Math 459 · Numerical Analysis

Keye Martin

Concord University
Department of Mathematics and Computer Science
keye@concord.edu

Abstract

Numerical Analysis is frequently taught as a dry subject of pragmatic significance with little or no connection to some of the most important ideas in modern mathematics. In these notes, we will remedy this defect.

1 The basic fields

The essence of this subject is pretty simple: we approximate real numbers using sequences of rational numbers that are produced iteratively by a device capable of computation.

- The set of *natural numbers* is $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$. A *positive integer* is an $n \in \mathbb{N}$ with $n > 0$.
- The set of *integers* is $\mathbb{Z} = \{-n : n \in \mathbb{N}\} \cup \{n : n \in \mathbb{N}\}$
- The set of *rational numbers* is $\mathbb{Q} = \{p/q : p \in \mathbb{Z} \ \& \ q \in \mathbb{Z} \setminus \{0\}\}$
- The set of all *real numbers* is $\mathbb{R}$

One integer *divides* another when the former is a factor of the latter.

Exercise 1.1 *Fun with number theory.* The traditional formulation of *Euclid's Lemma* says that any prime number p that divides the product ab of two positive integers a and b must then divide either a or b. It has the following useful generalization:

- If a, b and c are positive integers for which c divides ab and $\gcd(b, c) = 1$, then c divides a. Integers whose greatest common divisor is 1 are sometimes called *relatively prime* or *coprime*.

Intuitively, here is why this generalization is true: Since c divides ab, we know that

$$\frac{ab}{c} = \left(\frac{a}{c}\right) b = k \in \mathbb{Z}.$$

As b and c have no factors in common at all, the fraction a/c must be an integer i.e. c divides a. Can you give a formal proof of this?

Exercise 1.2 *The density of the rationals.* For any two real numbers x and y with $x < y$, there is a rational q with $x < q < y$. Thus, in principle, we should be able to approximate real numbers with fractions as accurately as we like – which is what Numerical Analysis is basically about.

2 The need for numerical analysis

A *polynomial* over the reals is a function $p : \mathbb{R} \to \mathbb{R}$ of the form

$$p(x) = a_n x^n + \cdots + a_1 x + a_0$$

where the constants a_i are real numbers called *coefficients*. A problem of major importance to humanity is to find a *real* solution x to the equation $p(x) = 0$, where we mention in passing that a *complex* solution always exists by the fundamental theorem of algebra. We now argue that one must resort to computational processes when solving equations – we do not use computational devices simply for convenience, mathematics has proven that algorithms are a *necessity*.

2.1 Rational solutions

In elementary algebra, we learn how to factor polynomials as a means of solving equations. The dream of people everywhere would be that factoring always worked. This of course is not true. For instance, the equation $x^2 = x + 1$ has no rational solutions. Here's why:

Theorem 2.1 *Any rational solution p/q of a polynomial equation*

$$f(x) = a_n x^n + \cdots + a_1 x + a_0 = 0$$

with integral coefficients a_i must have the property that p divides a_0 and that q divides a_n.

Proof. We can assume that the fraction p/q is reduced to simplest form by replacing p with $p/\gcd(p, q)$ and q with $q/\gcd(p, q)$ and then renaming everything to p and q again. (By the way, that is how a computer "reduces fractions"). Then $\gcd(p, q) = 1$ i.e. p and q are coprime.

Since $f(p/q) = 0$, we can multiply by q^n to get

$$a^n p^n + a^{n-1} p^{n-1} q + \cdots + a_1 p q^{n-1} + a_0 q^n = 0$$

But this means

$$p(a_n p^{n-1} + \cdots + a_1 q^{n-1}) = -a_0 q^n$$

which says that p divides $a_0 q^n$. But it can't divide q^n because it is coprime to q, so it divides a_0. Similarly,

$$q(a^{n-1} p^{n-1} + \cdots + a_1 p q^{n-2} + a_0 q^{n-1}) = -a_n p^n$$

which tells us that q divides a_n. $\square$

Exercise 2.2 In the proof of Theorem 2.1, why can't p divide q^n?

Theorem 2.1 tells us that if we wanted to write a program to determine whether or not a general polynomial could be factored, we could do this by factoring a_n and a_0 into a product of prime powers – the latter is believed but not proven to be very difficult and the security of many modern cryptographic systems rests solely on this assumption.

Okay, so factoring doesn't always work. But maybe something else does. For the equation $x^2 = x + 1$ and more generally any *quadratic equation*

$$ax^2 + bx + c = 0$$

we know there is a formula that expresses its solutions as

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Is there a series of formulae that can give us the solution to any polynomial equation?

Exercise 2.3 One of the solutions of $x^2 = x + 1$ is one of the most famous numbers in all of mathematics. What is its name and why is it so important?

Exercise 2.4 *Some irrational numbers are more equal than others.* A real number is *algebraic* when it is the solution of a nonzero polynomial equation with integral coefficients; otherwise, it is called *transcendental*.

- Prove that the numbers π and e are both transcendental. (I'm kidding, just wanted you to be aware of this amazing fact.)
- By Exercise 2.3, then, only certain irrational numbers can arise as the solution of a nontrivial polynomial equation with integral coefficients.

The title of this exercise is an indirect reference to a line in a book by a famous author. Who?

2.2 Solvability by radicals

As luck would have it, when it comes to solving a general *cubic equation*

$$ax^3 + bx^2 + cx + d = 0$$

there actually *is* a formula known as *Cardano's solution*. The basic idea is that one first makes a linear change of variable which puts the equation into the form

$$x^3 + px + q = 0.$$

The formula then depends on how p and q relate to each other. For instance, if $4p^3 + 27q^2 > 0$, then we can say that one of its real solutions is

$$\phi = \left(-\frac{q}{2} + \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}\right)^{1/3} + \left(-\frac{q}{2} - \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}\right)^{1/3}$$

To find all solutions, we next recall that the number of real roots of a cubic can only be 1 or 3. In the case of the former, the solution is ϕ. In the case of the latter, we divide by $x - \phi$ to obtain a quadratic we can then solve for the other two. So next we can ask about the *quartic equation*

$$ax^4 + bx^3 + cx^2 + dx + e = 0$$

Is there a formula for this one too? Believe it or not, there actually is, it's referred to as *Ferrari's solution*. The existence of formulae for quadratic, cubic and quartic equations makes the following one of the most amazing results in the history of mathematics:

Theorem 2.5 *There is no formula that expresses the solutions of the polynomial equation*

$$a^n x^n + \cdots + a_1 x + a_0 = 0$$

when $n \geq 5$. That is, the solutions in general cannot be computed in a finite number of steps using only addition, subtraction, multiplication, division, n^{th} roots and the coefficients in the equation.

