Outline

1. Nonlinear Equations
2. Numerical Methods in One Dimension
3. Methods for Systems of Nonlinear Equations

Nonlinear Equations

- Given function f, we seek value x for which

$$f(x) = 0$$

- Solution x is *root* of equation, or *zero* of function f
- So problem is known as *root finding* or *zero finding*

Nonlinear Equations

Two important cases

- Single nonlinear equation in one unknown, where

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

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

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

  $$\boldsymbol{f}: \mathbb{R}^n \to \mathbb{R}^n$$

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

Examples: Nonlinear Equations

- Example of nonlinear equation in one dimension

$$x^2 - 4\sin(x) = 0$$

for which $x = 1.9$ is one approximate solution

- Example of system of nonlinear equations in two dimensions

$$\begin{aligned} x_1^2 - x_2 + 0.25 &= 0 \\ -x_1 + x_2^2 + 0.25 &= 0 \end{aligned}$$

for which $\boldsymbol{x} = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}^T$ is solution vector

Existence and Uniqueness

- Existence and uniqueness of solutions are more complicated for nonlinear equations than for linear equations
- For function $f: \mathbb{R} \to \mathbb{R}$, *bracket* is interval $[a, b]$ for which sign of f differs at endpoints
- If f is continuous and $\text{sign}(f(a)) \neq \text{sign}(f(b))$, then Intermediate Value Theorem implies there is $x^* \in [a, b]$ such that $f(x^*) = 0$
- There is no simple analog for n dimensions

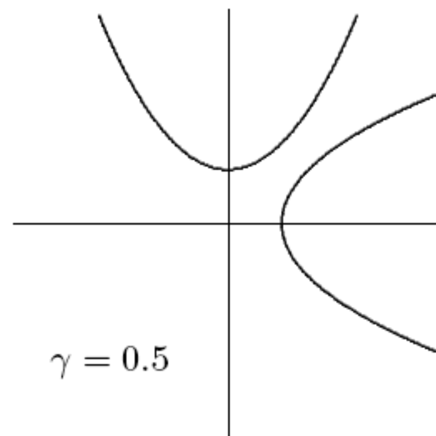
Examples: One Dimension

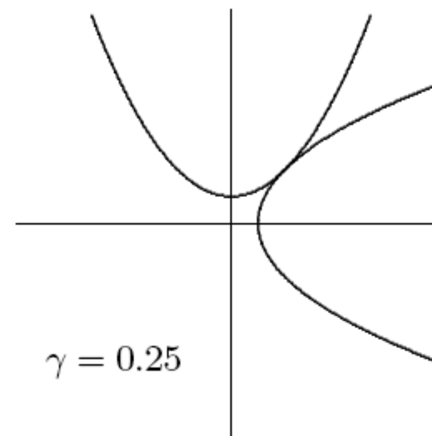
Nonlinear equations can have any number of solutions

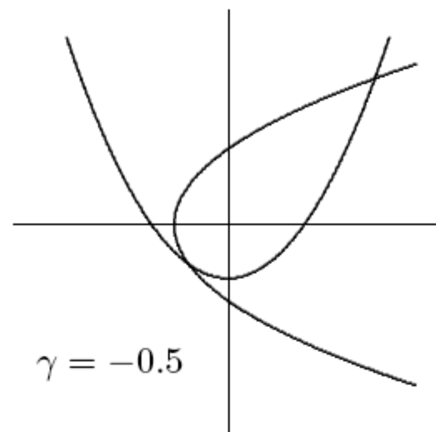
- $\exp(x) + 1 = 0$ has no solution
- $\exp(-x) - x = 0$ has one solution
- $x^2 - 4\sin(x) = 0$ has two solutions
- $x^3 + 6x^2 + 11x - 6 = 0$ has three solutions
- $\sin(x) = 0$ has infinitely many solutions

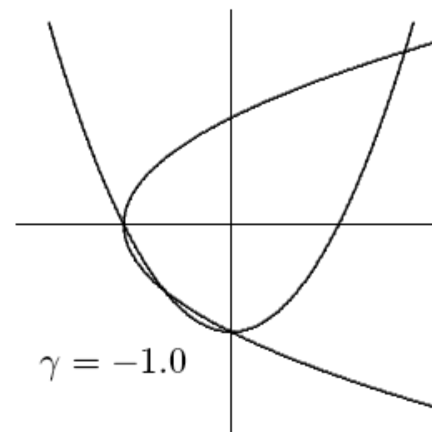
Example: Systems in Two Dimensions

$$\begin{aligned} x_1^2 - x_2 + \gamma &= 0 \\ -x_1 + x_2^2 + \gamma &= 0 \end{aligned}$$





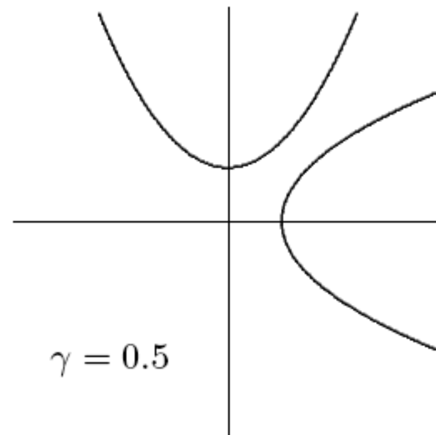


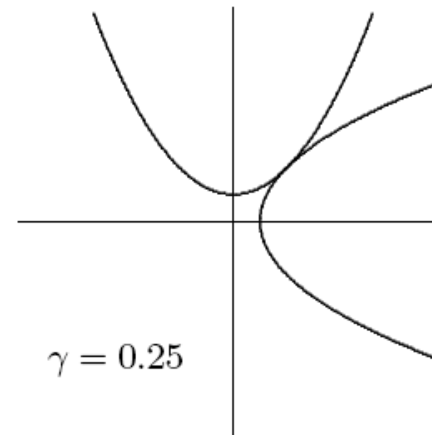


Example: Systems in Two Dimensions

$$\begin{aligned} x_1^2 - x_2 + \gamma &= 0 \\ -x_1 + x_2^2 + \gamma &= 0 \end{aligned}$$

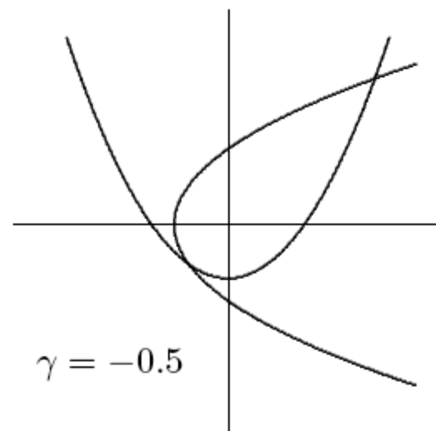
No solution

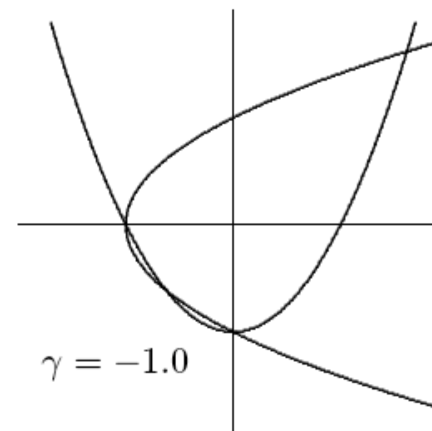




1 solution

2 solutions

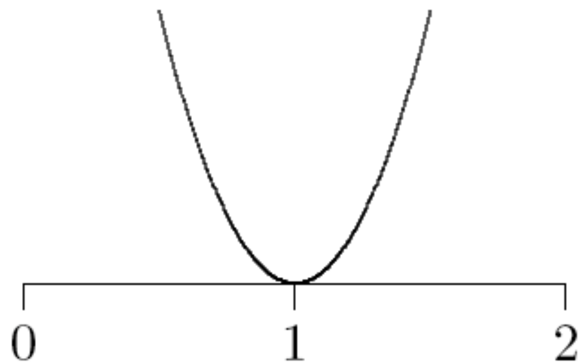


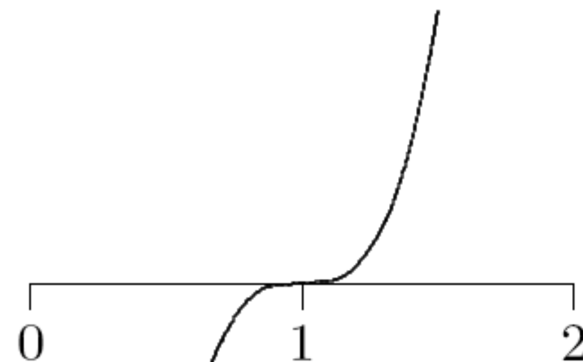


4 solutions

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 derivative of f that does not vanish at x^*), then root x^* has *multiplicity* m

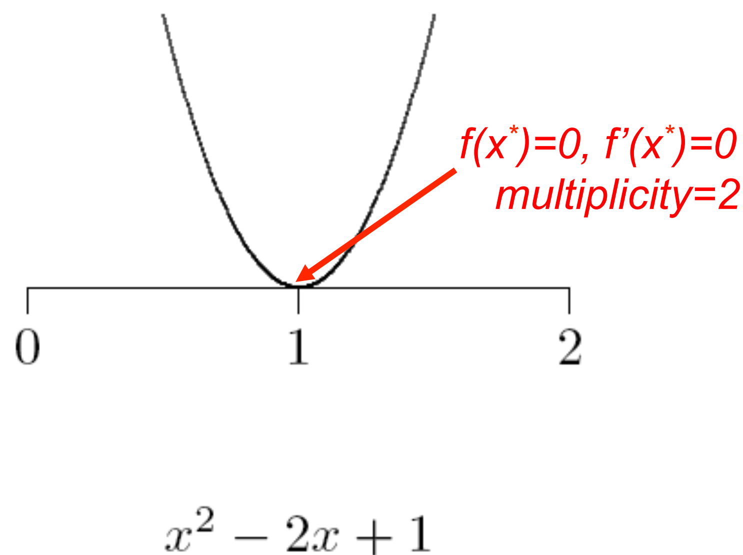


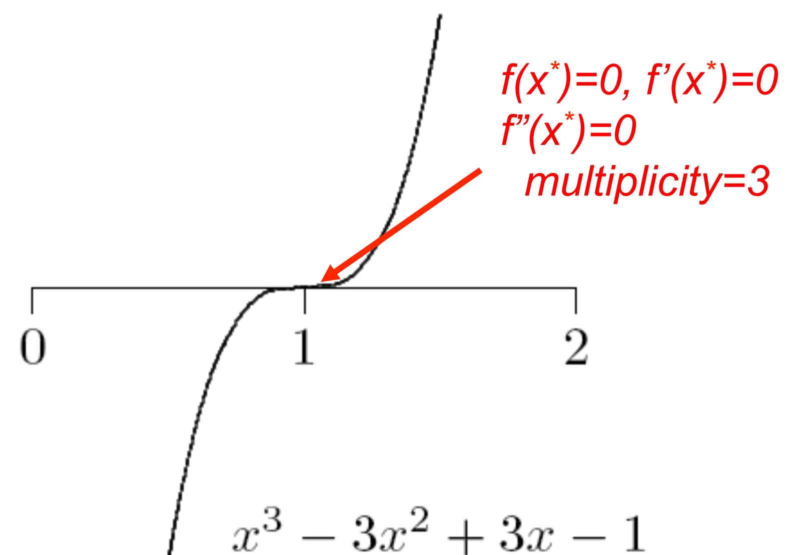


- If $m = 1$ ($f(x^*) = 0$ and $f'(x^*) \neq 0$), then x^* is *simple* root

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 derivative of f that does not vanish at x^*), then root x^* has *multiplicity* m





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

Sensitivity and Conditioning

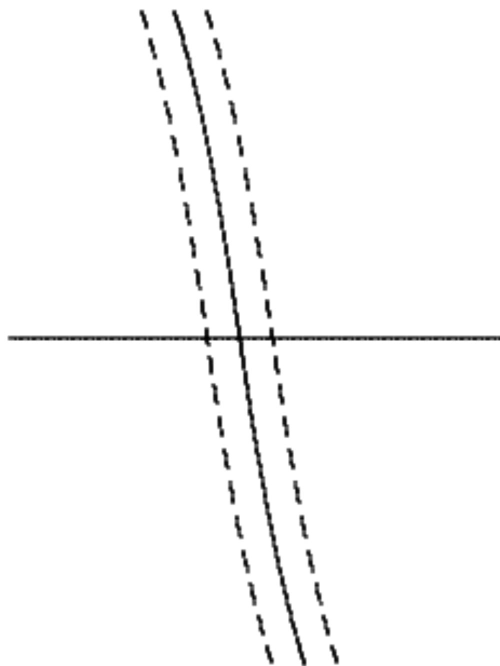
- Conditioning of root finding problem is opposite to that for evaluating function
- Absolute condition number of root finding problem for root x^* of $f: \mathbb{R} \to \mathbb{R}$ is $1/|f'(x^*)|$
- Root is ill-conditioned if tangent line is nearly horizontal
- In particular, multiple root ($m > 1$) is ill-conditioned
- Absolute condition number of root finding problem for root $\boldsymbol{x}^*$ of $\boldsymbol{f}: \mathbb{R}^n \to \mathbb{R}^n$ is $\|\boldsymbol{J}_f^{-1}(\boldsymbol{x}^*)\|$, where $\boldsymbol{J}_f$ is *Jacobian* matrix of $\boldsymbol{f}$,

$$\{\boldsymbol{J}_f(\boldsymbol{x})\}_{ij} = \partial f_i(\boldsymbol{x})/\partial x_j$$

- Root is ill-conditioned if Jacobian matrix is nearly singular

- Q: What is the Jacobian for the preceding 2D example ?

