5 Solving nonlinear equations

There should be no doubt that there is a great need to solve nonlinear equations with numerical methods. For most nonlinear equations, finding an analytical solution is impossible or very difficult. Just in one variable, we have equations such as $e^x = x^2$ which cannot be analytically solved. In multiple variables, the situation only gets worse. The purpose of this chapter is to study some common numerical methods for solving nonlinear equations. We will quantify how they work, when they work, how well they work, and when they fail.

5.1 Convergence criteria of iterative methods for nonlinear systems

This section will develop and discuss algorithms that will (hopefully) converge to solutions of a nonlinear equation. Each method we discuss will be in the form of:

Step 0: Guess at the solution with x_0 (or two initial guesses x_0 and x_1)

Step k: Use $x_0,\ x_1,\ x_2,\ ...,\ x_{k-1}$ to generate a better approximation x_k.

These algorithms all build sequences $\{x_k\}_{k=1}^{\infty} = \{x_0,\ x_1,\ x_2, ...\}$ which hopefully will converge to a solution x^*. As one might expect, some algorithms converge quickly, and some converge slowly or not at all. It is our goal to find rigorous criteria for when the discussed algorithms converge, and to quantify how quickly they converge when they are successful. Although there are several ways to determine how an algorithm is converging, it is typically best mathematically to measure error in the k^{th} iteration ($e_k = |x_k - x^*|$), that is, the error is the distance between x_k and the solution. Once this is sufficiently small, the algorithm can be terminated, and the last iterate becomes the solution.

We now define the notions of linear, superlinear, and quadratic convergence.

Definition 31. *Suppose an algorithm generates a sequence of iterates* $\{x_0,\ x_1,\ x_2,\ ...\}$ *which converges to* x^**. We say the algorithm converges linearly if*

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k} = \lim_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|} \leq \alpha < 1,$$

and that it converges superlinearly if

$$\lim_{k\to\infty} \frac{e_{k+1}}{e_k^p} = \lim_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|^p} \leq C,$$

where $p > 1$ (note that C need not be smaller than 1). If $p = 2$, the convergence is called quadratic.

To see the benefit of superlinear convergence, consider the error once the algorithm gets 'close' to the root, i.e. once $|x_k - x^*|$ gets small. Since for $p > 1$ we can write

$$\lim_{k\to\infty} \frac{|x_{k+1} - x^*|}{|x_k - x^*|} \leq C|x_k - x^*|^{p-1},$$

which allows us to observe that the term on the right, $C|x^k - x^*|^{p-1}$, becomes smaller with each iteration. This means that convergence to the limit becomes accelerated as the iteration progresses and will eventually converge faster than any linearly convergent method.

5.2 The bisection method

The bisection method is a very robust method for solving nonlinear equations in one variable. It is based on a special case of the Intermediate Value Theorem, which we recall now from first semester Calculus.

Theorem 32 (Intermediate Value Theorem)**.** *Suppose a function f is continuous on [a,b], and $f(a) \cdot f(b) < 0$. Then there exists a number c in the interval (a, b) such that $f(c) = 0$.*

The theorem says that if $f(a)$ and $f(b)$ are of opposite signs, and f is continuous, then the graph of f must cross the x-axis ($y = 0$) between a and b. Consider the following illustration of the theorem in Figure 5.1. In the top picture, two points are shown whose y-values have opposite signs. Do you think you can draw a continuous curve that connects these points without crossing the dashed line? No, you cannot! This is precisely what the Intermediate Value Theorem says.

Now that we have established that any continuous curve that connects the points must cross the dashed line, let's name the x-point where is crosses to be c, as in Figure 5.1. Note that a curve may cross multiple times, but we are guaranteed that it crosses at least once.

The bisection method for rootfinding is based on a simple idea and uses the intermediate value theorem. Consider two points, $(a, f(a))$, and $(b, f(b))$ from a

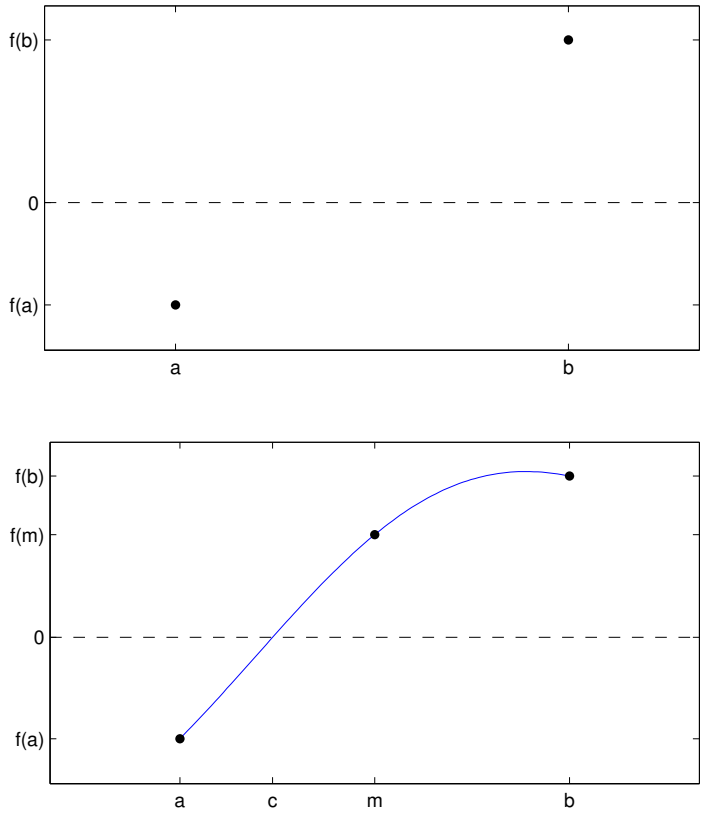

Fig. 5.1: Illustration of the Intermediate Value Theorem and the bisection method.

function/graph with $f(b)$ and $f(a)$ having opposite signs. Thus, we know that there is a root of f in the interval (a,b). Let m be the midpoint of the interval [a,b], so that $m = (a+b)/2$, and evaluate $f(m)$. One of the following is true: Either $f(m) = 0$, $f(a)$ and $f(m)$ have opposite signs, or $f(m)$ and $f(b)$ have opposite signs - it is simple to check which one. This means that we now know an interval that a root lives in, **which is half the width of the previous interval!** So if we repeat the process over and over, we can repeatedly reduce the size of the interval which we know contains a root. Do this enough times to "close in" on the root, and then the midpoint of that final interval can be considered a good estimate of a root. In a sense, this is the same basic concept as using your calculator to zoom in on a root. It iss just that we find where the curve crosses the axis with equations instead of a picture.

