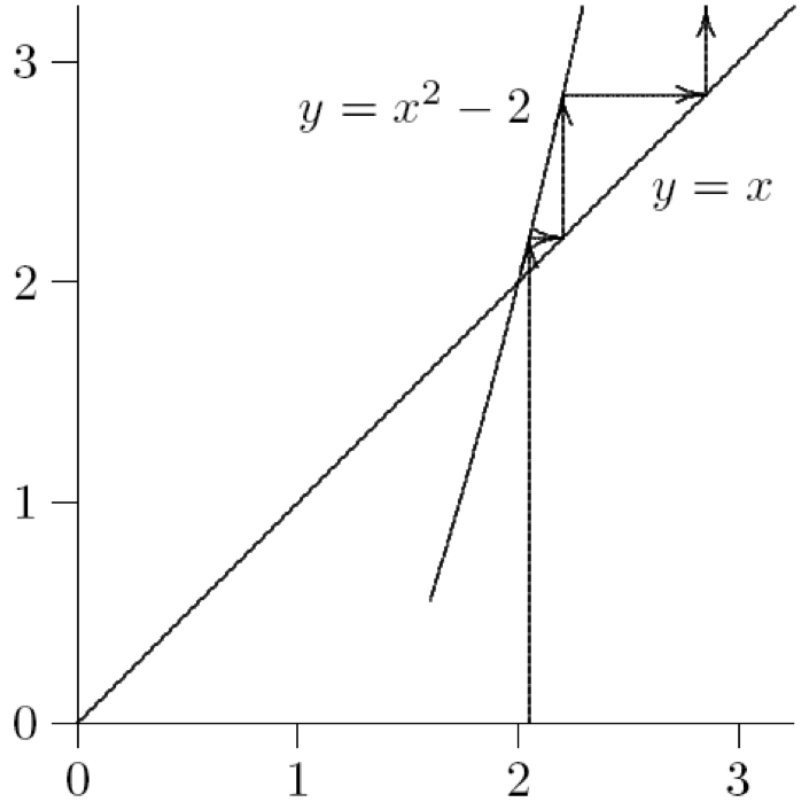




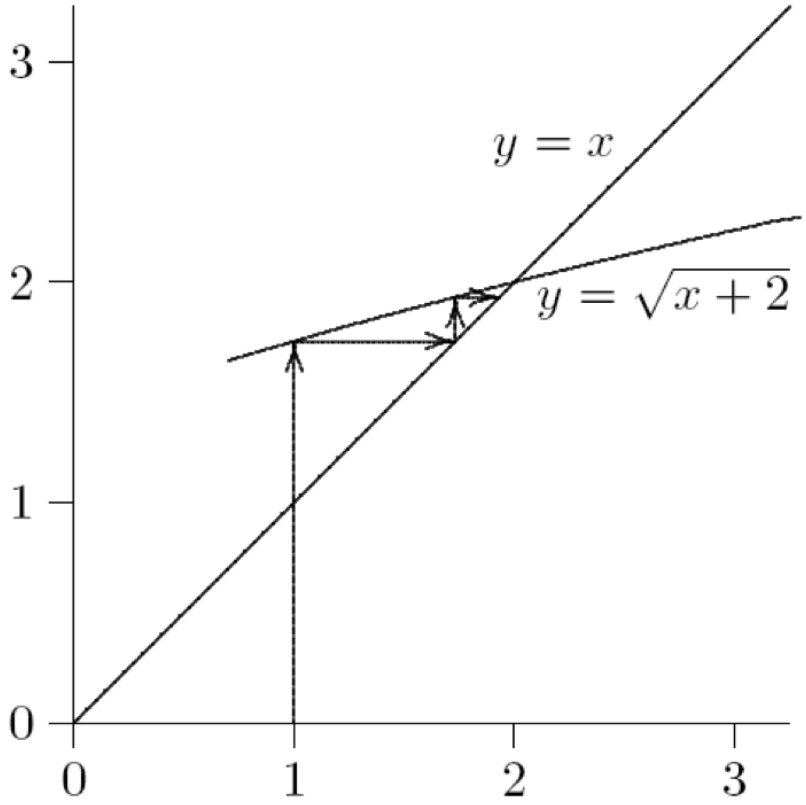
Q: Which of these will converge the fastest?

fixpt(k).m

Example: Fixed-Point Problems

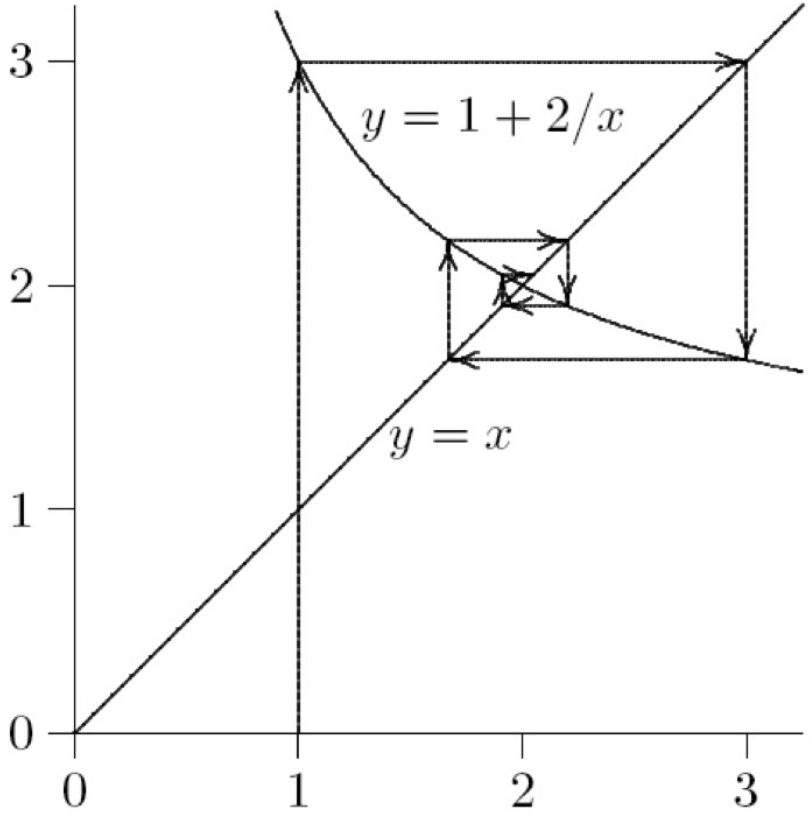


Diverges!

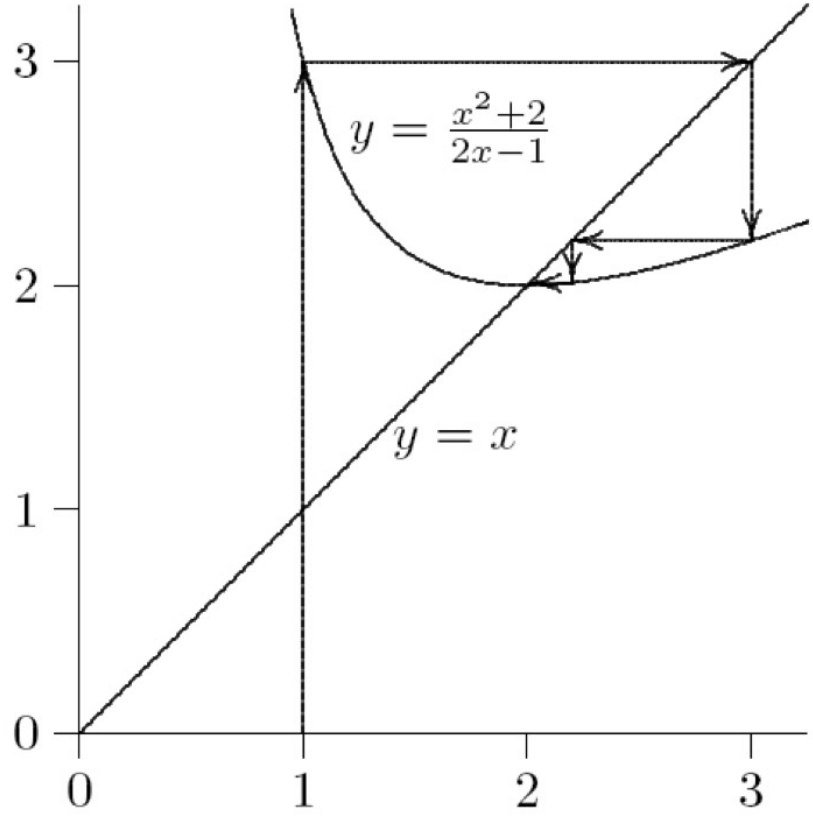


Converges!

Example: Fixed-Point Problems



Converges!



Converges!

Accelerating Linearly Convergent Sequences

- Often, we don't have an equation of the form $f(x) = 0$.
- Instead, we have a sequence x_k that is approaching a value x^* .
- Linear convergence can be accelerated via many methods.
- One historically important one is *Aitken's* Δ^2 *Method*:

$$y_{k+2} \;:=\; x_{k+2} \;-\; \frac{\Delta_2 \Delta_2}{\Delta_2 \;-\; \Delta_1},$$

with

$$\begin{aligned} \Delta_1 \;&:=\; x_{k+1} \;-\; x_k \\ \Delta_2 \;&:=\; x_{k+2} \;-\; x_{k+1}. \end{aligned}$$

- For a linearly-convergent sequence x_k, the corresponding y_ks will generally be closer to x^*. (One must be careful about round-off.)

- This improved convergence suggests the following modified fixed point iteration for the solution $x^* = g(x^*)$:

Fixed Point Iteration

- Start with x_0.

for $k = 0, 1, \ldots,$

$x_{k+1} = g(x_k)$

end

Accelerated Iteration

- Start with x_0.

for $k = 0, 2, 4, \ldots,$

$x_{k+1} = g(x_k)$

$x_{k+2} = g(x_{k+1})$

$\Delta_2 = x_{k+2} - x_{k+1},\ \Delta_1 = x_{k+1} - x_k$

$x_{k+2} = x_{k+2} - \Delta_2^2/(\Delta_2 - \Delta_1)$

end

- *matlab code:*

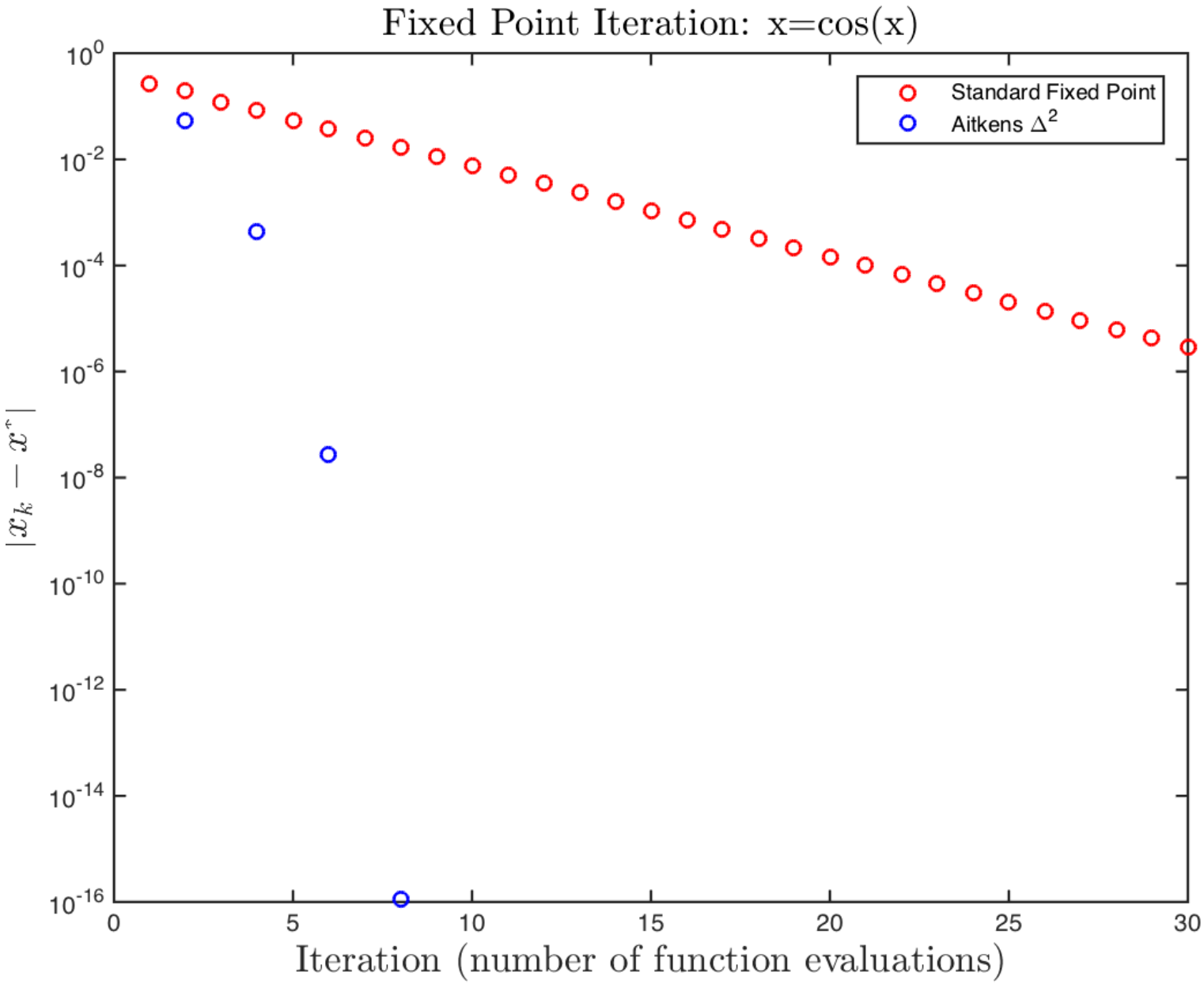
```
x0=0;
for k=1:5;
    x1=g(x0);
    x2=g(x1);
    d1=x1-x0; d2=x2-x1;
    x0=x2 - (d2*d2)/(d2-d1);
end;
```

- Aitken's Δ^2 method is essentially linear extrapolation of successive outputs from a linearly-convergence sequence.