Sensitivity and Conditioning



well-conditioned

f is near zero for large range of x in neighborhood of x^*.
Difficult to find x^* to significant precision.

ill-conditioned

Sensitivity and Conditioning

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

$$\|\boldsymbol{f}(\hat{\boldsymbol{x}})\| \approx 0 \qquad \text{or} \qquad \|\hat{\boldsymbol{x}} - \boldsymbol{x}^*\| \approx 0\ ?$$

- First corresponds to "small residual," second measures closeness to (usually unknown) true solution $\boldsymbol{x}^*$
- Solution criteria are not necessarily "small" simultaneously
- Small residual implies accurate solution only if problem is well-conditioned

Convergence Rate

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

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

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

Important Definition!

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

  for some finite nonzero constant C

Convergence Rate, continued

Some particular cases of interest

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

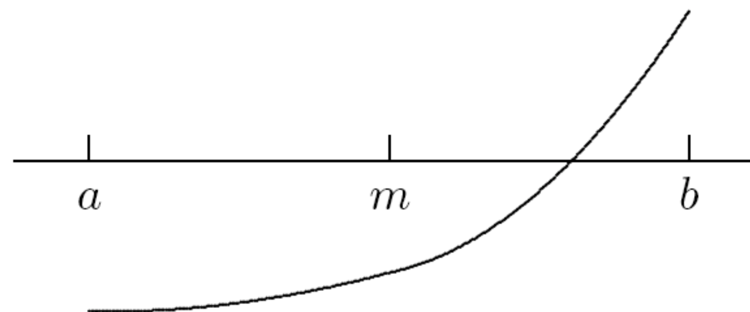
Convergence rate	Digits gained per iteration
linear	constant
superlinear	increasing
quadratic	double

Interval Bisection Method

Bisection method begins with initial bracket and repeatedly halves its length until solution has been isolated as accurately as desired

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

FYI: This formulation less sensitive to round-off than m = (a+b)/2



Example: Bisection Method

$$f(x) = x^2 - 4\sin(x) = 0$$

a	$f(a)$	b	$f(b)$
1.000000	−2.365884	3.000000	8.435520
1.000000	−2.365884	2.000000	0.362810
1.500000	−1.739980	2.000000	0.362810
1.750000	−0.873444	2.000000	0.362810
1.875000	−0.300718	2.000000	0.362810
1.875000	−0.300718	1.937500	0.019849
1.906250	−0.143255	1.937500	0.019849
1.921875	−0.062406	1.937500	0.019849
1.929688	−0.021454	1.937500	0.019849
1.933594	−0.000846	1.937500	0.019849
1.933594	−0.000846	1.935547	0.009491
1.933594	−0.000846	1.934570	0.004320
1.933594	−0.000846	1.934082	0.001736

Bisection Method, continued

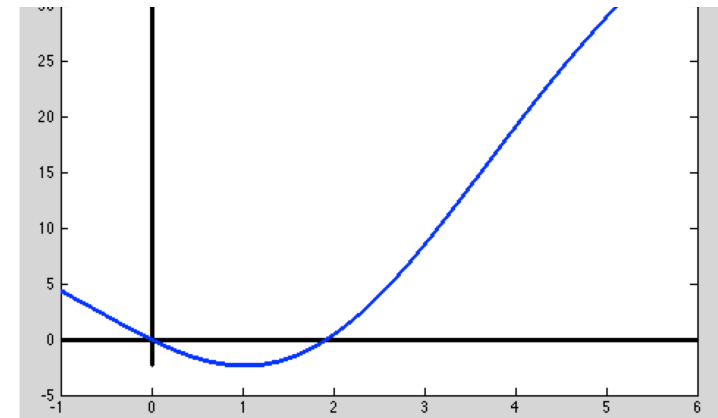
- Bisection method makes no use of magnitudes of function values, only their signs
- Bisection is certain to converge, but does so slowly
- At each iteration, length of interval containing solution reduced by half, convergence rate is *linear*, with $r = 1$ and $C = 0.5$
- One bit of accuracy is gained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

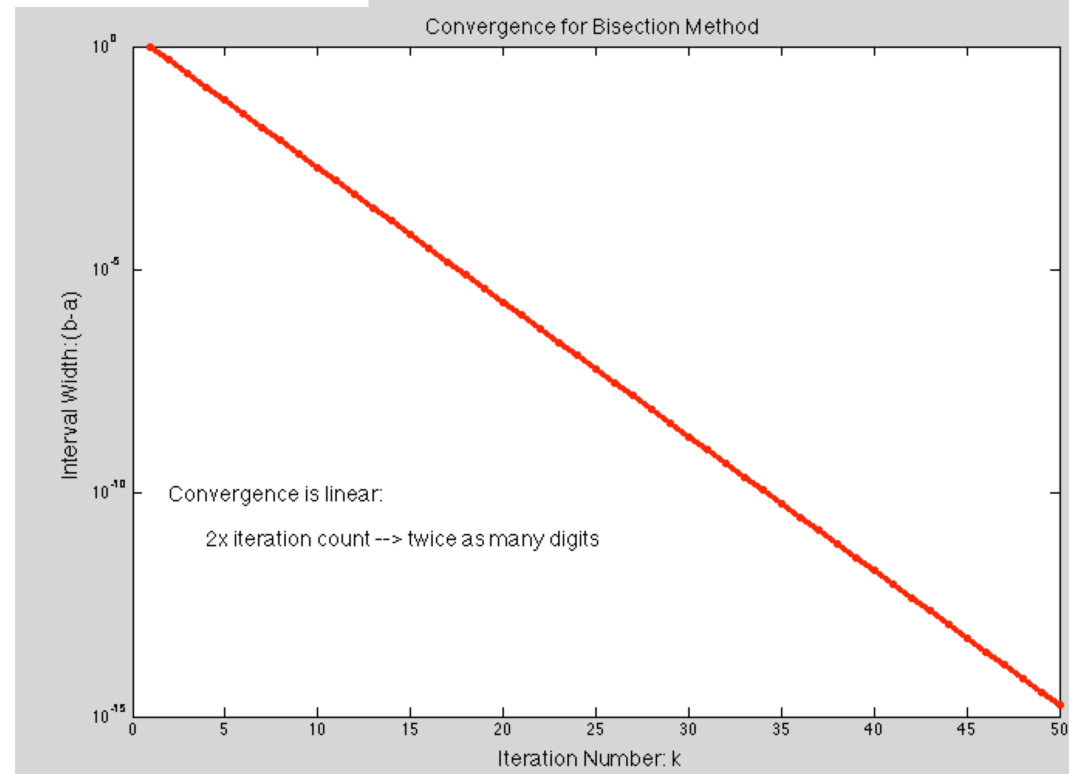
$$\left\lceil \log_2 \left(\frac{b - a}{tol} \right) \right\rceil$$

bisect.m

iterations, regardless of function f involved

bisect.m






```
format compact; format shorte; close all
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;
[k a fa b fb]

wk(k)=b-a; kk(k)=k;

end;

semilogy(kk,wk,'r.-','linewidth',2)
title('Convergence for Bisection Method','fontsize',14)
xlabel('Iteration Number: k','fontsize',14)
ylabel('Interval Width: (b-a)','fontsize',14)

text(2,1.e-10,'Convergence is linear:','fontsize',14)
text(4,1.e-11,'2x iteration count --> twice as many digits','fontsize',14)
```

Fixed-Point Problems

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

$$x = g(x)$$

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

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

  where fixed points for g are solutions for $f(x) = 0$

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

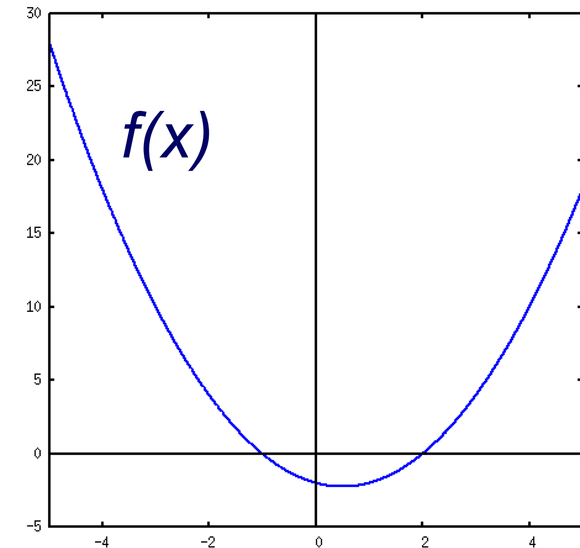
- For given equation $f(x) = 0$, there may be many equivalent fixed-point problems $x = g(x)$ with different choices for g

Example: Fixed-Point Problems

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

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

are solutions to equation $f(x) = 0$



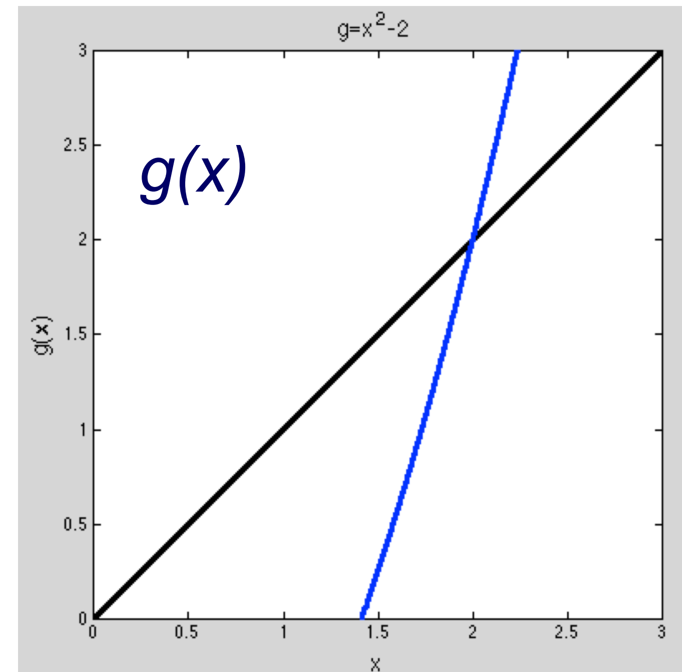
That is, when $g(x^*) = x^*$, then $f(x^*) = 0$.

