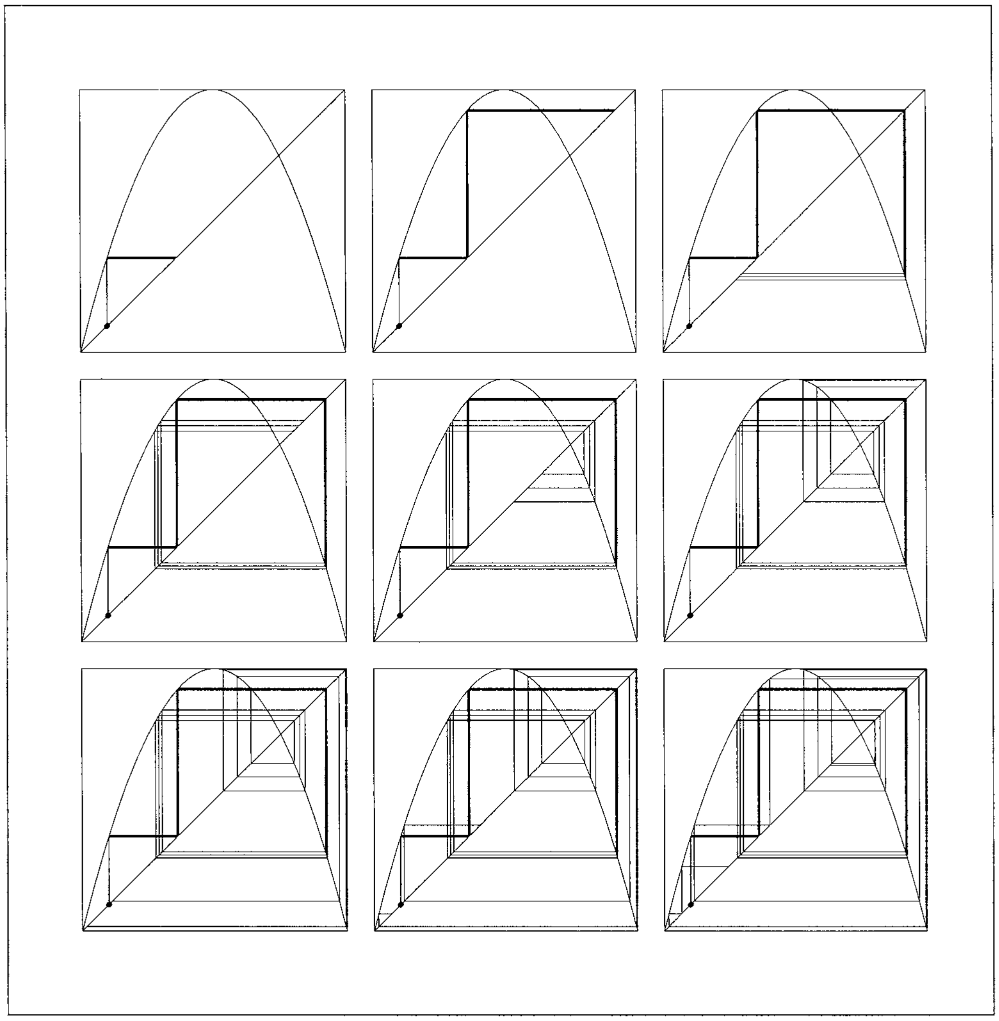
CAMBRIDGE

Cambridge University Press
0521663954 - Computer Science with Mathematica: Theory and Practice for Science, Mathematics, and Engineering
Roman E. Maeder
Excerpt
More information

Chapter 1

Computers and Science



© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521663954 - Computer Science with Mathematica: Theory and Practice for Science, Mathematics, and Engineering
Roman E. Maeder
Excerpt
More information

In the first section of this chapter we show how typical scientific problems can be solved with the help of a computer. We also discuss methods to develop programs to solve these problems. Section 1.2 covers computers and operating systems. We describe the historic development and operating principles of a modern workstation.

Programming languages have developed along with hardware. In Section 1.3, we look at a program in several forms – from machine language to a higher-level programming language. An overview over the branches of computer science (Section 1.4) concludes the chapter.