Matlab demo: aitken.m

```
%% Aitken's D^2 method applied to FPI

clear all; close all; format compact; format longe;

x=0; for k=1:100; x=cos(x); end; xstar = x;

x=0;
for k=1:30;
    x=cos(x); [k x]
    xk(k)=x;
    semilogy(k,abs(x-xstar),'ro','linewidth',1.3)
    legend('Standard Fixed Point')
    title('Fixed Point Iteration: x=cos(x)','fontsize',14)
    xlabel('Iteration (number of function evaluations)','fontsize',14)
    ylabel('|x_k - x^* |','fontsize',14)
    hold on; pause(.1)
end;
pause

x0=0;
for k=1:5;
    x1=cos(x0);
    x2=cos(x1);
    d1=x1-x0; d2=x2-x1;
    x0=x2 - (d2*d2)/(d2-d1);
    [2*k xk(2*k) x0]
    semilogy(2*k,abs(xk(2*k)-xstar),'ro',2*k,abs(x0-xstar),'bo','linewidth',1.3)
    title('Fixed Point Iteration: x=cos(x)','fontsize',14)
    xlabel('Iteration (number of function evaluations)','fontsize',14)
    ylabel('|x_k - x^* |','fontsize',14)
    legend('Standard Fixed Point','Aitkens \Delta^2')
    hold on; pause
end;
```

Matlab demo: aitken.m



Aitken's Δ^2 Method Converges for a Divergent Sequence!

aitken2.m
aitken3.m

```
%
%    Aitken's Delta-Squared Method applied
%    to a linearly DIVERGENT sequence.
%
%    x_k = x_k^2 - 2

format compact; format longe
x0=1.5;
for k=1:9;
    x1=x0*x0-2;
    x2=x1*x1-2;
    d1=x1-x0; d2=x2-x1;
    x0=x2 - (d2*d2)/(d2-d1);
    [k x0 x2 x1]
end;
```

	y_{k+2}	x_{k+2}	x_{k+1}
1	3.166666666666667e+00	-1.937500000000000e+00	2.500000000000000e-01
2	2.689827429609444e+00	6.244521604938275e+01	8.027777777777780e+00
3	2.322268653039224e+00	2.540702169274772e+01	5.235171601079349e+00
4	2.095202364357393e+00	9.511985499751427e+00	3.392931696888611e+00
5	2.010650222187136e+00	3.711492705712416e+00	2.389872947608809e+00
6	2.000148988746703e+00	2.172681776714464e+00	2.042714315981181e+00
7	2.000000029590617e+00	2.002384263926645e+00	2.000595977184460e+00
8	2.000000000000001e+00	2.000000473449884e+00	2.000000118362467e+00
9	2.000000000000000e+00	2.000000000000021e+00	2.000000000000005e+00

Aitken's Δ^2 Method Converges for a Divergent Sequence!

```
%
%    Aitken's Delta-Squared Method applied
%    to a linearly DIVERGENT sequence.
%
%    x_k = x_k^2 - 2

format compact; format longe
x0=1.5;
for k=1:9;
    x1=x0*x0-2;
    x2=x1*x1-2;
    d1=x1-x0; d2=x2-x1;
    x0=x2 - (d2*d2)/(d2-d1);
    [k x0 x2 x1]
end;
```

aitken2.m

	y_{k+2}	x_{k+2}	x_{k+1}
1	3.166666666666667e+00	-1.937500000000000e+00	2.500000000000000e-01
2	2.698274296094440e+00	6.244521604938275e+01	8.027777777777780e+00
3	2.322268653039224e+00	2.540702169274772e+01	5.235171601079349e+00
4	2.095202364357393e+00	9.511985499751427e+00	3.392931696888611e+00
5	2.010650222187136e+00	3.711492705712416e+00	2.389872947608809e+00
6	2.000148988746703e+00	2.172681776714464e+00	2.042714315981181e+00
7	2.000000029590617e+00	2.002384263926645e+00	2.000595977184460e+00
8	2.000000000000001e+00	2.000000473449884e+00	2.000000118362467e+00
9	2.000000000000000e+00	2.000000000000021e+00	2.000000000000005e+00

Alternatives to Aitken's Δ^2-Method

- If you start with a sequence

$$x_{k+1} \;=\; g(x_k),$$

with fixed point $x^* = g(x^*)$, you can rewrite this as a root-finding problem:

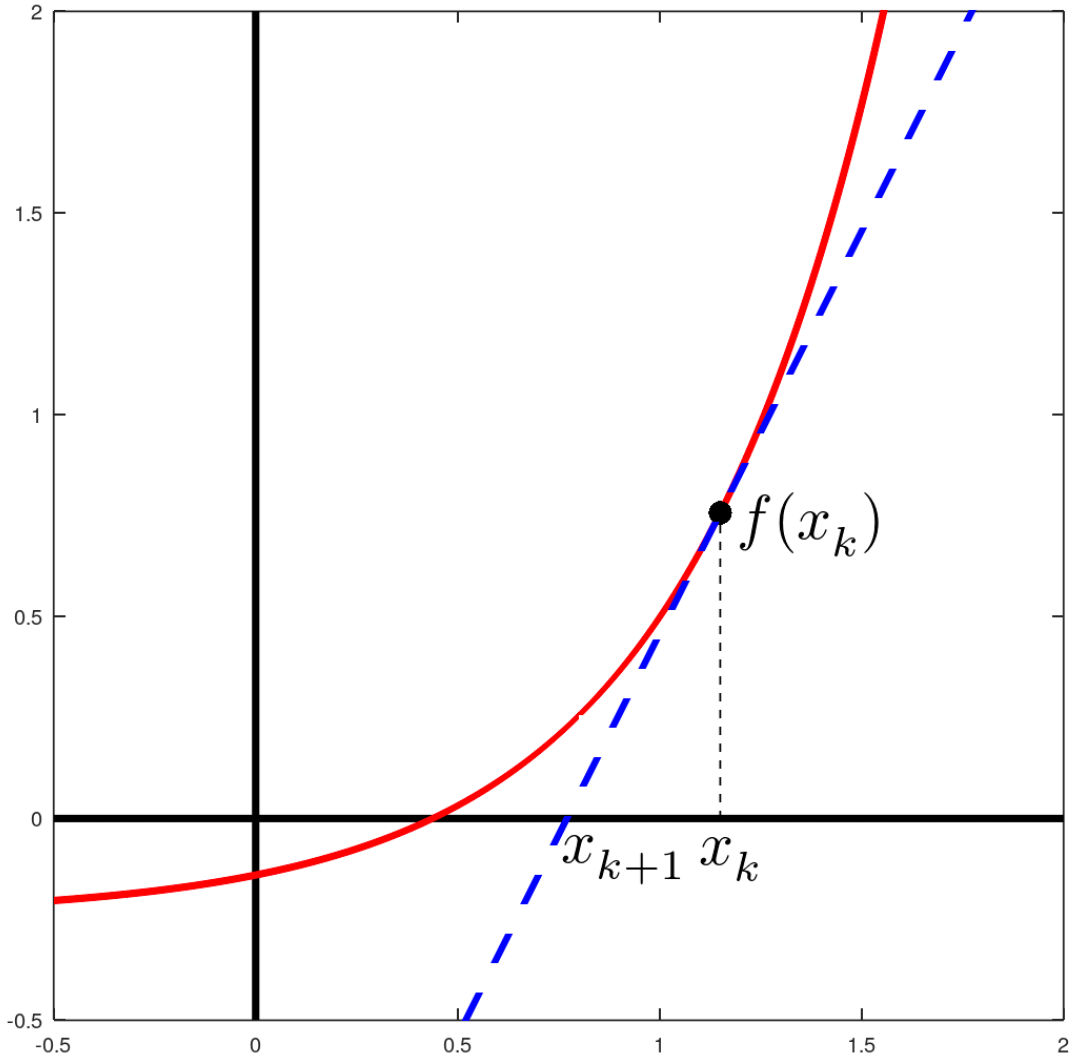
$$f(x) \;:=\; x - g(x),$$

for which $f(x^*) = 0$.

- Thus, any fixed-point iteration can also be transformed into a root-finding problem, to which we can apply standard acceleration techniques.
- The most common technique is *Newton's Method*, also known as the Newton-Raphson Method.
- Other related techniques (e.g., Secant Method) try to mimic the main idea of Newton's Method with lower cost.

Newton's Method

Newton's method approximates nonlinear function f near x_k by *tangent line* at $f(x_k)$



Newton's Method

- Truncated Taylor series

$$f(x+h) \approx f(x) + f'(x)h$$

  is linear function approximating f near x

- Replace f by this linear function, solve for $f(x+h) = 0$,

$$0 = f(x+h) = f(x) + f'(x)h \longrightarrow h = -f(x)/f'(x)$$

- Zero of this linear function does not exactly match that of $f(x)$, so repeat process starting at $x = x + h$, yields iteration scheme, *Newton's Method*

$$\boxed{x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}}$$

Example: Newton's Method

- Use Newton's method to find root of $f(x) = x^2 - 4\sin(x) = 0$
- Derivative is $f'(x) = 2x - 4\cos(x)$, so iteration is

$$x_{k+1} = x_k - \frac{x_k^2 - 4\sin(x_k)}{2x_k - 4\cos(x_k)}$$

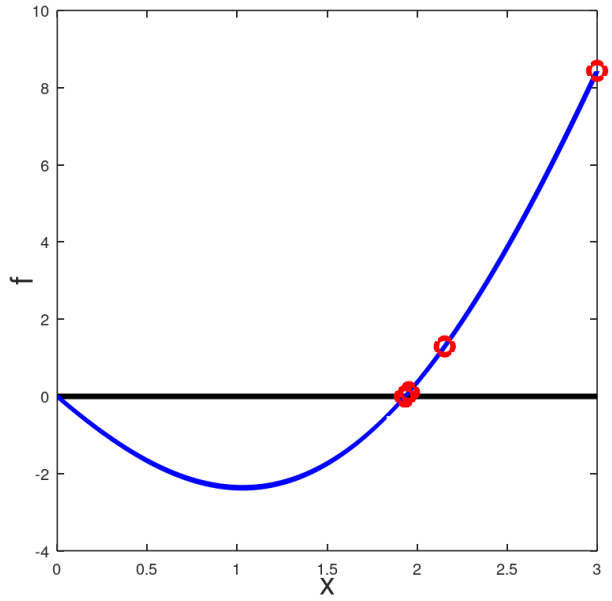
- Taking $x_0 = 3$ as starting point, we obtain

k	x_k	f_k	dx_k	df_k
1.0000e+00	2.1531e+00	8.4355e+00	-8.4694e-01	8.4355e+00
2.0000e+00	1.9540e+00	1.2948e+00	-1.9902e-01	-7.1407e+00
3.0000e+00	1.9340e+00	1.0844e-01	-2.0067e-02	-1.1863e+00
4.0000e+00	1.9338e+00	1.1516e-03	-2.1774e-04	-1.0729e-01
5.0000e+00	1.9338e+00	1.3605e-07	-2.5731e-08	-1.1515e-03
6.0000e+00	1.9338e+00	2.2204e-15	-4.1993e-16	-1.3605e-07

newton_4sin.m

```
hdr; format shorte

x=0:.01:3; f  = x.*x-4*sin(x);
hold off; plot(x,0*x,'k-',lw,2,x,f,'b-',lw,2); hold on;
xlabel('x',fs,20); ylabel('f',fs,20);

x=3; f=0;
disp('       k           x_k           f_k           dx_k        df_k ')
for k=1:10;
   fo = f;
   f  = x*x-4*sin(x); fp = 2*x-4*cos(x);
   plot(x,f,'ro',lw,2); pause(.4);

   s   = -f/fp;
   x   = x+s;

   y   = f-fo; disp([k x f s y]) % print convergence

end;
```

newton_4sin.m



Convergence of Newton's Method

- Newton's method is a fixed-point iteration with

$$g(x) \;=\; x \;-\; \frac{f(x)}{f'(x)}$$

and

$$g'(x) \;=\; 1 \;-\; \frac{(f'(x))^2 - f(x)f''(x)}{(f'(x))^2}$$

- If x^* is a simple root then $f(x^*) = 0$ and

$$g'(x^*) \;=\; 1 \;-\; \frac{(f'(x))^2}{(f'(x))^2} \;=\; 0$$

so we expect at *quadratic convergence* ($r = 2$)

- Iterations must start close enough to root to converge (i.e., need x_0 in "*ball of convergence*")

Examples of Newton's Method

- Newton's method is often used for intrinsic functions such as $x = 1/A$ and $x = \sqrt{A}$.

- In addition to the rate of convergence, it is important to know how to generate an initial guess that will be in the *interval of convergence*, so that for any x_0, $x_k \longrightarrow x^*$

- Let's look at three examples for these cases.

Examples of Newton's Method: Compute $1/A$.

- Start with $f(x) = A - \frac{1}{x}$ $\qquad f(\frac{1}{A}) = A - A = 0$

$$
\begin{aligned}
g(x) &= x - \frac{f}{f'} \qquad f' = x^{-2} \\
&= x - \frac{A - \frac{1}{x}}{x^{-2}} \\
&= x - \left(Ax^2 - x\right) \\
&= 2x - Ax^2
\end{aligned}
$$

- Verify: $g(1/A) = 2/A - 1/A = 1/A$

$$g'(1/A) = 2 - 2A/A = 0$$

- The fixed-point iteration is therefore

$$x_{k+1} = (2 - Ax_k)\, x_k$$

- Note that this expression involves two multiplies and one addition and of course no divisions since that is the operator we are trying to implement

Compute $1/A$: Interval of Convergence

- It is important to understand the interval of convergence for this method.
- That is, for what range of x_k will $|g'(x)| < 1$?
- Usually easiest to answer the question with respect to x^*.
- So, one has

$$g'(x) \;=\; 2 - 2Ax = 2 - 2\frac{x}{x^*}.$$

- Inserting into the bracketing range, $-1 < g' < 1$,

$$\frac{1}{2} < \frac{x_0}{x^*} < \frac{3}{2}$$

  will guarantee convergence.

- If $A = 1.bbb\ldots \times 2^k$, then

$$\frac{1}{2} \times 2^{-k} \leq A^{-1} = \frac{1}{1.bbb\ldots} \times 2^{-k} \leq 1 \times 2^{-k}.$$

- Can take as an initial guess

$$x_0 \;=\; \frac{3}{4} \times 2^{-k},$$

  from which one easily verifies that $x_0 A$ is in $[\frac{1}{2}, 1]$.

newton_inv.m

9.2 SINGLE-PRECISION DIVIDE

Example 9-3 computes Z = X ÷ Y for single-precision variables. The algorithm begins by using the reciprocal instruction **frcp** to obtain an initial guess for the value of 1/Y. The **frcp** instruction gives a result that can differ from the true value of 1/Y by as much as 2^{-8}. The algorithm then continues to make guesses based on the prior guess, refining each guess until the desired accuracy is achieved. Let G represent a guess, and let E represent the error, i.e. the difference between G and the true value of 1/Y. For each guess...

$G_{new} = G_{old}(2 - G_{old}*Y)$.
$E_{new} = 2(E_{old})^2$.

This algorithm is optimized for high performance and does not produce results that are rounded according to the IEEE standard. Worst case error is about two least-significant bits. If the result is referenced by the next instruction, 22 clocks are required to perform the divide.

```
// SINGLE-PRECISION DIVIDE

//      The dividend X is in f6
//      The divisor Y is in f2
//      The result Z is left in f3
//      f5 contains single-precision floating-point 2.

    frcp.ss f2,     f3          // first guess has 2**-8 error
    fmul.ss f2,     f3,     f4  // guess * divisor
    fsub.ss f5,     f4,     f4  // 2 - guess * divisor
    fmul.ss f3,     f4,     f3  // second guess has 2**-15 error
    fmul.ss f2,     f3,     f4  // avoid using f3 as src1
    fsub.ss f5,     f4,     f4  // 2 - guess * divisor
    fmul.ss f6,     f3,     f5  // second guess * dividend
    fmul.ss f4,     f5,     f3  // result = second guess * dividend
```

9.3 DOUBLE-PRECISION DIVIDE

Example 9-4 computes Z = X ÷ Y for double-precision variables. The algorithm is similar to that shown previously for single-precision divide. For double-precision divide, one more iteration is needed to achieve the required accuracy.

