This is a working set of notes for in-class discussion (APPM 4600) on algorithms proposed to solve **one non-linear equation** in one real variable, or equivalently, of methods for the **rootfinding problem**. This will cover four iterative methods (including one which is really a family of methods) to attempt to solve this family of problems: bisection, fixed point, Newton-Raphson and secant.

1 Non-linear equations and the rootfinding problem

Throughout school and even in college, we often are faced with problems of the form: "find x such that the following equation is satisfied". We are taught a number of formulas and algebraic manipulations we can use to solve for "x": this idea is so fundamental to algebra that the title of the book it comes from ("ilm al-jabr wa'l-muqabala" in arabic) translates to "the science of restoring what is missing and equating like with like".

This works fine for certain equations: quadratics, cubics, quartics, simple ones with sines, cosines, exponentials and logarithms. And yet, we quickly learn it has its limits. For example, consider solving the following problem: Find x between 3 and 4 that satisfies

$$x + \cos x = 3$$

No amount of algebraic manipulation will do, and yet, when we plot $x + \cos x$ between 3 and 4 and $y = 3$, we see a unique intersection, clear as day. In precalculus or calculus, we might have been taught to use our calculators to ask for the "intersection" of these two curves and find the solution. But how on Earth is the calculator finding it?

Before we dive into methods to try to solve this numerically (through better and better approximations), we note the following: this problem (and any other non-linear equation like it) can be related to the following: find x such that:

$$x + \cos x - 3 = 0$$

If you look at the graph intersecting $x + \cos x$ and $y = 3$, we have merely translated it 3 units down. This means any non-linear equation can be related to an equivalent problem of the form $f(x) = 0$, that is, one that asks what are the x such that $f(x)$ is zero (known as the *roots* of $f(x)$).

Definition 1 (Roots and Rootfinding Problem) *Given a function $f : \mathbb{R} \to \mathbb{R}$, we say a root $r \in \mathbb{R}$ is a value such that $f(r) = 0$. A rootfinding problem is that which asks us to find all roots (or all roots in a given interval) for a function $f(x)$.*

So from now on, we will be focusing on solving rootfinding problems for a given function $f(x)$, knowing that if you are given a non-linear equation, you can always turn it into a rootfinding problem.

2 The bisection method

The first method we will discuss to solve the rootfinding problem is one which we can come up with using the first theorem we learned in Calculus 1. For each method we come up with, we will have to make some assumptions on $f(x)$ for the method to have some guarantees / chance to work.

The thinking behind the bisection method is that for us to figure out a root for a function, one reasonable assumption to make is that $f(x)$ is *continuous* on an interval where the root or roots may be (a function which jumps all the time is very hard to predict evaluating it nearby). We can then use the Intermediate Value Theorem:

Theorem 2.1 (Intermediate Value Theorem) *Let $f(x)$ be continuous on the closed interval $[a, b]$ and $y_a = f(a) \neq y_b = f(b)$. Given any value y between y_a and y_b, there exists at least one $c \in (a, b)$ such that $f(c) = y$.*

In particular, if $f(a)f(b) < 0$, then there exists at least one root c (such that $f(c) = 0$).

This then suggests that to apply the IVT to find roots, we must make following assumptions:

- $f(x)$ is a continuous function on an interval $[a, b]$.
- $f(a)f(b) < 0$: that is, $f(x)$ changes sign on the interval.

For example: if $f(x) = x + \cos x - 3$, we can test the proposed interval and see that $f(3) = -0.9999$ and $f(4) = 0.3464$. By the IVT, there exists at least one root between 3 and 4. But... we already knew that. How can we turn this into an algorithm to figure out what this root is?

We come up with the following algorithm:

- If the interval and function $f(x)$ satisfies the assumptions, our initial guess is the midpoint $x_0 = (a + b)/2$ ($x_0 = 3.5$ in our example).
- We evaluate *the sign* of $f(x_0)$. One of three things is possible: we got lucky and x_0 is a root (we are done), $f(x_0) > 0$ or $f(x_0) < 0$.
- We then test whether the interval $[a, x_0]$ or $[x_0, b]$ satisfies the assumptions of the bisection method. *Since $f(a)$ and $f(b)$ have different signs, only one of them will.*
- We pick the interval that does satisfy the assumptions (e.g. $[a, x_0]$) and then make our next guess, x_1 to be its midpoint.
- Iterate until convergence.

We formalize this using a pseucocode format:

```
Inputs: $a, b, f(x), nmax, TOL$
Outputs: $r$
Initialize $n = 0, a_0 = a, b_0 = b, x_0 = (a + b)/2$
while Termination criteria is false do
    if $f(x_n)f(a_n) < 0$ then
        $a_{n+1} = a_n$ and $b_{n+1} = x_n$
    else if $f(x_n)f(b_n) < 0$ then
        $a_{n+1} = x_n$ and $b_{n+1} = b_n$
    else
        $r = x_n$, return $r$;
        $x_{n+1} = (a_{n+1} + b_{n+1})/2$
    end if
    $n = n + 1$
end while
Set $r = x_n$, return $r$;
```

2.1 Termination criteria: what does this mean and how to set one?

A *termination* or *stopping criterion* is a critical part of any iterative process trying to make better and better guesses x_n to a desired quantity x. For one, we do not want the iterative loop to go on forever (especially if and while we have bugs in our code). And for many reasons (efficiency and potential issues if we don't stop), we want the loop to end when we have some guarantee that $|x_n - x| < TOL_{abs}$ (absolute error target) or $|x_n - x| < TOL_{rel}|x|$ (relative error target), or both.