Example: Fixed-Point Problems

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

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

are solutions to equation $f(x) = 0$



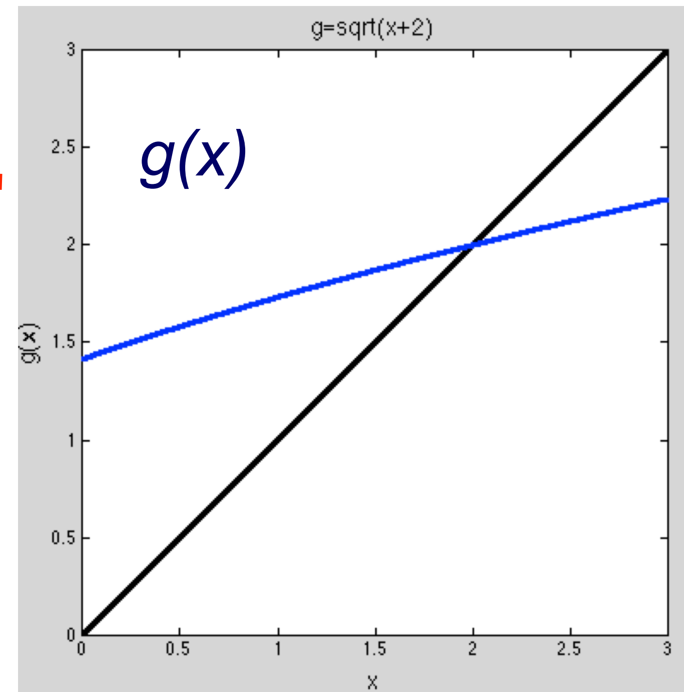
That is, when $g(x^*) = x^*$, then $f(x^*) = 0$.

Example: Fixed-Point Problems

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

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

are solutions to equation $f(x) = 0$



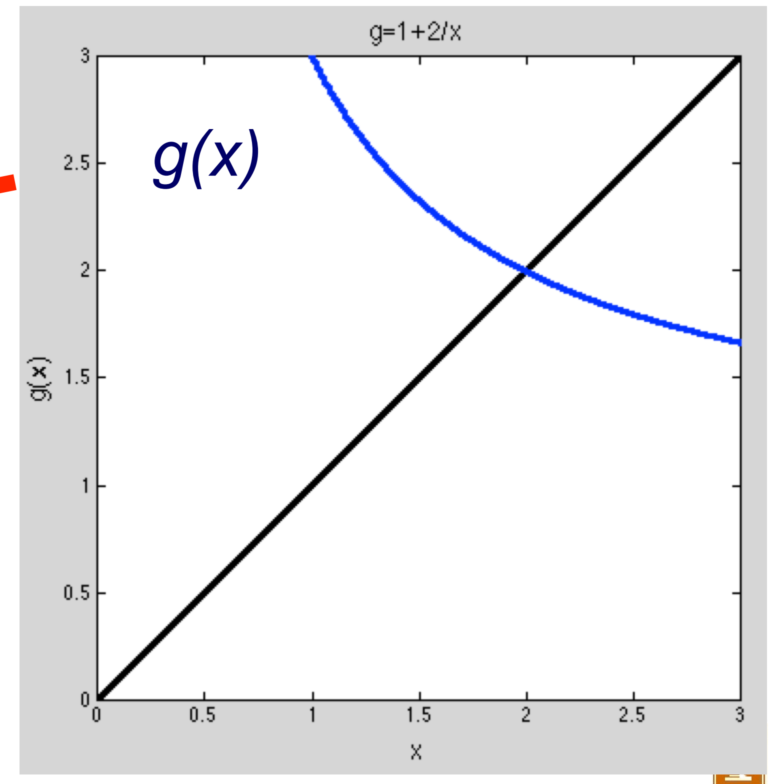
That is, when $g(x^*) = x^*$, then $f(x^*) = 0$.

Example: Fixed-Point Problems

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

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

are solutions to equation $f(x) = 0$



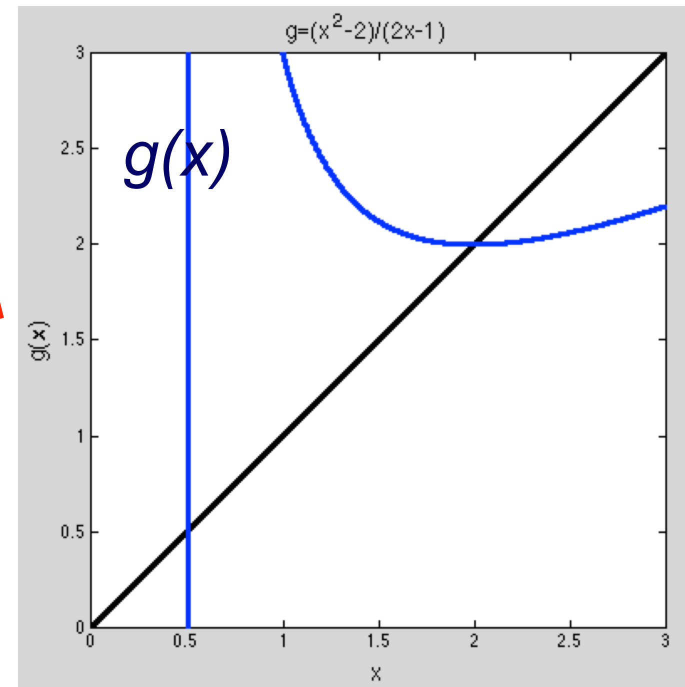
That is, when $g(x^*) = x^*$, then $f(x^*) = 0$.

Example: Fixed-Point Problems

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

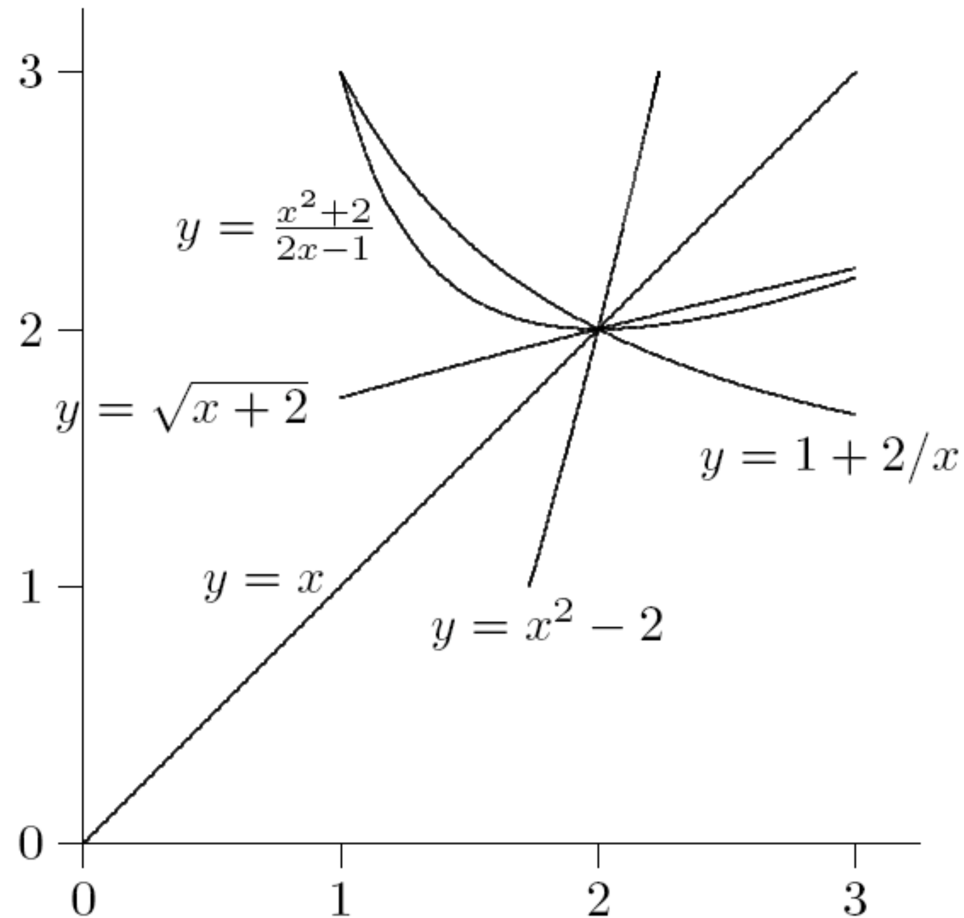
- $g(x) = x^2 - 2$
- $g(x) = \sqrt{x + 2}$
- $g(x) = 1 + 2/x$
- $g(x) = \dfrac{x^2 + 2}{2x - 1}$

are solutions to equation $f(x) = 0$



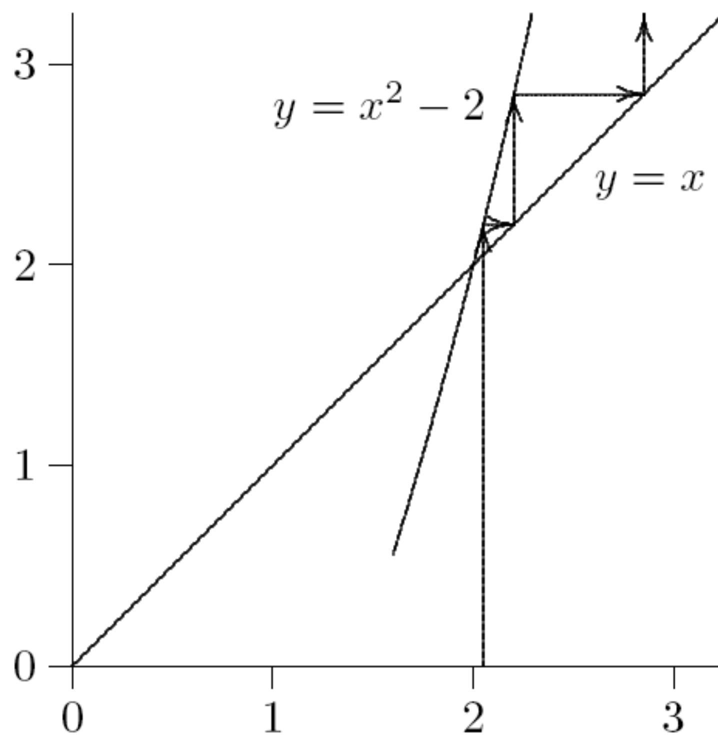
That is, when g(x*) = x*, then f(x*) = 0.

