AMTH250

Integration, Differential Equations

Gary Bunting

October 24, 2011

Contents

1 Numerical Integration

The problem we going to study is the numerical evaluation of integrals

$$I = \int_a^b f(x)\,dx.$$

Almost all approximations have the form

$$I \approx \sum_{i=1}^{n} w_i f(x_i).$$

The points x_i where the function is evaluated are called **nodes** and the multipliers w_i are called **weights**.

1.1 The Midpoint Rule

Divide the interval $[a, b]$ into n equal subintervals of length $h = (b - a)/n$. Let the nodes $x_i, i = 1, \ldots, n$ be the midpoint of the i-th interval and approximate the integral by the areas of the rectangles with base h centered on x_i and height $f(x_i)$.

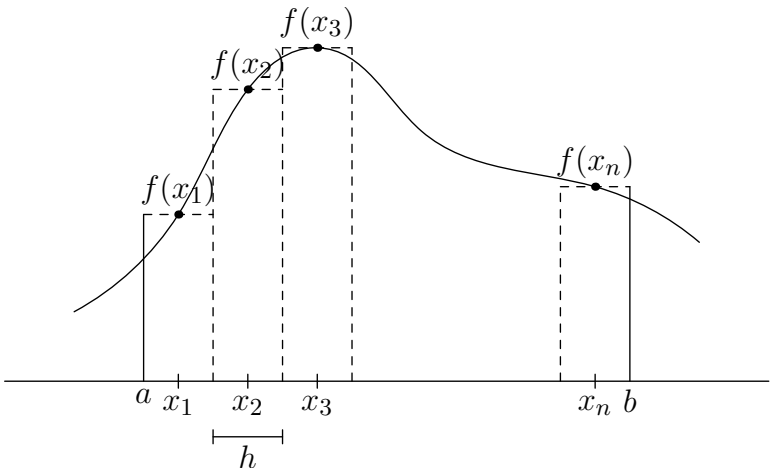


Figure 1: The midpoint rule

This gives the approximation

$$\int_a^b f(x)dx \approx h \sum_{i=1}^{n} f(x_i)$$

It is easily implemented in Octave:

```
function i = midint(f, a, b, n)
  h = (b-a)/n;
  x = linspace(a+h/2, b-h/2, n);
  i = h*sum(f(x));
endfunction
```

Example

As our example we will evaluate the integral

$$I = \frac{2}{\sqrt{\pi}} \int_0^1 e^{-x^2} dx$$

associated with the normal distribution. The function

$$\text{Erf}\,(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

is called the **error function** and is known to Octave.

An accurate value of our integral is

```
octave:> ii = erf(1)
ii =  0.842700792949715
```

We will evaluate the integral using the midpoint rule with $n = 2, 4, 8, \ldots, 1024$ points and examine the error:

```
octave:> f = @(x) 2*exp(-x.^2)/sqrt(pi);
octave:> intf = zeros(1,10);
octave:> for k = 1:10
> intf(k) = midint(f, 0, 1, 2^k);
> end
octave:> intf
intf =
 Columns 1 through 8:
   0.85147   0.84487   0.84324   0.84284   0.84273   0.84271   0.84270   0.84270
 Columns 9 and 10:
   0.84270   0.84270
octave:> errf= abs(intf-ii)
errf =
 Columns 1 through 6:
   8.7718e-03   2.1699e-03   5.4100e-04   1.3516e-04   3.3783e-05   8.4455e-06
 Columns 7 through 10:
   2.1114e-06   5.2784e-07   1.3196e-07   3.2990e-08
```

We see that when n increases by a factor of 2, so that h decreases by a factor of 2, the error decreases by a factor of 4, which suggests that the error is proportional to h^2. This is supported by the log-log plot of the error.

```
octave:> loglog(errf, 1./2.^(1:10),'o')
```

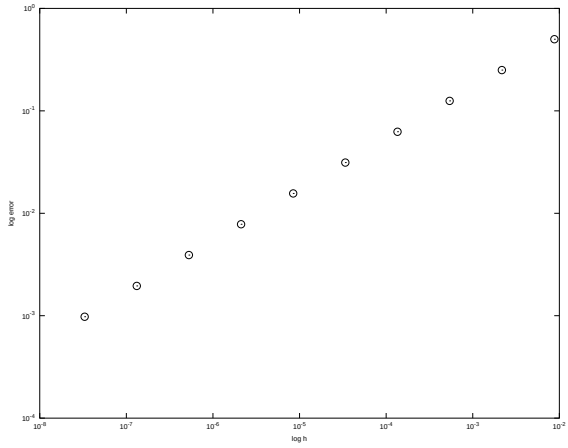


Figure 2: Error in midpoint rule

1.2 The Trapezoidal Rule

Again divide the interval $[a, b]$ into n equal subintervals of length $h = (b - a)/n$. Let the nodes $x_i, i = 0, \ldots, n$ be endpoints of the intervals and approximate the integral by the linearly interpolating the function values at the nodes and taking the areas of the resulting trapeziums.

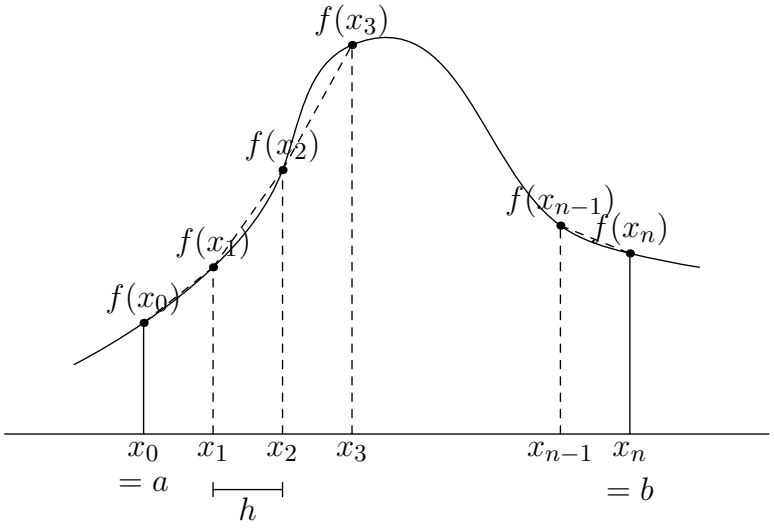
The area of the trapezium with base $[x_i, x_{i+1}]$ is

$$\frac{[f(x_i) + f(x_{i+1})]}{2} h$$

giving the approximation

$$\begin{aligned}\int_a^b f(x)dx &\approx \frac{h}{2}\left([f(x_0) + f(x_1)] + [f(x_1) + f(x_2)] + [f(x_2) + f(x_3)] + \right.\\ &\qquad \left. \cdots + [f(x_{n-1}) + f(x_n)]\right)\\ &= \frac{h}{2}\left(f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n)\right)\end{aligned}$$

Again, this is easy to implement in Octave:




```
function i = trapint (f, a, b, n)
  h = (b-a)/n;
  x = linspace(a, b, n+1);
  i = h*(sum(f(x))-(f(x(1)) + f((x(n+1))))/2);
endfunction
```

Example

We will perform the same calculations as for the midpoint rule:

```
octave:> for k = 1:10
> intf(k) = trapint(f, 0, 1,2^k);
> end
octave:> intf
intf =
 Columns 1 through 8:
   0.82526   0.83837   0.84162   0.84243   0.84263   0.84268   0.84270   0.84270
 Columns 9 and 10:
   0.84270   0.84270
octave:> errf= abs(intf-ii)
errf =
 Columns 1 through 6:
   1.7438e-02   4.3330e-03   1.0816e-03   2.7029e-04   6.7565e-05   1.6891e-05
 Columns 7 through 10:
   4.2227e-06   1.0557e-06   2.6392e-07   6.5980e-08
```

The results are very similar to those for the midpoint rule. The error is again proportional to h^2 but, for this function, the error for a given value of h is about twice error in the midpoint approximation.