- To make sure the iteration always stops, we add a $nmax$ (maximum number of iterations) parameter, and ask that a loop end when $n > nmax$ (or keep going if $n \leq nmax$).
- We do *not* know x, that is why we are running this method to begin with! How might we check $|x_n - x|$ or $|x_n - x|/|x|$, then?
  - **Error estimate:** For the bisection method, we actually know that $|x - x_n| < (b_n - a_n)/2$. So we can check whether $(b_n - a_n)/2 < TOL$ and add that to our criteria to end the loop (or $(b_n - a_n)/2 \geq TOL$ for the loop to keep going).
  - **Convergence:** We can use $|x_n - x_{n-1}|$ as a way to measure whether the sequence has converged.
  - **Objective function:** We can also test whether $|f(x_n)| < TOL$, but we should be careful in that this might not guarantee that $|x - x_n| < TOL$ (depending on the derivative of $f(x)$, if it is differentiable).

So, our pseudocode now looks as follows:

Algorithm 1 Bisection method algorithm

```
    Inputs: $a, b, f(x), n_{max}, TOL$
    Outputs: $r$
 1: Initialize $n = 0, a_0 = a, b_0 = b, x_0 = (a+b)/2$
 2: while $n \leq n_{max}$ and $(b_n - a_n)/2 \geq TOL$ do
 3:     if $f(x_n)f(a_n) < 0$ then
 4:         $a_{n+1} = a_n$ and $b_{n+1} = x_n$
 5:     else if $f(x_n)f(b_n) < 0$ then
 6:         $a_{n+1} = x_n$ and $b_{n+1} = b_n$
 7:     else
 8:         $r = x_n$, return $r$;
 9:         $x_{n+1} = (a_{n+1} + b_{n+1})/2$
10:     end if
11:     $n = n + 1$
12: end while
13: Set $r = x_n$, return $r$;
```

2.2 Error analysis and Convergence

As we mentioned above, since we pick each iterate as the midpoint of the interval $[a_n, b_n]$, we have the error bound $|\varepsilon_n| = |r - x_n| < (b_n - a_n)/2$. Now, we note that every step of the algorithm, we *bisect* the previous interval (cut it in half). So,

$$|\varepsilon_n| = |r - x_n| < \frac{(b_n - a_n)}{2} = \frac{(b_{n-1} - a_{n-1})}{4} = \cdots = \frac{(b-a)}{2^{n+1}}$$

So, we can actually say more about what n is needed to *guarantee* that the algorithm will stop:

$$\begin{aligned}
\frac{(b-a)}{2^{n+1}} &< TOL \\
2^{n+1} &> \frac{(b-a)}{TOL} \\
n+1 &> \log_2\left(\frac{(b-a)}{TOL}\right) \\
n &> \log_2\left(\frac{(b-a)}{TOL}\right) - 1
\end{aligned}$$

We can compute the right-hand-side and take the ceiling to determine what n has to be to guarantee that we will satisfy the error tolerance we want. Notice this is a worst-case scenario (we might get lucky and get lower error). In particular, for our example, we know that it will take $\log_2\left(\frac{1}{2^{-52}}\right) - 1 = 51$ iterations of bisection to reach machine precision.

Convergence order and rate of an algorithm

As we saw in our experiments implementing the bisection method, one very useful way to talk about how fast it converges is to look at how the error $e_n = r - x_n$ is reduced every iteration, or equivalently, how the error after taking one additional step e_{n+1} compares to e_n. We might have assurances that $e_n \to 0$, but how fast does this happen?

We introduce two important concepts to be able to say more, and they are both based on looking at the ratios $\frac{|e_{n+1}|}{|e_n|}$. For large enough n, we want to know whether one of the following things is true:

- **Linear Convergence:** $\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|} = C$ with $C \in (0,1)$. In this case, C is known as the *rate of linear convergence.*
- **Superlinear Convergence:** $\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|} = 0$.
- **Sublinear Convergence:** $\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|} = 1$.

If the limit exists but is greater than 1, or if the lim (and lim sup) do not exist, then e_n cannot possibly go to zero, so this is not a possibility.

How to detect linear convergence, and what does it tell me about my algorithm?

