1 Newton’s Method

Newton’s method is one of the most fundamental numerical algorithms. When working with smoothly varying data, it is as important as, if not more important than, the ubiquitous binary search. In fact, binary search and Newton’s method are both used to solve the same problem. But Newton’s method takes advantage of the knowledge of derivatives to solve the problem even faster (in fact, exponentially faster!) than binary search.

Like binary search, Newton’s method has a great many applications, some of them rather surprising. We will discuss a couple of examples at the end of this section.

To keep this discussion manageable, we will not discuss the full variety of circumstances in which Newton’s method can be applied. Instead, we will focus on the problem of computing square roots. This is equivalent to finding roots of the function

\[ f_1(x) := x^2 - a \]

since \( f_1(x) = 0 \) means \( x^2 - a = 0 \iff x^2 = a \).

1.1 Idea

Recall the basic idea of Newton’s method: approximate \( f \) by the straight line through the point \((x_0, f(x_0))\) with slope \( f'(x_0) \).

\[ f(x - x_0) \approx f(x_0) + f'(x_0)(x - x_0). \]

In other words, we approximate \( f \) by the first two terms of its Taylor series about the point \( x_0 \). (Note that \( x_0, f(x_0), \) and \( f'(x_0) \) are fixed constants in this formula, only \( x \) is a variable.)
If our function is a line, then it’s easy to find the (one) root. We just set the function equal to zero and solve for $x$:

$$f(x_0) + f'(x_0)(x - x_0) = 0$$

$$\Leftrightarrow f'(x_0)(x - x_0) = -f(x_0)$$

$$\Leftrightarrow x - x_0 = -\frac{f(x_0)}{f'(x_0)}$$

$$\Leftrightarrow x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

If our function $f$ were just a line, then this would be its root. Usually, our function is not a line, though. The idea of Newton’s method is to approximate $f$ by a line through $(x_0, f(x_0))$ and use the root of that linear function as our next guess $x_1$. Then we repeat the same idea for $x_1$, generating a new guess $x_2$, and so on. If we are close to the root, then there is very little error in approximating $f$ by a straight-line, and Newton’s method will quickly converge to the root.

**Example 1** For computing square roots, we have $f_1(x_i) = x_i^2 - a$ and $f'_1(x_i) = 2x_i$, so we get the Newton iteration:

$$x_{i+1} := x_i - \frac{x_i^2 - a}{2x_i} = x_i - \frac{1}{2}x_i + \frac{1}{2} \frac{a}{x_i} = \frac{1}{2} \left(x_i + \frac{a}{x_i}\right).$$

Each Newton iteration computes the next estimate $x_{i+1}$ from $x_i$ using one addition, one easy division by 2 (a shift), and one division by $x_i$. Unfortunately, since we don’t already have a fast algorithm for division (like we do for multiplication), we have to invent one.

**Example 2** In lecture, we created an algorithm for division by again using Newton’s method. Specifically, we used Newton’s method to compute the reciprocal $b^{-1}$. This is a root of the function $f_2(x) = \frac{1}{x} - b$. Since the derivative of this function is $f'_2(x) = -\frac{1}{x^2}$, we get the Newton iteration

$$x_{i+1} := x_i - \frac{x_i - b}{-\frac{1}{x_i^2}} = x_i + x_i^2 \left(\frac{1}{x_i} - b\right) = 2x_i - bx_i^2.$$

Even though we defined a function $f$ that we can’t compute (because it requires computing $x^{-1}$), we see that the Newton iteration only requires multiplication and subtraction, both operations that we know how to perform.

Thus, we have, in principle, a method for computing square roots. We do this by applying Newton’s method to the function $f_1$ above. Each Newton iteration of that, however, requires a division. We can compute $a/x_i$ as $ax_i^{-1}$, and we can compute the reciprocal $x_i^{-1}$ by again using Newton’s method, this time for the function $f_2$. 