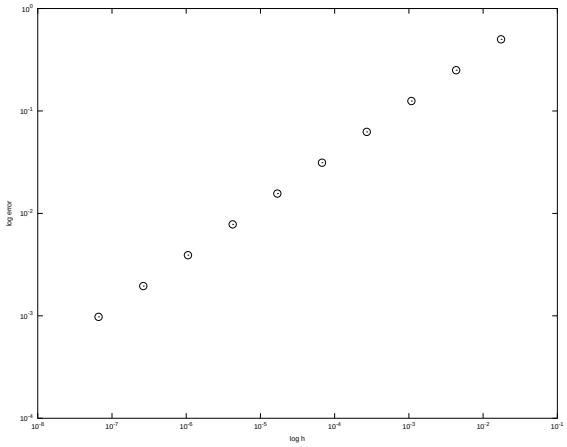


Figure 3: Error in trapezoidal rule

1.3 Simpson's Rule

Once more we divide the interval $[a, b]$ into n equal subintervals of length $h = (b-a)/n$. Again the nodes $x_i, i = 0, \ldots, n$ are endpoints of the intervals. For Simpson's rule we assume that n is *even* and we interpolate the function values over *pairs* of subintervals by a quadratic polynomial.

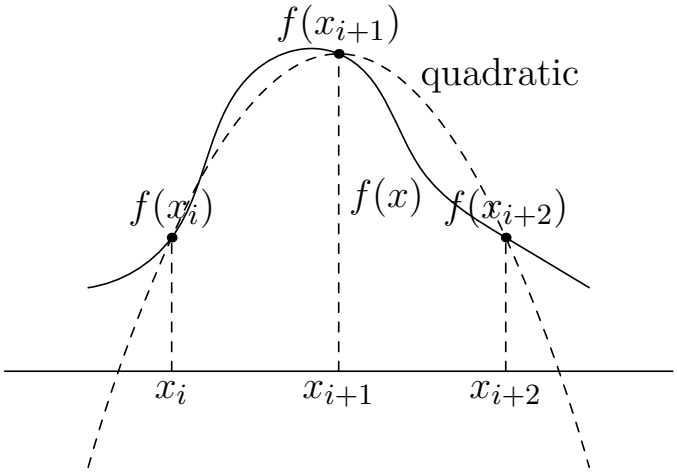


Figure 4: Simpson's rule

The integral is approximated by the area under the interpolating quadrat-

ics. A simple calculation shows that this area is

$$\frac{h}{6}\left[f(x_i) + 4f(x_{i+1}) + f(x_{i+2})\right]$$

giving the approximation

$$\begin{aligned}\int_a^b f(x)dx \approx \frac{h}{6}\left([f(x_0) + 4f(x_1) + f(x_2)] + [f(x_2) + 4f(x_3) + f(x_4)]\right.\\ +[f(x_4) + 4f(x_5) + f(x_6)] + \cdots + \left.[f(x_{n-2} + 4f(x_{n-2}) + f(x_n)]\right)\\ = \frac{h}{6}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + +2f(x_4) + \right.\\ \left.\cdots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\right)\end{aligned}$$

By considering the odd and even nodes separately, we see that Simpson's approximation with n subintervals is the sum of 2/3 times the midpoint approximation with $n/2$ subintervals plus 1/3 times the trapezoidal approximation with $n/2$ subintervals. We can use this to give a very simple Octave implementation of Simpson's method:

```
function y = simpint(f, a, b, n)
  y = (2*midint(f, a, b, n/2) + trapint(f, a, b, n/2))/3;
endfunction
```

Example

Using the same example as before:

```
octave:> for k = 1:10
> intf(k) = simpint(f, 0, 1,2^k);
> end
octave:> intf
intf =
 Columns 1 through 8:
   0.84310   0.84274   0.84270   0.84270   0.84270   0.84270   0.84270   0.84270
 Columns 9 and 10:
   0.84270   0.84270
octave:> errf= abs(intf-ii)
errf =
 Columns 1 through 6:
   4.0204e-04   3.5258e-05   2.2429e-06   1.4062e-07   8.7952e-09   5.4980e-10
 Columns 7 through 10:
   3.4364e-11   2.1477e-12   1.3434e-13   8.5487e-15
octave:> semilogy(errf,'o')
```

The error in Simpson's method decreases by a factor of 16 when h decreases by a factor of 2, indicating that the error is proportional to h^4, a substantial improvement of the midpoint and trapezoidal methods.

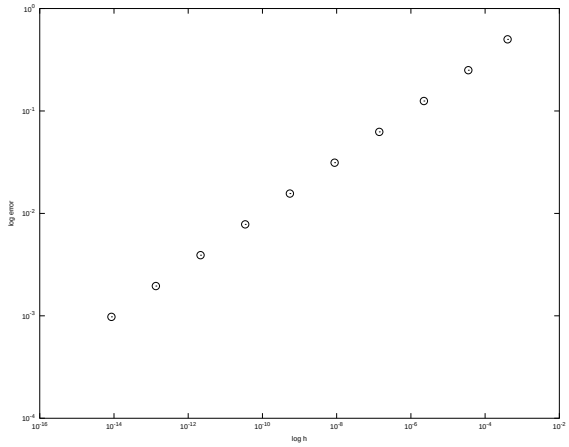


Figure 5: Error in Simpson's rule

1.4 Error Analysis

The major source of error in numerical integration is **truncation error** – the error due to approximating an infinite sum, the Riemann integral by a finite sum.

1.4.1 Truncation Error

Our experiments showed that error behaved regularly as a function of the step-size h; for the midpoint and trapezoidal methods the error was proportional to h^2, and for Simpson's rule the error was proportional to h^4.

1. **Midpoint Rule:**
$$\text{Error} \approx E(f)h^2.$$
2. **Trapezoidal Rule:**
$$\text{Error} \approx F(f)h^2.$$
3. **Simpson's Rule:**
$$\text{Error} \approx G(f)h^4.$$

1.4.2 Rounding Error

Numerical integration does not present any problems as far as rounding errors are concerned. In general, we expect the relative error due to rounding

to be the order of $\varepsilon_{\text{mach}}$ for any numerical integral. To see why we will return to the general form of the numerical approximation

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

Rounding errors will arise from the additions, multiplications by the w_i and in the function evaluations. In fact we can think of the general formula as repeatedly performing the operations (1) evaluate $f(x_i)$, (2) multiply by w_i, and (3) add. The sequence of operations will produce a rounding error of order

$$e_i \approx w_i f(x_i)\varepsilon_{\text{mach}}$$

and the total rounding error will be of order

$$\begin{aligned} e &\approx \sum_{i=1}^{n} w_i f(x_i)\varepsilon_{\text{mach}} \\ &\approx \varepsilon_{\text{mach}} \sum_{i=1}^{n} w_i f(x_i) \\ &\approx \varepsilon_{\text{mach}} \int_a^b f(x)dx \end{aligned}$$

giving a relative error in the integral of $\varepsilon_{\text{mach}}$.

