Chapter 4 - *Solution of Nonlinear Equations*

4.1 The Bisection Method

In this chapter, we will be interested in solving equations of the form

$$f(x) = 0.$$

Because $f(x)$ is not assumed to be linear, it could have any number of solutions, from 0 to ∞.

In one dimension, if $f(x)$ is continuous, we can make use of the Intermediate Value Theorem (IVT) to bracket a root; i.e., we can find numbers a and b such that $f(a)$ and $f(b)$ have different signs.

Then the IVT tells us that there is at least one magical value $x_* \in (a, b)$ such that $f(x_*) = 0$.

The number x_* is called a root or zero of $f(x)$.

Solving nonlinear equations is also called root-finding.

To “bisect” means to divide in half.

Once we have an interval (a, b) in which we know x_* lies, a systematic way to proceed is to sample $f(\frac{a+b}{2})$.

If $f(\frac{a+b}{2}) = 0$, then $x_* = \frac{a+b}{2}$, and we are done!

Otherwise, the sign of $f(\frac{a+b}{2})$ will either agree with the sign of $f(a)$, or it will agree with the sign of $f(b)$.

Suppose the signs of $f(\frac{a+b}{2})$ and $f(a)$ agree.

Then we are no longer guaranteed that $x_* \in (a, \frac{a+b}{2})$, but we are still guaranteed that $x_* \in (\frac{a+b}{2}, b)$.

So we have narrowed down the region where x_* lies.

Moreover, we can repeat the process by setting $\frac{a+b}{2}$ to a (or to b, as applicable) until the interval containing x_* is small enough.

See `bisectionDemo.m`

Interval bisection is a **slow-but-sure** algorithm for finding a zero of $f(x)$, where $f(x)$ is a real-valued function of a single real variable.

We assume that we know an interval $[a, b]$ on which a continuous function $f(x)$ changes sign.

However, there is likely no floating-point number (or even rational number!) where $f(x)$ is exactly 0.

So our goal is:

Find a (small) interval (perhaps as small as 2 successive floating-point numbers) on which $f(x)$ changes sign.

Sadly, bisection is *slow*!

It can be shown that bisection only adds 1 bit of precision per iteration.

Starting from 0 bits of accuracy, it always takes 52 steps to narrow the interval in which x_* lies down to 2 adjacent floating-point numbers.

However, bisection is *completely foolproof*.

If $f(x)$ is continuous and we have a starting interval on which $f(x)$ changes sign, then bisection is guaranteed to reduce that interval to two successive floating-point numbers that bracket x_*.

4.2 Newton's Method

Newton's method for solving $f(x) = 0$ works in the following fashion.

Suppose you have a guess x_n for a root x_*.

Find the tangent line to $y = f(x)$ at $x = x_n$ and follow it down until it crosses the x-axis; call the crossing point x_{n+1}.

This leads to the iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Often x_{n+1} will be closer to x_* than x_n was.

Repeat the process until we are close enough to x_*.

See `newtonDemo.m`

When Newton's method works, it is really great!

In fact, the generalization of the above description of Newton's method is the only viable general-purpose method to solve systems of nonlinear equations.

But, as a general-purpose algorithm for finding zeros of functions, it has 3 serious drawbacks.

1. The function $f(x)$ must be smooth.

2. The derivative $f'(x)$ must be computed.

3. The starting guess must be "sufficiently accurate".

• If $f(x)$ is not smooth, then $f'(x)$ does not exist, and Newton's method is not defined.

Estimating or approximating derivative values at points of non-smoothness can be hazardous.

• Computing $f'(x)$ may be problematic.

Nowadays, the computation of $f'(x)$ can (in principle) be done using automatic differentiation:

Suppose we have a function $f(x)$ and some computer code (in any programming language) to evaluate it.

By combining modern computer science parsing techniques with the rules of calculus (in particular the chain rule), it is theoretically possible to automatically generate the code for another function, `fprime(x)`, that computes $f'(x)$.