1.2 Error Analysis

This all sounds great in principle, but how do we know that Newton’s method really is converging to the answer. Here, we will skip the general theory and focus specifically on computing square roots.

Suppose that our current estimate $x_i$ is off from the correct answer $\sqrt{a}$ by a factor of $1 + \epsilon_i$ for some $\epsilon_i$. (This $\epsilon_i$ is the “relative error” of our current estimate.) We can see how much error there will be in the next estimate simply from the definition:

$$x_{i+1} = \frac{1}{2} \left( \sqrt{a}(1 + \epsilon_i) + \frac{a}{\sqrt{a}(1 + \epsilon_i)} \right)$$

$$= \sqrt{a} \frac{1}{2} \left( (1 + \epsilon_i) + \frac{1}{1 + \epsilon_i} \right)$$

$$= \sqrt{a} \frac{1}{2} \left( (1 + \epsilon_i)^2 + \frac{1}{1 + \epsilon_i} \right)$$

$$= \sqrt{a} \frac{1}{2} \left( \frac{1 + \epsilon_i}{1 + \epsilon_i} + \frac{1 + \epsilon_i}{1 + \epsilon_i} \right)$$

$$= \sqrt{a} \frac{1}{2} \left( 2 + 2\epsilon_i + \epsilon_i^2 \right)$$

$$= \sqrt{a} \left( \frac{1}{2} \frac{1 + \epsilon_i}{1 + \epsilon_i} + \frac{1}{2} \frac{\epsilon_i^2}{1 + \epsilon_i} \right)$$

$$= \sqrt{a} \left( 1 + \frac{\epsilon_i^2}{2(1 + \epsilon_i)} \right)$$

In other words, if estimate $x_i$ had relative error $\epsilon_i$, then $x_{i+1}$ has relative error $\frac{1}{2} \epsilon_i^2 / (1 + \epsilon_i)$.

The first question we might ask is: are we actually getting closer? In other words, do we have $\epsilon_{i+1} < \epsilon_i$? Actually, let’s be a bit bolder and test whether $\epsilon_{i+1} < \frac{1}{2} \epsilon_i$. (This is the improvement we would get from binary search.)

$$\epsilon_{i+1} < \frac{1}{2} \epsilon_i$$

$$\frac{\epsilon_i^2}{2(1 + \epsilon_i)} < \frac{1}{2} \epsilon_i$$

$$\epsilon_i^2 < \epsilon_i(1 + \epsilon_i)$$

$$0 < \epsilon_i$$

These equations are all equivalent for $\epsilon_i > 0$, and this final equation is then also true, so we deduce that we must always have $\epsilon_{i+1} < \frac{1}{2} \epsilon_i$. The case $\epsilon_i < 0$ is a little more tricky, but we get the same result. In other words, we can see that Newton’s method for computing square roots is always converging at least as fast as binary search, for any initial guess $x_0$.

---

1Assuming our initial guess was positive, $x_0 > 0$, we will have $-1 < \epsilon_i < 0$. It is easy to check that we then have $\epsilon_{i+1} > 0$. So for $\epsilon_{i+2}$, we get the same analysis as above.
Linear convergence (like binary search) is nice, but we can see directly from the formula that it converges much faster than that. In particular, as long as we have $\epsilon_i > 0$, we have $1 + \epsilon_i > 1$, which means

$$\epsilon_{i+1} = \frac{\epsilon_i^2}{2(1 + \epsilon_i)} < \frac{1}{2}\epsilon_i^2.$$ 

This is quadratic convergence.

In particular, suppose that our current estimate $x_i$ is accurate to $d$ digits in base 2. This means that $|x_i - \sqrt{a}| < \frac{1}{2^d}$, or in terms of our error variables, $\epsilon_i < \frac{1}{2^d}$. Plugging this into the above relation, we get

$$\epsilon_{i+1} < \frac{1}{2} \left(\frac{1}{2^d}\right)^2 = \frac{1}{2^{2d+1}}.$$ 

Hence, each iteration takes us from $d$ digits of precision to $2d + 1$ digits, which is (more than) doubling the precision on each iteration.