Let us now illustrate this process.

Example 33. *Use the bisection method to find the root of* $f(x) = \cos(x) - e^x$ *on [-2,-0.5].*

Clearly the function is continuous on the interval, and now we check that the function values at the endpoints have opposite signs:

$$f(-2) = -0.551482119783755 < 0 \quad f(-0.5) = 0.271051902177739 > 0.$$

Hence by the intermediate value theorem, we know that a root exists in [-2,-0.5], and we can proceed with the bisection algorithm.

Step 1: *Call m the midpoint of the interval, so $m = (-2 + (-0.5))/2 = -1.25$, and we evaluate $f(-1.25) = 0.028817565535079 > 0$. Thus, there is a sign change on [-2,-1.25], so we now know an interval containing a root is [-2,-1.25]. Hence, we have reduced in half the width of the interval.*

Step 2: *For our next step, call m the midpoint of [-2,-1.25], so $m = -1.625$ and then we calculate $f(-1.625) = -0.251088810231130 < 0$. Thus, there is a sign change in [-1.625,-1.25], so we keep this interval and move on.*

Step 3:*For our next step, call m the midpoint of [-1.625,-1.25], so $m = -1.4375$ and then we calculate $f(-1.4375) = -0.104618874642633 < 0$. Thus, there is a sign change in [-1.4375,-1.25], so we keep this interval and move on.*

Repeating this process enough will zoom in on the root -1.292695719373398. Note that, if the process is terminated after some number of iterations, we will know the maximum error between the last midpoint and true solution is half the length of the last interval.

Clearly, this process repeats itself on each step. We give below a code for the bisection process.

```
function [y,data] = bisect(a,b,func,tol)
% bisect(a,b,func,tol), uses the bisection method to find
% a root of func in (a,b) within tolerance tol

% evaluate the function at the end points
fa = func(a);
fb = func(b);

% Check that fa and fb have opposite signs
if fa*fb >= 0
    error('The function fails to satisfy f(a)*f(b)<0')
end

% keep track of progress of algorithm
data = [0 a b fa fb b-a];
```

```
% now we can run the algorithm
it_count = 0;
while (b-a) > tol
    it_count = it_count+1;
    % Get the midpoint and evaluate it
    xnew = (a+b)/2;
    fnew = func(xnew);

    % determine which interval to keep
    if fa * fnew <= 0
        b  = xnew;
        fb = fnew;
    else
        a  = xnew;
        fa = fnew;
    end

    datanew = [it_count a b fa fb b-a];
    data = [data;datanew];
end

% The solution needs to be a number
y = (a+b)/2;

end
```

Example 34. *Find a solution to* $e^x = x^2$ *using the bisection method to within a tolerance of* 10^{-6}.

The bisection method finds 0's of functions, so first we note that we will try to find a zero of the function $f(x) = e^x - x^2$. *For this, we define the function (here as an inline function):*

```
>> myfun1 = @(x) exp(x)-x^2;
```

Now we use our bisection function. Since $f(0) = 1 > 0$ *and* $f(-1) = e^{-1} - 1 < 0$, *we are guaranteed that bisection will find a root on this interval. So now run it:*

```
>> [root,data] = bisect(-1,0,myfun1,1e-6);
```

We can print out the root:

```
>> root

root =
```

```
  -7.0347e-01
```

and check that it really is a zero of the function:

```
>> myfun1(root)

ans =

  -8.0526e-07
```

Our code kept track of the data a, b, f(a), f(b), and b-a at each iteration. We can view this as well by displaying the Matlab variable 'data', which we omit; however, we note that it took 20 iterations for the interval width to shrink to below 10^{-6}.

Note that we did not need to run the algorithm to determine it would take 20 iterations. Since the interval width gets cut in half each time, the width of the interval-containing-a-root after n *steps is* $\frac{b-a}{2^n}$. *Hence, for the interval width to be less than* 10^{-6}, *we can solve*

$$\frac{0-(-1)}{2^n} \leq 10^{-6} \implies 10^6 \leq 2^n \implies n \geq \frac{\log(10^6)}{\log(2)} = 19.93.$$

Since the number of iterations is an integer, we conclude that 20 iterations will be the required amount for a tolerance of 10^{-6}.

In order to analyze the convergence of the bisection method, we must define what we mean by 'what is the k^{th} iterate x_k', since the algorithm continuously produces intervals, not points. Following the logic in Rocky IV, if confronted with an interval $[a_k,\ b_k]$ and you had to pick a single number from the interval, you should pick the one in the middle. Mathematically, this minimizes the maximum possible error, which is reasonable to do since we only know the root is in the interval, but we know nothing else about its location.

With this definition, we can consider the bisection algorithm as generating a sequence of approximations $\{x_0,\ x_1,\ ...,\ x_k,\ ...\}$, and we can show that this sequence converges linearly in the sense of the maximum possible error in bisection iterate x_k converges linearly.