This algorithm is optimized for high performance and does not produce results that are rounded according to the IEEE standard. Worst case error is about two least-significant bits. If the result is referenced by the next instruction, 38 clocks are required to perform the divide.

```
// DOUBLE-PRECISION DIVIDE

//      The dividend X is in f2
//      The divisor Y is in f4
//      The result Z is left in f8

    frcp.dd f4,     f6          // first guess has 2**-8 error
    fmul.dd f4,     f6,     f8  // guess * divisor
    fld.d   fltwo, f10          // load double-precision floating 2
// The fld.d is free. It completely overlaps the preceding fmul.dd
    fsub.dd f10,    f8,     f8  // 2 - guess * divisor
    fmul.dd f6,     f8,     f6  // second guess has 2**-15 error
    fmul.dd f4,     f6,     f8  // avoid using f6 as src1
    fsub.dd f10,    f8,     f8  // 2 - guess * divisor
    fmul.dd f6,     f8,     f6  // third guess has 2**-29 error
    fmul.dd f4,     f6,     f8  // avoid using f6 as src1
    fsub.dd f10,    f8,     f8  // 2 - guess * divisor
    fmul.dd f6,     f2,     f6  // guess * dividend
    fmul.dd f8,     f6,     f8  // result = third guess * dividend
```

Example 9-4. Double-Precision Divide

- Stop Here

Example: Newton for $\sqrt{A}$

A common usage for Newton iteration is in computation of square roots. Suppose we want to find $x^* = \sqrt{A}$, which can be expressed as the root-finding problem:

$$f(x) \quad = \quad x^2 - A. \tag{1}$$

Applying Newton's method to generate a fixed point scheme $x_{k+1} = g(x_k)$, we have

$$g(x) \quad = \quad x - \frac{f}{f'} = x - \frac{x^2 - A}{2x} = \frac{1}{2}\left(x + \frac{A}{x}\right).$$

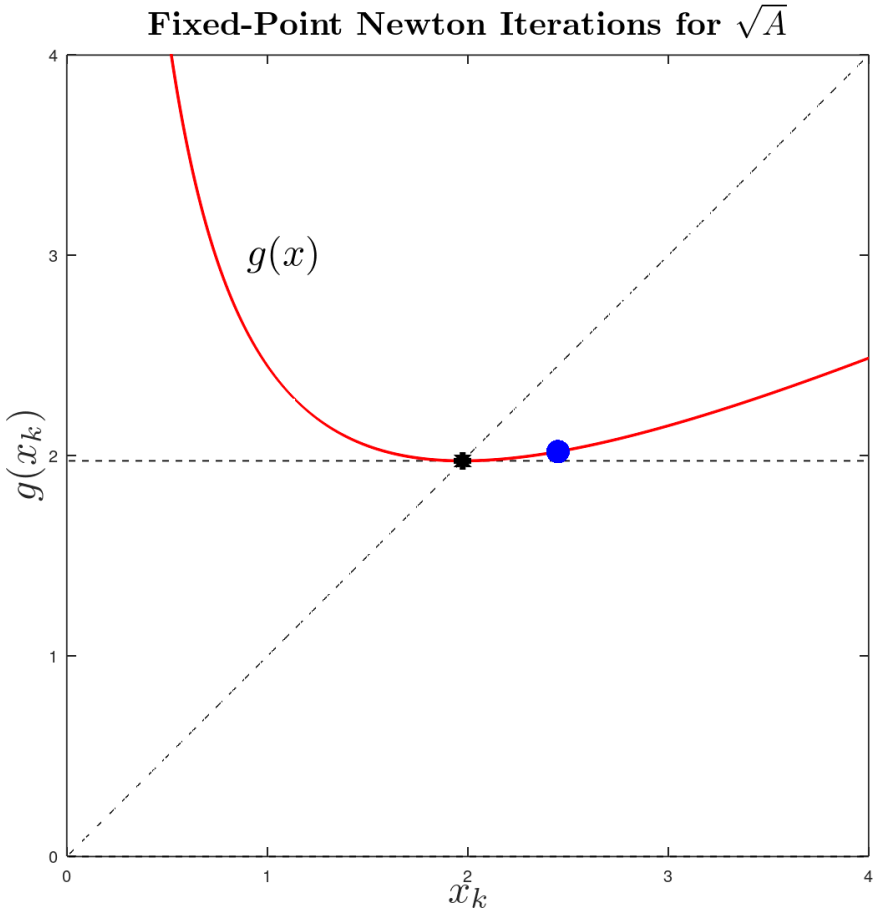
This is a very well-known scheme and is globally convergent. Assuming $A > 0$, we establish the latter claim as follows. Note that if $x_0 < x^*$ then $x_1 > x^*$. Moreover,

$$g'(x) \quad = \quad \frac{1}{2}\left(1 - \frac{A}{x^2}\right)$$

is between 0 and 1/2 for all $x > x^*$, so each iteration yields a contraction in the error for any $x_k > x^*$. Quadratic convergence results because $g'(x^*) = 0$.

Example: Newton for $\sqrt{A}$

- Because $g(x)$ is concave up with slope 0 at x^*, every value satisfies $g(x) > x^*$
- Therefore, for *any* $x_0 > 0$, we have $x_1 > x^*$.
- For $x_k > x^*$, $g'(x_k) \in [0 : 1/2]$, so we gain at least one bit per iteration for $k > 1$.



Generating a Good Initial Guess for $A^{1/2}$

A good initial guess x_0 can be obtained through *range reduction*, in which one maps the problem to a suitable range over which the error is known. Range reduction for the square-root problem begins by exploiting the binary representation of A,

$$\begin{aligned} A &= 1.bbb\ldots \times 2^k \\ &= Bb.bbb\ldots \times 2^l. \end{aligned}$$

In the second expression, the mantissa is normalized (via a shift) onto the interval $[1, 4)$ so that the exponent l is even. That is, if k is even, take B=0 and $l = k$. If k is odd, take $B = 1$ and $l = k - 1$. The exponent of x^* is thus $l/2$, which is effected as a bit shift to the right, with no information loss. The mantissa of x^* is the square-root of the normalized mantissa and will be on $[1, 2)$, such that the result will be normalized.

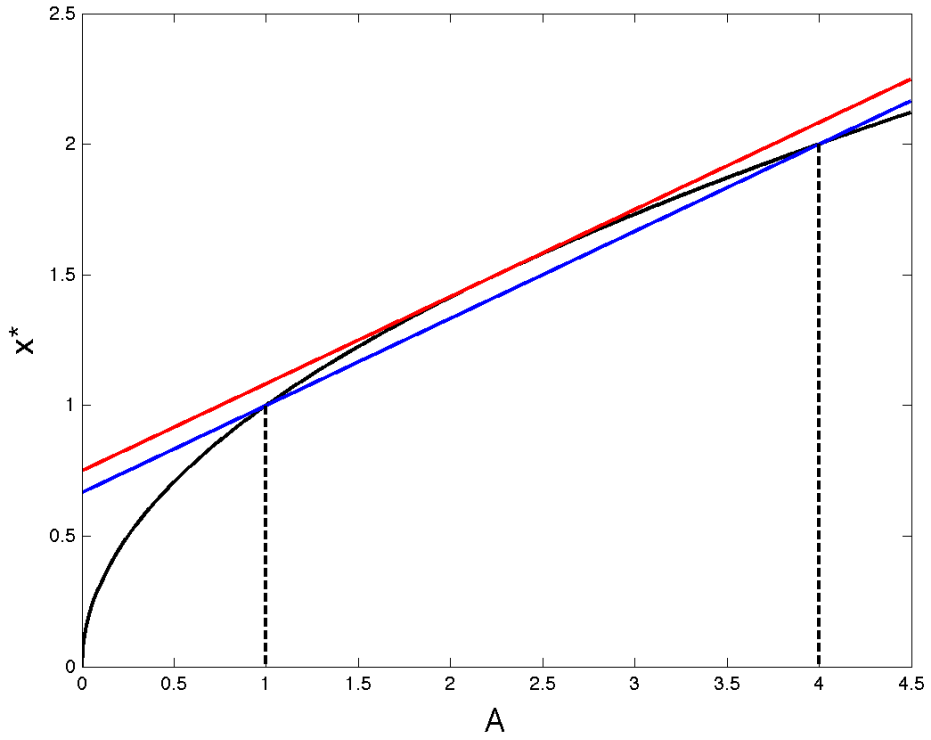


Figure 1: Plot of $x = \sqrt{A}$, $y_1 = 2/3 + A/3$, and $y_2 = 3/4 + A/3$.

Generating a Good Initial Guess for $A^{1/2}$

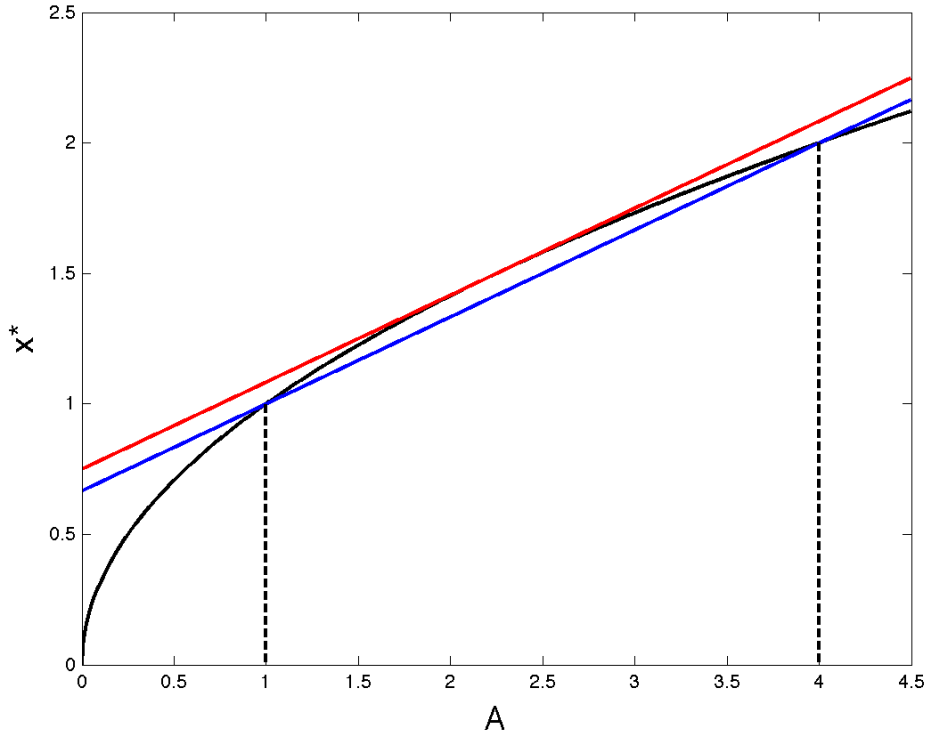


Figure 1: Plot of $x = \sqrt{A}$, $y_1 = 2/3 + A/3$, and $y_2 = 3/4 + A/3$.

There are many ways to generate a good initial guess for the mantissa. Without loss of generality, assume $A \in [1, 4)$. Figure 1 shows $x* = \sqrt{A}$ along with two lines, $y_1(A) = (2 + A)/3$ and $y_2(A) = 3/4 + A/3$, which bound x^* on the interval $[1, 4]$. The gap between y_1 and y_2 is $1/12$. Taking the average of these two lines, let

$$x_0 = \frac{17}{24} + \frac{A}{3}. \tag{2}$$

We know that $|x_0 - x^*| \leq 1/24 \approx .04$ and, with quadratic convergence, we can anticipate about 4 iterations to reduce the error to below ϵ_M.

Example: Newton for $\sqrt{A}$

A common usage for Newton iteration is in computation of square roots. Suppose we want to find $x^* = \sqrt{A}$, which can be expressed as the root-finding problem:

$$f(x) \quad = \quad x^2 - A. \tag{1}$$

Applying Newton's method to generate a fixed point scheme $x_{k+1} = g(x_k)$, we have

$$g(x) \quad = \quad x - \frac{f}{f'} = x - \frac{x^2 - A}{2x} = \frac{1}{2}\left(x + \frac{A}{x}\right).$$

This is a very well-known scheme and is globally convergent. Assuming $A > 0$, we establish the latter claim as follows. Note that if $x_0 < x^*$ then $x_1 > x^*$. Moreover,

$$g'(x) \quad = \quad \frac{1}{2}\left(1 - \frac{A}{x^2}\right)$$

is between 0 and 1/2 for all $x > x^*$, so each iteration yields a contraction in the error for any $x_k > x^*$. Quadratic convergence results because $g'(x^*) = 0$.

- Can you find a Newton method for $\sqrt{A}$ that does not require division on each iteration??

Example: Faster Newton for $\sqrt{A}$

- The problem with the classic fixed point iteration

$$g(x_k) = \frac{1}{2}\left(x + \frac{A}{x}\right)$$

  is that it requires evaluation of A/x on *each iteration*

- As we've seen, division also requires a Newton iteration

- A method that avoids repeated division can be found by starting with

$$f = \frac{A}{x^2} - 1 = Ax^{-2} - 1$$

$$f' = -2x^{-3}A$$

$$\frac{f}{f'} = \frac{x^3}{2A} - \frac{x}{2}$$

Example: Faster Newton for $\sqrt{A}$

- The resulting fixed-point iteration is based on

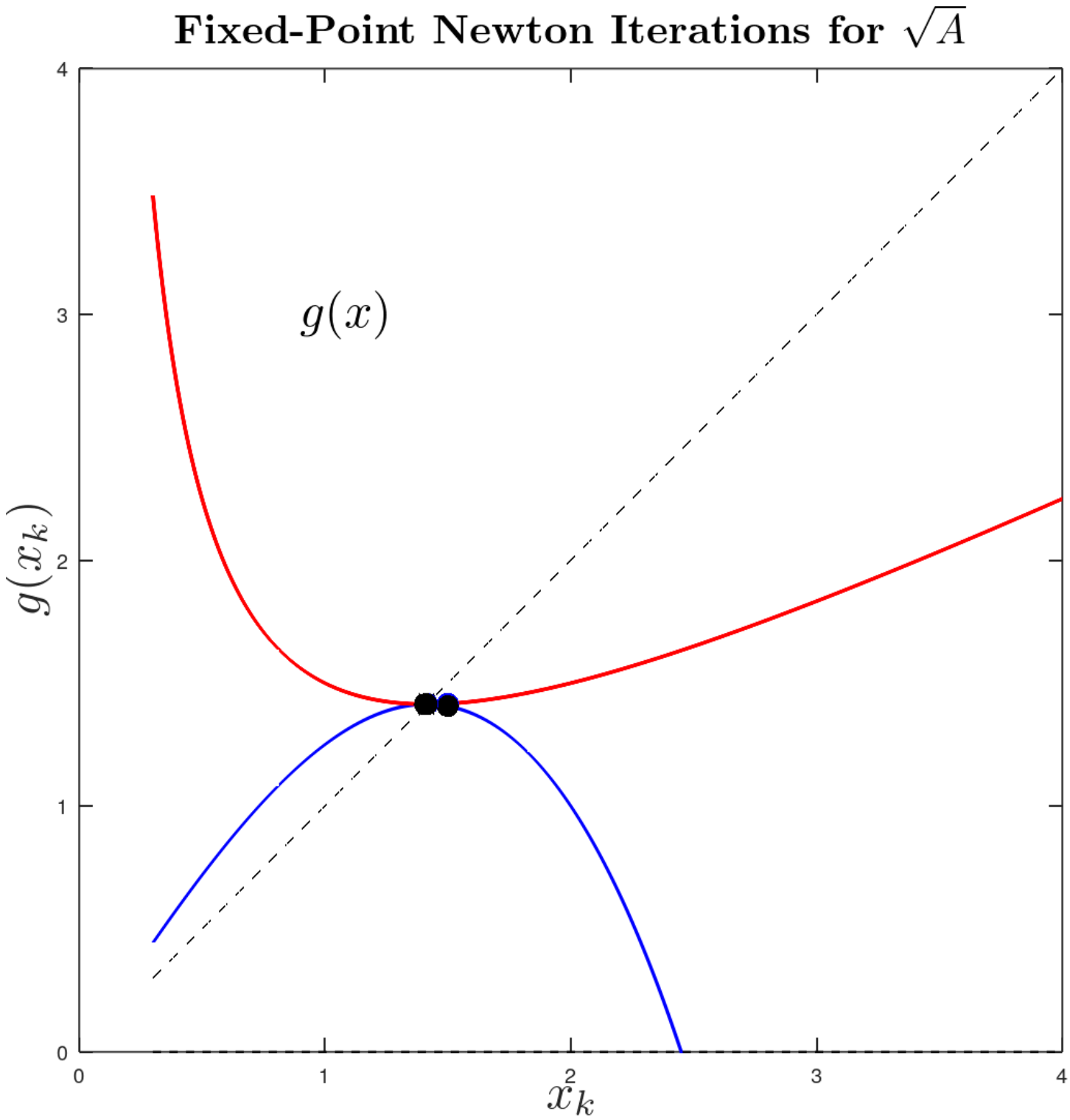
$$g(x) = x - \frac{f}{f'} = \frac{3}{2}x - \frac{x^3}{2A} = (c_1 - c_2 x^2)x,$$

  which requires one addition and three multiplies after an initial computation of $(2A)^{-1}$

- Unlike the classic iteration, this new one, which is polynomial in x, is not globally convergent

- However, after range reduction to $[1:4]$, it is convergent for the initial guess of the preceding slides or even for something as simple as $x_0 = .5(1 + A)$

sqrt_bad_init.m



sqrt_bad_init.m

Returning to Newton

- Recall main result

$$g(x) \;=\; x - \frac{f}{f'}$$

- At root, $f(x^*) = 0$, so

$$g(x^*) \;=\; x^* - \frac{f(x^*)}{f'(x^*)} \;=\; x^*$$

  provided $f'(x^*) \neq 0$.

- Moreover,

$$g'(x) \;=\; 1 - \frac{(f')^2 - f\,f''}{(f')^2}$$

$$g'(x^*) \;=\; 1 - \frac{(f')^2}{(f')^2} \;=\; 0 \quad \text{if } f'(x^*) \neq 0$$

- So convergence is quadratic if $f'(x^*) \neq 0$

Newton, continued

- What if $f'(x^*) = 0$?
- Assume $f''(x^*) \neq 0$. Then,

$$g(x^*) \;=\; x^* - \underbrace{\frac{f(x^*)}{f'(x^*)}}_{?}$$

- Because $f(x^*) = f'(x^*) = 0$, we have to use L'hôpital's Rule:

$$\lim_{x \longrightarrow x^*} \frac{f}{f'} \;=\; \lim_{x \longrightarrow x^*} \frac{f'}{f''} \;=\; 0$$

- So, yes, still a *fixed-point* because $g(x^*) = x^*$
- What about convergence?
- Does $g'(x^*) = 0$ in this case?