What does this mean in concrete terms? As long as our initial guess has 1 digit of precision (i.e., $\epsilon_i < \frac{1}{2}$), we will see a doubling of the number of digits of precision on each iteration. Hence, we can achieve a result with $d$ digits of precision in $\log d$ iterations. (In fact, it is sufficient to have $\epsilon_i < 1$ since we will gain one digit of precision on the first iteration.)

If our initial guess has much larger error, $\epsilon_i \gg 1$, then this analysis doesn’t tell us anything useful about the rate of convergence. All we know in this case is that we do at least as well as binary search, which has linear convergence.

In Problem Set 4, you will see that this analysis is not pessimistic. If you start with a very poor guess, then Newton’s method really does behave like binary search (i.e., converging slowly) until you get close to the answer. At that point, you get a doubling of precision at each step.

### 1.3 Correctness

So far we do not have an actual algorithm because we haven’t discussed termination. In general, there are two ways to do this:

1. Perform a fixed number of iterations and then return the final estimate.
2. Perform iterations until the estimates are changing very little, e.g., only beyond the $d$-th digit.

As with other search algorithms we have seen, one approach (1) makes it easy to check termination but not so easy to check safety, while the other approach (2) makes it easier to check safety but hard to check termination.

For method 1, we can use our error analysis to deduce the number of iterations we need to perform in order to have the desired accuracy (safety). However, as we saw above, this will depend on assuming that we have an initial guess that satisfies $0 < \epsilon_0 < 1$.

For method 2, both safety and termination require us to know that Newton’s method won’t get stuck somewhere. In general, this will hold provided that our initial guess is sufficiently good.

Thus, with either approach, getting a good initial guess is critical.
Of course, for square roots, we showed in the previous section that Newton’s method never gets stuck; it always converges to the right answer. So in fact, using method 2, we always get a correct algorithm. However, we also saw that this will not be a fast algorithm unless we have a good initial guess, so once again, the initial guess is highly important.

### 1.4 Complexity

In this subsection, we will review the complexity analysis done in lecture.

Let’s start with an analysis of the time to compute a reciprocal. One can show that this also converges quadratically, assuming a sufficiently good initial estimate $x_0$, which we will assume that we have. In that case, quadratic convergence tells us that we will need $\lg d$ iterations to compute $b^{-1}$ to $d$ digits of precision.

As noted in lecture, however, this is not the end of the story. While it is certainly true that the cost of computing the reciprocal is $O(d^\alpha \lg d)$, where each multiplication takes $\Theta(d^\alpha)$ time, this bound is not tight (i.e., not a $\Theta$-bound).

The reason for this is that we don’t need to multiply $d$-digit numbers at each step. We only need to multiply numbers as large as the current precision. Since we are already assuming any further digits are incorrect, we may as well make our lives easy by setting them to be zeros.

In more detail, suppose that, on the current iteration, we have $t$ digits of precision. If we set the remaining digits to zero when multiplying, then we want to compute $a \times b$, where $a$ and $b$ contain only $t$ digits. We can see that this is $\Theta(t^\alpha)$ time and then shifting (in linear time).

Due to quadratic convergence, on iteration $i$ of Newton’s method, we will have $2^i$ digits of precision. If we use the trick above, we can multiply just two $2^i$-digit numbers, which takes time $C(2^i)^\alpha = C2^{i\alpha}$ for some constant $C$. Thus the total time for all these multiplications is

$$C2^\alpha + C2^{2\alpha} + C2^{3\alpha} + \ldots + C2^{(\lg d)\alpha}.$$  

(The final multiply uses numbers with $d = 2^{\lg d}$ digits of precision.) This is a geometric series, so as we have seen in the the proof of the Master Theorem and elsewhere, the whole sum is dominated by the last term:

$$C2^\alpha + C2^{2\alpha} + C2^{3\alpha} + \ldots + C2^{(\lg d)\alpha} = \Theta(2^{(\lg d)\alpha}) = \Theta(d^\alpha).$$

This shows that the complexity of reciprocal is equal to that of multiply, even though we are performing $\lg d$ iterations of Newton’s method.