Theorem 35. *Suppose* a_0 *and* b_0 *are given,* $f \in C^0([a_0, b_0])$, $f(a_0)f(b_0) < 0$, *and there is a unique root* $x^* \in [a_0, b_0]$. *Call* $[a_k,\ b_k]$ *the interval used at step* k, *and* $x_k = \frac{a_k+b_k}{2}$. *Denoting* ϵ_k *the maximum possible error in* x_k, *the sequence* $\{x_k\}$ *converges linearly to* x^* *in the sense that*

$$\lim_{k\to\infty} \frac{|\epsilon_k|}{|\epsilon_{k-1}|} = \frac{1}{2}.$$

Proof. The intermediate value theorem immediately implies that the root x^* must be in each interval $[a_k,\ b_k]$. Thus,

$$\epsilon_k = |x_k - x^*| \leq \frac{b_k - a_k}{2},$$

since x_k is the midpoint of $[a_k, b_k]$ and thus the farthest away the root could be is at the endpoints.

Now since the algorithm cuts the intervals in half with each iteration $|b_k - a_k| = \frac{1}{2}|b_{k-1} - a_{k-1}|$, we have that $\epsilon_k = \frac{1}{2}\epsilon_{k-1}$. Thus,

$$\frac{|\epsilon_k|}{|\epsilon_{k-1}|} = \frac{1}{2},$$

for any k, and so it immediately follows that

$$\lim_{k\to\infty} \frac{|\epsilon_k|}{|\epsilon_{k-1}|} = \frac{1}{2}.$$

□

5.3 Fixed point theory and algorithms

We consider now how to find solutions to $f(x) = 0$ by finding solutions to $g(x) = x$. Solutions x^* of the latter equation are called 'fixed points' because they are fixed with respect to the function g; when you plug x^* into g, you get back x^*.

At first, it may seem strange why we would consider the equation $g(x) = x$, but it turns out that the mathematical theory for finding solutions to fixed point problems is easier to decipher than for rootfinding. Of course, it is equivalent mathematically, e.g. finding the roots of $x^2 - x - 2 = 0$ is equivalent to finding the fixed points of $g(x) = x^2 - 2$, and also $g(x) = 1 + \frac{2}{x}$.

We will consider only the scalar case herein: $g : \mathbb{R} \to \mathbb{R}$. However, almost all of the theory generalizes to $\mathbb{R}^n$, and even general metric spaces, without significant difficulties.

We will denote by I a generic closed interval, i.e. $I = [a, b]$. We will assume the function $g : I \to I$ is continuous. In some of the theory that follows, g will have additional smoothness properties.

Define the **fixed point iteration** by

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

Much of this section will be concerned with when this algorithm converges to a fixed point, and how quickly. Clearly, the fixed point (FP) iteration is easy to

code up and use. The difficulty with fixed point methods usually comes from creating the function g for which one would get fast and robust convergence of the FP iteration.

Theorem 36 (Existence and uniqueness of a fixed point)**.** *Suppose I is a closed and bounded interval, $g : I \to I$ is continuous, and $g(I) \subseteq I$. Further, assume it holds for all $x \neq y \in I$ that $|g(x) - g(y)| \leq \alpha|x - y|$, with $0 \leq \alpha < 1$. Then there exists a unique fixed point $x^* \in I$ satisfying $g(x^*) = x^*$.*

Furthermore, the fixed point iteration $x_{k+1} = g(x_k)$ converges to the fixed point x^*, for any initial $x_0 \in I$.

Remark 37. *The property that $|g(x) - g(y)| \leq \alpha|x - y|$ with $0 \leq \alpha < 1$ on I makes g a 'contraction on I'. That is, it makes the distance between points smaller. One can think that if you repeatedly squeeze 2 points closer and closer, then eventually they must converge to a point.*

Proof. We first prove uniqueness of a fixed point for g in I. Suppose there are 2 fixed points, x_1^* and x_2^*. Then $g(x_1^*) = x_1^*$ and $g(x_2^*) = x_2^*$, so

$$|g(x_1^*) - g(x_2^*)| = |x_1^* - x_2^*|.$$

But since $x_1^*, x_2^* \in I$, we also know from the contractive property of g that if $x_1^* \neq x_2^*$, then

$$|g(x_1^*) - g(x_2^*)| < |x_1^* - x_2^*|,$$

which is a contradiction. Thus, $x_1^* = x_2^*$, and we conclude that a fixed point of g on I must be unique.

For existence, consider a sequence generated by picking any $x_0 \in I$ and then defining $\{x_k\}_{k=1}^{\infty}$ by the fixed point iteration, $x_k = g(x_{k-1})$. Note that since $g(I) \subseteq I$, the entire sequence $\{x_k\} \subseteq I$. Using the contraction property of g, we have that

$$\begin{aligned} |x_k - x_{k-1}| &= |g(x_{k-1}) - g(x_{k-2})| \\ &\leq \alpha|x_{k-1} - x_{k-2}| \\ &\leq \alpha^2|x_{k-2} - x_{k-3}| \\ &\leq ... \\ &\leq \alpha^{k-1}|x_1 - x_0|. \end{aligned}$$

Since $\alpha < 1$ and $|x_1 - x_0|$ is fixed, this implies that the iterates get closer and closer together as k gets large. In Advanced Calculus, we refer to such a sequence as a contractive sequence. Also from Advanced Calculus, we know a contractive

sequence on a closed interval must converge to a point in that interval. Calling this point x^*, we have established the existence of a point $x^* \in I$ such that $x_k \to x^*$.

Using that $x_k = g(x_{k-1})$,

$$x_k \to x^* \implies g(x_k) \to x^*,$$

since $\{g(x_k)\}$ is the same sequence as $\{x_k\}$, with the indexing shifted by 1.

Also, since g is continuous on I,

$$x_k \to x^* \implies g(x_k) \to g(x^*).$$

Finally, since limits are unique, we have that $g(x^*) = x^*$. Since we have already established that $x^* \in I$, we have proven the existence of an $x^* \in I$ satisfying $g(x^*) = x^*$; i.e. we have established the existence of a fixed point in I.