Newton, continued

- Here, we need to evaluate $g'(x^*)$,

$$g'(x) = 1 - \frac{(f')^2 - f\,f''}{(f')^2} = \frac{f\,f''}{(f')^2}$$

$$\lim_{x \longrightarrow x^*} g'(x) = \lim_{x \longrightarrow x^*} \frac{f\,f''}{(f')^2} = \frac{0}{0}$$

- L'hôpital again,

$$\lim_{x \longrightarrow x^*} g'(x) = \lim_{x \longrightarrow x^*} \frac{\frac{d}{dx}(f\,f'')}{\frac{d}{dx}(f')^2} = \lim_{x \longrightarrow x^*} \frac{f'f'' + f\,f'''}{2f'f''} = \frac{0}{0}$$

$$= \lim_{x \longrightarrow x^*} \frac{1}{2} \frac{f''^2 + f'f''' + f'f''' + f\,f''''}{(f'')^2 + f'f'''}$$

$$= \lim_{x \longrightarrow x^*} \frac{1}{2} \frac{f''^2}{f''^2} = \frac{1}{2} \neq 0$$

- More generally, if multiplicity is m, convergence is **linear**, with constant

$$C = 1 - \frac{1}{m}$$

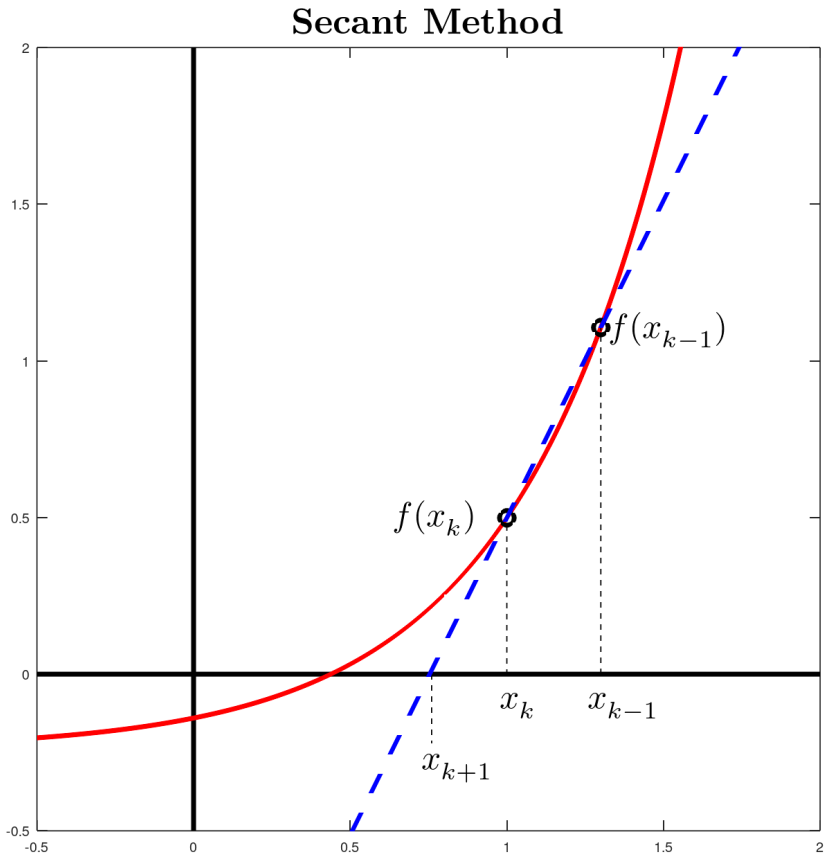
Convergence is slower with increasing m

Alternatives to Newton

- Main issue with Newton iteration is the need for $f'(x)$
- Often, only know $f(x)$, maybe only as a function call (i.e., $f(x)$ is a "black box")
- Several methods exist for constructing a model (i.e., *interpolant*) that approximates $f(x)$ and passes through $[x_k, f(x_k)]$ pairs.
- One can then approximate $f'(x_k)$ or simply use the interpolant to find approximate root, x_{k+1}
- Common choices:
  - Secant method
  - Muller's method
  - Inverse interpolation
  - Linear fractional iteration
- We'll look at three of these

Secant Method

- By far the most popular nonlinear solver for $f(x) = 0$ is the *secant method*
- It is similar to Newton in that it uses slope information and converges superlinearly, but it does not require evaluation of $f'(x)$
- Instead, on approximates the derivative as $f'(x_k) \approx \frac{\Delta f}{\Delta x} = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$



Secant Method, continued

- The resulting secant method is

$$x_{k+1} = x_k - \frac{f(x_k)}{\Delta f/\Delta x} = x_k - \frac{f(x_k)}{f(x_k) - f(x_{k-1})}(x_k - x_{k-1})$$

- Here, you need *two* starting values x_0 and x_1

- Don't make these too close or round-off (cancellation) may affect the result

Example: Secant Method

demo_sec.m

- We apply the secant method to $f(x) = x^2 - 4\sin(x)$

```
for k=1:15;
   dx = x1-x0;
   disp([k x1 f1 dx])
   if abs(f1) < tol; break; end;
   if abs(dx) < tol; break; end;

   [x1,x0,f1,f0]=secant(x1,x0,f1,f0);

   plot(x0,f0,'k.',ms,12,x1,f1,'r.',ms,12); pause(.4);

end;
```

```
function [x1,x0,f1,f0]=secant(x1,x0,f1,f0);

dx = x1-x0; df = f1-f0;

xnew = x1 - f1*dx/df;

x0=x1; x1=xnew;
f0=f1; f1=fnc(xnew);
```

```
          k                   x_k                   f_k                 x_k-x_{k-1}
 1.000000000000000e+00   2.000000000000000e+00   3.628102926972732e-01   1.000000000000000e+00
 2.000000000000000e+00   1.867038861132927e+00  -3.399264892893270e-01  -1.329611388670726e-01
 3.000000000000000e+00   1.931354568387107e+00  -1.266962609341338e-02   6.431570725418001e-02
 4.000000000000000e+00   1.933844526748519e+00   4.799532782651106e-04   2.489958361411304e-03
 5.000000000000000e+00   1.933753644474301e+00  -6.258100517797516e-07  -9.088227421760742e-05
 6.000000000000000e+00   1.933753762821192e+00  -3.082512023411255e-11   1.183468905097129e-07
 7.000000000000000e+00   1.933753762827021e+00  -4.440892098500626e-16   5.829559057701772e-12
```

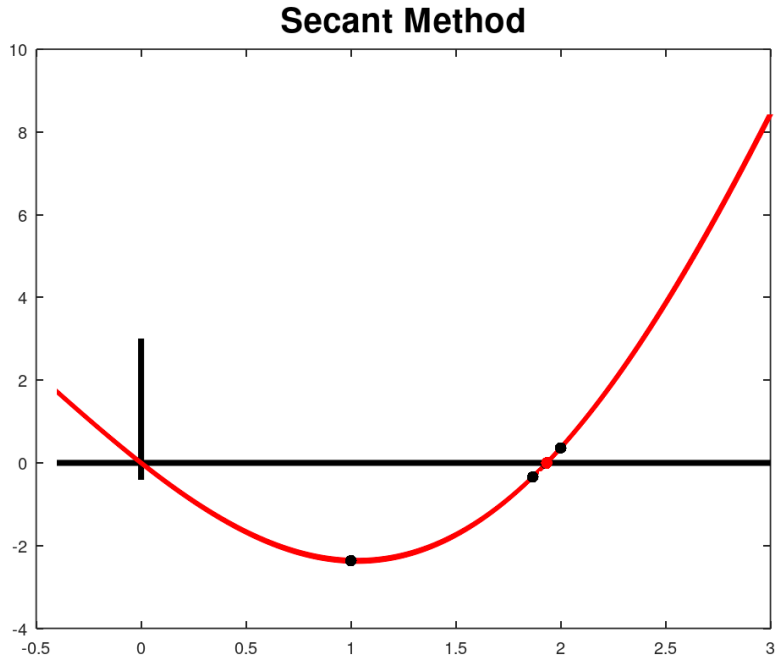
Example: Secant Method

demo_sec.m

- We apply the secant method to $f(x) = x^2 - 4\sin(x)$

```
for k=1:15;
   dx = x1-x0;
   disp([k x1 f1 dx])
   if abs(f1) < tol; break; end;
   if abs(dx) < tol; break; end;

   [x1,x0,f1,f0]=secant(x1,x0,f1,f0);

   plot(x0,f0,'k.',ms,12,x1,f1,'r.',ms,12); pause(.4);

end;
```

```
function [x1,x0,f1,f0]=secant(x1,x0,f1,f0);

dx = x1-x0; df = f1-f0;

xnew = x1 - f1*dx/df;

x0=x1; x1=xnew;
f0=f1; f1=fnc(xnew);
```



Example: Secant Method

- For reference, here compare to the Newton results:

```
octave:9> demo_sec
            k                   x_k                   f_k                 x_k-x_{k-1}
   1.000000000000000e+00   2.000000000000000e+00   3.628102926972732e-01   1.000000000000000e+00
   2.000000000000000e+00   1.867038861132927e+00  -3.399264892893270e-01  -1.329611388670726e-01
   3.000000000000000e+00   1.931354568387107e+00  -1.266962609341338e-02   6.431570725418001e-02
   4.000000000000000e+00   1.933844526748519e+00   4.799532782651106e-04   2.489958361411304e-03
   5.000000000000000e+00   1.933753644474301e+00  -6.258100517797516e-07  -9.088227421760742e-05
   6.000000000000000e+00   1.933753762821192e+00  -3.082512023411255e-11   1.183468905097129e-07
   7.000000000000000e+00   1.933753762827021e+00  -4.440892098500626e-16   5.829559057701772e-12
octave:10> demo_newt
            k                   x_k                   f_k                 x_k-x_{k-1}
   1.000000000000000e+00   2.000000000000000e+00   3.628102926972732e-01   1.000000000000000e+00
   2.000000000000000e+00   1.935951152215635e+00   1.163292318786135e-02  -6.404884778436526e-02
   3.000000000000000e+00   1.933756376157758e+00   1.381844935011145e-05  -2.194776057877101e-03
   4.000000000000000e+00   1.933753762830728e+00   1.959898909831281e-11  -2.613327029887813e-06
   5.000000000000000e+00   1.933753762827021e+00  -4.440892098500626e-16  -3.706590590013548e-12
```

- We see that, as expected, the convergence is superlinear, but not as fast as Newton

Secant Method: Rate of Convergence

- Note, the secant method is not a fixed-point iteration as it requires two function evaluations for the update
- As $k \longrightarrow \infty$, the *asymptotic* error behavior for the secant method for a simple root is

$$e_{k+1} \sim \frac{1}{2}\frac{f''(x^*)}{f'(x^*)} e_k e_{k-1} = c e_k e_{k-1}$$

- We sketch the derivation as follows.
- Near the root, assume

$$f = ax^2 + \tilde{b}x + c = a(x - x^*)^2 + b(x - x^*) = a\delta^2 + b\delta$$

with $\delta := x - x^*$

- Note that $a = \frac{1}{2}f''(x^*)$ and $b = f'(x^*)$

Secant Method: Rate of Convergence

- This approximation is simply retaining the first 3 terms in the Taylor series expansion of f at x^*

- Define successive iterates as x_0, x_1, and (to be determined) x_2, with corresponding δ's, δ_0, δ_1, and δ_2

- The secant update step is then

$$\begin{aligned}\delta_2 &= \delta_1 - (a\delta_1^2 + b\delta_1)\frac{\delta_1 - \delta_0}{a(\delta_1^2 - \delta_0^2) + b(\delta_1 - \delta_0)} \\ &= \delta_1 - \frac{a\delta_1^2 + b\delta_1}{a(\delta_1 + \delta_0) + b}\end{aligned}$$

- Now use the result that $\delta_1 \ll \delta_0$, retain only $b\delta_1$ in the numerator, and use the expansion, $\frac{1}{1+\epsilon} \sim 1 - \epsilon + \epsilon^2 \cdots$, to find

$$\begin{aligned}\delta_2 &\sim \delta_1 \left[\frac{a}{b}(\delta_1 + \delta_0)\right] + O(\delta_1^2) \\ &\sim \frac{a}{b}\delta_1\delta_0 + O(\delta_1^2)\end{aligned}$$

Secant Method: Rate of Convergence

- We've now established

$$e_{k+1} \sim \frac{1}{2}\frac{f''(x^*)}{f'(x^*)} e_k e_{k-1} = C e_k e_{k-1}$$

- What about the *convergence rate*, r, in the expression for the asymptotic convergence

$$\frac{|e_{k+1}|}{|e_k|^r} \sim C$$

- As $k \longrightarrow \infty$, we have

$$e_{k+1} \sim c\, e_k e_{-1} \sim C e_k^r$$

- For simplicity, assume errors are positive

$$C \sim \frac{e_{k+1}}{e_k^r} \sim c\frac{e_k e_{k-1}}{e_k^r}, \qquad C e_{k-1}^r = e_k$$

$$\sim \frac{c}{C}\frac{e_k e_k^{\frac{1}{r}}}{e_k^r} = \frac{c}{C} e_k^{1+\frac{1}{r}-r} = \text{constant, } C$$

Secant Method: Rate of Convergence

- Both sides of the preceding expression are constant,

$$C \sim \frac{c}{C} e_k^{1+\frac{1}{r}-r},$$

- If this is a constant, independent of k, then the exponent $1 + \frac{1}{r} - r$ must equate to 0, which implies that r is the positive root of the quadratic equation

$$r^2 - r - 1 = 0$$

- The solution is the golden ratio,

$$r = \frac{1+\sqrt{5}}{2} \approx 1.618$$

- Thus, convergence of the secant method is *superlinear*, with rate $r = 1.618...$

Muller's Method

- Muller's method is a natural extension of the secant method but (as we will see) is not as robust as the superior *quadratic inverse iteration* (or simply "inverse iteration")

- Nonetheless, Muller's method has one attractive feature that warrants attention, and that is that it can find *complex roots*

- The idea behind Muller's metho is to use three successive function pairs $[x_k, f(x_k)]$ to fit a parabola to approximate $f(x)$ and to then find the nearby root of this parabola.