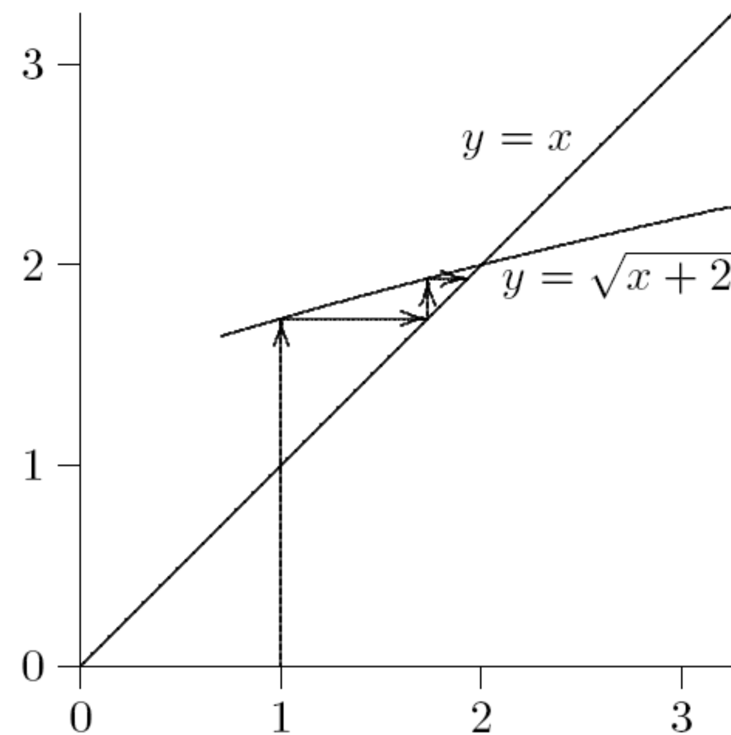
Example: Fixed-Point Problems



Example: Fixed-Point Iteration

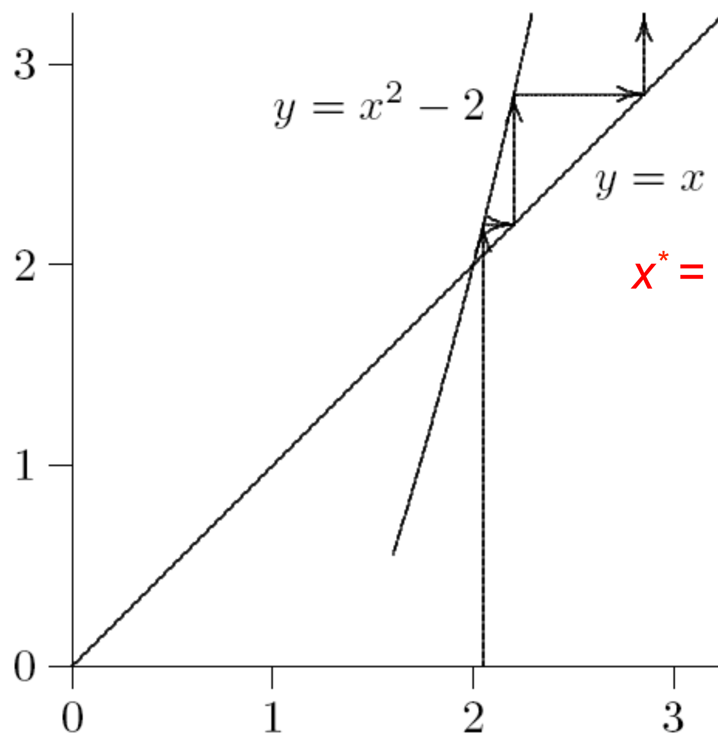
Goal: $x^* = g(x^*)$, $x_{k+1} = g(x_k)$, $x_k \to x^*$

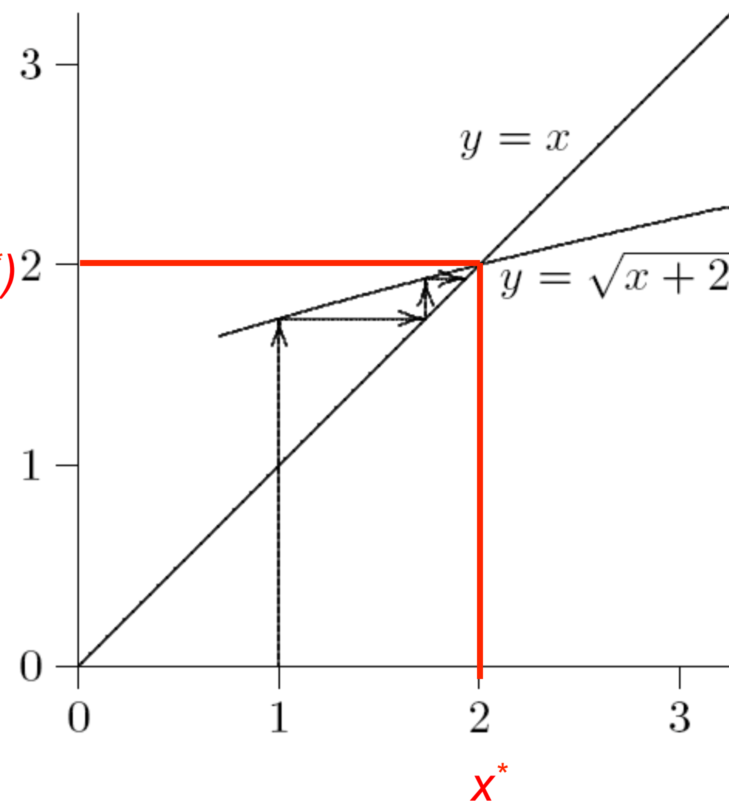




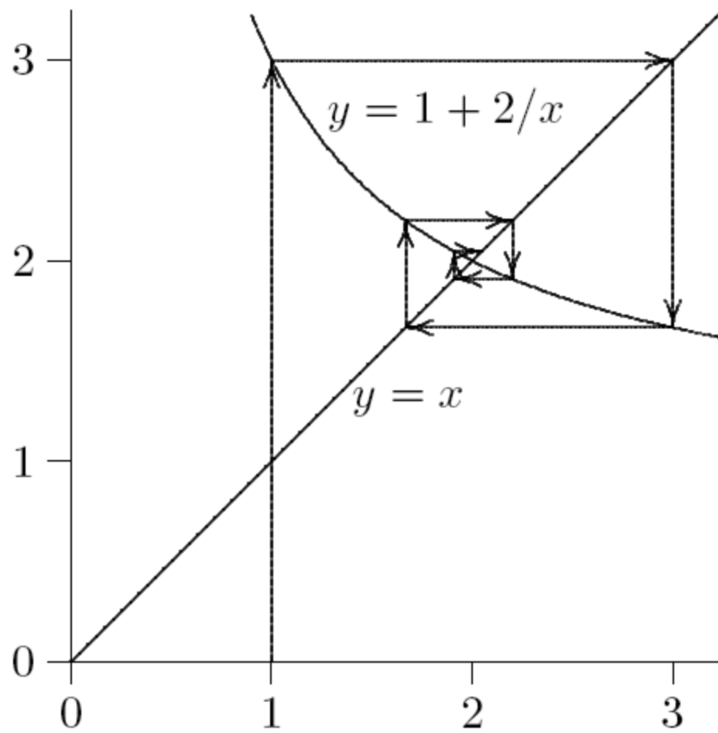
Example: Fixed-Point Iteration

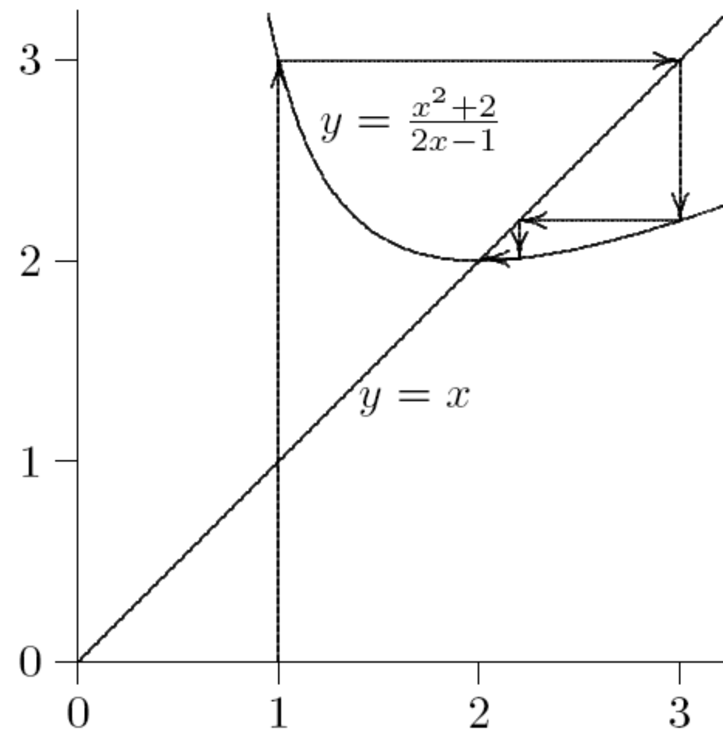
Goal: $x^* = g(x^*)$, $x_{k+1} = g(x_k)$, $x_k \to x^*$