The last claim of the theorem is already proven in the construction of the proof.

□

Let's consider now some examples.

Example 38. *The function $g(x) = x^2$ has a fixed point in $[0, 1/4]$ but not in $(0, 1/4]$. It is clear that g is continuous and is a contraction on both $[0, 1/4]$ and $(0, 1/4)$ since for $0 \le x, y \le \frac{1}{4}$,*

$$|g(x) - g(y)| = |x^2 - y^2| = |x + y||x - y| \le \frac{1}{2}|x - y|.$$

However, Theorem 36 does not apply to the interval $(0, 1/4]$ since it is not closed. The fact that the interval was closed was instrumental in the proof, since this is what allowed us to say the sequence converged to a limit in the interval.

Clearly the fixed point of $g(x)$ on $[0, 1/4]$ is 0. Consider the fixed point iteration, using $x_0 = \frac{1}{4}$. We calculate the first several iterations to be

```
0.0625
0.0039062
1.5259e-05
2.3283e-10
5.421e-20
2.9387e-39
8.6362e-78
7.4583e-155
5.5627e-309
0
0
0
```

Example 39. *The function $g(x) = x^2$ has 2 fixed points on the interval $[0, 1]$. However, the theorem does not apply since g is not a contraction on all of $[0, 1]$ since for $x = 0$ and $y = 1$, $|g(x) - g(y)| = |x - y| \not< |x - y|$.*

Theorem 40. *Suppose x^* is a fixed point of g, $g \in C^1(B_\epsilon(x^*))$, and $|g'(x^*)| < 1$ in $B_\epsilon(x^*)$. Then for x_0 chosen close enough to x^*, the fixed point algorithm $x_{k+1} = g(x_k)$ converges to x^*.*

Proof. We will prove this result using Theorem 36. To do so, we must show that there is an interval I such that $g(I) \subseteq I$, and there is a number $0 \leq \alpha < 1$ such that for every $x \neq y$ in I, $|g(x) - g(y)| \leq \alpha|x - y|$.

We start by proving that $g(I) \subseteq I$. Since $g' \in C^0(B_\epsilon(x^*))$ and $|g'(x^*)| < 1$, there exists a radius $\delta < \epsilon$ such that $\max_{x^*-\delta \leq x \leq x^*+\delta} |g'(x)| < 1$ (e.g. δ could be $\frac{\epsilon}{2}$). Set $I = [x^* - \delta, x^* + \delta]$, and set $\alpha := \max_{x \in I} |g'(x)|$ (note the max must exist since g' is continuous and I is a closed interval). Since $I \subset B_\epsilon(x^*)$, we have that $|g'(x)| \leq \alpha < 1$ on I.

We claim that $g(I) \subseteq I$, and prove by contradiction. Suppose it is not true. Then there is a $z \in I$ with $g(z) \notin I$, i.e. $g(z) > x^* + \delta$ or $g(z) < x^* - \delta$. Then $|g(z) - x^*| = |g(z) - g(x^*)| > |z - x^*|$, and so $\left|\frac{g(z)-g(x^*)}{z-x^*}\right| > 1$. Since $g \in C^1(\overline{B}_\delta(x^*))$, the mean value theorem implies the existence of $c \in (z, x^*)$ satisfying

$$g'(c) = \frac{g(z) - g(x^*)}{z - x^*}.$$

But this means we have $c \in I$ with $|g'(c)| > 1$, which is a contradiction. Hence we have established that $g(I) \subseteq I$.

We now prove that for every $x \neq y$ in I, $|g(x) - g(y)| \leq \alpha|x - y|$. Since $g \in C^1(I)$, we have from Taylor's theorem that for $x, y \in I$,

$$g(y) = g(x) + g'(r)(y - x),$$

for some r between x and y. Thus, $r \in I$, and so $|g'(r)| \leq \alpha < 1$. Using this and rearranging terms gives

$$|g(y) - g(x)| = |g'(r)(y - x)| \leq \alpha|y - x|.$$

We have now shown that the assumptions of Theorem 36 are satisfied, and so the conclusions of this theorem follow immediately, provided x_0 is chosen in I. □

Our next theorem shows that if we have a stronger contraction near the root, i.e. if $g'(x^*) = 0$ instead of being only less than 1, then the convergence is faster.

Theorem 41. *Suppose x^* is a fixed point of g, $g'(x^*) = 0$, and $g \in C^2(B_\epsilon(x^*))$. Then for x_0 chosen close enough to x^*, the fixed point algorithm $x_{k+1} = g(x_k)$ converges quadratically to x^*.*

Proof. Using Theorem 40, we immediately have that the fixed point iteration will converge to x^* for an initial guess x_0 chosen sufficiently close to x^*. We need only establish here that the convergence is quadratic.

Since $g \in C^2(B_\epsilon(x^*))$, Taylor's theorem gives us for $x \in C^2(B_\epsilon(x^*))$ that there exists c between x and x^* such that

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

Note that c is dependent on the choice of x.

Taking $x = x_k$ and using that $g(x^*) = x^*$ and $g'(x^*) = 0$, we obtain

$$g(x_k) = x^* + \frac{g''(c_k)}{2}(x_k - x^*)^2,$$

where c_k is between x_k and x^*. Now since $x_{k+1} = g(x_k)$ by definition, some arithmetic provides

$$\frac{x_{k+1} - x^*}{(x_k - x^*)^2} = \frac{g''(c_k)}{2}.$$

The squeeze theorem implies that $c_k \to x^*$ as $k \to \infty$ (since we already established that $x_k \to x^*$), and so taking the limit as $k \to \infty$ yields

$$\lim_{k\to\infty} \frac{x_{k+1} - x^*}{(x_k - x^*)^2} = \frac{g''(x^*)}{2}.$$

Thus, the definition of quadratic convergence has been met, and so the stated result has been proven. □

One can continue this theory and method of proof to show that if additional derivatives of g vanish at x^*, then even higher order convergence of the fixed point iteration can be achieved. You will prove this for the third order case in your homework.