A proof of this result would take us too far from the topic at hand. But those interested should read about a subject in modern algebra known as the *Galois group*.

By Theorem 2.5, then, one must resort to the use of an algorithm for *approximating* the solutions of a polynomial equation and therefore expects the same to be true of solving nonlinear equations in general (though they should at least try to find an exact solution since they are always prettier).

2.3 A closer look at the quintic

It is important to understand what Theorem 2.5 says and what it does not say. It says that for $n \geq 5$, there is no formula that solves the general problem – however, this does not mean that there aren't times when formulae exist in certain restricted cases. For instance, there is a formula that expresses the solution to $x^5 - a = 0$ given by $x = a^{1/5}$. In fact, just when you thought the story couldn't get more interesting, it some how manages to do exactly that.

Theorem 2.6 *Let p be a prime number such that $p - 3$ is divisible by 4. Then the polynomial*

$$x^5 + 2px + 2p^2$$

cannot be written as the product of two nonconstant polynomials and the equation

$$x^5 + 2px + 2p^2 = 0$$

is not solvable by radicals.

A proof of this result can be found in [3]. While the general quintic is not solvable by radicals, the quintic in Theorem 2.6 is probably one of the simplest not solvable by radicals. This is amazing:

Example 2.7 Take $p = 3$ in Theorem 2.6 so that the equation

$$x^5 + 6x + 18 = 0$$

is not solvable by factoring or radicals i.e. its solutions cannot be expressed in a finite number of steps using arithmetic, n^{th} roots and the coefficients of the equation. As a polynomial of odd degree, it has at least one real solution and it can be shown that it has exactly one.

If you graph this polynomial with the best freely available graphing program on the internet `https://www.desmos.com/calculator`, you will see that this solution is approximately -1.543 and that the program thinks this is an exact solution – but it can't be because this polynomial has no rational solutions. The actual solution is *irrational* and cannot be expressed with a formula!!!

Thus, it would seem that our only option is to seek a rational approximation of this solution by means of an algorithm capable of achieving any desired accuracy. Wait, what's an algorithm?

3 The elements of computation

Algorithms are built by repeatedly applying three operations:

- Initial functions (simple predefined operations that we are given for free),
- Composition (do one thing followed by another), and
- Iteration (repeat something until a desired condition is met).

Let's start at the beginning.

3.1 The recursive functions

Definition 3.1 Let $\mathbb{N}_\perp$ denote the set $\mathbb{N} \cup \{\perp\}$, where $\perp$ is an element that does not belong to $\mathbb{N}$.

Exercise 3.2 Given *any* set X, how can we always find an element $\perp$ that does not belong to X?

Definition 3.3 A *partial function* on the naturals is a function

$$f : \mathbb{N}^n \to \mathbb{N}_\perp,$$

where $n \geq 1$. We say that f is *undefined* at x exactly when $f(x) = \perp$.

Thinking of f as an algorithm, $f(x) = \perp$ means that the program f crashed on input x, where "crashing" can also refer to not terminating i.e. "looping forever".

Definition 3.4 The *composition* of a partial map $f : \mathbb{N}^n \to \mathbb{N}_\perp$ with partial mappings $g_i : \mathbb{N}^k \to \mathbb{N}_\perp$, $1 \leq i \leq n$, is the partial map

$$f(g_1, \cdots, g_n) : \mathbb{N}^k \to \mathbb{N}_\perp$$

$$f(g_1, \cdots, g_n)(x) = \begin{cases} f(g_1(x), \cdots, g_n(x)) & \text{if } (\forall i)\ g_i(x) \neq \perp; \\ \perp & \text{otherwise.} \end{cases}$$

That is, if in the process of trying to run the program f, the computation of one of its inputs fails, then the entire computation fails.

Definition 3.5 A partial map $f : \mathbb{N}^{n+1} \to \mathbb{N}_\perp$ is defined by *primitive recursion* from $g : \mathbb{N}^n \to \mathbb{N}_\perp$ and $h : \mathbb{N}^{n+2} \to \mathbb{N}_\perp$ if

$$f(x, y) = \begin{cases} g(x) & \text{if } y = 0; \\ h(x, y-1, f(x, y-1)) & \text{otherwise.} \end{cases}$$

where we have written $x \in \mathbb{N}^n$.

Notice that the rule of composition given above insists that in order for $f(x, y)$ to be defined, f must be defined at (x, z), for all $z < y$.

Definition 3.6 The *minimization* of a partial function $f : \mathbb{N}^{n+1} \to \mathbb{N}_\perp$ is the partial function

$$\mu f : \mathbb{N}^n \to \mathbb{N}_\perp$$

$$\mu f(x) = \min\{y \in \mathbb{N} : (\forall z < y)\ f(x,z) \neq \perp\ \&\ f(x,y) = 0\}$$

with the convention that $\mu f(x) = \perp$ if no such y exists.

The intent of minimization is that the least y such that $f(x,y) = 0$ is calculable according to the natural algorithm: We begin iterating at 0, if $f(x,0) = 0$, this is the answer. If $f(x,0) = \perp$, the program has crashed, so the output is $\perp$. Otherwise, we proceed to $f(x,1)$, and so on, in the same fashion. Hence the requirement that $f(x,z) \neq \perp$, for all $z < y$: Without it, we have no assurance that y was found using the algorithm above.

Definition 3.7 The class of *partial recursive functions* on the naturals is the smallest collection of partial maps $f : \mathbb{N}^n \to \mathbb{N}_\perp$ which contains the zero function, the successor, the projections, and is closed under composition, primitive recursion, and minimization.