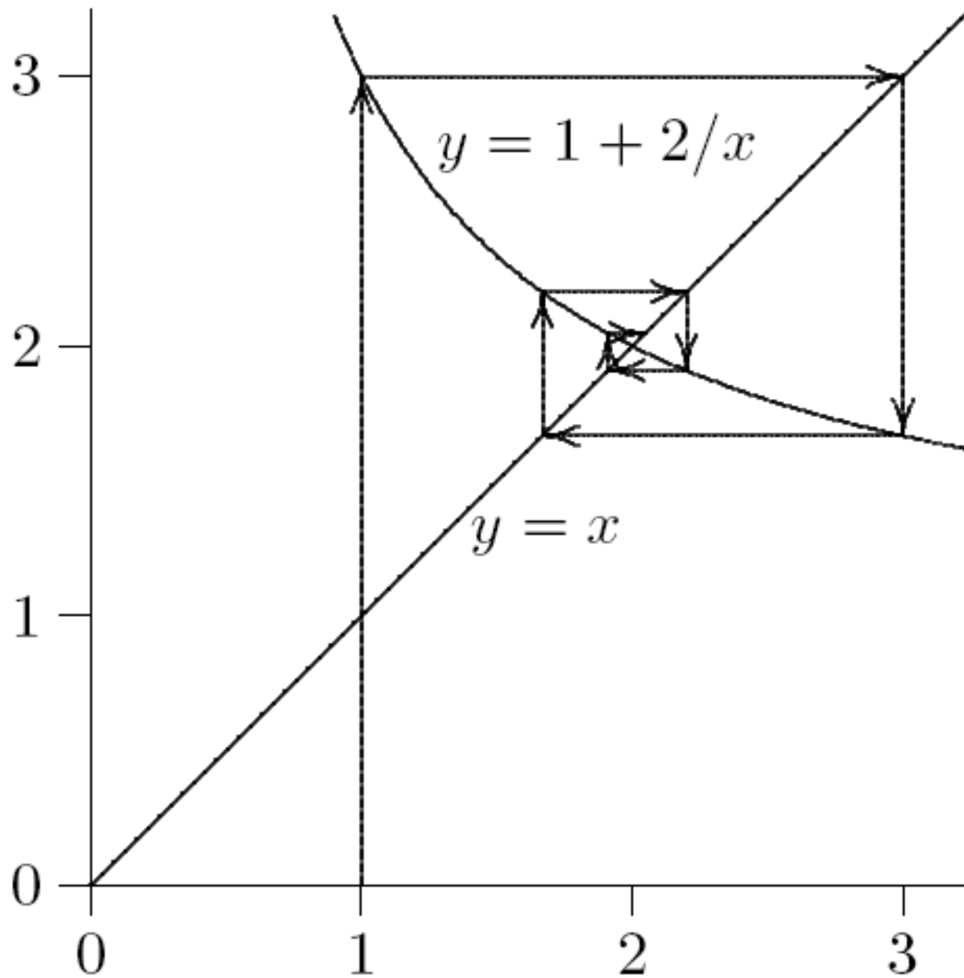


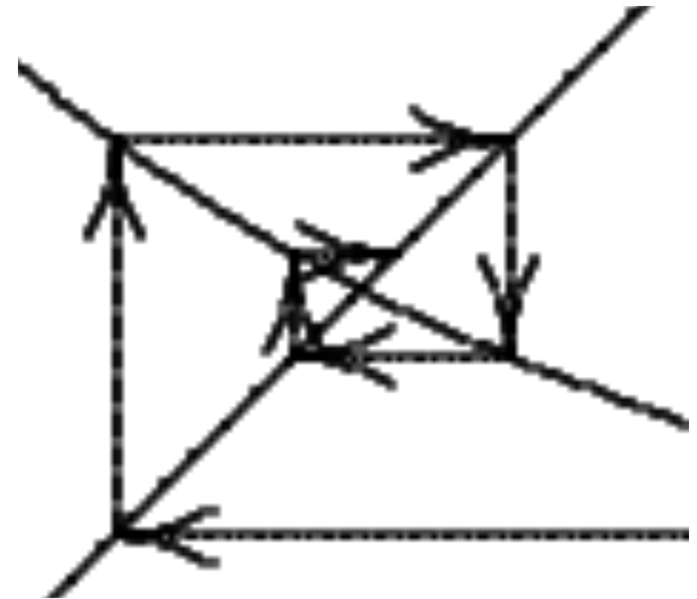
Example: Fixed-Point Iteration



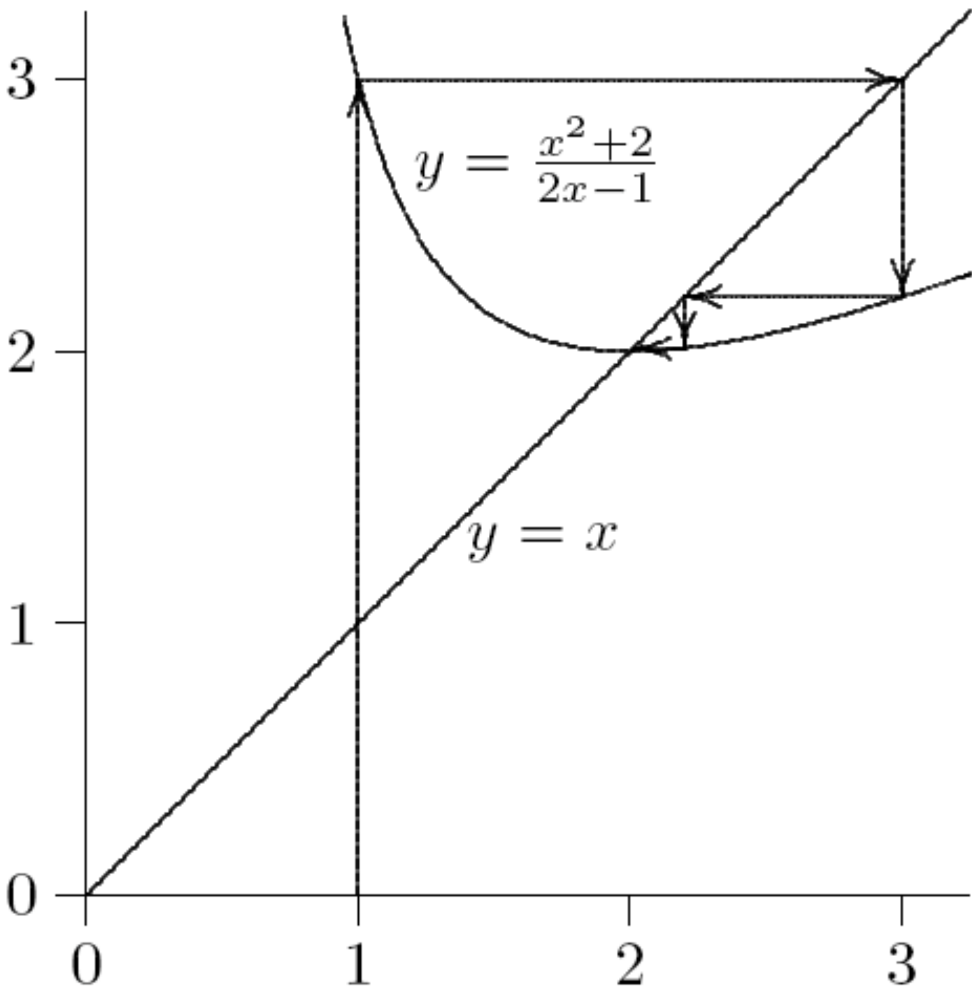


Close to x^*, Fixed Point Iteration is Linear Unless $g'(x^*)=0$





Close to x^*, Fixed Point Iteration is Linear Unless $g'(x^*)=0$

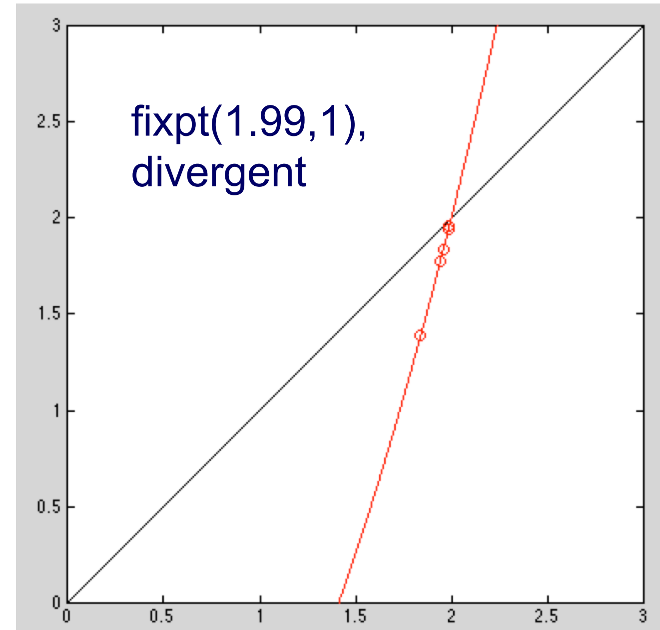


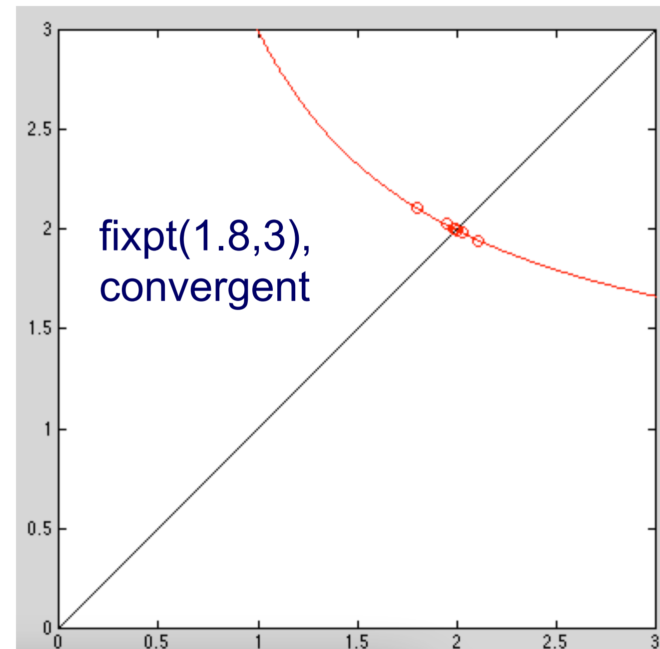
Convergence of Fixed-Point Iteration

- Matlab examples:
  - fixpt1.m
  - fixpt.m

fixpt.m

```
function fixpt(x0,icase)

x=0:.001:3; y=g(x,icase); xinf = 2;

hold off;
plot(x,x,'k-',x,0*x,'k-',0*x,x,'k-',x,y,'r-');
axis([0 3 0 3]);hold on;

xk=x0;
for k=1:10;
    ek(k)=abs(xk-xinf); kk(k)=k;
    y=g(xk,icase); plot(xk,y,'ro'); pause
    xk=y;
end;
format longe; [kk ek]
hold off
semilogy(kk,ek,'k.-')

function y=g(x,icase);

if icase==1; y=x.*x-2;                        end;
if icase==2; y=sqrt(x+2);                     end;
if icase==3; y=1+2./x;                        end;
if icase==4; y=(x.*x+2)./(2*x-1);             end;
```



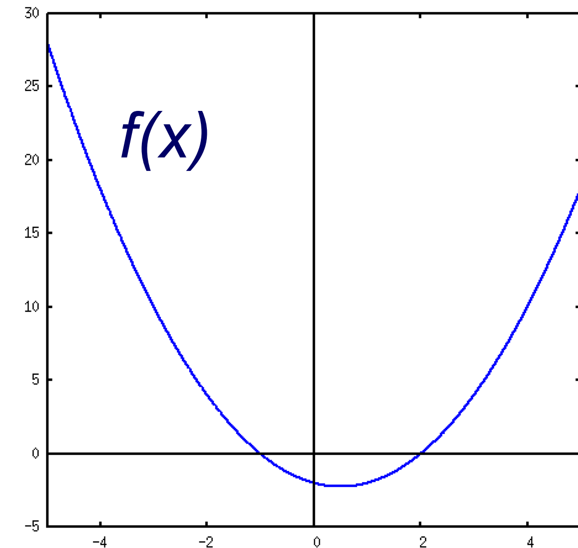


Example: Fixed-Point Problems

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

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

are solutions to equation $f(x) = 0$



Convergence of Fixed-Point Iteration

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

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

  converges to x^* if started within that interval

- If $|g'(x^*)| > 1$, then 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

Note: $g'(x^*) = 0$ → rapid convergence (good)

$f'(x^*) = 0$ → ill-conditioned, slow convergence (bad)

Convergence of Fixed-Point Iteration

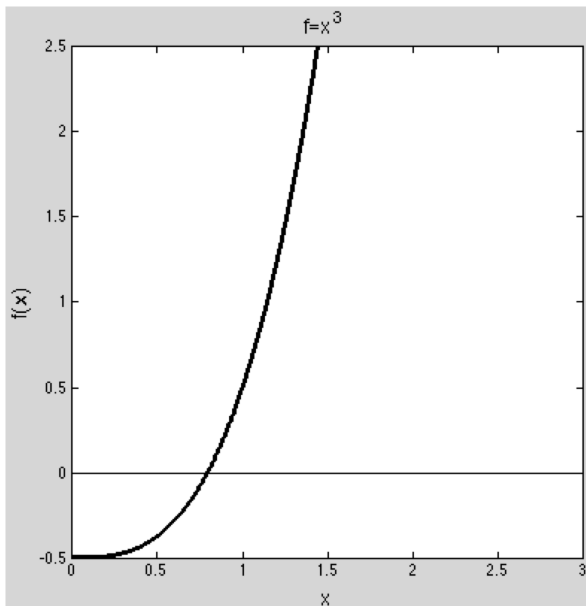
- Q: What is the iteration history for $g(x)=5$?

Convergence of Fixed-Point Iteration

- Q: Why does $g'(x^*)$ matter, but not any of the curvature information?
  (Unless of course if $g'(x^*)=0$.)

- With linear convergence, most of the time is spent evaluating g(x) **near** the root.

- Fast methods exploit the fact that g (or f) ***appear linear*** near the point of interest. This is the essence of *Taylor's Theorem*.



The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.

The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.

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

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

end

Accelerated Iteration

- Start with x_0.

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

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

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

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

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

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;
```

Matlab demo: aitken.m

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;
```

	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

Methods for Root Finding: $f(x^*) = 0$

- We return to the 1D root-finding problem, $f(x^*) = 0$.

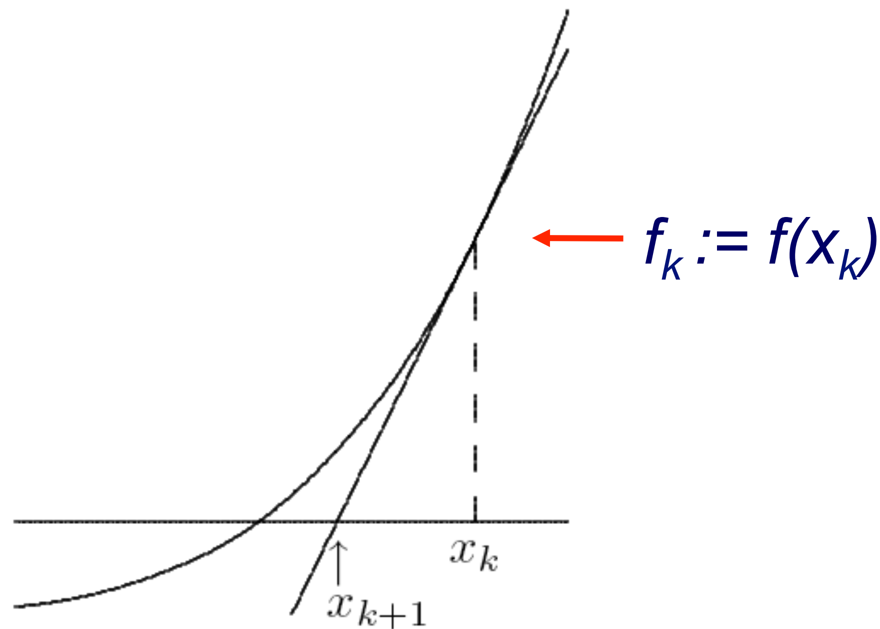
- We start with the most famous fixed-point scheme, ***Newton's method.***

- If x^* is a simple root, Newton's method converges quadratically.

- If x^* is a root with ***multiplicity m > 1***, the convergence is ***linear*** with contraction rate $C = [\ 1 – 1/m\ ]$.