You may be able to use software that estimates $f'(x)$ by means of finite differences; e.g.,

$$f'(x) \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}.$$

If we could take $\lim_{\epsilon \to 0}$ of the right-hand side, then we would precisely have $f'(x)$. (Why can't we?)

But it still may be expensive or inconvenient depending on your computing environment.

• If x_0 is not sufficiently accurate, Newton's method will diverge (often catastrophically!).

The local convergence properties of Newton's method are its strongest features.

Let x_* be a zero of $f(x)$, and let $e_n = x_* - x_n$ be the error in the nth iterate.

Assume

- $f'(x)$ and $f''(x)$ exist and are continuous.
- x_0 is sufficiently close to x_*.

Then it is possible to prove that

$$e_{n+1} = \frac{1}{2}\frac{f''(\xi)}{f'(x_n)}e_n^2,$$

where ξ is some point between x_n and x_*.

In other words, the new error is roughly the size of the *square* of the old error.

(In this case we say $e_{n+1} = \mathcal{O}(e_n^2)$.)

This is called quadratic convergence:

For nice,[1] smooth functions, once x_n is close enough to x_*, the error goes down roughly by its square with each iteration.

$\Longrightarrow$ *The number of correct digits approximately **doubles** with each iteration!*

(This is much faster than bisection, which only has linear convergence.)

The behaviour we saw for computing $\sqrt{x}$ is typical.

Beware! When the assumptions underlying the local convergence theory are not satisfied, Newton's method might not work.

If $f'(x)$ and $f''(x)$ are not continuous and bounded, or if x_0 is not "close enough" to x_*, then the local theory does not apply!

$\rightarrow$ We might get slow convergence, or even no convergence at all.

[1] By "nice", here we mean $f(x)$ such that $f''(x)$ is bounded and $f'(x_*) \neq 0$.

4.3 Bad Behaviour of Newton's Method

As we have indicated, Newton's method does not always work.

Potentially bad behaviour of Newton's method includes

- convergence to an undesired root,
- periodic cycling,
- catastrophic failure.

• It is easy to see geometrically how Newton's method can converge to a root that you were not expecting.

Let $f(x) = x^2 - 1$.

This function has 2 roots: $x = -1$ and $x = 1$.

From the graph of $f(x)$ and the geometric interpretation of Newton's method, we see that we will get convergence to $x = -1$ for any $x_0 < 0$ and to $x = 1$ for any $x_0 > 0$.

(Newton's method is undefined in this case for $x_0 = 0$.)

• To see how Newton's method can cycle periodically, we have to cook up the right problem.

Suppose we want to see Newton's method cycle about a point a.

Then the iteration will satisfy

$$x_{n+1} - a = a - x_n.$$

Therefore, we write

$$x - \frac{f(x)}{f'(x)} - a = a - x,$$

and solve this equation for the evil $f(x)$:

$$\frac{df}{f} = \frac{dx}{2(x-a)}.$$

Integrating both sides,

$$\ln|f(x)| = \frac{1}{2}\ln|x-a| + C,$$

where C is an arbitrary constant, or

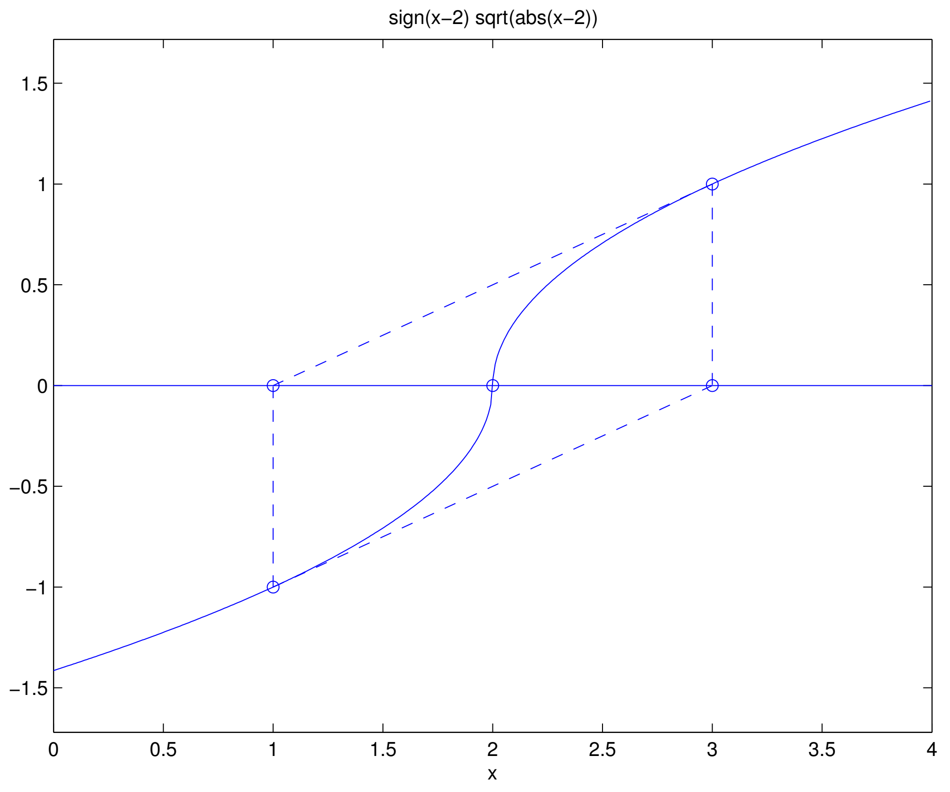
$$|f(x)| = A\sqrt{|x-a|},$$

where $A = e^C$. Noting

$$f(x) = \pm A\sqrt{|x-a|},$$

we can choose $A = \operatorname{sgn}(x-a)$.

Here is a plot of the $f(x)$ and the cyclic behaviour for $a = 2$ and $x_0 = 3$ or -1.



Of course, $x_* = a$.

What goes wrong?

The local convergence theory for Newton's method fails for this problem because $f'(x)$ is unbounded as $x \to a$.

• Newton's method can fail completely.

When it does, it is usually in a catastrophic fashion; i.e., the iterates x_n become numerically unbounded in only a few iterations.

(And if you are watching the iterates as they are computed, it becomes clear within an iteration or two that things aren't going to work.)

This bad behaviour is much more common when solving systems of nonlinear equations, so we defer more discussion until the end of this chapter.

What this tells us however is that having a good guess is even more important when solving a system of nonlinear equations.

(Unfortunately, it is also much harder to obtain one!)

For scalar equations, the problem of catastrophic failure generally stems from having $f'(x_n)$ "point in the wrong direction".

In practice, to enhance the global convergence properties of Newton's method, only part of the Newton "step" is taken when x_n is "far" from x_*:

The iteration is modified to

$$x_{n+1} = x_n - \lambda_n \frac{f(x_n)}{f'(x_n)}.$$

A computation is performed to compute a scaling factor $\lambda_n \in (0, 1]$ that tells us what fraction of the Newton correction to take.

As $x_n \to x_*$, $\lambda_n \to 1$;

i.e., as we approach the solution, we tend to take the whole Newton step (and get quadratic convergence).

This is called the damped Newton method.

The idea is that far from the root, you would not take the whole Newton step if it is bad (and hence avoid catastrophic failure).

Some other safeguards that are also added to software in practice to allow for graceful termination of the algorithm:

- Specify a maximum number of iterations allowed.
- Check that the Newton correction $f(x_n)/f'(x_n)$ is not too large (or alternatively λ_n is not too small).

4.4 The Secant Method

One of the greatest shortcomings of Newton's method is that it requires the derivative $f'(x)$ of the function $f(x)$ whose root we are trying to find.

The secant method replaces the derivative evaluation in Newton's method with a finite difference approximation based on the two most recent iterates.