About the illustration overleaf:
The illustration shows one of the simplest functions leading to chaotic behavior. We iterated the map

$$f\colon x \mapsto 4x(1-x)\,.$$

The first nine iterations with three nearby starting values are displayed. These values separate more and more, and show quite different behavior after only eight iterations. The picture was produced with the command (see Pictures.m):

```
Animate[ FunctionIteration[4#(1-#)&, {0.099,0.1,0.101}, 0, n, {0, 1},
                                    Frame->True, FrameTicks->None],
            {n, 1, 9, 1} ];.
```

With the *Mathematica* frontend you can produce a genuine animation; here, on paper, we have to put the frames next to each other.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521663954 - Computer Science with Mathematica: Theory and Practice for Science, Mathematics, and Engineering
Roman E. Maeder
Excerpt
More information

1.1 From Problems to Programs

For computer users, the possibility of solving problems by machine is the most interesting aspect of computer science. Many textbooks and introductory classes deal exclusively with (procedural) programming, however. Programming constructs are explained with the help of simple programming exercises. Because traditional languages are not well suited to solving mathematical and scientific problems, the courses usually fail to show how such problems – which are, after all, our main interest – can be solved. The overhead stemming from the low mathematical level of even so-called higher-level programming languages shadows the underlying scientific problem and requires knowledge of memory organization, operating systems, and so on. Many of these languages were developed by computer scientists for their own use (e.g., to write compilers). In this book, we want to show that there is another way of studying both computer science and its application to the sciences and engineering.

The following subsections describe some typical uses of computers in the sciences. The examples are simpler than what you would encounter in practice, however.

We have not yet talked about how to program in *Mathematica*, so do not dwell on the syntactic details; instead, observe how easy it is to solve a problem by computer. Most of the time, the syntax will be similar to traditional mathematical notation. In the rest of this book, you will learn how to express your computations in *Mathematica*.

1.1.1 Newton's Formula

A *zero* of a function f is a value x, such that $f(x) = 0$. Newton's method for approximate determination of zeroes of functions f proceeds as follows. From a rough estimate x_0 of the zero, we can find a better estimate x_1 according to the formula

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}, \tag{1.1–1}$$

where f' denotes the derivative of f. This formula is of the form

$$x_1 = g(x_0), \tag{1.1–2}$$

with

$$g(x) = x - \frac{f(x)}{f'(x)}. \tag{1.1–3}$$

We apply the same method to x_1 to get an even better approximation $x_2 = g(x_1)$, then repeat the process. This method leads to the following iteration:

$$x_{i+1} = g(x_i), \qquad i = 0, 1, 2, \ldots. \tag{1.1–4}$$

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521663954 - Computer Science with Mathematica: Theory and Practice for Science, Mathematics, and Engineering
Roman E. Maeder
Excerpt
More information

If this sequence converges, we have found a zero of f. For an example, let us compute square roots.

The function `RootOf2` has $\sqrt{2}$ as its zero.

```
In[1]:= RootOf2[x_] := x^2 - 2
```

Here is the right-hand side of the iteration.

```
In[2]:= x - RootOf2[x]/RootOf2'[x]

                    2
              -2 + x
Out[2]= x - -------
                2 x
```

This equivalent form is often given in the literature.

```
In[3]:= (2/x + x)/2

        2
        - + x
        x
Out[3]= -----
          2
```

Let us define the iteration function g.

```
In[4]:= g[x_] = (2/x + x)/2;
```

The start value $x_0 = 1$ gives this value of x_1.

```
In[5]:= g[1.0]
Out[5]= 1.5
```

Here is x_2. The shorthand notation `%` refers to the previous result in `Out[5]` above.

```
In[6]:= g[%]
Out[6]= 1.41667
```

After seven iterations, successive values of x are already equal; that is, we have found the solution. This computation was done to 30-digit accuracy.

```
In[7]:= NestList[ g, N[1, 30], 7 ] // TableForm
Out[7]//TableForm= 1.00000000000000000000000000000
                   1.50000000000000000000000000000
                   1.41666666666666666666666666667
                   1.41421568627450980392156862745
                   1.41421356237468991062629557889
                   1.41421356237309504880168962350
                   1.41421356237309504880168872421
                   1.41421356237309504880168872421
```

For verification of the result, we square the final value. It is correct to 29 decimal places.

```
In[8]:= Last[%]^2
Out[8]= 2.00000000000000000000000000000
```

Here is another example that shows that Newton's method does not always perform this well.