If an algorithm converges linearly with rate $C \in (0,1)$, looking at the limit definition, we see that this means that for *large enough* n, the reduction in error from step n to step $n+1$ is a factor of C. In other words: $|r - x_{n+1}| \simeq C|r - x_n|$. For example: the bisection method can be shown (this is not easy) to converge linearly with a rate of (1/2) (on average). This means (all of these are *equivalent*):

- The error reduces by a factor of about a (1/2) every step. This means it reduces by a factor of (1/10) every $3-4$ steps ($log_2(10) \simeq 3.3$). *Linear convergence means it takes a fixed number of iterations to reduce the error by a given factor* (say, a factor of 10).

- Looking at the digits of r, the above means that we "gain" one correct digit of r every $3-4$ steps as we perform the bisection method. *Linear convergence means it takes a fixed number of iterations to "gain" one digit in the answer.*

- If we plot $\log |e_{n+1}|$ against $\log |e_n|$ or $\log |e_n|$ as a function of n, we see a clear linear trend, especially for large enough n.

Note that, perhaps counterintuitively, if we compare two or more algorithms that converge linearly, the *smaller* C is, the faster the algorithm goes to a certain desired accuracy, and the closer C is to 1 (*larger*), the slower the algorithm goes.

Now, some algorithms converge faster than linear; we call those *superlinearly* convergent. To make more sense of what that means, we define the following:

Definition 2 (Order of convergence) *Let x_n be a sequence of iterates that converges to r as $n \to \infty$. We say that x_n converges to r with order q if the limit*

$$\lim_{n\to\infty} \frac{|r - x_{n+1}|}{|r - x_n|^q} = \lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^q} = C \neq 0$$

If $q = 1$, C has to be in $(0,1)$ and we say x_n converges linearly with rate C. If $q > 1$, we say x_n converges superlinearly with order q. If $q < 1$, we say x_n converges sublinearly with order q.

In the following sections, we will cover algorithms that may converge superlinearly under the right conditions. We will revisit then what it means for an algorithm to converge superlinearly, and how we can tell. In summary,

- The error reduces by a factor that goes to *zero* as we take more steps. *Superlinear convergence means the number of iterations it takes to reduce the error by a given factor becomes smaller / the number of digits we gain per iteration grows.*

- Looking at the digits of r, the above means that we "gain" more and more digits in one step as n increases. We may gain 1 digit, then 2, then 4, then 8, and so on. *Superlinear convergence means the number of digits gained increases.*

- If we plot $\log |e_{n+1}|$ against $\log |e_n|$ or $\log |e_n|$ as a function of n, we see a superlinear trend (concave down, like a downwards quadratic), especially for large enough n.

We leave an exercise to the reader: write down what the three points above will look like for a sublinearly convergent algorithm.

3 Fixed Point Methods

The bisection method is a slow, but reliable method that makes very little assumptions on our function $f(x)$. However, we may want to find other methods which are potentially faster, either because the rate of linear convergence is much smaller or because they converge superlinearly.

The **Fixed Point Method**, also known as **Fixed Point Iteration**, is really not one but a family of methods that can be applied to rootfinding problems. To better understand how it works, we first have to understand what a fixed point is and how the fixed point iteration tries to find it:

Definition 3 (Fixed Point) *Let $g : \mathbb{R} \to \mathbb{R}$. We say r is a* fixed point *of g if*

$$g(r) = r$$

If you think of g as a transformation of the space $\mathbb{R}$, r is a point that stays the same. The fixed point problem *is then the problem of finding all fixed points of g (on $\mathbb{R}$ or on a given interval $[a, b]$).*

You may reasonably ask: how is this in any shape or form related to rootfinding? The answer is that given a rootfinding problem of the form: find r such that $f(r) = 0$, there are actually *many* $g(x)$ such that if $f(r) = 0$, then $g(r) = r$, and if $g(r) = r$, then $f(r) = 0$ (in other words, such that the rootfinding problem for f is equivalent to the fixed point problem for g).

So, for example, consider the problem of finding the root of $f(x) = x + \cos x - 3$. This is equivalent to asking us to find the fixed point of $g(x) = x - f(x) = -\cos x + 3$, or finding the fixed point of $g(x) = x + 10f(x) = 11x + 10\cos x - 30$ (Why?). In fact, we can make the following general statement:

- Given the rootfinding problem find r such that $f(r) = 0$, that is equivalent to the fixed point problem for $g_c(x) = x + cf(x)$ for any $c \neq 0$.
- Given the rootfinding problem find r such that $f(r) = 0$, that is equivalent to the fixed point problem for $g_c(x) = x + c(x)f(x)$ for any function $c(x) \neq 0$ (at least near the regions around the roots of $f(r) = 0$).

And it does not stop there: depending on our function $f(x)$, we may find other $g(x)$ that work. This means that if we have an algorithm to find fixed points, we can then potentially play with the various options for g to solve a given rootfinding problem faster. So, what is this algorithm we keep hinting at?

Definition 4 (Fixed Point iteration) *Given the problem of finding a fixed point of $g(x)$ and an initial guess x_0, the fixed point iteration is an iterative method taking the sequence of steps defined by:*

$$x_n = g(x_{n-1})$$

That is, we apply g over and over again: $x_2 = g(x_1) = g(g(x_0))$, $x_3 = g(x_2) = g(g(g(x_0)))$, and so on.