As an demonstration, we will show that Simpson's rule can evaluate our example integral to an accuracy of order $\varepsilon_{\text{mach}}$. From the previous section we see that for $n = 1024$ we have an error of about 10^{-14}. Doubling n will decrease the error by a factor of 16 which will bring it down to the order of machine epsilon:

```
octave:> ef = simpint(f, 0, 1, 2048)
ef =  0.84270
octave:> ef - ii
ans =  5.5511e-16
```

1.4.3 Estimating Errors

For the methods we have looked at, the truncation errors varied in a predictable way with the step-size h. The same thing is typical of almost all methods of numerical integration. This fact can be used to estimate the error from the approximations themselves.

We will illustrate the idea using Simpson's method. In this case we know

$$\text{Error} \approx G(f)h^4.$$

Now suppose we approximate the integral using a step-size h giving the value I_1 and then half the step-size to $h/2$ giving the value I_2. The corresponding

errors will be called E_1 and E_2. Now, the error in I_2 will be about 1/16 the error in I_1, that is,

$$E_2 \approx \frac{1}{16} E_1.$$

The fact that I_2 is a much better approximation than I_1 also means that

$$E_1 \approx I_2 - I_1.$$

Combining these gives the estimate

$$E_2 \approx \frac{1}{16}(I_2 - I_1).$$

Example

We will see how this idea works using our example. We will use Simpson's rule to approximate our integral and also to estimate the error in the approximation:

```
octave:> i1 = simpint(f,0,1,32)
i1 =  0.842700801744932
octave:> i2 = simpint(f,0,1,64)
i2 =  0.842700793499513
octave:> e2 =(i2-i1)/16
e2 = -5.15338709206059e-10
```

This compares well with the actual error:

```
octave:> err = ii - i2
err = -5.49797651849815e-10
```

1.4.4 Adaptive Strategies

Consider the problem of approximating an integral to a prescribed accuracy. A crude approach is choose some initial step-size h and repeatedly halve the step-size, estimating the error as we go, until the prescribed accuracy is reached.

A process like this is most efficient, in the sense of minimizing the number of function evaluations, when the nodes at one step are reused in the next step. This is trivially the case for the methods we have looked at which have equally spaced nodes.

We can be more clever about reducing function evaluations by using the fact that we can estimate the error on individual subintervals just as well as the error in the total integral. This means that if the error on some subinterval is judged sufficiently small, relative to the required total error, no further subdivision of that interval need be attempted.

In general, functions which are oscillating rapidly or display rapid changes of shape are more difficult to integrate, that is, require more function evaluations to achieve a given accuracy, than functions that vary slowly. Thus the effect of adaptive methods is to concentrate nodes in regions where the integral is difficult to evaluate, while using a lower density of nodes in regions where the integral is easier to evaluate.

1.5 Numerical Integration in Octave

Octave has a number of procedures for numerical integration. The main one is `quad`, see §22.1 of the Octave documentation for the others. The most basic way to use it is

```
y = quad(func, a, b)
```

With our example

```
octave:> intf = quad(f, 0, 1)
intf =  0.84270
octave:> errf = intf - ii
errf = 0
```

Infinite values are allowed for the `a` and `b`.

```
octave:> quad(f, -Inf, Inf)
ans =  2.0000
```

`quad` can also handle integrands with singularities, as long as the function is integrable of course:

```
octave:> g = @(x) 1./sqrt(x);
octave:> quad(g, 0, 1)
ans =  2.0000
```

Absolute and relative tolerances for the integral can be specified and an estimate of the absolute error returned. The most general way to use `quad` is

```
[y, ierr, nf, err] = quad(func, a, b, [atol, rtol], sing)
```

`y` — an estimate of the integral.

`ierr` — error code (0 indicates a successful integration).

`nf` — number of function evaluations used.

`err` — an estimate of the absolute error in the integral.

`atol` **and** `rtol` — requirements on the absolute and relative error of the integral. If you only require one of these, set the other to zero.

`sing` — a vector of values at which the integrand is known to be singular.

Example

Let

$$I = \int_0^\infty e^{-x} \cos x \, dx = 0.5$$

We will use `quad` to evaluate the integral with an absolute error tolerance of 1×10^{-14}.

```
octave:> f = @(x) exp(-x).*cos(x);
octave:> [intf,ierr,nf,eerr] = quad(f,0,Inf,[1e-14 0])
intf =  0.50000
ierr = 0
nf =  465
eerr =  8.2034e-15
```

Experimenting with absolute tolerance $1 \times 10^{-1} \ldots 1 \times 10^{-14}$ gives the following results.

Tolerance	Function Evaluations	Estimated Error	Actual Error
1×10^{-1}	75	5.01×10^{-2}	5.60×10^{-6}
1×10^{-2}	105	4.92×10^{-4}	1.32×10^{-8}
1×10^{-3}	105	4.92×10^{-4}	1.32×10^{-8}
1×10^{-4}	135	4.32×10^{-7}	7.05×10^{-12}
1×10^{-5}	135	4.32×10^{-7}	7.05×10^{-12}
1×10^{-6}	135	4.32×10^{-7}	7.05×10^{-12}
1×10^{-7}	165	1.86×10^{-8}	7.05×10^{-12}
1×10^{-8}	195	3.11×10^{-9}	0
1×10^{-9}	255	6.78×10^{-10}	0
1×10^{-10}	285	3.31×10^{-11}	0
1×10^{-11}	315	5.28×10^{-13}	1.11×10^{-16}
1×10^{-12}	315	5.28×10^{-13}	1.11×10^{-16}
1×10^{-13}	375	7.32×10^{-14}	1.11×10^{-16}
1×10^{-14}	465	8.20×10^{-15}	0

As the tolerance decreases the number of function evaluations increases. Note that the error estimates produced by `quad` are quite conservative.

Example

Here is an example in which we must specify the singularity to avoid the failure of `quad`. The integral is

$$I = \int_{-1}^{1} \frac{1}{\sqrt{|x|}} \, dx = 4$$

```
octave:> f = @(x) 1./sqrt(abs(x));
octave:> quad(f, -1, 1)
warning: division by zero
 ABNORMAL RETURN FROM DQAGP
ans = Inf
```

If we specify the singularity at $x = 0$ we get the expected result.

```
octave:> quad(f, -1, 1, [1e-10 0], [0])
ans =  4.0000
```

1.6 Monte-Carlo Integration

1.6.1 One Dimension

We wish to evaluate an integral

$$I = \int_a^b f(x)dx$$

The average value of $f(x)$ is

$$\bar{f} = \frac{1}{b-a}I$$

so that

$$I = (b-a)\bar{f}$$

Suppose we choose points $x_1, \ldots, x_n$ randomly in the interval $[a, b]$ and use these to estimate the average value of $f(x)$:

$$\bar{f} \approx \frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

then

$$I \approx \frac{(b-a)}{n}\sum_{i=1}^{n} f(x_i)$$

This is **Monte-Carlo integration**.

A Note on Uniform Random Numbers

The Octave function `rand(m,n)` produces a `m` $\times$ `n` matrix of uniform $[0, 1]$ random numbers. To generate random numbers in an interval $[a, b]$ we use the fact that if x is uniformly distributed on $[0, 1]$, then $y = a + (b - a)x$ is uniformly distributed on $[a, b]$. For example, to get a matrix of $[-5, 5]$ uniform random numbers we use `10*rand(m,n)-5`.

