5. Numerics

An approximate answer to the right problem is worth a good deal more than an exact answer to an approximate problem.

John Tukey

Much of early computing revolved around numerical algorithms that approximate values really sought. Ancient records show algorithms to approximate the value of $\sqrt{2}$, $\pi$ and other useful numbers. Algorithms that compute numbers like $\sqrt{2}$ or $\pi$ cannot terminate: There is no finite positional notation for these numbers. However, these algorithms can be terminated once a computed value is determined to be good enough. To illustrate numerical algorithms, let’s develop Newton’s method for computing $\sqrt{m}$.

Newton’s Method

Although Newton’s name is attached to this method, initial knowledge of it was known to ancient mathematicians of Mesopotamia, the region of modern day Iraq and Iran. Clay tablets dated from around 1800 B.C. to 1600 B.C. have been found in Mesopotamia that show how to approximate $\sqrt{2}$ and perform other arithmetic operations.

These early mathematicians considered an isosceles right triangle with legs of length 1

They knew from the Pythagorean theorem theorem that

$$1^2 + 1^2 = h^2$$

so the hypotenuse $h$ has length $h = \sqrt{2} \approx 1.41421356 \ldots$ The Babylonians knew how to approximate the value of $h = \sqrt{2}$ to many decimals places. They used a sexigesimal notation. Their calculations indicate this is what they did:

In Western society, the method presented here to compute $\sqrt{m}$ is called Newton’s method. However, it was known in many cultures prior to Newton. He did generalize the idea to a broad class of functions, not just $f(m) = x^2 - m$.

The idea is: Let $x$ be a zero of function $f$. Use Taylor’s theorem. Solve for $x$, and discard the second order error.

$$f(x) \approx f(x_k) + f'(x_k)(x - x_k) + \frac{f''(\xi)}{2}(x - x_k)^2$$

$$x = x_k - \frac{f(x_k)}{f'(x_k)} - \frac{f''(\xi)}{2f'(x_k)}(x - x_k)^2$$

$$x \approx x_k - \frac{f(x_k)}{f'(x_k)}$$

The found artifacts showing computational dexterity date from the time when the Babylonians lived in Mesopotamia.
• Start with $h_0 = 1$ as an initial approximation to $\sqrt{2}$
• Clearly 1 is too small, as the Babylonians could easily measure
• But, if $1 = \sqrt{2}$, then $1 \cdot 1 = \sqrt{2} \cdot \sqrt{2} = 2$ and so $2/1 = \sqrt{2}$
• As it is, $2/1$ is too large
• The average of the under estimate 1 and the over estimate $2/1$
  provides a better approximation to $\sqrt{2}$, call this
  \[ h_1 = \frac{1}{2} \left( h_0 + \frac{2}{h_0} \right) = \frac{1}{2} \left( 1 + \frac{2}{1} \right) = \frac{3}{2} \]
• But $h_1 = 3/2$ is too large, as the Babylonians could measure
• But, if $3/2$ were the exact square root, then $2/(3/2) = 4/3$ would equal $\sqrt{2}$
• As it is, $4/3 \approx 1.333 \cdots$ is too small
• The average of the over estimate $3/2$ and the under estimate $4/3$
  will provide a better approximation
  \[ h_2 = \frac{1}{2} \left( h_1 + \frac{2}{h_1} \right) = \frac{1}{2} \left( \frac{3}{2} + \frac{4}{3} \right) = \frac{17}{12} \approx 1.41166 \cdots \]
• But $h_2 = 17/12$ is too small
• The Babylonians carried out this iteration more times computing the $\sqrt{2}$ accurately to at least 9 decimal places
• That is, they next computed the average of $h_2 = 17/12$ and $2/h_2 = 24/17$
  \[ h_3 = \frac{1}{2} \left( h_2 + \frac{2}{h_2} \right) \]
  \[ = \frac{1}{2} \left( \frac{17}{12} + \frac{24}{17} \right) \]
  \[ = \frac{1}{2} \left( \frac{289 + 288}{17 \times 12} \right) \]
  \[ \approx 1.41421568628 \cdots \]