Next, we want to analyze the cost of computing square roots. Due to quadratic convergene, we will perform $\lg d$ iterations of Newton’s method for the function $f_1$. Looking at the formula for the iteration, we can see that the cost of computing the next guess is dominated by the cost of the reciprocal. (We perform one reciprocal, one multiply, and some adds and shifts.) But as we just saw, the cost of reciprocal is $\Theta(d^\alpha)$, just like multiply.

Furthermore, we can use the same trick as above to reduce the time when we have fewer than $d$ digits of precision. In particular, if we have $t$ digits of precision on the current iteration, then we
can write our current guess as \( r \cdot 10^e \) where \( r \) is a number with \( t \) digits. It’s reciprocal is \((r \cdot 10^e)^{-1} = r^{-1}10^{-e}\). So we can compute its reciprocal just by computing the reciprocal of the \( t \)-digit number \( r \) and then shifting by \(-e\).

As before, we will have \( 2^i \) digits of precision on iteration \( i \). As we have just argued, we can compute the reciprocal of this number to \( 2^i \) digits of precision in time \( \Theta((2^i)^\alpha) \) rather than \( \Theta(d^\alpha) \). Hence, the total running time is the same sum as above. As we argued there, the last term dominates, so the total time is \( \Theta(d^\alpha) \).

Thus, the running time for this algorithm that computes square roots to \( d \) digits of precision is the same as that of reciprocal, which we saw is the same as that of multiply. They all take the same amount of time \( \Theta(d^\alpha) \), up to constant factors.

### 1.5 Applications

To emphasize the ubiquity of Newton’s method, we will quickly mention a few applications.

#### 1.5.1 Fast Inverse Square Root

Compute graphics packages perform many operations using vectors in \( \mathbb{R}^3 \). These usually represent a direction (rather than a distance), so it is important that they be normalized. If \( v = (x, y, z) \in \mathbb{R}^3 \) is a vector that is not necessarily normalized, we can make a normalized version

\[
\frac{v}{\|v\|} = \frac{1}{\sqrt{x^2 + y^2 + z^2}} v.
\]

It is easy to compute \( x^2 + y^2 + z^2 \), but computing this square root and then taking its reciprocal are not so easy (as we just saw!). However, there is a famous algorithm for computing this using 32-bit floating point numbers.

The short version of the story is the following. When the source code for Quake III was released, people started looking through it in detail and quickly stumbled onto this code. It was remarkable for a couple of reasons: (1) it was incredibly short (and fast) and (2) no one could figure out how it worked!

At a high level, it is clear that the algorithm is performing Newton’s method. However, the way it produces it’s initial guess is surprising and complicated, using bit operations that take advantage of the details of how 32-bit floating point numbers are stored in computers.

It turns out that, in addition to being incredibly fast, this method of guessing is so accurate that only a single iteration of Newton’s method is required to exhaust the precision of a 32-bit floating point number! The source code contains two iterations, but the second one is commented out.

You can read more details about the story and the code at [Wikipedia](https://en.wikipedia.org/wiki/Fast_inverse_square_root).
1.5.2 Solving Polynomial Equations

Consider a system of equations

\[
\begin{align*}
    p_1(x_1, \ldots, x_n) &= 0 \\
    p_2(x_1, \ldots, x_n) &= 0 \\
    &\vdots \\
    p_t(x_1, \ldots, x_n) &= 0
\end{align*}
\]

where each \( p_i \) is a polynomial in the variables \( x_1, \ldots, x_n \). These are very hard to solve, in general. (Indeed, they cannot be solved exactly in most cases. But even finding approximate solutions is quite complicated.)