Let us now consider some examples.

Example 42. *Consider the following fixed point problems that are equivalent to finding the root $x = 2$ of $x^2 - x + 2 = 0$. Classify which give fixed point iterations that are guaranteed to converge to the fixed point $x^* = 2$ (with sufficiently good initial guess x_0 and of those that converge, classify their rates of convergence.*

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

3. $g(x) = 1 + \frac{2}{x}$
4. $g(x) = \frac{x^2+2}{2x-1}$

For 1., calculate $g'(x) = 2x$, so $g'(2) = 4$. Thus, we do not expect the iteration $x_{k+1} = x_k^2 - 2$ to converge with x_0 chosen sufficiently close to 2. Calculating the first few iterations using $x_0 = 2.1$, we get for x_k and $|x_k - 2|$:

```
2.1000e+00   1.0000e-01
2.4100e+00   4.1000e-01
3.8081e+00   1.8081e+00
1.2502e+01   1.0502e+01
1.5429e+02   1.5229e+02
2.3804e+04   2.3802e+04
```

Thus, we observe it is not converging. In fact, it is blowing up.

For 2., calculate $g'(x) = \frac{1}{2\sqrt{x+2}}$. Thus, $g'(2) = \frac{1}{4} = 0.25 < 1$, so we expect the iteration $x_{k+1} = \sqrt{x_k + 2}$ to converge linearly with ratio $\frac{1}{4}$ if x_0 chosen sufficiently close to 2. Calculating the first few iterations using $x_0 = 2.1$, we get for x_k and $|x_k - 2|$:

```
2.1000e+00   1.0000e-01
2.0248e+00   2.4846e-02
2.0062e+00   6.2018e-03
2.0015e+00   1.5499e-03
2.0004e+00   3.8743e-04
2.0001e+00   9.6854e-05
```

Here, we observe that the iteration is converging, and if we consider the error at each iteration, we see that it gets cut roughly by a factor of 4 each time, which is consistent with the $\frac{1}{4}$ derivative of g at 2. This is linear convergence with $\alpha = \frac{1}{4}$.

For 3., calculate $g'(x) = -\frac{2}{x^2}$, so $g'(2) = -\frac{1}{2}$, so we expect the iteration $x_{k+1} = 1 + \frac{2}{x_k}$ to converge linearly with ratio $\frac{1}{2}$. Calculating the first few iterations using $x_0 = 2.1$, we get for x_k and $|x_k - 2|$:

```
2.1000e+00   1.0000e-01
1.9524e+00   4.7619e-02
2.0244e+00   2.4390e-02
1.9880e+00   1.2048e-02
2.0061e+00   6.0606e-03
1.9970e+00   3.0211e-03
```

Here, we observe that the iteration is converging, and if we consider the error at each iteration, we see that it gets cut roughly by a factor of 2 each time, which is consistent with the $\frac{1}{2}$ ratio.

For 4., $g'(x) = \frac{2x(2x-1)-2(x^2+2)}{(2x-1)^2} = \frac{2x^2-2x-4}{(2x-1)^2}$. *Thus* $g'(2) = 0$, *and so we expect at least quadratic convergence. Calculating the first few iterations using* $x_0 = 2.1$, *we get for* x_k *and* $|x_k - 2|$*:*

```
2.1000e+00   1.0000e-01
2.0031e+00   3.1250e-03
2.0000e+00   3.2484e-06
2.0000e+00   3.5176e-12
2.0000e+00   0.00000000
```

Here we observe quadratic convergence because the ratio between the errors get squared with each iteration. It is easy to observe, without any calculations, that the convergence is superlinear since the ratio between successive errors are getting smaller. But to check further, we can calculate 'quadratic ratios' e_k/e_{k-1}^2, *and we find the first three to be 3.12e-1, 3.33e-1, 3.33e-1, which is approximately a constant.*

Note that the quadratically convergent iteration has found the solution to 16 digits of accuracy in 4 iterations. It will take much longer for the linearly convergent algorithms to achieve this accuracy.

5.4 Newton's method

The next method we study is Newton's method. While the bisection method has the attractive property that it will converge if the method is continuous and has a sign change, it has a disadvantage in that it is not smart enough to take advantage of simple functions. For example, if we know a function is linear, then finding the root should take just one step. But we also know that if you 'zoom in' on a smooth function, it looks linear. Newton figured out how to take advantage of this idea and create a faster root finding algorithm.

Newton's idea is simple: To find the root of a function, find the tangent line of the function at your current guess for the root, then your next guess will be where the tangent line hits the x-axis. Note that if the function is linear, then the root will be found in just one step, and if the function is close to linear, then this will give a very good approximation of the root. To start the algorithm, we need an initial guess, and the better it is, the better the algorithm will work. The idea is illustrated in Figure 5.2.

If we know the previous guess x_k, Newton's idea allows us to explicitly find x_{k+1}. The tangent line to f at x_k is given by