We will see in the next section that the partial recursive functions are exactly what current devices compute. For that reason, we sometimes refer to partial recursive functions as *computable*. Let's look at a few examples of computable functions.

Example 3.8 *The constant functions.* As initial functions, $\text{zero}(x) = 0$ and $\text{succ}(x) = x + 1$ are computable by definition. By composition,

$$\text{one}(x) = \text{succ}(\text{zero}(x)) = 1$$

is computable, and so every constant function $c_k(x) = k$ is computable since

$$c_k = \text{succ}^k(\text{zero}(x))$$

where $\text{succ}^k = \text{succ} \circ \cdots \circ \text{succ}$ is the k^{th} iterate of succ for $k \in \mathbb{N}$.

Example 3.9 *Addition is computable.* Define $f : \mathbb{N}^2 \to \mathbb{N}$ by primitive recursion from $g : \mathbb{N} \to \mathbb{N}$ and $h : \mathbb{N}^3 \to \mathbb{N}$ as

$$f(x,y) = \begin{cases} g(x) & \text{if } y = 0; \\ h(x, y-1, f(x,y-1)) & \text{otherwise.} \end{cases}$$

where $g = \pi_1 : \mathbb{N} \to \mathbb{N}$ is the computable projection given by $\pi(x) = x$ and $h = \text{succ} \circ \pi_3 : \mathbb{N}^3 \to \mathbb{N}$ is computable as the composition of succ and the projection $\pi_3 : \mathbb{N}^3 \to \mathbb{N}$. Then f is given by

$$f(x,y) = \begin{cases} x & \text{if } y = 0; \\ 1 + f(x, y-1) & \text{otherwise.} \end{cases}$$

By induction on y, we can prove that $f(x,y) = x + y$ as follows: For $y = 0$, $f(x,0) = x = x + y$; assuming $f(x, y-1) = x + y - 1$, we then have

$$f(x,y) = 1 + f(x, y-1) = 1 + (x + y - 1) = x + y.$$

Thus, addition is computable.

Exercise 3.10 Prove that multiplication $\times : \mathbb{N}^2 \to \mathbb{N}$ is computable.

As we can see from the examples of addition and multiplication, primitive recursion amounts to the ability to program with "counting loops" i.e. repeat something a *fixed* number of times. Minimization, on the other hand, is much more powerful: It represents the ability to program with "while loops" i.e. repeat something until a certain condition is met. Since all counting loops may be rewritten as while loops, primitive recursion is essentially redundant in the definition of partial recursive. Kleene [2] confirmed this fact in 1936, provided one carries a few more initial functions.

Theorem 3.11 (Kleene) *The class of partial recursive functions on the naturals is the smallest collection of partial maps $f : \mathbb{N}^n \to \mathbb{N}_\perp$ which contains addition $+$, multiplication $\times$, the projection mappings, the predicate*

$$\leq (a, b) = \begin{cases} 1 & \text{if } a \leq b; \\ 0 & \text{otherwise,} \end{cases}$$

and is closed under composition and minimization.

We should point out here that we have used a different but equivalent set of initial functions. This result allows us to connect the partial recursives with current devices in a nice way.

Exercise 3.12 Find a recursive function $A : \mathbb{N}^2 \to \mathbb{N}$ defined everywhere that is not definable by primitive recursion but is definable by minimization. (Hint: We called it 'A' for a reason.)

3.2 From logic to language

Imagine that you have just written a program in your favorite language. You then compile it, and the compiler translates it to assembly code. From there, another program translates the assembly code to machine code so it can be executed on a machine. Assembly is a language where one programs with integers using very simple instructions. The programs one can write in assembly language can all be coded in the the following language:

$$\begin{array}{rcl} \text{stmt} & ::= & \textbf{clear}\ x \\ & | & \textbf{inc}\ x \\ & | & \textbf{dec}\ x \\ & | & \textbf{while}\ x \neq 0\ \textbf{do}\ \text{stmt} \\ & | & \text{stmt}; \text{stmt}. \end{array}$$

where x can be any element taken from a countable set of variables. By Theorem 3.11, the programs one may write in this language are exactly the partial recursive functions. Thus, every algorithm is translated to an assembly language program which is the coding of a partial recursive function on the naturals. Conversely, every partial recursive function can be computed by current devices.

Exercise 3.13 How do we implement conditional statements of the form

$$\textbf{if}\ x \neq 0\ \textbf{then}\ \text{stmt}\ \textbf{else}\ \text{stmt}$$

in the language for assembly given above?

There is a deep question lurking though: Can one imagine other ways of computing where something might be "intuitively computable" but not partial recursive? As far as anyone knows, the answer is No. This is the content of the *Church–Turing Thesis*: A function on the naturals is "intuitively computable" if and only if it is partial recursive. It is not a theorem, it's a belief. Could a quantum computer violate the Church–Turing Thesis? Most researchers have argued no, but the author still considers the question open.

3.3 A simple algorithm that no one understands

For a natural number n, let

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ even;} \\ 3n+1 & \text{if } n \text{ odd.} \end{cases}$$

Beginning from an $n > 0$, suppose we continue applying f unless we reach 1.

Example 3.14 Take $n = 12$. Then $f(12) = 6$, $f(6) = 3$, $f(3) = 10$, $f(10) = 5$, $f(5) = 16$, $f(16) = 8$, $f(8) = 4$, $f(4) = 2$, $f(2) = 1$.

For $n = 12$, it took 9 applications of f before arriving at 1. For $n = 27$, it takes 111.

Question: For any positive integer n, does this sequence eventually arrive at 1?

We can write a partial recursive function that counts the number of applications of f needed to arrive at 1 as follows:

$$c(n) = \begin{cases} 0 & \text{if } n = 1; \\ 1 + c(f(n)) & \text{otherwise.} \end{cases}$$

Looked at this way, the question is asking whether this simple code for c terminates for every positive integer n, or whether there is some n for which c loops forever. There are some people who believe that this question may be one of the most important unsolved problems in mathematics. It is usually referred to as the *Collatz conjecture*.

Exercise 3.15

(1) Implement c with a 'while loop' in Python.