The general idea behind this iteration comes from both mathematical and physical intuition of what a fixed point "is" in real life systems. A fixed point in a dynamical system (one that changes in time) is an *equilibrium point*: if you are in that state, you stay at that state. However, some equilibria are *stable*, in that if you start near it, you will be attracted towards it (e.g. terminal velocity, thermal equilibria, population models). Others are *unstable* in that if you start near it (but not at it), you will be pushed away from it (e.g. balancing on a beam, being pushed off a bike).

Under certain conditions (which we will make precise below), we can say that applying g again and again will drive us closer and closer to a fixed point (the fixed point would be a stable one). If this is true, then fixed point iteration will converge to it.

3.1 What can we say about the fixed point problem, and the performance of the Fixed Point Iteration?

As is true of many mathematical problems, if we are asked to "find r such that it is a fixed point of $g(x)$ in the interval $[a, b]$, we may want to ask a few questions. Namely: what needs to be true about $g(x)$ such that this problem (1) Has a solution at all and such that (2) That solution is unique? Here, we will insert a third question: if the problem has a unique solution, when (if at all) does the fixed point iteration converge to it, and how fast does it converge?

Fixed points appear everywhere in pure and applied mathematics, and so, the theory behind them is quite rich in results. We have the following theorem, which makes some basic assumptions on g (starting, again, with assuming continuity):

Theorem 3.1 *We have the following for the fixed point problem:*

(Existence) *If $g(x)$ is continuous in $[a, b]$ and $g(x) \in [a, b]$ for all $x \in (a, b)$, then g has* at least one *fixed point $r \in [a, b]$, that is, such that $g(r) = r$.*

(Uniqueness) *If, in addition, $g'(x)$ exists in (a, b) and there exists a positive constant $k < 1$ such that $|g'(x)| \leq k$ for all $x \in [a, b]$, then there is* exactly one *fixed point in $[a, b]$ (the solution is unique).*

(Fixed Point iteration performance) *If the assumptions for uniqueness are met, then for any x_0 in (a, b), the fixed point iteration will converge* at least linearly *(it could be faster) to the unique solution r.*

We prove each of these three statements in order:

1. To show the first (existence), we will use the Intermediate Value Theorem (in class, we proved this "by picture"). Consider the functions $y = g(x)$ and $y = x$ on the interval $[a, b]$; a fixed point is the same as an intersection between these two lines. The function $f(x) = g(x) - x$ then is a continuous function which is positive if $g(x)$ is above the diagonal $y = x$, and negative if it is below (a root of f is a fixed point of g).

   If $g(a) = a$ or $g(b) = b$, then we are done (there is at least a fixed point). Otherwise, by assumption, $g(a) > a$, and so $f(a) > 0$, and also by assumption, $g(b) < b$, and so $f(b) < 0$. That means f is a continuous function that changes sign in $[a, b]$. By IVT, there exists at least one root of $f(x)$. But then there exists at least one fixed point of g.

2. To show the second (uniqueness), since it involves derivatives, we can use the Mean Value Theorem. We will proceed by contradiction: we will assume there are two *distinct* fixed points r_1 and r_2, and show this contradicts our assumptions on g (and so there cannot be more than one).

   If there were these two distinct fixed points, by MVT, we would conclude that there exists ζ between r_1 and r_2 such that

   $$g'(\zeta) = \frac{g(r_2) - g(r_1)}{r_2 - r_1} = \frac{r_2 - r_1}{r_2 - r_1} = 1$$

   But this is, of course, impossible, since $|g'(x)| \leq k < 1$ for all $x \in (a, b)$. So, the fixed point must be unique!

3. Finally, say we take $x_0 \in [a,b]$ and run the fixed point iteration. We then look at the error $|r - x_n|$, and notice that:

$$|r - x_n| = |g(r) - g(x_{n-1})|$$

by definition of fixed point and of the iteration method. Using (again) the MVT, we can conclude that there exists ζ_{n-1} between r and x_{n-1} such that $g(r) - g(x_{n-1}) = g'(\zeta_{n-1})(r - x_{n-1})$. But then,

$$|r - x_n| = |g(r) - g(x_{n-1})| \leq k|r - x_{n-1}|$$

That is, $|e_n| \leq k|e_{n-1}|$! Since $k \in (0,1)$, this already suggests to us that the algorithm converges at least linearly, with rate better or equal to k. To confirm it, we reason that $|e_n| \leq k|e_{n-1}| \leq k^2|e_{n-2}| \leq \cdots \leq k^n|e_0|$, and that goes to zero as $n \to \infty$.