$$y - f(x_k) = f'(x_k)(x - x_k).$$

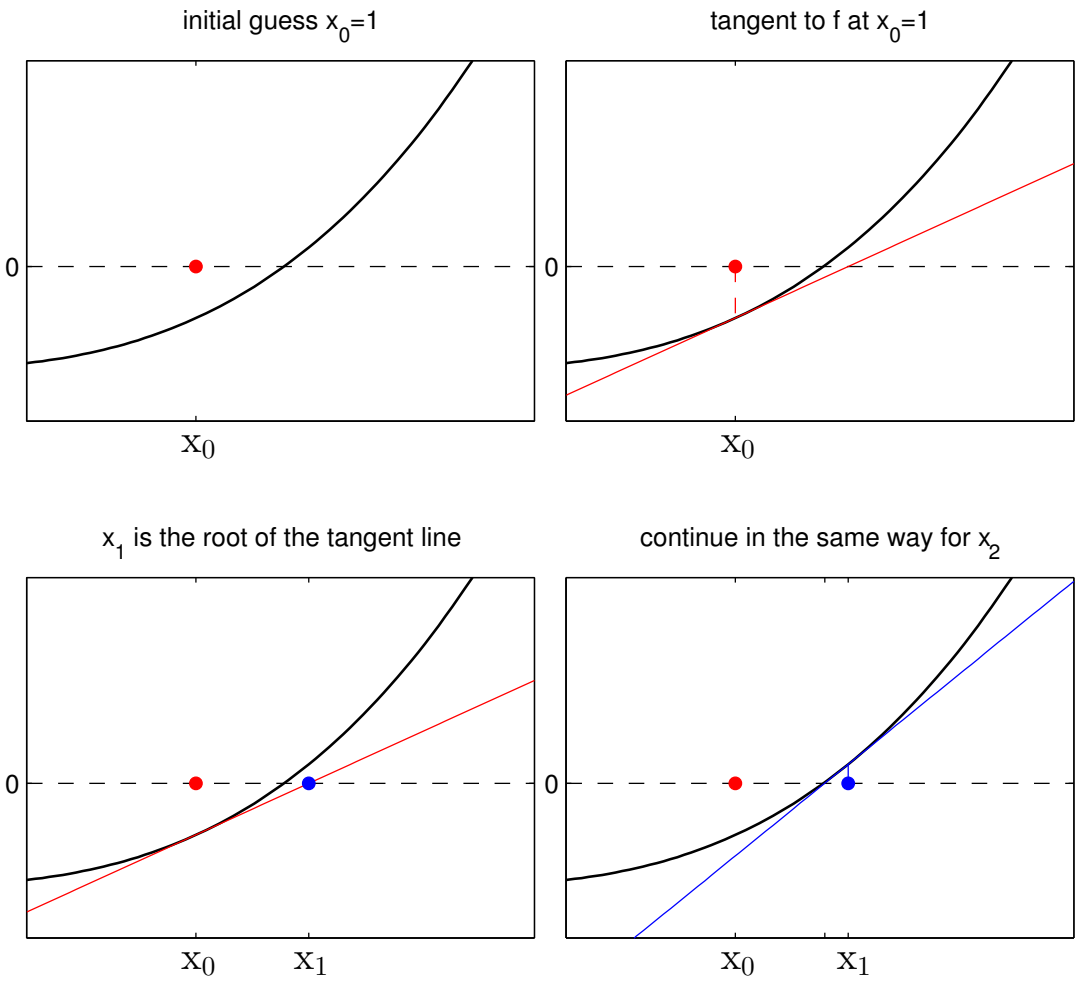

Fig. 5.2: Illustration of Newton's method.

The next Newton iteration is defined to be where this line crosses the x-axis, and we call this point $(x_{k+1}, 0)$. Plugging this point into the tangent line gives

$$0 - f(x_k) = f'(x_k)(x_{k+1} - x_k).$$

Now solve for x_{k+1}:

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

Hence given an initial guess x_0, the formula above can be used to calculate x_1, x_2, and so on, as far as you like. We would stop once $|x_{k+1} - x_k| < tol$ for some specified tolerance. Alternative stopping criteria are $|f(x_k)| < tol$ (check error) or $\frac{x_{k+1}-x_k}{x_k} < tol$ (relative tolerance). Thus we have defined another algorithm for rootfinding:

Algorithm 43 (Newton's method)**.**
Given: f, f', *tol*, x_0
while($|x_{k+1} - x_k| > tol$*):*
$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$

The code is simple to write

```
function [root,numits] = newt(fun,funderiv,x0,tol)
% solve the equation fun(x)=0 using Newton's method with starting
% value x0 up to the tolerance tol. 'funderiv' is the derivative of
% 'fun'

xold=x0+1; % initialize to something other than x0 to enter loop
xnew=x0;
numits = 0;

while ( abs( xnew-xold ) > tol )
    % perform the Newton iteration
    fxnew = fun(xnew);
    fprimexnew = funderiv(xnew);
    numits = numits+1;

    xold = xnew;
    xnew = xnew - fxnew/fprimexnew;
end

root = xnew;
end
```

Example 44. *Use Newton's method to solve* $e^x = x^2$ *using initial guess* $x_0 = 1$ *and tolerance* 10^{-14}*. We need to find the zero of* $f(x) = e^x - x^2$*, so we define* $f(x)$ *(like in the last section) and the derivative* $f'(x)$ *so we can pass them to the newt function.*

```
>> myfun1 = @(x) exp(x) - x^2;
>> myfun1prime = @(x) exp(x) - 2*x;
>> [root,numiterations]=newt(myfun1,myfun1prime,-1,1e-14)
root =
  -0.703467422498392
numiterations =
     5
```

We observe that the correct answer is found to a much smaller tolerance in 5 iterations than the bisection method found in 20! It is typical for Newton's method to converge much faster than bisection.

While the bisection method always converges if the function is continuous and the starting interval contains a root, Newton's method can run into one of several difficulties:

1. If the initial guess is not sufficiently close, the initial tangent line approximation could be terrible and prevent convergence.
2. Vanishing derivatives. If $f'(x_k) \approx 0$, the tangent line has no root, or one that is very far away. Here, Newton's method can become numerically unstable and may not converge.
3. Cycles. You can construct examples where Newton's method jumps between two points (so $x_{k+2} = x_k$) and never converges. One example is $f(x) = x^3 - 2x + 2$ with $x_1 = 0$.

In summary, Newton's method is not guaranteed to work, but when it does, it is very fast. We prove next that it converges quadratically, under some assumptions on f and a good initial condition.