(2) Use (1) to explain how c can be defined by minimization, proving that c is partial recursive.

(3) The function f is primitive recursive (one definable using initial functions, composition and primitive recursion only). Is c primitive recursive? If it somehow were, this would imply that the Collatz conjecture were true, since primitive recursive functions are defined everywhere.

4 Types

Even though everything computable can be understood as a partial recursive function on the naturals, it doesn't mean that we actually want to program that way. Some level of abstraction is needed to make coding easier. For that reason, primitive recursion becomes a 'for loop', minimization becomes a 'while loop' and we classify data according to type: A *data type* is a set of values and a set of operations on those values.

Example 4.1 Standard data types in most languages.

- The *Boolean* data type has values $\{\texttt{t}, \texttt{f}\}$ and operations such as `and`, `or` and `not`. Expressions formed in the boolean data type control program flow with `if` statements and also define the test for termination in a `while` loop.
- The *Integer* data type has values $\mathbb{Z}$ and operations such as $+, -, \times, \div$ and `mod`.
- The *Character* data type has values that include any symbol that can be typed on a keyboard, such as letters or numbers.

A specific value in a data type is called a *literal*. So for instance, `t` is a Boolean literal, 1 is an integer literal and 'x' is a character literal.

4.1 Product types

Given a type t, we write $a : t$ to mean that a has type t.

Definition 4.2 The *product* of types s and t is the type with values belonging to

$$s \times t = \{\ (a, b) \mid a : s\ \&\ b : t\ \}$$

and whose operations we leave purposefully unspecified.

Notice that we can take the product of as many types as we want. For instance, if s, t, u are types, we first form a type $t \times u$ and then take its product with s to obtain $s \times (t \times u)$. In a similar way, we can form $(s \times t) \times u$. We treat these types as equal and write

$$s \times t \times u = (s \times t) \times u = s \times (t \times u).$$

Product types are sometimes referred to as *records* while in Python they are called *dictionaries*.

Exercise 4.3 Define a data type for the rationals whose operations include $+, -, \times, \div$. Make sure that operations yield rational numbers reduced in least terms.

Product types can be used to represent lists of a fixed length, but for most applications we need a more powerful way of structuring data.

4.2 Lists

In Python, the list 'data type' can be thought of as the union of all finite products over all types. To put it simply, a list can contain an object of any type at any position and can have any length. So for instance, the list

$$[1, \text{True}, [2, 3, \text{'x'}]]$$

contains three objects: the integer literal 1, the boolean literal True and a list of length three that contains two integer literals followed by a character literal 'x'. A *string* is a list of characters. A string literal like ['a','b','c'] is more succinctly written as "abc".

4.3 A data type for polynomials

One way to represent the polynomial $p(x) = a_n x^n + \cdots + a_0$ is with a list of coefficients

$$p = [a_0, \ldots, a_n]$$

If the indexing of lists begins at 0, the way it does in Python, then the degree i of the term that a coefficient a_i belongs to satisfies $p[i] = a_i$. For a polynomial like x^{100}, though, this would waste a lot of memory to store mostly zero coefficients when it isn't necessary. Instead, we can give a *sparse representation* of $p(x)$ as a list of terms, where each term is itself a value in a product type:

$$p = [(a_i, i) : a_i \neq 0]$$

In the second appendix, Python code is given which handles the reading and writing for this version of the polynomial data type. Using that code, it is super fun to implement the following operations for polynomial arithmetic.

Exercise 4.4 *Polynomial arithmetic.*

- Implement addition of polynomials with a single return statement.
- Implement multiplication of polynomials by first writing a one line function that multiplies a polynomial by a term and returns the result.
- Implement subtraction of polynomials using a single return statement.
- Implement exponentiation of polynomials to nonnegative integral powers.
- Implement composition of polynomials.

In a short time, the ability to handle polynomials will allow us to write a very cool equation solver.

Exercise 4.5 The *convolution* of vectors can be calculated as multiplication of polynomials. Implement an application of convolution after doing some research on its uses.

Exercise 4.6 *A data type for rational expressions.* A *rational expression* is a quotient of polynomials. Much like with rational numbers, we can then develop a data type for rational expressions using a product of the polynomial data type with itself.

5 The real line as a data type

As we said at the beginning, Numerical Analysis is about using algorithms to calculate rational approximations of real numbers. While the entire real line cannot be represented on a device, so that one might be uncomfortable calling it a 'data type', the parallel between its structure and its approximate representation is close enough that we will refer to it as such.

5.1 The values: approximation and convergence

Denote the positive integers by

$$\mathbb{P} = \{n \in \mathbb{N} : n > 0\}$$

A *sequence* is a function $x : \mathbb{P} \to \mathbb{R}$. The value of a sequence x at n is denoted $x_n = x(n)$ and sequences themselves are denoted (x_n). They are what algorithms generate in an effort to approximate real numbers.

Example 5.1 The sequence

$$x_n = \left(1 + \frac{1}{n}\right)^n$$

defines a rational number for each positive integer n. As n gets larger and larger, x_n gets closer and closer to the irrational number e, and is capable of getting as close to e as one wishes.

But how do we formalize the idea that a sequence can approximate a real number as close as desired? This is called *convergence*.

Definition 5.2 A sequence (x_n) *converges* to a real number x if and only if

$$(\forall \varepsilon > 0)(\exists K \in \mathbb{P})(\forall n)(n \geq K \Rightarrow |x_n - x| < \varepsilon).$$

We denote this convergence by $x_n \to x$. We call x the *limit* of (x_n) and sometimes write $\lim x_n = x$.

The convergence of certain sequences is inseparable from our intuitive grasp of the real line. A sequence (x_n) is *increasing* when $x_n \leq x_{n+1}$ for all n; it has an upper bound when $(\exists u)(\forall n)(x_n \leq u)$. The *completeness axiom* implies that every increasing sequence with an upper bound has a limit which is equal to its least upper bound.

Exercise 5.3 Show that the completeness axiom of the real line implies that every decreasing sequence with a lower bound has a limit which is equal to its greatest lower bound.

