

Unit 1: Introduction, Basics, Matlab

Notes prepared by: Amos Ron, Yunpeng Li, Mark Cowlshaw, Steve Wright
Instructor: Steve Wright

1 A Numerical Scheme

Consider the following example.

EXAMPLE 1.1. *Solve the quadratic equation*

$$x^2 + bx + c = 0 \tag{1}$$

We can find the solution by performing some simple algebraic manipulations leading to the familiar quadratic formula:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4c}}{2} \tag{2}$$

Does this formula bring us any closer to finding the numerical solutions of this problem? If we are lucky, and $b^2 - 4c$ happens to be the square of some easily identifiable number, we can compute the solutions using a couple of additions and divisions. Likewise, almost any calculator nowadays has a built-in square root operation, and we can use it to find a numerical solution even for general b and c . Suppose however that we are able only to perform the four fundamental arithmetic operations of addition, subtraction, multiplication, and division. In this case, the formula (2) is not much use, unless we can somehow come up with a good numerical approximation to the square root, obtained using only $+$, $-$, \times , and $/$. Fortunately such schemes exist, as we discuss next.

1.1 Approximating the Square Root

So how do we calculate an approximate square root? Before we can answer, we must consider two of the fundamental limitations of numerical computation:

- Limited precision. Since there is only a limited amount of space to store any number in a computer, we cannot always store it exactly. Rather, we can only get a solution that is accurate up to the limit of our machine precision (for example, 16 digits of accuracy).
- Limited time. Each arithmetic operation (e.g. addition, multiplication) takes time to perform. Often we face a resource budget; the approximate solution must be available within a limited time.

Just to evaluate the quadratic function $f(x) = ax^2 + bx + c$ at a point x takes time. Doing it in the obvious way takes three multiplications ($x * x$, $a * x^2$, $b * x$) and two additions. (Q: Can you see a faster way?)

We now present a method for finding the square root of a given (positive) number c by generating a sequence of guesses x_0, x_1, x_2, \dots , each of which is usually more accurate than its predecessor. x_0 is the initial guess, which we supply; x_1 is the next guess obtained from our method; x_2 is the one after that, and so on.

Our procedure uses the following simple formula to obtain each new guess x_{k+1} from the previous guess x_k :

$$x_{k+1} = \frac{x_k^2 + c}{2x_k} \tag{3}$$

If we start with an initial value $x_0 = 2.2$ and carry out this computation for three steps, we obtain the following values of x_1, x_2 , and x_3 . (We also tabulate the squares of these values, for interest.)

k	x_k	x_k^2
0	2.2	2.84
1	2.23636363636364	5.00132231404959
2	2.23606799704361	5.00000008740261
3	2.23606797749979	5.00000000000000

Using this procedure, only $3 \times 4 = 12$ simple arithmetic operations are required to arrive at a numerical solution that is accurate to the limit of Matlab's precision. Taking x_3 to be essentially exact, and comparing it with the earlier values x_0, x_1 , and x_2 , we see that x_0 has 2 correct digits, x_1 has 4 correct digits, and x_2 has 8 correct digits. The number of correct digits doubles with each guess - a rapid rate of convergence.

Soon we will study the basis for the excellent performance of this scheme, which is a particular case of Newton's method, and apply it to more complicated problems. In fact, before going further, we derive a generalization of this scheme for finding the roots of the quadratic (1) directly, rather than computing the square root approximation and plugging it into the formula (2). Given one approximation x_k to a root of (1), we obtain the next guess by applying the following formula:

$$x_{k+1} = x_k - \frac{x_k^2 + bx_k + c}{2x_k + b}, \tag{4}$$

which, by simple elimination, is equivalent to the following:

$$x_{k+1} = \frac{x_k^2 - c}{2x_k + b}.$$

2 Loss of Significance

Another issue in numerical computation is loss of significance. Consider two real numbers stored with 10 digits of precision beyond the decimal point:

$$\begin{aligned} a &= 3.1415926535 \\ b &= 3.1415926571 \end{aligned}$$

If we subtract a from b , we will get the answer 0.0000000036. This result has only two significant digits; it may differ from the underlying true value of this difference by more than 1%. We have lost most of the precision of the two original numbers.

One might try to solve this problem by increasing the precision of the original numbers, but this only improves the situation up to a point. Regardless of how high the precision is, if it remains finite, numbers that are close enough together will be indistinguishable. An example of the effect of loss of significance occurs in calculating derivatives.

EXAMPLE 2.1 (Calculating Derivatives). *Given the function $f(t) = \sin t$, what is $f'(2)$?*

We all know from calculus that the derivative of $\sin t$ is $\cos t$, so one could calculate $\cos(2)$ in this case. However, just as with calculating definite integrals, it is not always possible, or efficient to evaluate the derivative of a function by hand.