Remark 3.1 (Contractive map condition) *There is a more general condition we can ask of g to ensure uniqueness and get the same performance from the Fixed Point Iteration, and that is that g is a* contractive map. *That is: we assume that there exists a positive $k < 1$ such that, for* any *x, y in $[a,b]$, it is true that $|g(x) - g(y)| \leq k|x - y|$. We leave the exercise to the reader to check that if g is contractive, we can show both of these results following the same arguments as above. This is known as the **Banach Fixed Point Theorem**.*

Finally, we write a pseudocode for the fixed point iteration. This is a very simple algorithm to write down and to code:

Algorithm 2 Fixed Point Iteration algorithm

```
    Inputs: $g(x), x_0, n_{max}, TOL$
    Outputs: $r$
1: Initialize $n = 0$
2: while ($n \leq n_{max}$ AND $|x_n - x_{n-1}| \geq TOL$ ) do
3:     Compute $x_{n+1} = g(x_n)$
4:     $n = n + 1$
5: end while
6: Set $r = x_n$, return $r$.
```

Example: Apply the fixed point iteration to

- $g_1(x) = 1 + 0.5\sin(x)$
- $g_2(x) = 3 + 2\sin(x)$

The fixed points for these are $r_1 = 1.49870113351785$ and $r_2 = 3.09438341304928$. Choose an initial guess near but not at the fixed point. Do the fixed point iterations converge? If not, do we have a good idea as to why? (Hint: compute the derivative and plot it on an interval around the fixed point).

Can we find the order and convergence rate of the FPI?

Above, we saw that the behavior of the FPI near a fixed point can be linked to whether the derivative is, in absolute value, smaller than or greater than 1 at an interval near the fixed point. However, this requires us to *check a condition for a whole interval.* Also: it tells us that the algorithm converges at least linearly, but it does not tell us the order (if it is linear, quadratic, etc).

To get more information, we now use a Taylor expansion around a fixed point r of a function $g(x)$ (assuming we have as many continuous derivatives as needed). If we try to evaluate $g(x_n)$ for x_n close to r (say, we are performing the FPI), we see that:

$$g(x_n) = g(r) + g'(r)e_n + \frac{1}{2}g''(r)e_n^2 + \frac{1}{3!}g'''(r)e_{n+1}^3 + \cdots$$

If our function is at least twice continuously differentiable, we can also say:

$$g(x_n) = g(r) + g'(r)e_n + \frac{1}{2}g''(\zeta_n)e_n^2$$

where ζ_n between x_n and r. But, by definition, $g(r) = r$. So,

$$x_{n+1} - r = e_{n+1} = g'(r)e_n + \frac{1}{2}g''(\zeta_n)e_n^2$$
$$\frac{e_{n+1}}{e_n} = g'(r) + \frac{1}{2}g''(\zeta_n)e_n$$

As we take the limit as $n \to \infty$, this tells us that $\lim_{n\to\infty} \frac{e_{n+1}}{e_n} = g'(r)$!

Recall that this limit is exactly what we used to separate between linear, superlinear and sublinear convergence. We have enough evidence to prove the following theorem:

Theorem 3.2 (Order and rate of convergence of Fixed Point Iteration) *Let $g(x)$ a function that is at least twice continuously differentiable on a neighborhood of the fixed point r. Then,*

- *If $0 < |g'(r)| < 1$, then there exists a δ such that if $x_0 \in (r-\delta, r+\delta)$, then the FPI converges linearly with rate $|g'(r)|$.*
- *If $g'(r) = 0$, then there exists a δ such that if $x_0 \in (r-\delta, r+\delta)$, then the FPI converges at least quadratically (order 2).*

We have already shown the first statement. The second one can be easily obtained by looking at the expression obtained from the Taylor residual theorem: if $g'(r) = 0$,

$$\frac{e_{n+1}}{e_n^2} = \frac{g'(r)}{e_n} + \frac{1}{2}g''(\zeta_n) = \frac{1}{2}g''(\zeta_n)$$

And the limit of this expression as $n \to \infty$ is $\frac{1}{2}g''(r)$. You can also generalize this statement to the following:

- If g is q times continuously differentiable on a neighborhood of r, and $g'(r) = g''(r) = \cdots = g^{(q-1)}(r) = 0$, then there exists a δ such that if $x_0 \in (r-\delta, r+\delta)$, then the FPI converges at least with order q.

Remark 3.2 (Basins of convergence) *The main "weakness" of the result above (which is otherwise very powerful) is that it says "there exists an interval around r such that", but it doesn't say how large or small that interval is. That means all we know is that "if our initial guess is close enough, then our algorithm will converge with this order". As we will see when we study the Newton-Raphson method, this means outside of this interval, we have little to no guarantees.*

Examples

4 The Newton-Raphson Method

A question that could arise as we understand the performance of Fixed Point Iteration is: If I want to find the roots of $f(x)$, could I find a $g(x)$ such that the Fixed Point Iteration converges at least quadratically on a neighborhood of r?

Another question we could ask given the methods we have proposed so far, is that to compute the iterates (guesses) for bisection and fixed point, we have only used function evaluations. What if we had the derivative $f'(x)$ available as well? Could we use it to come up with a better method?