Armed with the completeness axiom, an entire universe opens. Let's take a look at a fun example.

Theorem 5.4 *Let $p \geq 0$ be a real number. Define a sequence by setting $x_1 = p$ and then use x_n to define the next term by*

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{p}{x_n}\right).$$

Then $x_n \to \sqrt{p}$.

Proof. Let $p > 0$. We show that (x_n) is a decreasing sequence with a lower bound of $\sqrt{p}$. We will then argue that its limit, guaranteed to exist by the completeness axiom, is in fact $\sqrt{p}$. First, the sequence has $\sqrt{p}$ as a lower bound: $x_1 = p \geq \sqrt{p}$ and for all other terms we have

$$x_{n+1} \geq \sqrt{p} \;\Leftrightarrow\; \frac{1}{2}\left(x_n + \frac{p}{x_n}\right) \geq \sqrt{p} \;\Leftrightarrow\; x_n^2 - 2\sqrt{p}x_n + p \geq 0 \;\Leftrightarrow\; (x_n - \sqrt{p})^2 \geq 0.$$

Second, the sequence is decreasing: for each n,

$$x_{n+1} \leq x_n \;\Leftrightarrow\; \frac{1}{2}\left(x_n + \frac{p}{x_n}\right) \leq x_n \;\Leftrightarrow\; \frac{p}{2x_n} \leq \frac{x_n}{2} \;\Leftrightarrow\; x_n \geq \sqrt{p},$$

and the inequality on the far right was just shown to be true. As a decreasing sequence with a lower bound, (x_n) has a limit. We claim that this limit is $\sqrt{p}$. To see why, notice that

$$\lim x_n = \lim x_{n+1} \implies \lim x_n = \lim \frac{1}{2}\left(x_n + \frac{p}{x_n}\right) = \frac{1}{2}\left(\lim x_n + \frac{p}{\lim x_n}\right).$$

Put another way, $x = \lim x_n$ is a nonnegative real number that satisfies

$$x = \frac{1}{2}\left(x + \frac{p}{x}\right)$$

which means that it must be $\sqrt{p}$. $\square$

When p is rational, which it always will be on a machine, every member of this sequence is also rational. So, for instance, $x_1 = 2$ and

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right)$$

defines a sequence of rational approximations that converge to the irrational number $\sqrt{2}$.

Exercise 5.5 Implement the algorithm in Theorem 5.4 using the rational data type to compute absurdly interesting fractions that provide extremely accurate approximations of $\sqrt{p}$.

Notice how the scheme used to define (x_n) is similar to the way the Collatz process in Section 3.3 is defined. In each case, a sequence is defined by iterating a certain function. Speaking of which.

5.2 The operations: continuous functions

So the real line is a data type most of whose values can only be approximated with rationals. What are its operations?

Definition 5.6 A function $f : I \to \mathbb{R}$ defined on an interval $I \subseteq \mathbb{R}$ is *continuous* if $f(x_n) \to f(x)$ whenever $x_n \to x$. We also refer to them as *continuous maps*.

The intervals one encounters in most applications are $[a, b]$, (a, b) and $(-\infty, \infty) = \mathbb{R}$. Continuous functions are used to define equations. Equations in turn are defined by operations on functions. These:

Exercise 5.7 Let f and g be continuous functions. Then $f + g, f - g, fg, f/g$ and $f \circ g$ are also continuous wherever they are defined.

Just for fun:

Example 5.8 *Polynomials are continuous.* The function $f(x) = x$ is continuous by definition. So every power $p_n(x) = x^n$ is continuous as the product of continuous functions. The constant function $a_n(x) = a_n$ is continuous. So $a_n(x)p_n(x)$ is continuous. Thus the polynomial

$$p(x) = a_n x^n + \ldots + a_0 = a_n(x)p_n(x) + \ldots + a_0(x)$$

is a sum of continuous functions and is therefore continuous.

Here is an extremely important property of continuous maps.

Theorem 5.9 *If $f : I \to \mathbb{R}$ is a continuous function with $f(a) < 0$ and $f(b) > 0$, then there is a point c between a and b with $f(c) = 0$.*

Proof. Without loss of generality, $a < b$. Define a sequence of intervals by setting $I_1 = [a, b]$ and then using I_n to define

$$I_{n+1} = \begin{cases} \text{left}(I_n) & \text{if } f(s)f(t) \leq 0 \text{ on } \text{left}(I_n) = [s, t]; \\ \text{right}(I_n) & \text{otherwise;} \end{cases}$$

where $\text{left}([x, y]) = [x, (x + y)/2]$ and $\text{right}([x, y]) = [(x + y)/2, y]$. Then (I_n) is a decreasing sequence of nonempty closed bounded intervals and therefore has nonempty intersection. (Why?) By induction, though, the length of these intervals tends to zero since

$$\text{length}(I_{n+1}) = \frac{\text{len}(I_1)}{2^n}.$$

Thus, the intersection of the I_n consists of a single point c. Now write $I_n = [a_n, b_n]$. Then

$$|a_n - c| \leq \text{len}(I_n) \ \& \ |b_n - c| \leq \text{len}(I_n)$$

and since $\text{len}(I_n) \to 0$, we have $a_n \to c$ and $b_n \to c$. By continuity, $f(a_n)f(b_n) \to f(c)f(c) = f^2(c)$. However, by the construction of the I_n, $f(a_n)f(b_n) \leq 0$, and therefore its limit is also not positive. Then $f^2(c) \leq 0$ which gives $f(c) = 0$ as desired. □

This provides an extremely valuable technique for proving that equations have solutions.

Example 5.10 *The equation $x^n = x + 1$ for $n > 1$ has a solution.* Let $p(x) = x^n - x - 1$. As a polynomial, p is continuous. Since $p(1) = -1 < 0$ and $p(2) = 2^n - 2 - 1 > 0$, Theorem 5.9 yields a point c between 1 and 2 with $p(c) = 0$. For $n = 2$, this solution is the golden ratio.

In fact, not only does Theorem 5.9 provide a valuable technique for proving that equations have solutions, but its *proof* yields an algorithm for actually solving them!