Theorem 45. *Suppose $f \in C^3(B_\epsilon(x^*))$, where x^* is a root of f, and $f'(x^*) \neq 0$. Then Newton's method converges quadratically, provided the initial guess is good enough.*

Proof. The Newton iteration $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ can be considered as a fixed point iteration $x_{k+1} = g(x_k)$, where $g(x) = x - \frac{f(x)}{f'(x)}$. Note that the root x^* of f is also a fixed point of g.

For this proof, we will directly apply Theorem 41. This involves showing that $g \in C^2(B_\epsilon(x^*))$ and $g'(x^*) = 0$. We have assumed that x^* is a root of f and is, therefore, a fixed point of g, so $g(x^*) = x^*$ and $f(x^*) = 0$.

Since $f \in C^3(B_\epsilon(x^*))$, it immediately holds that $g \in C^2(B_\epsilon(x^*))$. We now calculate $g'(x)$ to be

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

Plugging in x^* yields, since $f(x^*) = 0$, $f''(x^*)$ exists, and $f'(x^*) \neq 0$, we calculate

$$\begin{aligned} g'(x^*) &= 1 - \frac{(f'(x^*))^2 - f(x^*)f''(x^*)}{(f'(x^*))^2} \\ &= 1 - \frac{(f'(x^*))^2 - 0 \cdot f''(x^*)}{(f'(x^*))^2} \\ &= 1 - \frac{(f'(x^*))^2}{(f'(x^*))^2} \\ &= 0. \end{aligned}$$

This completes the proof. □

5.5 Secant method

Although Newton's method is much faster than bisection, it has a disadvantage in that it explicitly uses the derivative of the function. In many processes where rootfinding is needed (optimization processes for example), we may not know an analytical expression for the function, and thus may not be able to find an expression for the derivative. That is, a function evaluation may be the result of a process or experiment. While we may theoretically expect the function to be differentiable, we cannot find the derivative explicitly. For situations like this, the secant method has been developed, and one can think of the secant method as being the same as Newton's method, except that instead of using the tangent line at x_k, you use the secant line at x_k and x_{k-1}. This defines the following algorithm:

Algorithm 46 (Secant method)**.**
Given: f, tol, x_0, x_1
while($|x_{k+1} - x_k| > tol$):

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

Hence we may think of the secant method as Newton's method, but with the derivative term $f'(x_k)$ replaced by the backward difference $\frac{f(x_k)-f(x_{k-1})}{x_k - x_{k-1}}$.

It is tedious (but not hard) to prove that the secant method converges superlinearly, with rate $p = \frac{1+\sqrt{5}}{2} \approx 1.618$.

5.6 Comparing bisection, Newton, Secant method

We now compare the bisection, Newton, and secant methods in the following table.

	Bisection	**Newton**	**Secant**
requirements on f?	**continuous, sign change in $[a, b]$**	**continuous, differentiable**	**continuous, differentiable**
needs	**interval $[a, b]$**	**point x_0**	**points x_0 and x_1**
always converges?	**yes**	**no, only if "close"**	**no, only if "close"**
higher dimensions?	**no**	**yes**	**yes**(1)
work per step	**1 function**	**1 function, 1 derivative**(2)	**1 function**
convergence rate(3)	**1**	**2**	**$\sim$ 1.6**

(1) The secant method can be extended to higher dimensions, but the approximation of the derivative requires more work. There are several similar methods known as "Quasi-Newton" or "Jacobian-free Newton" methods.

(2) The user needs to supply the derivative to the algorithm, which can be problematic, for example if it is difficult to compute by hand or not accessible.

(3) To achieve the given convergence rate there are more technical requirements on f (smoothness) and it only works if starting "close enough" (see above). Nevertheless, Newton's is typically faster than secant, which is in turn faster than bisection.

5.7 Combining secant and bisection and the fzero command

From the comparisons, we see that bisection is the most robust (least number of assumptions to get guaranteed convergence), but is slow. On the other hand, secant is much faster but needs good initial guesses. Hence, it seems reasonable that an algorithm that used bisection method to get close to a root, then the secant method to 'zoom in' quickly, would get the best of both worlds. Indeed, this can be done with a simple algorithm: Start with an interval that has a sign change, and then instead of using the midpoint to shrink the interval, use the secant method. But if the secant method gives a guess that is outside of the

interval, then use the midpoint instead. Eventually, the secant method will be used at each step.

Matlab/octave have a function built-in that does this (and more) for functions in one variable, and it is called 'fzero'. To use it, you simply need to give it a user-defined function and an initial guess or interval. The function intelligently alternates between bisection, secant, and a third method called inverse quadratic interpolation (use 3 points to fit a quadratic and use a root - if one exists - as the next guess).

Consider now using 'fzero' to solve $x^3 = \sin(x) + 1$. First, make a function where the solution of the equation is a root of the function, as in the following:

```
function y = myNLfun1(x)
    y=x^3 - sin(x) - 1;
```

Then simply pass this function into fzero, and give it an initial guess:

```
>> fzero(@myNLfun1,5)

ans =

   1.249052148501195
```

We can test that it worked by plugging the answer into the function.

```
>> myNLfun1(1.249052148501195)

ans =

     1.332267629550188e-15
```

5.8 Equation solving in higher dimensions

Without getting too deep into details, we simply state here that the Newton algorithm works in higher dimensions, with the same 'good' and 'bad' points as in the 1D case. The only difference in the algorithm is that instead of solving $f(x)0 = 0$ with

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

we use the analogue for vector valued functions to solve $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ using

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

In the 2x2 case,

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

and

$$\nabla \mathbf{f} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{pmatrix}.$$

As an example, consider solving the system of nonlinear equations,

$$\begin{aligned} x_1^3 &= x_2, \\ x_1 + \sin(x_2) &= -3. \end{aligned}$$

First, we create MATLAB functions for the function and its derivative (put these into separate .m files):

```
function y = myNDfun( x )
y(1,1) = x(1)^3 - x(2);
y(2,1) = x(1) + sin(x(2)) + 3;
end

function y = myNDfunprime( x )
y(1,1) = 3*x(1)^2;
y(1,2) = -1;
y(2,1) = 1;
y(2,2) = cos(x(2));
end
```