Implementation

Here is a Octave implementation of Monte-Carlo integration of a function `f` over the interval `[a,b]` using `n` points. Note that the function `f` must be able to operate on a vector of values.

```
function ii = monte1d(f, a, b, n)
  x = (b-a)*rand(1,n) + a;   % choose n random points in [a,b]
  fx = f(x);                 % evaluate f(x) at each point
  ii = (b-a)*sum(fx)/n;      % (b-a) * average of function
endfunction
```

For our example

$$I = \frac{2}{\sqrt{\pi}} \int_0^1 e^{-x^2} dx$$

we will perform a Monte-Carlo approximation with $2, 4, 8, \ldots, 2^{20}$ points using a simple `for` loop and then compute the relative error for each approximation:

```
octave:> intf = zeros(1,20);
octave:> for k =1:20
> intf(k) = monte1d(f, 0, 1, 2^k);
> end
octave:> errf = abs(intf-ii)/ii
errf =
 Columns 1 through 6:
   3.6284e-02   2.5553e-04   6.0049e-03   7.2232e-02   3.4352e-03   2.7370e-02
 Columns 7 through 12:
   1.7148e-02   1.7599e-02   6.6710e-03   9.9621e-03   1.2955e-03   7.0485e-03
 Columns 13 through 18:
   2.7914e-03   5.6502e-04   1.7187e-03   1.3316e-03   1.5152e-03   1.6540e-04
 Columns 19 and 20:
   3.9963e-04   2.1032e-05
```

As expected the error generally decreases as the number of points increases. Plotting the log of the error is the most instructive:

```
octave:> semilogy(errf)
```

1.6.2 The Central Limit Theorem

The central limit theorem of probability theory gives an estimate of the error in Monte-Carlo integration. For our purposes it can be formulated as follows: Suppose the mean of a function $f(x)$ is estimated by random sampling

$$\bar{f}_{\text{est}} = \frac{1}{n} \sum_{i=1}^{n} f(x_i)$$

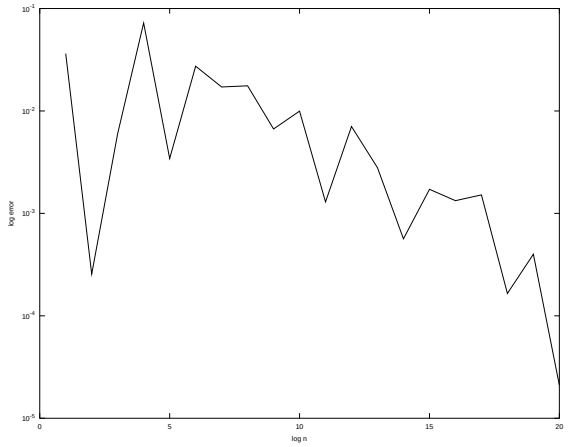


Figure 6: Error in Monte-Carlo integration

Then the variance of the estimated mean is

$$\text{Var } \bar{f} = \frac{\sigma^2}{n}$$

where σ^2 is the variance of $f(x)$.

It is usual to take the standard deviation as a measure of error. Then we have

$$\text{Error} = \frac{\sigma}{\sqrt{n}}$$

The important point here is that the error goes to zero like $1/\sqrt{n}$. This says, for example, that to decrease the error by a factor of 1000, we must increase the sample size by a factor of 1000000.

Example

Let us see how the error in our example compares to theory. The variance of

$$f(x) = \frac{2}{\sqrt{\pi}} e^{-x^2}$$

is

$$\sigma^2 = \int_0^1 (f(x) - \bar{f})^2 \approx 0.14$$

(by numerical integration) so the expected error in Monte-Carlo integration is

$$E_n = \frac{\sigma}{\sqrt{n}} \approx \frac{0.14}{\sqrt{n}}$$

```
octave:> n=1:20;
octave:> mcerr = 0.104*((2.^n).^(-1/2));
octave:> hold on
octave:> semilogy(mcerr)
```

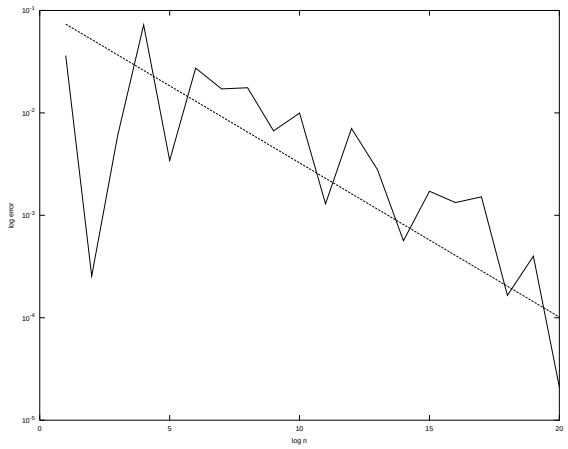


Figure 7: Error in Monte-Carlo integration

The error in the computed values show random fluctuations, as is to be expected, but that the theory gives a good account of the error.

1.6.3 Estimating Volumes

What is the volume of a unit sphere in 4-dimensions? We can obtain an estimate by Monte-Carlo methods.

The unit sphere is the region

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 \leq 1$$

If we generate random points in the four dimensional cubic region $[-1, 1] \times [-1, 1] \times [-1, 1] \times [-1, 1]$ which contains the unit sphere, then an estimate of the volume of the sphere is

$$\frac{\text{Volume of Sphere}}{\text{Volume of Cube}} \approx \frac{\text{No. Points in Sphere}}{\text{No. of Points in Cube}}$$

By symmetry we get the same result if work in a single quadrant, say $[0, 1] \times [0, 1] \times [0, 1] \times [0, 1]$.

Here is the calculation in Octave. First we write a function `monte4d` to do the calculation:

```
function v = monte4d(n)
  k = 0;               % count of number of points in sphere
  for i = 1:n
    x = rand(1,4);     % x = point in unit cube
    if (norm(x) <= 1)  % point x lies in sphere
      k = k+1;
    end
  end
  v = 16*k/n;          % 16 quadrants!
endfunction

octave:> monte4d(100000)
ans =  4.9509
```

By the way, the exact volume is $\pi^2/2 = 4.9348$.

1.6.4 Multiple Integrals

Generalizing the formula for Monte-Carlo integration in one dimension, we have a formula for Monte-Carlo integration in any number of dimensions: if

$$I = \int_\Omega f \, dV$$

then

$$I \approx \text{Volume of } \Omega \times \text{Average Value of } f \text{ in } \Omega$$

For one-dimensional integration problems Monte-Carlo integration is quite inefficient. For high-dimensional integration it is a useful technique. The reasons for this are twofold:

1. The fact that error is proportional to $1/\sqrt{n}$ does not depend on the dimension. In other words it performs just as well in high dimensions as in one dimension.

2. It can easily handle regions with irregular boundaries. The method used in the previous section to estimate volumes can easily be adapted to Monte-Carlo integration over any region.

Example

We will estimate the integral

$$I = \iint_\Omega \sin \sqrt{\ln(x + y + 1)} \, dx \, dy$$

where Ω is the disk

$$\left(x - \frac{1}{2}\right)^2 + \left(y - \frac{1}{2}\right)^2 \leq \frac{1}{4}$$

Since the disk Ω is contained within the square $[0,1] \times [0,1]$, we can generate x and y as uniform $[0,1]$ random numbers, and keep those which lie in the disk Ω.

```
function ii = monte2da(n)
  k = 0;              % count no. of points in disk
  sumf = 0;           % keep running sum of function values
  while (k < n)       % keep going until we get n points
    x = rand(1,1);
    y = rand(1,1);
    if ((x-0.5)^2 + (y-0.5)^2 <= 0.25)       % (x,y) is in disk
      k = k + 1;                             % increment count
      sumf = sumf + sin(sqrt(log(x+y+1)));   % increment sumf
    end
  end
  ii = (pi/4)*(sumf/n);     % pi/4 = volume of disk
endfunction

octave:> monte2da(100000)
ans =  0.56794
```

Example

In the Monte-Carlo approximation

$$I \approx \text{Volume of } \Omega \times \text{Average Value of } f \text{ in } \Omega$$

we can estimate the volume of the region Ω at the same time as we estimate the average of the function f.