5.3 An algorithm for solving equations

If we want to solve the equation $f(x) = 0$ and have two points $a < b$ with $f(a)f(b) \leq 0$, the proof of Theorem 5.9 tells us we can do so by computing with intervals. So first, a data type for intervals.

Example 5.11 *The interval data type.* Its values are taken in a subtype of $\mathbb{Q} \times \mathbb{Q}$

$$\mathbb{I} = \{[a, b] : a, b \in \mathbb{Q} \ \& \ a \leq b\}$$

and its operations extract the left, middle and right points of an interval

$$l[a, b] = a, \quad m[a, b] = \frac{a + b}{2}, \quad r[a, b] = b$$

split intervals into their left and right halves

$$\text{left}(x) = [l(x), m(x)] \quad \& \quad \text{right}(x) = [m(x), r(x)]$$

and allow for the calculation of lengths

$$\text{len}(x) = r(x) - l(x)$$

And now, the algorithm: define s on an interval $x = [a, b]$ where f changes sign ($f(a)f(b) \leq 0$) by

$$s(x) = \begin{cases} \text{left}(x) & \text{if } f(l(x))f(m(x)) \leq 0; \\ \text{right}(x) & \text{otherwise.} \end{cases}$$

By the proof of Theorem 5.9, $s(x)$ contains a solution of $f(x) = 0$ anytime that x does, but the difference is that $\text{len}(s(x)) = \text{len}(x)/2$, so the more we apply s, the smaller the interval and so the better our approximation to the solution. This technique is called *the bisection method*.

Exercise 5.12 *The bisection method.*

(i) Use the rational data type in the appendix to implement a data type for intervals of rationals.

(ii) Implement `evalr`(p, r) which evaluates a polynomial p at a rational r and returns a rational.

(iii) Implement `changes`(p, i) which tells if a polynomial p changes sign on a rational interval i.

(iv) Implement `bisect`(p, i) which given a polynomial p that changes sign on a rational interval i returns a really small rational interval on which p changes sign.

(v) Apply the bisection method to some of the amazing polynomials studied in Section 2.3. For instance, $p(x) = x^5 + 6x + 18$. How accurate are the approximate solutions provided by your implementation?

Again, we see the same pattern with the bisection method that we saw in the last section with the square root algorithm and the Collatz process: we are iterating a function – this time one defined on the interval data type.

In closing this section, one might think that a small interval produced by the bisection method results in an accurate approximation, but it's not necessarily true. It depends on the way the curve changes.

6 Approximation of operations

With the real line as a data type, we saw in the last section that we approximate most of its *values* by convergent sequences of rationals. In a similar way, we can also approximate its *operations* with correspondingly simpler operations. Remarkably, with linear functions. A function f is *linear* when there are constants m and b such that $f(x) = mx + b$. That is, when it is a polynomial whose degree does not exceed one.

6.1 Linear approximation

If we want to approximate a continuous map f at a point p with a linear map $L_p(x) = mx + b$, then we would like $L_p(p) = f(p)$, which means that

$$L_p(x) = m(x - p) + f(p).$$

But what should the slope m of L_p be? Intuitively, the slope of f at p – but what does this mean?

Definition 6.1 If $f : I \to \mathbb{R}$ is a function and $p \in I$, we say that it has a *derivative* at p when there is an $m \in \mathbb{R}$ such that for all convergent (x_n) with $x_n \to p$, we have

$$\frac{f(x_n) - f(p)}{x_n - p} \to m.$$

The value m is called the *derivative* of f at p and is denoted $f'(p) = m$.

The *linear approximation* L_p of f at p is then given by

$$L_p(x) = f'(p)(x - p) + f(p)$$

which is sometimes also called the line tangent to f at p.

Exercise 6.2

(a) If f has a derivative at p, then f is continuous at p.

(b) The derivative of a linear map $f(x) = mx + b$ is $f'(x) = m$.

(c) For functions f and g with derivatives, we have

$$(f \pm g)'(x) = f'(x) \pm g'(x)$$
$$(fg)'(x) = f(x)g'(x) + f'(x)g(x)$$
$$\left(\frac{f}{g}\right)' = \frac{g(x)f'(x) - f(x)g'(x)}{g^2(x)}$$
$$(f \circ g)'(x) = f'(g(x))g'(x)$$

(d) Use parts (b) and (c) to show that $f'(x) = 2ax + b$ if $f(x) = ax^2 + bx + c$.

(e) By induction, $(x^n)' = nx^{n-1}$. Use this to calculate the derivative of any polynomial.

The linear approximation provided by the derivative has some amazing uses. For instance, suppose we want to calculate a solution r of the equation $f(x) = 0$ and that we have some initial approximation p of r. However close p is to r, there are many times when the linear approximation L_p of f at p has a zero that is even closer to r. But the zero of L_p is easy to calculate:

$$x = p - \frac{f(p)}{f'(p)}.$$

Put another way, if x_n is an approximation of r, then

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

may be an even better approximation of r. This iterative scheme is called *Newton's method*.

Exercise 6.3 Calculating the square root of a number $p \geq 0$ means solving the equation $x^2 - p = 0$. Use Newton's method to derive an iterative scheme for calculating $\sqrt{p}$ and compare it to the method given in Theorem 5.4. Extend this scheme to calculate $p^{1/n}$ for $n > 2$.

There are times when a derivative can be difficult or impossible to reliably calculate; in those cases, the Bisection method is preferable. In cases where the derivative can be computed, applying the bisection first to get a decent approximation and then finishing with the rapid convergence of Newton's method may be the way to go.