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 of h approximating f near x

- Replace nonlinear function f by this linear function, whose zero is $h = -f(x)/f'(x)$

- Zeros of original function and linear approximation are not identical, so repeat process, giving *Newton's method*

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

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 scheme 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 value, we obtain

x	$f(x)$	$f'(x)$	h
3.000000	8.435520	9.959970	−0.846942
2.153058	1.294772	6.505771	−0.199019
1.954039	0.108438	5.403795	−0.020067
1.933972	0.001152	5.288919	−0.000218
1.933754	0.000000	5.287670	0.000000

newton.m demo

Convergence of Newton's Method

- Newton's method transforms nonlinear equation $f(x) = 0$ into fixed-point problem $x = g(x)$, where

  $$g(x) = x - f(x)/f'(x)$$

  and hence

  $$g'(x) = f(x)f''(x)/(f'(x))^2$$

- If x^* is simple root (i.e., $f(x^*) = 0$ and $f'(x^*) \neq 0$), then $g'(x^*) = 0$
- Convergence rate of Newton's method for simple root is therefore *quadratic* ($r = 2$)
- But iterations must start close enough to root to converge

Question

- Newton's method is:

$$x_k = x_{k-1} - f(x_{k-1}) / f'(x_{k-1})$$

- Will Newton's method converge if you make a mistake in evaluating $f'(x_{k-1})$??

Newton's Method, continued

For multiple root, convergence rate of Newton's method is only linear, with constant $C = 1 - (1/m)$, where m is multiplicity

k	$f(x) = x^2 - 1$	$f(x) = x^2 - 2x + 1$
0	2.0	2.0
1	1.25	1.5
2	1.025	1.25
3	1.0003	1.125
4	1.00000005	1.0625
5	1.0	1.03125

Examples of Newton's Method

- $x = \sqrt{A}$
  - Method 1
  - Method 2
- $x = 1/A$
- Interesting questions:
  - Convergent?
  - At what rate?
  - For what initial guess? (Use *range reduction* to get initial guess.)

1 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 $g'(x^*) = 0$.

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.

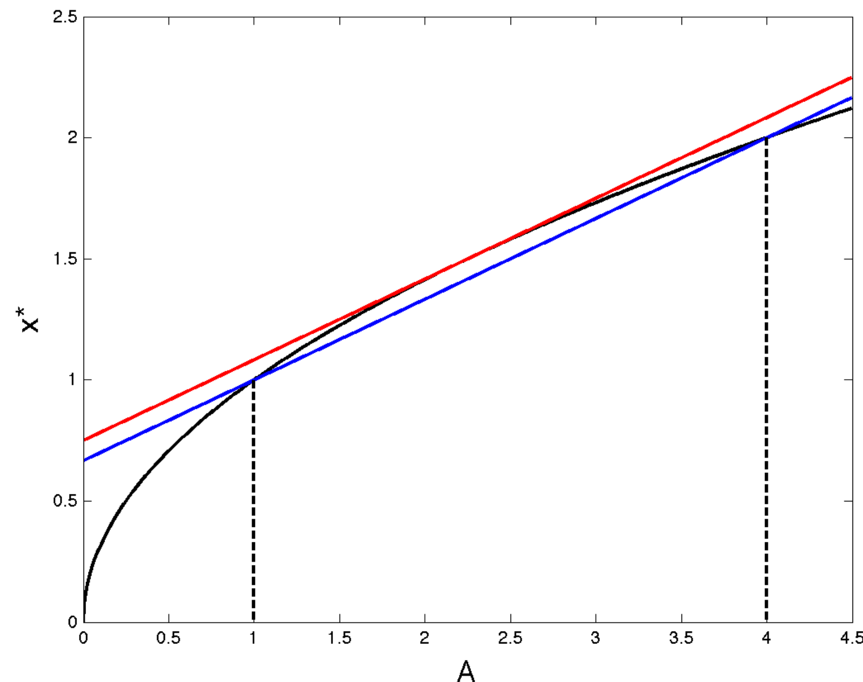


Figure 1: Plot of $x = \sqrt{A}$, $y_1 = 2/3 + A/3$, and $y_2 = 3/4 + A/3$.

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

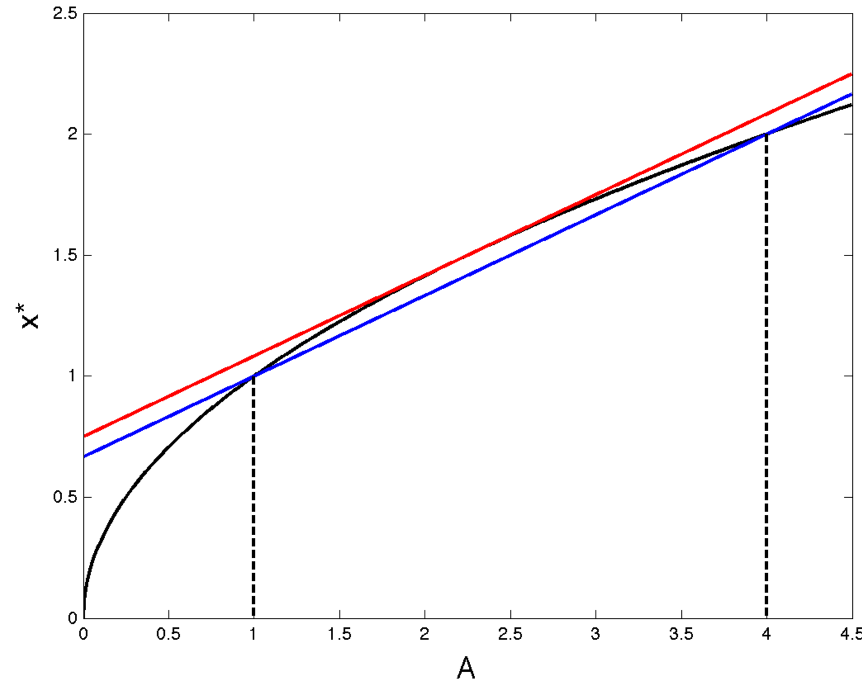


Figure 1: Plot of $x = \sqrt{A}$, $y_1 = 2/3 + A/3$, and $y_2 = 3/4 + A/3$.

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.

2 Newton for $1/A$.

While most machines are equiped with one or more fused multiply-add (FMA) units capable of producing (in pipelined fashion) one result per clock cycle, it is not uncommon for division to take multiple clock cycles because it generally requires some type of iteration. A classic example is the Intel's RISC processor, the i860, which required about 50 clock cycles to compute $c = a/b$. The respective assembly code for 32-bit and 64-bit division is shown in Figs. 2 and 3.

The scheme is based on Newton's method applied to

$$f(x) = \frac{1}{Ax} - 1$$

to arrive at the fixed-point iteration $x_{k+1} = g(x_k)$, with

$$\boxed{g(x) = 2x - Ax^2 = x(2 - Ax).}$$

We note that $g(1/A) = 1/A$ and $g'(1/A) = 2 - 2A(1/A)$=0, so the scheme has $x^* = 1/A$ as a fixed point and is quadratically convergent.

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

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

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

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

will guarantee convergence. If

$$A \quad = \quad 1.bbb\ldots \times 2^k,$$

then

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

We can take as an initial guess

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

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   flttwo, 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

Newton's Method for $A^{1/2}$

- Can you find a quadratically-convergent way of computing $A^{1/2}$ that does not require division on each iteration?

  (One division is OK.)

Methods for Root Finding: $f(x^*) = 0$

- We return to the 1D root-finding problem, $f(x^*) = 0$.
- We next consider the *secant method*, which is similar to Newton's method but does not require knowing ***f'(x)***.
- If x^* is a simple root, the secant method has superlinear convergence (but not quite quadratic).

Secant Method

- For each iteration, Newton's method requires evaluation of both function and its derivative, which may be inconvenient or expensive
- In *secant method*, derivative is approximated by finite difference using two successive iterates, so iteration becomes

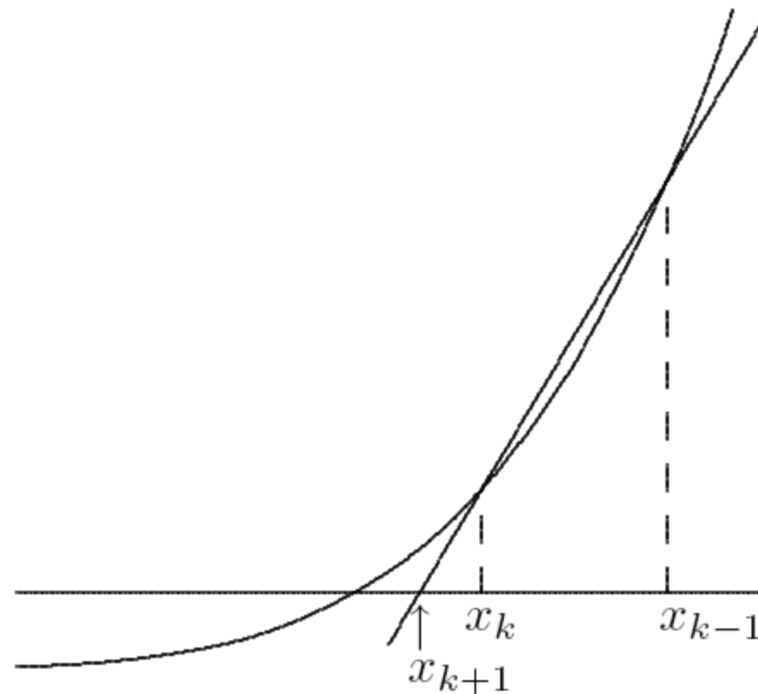
$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

≈ 1 / f'(x)

- Convergence rate of secant method is normally *superlinear*, with $r \approx 1.618$

Secant Method, continued

Secant method approximates nonlinear function f by secant line through previous two iterates



Convergence of Secant Method

- Convergence rate is $r \sim 1.62$, meaning

$$| e_{k+1} | \sim C \, | e_k |^r \qquad (\text{as } k \rightarrow \infty)$$

- Convergence behavior is $| e_{k+1} | \sim A \, | e_k | \, | e_{k-1} |$

- See secant.pdf on relate for notes.

Example: Secant Method

- Use secant method to find root of

$$f(x) = x^2 - 4\sin(x) = 0$$

- Taking $x_0 = 1$ and $x_1 = 3$ as starting guesses, we obtain

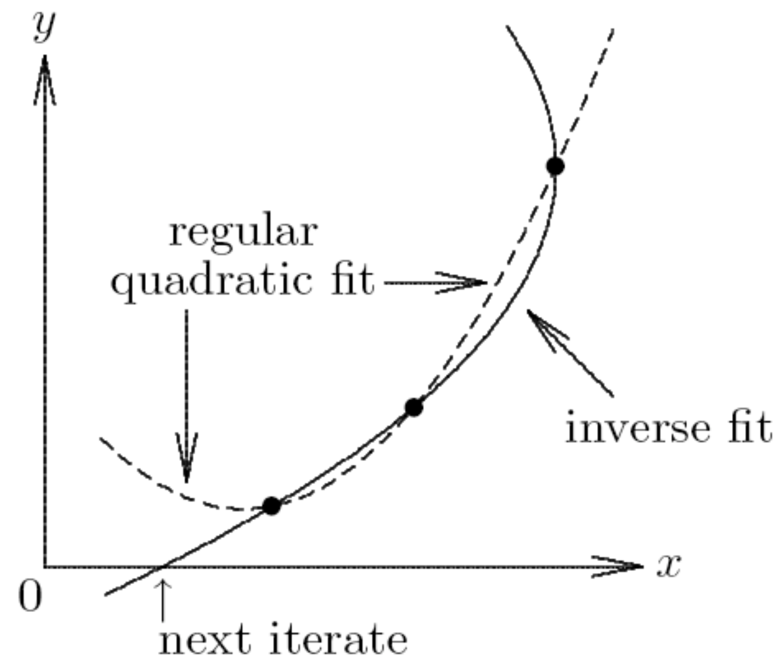
x	$f(x)$	h
1.000000	−2.365884	
3.000000	8.435520	−1.561930
1.438070	−1.896774	0.286735
1.724805	−0.977706	0.305029
2.029833	0.534305	−0.107789
1.922044	−0.061523	0.011130
1.933174	−0.003064	0.000583
1.933757	0.000019	−0.000004
1.933754	0.000000	0.000000

Higher-Degree Interpolation

- Secant method uses linear interpolation to approximate function whose zero is sought
- Higher convergence rate can be obtained by using higher-degree polynomial interpolation
- For example, quadratic interpolation (Muller's method) has superlinear convergence rate with $r \approx 1.839$
- Unfortunately, using higher degree polynomial also has disadvantages
  - interpolating polynomial may not have real roots
  - roots may not be easy to compute
  - choice of root to use as next iterate may not be obvious

Inverse Interpolation

- Good alternative is *inverse interpolation*, where x_k are interpolated as function of $y_k = f(x_k)$ by polynomial $p(y)$, so next approximate solution is $p(0)$
- Most commonly used for root finding is inverse quadratic interpolation



This method, however, can also have difficulty if not close enough to root.