We generate points in a volume V – usually rectangular – containing Ω. If we generate n points in V of which k lie in the region Ω then

$$\text{Volume } \Omega \approx \frac{k}{n} \text{ Volume } V$$

Since

$$\text{Average Value of } f \text{ in } \Omega \approx \frac{1}{k} \sum f$$

the k's cancel and we have

$$I \approx \text{Volume } V \times \frac{1}{n} \sum f.$$

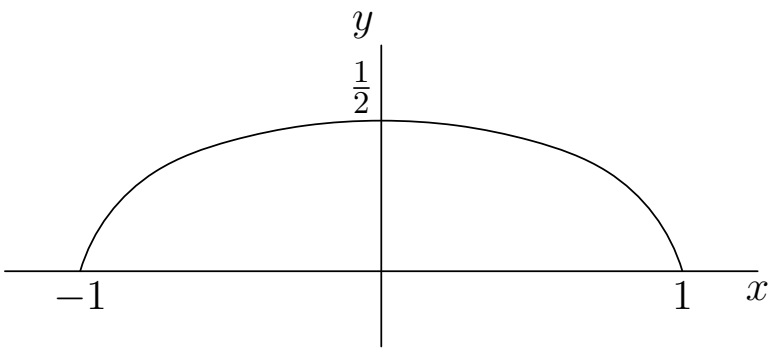
where the sum is over the points lying in the region Ω.

We will now evaluate the integral

$$I = \iint_{\Omega} y\, dxdy$$

over the semi-elliptical region Ω given by

$$x^2 + 4y^2 \leq 1, \quad y \geq 0$$



Although there is a simple formula for the area of an ellipse, we will use Monte-Carlo to estimate the area as we perform the integration. Since the region Ω is contained within the rectangle $[-1, 1] \times [0, 1/2]$, we generate x and y as uniform $[-1, 1]$ and uniform $[0, 1/2]$ random numbers respectively. We will generate `n` random numbers, and use those lying in Ω to form the sum of the function. Note that the rectangular region has area = 1.

```
function ii = monte2db(n)
  sumf = 0;           % keep running sum of function values
  for i = 1:n
    x = 2*rand(1,1) - 1;       % x in [-1,1]
    y = rand(1,1)/2;           % y in [0,1/2]
    if (x^2 + 4*y^2 <= 1)      % point lies in region
      sumf = sumf + y;         % increment sumf
    end
  end
  ii = sumf/n;                 % Volume = 1
endfunction

octave:> monte2db(100000)
ans =  0.16639
```

The exact value is $1/6$.

2 Differential Equations

2.1 First Order Equations

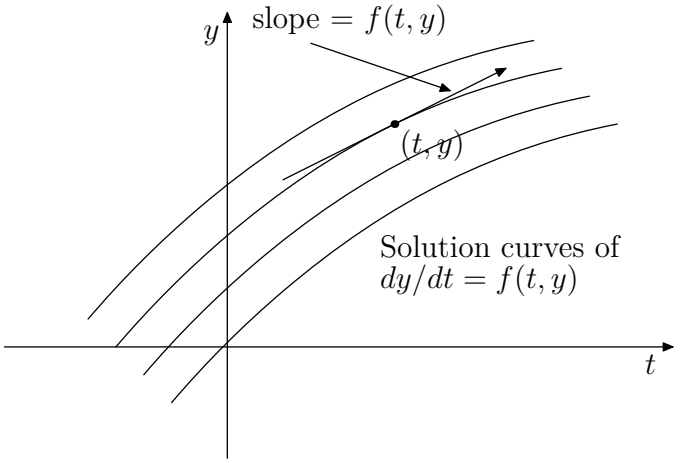
A **first order differential equation** is an equation of the form

$$\frac{dy}{dt} = f(t, y). \tag{1}$$

Once an **initial value**

$$y(t_0) = y_0$$

is specified, we have an **initial value problem**. An initial value problem has a unique solution $y(t)$ giving y as a function of t. Another way of saying the same thing is that for each point (t_0, y_0) of the t-y plane there is a unique solution of equation (1) passing through that point.



Geometrically, the differential equation can be interpreted as saying that the solution curve through a point (t, y) has slope $f(t, y)$ at that point.

2.1.1 Euler's Method

The simplest numerical method for solving initial value problems for differential equations is **Euler's method**. Suppose we have an initial value problem

$$\frac{dy}{dt} = f(t, y), \qquad y(t_0) = y_0.$$

Choose a step-size Δt, then, by the first order Taylor's approximation

$$y(t_0 + \Delta t) \approx y(t_0) + \frac{dy}{dt}(t_0)\Delta t$$

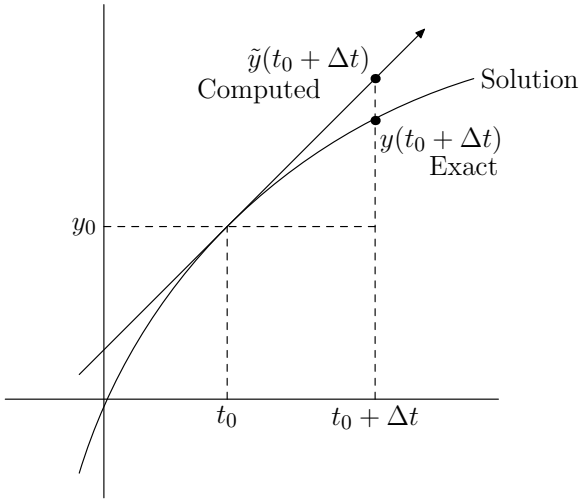
The differential equation says $dy/dt = f(t, y)$ so that

$$y(t_0 + \Delta t) \approx y_0 + f(t_0, y_0)\Delta t$$

Denote the approximate solution obtained in this way by $\tilde{y}(t)$, then Euler's method can be written

$$\tilde{y}(t_0 + \Delta t) = y_0 + f(t_0, y_0)\Delta t$$

Euler's method has a simple geometric interpretation – we approximate the solution curve through a point (t_0, y_0) by its tangent which we know has the slope $f(y_0, t_0)$.