6.2 Optimization

We begin with a fundamental property possessed by continuous maps:

Theorem 6.4 *A continuous map $f : [a, b] \to \mathbb{R}$ assumes both an absolute maximum and an absolute minimum on $[a, b]$. That is, there are $s, t \in [a, b]$ such that $f(s) \leq f(x) \leq f(t)$ for all $x \in [a, b]$.*

That is, optimization problems have solutions if they are described by continuous maps defined on closed bounded intervals. We refer to the absolute maximum and minimum as *extrema*. Here is how we calculate them:

Theorem 6.5 *If $f : [a, b] \to \mathbb{R}$ is continuous and differentiable on (a, b), then the extrema of f occur at a, b or at some point $c \in (a, b)$ with $f'(c) = 0$.*

Proof. If the absolute maximum of f does not occur at a or b, then it occurs at some point $c \in (a, b)$ where f has a derivative. If (x_n) is a sequence with $x_n < c$ and $x_n \to c$, then

$$f'(c) = \lim \frac{f(x_n) - f(c)}{x_n - c} \geq 0$$

since the numerator and denominator are both not positive. On the other hand, if (x_n) is a sequence with $x_n > c$ and $x_n \to c$, then

$$f'(c) = \lim \frac{f(x_n) - f(c)}{x_n - c} \leq 0.$$

Thus, $f'(c) = 0$. The proof for the absolute minimum is similar. □

Thus, when we want to solve optimization problems, we may be able to do so by solving the equation $f'(x) = 0$.

Example 6.6 In estimating the performance of certain quantum technologies [1], one uses results from information theory to derive lower and upper bounds on their ability to transfer information, and then asks how large the distance between these two bounds can ever be. This requires maximizing $p : [0, 1] \to \mathbb{R}$ given by

$$p\left(x\right) = \log_2 \left(2^{\left(\frac{-s(x)}{x}\right)} + 1 \right) - \left(1 - s\left(\frac{1-x}{2} \right) \right)$$

where $s\left(x\right) = -x \log_2 \left(x\right) - \left(1 - x\right) \log_2 \left(1 - x\right)$ is the binary Shannon entropy. If one graphs p, it appears to be symmetric about the line $x = 1/2$. This would mean that its maximum were assumed at $1/2$. However, its maximum is actually assumed at a a value near $317/625 = 0.5072$, which can be found by finding the unique solution of the equation $p'(x) = 0$. This yields a maximum value of around 0.1332 while its minimum is $p(0) = p(1) = 0$.

6.3 Estimation of error

How can we tell how close an iterative scheme is to solving a problem? For instance, people trust that Python does a good job of calculating square roots, but why?

Theorem 6.7 *If $f : [a, b] \to \mathbb{R}$ is continuous and differentiable on (a, b) with $f(a) = f(b)$, then there is $c \in (a, b)$ such that $f'(c) = 0$.*

Proof. By Theorem 6.4, let s and t be the points where $s, t \in [a, b]$ for which $f(s) = \min_{x\in[a,b]} f(x)$ and $f(t) = \max_{x\in[a,b]} f(x)$. If $s \in (a, b)$, we take $c = s$ and then have $f'(c) = 0$ by Theorem 6.5. The same is true of t. Thus, neither s nor t belong to (a, b), in which case $f(a) = f(b)$ then implies that f is constant. But now any $c \in (a, b)$ will satisfy $f'(c) = 0$. □

If there is a single most important theorem in differential calculus, this is probably it:

Theorem 6.8 (The Mean Value Theorem) *If $f : [a, b] \to \mathbb{R}$ is continuous and differentiable on (a, b), then there is $c \in (a, b)$ such that $f(b) - f(a) = f'(c)(b - a)$.*

Proof. Define $g : [a, b] \to \mathbb{R}$ by

$$g(x) = f(x) + \frac{f(b) - f(a)}{b - a}(b - x).$$

This function is continuous and differentiable on (a, b) with $g(a) = g(b) = f(b)$. By Theorem 6.7, there is $c \in (a, b)$ with $g'(c) = 0$, finishing the proof. □

The accuracy of an approximation can be proven using the Mean Value Theorem.

Exercise 6.9 Show that the iterates

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{p}{x_n}\right)$$

from Theorem 5.4 satisfy

$$|x_{n+1} - \sqrt{p}| \leq \frac{1}{2^n}|p - 1|$$

for $n \geq 0$ and $p > 1$ as follows:

- Apply the mean value theorem to

  $$f(x) = \frac{1}{2}\left(x + \frac{p}{x}\right)$$

  on $[\sqrt{p}, p]$ to show

  $$|f(x) - \sqrt{p}| \leq \frac{1}{2}\left(1 - \frac{1}{p}\right)|x - \sqrt{p}|.$$

- Now use $p - \sqrt{p} < p - 1$ when $p > 1$.

One might ask why we can't just look at the difference between the programming language's built in square root function and our approximation, but the reason is that at some point we need a way of *proving* that a certain scheme is accurate to a desired number of digits – why should we trust the accuracy of a particular language's implementation to begin with?

References

[1] K. Chatzikokolakis and K. Martin. *A monotonicity principle for information theory.* Electronic Notes in Theoretical Computer Science, Elsevier Science, Volume 218, p. 111-129, 2008.

[2] S. Kleene. *A note on recursive functions.* Bulletin of the American Math Society, 1936, Volume 42, p. 544–546.

[3] Spearman and Williams. *Characterization of solvable quintics.* The American Mathematical Monthly, December 1994, Volume 101, No. 10, p. 986–992.

7 Appendix: Infinite stream of characters

```
# filename: stream.py

line = ""

def empty():

    global line

    return line == ""

def first():    # first character in an infinite stream of characters

    global line

    while empty():
       line = input().strip()

    return line[0]

def advance():   # move to the next character

    global line

    line = line[1:].strip()   # remove the first character and then white space

def get_number():

    n = ""

    while first().isdigit():
        n = n + first()
        advance()

    return int(n)
```