To generalize the $\sqrt{2}$ method, pretend you want to compute $\sqrt{m}$. This is equivalent to computing a solution to the equation
\[ x^2 - m = 0 \]
Consider the recurrence equation
\[ x_k = \frac{x_{k-1} + m/x_{k-1}}{2}, \quad k \geq 1 \quad (1) \]
Given an initial value $x_0$, equation 1 can be used to generate a sequence

$$\langle x_0, x_1, x_2, \ldots \rangle.$$ 

If you pretend that $x_k$ converges to $x$ as $k$ goes to infinity, that is,

$$\lim_{k \to \infty} x_k = x$$

Then $x$ satisfies the equations

$$x = \frac{x + m/x}{2}$$
$$2x = x + m/x$$
$$x = m/x$$
$$x^2 = m$$
$$x = \sqrt{m}$$

A first step in implementing Newton's method for computing $\sqrt{m}$ is to define the function that maps $m$ and $x_{k-1}$ to the next value $x_k$.

### Listing 10: Newton's square root recurrence

51a

$$(\text{Newton's square root recurrence 51a}) \equiv$$

```
next :: Double -> Double -> Double
next m 0 = error "Division by zero"
next m x = (x + m/x)/2
```

We want to repeatedly apply `next` to some initial value and generate a list of Doubles. Let's define `repeatedly` to be a function that applies a function $f :: \text{Double} \to \text{Double}$ to itself repeatedly. An initial value (seed) $a$ for $f$ starts the iteration, generating an infinite list.

### Listing 11: Repeatedly Apply a Function

51b

$$(\text{Repeatedly apply a function 51b}) \equiv$$

```
repeatedly :: (Double -> Double) -> Double -> [Double]
repeatedly f a = a : repeatedly f (f a)
```

Although `repeatedly` does not terminate, it can be terminated once a computed value is close enough. A common way to do this is to define a tolerance usually the machine epsilon and declare that the last computed approximation is good enough once it and the previous approximation are within the tolerance.

The absolute difference $|x_k - x_{k-1}|$ between successive iterates is a measure of closeness. One way to terminate Newton's iteration is to stop when the absolute difference is within the tolerance.

$$|x_k - x_{k-1}| \leq \tau$$
Computer arithmetic on floating point numbers is not exact. The absolute difference can be small because the numbers $x_k$ and $x_{k-1}$ themselves are small. The absolute difference may never be small because the numbers themselves are large.

Instead of computing until the difference of successive approximations approaches 0, it is often better to compute until the ratio of successive approximations approach 1. This measure of closeness is the relative difference $|x_{k-1}/x_k - 1|$. Newton’s method terminates when

$$\left|\frac{x_{k-1}}{x_k} - 1\right| \leq \tau$$

The relative function maps a tolerance $\tau$ and a sequence to the first value in the sequence where the relative error is within tolerance.

**Listing 12: Convergence of Relative Error**

52a

$$(Test\ if\ successive\ values\ meet\ a\ relative\ tolerance) \equiv$$

relative :: Double -> [Double] -> Double

relative tau (a:b:rest)

| abs (a/b-1) <= tau = b |
| otherwise = relative tau (b:rest)

Now we can express Newton’s method to compute the square root of $m$ to within a relative error tolerance $\tau$ starting with an initial guess $x_0$ as the function `mysqrt`.

**Listing 13: Newton’s Square Root Method**

52b

$$(Newton’s\ square\ root) \equiv$$

mysqrt x0 tau m = relative tau (repeatedly (next m) x0)

**Convergence of Newton’s Method**

The number of times the `(next m)` function is evaluated measures the time complexity of the `mysqrt` algorithm. It is not obvious what this number is. What can be shown is that Newton’s method converges quadratically, under certain assumptions that are often True.

Consider the function $f(x) = x^2 - m$. Using Taylor’s theorem, you can derive the equation

$$x^2 - m = (x_{k-1}^2 - m) + 2x_{k-1}(x - x_{k-1}) + (x - x_{k-1})^2$$

Pretend that $x = \sqrt{m}$ so that both sides of the above equation are zero. Divide by $2x_{k-1}$ to get

$$0 = (x_{k-1}^2 - m)/2x_{k-1} + (x - x_{k-1}) + (x - x_{k-1})^2/2x_{k-1}$$

Some define the relative error as

$$re_k = \frac{x_k - x}{x}$$

where $x$ root being sought. But this requires knowledge of $x$ to compute.
Notice that

\[
\frac{x_{k-1}^2 - m}{2x_{k-1}} - x_{k-1} = -\frac{x_{k-1}^2 + m}{2x_{k-1}} = -x_k
\]

Therefore,

\[
x_k - x = \frac{(x_{k-1} - x)^2}{2x_{k-1}} \quad \text{or} \quad e_k = \frac{e_{k-1}^2}{2x_{k-1}}
\]

That is, the absolute error \( x_k - x \) at step \( k \) is proportional to the square of the error at step \( k - 1 \). When the error is less than 1, the number of correct digits doubles with each iteration.