We have seen how one step of Euler's method works. The same procedure can be repeated over any number of steps: start with the initial condition

$$\tilde{y}(t_0) = y_0$$

and then apply Euler's method to obtain the approximations

$$\begin{aligned}
\tilde{y}(t_0 + \Delta t) &= \tilde{y}(t_0) + f(t_0, \tilde{y}(t_0))\Delta t \\
\tilde{y}(t_0 + 2\Delta t) &= \tilde{y}(t_0 + \Delta t) + f(t_0 + \Delta t, \tilde{y}(t_0 + \Delta t))\Delta t \\
\tilde{y}(t_0 + 3\Delta t) &= \tilde{y}(t_0 + 2\Delta t) + f(t_0 + 2\Delta t, \tilde{y}(t_0 + 2\Delta t))\Delta t \\
\tilde{y}(t_0 + 4\Delta t) &= \tilde{y}(t_0 + 3\Delta t) + f(t_0 + 3\Delta t, \tilde{y}(t_0 + 3\Delta t))\Delta t \\
&\vdots
\end{aligned}$$

Example

We will use as our example the differential equation

$$\frac{dy}{dt} = -2ty$$

This equation can be solved by separation of variables to give the general solution

$$y(t) = Ae^{-t^2}$$

The initial value problem

$$\frac{dy}{dt} = -2ty, \qquad y(0) = 1$$

then has the solution

$$y(t) = e^{-t^2}$$

We will write a Octave function `euler(f, t0, y0, dt, n)` to approximate the solution of

$$\frac{dt}{dt} = f(t, y)$$

using Euler's method with initial conditions $t =$ `t0`, $y =$ `y0`, and time-step `dt` over `n` steps.

```
function [t,y] = euler(f, t0, y0, dt, n)
  t = zeros(n+1,1);
  y = zeros(n+1,1);
  t(1) = t0;
  y(1) = y0;
  for i = 1:n
    t(i+1) = t(i) + dt;
    y(i+1) = y(i) + f(t(i), y(i))*dt;
  end
endfunction
```

Applying `euler` to our example:

```
octave:> f = @(t,y) -2*t*y;
octave:> [t,y] = euler(f, 0, 1, 0.01, 200);
octave:> plot(t,y)
```

We can compare the computed solution to the exact solution:

```
octave:> err = y - exp(-t.^2);
octave:> plot(t,err)
```

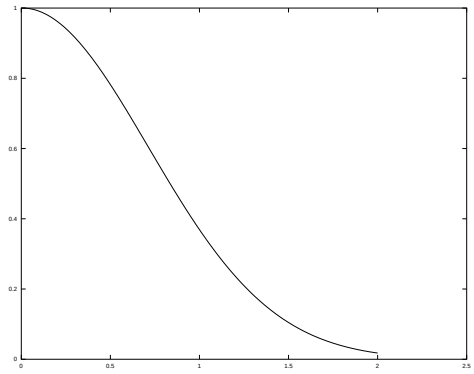


Figure 8: Numerical solution by Euler's method

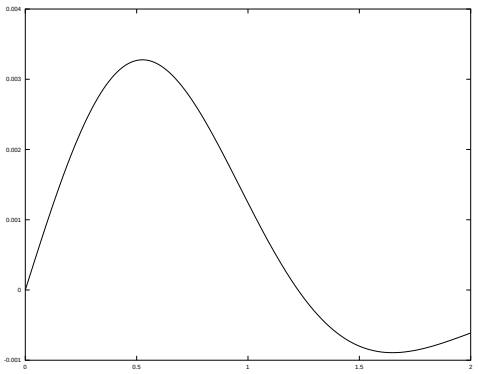


Figure 9: Error in Euler's method

We see that the error is small enough, of the order of 10^{-3}, for the computed and exact solutions to be barely distinguishable on a graph. The shape of the error curve is not significant.

[Aside: the general shape of the error curve can be simply explained. From the graphical interpretation of Euler's method it follows that Euler's method overestimates the solution when the slope of the solution is increasing (as in the diagrams) and underestimates the solution when the slope of the solution is decreasing. The change from one type of behaviour to the other will occur near a point of inflection, which, in the exact solution to our example occurs at $x = 1/\sqrt{2}$. There are also other types of error to consider as well to give a complete account.]

Euler's method does always perform as well as it does in the example above. Here is an example where it gives wildly inaccurate results:

```
octave:> [t,y]=euler(f,0,1,1,10);
octave:> y'
ans =
 Columns 1 through 7:
          1           1          -1           3         -15         105        -945
 Columns 8 through 11:
      10395     -135135     2027025  -34459425
```

The important change here was the step size. It is clear from the derivation of Euler's method that the accuracy should increase as the step size is decreased, but this example shows that if the step size is too large then totally inaccurate results can be obtained.

2.1.2 Other Methods

The methods used in practice for the numerical solution of differential equations are much more sophisticated than Euler's method. The main issue is the trade-off between smaller step sizes to increase accuracy and larger step sizes to increase efficiency, with larger step sizes fewer steps are needed to cover a given range. Here are brief descriptions of the rationale behind some common numerical methods:

1. **Runge-Kutta methods:** Euler's method uses only the information at one point (t, y) and takes a step with slope $f(t, y)$. Runge-Kutta methods, by contrast, evaluate $f(t, y)$ at a number of points for each step. A simple example of this idea is to evaluate $f(t, y)$ and $f(t + \Delta t, y + \Delta y)$, where $y + \Delta y$ is obtained from Euler's method, and take a step whose slope is the average of the two values.

2. **Implicit methods:** A simple example of an implicit method is the *backward Euler method*. Euler's formula is

$$\tilde{y}(t_0 + \Delta t) = \tilde{y}(t_0) + f(t_0, \tilde{y}(t_0))\Delta t$$

   The backward Euler formula is

$$\tilde{y}(t_0 + \Delta t) = \tilde{y}(t_0) + f(t_0 + \Delta t, \tilde{y}(t_0 + \Delta t))\Delta t$$

   this is an *implicit* formula for $\tilde{y}(t_0 + \Delta t)$ which needs to be solved numerically before the step can be taken. Implicit methods are typically more stable, in the sense of avoiding the problems associated with large step sizes, than explicit methods. The disadvantage is that a nonlinear equation has to solved at each step.

3. **Multistep methods:** These use not only the information at the current point but information at previously computed points in making

each step. Since such information is not available at the first step they need other methods to get started. There are explicit and implicit multistep methods. A typical use is to use an explicit method to make a tentative step and then use the value obtained as an initial approximation for an implicit step. Such methods are often referred to as *predictor-corrector* methods.

2.1.3 Octave

The Octave function for solving initial value problems is `lsode`. Its simplest use is of the form

```
y = lsode(f, y0, t)
```

Here `f` is the right hand side of the differential equation, `y0` is the initial value and `t` is the vector of t values at which the solution is to be computed. The solution at these points is returned in the vector `y`. Don't confuse the vector `t` with steps taken by the solver; the solver itself determines the step sizes it takes and then computes the values of `y` at the values specified by `t` by interpolation.

Here is our example using `ode`:

```
octave:> t = (0:0.01:2)';
octave:> y = lsode(f,1,t);
octave:> plot(t,y)
octave:> err = y - exp(-t.^2);
octave:> plot(t,err)
```

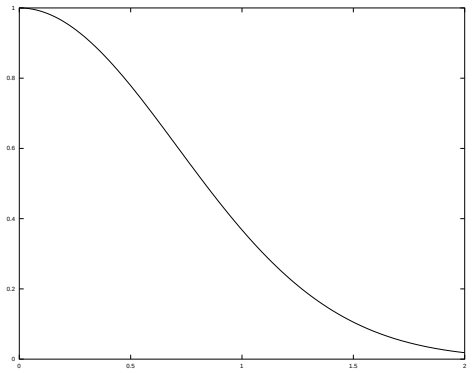


Figure 10: Numerical solution by `lsode`

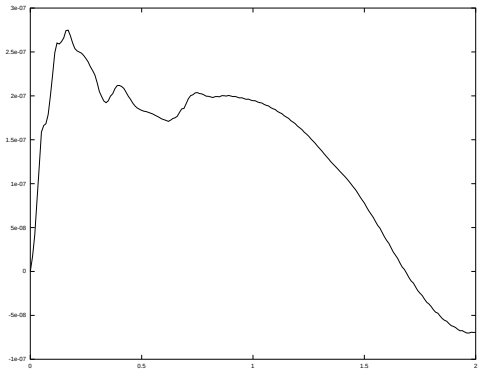


Figure 11: Error in `lsode` solution

The rough shape of the graph of the error is mostly due to the step size changing during the computation.