Let's go back to our Calculus I roots: let's say I have an initial guess x_0 for the root. What does knowing $f'(x_0)$ give me in terms of understanding / approximating the function $f(x)$ better? The very definition of the derivative (as the slope of the line tangent to the graph) says that the best linear approximation of $f(x)$ around $x = x_0$ is:

$$\ell_0(x) = f(x_0) + f'(x_0)(x - x_0)$$

If this line is not horizontal (the slope $f'(x_0) \neq 0$), it has a unique intersection with the x axis (in other words, a unique root). We set x_1 as the root of $\ell_0(x)$:

$$\begin{aligned} \ell(x_1) &= f(x_0) + f'(x_0)(x_1 - x_0) = 0 \\ x_1 - x_0 &= -\frac{f(x_0)}{f'(x_0)} \\ x_1 &= x_0 - \frac{f(x_0)}{f'(x_0)} \end{aligned}$$

We can then just iterate on this procedure: that is the Newton-Raphson method. So, the Newton step is given by:

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

Pseudocode for Newton-Raphson

Algorithm 3 Newton-Raphson algorithm

```
    Inputs: $f(x), f'(x), x_0, n_{max}, TOL$
    Outputs: $r$
1: Initialize $n = 0$
2: while ($n \leq n_{max}$ AND $|x_n - x_{n-1}| \geq TOL$ ) do
3:     Compute $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
4:     $n = n + 1$
5: end while
6: Set $r = x_n$, return $r$.
```

Notice this algorithm looks very close to the one for fixed point iteration. One thing we might want to add is a line that checks if $|f'(x_0)|$ (or even $|f'(x_n)|$ during the iteration) is dangerously close to zero, as that might mean the Newton algorithm will likely fail (the line is almost horizontal, so x_n is very far from x_{n-1}).

4.1 Convergence analysis

Note that if we define $g_{NR}(x) = x - \frac{1}{f'(x)}f(x)$, the Newton-Raphson method is simply $x_n = g_{NR}(x_{n-1})$. This corresponds to choosing $g_c(x) = x - c(x)f(x)$ with $c(x) = -\frac{1}{f'(x)}$! This means we will be able to use all of our results from Fixed Point Iteration and apply them to analyze Newton.

We have the following theorem:

Theorem 4.1 (Convergence of the Newton-Raphson method) *Let f at least twice continuously differentiable in $[a, b]$, and r a root of f in $[a, b]$, such that $f'(r) \neq 0$. Then, there exists a $\delta > 0$ such that if $x_0 \in (r - \delta, r + \delta)$, the Newton method generates a sequence x_n that converges to r at least quadratically.*

There are two well-known ways to prove this theorem; we will outline them both.

Newton as FPI: First, as promised, we interpret Newton-Raphson as a Fixed Point method. We know the order of convergence depends on the derivative of $g_{NR}(x)$ at $x = r$. Let's compute it, given the assumption that f is twice continuously differentiable:

$$\begin{aligned} g'_{NR}(x) &= 1 - \frac{f'(x)^2 - f(x)f''(x)}{f'(x)^2} = \frac{f(x)f''(x)}{f'(x)^2} \\ g'_{NR}(r) &= \frac{f(r)f''(r)}{f'(r)^2} = 0 \end{aligned}$$

So, by the theorem we showed on convergence of FPI, Newton converges at least quadratically for x_0 close enough to r.

The other proof (using Taylor): Let $e_n = x_n - r$. By applying the Taylor residual theorem to a Taylor expansion centered at $x = x_n$, we get the following:

$$0 = f(r) = f(x_n - e_n) = f(x_n) - e_n f'(x_n) + \frac{e_n^2}{2} f''(\zeta_n)$$

Since the derivative is continuous and $f'(r) \neq 0$, we can take n large enough that $f'(x_n) \neq 0$ as well. Dividing the expression above by $f'(x_n)$ we get:

$$\begin{aligned} 0 &= \frac{f(x_n)}{f'(x_n)} - x_n + r + \frac{f''(\zeta_n)}{2f'(x_n)} e_n^2 \\ \left(x_n - \frac{f(x_n)}{f'(x_n)}\right) - r &= \frac{f''(\zeta_n)}{2f'(x_n)} e_n^2 \\ e_{n+1} &= \frac{f''(\zeta_n)}{2f'(x_n)} e_n^2 \end{aligned}$$

4.2 What happens if $f'(r) = 0$?

If $f'(r) = 0$, the Newton method will still converge for initial guesses in a neighborhood of r, but this convergence will be linear. This is, in a sense, a failure mode since the whole reason we are using Newton is to get quadratic convergence near the root. We can use the FPI analysis to show this and even to come up with a couple options to ”fix” this issue.