Geometrically, instead of drawing a tangent to $f(x)$ at the current iterate x_n, you draw a straight line (secant) through the two points $(x_n, f(x_n))$ and $(x_{n-1}, f(x_{n-1}))$.

The next iterate x_{n+1} is again the intersection of this secant with the x-axis.

The iteration requires two starting values, x_0 and x_1.

The subsequent iterates are given by

$$\begin{aligned} s_n &= \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}, \\ x_{n+1} &= x_n - \frac{f(x_n)}{s_n}. \end{aligned}$$

It should be clear that the slope of the secant s_n approximates $f'(x_n)$ in Newton's method.

See `secantDemo.m`

The convergence properties of the secant method are similar to those of Newton's method. Assuming $f'(x)$ and $f''(x)$ are continuous, it can be shown that

$$e_{n+1} = \frac{1}{2} \frac{f''(\xi) f'(\xi_n) f'(\xi_{n-1})}{f'(\xi)^3} e_n e_{n-1},$$

where ξ_n, ξ_{n-1} are points between x_n, x_{n-1}, and x_*; ξ is a point in the interval in x corresponding to the interval in y spanned by $f(x_{n-1})$, $f(x_n)$, and 0.

This is not quadratic convergence, but it is superlinear convergence.

It can be shown that

$$e_{n+1} = \mathcal{O}(e_n^{\phi}),$$

where $\phi = (1+\sqrt{5})/2 \approx 1.6$ is the golden ratio.

In other words, when x_n gets close to x_*, the number of correct digits is multiplied by ϕ with each iteration.

That's almost as fast as Newton's method!

It is a great deal faster than bisection.

Typically, the secant method is very popular because although the convergence rate is not as fast as that of Newton's method (and so you need a few more iterations to reach a given accuracy), a secant iteration is usually much cheaper than a Newton iteration.

This means that ultimately the secant method is actually *faster* than Newton's method to find a root to a given accuracy.

4.5 Inverse Quadratic Interpolation

A typical train of thought in numerical analysis is the following.

We notice that the secant method uses 2 previous points x_n, x_{n-1} in determining the next one, x_{n+1}.

So we begin to wonder: is there anything to be gained by using 3 points?

Suppose we have 3 values, x_n, x_{n-1}, and x_{n-2}, and their corresponding function values, $f(x_n)$, $f(x_{n-1})$, and $f(x_{n-2})$.

We could interpolate these values by a parabola and take x_{n+1} to be the point where the parabola intersects the x-axis.

Problem: The parabola might not intersect the x-axis!

(This is because a quadratic function does not necessarily have real roots.)

Note: This could be regarded as an advantage!

In fact an algorithm known as Muller's method uses the complex roots of the quadratic to produce approximations to complex zeros of $f(x)$.

Unfortunately, we have to avoid complex arithmetic.

So instead of a quadratic in x, *we interpolate the 3 points with a quadratic function in* y!

This leads to a *sideways* parabola, $P(y)$, determined by the interpolation conditions

$$x_n = P(f(x_n)),\ x_{n-1} = P(f(x_{n-1})),\ x_{n-2} = P(f(x_{n-2})).$$

Note: We are reversing the traditional roles of the data x and y in this process!

A sideways parabola always hits the x-axis ($y = 0$).

So, $x_{n+1} = P(0)$ is the next iterate.

This method is known as inverse quadratic interpolation (IQI).

The biggest problem with this "pure" IQI algorithm is that polynomial interpolation assumes the x data (also called the abscissae), which in this case are $f(x_n)$, $f(x_{n-1})$, and $f(x_{n-2})$, to be *distinct*.

Doing things as we propose, we have no guarantee that they will be!

For example, suppose we want to compute $\sqrt{2}$ using $f(x) = x^2 - 2$.

If our initial guesses are $x_0 = -2,\ x_1 = 0,\ x_2 = 2$, then $f(x_0) = f(x_2)$ and x_3 is undefined!