2.2 First Order Systems

A **system of first order differential equations** has the form

$$\begin{aligned} \frac{dy_1}{dt} &= f_1(t, y_1, \ldots, y_n) \\ \frac{dy_2}{dt} &= f_2(t, y_1, \ldots, y_n) \\ &\vdots \\ \frac{dy_n}{dt} &= f_n(t, y_1, \ldots, y_n) \end{aligned}$$

that is, we have a system of n first order differential equations for n functions $y_1(t), \ldots, y_n(t)$.

An initial value problem specifies the initial values of all n functions at the same point t_0:

$$\begin{aligned} y_1(t_0) &= y_{1,0} \\ y_2(t_0) &= y_{2,0} \\ &\vdots \\ y_n(t_0) &= y_{n,0} \end{aligned}$$

As in the case of a single equation, an initial value problem for a first order system has a unique solution.

It is often convenient to write the equations in vector form; if $\mathbf{y} = (y_1, \ldots, y_n)$ then the system of differential equations can be written

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y})$$

where $\mathbf{f}$ is the vector of functions $f_1, \ldots, f_n$. The initial values can also be written in vector form

$$\mathbf{y}(t_0) = \mathbf{y}_0$$

2.2.1 Lotka-Volterra Equations

As an example we will look at the Lotka-Volterra equations which are used to model the dynamics of predator-prey interactions. The prey population is denoted by y_1 and the predator population by y_2. The differential equations are

$$\begin{aligned} \frac{dy_1}{dt} &= y_1(\alpha_1 - \beta_1 y_2) \\ \frac{dy_2}{dt} &= y_2(-\alpha_2 + \beta_2 y_1) \end{aligned}$$

The parameters α_1 and α_2 are the natural growth rates of the populations, and the parameters β_1 and β_2 determine the interaction between the two populations.

2.2.2 Euler's Method

Recall that Euler's method for a single equation takes the form

$$y(t + \Delta t) = y(t) + f(t, y)\Delta t.$$

Simply replacing the scalars y and f by vectors gives a formula which makes sense for systems of equations

$$\mathbf{y}(t + \Delta t) = \mathbf{y}(t) + \mathbf{f}(t, \mathbf{y})\Delta t.$$

In terms of the individual equations this reads

$$\begin{aligned} y_1(t + \Delta t) &= y_1(t) + f_1(t, y_1, \ldots, y_n)\Delta t \\ y_2(t + \Delta t) &= y_2(t) + f_2(t, y_1, \ldots, y_n)\Delta t \\ &\vdots \\ y_n(t + \Delta t) &= y_n(t) + f_n(t, y_1, \ldots, y_n)\Delta t \end{aligned}$$

We will adapt our procedure `euler` for a single equation to deal with systems of equations. Again it will take the form

```
[t, y] = euler(f, t0, y0, dt, n)
```

and perform `n` steps of Euler's method with step size `dt`. The difference now is that since we have a system of equations we need to careful about the sizes of the variables. If we have a system of `k` equations then we specify:

`t`:	a column vector of size `n+1`.
`y`:	a matrix of size `n+1` $\times$ `k`,
	the rows of `y` are the components at a given value of t,
`f(y,t)`:	the first argument is a row vector of size `k`,
	it returns a row vector of size `k`.
`y0`:	a row vector of size `k`.

Our function `euler` is essentially the same as for a single equation, except for the dimensions of the variables involved:

```
function [t, y] = euler(f, t0, y0, dt, n)
  k = length(y0);       % this determines the number of equations
  t = zeros(n+1, 1);
  y = zeros(n+1, k);
  t(1) = t0;
  y(1,:) = y0;
  for i = 1:n
    t(i+1) = t(i) + dt;
    y(i+1,:) = y(i,:) + f(y(i,:), t(i))*dt;
  end
endfunction
```

2.2.3 Example

As our example we will use the Lotka-Volterra equations with parameter values:

$$\alpha_1 = 1.0, \qquad \beta_1 = 0.1$$
$$\alpha_2 = 0.5, \qquad \beta_2 = 0.02$$

and initial conditions

$$y_1(0) = 100, \qquad y_2(0) = 10.$$

Rather than define the parameters within the differential equation, we define the parameters as **global variables**. Doing this allows us to easily change them if we wanted to experiment with different parameter values.

```
octave:> global a1 a2 b1 b2
octave:> a1 = 1;
octave:> a2 =0.5;
octave:> b1 = 0.1;
octave:> b2 = 0.02;
```

Now the function for the right hand side of our system of equations is:

```
function dy = lv(y, t)
  global a1 a2 b1 b2
  dy = zeros(2,1);
  dy(1) = y(1)*(a1 - b1*y(2));
  dy(2) = y(2)*(-a2 + b2*y(1));
endfunction
```

We will solve the differential equations over the interval $t = [0, 25]$ using steps of $\Delta t = 0.01$.

```
octave:> [t,y] = euler(@lv, 0, [100 10], 0.01, 2500);
octave:> plot(t,y)
```

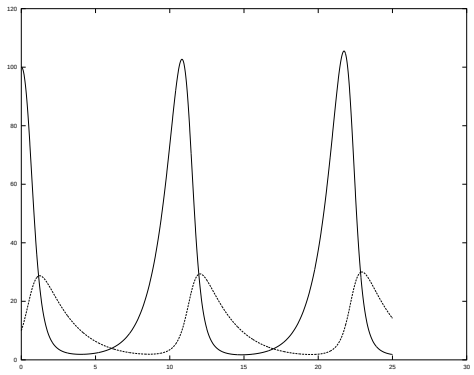


Figure 12: Solution Lotka-Volterra equations by Euler's method

Note that the plot works because `plot2d` plots each of the columns of `y` and we have set things up so that the columns correspond to the two populations y_1 and y_2.

The curve with larger values is y_1 the prey population. Both populations show noticeable oscillations but they do not oscillate in phase. This has a biological interpretation. First the prey numbers are decreasing as the predators are increasing – the predators are eating the prey and increasing in number because of the plentiful supply of food. Then the predator numbers decrease as the supply of food continues to decreases. When the predator numbers decrease enough the prey numbers start to increase and the cycle repeats itself. These sorts of oscillations has been observed in natural predator/prey populations.

Another useful graph in examples like this is a **phase-plane** plot when one component of the solution, y_1, is plotted against the other, y_2, rather than plotting each against t.

```
octave:> plot(y(:,1), y(:,2))
```

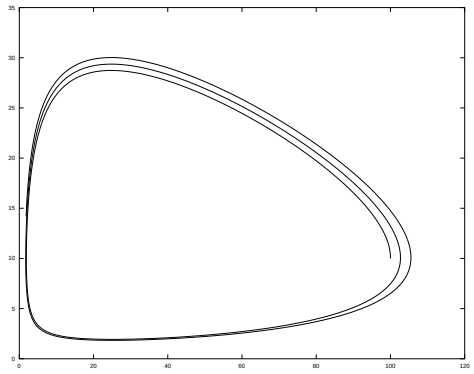


Figure 13: Phase plane plot for Lotka-Volterra equations

The main thing to be gained from a graph like this is how the two populations vary with one another. There is a clear cyclic pattern to the populations over time. Also apparent is the fact that the maxima and minima of the two populations do not coincide.

