Chapter 2

Simulation

Main concepts: In this first chapter we introduce numerical integrators and time stepping. Keywords are: Euler's method, trapezoidal rule, Matlab code

2.1 Computational modelling

A solution to a differential equation may exhibit very complex behavior, as we will see later. The complexity increases with the dimension d, and problems becomes more and more difficult to treat by purely analytical means. Instead, in this course we will discuss the approximation of solutions of differential equations on a computer.

The most fundamental property of the numerical approximation is that it must be computable in a finite number of operations (unless you are willing to wait an eternity for the solution). The process of replacing the continuous solution by a finite one is called *discretization*.

To do so, we replace the interval $D = [t_0, t_0 + T]$ by a finite number N of discrete times $t_n = t_0 + nh$, $n = 0, 1, \ldots, N$, where $h = T/N$ is the step size. Similarly, we replace the continuous solution $y(t)$ on D with the numerical solution $y_n \approx y(t_n)$, $n = 0, 1, \ldots, N$. The y_n can be thought of as snapshots of the system state at the discrete times, and the sequence $\{y_0, y_1, \ldots, y_N\}$ as a movie. Finally, we need a procedure to generate the y_n for $n > 0$ (y_0 is the given initial condition). In this course, we will consider methods that approximate y_{n+1} recursively given y_n, h, and the explicit form of $f(y)$. Such methods are termed *one-step methods* to distinguish them from *multistep methods* which use information from several preceding time steps. A time-stepping procedure is referred to as a *numerical integrator*or *scheme*.

There are several approaches to the construction of integrators. The simplest and oldest method is based on the rectangle rule for approximation of an integral. Consider the special case of (1.2) with $f(t, y) = f(t)$ (f is independent of y). The rectangle rule is just $y_{n+1} = y_n + hf(t_n)$. Generalizing this to y-dependent problems is straightforward, and results in **Euler's method**

$$y_{n+1} = y_n + hf(t_n, y_n). \tag{2.1}$$

Euler's method can be interpreted in several other ways: (1) as the direct extrapolation of the local slope through (t_n, y_n), (2) as the single term truncation of a Taylor expansion, or (3) as a finite difference. For the last case, consider the definition of the derivative:

$$\frac{dy}{dt} = \lim_{h \to 0} \frac{y(t+h) - y(t)}{h} = f(t, y),$$

and to approximate, simply do not take the limit. Different interpretations can be used to generalize Euler's method and construct other methods. We will see some of these in later chapters.

To program this method, we write a loop and compute the successive iterates as follows:

Algorithm 2.1.1 (Euler's Method) Given: *initial time* t_0*, initial value* y_0*, stepsize* h*, a vector field* f*, and a number of time steps* N*,*
Output: $y_1, y_2, \ldots, y_N$ *approximating the solution of* $\frac{dy}{dt} = f(y,t)$ *at equally spaced points* $t_1 := t_0 + h, t_2 = t_0 + 2h, \ldots$.
for $n = 0, 1, \ldots, N-1$
$t_{n+1} := t_0 + nh$;
$y_{n+1} := y_n + hf(y_n, t_n)$;
end □

Let us apply Euler's method to solve the Lotka-Volterra model (1.6) with $r = 2$. Take the initial value to be $(n, p) = (0.5, 0.5)$. The relationship between predators and prey is shown in Figure 2.1. This figure is troublesome because it appears to show a trajectory which spirals out, which seems to suggest a problem with the model (if this trend continues, both populations may grow unbounded, which is not the kind of behavior we would expect from an isolated ecological system.) The cause of this behavior becomes more clear if we consider the effect of decreasing the stepsize, as shown in the right panel of Figure 2.1, where stepsizes $h = 0.1$, $h = 0.01$ and $h = 0.001$ are compared. Evidently the growth of the solution is in fact a consequence of growing solution error. A solution with a small stepsize of $h = 0.001$ appears to close up, suggesting the correct trajectory is a periodic orbit.

This obviously raises a very serious question. Since many studies of dynamical systems now rely heavily on computer experiments, there is a serious risk of drawing completely wrong conclusions, when the numerical dynamics and the true continuous dynamics are doing different things. A goal of this course is to understand what sort of properties a numerical method should have so that we can trust the computed results.