Appendix: Data type for polynomials with integer coefficients

```
# filename: poly.py

# many of the functions we write will work for any implementation of the polynomial data type
# assuming only that it is a list of terms; only a couple depend on the definition of "term"
#
# poly ::= term {term}  a poly is a term followed by 0 or more terms
# term ::= coef [x][^n]
# coef ::= [sign] [pos_int]
# n ::= pos_int
# sign ::= + | -
# pos_int :: digit {digit}
# digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

# import the stream of characters
from stream import empty
from stream import first
from stream import advance

# selectors for terms

def term_coef(t):

    return t["coef"]

def term_degree(t):

    return t["exp"]

def form_term(a,n):   # build the term ax^n

    return { "coef" : a, "exp": n }

# read in a term
from stream import get_number

def get_term():

    term = form_term(0,0)

    sign = 1

    if first() in ['+', '-']:
       sign = int(first()+"1")
       advance()
```

```
    if first().isdigit():
       term["coef"] = sign * get_number()
    elif first() == 'x':
       term["coef"] = sign * 1

    if first() == 'x':
       advance()
       term['exp'] = 1   # this will only stay at 1 if it is not changed by a '^'

    if first() == '^':
       advance()
       term["exp"] = get_number()

    return term

# read in a poly

def form_poly(t):    # form a poly from a term t

     if term_coef(t) == 0 and term_degree(t) == 0:
        return []
     else:
         return [t]

def constant_poly(c):   # build the constant polynomial c

    return form_poly(form_term(c,0))

def add_term(t,p): # add a term t to a polynomial p and return the result

    # a good programming practice is to avoid making assignments to parameters passed to functi
    # a function is a subprogram that you intend to return a value but not alter its input

    # make a copy of p first! copying p as a list does not work! you have to copy each term inc
    q = [x.copy() for x in p]

    try:
        i = [term_degree(s) for s in q].index(term_degree(t))
        q[i] = form_term( term_coef(q[i]) + term_coef(t), term_degree(t))
    except:
        q = q + [t]
        q.sort(key=term_degree, reverse=True)  # store terms in decreasing order of degree

    return [t for t in q if term_coef(t) != 0]
```

```
def simplify(p):  # given a list of terms we combine like terms to simplify it as much as possi

    s = constant_poly(0)

    for t in p:
        s = add_term(t, s)

    return s

def poly_character(s):
    return s in ['+','-','x'] or s.isdigit()

def get_poly():  # any non white space character not relevant to polys terminates it

    # a poly is a list of terms
    p = []

    while poly_character(first()):
        p = p + [get_term()]

    advance()   # skip the character that terminated the poly

    return simplify(p)

# routines to print terms and polys

def sign(n):   # string representing the sign of a nonzero number

    if n > 0:
       return '+'
    else:
        return '-'

def coef_string(n):   # assign a string for a positive coefficient

    if n == 1:
       return ""
    else:
        return str(n)

def term_string(t):  # make a string that looks like a term

    if term_degree(t) == 0:
       return sign(term_coef(t))+' '+str(abs(term_coef(t)))
```

```
    elif term_degree(t) == 1:
        return sign(term_coef(t)) + ' ' + coef_string(abs(term_coef(t))) + 'x'
    else:
        return sign(term_coef(t)) + ' ' + coef_string(abs(term_coef(t)))+'x^'+\
               str(term_degree(t))

def poly_string(p):  # make a string that looks like a poly

    if p == []:
       return "0"
    else:
        # the first term is handed differently than the others
        s = term_string(p[0])

        if s[0] == '+':
           s = s[1:].strip() + ' ' # remove a leading plus
        else:
            # remove the space after a leading negative
            s = '-'+s[1:].strip()+ ' '

        for t in p[1:]:                    # rest of the terms in the poly
            s = s + term_string(t) + ' '

        return s.strip()

# arithmetical operations on polys

def add_polys(p,q):

    return simplify(p+q)

def mult_term(t,p):   # multiply poly p by term t and return result

    return [form_term(term_coef(t)*term_coef(s), term_degree(t)+term_degree(s)) for s in p]

def mult_polys(p,q):

    sum = constant_poly(0)

    for t in p:
        sum = add_polys(sum,mult_term(t,q))

    return sum
```

```
def sub_polys(p,q):

    minus_one = constant_poly(-1)

    return add_polys(p, mult_polys(minus_one,q))

def exp_poly(p,n):   # exponentiate a poly p to a positive power n

    one = constant_poly(1)

    if n == 0:
       return one;   # the polynomial that is always one
    else:
        return mult_polys(p, exp_poly(p,n-1))

def compose(p,q):  # calculate the composition p(q)

    sum = constant_poly(0)

    for t in p:
        coef = constant_poly(term_coef(t))      # treat coefficient as a constant polynomial
        sum = add_polys(sum, mult_polys(coef, exp_poly(q, term_degree(t))))

    return sum
```

Appendix: The rational data type

```
# filename: rational.py

# selectors for rationals

def num(r):
    return r["num"]

def den(r):
    return r["den"]

# construct rational numbers in simplest form
from math import gcd

def sgn(n):

    if n == 0:
       return 0
    elif n > 0:
        return 1    # could just use n/abs(n) but im worried about a roundoff error
    else:
        return -1

def form_rational(p,q):
    n = gcd(abs(p),abs(q))  # store negative numbers in numerator
    return {"num":sgn(p*q)*(abs(p)//n), "den":abs(q)//n}

def rational_int(n):    # make a rational number out of an integer
    return form_rational(n,1)

# rational arithmetic
def addr(r,s):
    return form_rational( num(r)*den(s)+den(r)*num(s), den(r)*den(s))

def multr(r,s):
    return form_rational( num(r)*num(s), den(r)*den(s))

def subr(r, s):
    return addr(r, multr(rational_int(-1),s))

def reciprocal(r):
    return form_rational(den(r),num(r))

def divr(r,s):
    return multr(r,reciprocal(s))
```

```
def expr(r,n):

    if n == 0:
       return form_rational(1,1)
    else:
        return multr(r,expr(r,n-1))

def decimal(r):                      # decimal form of a rational
    return float(num(r))/float(den(r))

#input and output of rationals
from stream import first
from stream import advance
from stream import get_number

def getr():   # read in a rational
              # any non white space character not relevant to rationals terminates it
    if first() == '-':
       advance()
       p = -get_number()
    else:
        p = get_number()

    if first() == '/':
       advance()
       q = get_number()
       return form_rational(p,q)
    elif first() == '.':
       advance()
       q = get_number()
       return addr(form_rational(p,1), form_rational(q,10**len(str(q))))
    else:
        return form_rational(p,1)

def rstring(r):    # string for a rational
    if den(r) == 1:
       return str(num(r))
    elif den(r) == -1:
       return str(-num(r))   # this can't happen if all rationals are built from form_rational
    else:
       return str(num(r))+"/"+str(den(r))
```