2.2.4 Octave

The function `lsode` works the same way for systems of equations as for a single equation

```
y = lsode(f, y0, t)
```

Here `y0` is the vector of initial values and the result, `y`, contains the components of the solution as columns.

Here is the Lotka-Volterra example using `ode`:

```
octave:> tt = 0:0.01:25;
octave:> y = lsode(@lv, [100,10], tt);
octave:> plot(t,y)
octave:> plot(y(:,1), y(:,2))
```

The results are very similar to those obtained with `euler`, although the phase plane plot shows an almost exactly closed curve indicating exact periodic behaviour in the solutions. (We would expect the more sophisticated method `lsode` to give more accurate solutions than `euler`.)

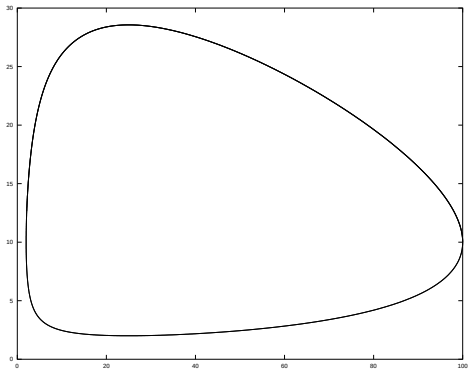


Figure 14: Phase plane plot for Lotka-Volterra equations using `lsode`

2.3 Higher Order Equations

2.3.1 Formulation

The differential equations we have studied so far contain only first derivatives. Higher order differential equations involve the second and possibly higher derivatives. An **nth-order differential equation** has the form

$$\frac{d^n y}{dt^n} = f\left(t, y, \frac{dy}{dt}, \frac{d^2y}{dt^2}, \ldots, \frac{d^{n-1}y}{dt^{n-1}}\right).$$

An initial value problem specifies the initial values of all the derivatives up to order $n-1$:

$$y(t_0) = y_0, \qquad \frac{dy}{dt}(t_0) = {y_0}', \quad \ldots \quad , \frac{d^{n-1}y}{dt^{n-1}}(t_0) = {y^{(n-1)}}_0$$

Again, such an initial value has a unique solution.

2.3.2 Equivalent 1st Order System

An initial value problems for a higher order equation can be reduced to an equivalent initial problem for a system of first order equations. This is convenient since we understand how to deal with the latter numerically.

Given an n-th order equation

$$\frac{d^n y}{dt^n} = f\left(t, y, \frac{dy}{dt}, \frac{d^2y}{dt^2}, \ldots, \frac{d^{n-1}y}{dt^{n-1}}\right) \tag{2}$$

introduce n new variables

$$\begin{aligned} w_1 &= y \\ w_2 &= \frac{dy}{dt} \\ w_3 &= \frac{d^2y}{dt^2} \\ &\vdots \\ w_n &= \frac{d^{n-1}y}{dt^{n-1}} \end{aligned}$$

There are $n-1$ relationships between the new variables:

$$\begin{aligned} \frac{dw_1}{dt} &= w_2 \\ \frac{dw_2}{dt} &= w_3 \\ &\vdots \\ \frac{dw_{n-1}}{dt} &= w_n \end{aligned}$$

which together with equation (2), which can be rewritten as,

$$\frac{dw_n}{dt} = f(t, w_1, w_2, \ldots, w_n)$$

gives a system of n first order equations for the n unknown functions $w_1, \ldots, w_n$.

The initial values for the n-th order equation

$$y(t_0) = y_0, \qquad \frac{dy}{dt}(t_0) = {y'}_0, \quad \ldots \quad , \frac{d^{n-1}y}{dt^{n-1}}(t_0) = {y^{(n-1)}}_0$$

are equivalent to the following initial values for our new variables

$$\begin{aligned} w_1(t_0) &= y(t_0) = y_0 \\ w_2(t_0) &= \frac{dy}{dt}(t_0) = {y'}_0 \\ w_3(t_0) &= \frac{d^2y}{dt^2}(t_0) = {y''}_0 \\ &\vdots \\ w_n(t_0) &= \frac{d^{n-1}y}{dt^{n-1}}(t_0) = {y^{(n-1)}}_0 \end{aligned}$$

Thus any initial value problem for an n-th order differential equation can be converted to an initial value problem for a system of n first order equations. Similarly, initial value problems for systems of higher order equations can be converted to equivalent problems for 1st order systems using the same techniques. The method is obvious but tedious to write out.

2.3.3 Example

According to Newton's laws the the position $(x(t), y(t))$ in the orbital plane of a body orbiting a much larger body of mass M centered at the origin is governed by the differential equations

$$\begin{aligned} \frac{d^2x}{dt^2} &= -GM\frac{x}{r^3} \\ \frac{d^2y}{dt^2} &= -GM\frac{y}{r^3} \end{aligned}$$

where G is the gravitational constant and $r = \sqrt{x^2 + y^2}$. Each of the differential equations is 2nd order so the pair of equations is equivalent to a system of four 1st order equations.

Introducing variables

$$\begin{aligned} w_1 &= x \\ w_2 &= \frac{dx}{dt} \\ w_3 &= y \\ w_4 &= \frac{dy}{dt} \end{aligned}$$

We have the relations

$$\frac{dw_1}{dt} = w_2$$

and

$$\frac{dw_3}{dt} = w_4$$

The original 2nd order equations can be written in terms of the new variables as

$$\frac{dw_2}{dt} = -GM\frac{w_1}{(w_1^2 + w_3^2)^{3/2}}$$

and

$$\frac{dw_4}{dt} = -GM\frac{w_3}{(w_1^2 + w_3^2)^{3/2}}$$

giving our system of four 1st order equations.

To solve these equations numerically, we will let `y` denote the vector (w_1, w_2, w_3, w_4). To simplify the computation we can choose units such that $GM = 1$, so the right hand side of system is given by the Octave function

```
function ydot = orbit(y, t)
  ydot = zeros(4,1);
  ydot(1) = y(2);
  ydot(3) = y(4);
  ydot(2) = - y(1)*(y(1)^2 + y(3)^2)^(-3/2);
  ydot(4) = - y(3)*(y(1)^2 + y(3)^2)^(-3/2);
endfunction
```

With the initial conditions

$$x(0) = 0.4 \qquad \frac{dx}{dt}(0) = 0$$
$$y(0) = 0 \qquad \frac{dy}{dt}(0) = 2$$

we solve over the interval $t = [0, 50]$:

```
octave:> y0 = [0.4 0 0 2];
octave:> t = 0:0.01:50;
octave:> y = lsode(@orbit, y0, t);
octave:> plot(y(:,1), y(:,3))
octave:> hold on
octave:> plot(0,0,'x')    % position of central body
```

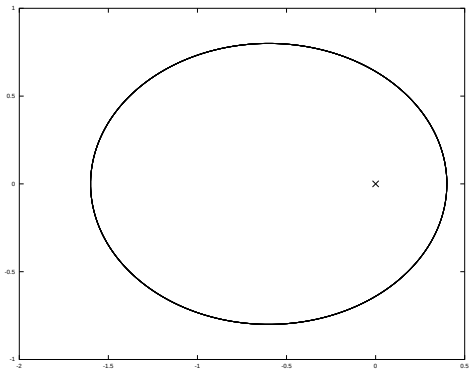


Figure 15: Phase plane plot for orbital motion

Note that `y(1,:)` and `y(3,:)` are the data corresponding to the variables $x(t)$ and $y(t)$. The graph shows that the orbit is an ellipse with the central body at a focus, something well known to Newton.