Even if we start near this singular situation, e.g., with $x_0 = -2.001,\ x_1 = 0,\ x_2 = 1.999$, then $x_3 \approx 500$.

Cleve likens IQI to an immature race horse: It moves very quickly when it is near the finish line, but its overall behavior can be erratic.

$\rightarrow$ It needs a good trainer to keep it under control.

4.6 The Zero-in Algorithm

The zero-in algorithm is a hybrid algorithm that combines the reliability of bisection with the speed of secant and IQI.

The algorithm was first proposed by Dekker in the 1960s. It was later refined by Brent (1973).

The steps of the algorithm are as follows.

- Start with points a, b so that $f(a)$, $f(b)$ have opposite signs, $|f(b)| \leq |f(a)|$, and $c = a$.

- Use a secant step to give b^+ between a and b.

- Repeat the following steps until $|b - a| < \epsilon_{\text{machine}} |b|$ or $f(b) = 0$.
  - Set a or b to b^+ in such a way that
    - $f(a)$ and $f(b)$ have opposite signs;
    - $|f(b)| \leq |f(a)|$;
    - Set c to the *old* b if $b = b^+$; else set $c = a$.
  - If $c \neq a$, consider an IQI step.
  - If $c = a$, consider a secant step.
  - If the result of the IQI or secant step is in the interval $[a, b]$, take it as the new b^+.
  - If the step is not in the interval, use bisection to get the new b^+.

This algorithm is foolproof: It maintains in a shrinking interval that brackets the root.

It uses rapidly convergent methods when they are reliable; but it uses a slow-but-sure method when necessary to maintain a bracket for the root and guarantee convergence.

This is the algorithm implemented in MATLAB's `fzero` function.

See fzeroDemo.m and fzerogui.m

Solving Systems of Nonlinear Equations

We now consider systems of nonlinear equations.

We are now looking for a solution vector $\mathbf{x} = (x_1, x_2, \ldots, x_m)^T$ that satisfies a set of m nonlinear equations $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, where

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} f_1(x_1, x_2, \ldots, x_m) \\ f_2(x_1, x_2, \ldots, x_m) \\ \vdots \\ f_m(x_1, x_2, \ldots, x_m) \end{pmatrix}.$$

This case is much more difficult than the scalar case for a number of reasons; e.g.,

- Much more complicated behaviour is possible, making analysis of existence and number of solutions much harder or even impossible.
- There is no concept of bracketing in higher dimensions, so foolproof methods that are guaranteed to converge to the correct solution do not exist.
- Computational costs increase exponentially with increasing dimension.