Convergence rate is
r ~ 1.8
as with Mueller's method.

Inverse Quadratic Interpolation

- Given approximate solution values a, b, c, with function values f_a, f_b, f_c, next approximate solution found by fitting quadratic polynomial to a, b, c as function of f_a, f_b, f_c, then evaluating polynomial at 0
- Based on nontrivial derivation using Lagrange interpolation, we compute

$$u = f_b/f_c, \quad v = f_b/f_a, \quad w = f_a/f_c$$

$$p = v(w(u-w)(c-b) - (1-u)(b-a))$$

$$q = (w-1)(u-1)(v-1)$$

then new approximate solution is $b + p/q$

- Convergence rate is normally $r \approx 1.839$

Example: Inverse Quadratic Interpolation

- Use inverse quadratic interpolation to find root of

$$f(x) = x^2 - 4\sin(x) = 0$$

- Taking $x = 1$, 2, and 3 as starting values, we obtain

x	$f(x)$	h
1.000000	−2.365884	
2.000000	0.362810	
3.000000	8.435520	
1.886318	−0.244343	−0.113682
1.939558	0.030786	0.053240
1.933742	−0.000060	−0.005815
1.933754	0.000000	0.000011
1.933754	0.000000	0.000000

Linear Fractional Interpolation

- Interpolation using rational fraction of form

$$\phi(x) = \frac{x - u}{vx - w}$$

  is especially useful for finding zeros of functions having horizontal or vertical asymptotes
- ϕ has zero at $x = u$, vertical asymptote at $x = w/v$, and horizontal asymptote at $y = 1/v$
- Given approximate solution values a, b, c, with function values f_a, f_b, f_c, next approximate solution is $c + h$, where

$$h = \frac{(a - c)(b - c)(f_a - f_b)f_c}{(a - c)(f_c - f_b)f_a - (b - c)(f_c - f_a)f_b}$$

- Convergence rate is normally $r \approx 1.839$, same as for quadratic interpolation (inverse or regular)

Example: Linear Fractional Interpolation

- Use linear fractional interpolation to find root of

$$f(x) = x^2 - 4\sin(x) = 0$$

- Taking $x = 1$, 2, and 3 as starting values, we obtain

x	$f(x)$	h
1.000000	−2.365884	
2.000000	0.362810	
3.000000	8.435520	
1.906953	−0.139647	−1.093047
1.933351	−0.002131	0.026398
1.933756	0.000013	−0.000406
1.933754	0.000000	−0.000003

Safeguarded Methods

- Rapidly convergent methods for solving nonlinear equations may not converge unless started close to solution, but safe methods are slow
- Hybrid methods combine features of both types of methods to achieve both speed and reliability
- Use rapidly convergent method, but maintain bracket around solution
- If next approximate solution given by fast method falls outside bracketing interval, perform one iteration of safe method, such as bisection

Safeguarded Methods, continued

- Fast method can then be tried again on smaller interval with greater chance of success
- Ultimately, convergence rate of fast method should prevail
- Hybrid approach seldom does worse than safe method, and usually does much better
- Popular combination is bisection and inverse quadratic interpolation, for which no derivatives required

Zeros of Polynomials

- For polynomial $p(x)$ of degree n, one may want to find all n of its zeros, which may be complex even if coefficients are real
- Several approaches are available
  - Use root-finding method such as Newton's or Muller's method to find one root, deflate it out, and repeat
  - Form companion matrix of polynomial and use eigenvalue routine to compute all its eigenvalues

Note that standard polynomial forms are not very stable:

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \\ &= a_0\phi_0 + a_1\phi_1 + a_2\phi_2 + \cdots + a_n\phi_n, \end{aligned}$$

with $\phi_j(x)$ nearly linearly dependent for $j > 8$ or so.
Recall the Hilbert matrix example.
Might want to rethink the question.

Systems of Nonlinear Equations

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

- Wider variety of behavior is possible, so determining existence and number of solutions or good starting guess is much 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 increases rapidly with dimension of problem

Fixed-Point Iteration

- *Fixed-point problem* for $\boldsymbol{g}\colon \mathbb{R}^n \to \mathbb{R}^n$ is to find vector $\boldsymbol{x}$ such that

$$\boldsymbol{x} = \boldsymbol{g}(\boldsymbol{x})$$

- Corresponding *fixed-point iteration* is

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

- If $\rho(\boldsymbol{G}(\boldsymbol{x}^*)) < 1$, where ρ is *spectral radius* and $\boldsymbol{G}(\boldsymbol{x})$ is Jacobian matrix of $\boldsymbol{g}$ evaluated at $\boldsymbol{x}$, then fixed-point iteration converges if started close enough to solution
- Convergence rate is normally linear, with constant C given by spectral radius $\rho(\boldsymbol{G}(\boldsymbol{x}^*))$
- If $\boldsymbol{G}(\boldsymbol{x}^*) = \boldsymbol{O}$, then convergence rate is at least quadratic

Newton's Method

- In n dimensions, *Newton's method* has form

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

where $\boldsymbol{J}(\boldsymbol{x})$ is Jacobian matrix of $\boldsymbol{f}$,

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

- In practice, we do not explicitly invert $\boldsymbol{J}(\boldsymbol{x}_k)$, but instead solve linear system

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

for *Newton step* $\boldsymbol{s}_k$, then take as next iterate

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

Notes for Newton's Method in n Dimensions

Example: Newton's Method

- Use Newton's method to solve nonlinear system

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

- Jacobian matrix is $\boldsymbol{J}_f(\boldsymbol{x}) = \begin{bmatrix} 1 & 2 \\ 2x_1 & 8x_2 \end{bmatrix}$
- If we take $\boldsymbol{x}_0 = \begin{bmatrix} 1 & 2 \end{bmatrix}^T$, then

$$\boldsymbol{f}(\boldsymbol{x}_0) = \begin{bmatrix} 3 \\ 13 \end{bmatrix}, \quad \boldsymbol{J}_f(\boldsymbol{x}_0) = \begin{bmatrix} 1 & 2 \\ 2 & 16 \end{bmatrix}$$

- 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 $\boldsymbol{x}_1 = \boldsymbol{x}_0 + \boldsymbol{s}_0 = \begin{bmatrix} -0.83 & 1.42 \end{bmatrix}^T$

Example, continued

- Evaluating at new point,

$$\boldsymbol{f}(\boldsymbol{x}_1) = \begin{bmatrix} 0 \\ 4.72 \end{bmatrix}, \quad \boldsymbol{J}_f(\boldsymbol{x}_1) = \begin{bmatrix} 1 & 2 \\ -1.67 & 11.3 \end{bmatrix}$$

- Solving system $\begin{bmatrix} 1 & 2 \\ -1.67 & 11.3 \end{bmatrix} \boldsymbol{s}_1 = \begin{bmatrix} 0 \\ -4.72 \end{bmatrix}$ gives $\boldsymbol{s}_1 = \begin{bmatrix} 0.64 & -0.32 \end{bmatrix}^T$, so $\boldsymbol{x}_2 = \boldsymbol{x}_1 + \boldsymbol{s}_1 = \begin{bmatrix} -0.19 & 1.10 \end{bmatrix}^T$
- Evaluating at new point,

$$\boldsymbol{f}(\boldsymbol{x}_2) = \begin{bmatrix} 0 \\ 0.83 \end{bmatrix}, \quad \boldsymbol{J}_f(\boldsymbol{x}_2) = \begin{bmatrix} 1 & 2 \\ -0.38 & 8.76 \end{bmatrix}$$

- Iterations eventually convergence to solution $x^* = \begin{bmatrix} 0 & 1 \end{bmatrix}^T$

Convergence of Newton's Method

- Differentiating corresponding fixed-point operator

$$\boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{x} - \boldsymbol{J}(\boldsymbol{x})^{-1}\boldsymbol{f}(\boldsymbol{x})$$

  and evaluating at solution $\boldsymbol{x}^*$ gives

$$\boldsymbol{G}(\boldsymbol{x}^*) = \boldsymbol{I} - (\boldsymbol{J}(\boldsymbol{x}^*)^{-1}\boldsymbol{J}(\boldsymbol{x}^*) + \sum_{i=1}^{n} f_i(\boldsymbol{x}^*)\boldsymbol{H}_i(\boldsymbol{x}^*)) = \boldsymbol{O}$$

  where $\boldsymbol{H}_i(\boldsymbol{x})$ is component matrix of derivative of $\boldsymbol{J}(\boldsymbol{x})^{-1}$

- Convergence rate of Newton's method for nonlinear systems is normally *quadratic*, provided Jacobian matrix $\boldsymbol{J}(\boldsymbol{x}^*)$ is nonsingular

- But it must be started close enough to solution to converge

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 system costs $\mathcal{O}(n^3)$ operations

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 approximate Jacobian 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

- *Broyden's method* is typical secant updating method
- Beginning with initial guess $\boldsymbol{x}_0$ for solution and initial approximate Jacobian $\boldsymbol{B}_0$, following steps are repeated until convergence

$\boldsymbol{x}_0 =$ initial guess
$\boldsymbol{B}_0 =$ initial Jacobian approximation
for $k = 0, 1, 2, \ldots$
  Solve $\boldsymbol{B}_k\, \boldsymbol{s}_k = -\boldsymbol{f}(\boldsymbol{x}_k)$ for $\boldsymbol{s}_k$
  $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{s}_k$
  $\boldsymbol{y}_k = \boldsymbol{f}(\boldsymbol{x}_{k+1}) - \boldsymbol{f}(\boldsymbol{x}_k)$
  $\boldsymbol{B}_{k+1} = \boldsymbol{B}_k + ((\boldsymbol{y}_k - \boldsymbol{B}_k \boldsymbol{s}_k)\boldsymbol{s}_k^T)/(\boldsymbol{s}_k^T \boldsymbol{s}_k)$
end

Broyden's Method

- *Broyden's method* is typical secant updating method
- Beginning with initial guess $\boldsymbol{x}_0$ for solution and initial approximate Jacobian $\boldsymbol{B}_0$, following steps are repeated until convergence

$\boldsymbol{x}_0 =$ initial guess
$\boldsymbol{B}_0 =$ initial Jacobian approximation
for $k = 0, 1, 2, \ldots$
  Solve $\boldsymbol{B}_k\, \boldsymbol{s}_k = -\boldsymbol{f}(\boldsymbol{x}_k)$ for $\boldsymbol{s}_k$
  $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{s}_k$
  $\boldsymbol{y}_k = \boldsymbol{f}(\boldsymbol{x}_{k+1}) - \boldsymbol{f}(\boldsymbol{x}_k)$
  $\boldsymbol{B}_{k+1} = \boldsymbol{B}_k + ((\boldsymbol{y}_k - \boldsymbol{B}_k\boldsymbol{s}_k)\boldsymbol{s}_k^T)/(\boldsymbol{s}_k^T\boldsymbol{s}_k)$
end