The zero of this function is 0, of course.

```
In[9]:= slow[x_] := x^3

In[10]:= slow[0]
Out[10]= 0
```

Again, we define the iteration function h.

```
In[11]:= h[x_] = x - slow[x]/slow'[x]

         2 x
Out[11]= ---
          3
```

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521663954 - Computer Science with Mathematica: Theory and Practice for Science, Mathematics, and Engineering
Roman E. Maeder
Excerpt
More information

With the function h we get slow convergence, as you can see here.

```
In[12]:= NestList[ h, N[1, 30], 22 ] // TableForm
Out[12]//TableForm= 1.00000000000000000000000000000
                    0.666666666666666666666666666667
                    0.444444444444444444444444444444
                    0.296296296296296296296296296296
                    0.197530864197530864197530864198
                    0.131687242798353909465020576132
                    0.0877914951989026063100137174211
                    0.0585276634659350708733424782807
                    0.0390184423106233805822283188538
                    0.0260122948737489203881522125692
                    0.0173415299158326135921014750461
                    0.0115610199438884090614009833641
                    0.00770734662925893937426732224273
                    0.00513823108617262624951154816182
                    0.00342548739078175083300769877455
                    0.00228365826052116722200513251637
                    0.00152243884034744481467008834424
                    0.00101495922689829654311339222950
                    0.000676639484598864362075594819664
                    0.000451092989732576241383729879776
                    0.000300728659821717494255819919851
                    0.000200485773214478329503879946567
                    0.000133657182142985553002586631045
```

Even after 100 iterations, we have only 18 digits of the zero.

```
In[13]:= Nest[h, N[1, 30], 100]
                                                     -18
Out[13]= 2.45965442657982926924379399594 10
```

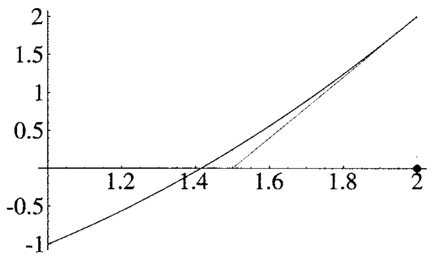
We can visualize easily the progress of Newton's method. We draw a line from the point $(x_0, 0)$ up to $(x_0, f(x_0))$, then along the tangent to the intersection with the x axis, which is the point $(x_1, 0)$, then back to $(x_1, f(x_1))$, and so on.

These steps have been collected in an extension of *Mathematica*, which we can read into our session. Doing so will define the command `NewtonIteration`.

```
In[14]:= << CSM`Iterate`
```

The start value is 2, and we perform four steps. Because of the fast convergence of the square-root iteration, we can see only the first two steps; the remaining lines are too close to the graph of the function.

```
In[15]:= NewtonIteration[ RootOf2, 2, 4, {1, 2} ];
```

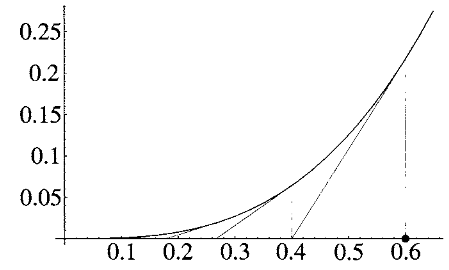


© Cambridge University Press www.cambridge.org

Cambridge University Press
0521663954 - Computer Science with Mathematica: Theory and Practice for Science, Mathematics, and Engineering
Roman E. Maeder
Excerpt
More information