- When it works, the convergence rate of Muller's method is $r \approx 1.839$, the same as inverse iteration

- We will see in the following demos that it is not as robust as secant or inverse iteration

Muller's Method

demo_mul.m

```
x0=2.0;   f0=fnc(x0);
x1=1.5;   f1=fnc(x1);
x2=1.0;   f2=fnc(x2);
plot(x0,f0,'k.',ms,12,x1,f1,'k.',ms,12,x2,f2,'b.',ms,8);

for k=1:15;
   dx = x2-x1;
   disp([k x2 f2 dx])
   if abs(f2) < tol; break; end;
   if abs(dx) < tol; break; end;

   [x2,x1,x0,f2,f1,f0]=muller(x2,x1,x0,f2,f1,f0);

   plot(x0,f0,'k.',ms,12,x1,f1,'k.',ms,12,x2,f2,'b.',ms,8);

end;
```

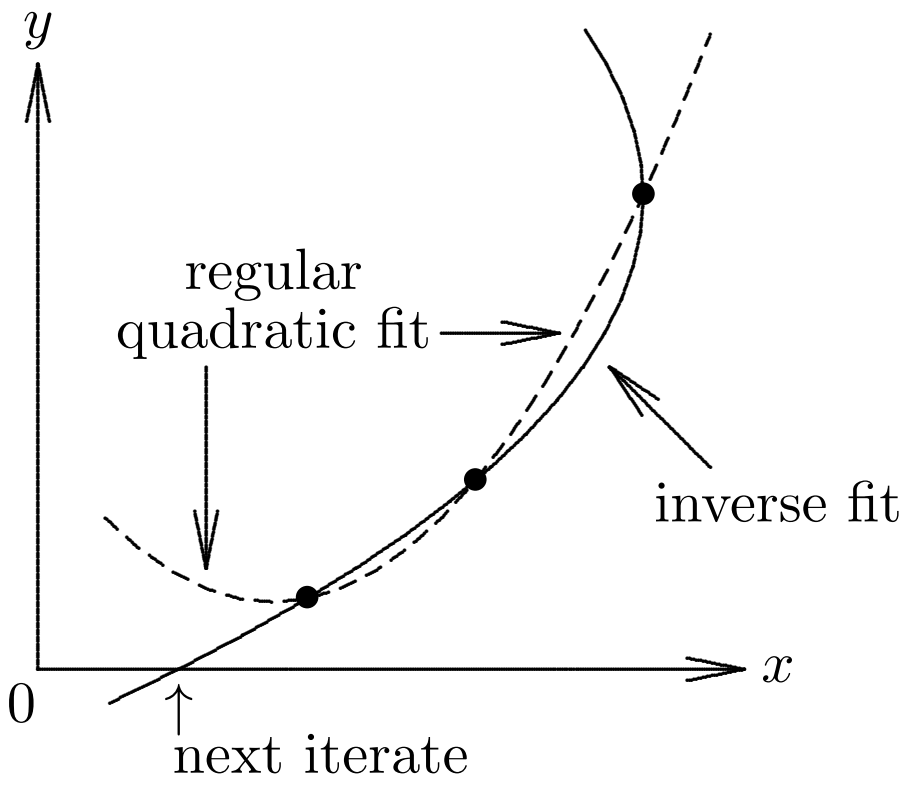
```
%% Update step for Muller's Method
%%
%% Following Wofram Mathworld Implementation

function [x2,x1,x0,f2,f1,f0]=muller(x2,x1,x0,f2,f1,f0);

h2=x2-x1;  %% recent
h1=x1-x0;  %% older

q=h2/h1;  %% recent/older
q1=q+1;

A=q*(f2-q1*f1+q*f0);
B=(q+q1)*f2-q1*q1*f1+q*q*f0;
C=q1*f2;

Dp = B-sqrt(B^2-4*A*C);
Dm = B+sqrt(B^2-4*A*C);
D  = max(Dp,Dm);   %% Wolfram MathWorld suggestion
D  = min(Dp,Dm);   %% OPPOSITE of Wolfram MathWorld suggestion
xnew = x2-h2*(2*C)/D;

x0=x1; x1=x2; x2=xnew;
f0=f1; f1=f2; f2=fnc(xnew);
```

Inverse Quadratic Interpolation

- With *inverse quadratic interpolation*, the idea is to fit a curve of the form $x = x(y)$ which is quadratic in y

- One then evaluates that curve at $y = 0$ to get the next iterate, x_{k+1}.

- We can see how this compares to Muller's method (dashed-line) in the figure from the tex, below



Inverse Quadratic Interpolation

- If f_c, f_a, f_b, are three successive function evaluations at (c, a, b), then the update is given by,

$$c' = a, \qquad a' = b, \qquad b' = b + p/q,$$

where p and q are

$$p = v(w(u-w)(c-b) - (1-u)(b-a), \qquad q = (w-1)(v-1)(u-1)$$

and $u = f_b/f_c$, $v = f_b/f_a$, and $w = f_a/f_c$

- For the test problem of $f = x^2 - 4\sin(x)$ we have the results below, which are superlinear but not as fast as Muller for this particular case.

- Generally, however, inverse iteration is more robust than Muller

```
octave:13> demo_inv
             k                    x_k                    f_k                 x_k-x_{k-1}
   1.000000000000000e+00   1.000000000000000e+00  -2.365883939231586e+00  -5.000000000000000e-01
   2.000000000000000e+00   2.043533508306118e+00   6.147279280058893e-01   1.043533508306118e+00
   3.000000000000000e+00   2.105473554891674e+00   9.912863030585370e-01   6.194004658555574e-02
   4.000000000000000e+00   1.908724959689660e+00  -1.305426952747446e-01  -1.967485952020138e-01
   5.000000000000000e+00   1.933511046619406e+00  -1.283234075938555e-03   2.478608692974582e-02
   6.000000000000000e+00   1.933754349354884e+00   3.101366593316612e-06   2.433027354777906e-04
   7.000000000000000e+00   1.933753762829330e+00   1.220801237877822e-11  -5.865255536807723e-07
   8.000000000000000e+00   1.933753762827021e+00  -4.440892098500626e-16  -2.308819802010476e-12
```

Inverse Quadratic Interpolation

demo_inv.m

```
hdr

tol = 100*eps;
x=-.4:.001:3;
f=fnc(x);

hold off;
plot(x,0*x,'k-',lw,2,0*x,x,'k-',lw,2,x,f,'r-',lw,2);
title('Quadratic Inverse Iteration',fs,20);
hold on;

format longe;
disp('   k      x_k      f_k       x_k-x_{k-1}')

x0=2.0;   f0=fnc(x0);
x1=1.5;   f1=fnc(x1);
x2=1.0;   f2=fnc(x2);
plot(x0,f0,'k.',ms,12,x1,f1,'k.',ms,12,x2,f2,'b.',ms,8);

for k=1:15;
   dx = x2-x1;
   disp([k x2 f2 dx])
   if abs(f2) < tol; break; end;
   if abs(dx) < tol; break; end;

   [x2,x1,x0,f2,f1,f0]=inverse(x2,x1,x0,f2,f1,f0);

   plot(x0,f0,'k.',ms,12,x1,f1,'k.',ms,12,x2,f2,'b.',ms,8);

end;
```

```
function [x2,x1,x0,f2,f1,f0]=inverse(x2,x1,x0,f2,f1,f0);

% Following Heath, p.234:

% Generate: new a = old b
%           new c = old a
%           new b = b + p/q
%           old c is discarded

% Initially:   x0=c
%              x1=a
%              x2=b

c=x0; fc=f0;
a=x1; fa=f1;
b=x2; fb=f2;

hc=c-b;
hb=b-a;

u=fb/fc; v=fb/fa; w=fa/fc;

p=v*( w*(u-w)*hc - (1-u)*hb );
q=(w-1)*(v-1)*(u-1);

xnew = b + (p/q);
x0=x1; f0=f1;
x1=x2; f1=f2;
x2=xnew; f2=fnc(xnew);

function f=fnc(x);

f = x.*x-4*sin(x);
```

Other Test Cases

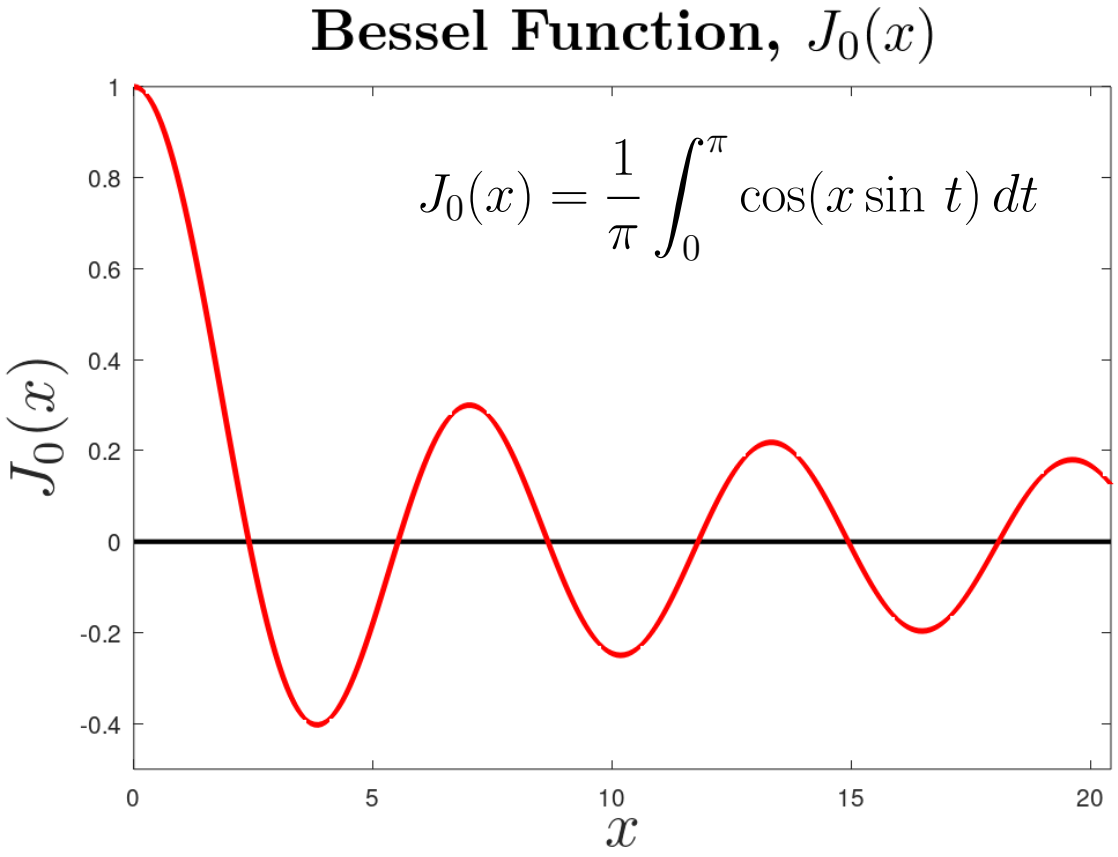
- We can explore secant, Muller, and inverse iteration on other functions and starting conditions

- One case is $f = e^{x/2} - 2$

Stopped Here

Finding Multiple Roots

- Often we need to find more than one root of a scalar function $f(x)$
- We may have a precise definition of the function but the particular zeros of interest are not known or tabulated .
- Examples include:
  - Chebyshev polynomials
  - Legendre polynomials
  - Bessel functions
- The Bessel function is an example of a relatively expensive function to evaluate, so rapid root finders are of interest



Finding Multiple Roots

- Zeros of these functions are of interest for many reasons

- The Chebyshev zeros are central to many approximation problems because their monomial product $T_n(x) = (x - r_1)(x - r_2) \cdots (x - r_n)$ (the nth-order Chebyshev polynomial) has the property that all of its extrema on the standard $[-1, 1]$ interval are ± 1.

- Out of all nth-order polynomials, only T_n has this property.

- If one can bound an error by $\max |p_n(x)|$ for any nth-order polynomial, then the *minimal error bound* will be realized by T_n.

- This is the mini-max property that is invariably linked to some form of Chebyshev polynomial

- Fortunately, the roots of $T_n(x)$ are known in closed form:

$$r_j = \cos(j\theta - \theta/2), \quad \theta := \pi/n$$

Finding Multiple Roots

- For Legendre polynomials and Bessel functions the zeros are not availabe in closed form.

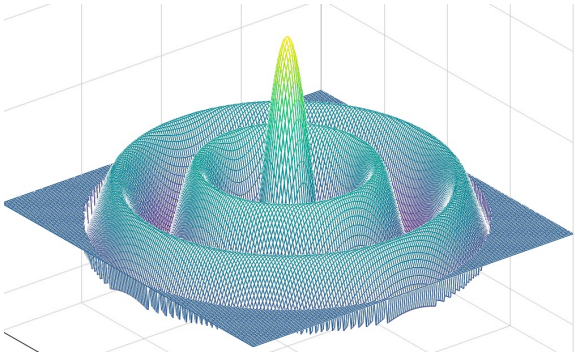
- The Legendre zeros are important for numerical integration.

- One robust way to find them is to develop a companion matrix that exploits the 3-term recurrence, starting with $P_0 = 1$ and $P_1 = x$

$$P_{n+1}(x) = \frac{2n+1}{n+1} x P_n(x) - \frac{n}{n+1} P_{n-1}(x)$$

  and to solve for the eigenvalues of the $n \times n$ tridiagonal companion matrix that results from this relationship

- The zeros of Bessel functions correspond to eigenvalues of functions constrained to disk shapes, such as drum heads and the liquid surface of a cup of coffee



bessel_surf.m

Main Ingredients for Multiple Roots

- **Safeguards**
  - bracketing or other known limits (e.g., roots $r_i \in [-1, 1]$ for orthogonal polynomials)
  - Brent's (also Van Wijngaarden-Dekker) method combines bracketing/bisection with inverse quadratic interpolation

Numerical Recipes, SciPy, etc.

- **Good initial guess, if possible**
  - roots of Legendre polynomials are close to Chebyshev roots
  - For Bessel function, J_0, successive roots satisfy $r_{i+1} \sim r_i + \pi$ as i increases
  - Start with first root near 0, then set initial guess for r_2 to be $r_1 + \pi$

Main Ingredients for Multiple Roots

Deflation

- A key point is to ensure that the root finder does not return to known roots
- If $f(x)$ has say, 3 roots of interest, idea is to use one of the methods discussed so far (e.g., inverse quadratic interpolation) to find a root, denoted as r_1
- Define new function

$$\hat{f}(x) = \frac{f(x)}{(x - r_1)}$$

  and apply root finder to get r_2
- Next define

$$\hat{f}(x) = \frac{f(x)}{(x - r_1)(x - r_2)}$$

  and apply root finder to get r_3

```
function f=fnc_deflate(x,rts);

f=fnc(x);

nrts = length(rts);

for k=1:nrts;
    f=f./(x-rts(k));
end;
```

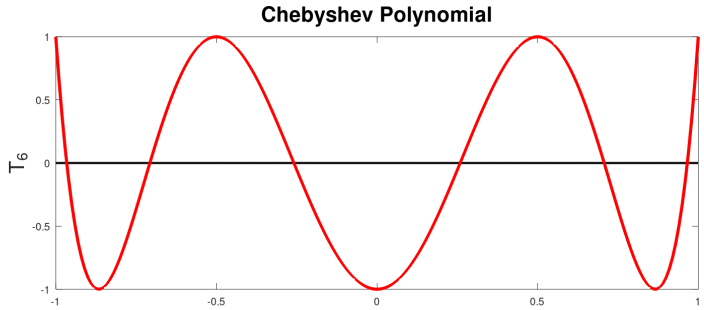
Code Example

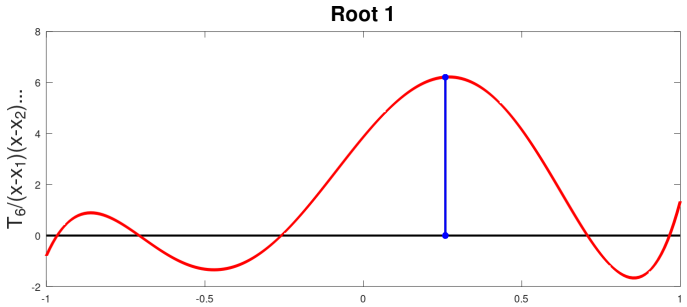
```
hdr; format shorte; clear rts;

tol = 100*eps;

N=6;  %% Number of roots (must match "fnc")

x0=0.0;   f0=fnc(x0); % Chebyshev roots are on (-1:1)
x1=0.1;   f1=fnc(x1);
x2=0.2;   f2=fnc(x2);

rts=[];
for j=1:N
   for k=1:30;
      dx = x2-x1;
      disp([j k x2 f2 dx])
      if abs(f2) < tol || abs(dx) < tol; break; end;
      [x2,x1,x0,f2,f1,f0]=inverse_def(x2,x1,x0,f2,f1,f0,rts);
   end;
   disp(' ');

   rts(j) = x2;

   if j<N; x0=0.01; x1=0.02; x2=0.03;  % Reinitialize guesses
     f0=fnc_deflate(x0,rts);
     f1=fnc_deflate(x1,rts);
     f2=fnc_deflate(x2,rts);
     if f2 > f0; t=f2;f2=f0;f0=t; t=x2;x2=x0;x0=t; end;
   end;

end;

rts=reshape(rts,N,1); rts=sort(rts,'descend');

%% EXACT ROOTS FOR Nth-Order Chebyshev Polynomial

theta = pi/N; j=[1:N]'; exact = cos(j*theta - theta/2);
```