Some of the best algorithms for solving this operate as follows. They perform a sequence of changes to coefficients of the polynomials, slowly changing them into different polynomials. Eventually, these polynomials have a special form (they become so-called binomial equations) that makes them easy to solve using a simple algorithm.

Unfortunately, these are no longer the solutions for the polynomials we want, but rather some other polynomials. To get the solution of the original equations, these algorithms use Newton’s method.

The idea is that each change that was made to the coefficients of the polynomials was individually small enough that it did not change the roots of the polynomial by very much. In particular, the change was small enough that we can use the roots of one set of polynomials as a good guess of the roots of the other. Then we apply a few iterations of Newton’s method to get back to the desired precision.

This process is repeated for each change made to the coefficients. We slowly walk back from the solution of the special polynomials, undoing the changes one at a time, until we arrive at solutions of the original polynomial equations.

2 Problem Set 4

In this section, we discuss a couple of additional issues related to the Problem Set 4, whose coding portion requires computing cube roots to arbitrary precision.

2.1 Floating-Point Representation

The presentation of computing square roots given in lecture was carefully crafted to always work with integers. For example, rather than computing \( \sqrt{2} \) to \( d \) digits of precision after the decimal point, we instead computed \( \lfloor \sqrt{2} \times 10^{2d} \rfloor \). This is an integer, and we know immediately that it equals \( \lfloor \sqrt{2} \times 10^d \rfloor \), so the fractional answer is just a shifted version of this integer.

Similarly, when we computed reciprocals, instead of computing \( b^{-1} \), we computed the ratio \( 10^k / b \). For sufficiently large \( k \), this will give us \( d \) digits of precision and will be an integer.
It would be nicer, however, if we did not have to deal with such issues. In other words, it would be simpler for us, in writing numerical algorithms, to assume that we have not only arbitrary-precision integers, but arbitrary-precision floating-point numbers.

This can be done. In fact, the idea is basically just to generalize the tricks above. Whenever you have a floating point number with $d$ digits, we turn it into an integer by multiplying by $10^d$.

In particular, suppose that our $d$-digit number is $x_0.x_1x_2x_3\ldots x_{d-1} \times 10^e$ for some exponent $e$. (Here, we choose $e$ so that the decimal point falls just after the first digit.) If we multiply the number $x_0.x_1x_2\ldots x_{d-1}$ by $10^{d-1}$, then it becomes an integer. Thus, we can equivalently write our number as $y \times 10^{e-d+1}$, where $y$ is an integer in the range $10^{d-1} \leq d < 10^d$.

The above representation, using $y$ and $e$, is precisely what is stored in the Decimal class used in Problem Set 4. You will use these floating-point numbers when calculating cube roots to arbitrary precision.

### 2.2 Realistic Error Analysis

When you implement the problem set, you will discover that the error analysis done above is unrealistic. This is because, when we computed $\epsilon_{i+1}$ in terms of $\epsilon_i$, we assumed that all of the operations are performed exactly. This does not hold in practice because each result is rounded to the nearest number with $d$-digits of precision.

For the most part, this does not affect the analysis we did. The errors introduced by these operations are typically much smaller than the error in the current estimate, so we still see rapid, quadratic convergence.

However, once the error gets close to that of the numbers themselves (that is, close to $10^{-d}$), then the errors in the operations become relevant in the above calculations.

This has the following practical result: Newton’s method may not every converge exactly! Instead, it may simply “stabilize” at some point, where each iteration differs from the previous in only the few digits. (Typically, this will be only the last digit, but that depends on the Newton iteration. If we are multiplying by large numbers on each iteration, then small errors in the operations get magnified.)

The most important practical impact of this for your coding on Problem Set 4 is that you cannot just terminate when the estimates stop changing. They may continue changing forever. Instead, you must stop when the estimates are only changing by a sufficiently small amount.