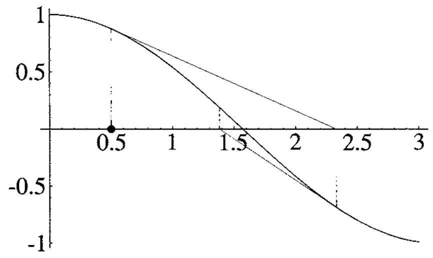
The second example shows slow convergence so we can see several more steps.

```
In[16]:= NewtonIteration[ slow, 0.6, 6, {0, 0.65},
                           PlotRange->All ];
```



The zero of the cosine at $x = \pi/2$ can also be found with this method. The values x_i alternate between being too small and too large, giving this picture.

```
In[17]:= NewtonIteration[ Cos, 0.5, 4, {0, 3} ];
```



Numerical approximation techniques were among the first algorithms developed for computers. One of the most important methods for numerical approximation is iteration. It is also used to solve systems of equations and differential equations.

1.1.2 Formulae: Uniformly Accelerated Motion

A standard topic in an introductory physics course is uniformly accelerated motion. The formulae for its special cases, such as free fall and braking distance, are easily derived from the general formula by symbolic manipulation.

The velocity at time t is $v(t) = v_0 + at$. The constant acceleration is denoted by a, and v_0 is the initial velocity.

```
In[1]:= v[t_] = v0 + a t

Out[1]= a t + v0
```

The distance traveled is the integral of the velocity

$$s(t) = \int_0^t v(\tau)d\tau$$

```
In[2]:= s[t_] = Integrate[v[tt], {tt, 0, t}]

           2
        a t
Out[2]= ---- + t v0
         2
```

CAMBRIDGE

Cambridge University Press
0521663954 - Computer Science with Mathematica: Theory and Practice for Science, Mathematics, and Engineering
Roman E. Maeder
Excerpt
More information

If we set v_0 to 0 and a to g, we get the formula for the distance traveled in free fall.

```
In[3]:= s[t] /. {v0 -> 0, a -> g}

              2
         g t
Out[3]= ----
          2
```

The time it takes to bring a vehicle to a complete stop is obtained as the solution of this equation for final velocity 0.

```
In[4]:= Solve[v[tb] == 0, tb][[1]]

                     v0
Out[4]= {tb -> -(--)}
                     a
```

This time gets us the braking distance. The value grows quadratically with initial velocity v_0. When the brakes are applied, the acceleration a is negative. The value is therefore positive, despite the minus sign.

```
In[5]:= s[tb] /. %

             2
         -v0
Out[5]= ----
         2 a
```

This kind of formula manipulation is typical of many scientific problems. A symbolic computation system can work with formulae and equations just as easily as an ordinary programming language can work with numbers.

How such a symbolic computation system works is quite a different matter. The first symbolic computation systems were written in LISP, which allows us to work with symbolic expressions directly. We need "only" implement the underlying mathematical algorithms. We shall take a look at LISP in Section 9.2.

1.1.3 Simulation: The Value of π

A simple physical experiment allows us to measure the area of a quarter of a disk and thus to determine the value of π. We choose repeatedly a uniformly distributed random point in the unit square and count how often it lies in the unit circle as well. The ratio of the number of points in the unit circle to the total number of points is equal to the ratio of the areas of the quarter disk and the unit square. Instead of performing the experiment in reality, we can simulate it on the computer.

Each invocation of `Random[]` returns a real number distributed uniformly in the interval from 0 to 1.

```
In[1]:= Random[]

Out[1]= 0.753989
```

This function (which has no arguments) gives a randomly chosen point in the unit square.

```
In[2]:= randomPoint := { Random[], Random[] }
```

Here is a list of 200 simulations in abbreviated form.

```
In[3]:= (data = Table[ randomPoint, {200} ]) // Short

Out[3]//Short=
  {{0.524444, 0.759749}, {0.989753, 0.518709},
   {0.46092, <<7>>31}, <<196>>, {0.51534, 0.801726}}
```