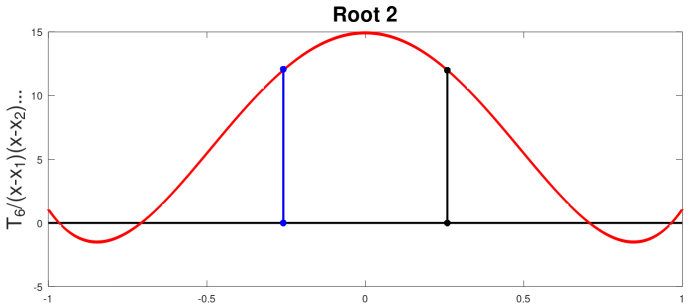
Code Example

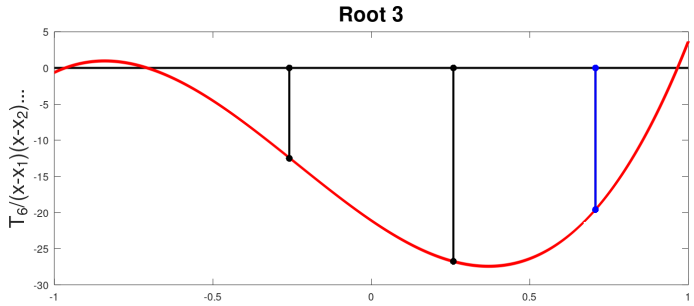
```
hdr; format shorte; clear rts;

tol = 100*eps;

N=6;  %% Number of roots (must match "fnc")

x0=0.0;   f0=fnc(x0); % Chebyshev roots are on (-1:1)
x1=0.1;   f1=fnc(x1);
x2=0.2;   f2=fnc(x2);

rts=[];
for j=1:N
   for k=1:30;
      dx = x2-x1;
      disp([j k x2 f2 dx])
      if abs(f2) < tol || abs(dx) < tol; break; end;
      [x2,x1,x0,f2,f1,f0]=inverse_def(x2,x1,x0,f2,f1,f0,rts);
   end;
   disp(' ');

   rts(j) = x2;

   if j<N; x0=0.01; x1=0.02; x2=0.03;  % Reinitialize guesses
     f0=fnc_deflate(x0,rts);
     f1=fnc_deflate(x1,rts);
     f2=fnc_deflate(x2,rts);
     if f2 > f0; t=f2;f2=f0;f0=t; t=x2;x2=x0;x0=t; end;
   end;

end;

rts=reshape(rts,N,1); rts=sort(rts,'descend');

%% EXACT ROOTS FOR Nth-Order Chebyshev Polynomial

theta = pi/N; j=[1:N]'; exact = cos(j*theta - theta/2);
```

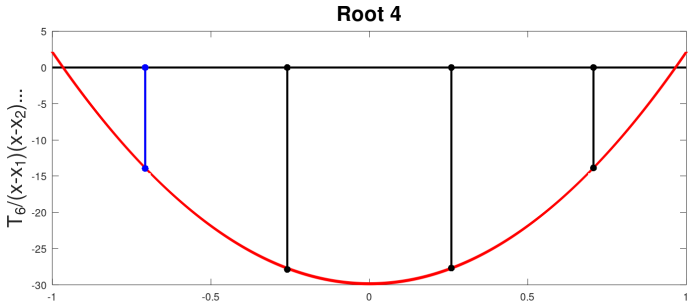
Deflating Roots of Chebyshev Polynomial

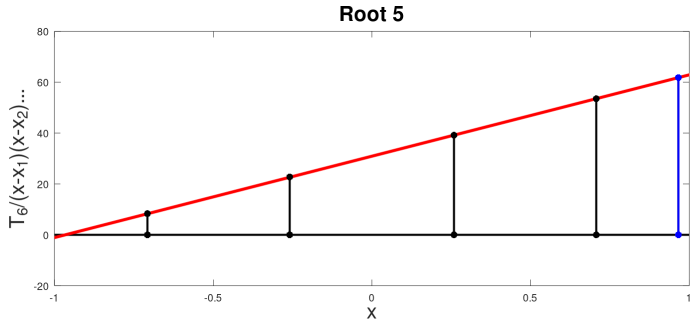


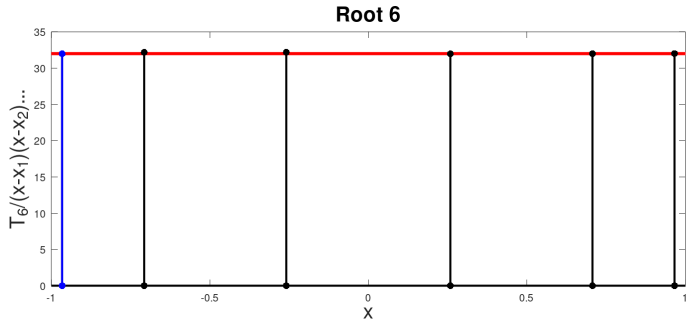












Systems of Nonlinear Equations

Systems of Nonlinear Equations

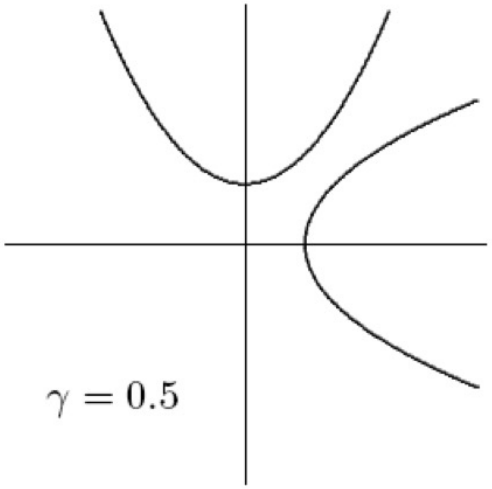
Solving systems of nonlinear equations is much more difficult than the scalar case because

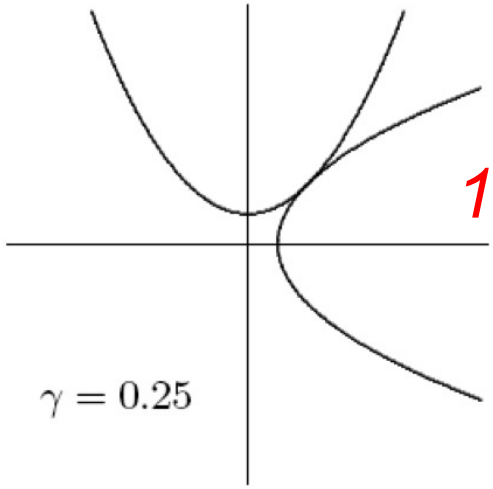
- Wider variety of behavior is possible, so determining existence and number of solutions or good starting guess is more complex
- There is no simple way, in general, to guarantee convergence to desired solution or to bracket solution to produce absolutely safe method
- Computational overhead inreases rapidly with problem dimension, n

Examples: Systems in Two Dimensions

$$\left.\begin{aligned} x_1^2 - x_2 + \gamma &= 0 \\ -x_1 + x_2^2 + \gamma &= 0 \end{aligned}\right\}$$ *A pair of parabolas*

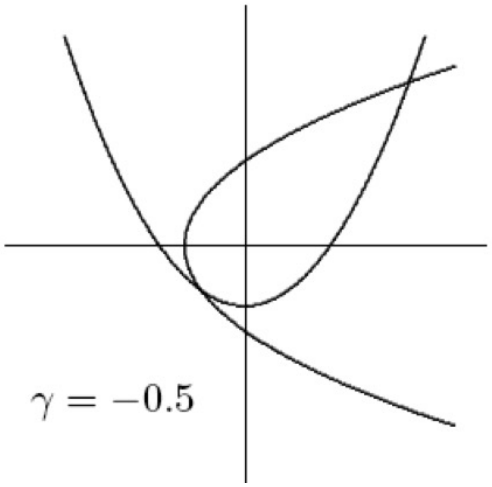
No solution

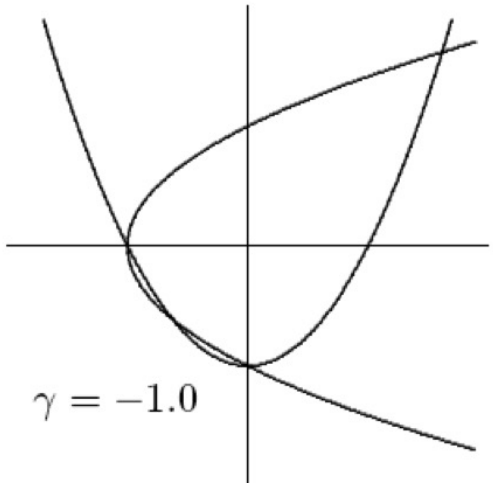




1 solution

2 solutions





4 solutions

Fixed-Point Iteration

- Fixed-point problem for $\mathbf{g}$: $\mathbb{R}^n \longrightarrow \mathbb{R}^n$ is to find vector $\mathbf{x}$ such that

$$\mathbf{x} \;=\; \mathbf{g}(\mathbf{x})$$

- Corresponding *fixed-point iteration* is

$$\mathbf{x}_{k+1} \;=\; \mathbf{g}(\mathbf{x}_k)$$

- If spectral radius $\rho(\mathbf{G}(\mathbf{x}^*)) < 1$, where $\mathbf{G}(\mathbf{x})$ is the Jacobian matrix of $\mathbf{g}$ evaluated at $\mathbf{x}$, then fixed-point iteration converges if started close enough to solution

- Convergence rate is normally linear with constant $C = \rho(\mathbf{G}(\mathbf{x}^*))$

- Since $\rho(\mathbf{G}) \leq \|\mathbf{G}\|$ for any matrix $\mathbf{G}$, can bound ρ by some easy-to-compute norm (e.g., 1- or ∞-norm)

- If $\mathbf{G}(\mathbf{x}^*) = \mathbf{O}$ (the zero matrix) at $\mathbf{x}^*$, then convergence is at least quadratic

Fixed-Point Iteration: Newton's Method

- In $n > 1$ dimensions, Newton's method for $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ takes the form

$$\mathbf{x}_{k+1} \;=\; \mathbf{x}_k \;-\; \mathbf{J}_k^{-1}\mathbf{f}_k,$$

  where $\mathbf{f}_k := \mathbf{f}(\mathbf{x}_k)$ and $\mathbf{J}_k := \mathbf{J}(\mathbf{x}_k)$ is the Jacobian matrix of $\mathbf{f}$ evaluated at $\mathbf{x}_k$,

$$[\mathbf{J}_k]_{ij} \;:=\; \left[\frac{\partial f_i}{\partial x_j}\right]$$

Comments about the Jacobian

- Suppose $\mathbf{f}(\mathbf{x})$ is *linear*, with form

$$\mathbf{f}(\mathbf{x}) \;=\; \mathbf{A}\mathbf{x} \;-\; \mathbf{b}$$

$$[\mathbf{J}]_{ij} \;=\; \frac{\partial \mathbf{f}_i}{\partial x_j}$$

- ith equation:

$$f_i(\mathbf{x}) \;=\; \sum_{j=1}^{n} a_{ij}x_j \;-\; b_i \;=\; \sum_{k=1}^{n} a_{ik}x_k \;-\; b_i$$

$$\frac{\partial f_i}{\partial x_j} \;=\; \sum_{k=1}^{n} a_{ik}\frac{\partial x_k}{\partial x_j} \;-\; \frac{\partial b_i}{\partial x_j}$$

$$=\; a_{ij}$$

Therefore, for linear equations of this form, $\mathbf{J} = \mathbf{A}$

Newton Example: Linear Case

- Suppose $\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}$
- Newton step is

$$\mathbf{x}_{k+1} = \mathbf{g}(\mathbf{x}_k) = \mathbf{x}_k - \mathbf{J}_k^{-1}\mathbf{f}_k$$

- Start with $\mathbf{x}_0 = \mathbf{0}$,

$$\mathbf{f}_0 = \mathbf{A0} - \mathbf{b} = -\mathbf{b}$$

$$\mathbf{J}_0 = \mathbf{A}$$

$$\mathbf{x}_1 = \mathbf{0} - \mathbf{A}^{-1}(-\mathbf{b}) = \mathbf{A}^{-1}\mathbf{b} = \mathbf{x}^*$$

- Newton is just $\mathbf{A}\mathbf{x} = \mathbf{b}$ for this case.

- It will always converge in one (full-Newton) step

What About G?

- Consider the Jacobian of the Newton-derived fixed-point operator, $\mathbf{g}(\mathbf{x})$,

$$G_{ij} = \frac{\partial g_i}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\mathbf{x} - \mathbf{A}^{-1}[\mathbf{A}\mathbf{x} - \mathbf{b}]\right)$$

$$= \frac{\partial}{\partial x_j}\left(\mathbf{x} - \mathbf{A}^{-1}\mathbf{A}\mathbf{x} + \mathbf{A}^{-1}\mathbf{b}\right)$$

$$= \frac{\partial}{\partial x_j}\mathbf{A}^{-1}\mathbf{b} = \mathbf{O}$$

- The last term is zero because $\mathbf{A}^{-1}\mathbf{b}$ is a constant vector that does not depend on $\mathbf{x}$

Simple Example for Nonlinear f

- Consider $\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \sigma e^{\mathbf{x}}$
- This is known as the *Bratu* problem when $\mathbf{A}\mathbf{u} \approx -\nabla^2 u$ and is a mathematical model for reaction-diffusion that governs many combustion processes
- Here,

$$e^{\mathbf{x}} = \begin{pmatrix} e^{x_1} \\ e^{x_2} \\ \vdots \\ e^{x_n} \end{pmatrix}$$

$$\frac{\partial}{\partial x_j} e^{\mathbf{x}} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} e^{x_j} = \mathbf{e}_j \, e^{x_j}, \qquad \mathbf{e}_j = j\text{th col. of } n \times n \text{ identity}$$

$$\mathbf{J} = \mathbf{A} - \sigma \begin{bmatrix} e^{x_1} & & & \\ & e^{x_1} & & \\ & & \ddots & \\ & & & e^{x_n} \end{bmatrix}$$

Jacobian Example

- Nonlinear example:

$$\begin{pmatrix} f_1 \\ f_2 \end{pmatrix} = \begin{pmatrix} e^{x_1 x_2} \\ e^{x_1} + e^{x_2} \end{pmatrix} - \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

- Jacobian:

$$J = \begin{pmatrix} x_2 e^{x_1 x_2} & x_1 e^{x_1 x_2} \\ e^{x_1} & e^{x_2} \end{pmatrix}.$$

Linear Example for g

- Here, we have the Jacobian of $\mathbf{g}(\mathbf{x})$, $[\mathbf{G}]_{ij} = \dfrac{\partial g_i}{\partial x_j}$
- The fixed-point function evaluation, $\mathbf{g}(\mathbf{x})$ is,

$$y_i = [\mathbf{G}\mathbf{x}]_i = \sum_{j=1}^{n} G_{ij} x_j = \sum_{j=1}^{n} \frac{\partial g_i}{\partial x_j} x_j$$

- For this linear case, fixed-point evaluation simplifies to $\mathbf{g}(\mathbf{x}) = \mathbf{G}\mathbf{x}$
- Can analyze error as follows

$$\begin{aligned} \mathbf{x}^* &= \mathbf{g}(\mathbf{x}^*) \\ \mathbf{x}^* &= \mathbf{G}\mathbf{x}^* - \mathbf{b} \\ \mathbf{x}_{k+1} &= \mathbf{G}\mathbf{x}_k - \mathbf{b} \\ \hline \mathbf{e}_{k+1} &= \mathbf{G}\mathbf{e}_k \end{aligned}$$

Linear Example for g, continued

- Have a *contractor* if $\rho(\mathbf{G}) < 1$

$$\Longrightarrow \|\mathbf{e}_{k+1}\| \leq \rho\|\mathbf{e}_k\| \qquad (\text{ for } k \text{ sufficiently large })$$

$$\frac{\|\mathbf{e}_{k+1}\|}{\|\mathbf{e}_k\|} \leq \rho$$

- In *nonlinear case*, $\mathbf{G}(\mathbf{x}) \approx \mathbf{G}(\mathbf{x}^*)$ for $\mathbf{x} \approx \mathbf{x}^*$.

- If $\rho(\mathbf{G}(\mathbf{x})) < 1$ at $\mathbf{x} = \mathbf{x}^*$, there is a neighborhood near $\mathbf{x}^*$ for which we will have a contractor

Linear Example for g, continued

- Suppose $\mathbf{g}(\mathbf{x})$ is given as

$$\mathbf{g}(\mathbf{x}) = \begin{bmatrix} 0.1 & -.1 \\ 0.0 & 0.5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1.9 \\ 0.5 \end{bmatrix},$$

  which has $\mathbf{x}^* = [2\ 1]^T$ as a fixed point

- *What is the expected convergence rate?*

- For upper triangular matrix $\mathbf{G}$, eigenvalues are diagonal entries and max diagonal entry is 0.5, corresponding to $\rho(\mathbf{G})$

- Therefore, expect asympotic error, $\|\mathbf{x} - \mathbf{x}^*\|$ to contract by a factor of two with each iteration.