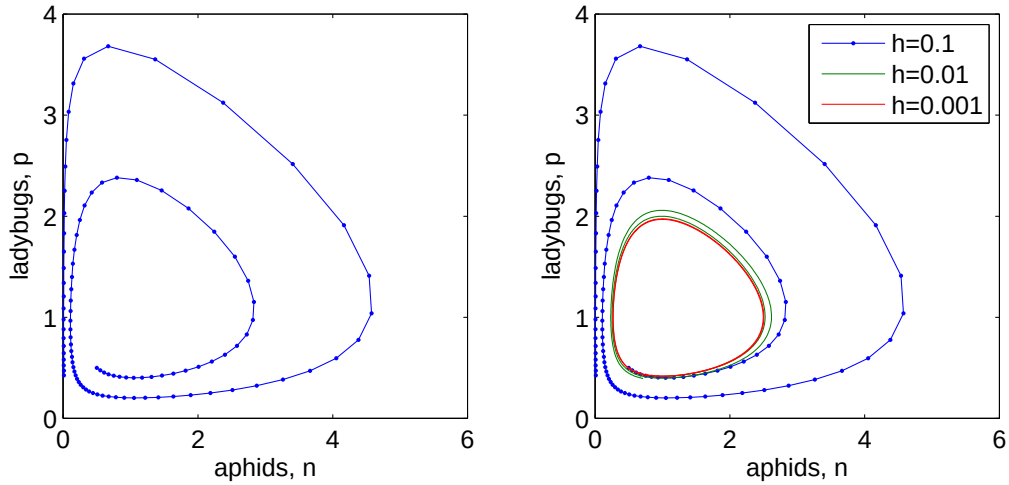

Figure 2.1: The locus curve of predators and prey for the Lotka-Volterra model, left with $h = 0.1$ and right, with $h = 0.1$ (100 steps), $h = 0.01$ (1000 steps) and $h = 0.001$ (10000 steps).

2.2 Trapezoidal rule

Before concluding this chapter with some example codes in Matlab, we introduce a second numerical method. From elementary calculus, you may remember that a better approximation than the rectangle rule is the trapezoidal rule. Again consider the equation $\frac{dy}{dt} = f(t)$. The trapezoidal rule approximation is $y_{n+1} = y_n + \frac{h}{2}(f(t_{n+1}) + f(t_n))$. Generalizing to the ODE (1.2), the **trapezoidal rule** is

$$y_{n+1} = y_n + \frac{h}{2}\left(f(t_n, y_n) + f(t_{n+1}, y_{n+1})\right). \tag{2.2}$$

Notice that this method is of a character very different from Euler's method. It is not possible to evaluate the last term in the equation without knowing y_{n+1}, and it is not possible to compute y_{n+1} without evaluating this term. The trapezoidal rule defines y_{n+1} *implicitly* as function of y_n. In other words, we must solve the typically nonlinear system of algebraic equations

$$0 = r(y) := y - y_n + \frac{h}{2}\left(f(t_n, y_n) + f(t_{n+1}, y)\right) \tag{2.3}$$

for y to determine y_{n+1}. Such methods are termed *implicit methods*, and generally demand much more work from the computer per time step than an *explicit method* such as Euler's method. Methods for solving systems of algebraic equations are covered in an appendix to these notes. Obviously an implicit method must be significantly advantageous in some other sense to justify its increased computational cost. We will say more about this in later chapters. Also, it is uncertain if (2.3) will have any real solutions, and if so, how many. For $h = 0$, there is the unique solution $y = y_n$, and for h small enough, the implicit function theorem assures us there is a unique solution. For larger h, uniqueness and existence are not guaranteed.

2.3 Matlab Code

The following m-file computes the value of the vector field of the Lotka-Volterra model (1.6) at a given point.
lv.m

```
function yprime = lv(t,y)
% lotka volterra vector field
r=2;  %coefficient
yprime =[0;0];   %creates a column vector with two elements
yprime(1) =  r*y(1)*(1  -  y(2));
yprime(2) =  y(2)*(y(1) - 1);
```

The following m-file implements Euler's Method.
euler.m

```
function [ylist,tlist] = euler(y0, t0, m, vf, h, N)
% solves dy/dt = f(y,t) in R^m starting from y(t0)=y0, using N steps of size h
% the initial condition should be an m-dimensional column vector
% vf(y,t) should evaluate the vector field f at the point (y,t)
% results are returned in tlist and ylist, the latter in the form of a matrix, each
% of whose columns represent the numerical solution at a timestep

tlist = zeros(1,N+1);
ylist = zeros(m, N+1);
tlist(1) = t0;
ylist(:,1) = y0;

for nn=1:N,
tlist(nn+1) = t0 + nn*h;
ylist(:,nn+1) = ylist(:,nn) + h*feval(vf, tlist(:,nn), ylist(:,nn));
end
```

A typical run would be performed and plotted as follows.

```
>> t0 = 0;  y0=[0.5; 0.5];
>> [ylist,tlist] = euler(y0,t0,2,'lv',0.1,100);
>> plot(tlist,ylist,'o-');
```