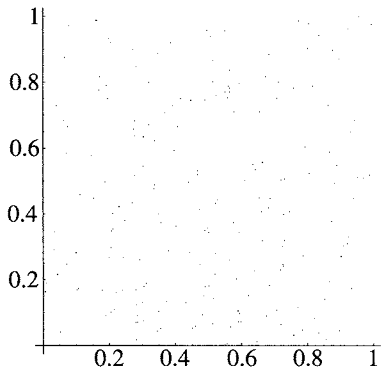
© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521663954 - Computer Science with Mathematica: Theory and Practice for Science, Mathematics, and Engineering
Roman E. Maeder
Excerpt
More information

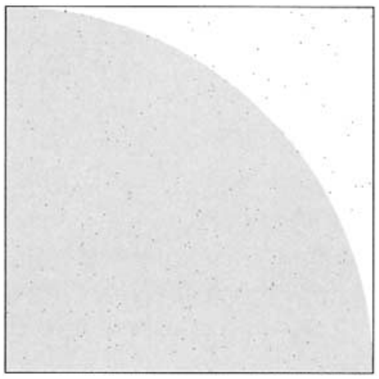
We can take a better look at the simulation results by drawing the points in the unit square.

```
In[4]:= ListPlot[ data, AspectRatio->Automatic ];
```



We are interested in the number of points in the circle. This picture highlights the circle inside the unit square.

```
In[5]:= Show[
          Graphics[{
            {GrayLevel[0.89], Disk[{0,0}, 1, {0,Pi/2}]},
            Line[{{0,0},{1,0},{1,1},{0,1},{0,0}}] }],
          %, AspectRatio->Automatic ];
```



This predicate tests whether a point lies in the circle, that is, whether the point's distance from the origin is ≤ 1.

```
In[6]:= inCircle[pt_] := Apply[Plus, pt^2] <= 1
```

Here is the fraction of points lying in the circle.

```
In[7]:= Count[ data, _?inCircle ] / Length[data] // N
Out[7]= 0.845
```

We repeat the experiment with 100,000 points. The result is an approximation of $\pi/4$.

```
In[8]:= data = Table[ randomPoint, {100000} ]; \
        Count[ data, _?inCircle ] / Length[data] // N
Out[8]= 0.78338
```

Here are six exact decimals of $\pi/4$.

```
In[9]:= N[ Pi/4 ]
Out[9]= 0.785398
```

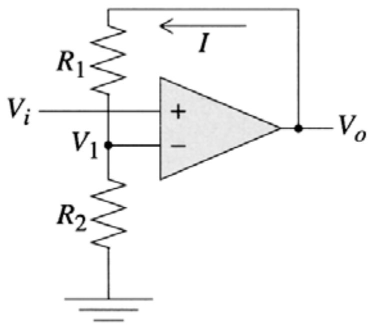
© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521663954 - Computer Science with Mathematica: Theory and Practice for Science, Mathematics, and Engineering
Roman E. Maeder
Excerpt
More information

This example uses the computer to simulate a physical experiment. One requirement for such a simulation is a *random-number generator*. It is used in *Monte Carlo* simulation methods to simulate a large number of trials. The results are then evaluated statistically. (Here, we simply calculated an average.)

1.1.4 Solution of Equations: Operational Amplifiers



The circuit shown on the left is a noninverting amplifier, realized using an *operational amplifier* (*op amp*). Because of the almost ideal properties of op amps, this circuit can be computed with a linear system of equations. We can assume that the difference of the two input voltages is zero, that is, $V_i = V_1$. Furthermore, the input resistance is infinite, which implies that the current through the two resistors R_1 and R_2 is the same. From Ohm's law, we arrive at equations $V_1 = IR_2$ and $V_o - V_1 = IR_1$. We are interested in the *voltage gain* $A_v = V_o/V_i$.

Here are the equations.