To understand what is going on, the following definition will be very useful:

Definition 5 (Root multiplicity) *Say f is an at least n continuously differentiable function such that $f(r) = 0$ and $f'(r) = \cdots = f^{(n-1)}(r) = 0$, with $f^{(n)}(r) \neq 0$. That is, f and its derivatives up to $n-1$ are zero at the root. We then say that r is a root of multiplicity n, and there exists $h(x)$ such that $h(r) \neq 0$ and*

$$f(x) = (x-r)^n h(x)$$

This definition should remind us of what a root of multiplicity n is in the case of polynomials; it turns out that we can extend it to general functions. Since we will be using $f'(x)$, we compute it assuming r is a root of multiplicity $n \geq 1$:

$$f'(x) = n(x-r)^{n-1}h(x) + (x-r)^n h'(x) = (x-r)^{n-1}\left(nh(x) + (x-r)h'(x)\right)$$

This leads to hairy formulas for g'_{NR}, but they simplify a ton once we plug in $x = r$. We have that:

$$\begin{aligned}
g_{NR}(x) &= x - \frac{(x-r)^n h(x)}{(x-r)^{n-1}\left(nh(x) + (x-r)h'(x)\right)} = x - \frac{(x-r)h(x)}{(nh(x) + (x-r)h'(x))} \\
g'_{NR}(x) &= 1 - \frac{(h(x) + (x-r)h'(x))\,(nh(x) + (x-r)h'(x)) - (x-r)h(x)\,((n+1)h'(x) + (x-r)h''(x))}{(nh(x) + (x-r)h'(x))^2} \\
g'_{NR}(r) &= 1 - \frac{nh(r)^2}{n^2 h(r)^2} = 1 - \frac{1}{n}
\end{aligned}$$

If $n > 1$, this is a number between 0 and 1 (and it gets closer to 1 as n increases). This tells us that the Newton Raphson method will converge linearly with rate $1 - \frac{1}{n}$ if the root has multiplicity $n > 1$ (is double, triple, etc). An exercise we can leave the reader is to perform fixed point analysis on the following modifications of the Newton Raphson method which are potential fixes:

1. If the multiplicity n of the root is somehow known ahead of time, we can try the following modified Newton step:

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

   This effectively means picking a slightly different $\widetilde{g_{NR}}$. Use the same analysis as above to show $\widetilde{g'_{NR}}(r) = 0$.

2. If $f(x)$ is at least three times continuously differentiable and we have easy access to first and second derivatives, we can apply the Newton method to $\widetilde{f}(x) = \frac{f(x)}{f'(x)}$. This new function has a simple root (multiplicity 1) at r.

4.3 Examples of Success and Failure with Newton, and what Quadratic convergence looks like

Given what we know about the Newton method, it should be no surprise that we cannot in general guarantee it will converge from a chosen x_0. Even if it does, the iteration might slowly converge until it enters the basin of quadratic convergence. And, as illustrated above, it only exhibits linear convergence for repeated (multiple) roots, and needs to be "fixed" to recover quadratic convergence.

Here are some representative examples of what it looks like when Newton converges quadratically, and when it does not. If we want to have some guarantees, *we cannot use Newton alone.* We either need to combine it with another more robust method like bisection (producing a hybrid method) OR we need some way to "guide" the Newton step.

5 The Secant Method (QuasiNewton)

Imagine the rootfinding problem we need to solve in our application is such that computing $f'(x)$ accurately is either very expensive or not possible at all. However, we still may want to produce a method that "imitates" Newton, keeping its best feature: that close enough to the root, it converges superlinearly. So, we want a method that:

- Does not ask us to compute derivatives, or at worst requires computing $f'(x_0)$, but not more.
- Is superlinearly convergent to r for x_0 close to r.

First idea: Lazy Newton (chord iteration) If we can compute or even approximate $f'(x_0)$, we can then modify the Newton step to not change the derivative value in the step formula. That is:

$$x_n = x_{n-1} - \frac{1}{f'(x_0)} f(x_{n-1})$$

we (somewhat jokingly) call this the "Lazy Newton method", since it does not bother to "update" the derivative value; it is more formally known as the Chord Iteration. Using FPI analysis, we can show that if it converges at all, this method converges linearly to r, and so we have lost the desired superlinear convergence.

Pseudocode for Chord Iteration

Algorithm 4 Chord Iteration (Lazy Newton) algorithm

```
    Inputs: $f(x), f'(x_0), x_0, n_{max}, TOL$
    Outputs: $r$
1: Initialize $n = 0$
2: while ($n \leq n_{max}$ AND $|x_n - x_{n-1}| \geq TOL$ ) do
3:     Compute $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_0)}$
4:     $n = n + 1$
5: end while
6: Set $r = x_n$, return $r$.
```

Second idea: Secant method A better idea is to use how we derived the Newton method in terms of finding roots for the line tangent to the curve. Let's say instead of a tangent line, we have two initial guesses x_0, x_1, and we find the line *secant* to the graph of $f(x)$ at the points $(x_0, f(x_0))$ and $(x_1, f(x_1))$. The equation is now

$$\ell_{sec}(x) = f(x_1) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1) = f(x_1) + m_{sec}(x - x_1)$$

Finding the root of this line (if it is not horizontal), we get:

$$x_2 = x_1 - \left(\frac{x_1 - x_0}{f(x_1) - f(x_0)}\right) f(x_1) = x_1 - \frac{1}{m_{sec}} f(x_1)$$

More generally,

$$x_n = x_{n-1} - \left(\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}\right) f(x_{n-1}) = x_{n-1} - \frac{1}{m_{sec}} f(x_{n-1})$$

We apply this idea and write down a pseudocode for the secant method. Note that in your actual implementation of it, you need to keep track of the most recent two iterates.