Next, create the n dimensional Newton method function:

```
function [root,numits] = newton(fun,gradfun,x0,tol)
% Solve fun(x)=0 using Newton's method
% We are given the function and its gradient gradfun
% Starting from the initial guess x0.

x0 = x0(:); % this will force x0 to be a column vector
xold = x0+1; % this needs to be ~= x0 so that we enter the while loop
xnew = x0;
numits = 0;

n = length(x0);

while norm(xnew-xold)>tol
```

```
    gradfxk = gradfun(xnew);
    fxk = fun(xnew);
    fxk = fxk(:); % this will force fxk to be a column vector

    [a,b]=size(fxk);
    if a~=n || b~=1
        error('function has wrong dimension')
    end
    [a,b]=size(gradfxk);
    if a~=n || b~=n
        error('gradient has wrong dimension')
    end

    xold = xnew;
    % x_k+1 = x_k - (grad f(xk))^{-1} * f(xk),
    % but implement as a linear solve
    xnew = xnew - gradfxk \ fxk;

    numits = numits+1;
    if (numits>=100)
        error('no convergence after 100 iterations', numits);
    end
end

root = xnew;
end
```

Running it gives the following answer:

```
>> [root,numits] = newton(@myNDfun,@myNDfunprime,[-2;-15],1e-8)
root =
  -2.474670119857577
 -15.154860516817051
numits =
     5
```

We can check that we are very close to a root:

```
>> value = myNDfun(root)
value =
   1.0e-14 *
  -0.532907051820075
  -0.088817841970013
```

If we start too far away from the root, the algorithm will not converge:

```
>> [root,numits] = newton(@myNDfun,@myNDfunprime,[100;-10],1e-8)
current step:
  -1.990536027147847
```

```
  -7.886133867955427
Error using newton (line 36)
no convergence after 100 iterations
Error in ch4_newtondimex3 (line 3)
[root,numits] = newton(@myNDfun,@myNDfunprime,[100;-10],1e-8)
```

5.9 Exercises

1. Suppose you had two algorithms for rootfinding: one is linearly convergent with a given $\alpha < 1$, and the other is superlinearly convergent with a given C and p. How close must x_k be to the root x^* for the superlinear method to outperform (reduce the error more) the linear method for every step after step k?
2. Suppose you ran a nonlinear solver algorithm and calculated errors at each step to be

   ```
   9.0579e-03
   3.4483e-03
   8.0996e-04
   9.2205e-05
   3.5416e-06
   2.6659e-08
   ```

   Classify the convergence as linear or superlinear, and determine the associated constants (α, or C and p)
3. Determine the root of $f(x) = x^2 - e^{-x}$ by hand using a) bisection method with $a = 0$ and $b = 1$ and b) Newton's method with $x_1 = 0$. Carry out three iterations each. You can round the numbers to 5 digits and use a calculator or Matlab to do each individual calculation.
4. Since the bisection method is known to cut the interval containing the root in half at each iteration, once we are given the stopping criteria "tol" (i.e. stop when $|b - a| <$tol), we immediately know how many steps of bisection need to be taken.
   Change the code "bisect.m" to use a 'for loop' instead of a 'while loop', so that the approximation to the root is within tol of a root, but the minimum number of bisection iterations are used. (you will need to calculate the number of 'for loop' iterations)
   Verify your code by solving the same equation as the previous problem, and making sure you get the same answer.
5. Consider the trisection method, which is analogous to bisection except that at each iteration, it creates 3 intervals instead of 2 and keeps one interval where there is a sign change.

(a) Is the trisection method guaranteed to converge if the initial interval has a sign change? Why or why not?

(b) Adapt the bisection code to create a code that performs the trisection method. Compare the results of bisection to trisection for "myfun1.m" for the same stopping tolerance, but several different starting intervals. How do the methods compare?

(c) The main computational cost of bisection and trisection is function evaluations. The other operations it does are a few additions and divisions, but the evaluation of most functions, including sin, cos, exp, etc. are much more costly. Based on this information, explain why you would never want to use trisection over bisection.

6. Use the bisection, Newton, and secant methods to find the smallest positive solution to $\sin(x) = \cos(x)$, using a tolerance of 10^{-12}. Note you will create your own secant code - it is a small change from the Newton code in this chapter. Report the solutions, the number of iterations each method needed, the number of function evaluations performed, and the commands you used to find the solutions. Conclude which method is fastest.
7. Prove that if x^* is a fixed point of g, $g'(x^*) = 0$, $g''(x^*) = 0$, and $g \in C^3(B_\epsilon(x^*)$, then for x_0 chosen close enough to x^*, the fixed point algorithm $x_{k+1} = g(x_k)$ converges cubically to x^*.
8. Show that Theorem 36 need not hold if the interval I is not closed.
9. Which of Newton, secant and bisection is most appropriate to find the smallest positive root of the function $f(x) = |\cos(50x)| - 1/2$, and why?
10. Use the 'fzero' command to find a solution to $\sin^2(x) = \cos^3(x)$
11. Use Newton's method with initial guess $\langle 0,\ 0,\ 0\rangle$ to find the solution of

$$\begin{aligned} x_1^2 + x_2 - 33 &= 0 \\ x_1 - x_2^2 - 4 &= 0 \\ x_1 + x_2 + x_3 - 2 &= 0 \end{aligned}$$

to 10 digits of accuracy.

12. Solve the following nonlinear system using Newton's method in Matlab:

$$\begin{aligned} x^2 &= y + \sin(z) \\ x + 20 &= y - \sin(10y) \\ (1-x)z &= 2 \end{aligned}$$

Hint: You need to find a suitable starting value for x, y, and z so that the method converges.

13. Show that for a linear function $f(x) = ax + b$, for any initial guesses x_0 and x_1, the secant method will find the root of f in one step.