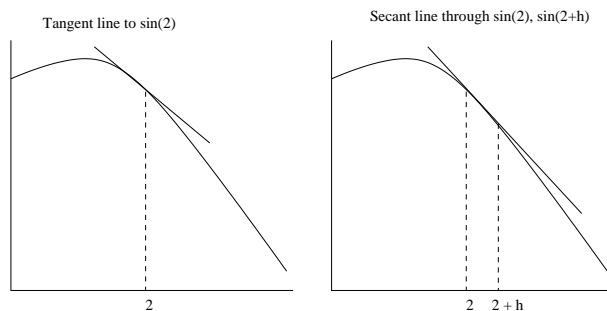


Figure 1: Tangent and Secant Lines on $f(t) = \sin t$

Instead, recall that the value of the derivative of a function f at a point t_0 is the slope of the line tangent to f at t_0 . Recall, also, that we can approximate this slope by calculating the slope of a secant line that intersects f at t_0 and at a point very close to t_0 , say $t_0 + h$, as shown in Figure 2. We learned in calculus, that as $h \rightarrow 0$, the slope of the secant line approaches the slope of the tangent line, that is,

$$f'(t_0) = \lim_{h \rightarrow 0} \frac{f(t_0 + h) - f(t_0)}{h} \quad (5)$$

Just as with the definite integral, we could approximate the derivative at t_0 by performing this calculation for a small (but nonzero) value of h , and setting our approximate derivative to be

$$\frac{f(t_0 + h) - f(t_0)}{h}. \quad (6)$$

This procedure is known as *finite differencing*. However, notice that if h is very small, $f(t_0 + h)$ and $f(t_0)$ are *the same* in finite precision. That is, the numerator of (6) will be zero in finite precision, and this calculation will erroneously report that all derivatives are zero.

Preventing loss of significance in our calculations will be an important part of this course.

3 Calculating Definite Integrals

Another good example of the difference between numerical computation and analytic techniques is found in the calculation of definite integrals. Recall that the definition of the definite integral of a continuous (and we will assume positive) function $f(t)$ over the interval $[a, b]$, $\int_a^b f(t) dt$ is the area under the curve, as shown in Figure 3.

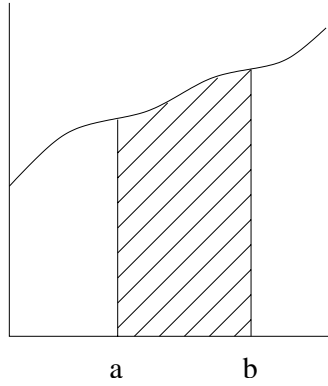


Figure 2: The Definite Integral of $f(t)$ over $[a, b]$

What is the best way to calculate the definite integral? Our experience in math classes suggests that we might use the anti-derivative, that is a function F with $F' = f$:

$$\int_a^b f(t)dt = F(b) - F(a) \quad (7)$$

But this technique raises two important questions:

1. How do we determine an anti-derivative function?
2. How difficult is it to evaluate the anti-derivative function at a and b ?

The answer to the first question may seem simple - we all learned to calculate anti-derivatives in calculus class. But we recall from the same classes that are many situations in which we do not have a simple rule for determining the anti-derivative.

There are, likewise, many examples of anti-derivative functions that are more difficult to evaluate than the original function, for example:

$$\int_1^x \frac{1}{t} dt = \log_e x - \log_e 1 = \log_e x \quad (8)$$

So is there a method for calculating the definite integral that uses only information we already know about the function - namely, the value of f at any given point? The answer is yes, and the key is to use the definition of the definite integral to approximate its value, as shown in Figure 3. We start by splitting up the interval into subintervals of equal size h , then estimating the definite integral over each subinterval, then adding them all together. If we split the interval $[a, b]$ into N subintervals of equal width $h = \frac{b-a}{N}$, we can calculate the definite integral as:

$$\int_a^b f(t)dt = \sum_{j=1}^N \int_{a+(j-1)h}^{a+jh} f(t)dt \quad (9)$$

We have reduced the problem to estimating the definite integral over smaller subintervals. Here are two possible strategies for estimating the smaller definite integrals:

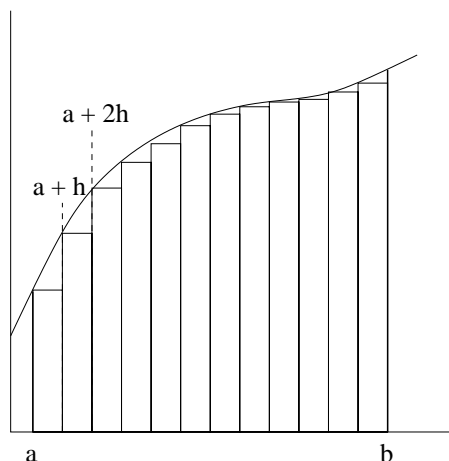


Figure 3: Approximating the Definite Integral Using Subintervals

Rectangle Rule

$$\int_{a+(j-1)h}^{a+jh} f(t)dt \simeq f(a + (j - 1)h)h \quad (10)$$

In the rectangle rule, we calculate the area of the rectangle with width h and height determined by evaluating the function at the left endpoint of the interval.