Pseudocode for Secant

Algorithm 5 Secant method algorithm

```
    Inputs: $f(x), x_0, x_1, n_{max}, TOL$
    Outputs: $r$
1: Initialize $n = 0$
2: while ($n \leq n_{max}$ AND $|x_n - x_{n-1}| \geq TOL$ ) do
3:     $m_{sec} = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$
4:     Compute $x_{n+1} = x_n - \frac{f(x_n)}{m_{sec})}$
5:     $n = n + 1$
6: end while
7: Set $r = x_n$, return $r$.
```

Note that this method is *not* a fixed point iteration, as the formula for the step depends on the previous two iterations. The secant method is the real deal: as x_n approaches r, the secant line approaches the tangent line, and the iteration resembles Newton-Raphson more and more. *Under the same assumptions required for Newton to converge quadratically, we can show that Secant converges superlinearly to the root.* However, we do lose a bit in terms of the order of convergence.

The following can be proven about secant method: Assuming that $f(x)$ is twice continuously differentiable on a neighborhood of a simple root ($f'(r) \neq 0$), there exists a $\delta > 0$ such that if $x_0 \in (r - \delta, r + \delta)$,

- x_n converges superlinearly to r.
- There exists an $M > 0$ such that for large enough n, $e_n \simeq M e_{n-1} e_{n-2}$ (this M is the same as that for Newton).
- For $q = \phi = \frac{1+\sqrt{5}}{2} \simeq 1.618$,
$$\lim_{n\to\infty} \frac{|e_{n+1}|}{|e_n|^{\phi}} = C \neq 0$$

The proof of this, which we will not include here, uses two Taylor expansions centered around the two iterates used to compute the secant method. However, one thing we can go over is what it means to say $e_n \simeq M e_{n-1} e_{n-2}$, and how we can go from there to conclude the order is $q = \phi$. Let's consider the following things:

1. If we look at a table of errors, it is best if we focus on the $\log_{10} e_n$ (the power of 10). So, if we apply logarithm, we get:
$$\log_{10} e_n \simeq \log_{10} e_{n-1} + \log_{10} e_{n-2} + \log_{10} M$$
If we write $L(n) = \log_{10} e_n$, this looks almost like Fibonacci: $L(n) \simeq L(n-1) + L(n-2) + \log_{10} M$. We see this in the number of digits gained: it has a Fibonacci pattern where you add the previous two results: $1, 1, 2, 3, 5, 8, 13, \cdots$ or $1, 3, 4, 7, 11, \cdots$

2. Intuitively, this means the error does reduce faster than linear, but slower than quadratic. If we want to find the exact order, we can write $e_n = Ce_{n-1}^q$ and then "plug in" that into the equation $e_n = Me_{n-1}e_{n-2}$. This will give us equations for C and q. We get:

$$\begin{aligned} Ce_{n-1}^q &= Me_{n-1}e_{n-2} \\ Ce_{n-1}^{q-1} &= Me_{n-2} \\ C^q e_{n-2}^{q(q-1)} &= Me_{n-2} \end{aligned}$$

   Since these need to match for all large enough n, $C = M^{1/q}$ and $q(q-1) = q^2 - q = 1$. Solving this quadratic gives us two solutions $q = \frac{1 \pm \sqrt{5}}{2}$, and only one of them is positive: $q = \phi \simeq 1.618$. If you are aware of the relationship between the Fibonacci numbers and the golden ratio, this is maybe not (entirely) surprising.

A good rule of thumb is that Secant will perform similarly to Newton, but with slightly lower order of superlinear convergence. The main advantage of the secant method is that it does not require computing derivatives, which might mean each iteration is less expensive. This will become *extremely valuable* when we extend this idea to systems of non-linear equations.

6 Summary of algorithms, assumptions and convergence analysis

As was done in class (and we strongly advise you to write summary tables of your own for each section of the class), we write a table to summarize what we know about the rootfinding algorithms we have proposed and studied in this block. This table should aid you to answer questions like: what are the advantages and disadvantages of each method? When would I use one over the other? Remember: there is no silver bullet method; there are always circumtances where a slow but reliable algorithm like bisection is preferable (or can be used in combination with) a fast but less reliable method like Newton.

Method	Assumptions	Cost per Iteration	Convergence
Bisection	$f \in C[a,b]$, $f(a)f(b) < 0$	$f(x_n)$ eval	Linear (rate 0.5)
Fixed Point	$\lvert g'(x)\rvert \leq k < 1$ near r (Contractive) x_0 near r	$g(x_k)$ eval	Linear (at least)
Newton	$f \in C^2((r-\delta, r+\delta))$ x_0 near r, r simple	$f(x_n), f'(x_n)$ eval	Quadratic!
Lazy Newton	Newton	$f(x_n)$ eval	Linear
Secant	Newton	$f(x_n)$ eval	Superlinear (ϕ)