Linear Example for g, continued

fixpt_2d.m

```
hdr; format shorte

G=[ 0.1 -.1 ;
    0.0 0.5 ];

b=[1.9 .5]';
x=[0 0]';

for k=1:9;
    x = G*x + b;
    disp([k x' x(1)-2 x(2)-1 ])
end;
```

```
octave:2> fixpt_2d
   1.0000e+00   1.9000e+00   5.0000e-01  -1.0000e-01  -5.0000e-01
   2.0000e+00   2.0400e+00   7.5000e-01   4.0000e-02  -2.5000e-01
   3.0000e+00   2.0290e+00   8.7500e-01   2.9000e-02  -1.2500e-01
   4.0000e+00   2.0154e+00   9.3750e-01   1.5400e-02  -6.2500e-02
   5.0000e+00   2.0078e+00   9.6875e-01   7.7900e-03  -3.1250e-02
   6.0000e+00   2.0039e+00   9.8438e-01   3.9040e-03  -1.5625e-02
   7.0000e+00   2.0020e+00   9.9219e-01   1.9529e-03  -7.8125e-03
   8.0000e+00   2.0010e+00   9.9609e-01   9.7654e-04  -3.9062e-03
   9.0000e+00   2.0005e+00   9.9805e-01   4.8828e-04  -1.9531e-03
```

Note the asymptotic 2-fold reduction in error

Fixed-Point Iteration: Newton's Method

- In $n > 1$ dimensions, Newton's method for $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ takes the form

$$\mathbf{x}_{k+1} \;=\; \mathbf{x}_k \;-\; \mathbf{J}_k^{-1}\mathbf{f}_k,$$

  where $\mathbf{f}_k := \mathbf{f}(\mathbf{x}_k)$ and $\mathbf{J}_k := \mathbf{J}(\mathbf{x}_k)$ is the Jacobian matrix of $\mathbf{f}$ evaluated at $\mathbf{x}_k$,

$$[\mathbf{J}_k]_{ij} \;:=\; \left[\frac{\partial f_i}{\partial x_j}\right]$$

Fixed-Point Iteration: Newton's Method

Why this works:

- Consider Taylor series in n dimensions:

$$\mathbf{f}(\underbrace{\mathbf{x}+\epsilon\mathbf{v}}_{\mathbf{x}_{k+1}})\Big|_i = \mathbf{f}(\mathbf{x})|_i + \epsilon\sum_i \frac{\partial f_i}{\partial x_j}\mathbf{v}_j + O(\epsilon^2)$$

$$= 0 \quad \Longrightarrow \epsilon\mathbf{v} = -\mathbf{J}_k^{-1}\mathbf{f}_k$$

- Thus, the fixed point iteration is

$$\begin{aligned}\mathbf{x}_{k+1} &= \mathbf{x}_k + \epsilon\mathbf{v} \\ &= \mathbf{x}_k - \mathbf{J}_k^{-1}\mathbf{f}_k \\ &= \mathbf{g}(\mathbf{x}_k)\end{aligned}$$

Convergence

- Let

$$
\begin{aligned}
[\mathbf{G}(\mathbf{x})]_{ij} &:= \frac{\partial g_i}{\partial x_j} \\
&= \frac{\partial}{\partial x_j}\left[x_i - \left(\mathbf{J}_k^{-1}\mathbf{f}\right)_i\right] \\
&= I - \mathbf{J}_k^{-1}\underbrace{\frac{\partial f_i}{\partial x_j}}_{\mathbf{J}_k} - \sum_{p=1}^{n} f_p(\mathbf{x})\mathbf{H}_p(\mathbf{x}) \\
&= \mathbf{0}
\end{aligned}
$$

Convergence is at least quadratic

- Here,

$$
[\mathbf{H}_p]_{ij} = \frac{\partial}{\partial x_p}\left[\mathbf{J}^{-1}\right]_{ij}
$$

Example

- Here, we initiate a sketch of how the terms are evaluated

$$\mathbf{f} = \begin{bmatrix} x_1 + 2x_2 - 2 \\ x_1^2 + 4x_2^2 - 4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \qquad \mathbf{x}^* = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\mathbf{J} = \frac{\partial f_i}{\partial x_j} = \begin{bmatrix} 1 & 2 \\ 2x_1 & 8x_2 \end{bmatrix}$$

$$\mathbf{J}^{-1} = \frac{1}{8x_2 - 4x_1} \begin{bmatrix} 8x_2 & -2 \\ -2x_1 & 1 \end{bmatrix}$$

$$\mathbf{g} = \mathbf{x} - \mathbf{J}^{-1}\mathbf{f}$$

- Note that if $\mathbf{f}(\mathbf{x})$ is linear with form

$$\mathbf{f}(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$$

then the Jacobian is simply $\mathbf{J} = \mathbf{A}$.

Example, continued

- We look at the gradient of $\mathbf{g}(\mathbf{x})$:

$$\frac{\partial g_i}{\partial x_j} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \mathbf{J}^{-1} \underbrace{\left[\frac{\partial f_i}{\partial x_j}\right]}_{\mathbf{J}} - \sum_p f_p(\mathbf{x}) \left(\frac{\partial}{\partial x_p} \left[\mathbf{J}^{-1}\right]_{ip} \right)$$

$$= \sum_p f_p(\mathbf{x}) \mathbf{H}_p(\mathbf{x})$$

- For $\mathbf{x} = \mathbf{x}^*$, each term is 0.

- **Q:** What is $\mathbf{H}_p$??

Example, continued

- Let $\mathbf{J}^{-1} := \mathbf{A}$

$$\mathbf{J}^{-1}\mathbf{f} = \underbrace{\left[\frac{1}{8x_2 - 4x_1}\begin{pmatrix} 8x_2 & -2 \\ -2x_1 & 1 \end{pmatrix}\right]}_{\mathbf{A}(\mathbf{x})} \underbrace{\begin{pmatrix} x_1 + 2x_2 - 2 \\ x_1^2 + 4x_2^2 - 4 \end{pmatrix}}_{\mathbf{f}(\mathbf{x})}$$

- Evaluate the terms involving $\mathbf{H}_p$

$$\frac{\partial}{\partial x_j}\sum_p A_{ip} f_p = \sum_p \left(\frac{\partial}{\partial x_j} A_{ip}\right) + \sum_p A_{ip} \underbrace{\frac{\partial f_p}{\partial x_j}} \mathbf{A}^{-1}$$

- Evaluate $\mathbf{H}_p$:

$$\frac{\partial}{\partial x_1}\mathbf{A} = \frac{4}{(8x_2 - 4x_1)^2)}\begin{pmatrix} 8x_2 & -2 \\ -2x_1 & 1 \end{pmatrix} + \frac{1}{8x_2 - 4x_1}\begin{pmatrix} 0 & 0 \\ -2 & 0 \end{pmatrix} =: \mathbf{H}_1$$

$$\frac{\partial}{\partial x_2}\mathbf{A} = \frac{-8}{(8x_2 - 4x_1)^2}\begin{pmatrix} 8x_2 & -2 \\ -2x_1 & 1 \end{pmatrix} + \frac{1}{8x_2 - 4x_1}\begin{pmatrix} 8 & 0 \\ 0 & 0 \end{pmatrix} =: \mathbf{H}_2$$

Cost of Newton's Method

Cost per iteration of Newton's method for dense problem in n dimensions is substantial

- Computing Jacobian matrix costs n^2 scalar function evaluations
- Solving linear systems costs $O(n^3)$ operations
- Can, however, re-use the LU factorization for several iterations, rather than recomputing at each step
- Alternatives are to seek secant-like methods

Secant Updating Methods

- Secant updating methods reduce cost by
  - Using function values at successive iterates to build approximate Jacobian and avoiding explicit evaluation of derivatives
  - Updating factorization of approximation using Sherman-Morrison, rather than refactoring it each iteration

- Most secant updating methods have superlinear but not quadratic convergence rate

- Secant updating methods often cost less overall than Newton's method because of lower cost per iteration

Broyden's Method

- As in the 1D case, working with the Jacobian is potentially painful
- Idea is to build a secant type method by solving the linear model problem

$$\mathbf{0} \approx \mathbf{f}_{k+1} \approx \mathbf{f}_k + \mathbf{J}_k \underbrace{(\mathbf{x}_{k+1} - \mathbf{x}_k)}_{\mathbf{s}}$$

$$\implies \mathbf{s} = -\mathbf{J}_k^{-1}\mathbf{f}_k$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}$$

- Here, we let $\mathbf{B}_k \approx \mathbf{J}_k$ be a low-cost approximation to $\mathbf{J}_k$
- Redefine Newton step as

$$\mathbf{B}_k\mathbf{s} = -\mathbf{f}_k$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}$$

update $\mathbf{B}_k$

Broyden's Method

- As in the 1D case, we want to update $\mathbf{B}_k$
- Nominally, we need something like

$$\mathbf{B}_{k+1} = \frac{\mathbf{f}_{k+1} - \mathbf{f}_k}{\mathbf{x}_{k+1} - \mathbf{x}_k}$$

- Of course, fractions with vectors are difficult, so consider

$$\mathbf{f}_{k+1} - \mathbf{f}_k = \mathbf{B}_{k+1}(\mathbf{x}_{k+1} - \mathbf{x}_k) = \mathbf{B}_{k+1}\mathbf{s}_k$$ ***Secant condition***

- Not enough to uniquely determine $\mathbf{B}_{k+1}$

Broyden's Method, continued

- To resolve uniqueness issue in a simple (low-cost) way, choose $\mathbf{B}_{k+1}$ to be a rank-one perturbation of $\mathbf{B}_k$

- That is $\mathbf{B}_{k+1} - \mathbf{B}_k$ should be a rank-one matrix such that $\mathbf{B}_{k+1}$ satisfies the secant condition, $\mathbf{B}_{k+1}\mathbf{s}_k = \mathbf{f}_{k+1} - \mathbf{f}_k$.

- Resulting Jacobian update is

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{1}{\mathbf{s}_k^T \mathbf{s}_k} \left(\mathbf{f}_{k+1} - \mathbf{f}_k - \mathbf{B}_k \mathbf{s}_k\right) \mathbf{s}_k^T$$

- Main advantage is that Sherman-Morrison can be used to update the LU factorization of $\mathbf{B}_k$, which avoids the $O(n^3)$ refactor cost

Robust Newton-Like Methods

- Newton's method and its variants may fail to converge when started far from solution.

- Safeguards can enlarge region of convergence for Newton-like methods

- Simlest precaution is *damped-Newton method*, in which new iterate is

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{s}_k$$

  where $\mathbf{s}_k$ is Newton (or Newton-like) step and α_k is scalar parameter chosen to ensure progess toward solution

- Parameter α_k reduces Newton step when it is too large, but $\alpha_k = 1$ suffices near solution and still yields fast asymptotic convergence rate

Example: Broyden's Method

Use Broyden's method to solve nonlinear system

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} x_1 + 2x_2 - 2 \\ x_1^2 + 4x_2^2 - 4 \end{bmatrix} = \boldsymbol{0}$$

If $\boldsymbol{x}_0 = \begin{bmatrix} 1 & 2 \end{bmatrix}^T$, then $\boldsymbol{f}(\boldsymbol{x}_0) = \begin{bmatrix} 3 & 13 \end{bmatrix}^T$, and we choose

$$\boldsymbol{B}_0 = \boldsymbol{J}_f(\boldsymbol{x}_0) = \begin{bmatrix} 1 & 2 \\ 2 & 16 \end{bmatrix}$$

Solving system

$$\begin{bmatrix} 1 & 2 \\ 2 & 16 \end{bmatrix} \boldsymbol{s}_0 = \begin{bmatrix} -3 \\ -13 \end{bmatrix}$$

gives $\boldsymbol{s}_0 = \begin{bmatrix} -1.83 \\ -0.58 \end{bmatrix}$, so $\boldsymbol{x}_1 = \boldsymbol{x}_0 + \boldsymbol{s}_0 = \begin{bmatrix} -0.83 \\ 1.42 \end{bmatrix}$

Example: Broyden's Method

Evaluating at new point $\boldsymbol{x}_1$ gives $\boldsymbol{f}(\boldsymbol{x}_1) = \begin{bmatrix} 0 \\ 4.72 \end{bmatrix}$, so

$$\boldsymbol{y}_0 = \boldsymbol{f}(\boldsymbol{x}_1) - \boldsymbol{f}(\boldsymbol{x}_0) = \begin{bmatrix} -3 \\ -8.28 \end{bmatrix}$$

From updating formula, we obtain

$$\boldsymbol{B}_1 = \begin{bmatrix} 1 & 2 \\ 2 & 16 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ -2.34 & -0.74 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ -0.34 & 15.3 \end{bmatrix}$$

Solving system

$$\begin{bmatrix} 1 & 2 \\ -0.34 & 15.3 \end{bmatrix} \boldsymbol{s}_1 = \begin{bmatrix} 0 \\ -4.72 \end{bmatrix}$$

gives $\boldsymbol{s}_1 = \begin{bmatrix} 0.59 \\ -0.30 \end{bmatrix}$, so $\boldsymbol{x}_2 = \boldsymbol{x}_1 + \boldsymbol{s}_1 = \begin{bmatrix} -0.24 \\ 1.120 \end{bmatrix}$

Example: Broyden's Method

Evaluating at new point $\boldsymbol{x}_2$ gives $\boldsymbol{f}(\boldsymbol{x}_2) = \begin{bmatrix} 0 \\ 1.08 \end{bmatrix}$, so

$$\boldsymbol{y}_1 = \boldsymbol{f}(\boldsymbol{x}_2) - \boldsymbol{f}(\boldsymbol{x}_1) = \begin{bmatrix} 0 \\ -3.64 \end{bmatrix}$$

From updating formula, we obtain

$$\boldsymbol{B}_2 = \begin{bmatrix} 1 & 2 \\ -0.34 & 15.3 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1.46 & -0.73 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 1.12 & 14.5 \end{bmatrix}$$

Iterations continue until convergence to solution $\boldsymbol{x}^* = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

broyden4.m

```
format compact; format longe; hold off
x=2*ones(2,1); f=0*x;

J = [ 1 2 ; 2*x(1) 8*x(2) ];
f(1) = x(1)     + 2*x(2)     - 2;
f(2) = x(1).^2 + 4*x(2).^3 - 3;


for iter=1:10;

   s = -J\f;
   x = x+s;

   fo = f;
   f(1) = x(1)     + 2*x(2)     - 2;
   f(2) = x(1).^2 + 4*x(2).^3 - 3;

   y = f-fo;

   J = J - ((J*s-y)*s')/(s'*s);

   plot(x(1),x(2),'ro'); hold on

   ns = norm(s); nf = norm(f); [ns nf]

end;
```

```
ans =
     2.139655346077961e+00     2.054687500000000e+00
ans =
     6.734825454103858e-01     4.826427692876747e+00
ans =
     1.172734304676712e+00     2.562485091574165e-01
ans =
     6.575484354786346e-02     5.210384348368891e-02
ans =
     1.678260886810548e-02     1.508348427554207e-03
ans =
     5.003216525182486e-04     9.672703087826307e-06
ans =
     3.229159393332246e-06     1.828977858053804e-09
ans =
     6.107060192670134e-10     1.776356839400250e-15
ans =
     5.931361405127450e-16     4.440892098500626e-16
ans =
     1.977120468375818e-16     4.440892098500626e-16
```

- Notice the $e_k e_{k-1}$ convergence behavior.

Broyden's Method, continued

- Note that we can also initiate Broyden's method with $\mathbf{B}_0 = \mathbf{I}$, which avoids *all* derivative evaluation.