In the general case, assume that function \( f \) has a continuous second derivative. Assume \( x \) is a root of \( f \), that is \( f(x) = 0 \). By Taylor's theorem

\[
0 = f(x) \approx f(x_k) + f'(x_k)(x - x_k) + \frac{f''(\xi)}{2}(x - x_k)^2
\]

\[
x = x_k - \frac{f(x_k)}{f'(x_k)} - \frac{f''(\xi)}{2f'(x_k)}(x - x_k)^2
\]

\[
x \approx x_k - \frac{f(x_k)}{f'(x_k)}
\]

**Exercises**

1. The convergence described above is for the absolute error. What can you say about the rate of convergence for the relative error?

2. **Simpson's rule** is a simple quadrature algorithm. It is defined by the equation

\[
\int_{a}^{b} f(x) dx \approx \frac{b - a}{6} \left[ f(a) + 4f(m) + f(b) \right] \quad \text{where} \quad m = \left( \frac{a + b}{2} \right)
\]

   (a) Explain the idea behind Simpson’s rule.

   (b) Write a program that implements Simpson’s rule.

   (c) Over a large interval \([a, b]\) you would apply Simpson’s rule over many short intervals. Explain how this would be done.
Bibliography


Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). Introduction to Algorithms. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 135], [page 141], [page 163], [page 169], [page 178], [page 187], [page 193], [page 201]


**Index**

<table>
<thead>
<tr>
<th>Haskell programming language</th>
<th>20, 31, 44, 116, 137</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java programming language</td>
<td>31</td>
</tr>
<tr>
<td>Ackermann function</td>
<td>174</td>
</tr>
<tr>
<td>Axioms</td>
<td></td>
</tr>
<tr>
<td>Peano</td>
<td>192</td>
</tr>
<tr>
<td>Bézout, Étienne</td>
<td>47</td>
</tr>
<tr>
<td>C programming language</td>
<td>30, 31, 42, 43, 55, 109, 152, 153</td>
</tr>
<tr>
<td>Calendar</td>
<td>7</td>
</tr>
<tr>
<td>Cartesian product</td>
<td>142</td>
</tr>
<tr>
<td>Catalan numbers</td>
<td>156, 161</td>
</tr>
<tr>
<td>Computability</td>
<td>191</td>
</tr>
<tr>
<td>Computable function</td>
<td>28</td>
</tr>
<tr>
<td>Computer Language Benchmark Game</td>
<td>30</td>
</tr>
<tr>
<td>Contradiction</td>
<td>193</td>
</tr>
<tr>
<td>Dijkstra's algorithm</td>
<td>183</td>
</tr>
<tr>
<td>DNA</td>
<td>146</td>
</tr>
<tr>
<td>Empty String λ</td>
<td>61</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>44, 46, 142</td>
</tr>
<tr>
<td>Floating point</td>
<td></td>
</tr>
<tr>
<td>Machine epsilon</td>
<td>50</td>
</tr>
<tr>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>Ackermann</td>
<td>174</td>
</tr>
<tr>
<td>Functional programming</td>
<td>31</td>
</tr>
<tr>
<td>Gödel, Kurt</td>
<td>192</td>
</tr>
<tr>
<td>Golden ratio</td>
<td>142</td>
</tr>
<tr>
<td>Greatest common divisor</td>
<td>47</td>
</tr>
<tr>
<td>Gutenberg Project</td>
<td>84</td>
</tr>
<tr>
<td>Hash Tables</td>
<td>93</td>
</tr>
<tr>
<td>Hofstadter, Douglas</td>
<td>25</td>
</tr>
<tr>
<td>Kleene, Stephen</td>
<td>53</td>
</tr>
<tr>
<td>Knuth, Donald</td>
<td>30</td>
</tr>
<tr>
<td>Lamé, Gabriel</td>
<td>44</td>
</tr>
<tr>
<td>λ calculus</td>
<td>26, 28</td>
</tr>
<tr>
<td>Linear congruence</td>
<td>47</td>
</tr>
<tr>
<td>Listing</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Matrix Chain Multiplication</td>
<td>158</td>
</tr>
<tr>
<td>Listings</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>A Minimum Algorithm</td>
<td>135</td>
</tr>
<tr>
<td>Aho-Corasick</td>
<td>89</td>
</tr>
<tr>
<td>Boyer-Moore Pattern Matching</td>
<td>72</td>
</tr>
<tr>
<td>Brute-force Left-to-Right Pattern Matching</td>
<td>56</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>109</td>
</tr>
<tr>
<td>Build a Heap</td>
<td>126</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>128</td>
</tr>
</tbody>
</table>
| Direct Address Table Opera-
| tions                      | 94                    |
| Edit Distance               | 152                   |
| Find Set with Path Compress-
| sion                      | 177                   |
| Greatest Common Divisor     | 42                    |
| Hash Table with Chaining    | 95                    |
| Heap Sort                   | 127                   |
| Heapify a List              | 126                   |
| Inner Product               | 155                   |
| Insertion Sort              | 112                   |
| Knuth–Morris–Pratt Pattern Matching | 67 |
| Maximum Subsequence Sum     |                       |
| (Cubic)                     | 34                    |
| Maximum Subsequence Sum     |                       |
| (Linear)                    | 35                    |
| Merge Sort                  | 118                   |
| Morris–Pratt Pattern Matching |                   |