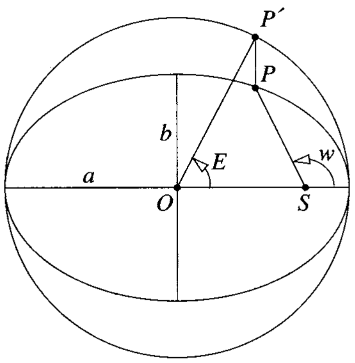
```
In[1]:= gl = { vi == v1,
               v1 == i r2,
               vo - v1 == i r1,
               av == vo/vi      };
```

Solving for A_v, we immediately get the standard formula for the voltage gain.

```
In[2]:= Solve[ gl, av, {vi, vo, v1, i} ]

                  r1 + r2
Out[2]= {{av -> -------}}
                    r2
```

1.1.5 Numerical Computation: Kepler's Equation



A planet P moves on an elliptic orbit with one focal point S being the sun. To determine its location on the orbit, according to the illustration, we need to determine the angle w, the *true anomaly*. The time M, called the *mean anomaly*, is measured starting from the point closest to the sun, such that one revolution is equal to 2π.

First, we determine the angle E, called the *eccentric anomaly*, according to Kepler's equation:

$$M = E - \varepsilon \sin E\,, \qquad (1.1\text{–}5)$$

where $\varepsilon = (a^2 - b^2)/a$ denotes the eccentricity of the ellipse with semimajor axes a and b. Equation 1.1–5 cannot be solved for E in closed form, but we can use an iterative method.

© Cambridge University Press www.cambridge.org

CAMBRIDGE

Cambridge University Press
0521663954 - Computer Science with Mathematica: Theory and Practice for Science, Mathematics, and Engineering
Roman E. Maeder
Excerpt
More information

We write the equation in the form

$$E = M + \varepsilon \sin E\,. \tag{1.1–6}$$

The equation is now of the form $E = f(E)$, with $f(E) = M + \varepsilon \sin E$, and it can be solved by iteration. We let $e_0 = M$ and iterate $e_i = f(e_{i-1})$, for $i = 1, 2, 3, \ldots$.

In our example, the ratio of the axes is $a/b = 3/2$. Here is the corresponding value of the eccentricity.

```
In[1]:= eps = N[ Sqrt[3^2 - 2^2]/3 ]
Out[1]= 0.745356
```

Here is the iteration function.

```
In[2]:= f[E_, M_] := M + eps Sin[E]
```

We perform 14 iterations for $M = \pi/2$, that is, after one-quarter of a revolution. We can see that values converge quickly toward a solution.

```
In[3]:= NestList[ Function[E, f[E, Pi/2]], N[Pi/2], 14 ]
Out[3]= {1.5708, 2.31615, 2.11852, 2.20712, 2.17028,
  2.18618, 2.17942, 2.18231, 2.18108, 2.18161, 2.18138,
  2.18148, 2.18144, 2.18145, 2.18145}
```

The command `FixedPoint[]` performs the iteration as many times as is necessary to find the solution.

```
In[4]:= FixedPoint[ Function[E, f[E, Pi/2]], N[Pi/2] ]
Out[4]= 2.18145
```

The function `Kepler[`M`]` allows us to compute E for any given value of M.

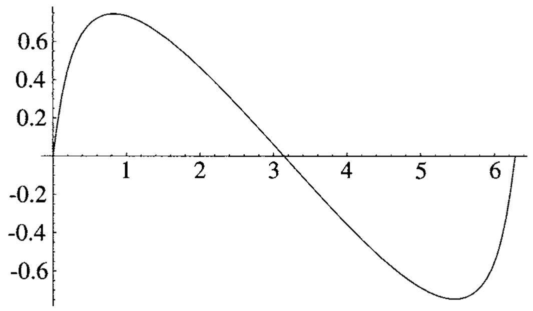
```
In[5]:= Kepler[M_] :=
            FixedPoint[ Function[E, f[E, M]], N[M],
                        SameTest -> Equal ]
```

Again, here is the solution for $M = \pi/2$.

```
In[6]:= Kepler[ Pi/2 ]
Out[6]= 2.18145
```

This curve shows the difference between mean and eccentric anomalies during one revolution of the planet.

```
In[7]:= Plot[ Kepler[M] - M, {M, 0, 2Pi} ];
```



This example uses one of the simplest approximation methods for solving equations that do not have a solution in closed form. Because most equations occurring in practice are of this kind, such methods are important.

© Cambridge University Press www.cambridge.org