Rank-One Update

Broyden's Method, continued

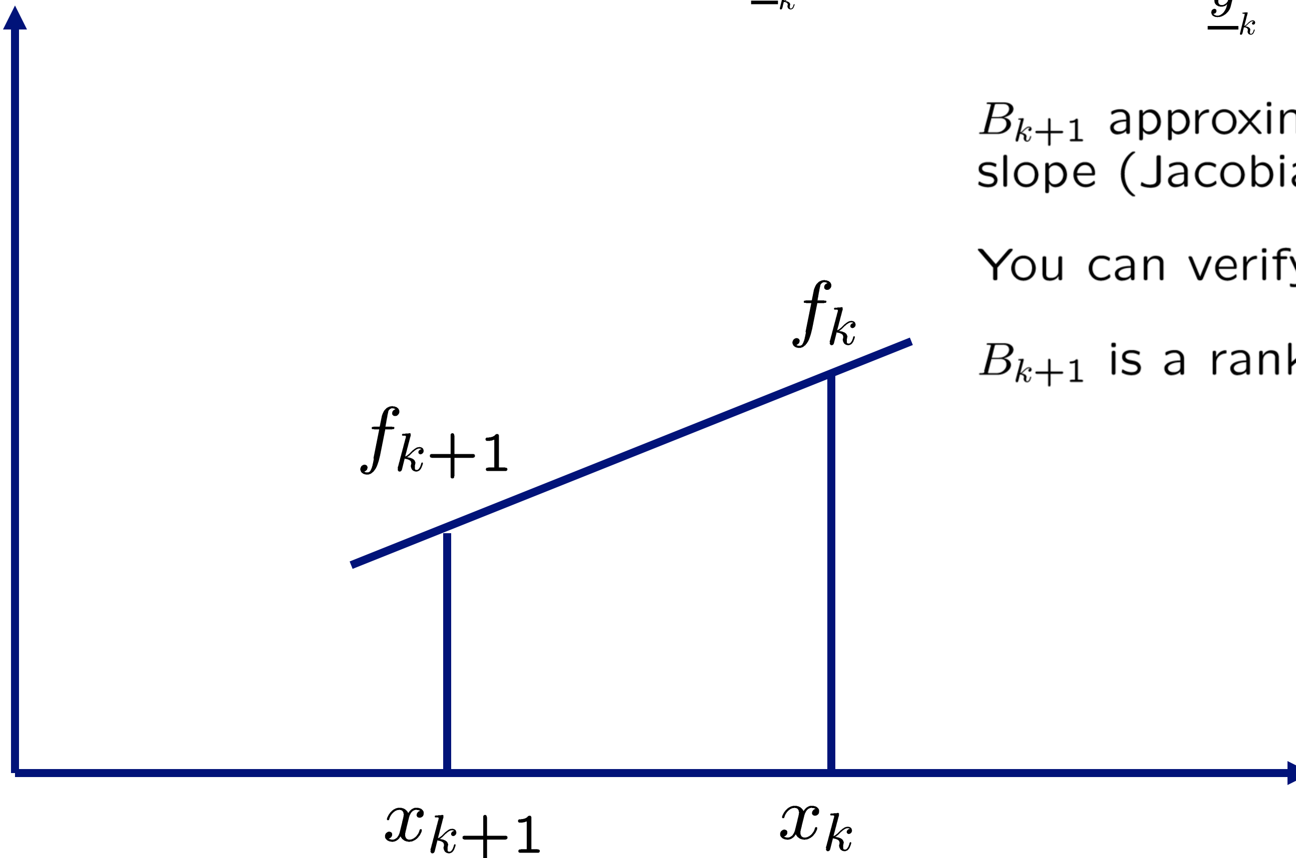
- Motivation for formula for $\boldsymbol{B}_{k+1}$ is to make least change to $\boldsymbol{B}_k$ subject to satisfying *secant equation*

$$\boldsymbol{B}_{k+1}(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) = \boldsymbol{f}(\boldsymbol{x}_{k+1}) - \boldsymbol{f}(\boldsymbol{x}_k)$$

- In practice, factorization of $\boldsymbol{B}_k$ is updated instead of updating $\boldsymbol{B}_k$ directly, so total cost per iteration is only $\mathcal{O}(n^2)$

Broyden – Choice of B_{k+1}

$$B_{k+1} \underbrace{\Delta \underline{x}_{k+1}}_{\underline{s}_k} = \underbrace{\Delta \underline{y}_{k+1}}_{\underline{y}_k}$$



B_{k+1} approximates the slope (Jacobian) at $\mathbf{x}_{k+1}$.

You can verify this.

B_{k+1} is a rank-one update to B_k.

Broyden Matrix Update

- $B_k \mathbf{s}_k \;=\; -\mathbf{f}(\mathbf{x}_k), \qquad \mathbf{x}_{k+1} \;=\; \mathbf{x}_k \;+\; \mathbf{s}_k.$

- Find rank-one update: $B_{k+1} = B_k + \mathbf{v}\mathbf{u}^T$ such that secant condition,

$$B_{k+1} \underbrace{(\mathbf{x}_{k+1} \;-\; \mathbf{x}_k)}_{\mathbf{s}_k} \;=\; \mathbf{f}(\mathbf{x}_{k+1}) \;-\; \mathbf{f}(\mathbf{x}_k),$$

  is satisfied:

$$\begin{aligned} \left(B_k \;+\; \mathbf{v}\mathbf{u}^T\right) \mathbf{s}_k \;&=\; \mathbf{f}_{k+1} \;-\; \mathbf{f}_k \\ \mathbf{v}\mathbf{u}^T \mathbf{s}_k \;&=\; \mathbf{f}_{k+1} \end{aligned}$$

- Take $\mathbf{v} \;=\; \mathbf{f}_{k+1}/(\mathbf{u}^T \mathbf{s}_k)$.

- Choose $\mathbf{u} \;=\; \mathbf{s}_k$ so denominator is not zero.

- $B_{k+1} \;=\; B_k \;+\; \dfrac{\mathbf{f}_{k+1}\mathbf{s}_k^T}{\mathbf{s}_k^T \mathbf{s}_k}.$

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, continued

- 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, continued

- 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 $\boldsymbol{x}^* = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

< interactive example >

broyden4.m

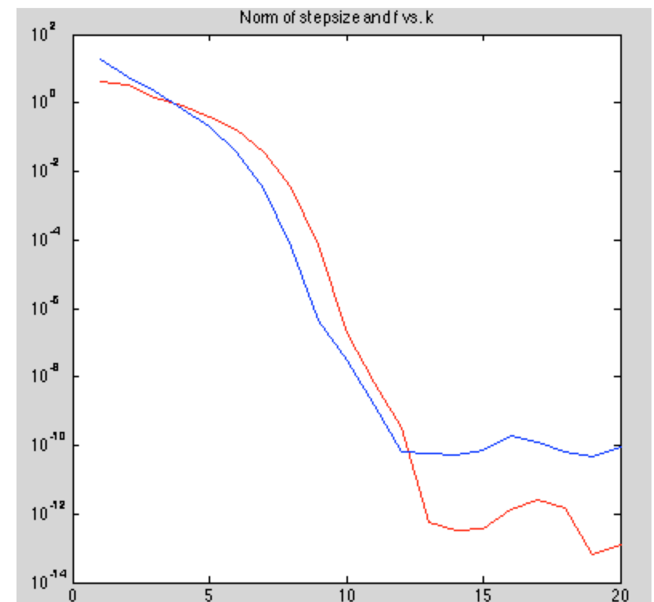
```
format compact; format longe; hold off
x=2*ones(2,1); f=0*x;

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_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
```



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 of Newton-like methods
- Simplest precaution is *damped Newton method*, in which new iterate is

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

  where $\boldsymbol{s}_k$ is Newton (or Newton-like) step and α_k is scalar parameter chosen to ensure progress 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

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

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

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

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

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 $\mathbf{f}(\mathbf{u}) = 0$,

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

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.

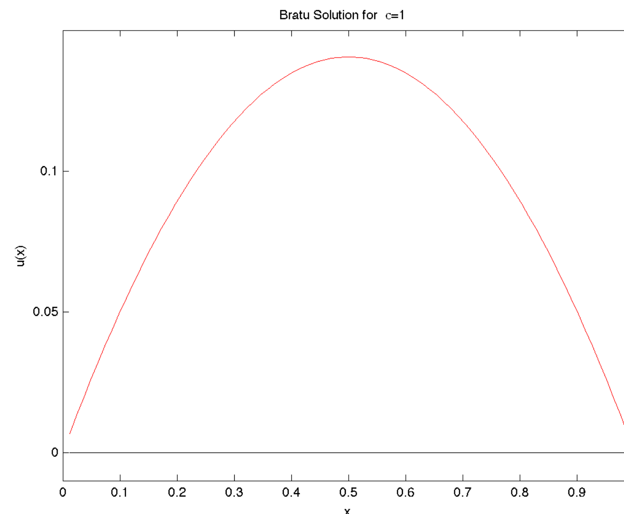


Figure 1: Solution of the Bratu Problem for $\sigma = 1$.

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
```

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]

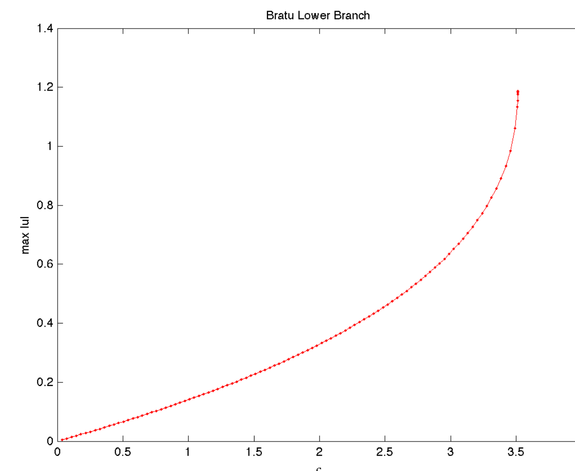


Figure 2: Lower branch of Bratu solutions vs σ.

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

4 JFNK: Reducing Factorization Costs

For large sparse Jacobians, the solution of

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

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

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

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

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

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

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

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

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

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.

JFNK: More Precise Jacobian Evaluation

- The finite difference approach for evaluating $J\mathbf{s}$,

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

  suffers from round-off.

- We are limited to $\epsilon \approx \sqrt{\epsilon_M}$
  (assuming that all vectors in question have norm ≈ 1).

- An alternative, suggested originally by Moler & Lyness, is to evaluate $\mathbf{f}(\mathbf{x}_k + i\epsilon\mathbf{s})$, with $i = \sqrt{-1}$.

- Assuming $\mathbf{f}$ is analytic, Taylor series:

$$\mathbf{f}(\mathbf{x}_k + i\epsilon\mathbf{s}) = \mathbf{f}(\mathbf{x}_k) + i\epsilon J(\mathbf{x}_k)\mathbf{s} + O(\epsilon^2) + iO(\epsilon^3).$$

- Take $J(\mathbf{x}_k)\mathbf{s}$ to be the imaginary part of the expression above:

$$\frac{1}{\epsilon}\mathcal{I}m\left[\mathbf{f}(\mathbf{x}_k + i\epsilon\mathbf{s})\right] = J(\mathbf{x}_k)\mathbf{s} + O(\epsilon^2)$$

- Without subtraction we avoid the usual round-off issues and can take $\epsilon \approx \sqrt{\epsilon_M}$.

Example: Imaginary Number Trick for Derivative Evaluation

cos_test.m

```
%
% Comparison of derivative via finite differences
% and imaginary number trick
%

h=1; x=1;  ue = cos(x); i=sqrt(-1.);
for k=1:100;
   xp = x+h; xm = x-h;
   fd = ( sin(xp)-sin(x) )/h;         % 1st-order FD
   f2 = ( sin(xp)-sin(xm) )/(2*h);    % 2nd-order FD
   fi = imag( sin(x+i*h) )/h;         % 2nd-order Imag
   hk(k)=h;
   ed(k)=abs(fd-ue)+1.e-20;
   e2(k)=abs(f2-ue)+1.e-20;
   ei(k)=abs(fi-ue)+1.e-20;
   h = h/1.41;
end;

loglog(hk,ed,'ro-',hk,e2,'go-',hk,ei,'b-','linewidth',1.2)
title('Comparison of Finite Difference and Imag Trick','fontsize',16)
xlabel('Stepsize, h','fontsize',16)
ylabel('Error','fontsize',16)
```