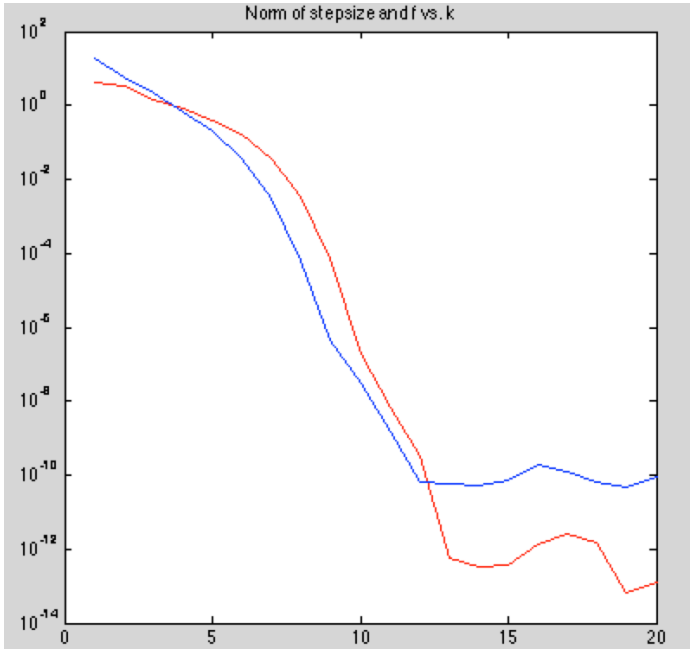
- In this case, a damped update is generally needed to keep the update in scale, at least for the early iterations

- We explore this option in the following demo

- *demo_broyden.m*
- *demo_broyq.m*

broyden_bratu.m

```
%  Solve 1D Bratu Problem:   -u'' = sigma exp(u), u(0)=u(1)=0
%
%  -u''  approximated by 2nd-order finite differences
%
%  Usage: sigma=3.5; broyden_bratu


n=180; % sigma = 3.3;  % Sigma_c is ~ 3.5+

h=1./(n+1); b = ones(n,1); x=1:n; x=h*x'; h2i = 1./(h*h);

a=-2*b; A = h2i*spdiags([b a b],-1:1, n,n);
c=-2*b + sigma*h*h*exp(x); J = h2i*spdiags([b c b],-1:1, n,n);

J = A;

u=b*0; f = A*u + sigma*exp(u);

for iter=1:20;
%  mesh(log(abs(J))); pause(1);
   s = -J\f;
   u = u+s;
   f = A*u + sigma*exp(u);
   c=-2*b + sigma*h*h*exp(u);
%  J = A;                                  % Constant
%  J = A + (f*s')/(s'*s);                  % A + rank 1
%  J = h2i*spdiags([b c b],-1:1, n,n);     % Newton
   J = J + (f*s')/(s'*s);                  % Broyden

   k(iter) = iter; ns(iter) = norm(s); nf(iter) = norm(f);
   [ns(iter) nf(iter)]

   plot(x,u,'r-',x,0*x,'k-',x,f,'g-'); pause(.1)

end;
plot(x,u,'r-',x,0*x,'k-',x,f,'g-'); pause
semilogy(k,ns,'r-',k,nf,'b-')
title('Norm of stepsize and f vs. k'); axis square
```



Higher-Dimensional Examples

- Bratu Problem – a nonlinear ODE or PDE
- Jacobi-Free Newton-Krylov methods

1 Newton's Method in Higher Space Dimensions

We are interested in solving a system of nonlinear equations in n dimensions, $\mathbf{f}(\mathbf{x}) = 0$. As with the scalar ($n$=1) case, we recast the problem as a fixed point iteration using Newton's method

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{x}_k - J^{-1}\mathbf{f}(\mathbf{x}_k) \\ &= \mathbf{x}_k + \mathbf{s}_k \end{aligned} \tag{1}$$

where the update step s_k satisfies $J\mathbf{s}_k = -\mathbf{f}_k$ and

$$J_{ij} := = \left.\frac{\partial f_i}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}_k} \tag{2}$$

is the Jacobian matrix associated with the $\mathbf{f}(\mathbf{x})$ at $\mathbf{x} = \mathbf{x}_k$. In some cases we use a *damped* Newton update of the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha\mathbf{s}_k$$

with $\alpha < 1$ chosen to guarantee that $||\mathbf{f}_{k+1}|| < ||\mathbf{f}_k||$.

A major difference between the scalar and vector case is that (1) requires the solution of an $n \times n$ system for each iteration. Given that the factor (LU decomposition of J) cost nominally scales as $O(n^3)$, a great deal of effort is expended to develop algorithms that can reduce this overhead. We explore one of these, Jacobi-Free Newton-Krylov (JFNK) methods at the end of this discussion. Presently, we carry on with the notion that we can solve systems in J, ever mindful that it typically represents the leading-order overhead in our method.

2 The Bratu Example

The text has a couple of nonlinear system examples for the case n=2. Here, we consider a larger problem that is motivated by (but not exactly like) the reaction-diffusion problem. In the following, we seek an unknown function $u(x)$ (where $x \in [0,1]$ is a spatial coordinate) that satisfies the steady-state heat (diffusion) equation

$$-\frac{d^2u}{dx^2} = q, \qquad u(0) = u(1) = 0, \tag{3}$$

where $q(x)$ represents the heat source. For the Bratu problem, we define

$$q = \sigma e^{u(x)},$$

where σ is a parameter.

Equation (3) is a ordinary differential equation (ODE) and in particular it is a nonlinear two-point boundary value problem with boundary conditions prescribed as above. To turn this continous problem into a system of nonlinear equations we first discretize the second derivate term in (3) using a finite difference appoximation. Through application of Taylor series at points $x_j := jh$, $j = 1, \ldots, n$, with grid spacing $h = 1/(n+1)$. we derive

$$-\frac{u_{j-1} - 2u_j - u_{j+1}}{h^2} = -\left.\frac{d^2u}{dx^2}\right|_j + O(h^2) = \sigma e^{u_j} + O(h^2). \tag{4}$$

If we neglect the $O(h^2)$ error term then the system is solvable we can anticipate that our solution u_j will approximate $u(x_j)$ to order h^2.

Subtracting the right-hand side from both sides of (4) and changing the sign, we arrive at the n-dimensional root-finding problem $\mathbf{f}(\mathbf{u}) = 0$,

$$f_j \;=\; \frac{u_{j-1} - 2u_j - u_{j+1}}{h^2} + \sigma e^{u_j} \;=\; 0, \;\; j = 1, \ldots, n \tag{5}$$

To apply (1), we need the Jacobian (2), which is given by the tridiagonal matrix

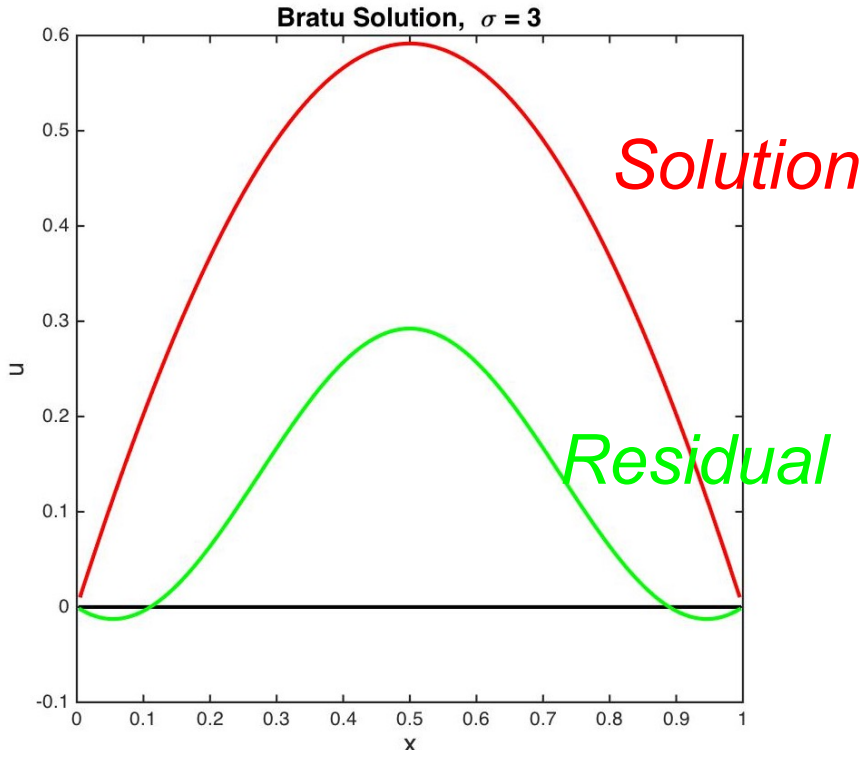
$$J \;=\; \frac{1}{h^2} \begin{pmatrix} a_1 & b & & & \\ b & a_2 & b & & \\ & b & \ddots & \ddots & \\ & & \ddots & \ddots & b \\ & & & b & a_n \end{pmatrix}, \tag{6}$$

with $b = 1$ and $a_j = -2 + h^2 \sigma e^{u_j}$. Note that, as is often the case with systems arising from differential equations, J is *sparse*. That is, it has a fixed number of nonzeros per row, independent of n and thus has $O(n)$ nonzeros. Moreover, because this system is tridiagonal, the factor cost is only $O(n)$, which is of the same order as the other update steps in the algorithm. (In higher space dimensions, the factor costs is $O(n^\gamma)$ with $\gamma > 1$ and direct factorization loses favor in comparison to iterative JFNK methods.)

We illustrate the result for $n = 80$ and $\sigma = 1$. Using (1), and $\mathbf{u}_0 = 0$, we have the following results for the norms of the step size and residual,

k	|| s_k ||	|| f_k ||
1	9.141106002022624e-01	8.944271909999159e+00
2	6.555298143445134e-03	5.803485294158030e-02
3	3.746387054601207e-07	3.363605689013008e-06
4	1.553772829606369e-15	1.007219709041583e-12
5	1.184226711286128e-15	1.430323637721320e-12
6	1.746857971735465e-16	1.074544804307374e-12

from which see that we are converging to a fixed point ($||\mathbf{s}_k|| \longrightarrow 0$) quadratically, as is typical when Newton's method is working. In addition, we see that $||\mathbf{f}_k||$ is not going to ϵ_M, which might be expected given that the condition number of J is about 4×10^3.



The corresponding source code is

```
n=80; sigma = 1;
h=1./(n+1); b = ones(n,1); x=1:n; x=h*x'; h2i = 1./(h*h);

a=-2*b; A = h2i*spdiags([b a b],-1:1, n,n);
c=-2*b + sigma*h*h*exp(x); J = h2i*spdiags([b c b],-1:1, n,n);

u=b*0;
for iter=1:31;
   f=A*u + sigma*exp(u);
   c=-2*b + sigma*h*h*exp(u);
   J = h2i*spdiags([b c b],-1:1, n,n);  % J is sparse
   s = -J\f;
   u = u+s;
   ns = norm(s); nf = norm(f); [ns nf]
end;
plot(x,u,'r-',x,0*x,'k-'); hold on
```

Nonlinear term

The solution $u(x)$ is not terribly interesting, but we plot it for completeness in Fig. 1.

3 Refinements of the Algorithm

It turns out that for some values of σ the Bratu problem has two solutions, whereas above a critical value there are no solutions. We discuss a bit of the behavior for σ on the interval $[0, \sigma_c]$, where $\sigma_c \approx 3.51355$. In Fig. 2, we plot $\max_x |u(x)|$ as a function of σ on the lower branch of solutions. There is another branch (not shown) which sits above this one. The existance of this branch is indicated by the fact that the solution is turning as $\sigma \longrightarrow \sigma_c$. The path that is shown here was found by monitoring convergence of Newton's method for a sequence $\sigma_{l+1} = \sigma_l + \delta\sigma$, and reducing $\delta\sigma$ whenever Newton's method required more than 20 iterations. The work was reduced by using the solution at σ_l as the starting point for $\sigma = \sigma_{l+1}$. Finding the upper branch is beyond the scope of this discussion. We mention, however, that a commonly used approach that has proven quite successful is pseudo-arclength continuation, developed by H.B. Keller and coworkers. With this approach, σ is taken as an additional unknown and a new parameter, s, the arclength of the path, is introduced as an auxiliary parameter (which will not be multivalued).[1]

[1] See, for example, Sec. 4.5 in http://www.math.tifr.res.in/~publ/ln/tifr79.pdf

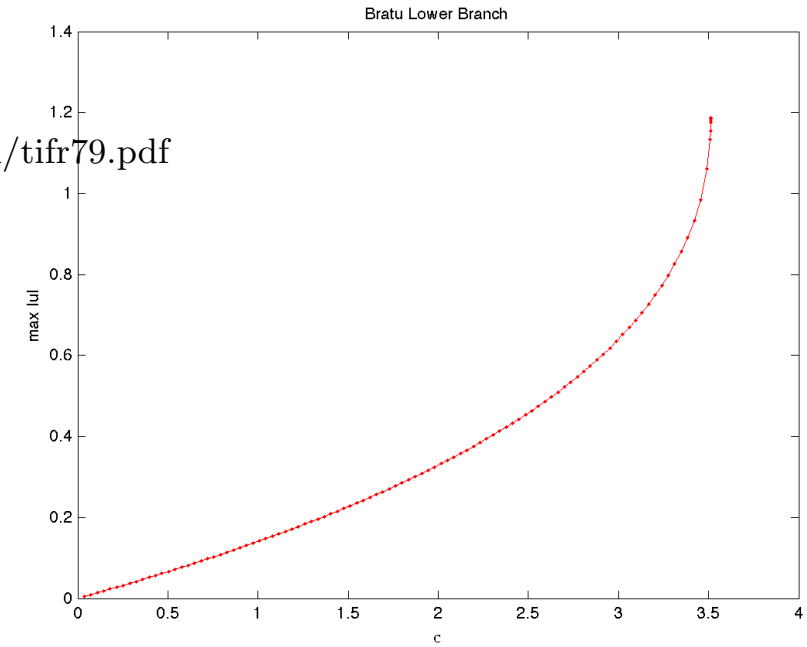


Figure 2: Lower branch of Bratu solutions vs σ.

4 JFNK: Reducing Factorization Costs

For large sparse Jacobians, the solution of

$$J\mathbf{s}_k \quad = \quad -\mathbf{f}_k$$

is best effected with iterative methods such as conjugate Gradients (if J is SPD, which is rare) or GMRES.

Iterative methods for $A\mathbf{x} = \mathbf{b}$ simply require repeated matrix-vector product evaluation of the form

$$\mathbf{p} \quad = \quad A\mathbf{w}. \tag{7}$$

For Newton iteration in higher space dimensions where $A = J$ this apparently requires forming the Jacobian,

$$J_{ij} \quad := \quad = \left.\frac{\partial f_i}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}_k},$$

which may be very complex for a large nonlinear system arising from a partial differential equation.

Fortunately, careful inspection of (7) reveals that we do not need to produce A (=: J). We only need to produce $\mathbf{w}$. This observation leads to the idea of developing a *Jacobi-free* method that does precisely that.

Consider a Taylor series about $\mathbf{x}_k$ in terms of $\mathbf{x}_k + \epsilon\mathbf{s}$. We can write

$$\mathbf{f}(\mathbf{x}_k + \epsilon\mathbf{s}) \;=\; \mathbf{f}(\mathbf{x}_k) + \epsilon J(\mathbf{x}_k)\mathbf{s} + O(\epsilon^2).$$

Neglecting the $O(\epsilon^2)$ term, we solve for $J(\mathbf{x}_k)\mathbf{s} =: J\mathbf{s}$,

$$J\mathbf{s} \;\approx\; \frac{\mathbf{f}(\mathbf{x}_k + \epsilon\mathbf{s}) - \mathbf{f}(\mathbf{x}_k)}{\epsilon}, \tag{8}$$

which is a finite difference approximation to $J\mathbf{s}$. As we know from Chapter 1, we can expect this approximation to be accurate to only $\approx \sqrt{\epsilon_M}$ and we should take ϵ no smaller than $\approx \sqrt{\epsilon_M}$. In general, one needs to consider norms of the terms involved in order to better understand how ϵ should be selected. There is a vast literature on the topic.[2]

The key advance here is that one can use (8) inside an iterative method for solving $J\mathbf{s} = -\mathbf{f}$ without ever forming (or factoring!) J. One simply needs repeated evaluation of the nonlinear functional, $f(\mathbf{x}_k + \epsilon\mathbf{s})$ for varying values of $\mathbf{s}$.

[2] An excellent starting point is Knoll and Keyes, *Jacobian-free NewtonKrylov methods: a survey of approaches and applications*, J. Comp. Phys, 2004.