| Haskell Ackermann function | 174 |
| Haskell Apply a Function Repeatedly | 50 |
| Haskell Brute-force Left-to-Right Pattern Matching | 57 |
| Haskell Bubble Function | 108 |
| Haskell Bubble Sort | 109 |
| Haskell Extended Euclidean Algorithm | 48 |
| Haskell Fold from the left | 37 |
| Haskell Greatest Common Divisor | 44 |
| Haskell Inner Product | 155 |
| Haskell Insertion Sort | 112 |
| Haskell Matrix Multiplication | 155 |
| Haskell Maximum Subsequence Sum | 37 |
| Haskell Merge Sort | 117 |
| Haskell Merging Sorted Lists | 117 |
| Haskell Newton's Square Root Method | 51 |
| Haskell Newton's Square Root Recurrence | 50 |
| Haskell Quicksort | 120 |
| Haskell Randomized Partition | 137 |
| Haskell Relative Error Convergence Test | 50 |
| Haskell Sample Code Header | 31 |
| Haskell Second of a pair | 36 |
| Haskell Selection Sort | 114 |
| Haskell Sorted Decision Problem | 106 |
Splitting a List, 117
Swapping head with Another Element, 137

Pseudocode
Activity Selection, 170
Bucket Sort, 130
Dijkstra’s Algorithm, 186
Kruskal Minimum Spanning Tree, 172
Prim Minimal Spanning Tree, 181

Lucas, Édouard, 44
Memorization, 141
Metric space, 152
Newton’s method, 13
Newton, Issac, 49
Noweb, 30, 55
Object-oriented programming, 31
Pattern Matching, 53, 54
  Boyer-Moore, 71
  Brute Force, 55
Knuth–Morris–Pratt, 67
Morris–Pratt, 58
Right-to-Left Brute Force, 70
φ, 142
Presburger arithmetic, 191

RAM, 26, 29
Sequence, 19
Lucas, 44

Sets
  Cartesian product, 142
  Simpson’s rule, 51

Sorting Algorithms
  Bubble Sort, 107
  Bucket Sort, 129
  Counting Sort, 128
  Heap Sort, 124
  Insertion Sort, 111
  Merge Sort, 116
  Quick Sort, 119
  Radix Sort, 129
  Selection Sort, 114
  Shell Sort, 115

Time complexity, 127
String
  Border, 61
  Period, 61
  Prefix, 61
  Suffix, 61

Syllabus, 5

Theorems
  Bezout’s, 47
  Euclidean division, 41
  Lamé’s, 45
  Taylor’s, 49, 51

Turing
  Machine, 9, 26, 28, 192

Wikipedia, 201

xkcd
  Algorithms by Complexity, 25
  Calendar of meaningful dates, 7
  Haskell, 30
  Ineffective Sorts, 105
  Traveling Salesman Problem, 141