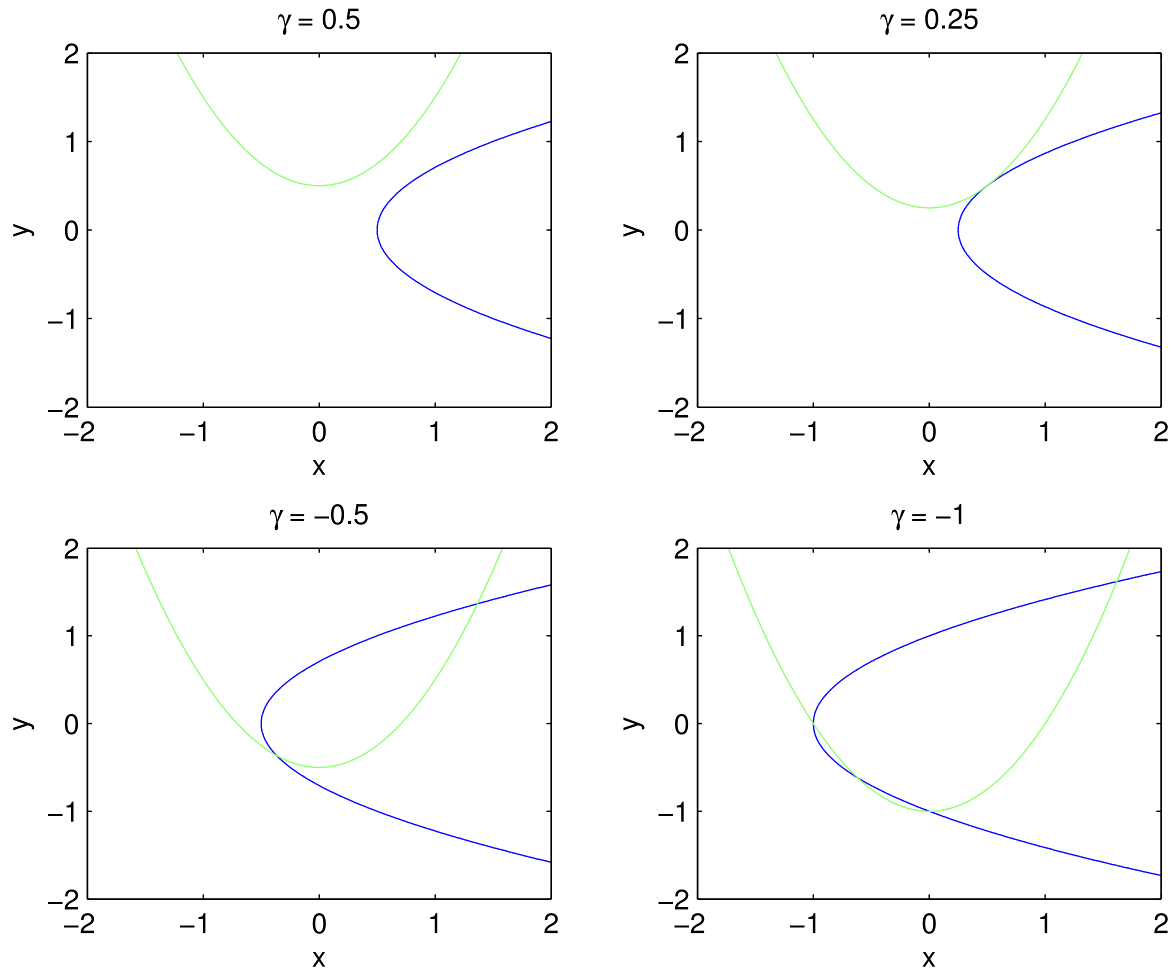
Consider the following system of 2 equations:

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

where γ is a parameter.

Each equation defines a parabola, and the solutions (if any) are the intersection points of these parabolas.

Depending on γ, this system can have $0, 1, 2,$ or 4 solutions.



Newton's Method

Many methods for finding roots in one dimension do not generalize directly to m dimensions.

Fortunately, Newton's method can generalize to higher dimensions quite easily.

It is arguably the most popular and powerful method for solving systems of nonlinear equations.

Before discussing it, we will first need to introduce the concept of the Jacobian matrix (also known as the matrix of first partial derivatives).

Let

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} f_1(x_1, x_2, \ldots, x_m) \\ f_2(x_1, x_2, \ldots, x_m) \\ \vdots \\ f_m(x_1, x_2, \ldots, x_m) \end{pmatrix}.$$

Then the Jacobian matrix $\mathbf{J}_\mathbf{f}(\mathbf{x}) = \frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}}$ is an $m \times m$ matrix whose (i, j) element is given by

$$[\mathbf{J}_\mathbf{f}(\mathbf{x})]_{ij} = \frac{\partial f_i}{\partial x_j}.$$

e.g., when $m = 2$,

$$\mathbf{J}_\mathbf{f}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix}.$$

Let

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} x_1^2 - x_2 + \gamma \\ -x_1 + x_2^2 + \gamma \end{pmatrix}.$$

Then,

$$\mathbf{J}_\mathbf{f}(\mathbf{x}) = \begin{bmatrix} 2x_1 & -1 \\ -1 & 2x_2 \end{bmatrix}.$$

The Newton iteration for systems of nonlinear equations takes the form

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J_f}^{-1}(\mathbf{x}_n)\mathbf{f}(\mathbf{x}_n).$$

Of course, in practice we never actually form $\mathbf{J_f}^{-1}(\mathbf{x}_n)$!