Midpoint Rule

$$\int_{a+(j-1)h}^{a+jh} f(t)dt \simeq f\left(a + (j - 1)h + \frac{h}{2}\right)h \quad (11)$$

In the midpoint rule, we calculate the area of the rectangle with width h and height determined by evaluating the function at the midpoint of the interval

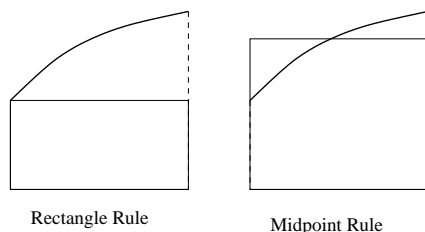


Figure 4: The Rectangle and Midpoint Rules

These two methods are depicted in Figure 3. Using these methods, we can calculate the definite integral by simply evaluating the original function at a number of points, which we should always be able to do (otherwise, we don't know what the function is). We can control the precision in each case by changing the size of the subintervals h (and therefore increasing the number of subintervals). The smaller the subinterval, the greater the precision.

The next question is, how can we decide which of these algorithms to use? It seems that the midpoint rule would be a better choice (Q: why?), but how can we justify this suspicion?

4 Evaluating Numerical Algorithms

We evaluate numerical algorithms using 3 major criteria:

Cost - The amount of time (usually calculated as the number of operations) the computation will take.

Speed - How quickly the algorithm approaches our desired precision. If we define the error as the difference between the actual answer and the answer returned by the numeric algorithm, speed is often measured as the rate at which the error approaches 0. The rate is measured in terms of some quantity that's relevant to the problem at hand, such as one evaluation of a function or derivative, or one step of an iterative process.

Robustness - How the correctness of the algorithm is affected by different types of functions and different types of inputs. For example, can an iterative method be relied on to find a solution, even if we start from a bad initial point?

Let's use these criteria to compare and contrast the midpoint and rectangle rules.

Cost Each algorithm performs a single function evaluation to estimate the definite integral over a subinterval, so the cost is identical. The total cost over the entire computation grows linearly with the number of intervals N in each case, meaning that there is some fixed constant c_1 , not dependent on the number of intervals, such that the total cost is approximately $c_1N = O(N)$. For example, if the number of intervals doubles, the cost of evaluating the approximate integral will double as well.

Speed Clearly as the size of each subinterval h gets smaller, the error gets smaller as well. Is there any difference between the speed with which the rectangle and midpoint rules approach the true solution, as h decreases (and the number of function evaluations increases?)

Rectangle Rule -For the rectangle rule, the error approaches 0 in direct proportion to the rate at which h approaches 0. In other words, the error $\epsilon = c_1h$ for some fixed constant c_1 , or $\epsilon = O(h)$. For example, halving the size of h will halve the amount of error.

Midpoint Rule -For the midpoint rule, the error approaches 0 in proportion to the rate at which the *square* of h approaches 0. That is the error $\epsilon = c_2h^2$ for some fixed constant c_2 , or $\epsilon = O(h^2)$. For example, halving the size of h will divide the error by 4. This is a much faster rate of convergence in general.

Robustness For functions that are reasonably well-behaved, both methods are quite robust. By "well-behaved" we mean that the function f is not too "wiggly" relative to the interval width h . We make this concept more precise later.

Through this evaluation, we have shown that, in general, the midpoint rule is a *far* better choice than the rectangle rule for approximating the definite integral, since, for the same cost, it will give us a much more accurate answer.

5 Matlab

Matlab will be used for many of the assignments in this class. Many of you already have experience with it. There are a few ways to get up to speed:

- Links to some online resources are available from the course web site.
- A “Matlab Tutorial” is available from the DoIT Tech Store. A later version of this same document is the book *Matlab Primer* by K. Sigmon and T. A. Davis, 7th edition, 2004, available from Amazon and elsewhere.
- The “help” command within Matlab itself is invaluable.
- Assignment 0 for this class has a brief introduction, as well as some exercises to be handed in.