Instead we solve the linear system

$$\mathbf{J_f}(\mathbf{x}_n)\mathbf{s}_n = -\mathbf{f}(\mathbf{x}_n),$$

then update

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{s}_n.$$

$\rightarrow$ Newton's method reduces the solution of a nonlinear system to the solution of a sequence of linear equations.

Example

Solve the nonlinear system

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

First compute the Jacobian:

$$\mathbf{J_f}(\mathbf{x}) = \begin{bmatrix} 1 & 2 \\ 2x_1 & 8x_2 \end{bmatrix}.$$

Start with initial guess $\mathbf{x}_0 = [1, 2]^T$.

Then,

$$\mathbf{f}(\mathbf{x}_0) = \begin{bmatrix} 3 \\ 13 \end{bmatrix}, \qquad \mathbf{J_f}(\mathbf{x}_0) = \begin{bmatrix} 1 & 2 \\ 2 & 16 \end{bmatrix}.$$

Solve for the correction $\mathbf{s}_0$ from

$$\begin{bmatrix} 1 & 2 \\ 2 & 16 \end{bmatrix} \mathbf{s}_0 = - \begin{bmatrix} 3 \\ 13 \end{bmatrix} \implies \mathbf{s}_0 = \begin{bmatrix} -1.83 \\ -0.58 \end{bmatrix}.$$

So

$$\mathbf{x}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} -1.83 \\ -0.58 \end{bmatrix} = \begin{bmatrix} -0.83 \\ 1.42 \end{bmatrix},$$

etc.

If the process converges, we iterate until we have satisfied some stopping criterion.

As a compromise between checking for absolute and relative change in the iterates, we use a stopping criterion like

$$e = \|\mathbf{x}_{n+1} - \mathbf{x}_n\| + 4\epsilon_{\text{machine}} \|\mathbf{x}_{n+1}\|,$$

and stop when e is less than some tolerance (say 10^{-8}).

See systemDemo.m

Notes:

1. Newton's method is quadratically convergent is m dimensions, but the proof is beyond the scope of this course.

2. The computational expense of solving a system of m nonlinear equations can be substantial!
$\rightarrow$ computing $\mathbf{J_f}(\mathbf{x})$ requires m^2 function evaluations
$\rightarrow$ solving the linear systems by Gaussian elimination costs $\mathcal{O}(m^3)$ operations

These statements assume that the Jacobian is dense.

If it is sparse or has some special structure or properties, some savings can usually be had.

Quasi-Newton Methods

These are also called secant-updating methods.

The main computational cost of Newton's method in higher dimensions is in evaluating the Jacobian and solving the linear system

$$\mathbf{J}_{\mathbf{f}}(\mathbf{x}_n)\mathbf{s}_n = -\mathbf{f}(\mathbf{x}_n).$$

There are 2 main ways to reduce this cost:

1. only update the Jacobian every few iterations
   This is known as freezing the Jacobian.
   If you store the $\mathbf{LU}$ factors from the first linear system solve, the cost of solving subsequent systems with the frozen Jacobian is much less than the original cost ($\mathcal{O}(m^2)$ vs. $\mathcal{O}(m^3)$).
   If you only ever use $\mathbf{J}_{\mathbf{f}}(\mathbf{x}_0)$, this is known as the chord method.
   It is the analogue of the secant method in higher dimensions.

2. build up an approximation to the Jacobian using successive iterates and function values
   The most well-known of these methods is called Broyden's method.
   Unfortunately its description is beyond the scope of this course.

Again, more iterations are generally required of a quasi-Newton method compared to Newton's method.

However, the cost of these extra iterations is usually more than made up for with savings in costs per iteration.

Hence, there is often a net savings in using such methods over Newton's method.