Differential equations and vectors

1 Differential equations

A **differential equation** (DE) is an equation that describes the derivatives of an unknown function. "Solving a DE" means finding a function whose derivatives satisfy the equation.

For example, when bacteria grow in particularly bacteria-friendly conditions, the rate of growth at any point in time is proportional to the current population. What we might like to know is the population as a function of time. Toward that end, let's define f to be a function that maps from time, t, to population y. We don't know what it is, but we can write a differential equation that describes it:

$$\frac{df}{dt} = af$$

where a is a constant that characterizes how quickly the population increases.

Notice that both sides of the equation are functions. To say that two functions are equal is to say that their values are equal at all times. In other words:

$$\forall t : \frac{df}{dt}(t) = af(t)$$

This is an **ordinary** differential equation (ODE) because all the derivatives involved are taken with respect to the same variable. If the equation related derivatives with respect to different variables (partial derivatives), it would be a **partial** differential equation.

This equation is **first order** because it involves only first derivatives. If it involved second derivatives, it would be second order, and so on.

This equation is **linear** because each term involves t, f or df/dt raised to the first power; if any of the terms involved products or powers of t, f and df/dt it would be nonlinear.

Linear, first order ODEs can be solved analytically; that is, we can express the solution as a function of t. This particular ODE has an infinite number of solutions, but they all have this form:

$$f(t) = be^{at}$$

For any value of b, this function satisfies the ODE. If you don't believe me, take its derivative and check.

If we know the population of bacteria at a particular point in time, we can use that additional information to determine which of the infinite solutions is the (unique) one that describes a particular population over time.

For example, if we know that $f(0) = 5$ billion cells, then we can write

$$f(0) = 5 = be^{a0}$$

and solve for b, which is 5. That determines what we wanted to know:

$$f(t) = 5e^{at}$$

The extra bit of information that determines b is called the **initial condition** (although it isn't always specified at $t = 0$).

Unfortunately, most interesting physical systems are described by nonlinear DEs, most of which can't be solved analytically. The alternative is to solve them numerically.

2 Euler's method

The simplest numerical method for ODEs is Euler's method. Here's a test to see if you are as smart as Euler. Let's say that you arrive at time t and measure the current population, y, and the rate of change, r. What do you think the population will be after some period of time Δt has elapsed?

If you said $y + r\Delta t$, congratulations! You just invented Euler's method (but you're still not as smart as Euler).

This estimate is based on the assumption that r is constant, but in general it's not, so we only expect the estimate to be good if r changes slowly and Δt is small.

But let's assume (for now) that the ODE we are interested in can be written so that

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

where g is some function that maps (t, y) onto r; that is, given the time and current population, it computes the rate of change. Then we can advance from one point in time to the next using these equations:

$$T_{n+1} = T_n + \Delta t \tag{1}$$

$$F_{n+1} = F_n + g(t, y)\ \Delta t \tag{2}$$

Here $\{T_i\}$ is a sequence of times where we estimate the value of f, and $\{F_i\}$ is the sequence of estimates. For each index i, F_i is an estimate of $f(T_i)$. The interval Δt is called the **time step**.

Assuming that we start at $t = 0$ and we have an initial condition $f(0) = y_0$ (where y_0 denotes a particular, known value), we set $T_1 = 0$ and $F_1 = y_0$, and then use Equations 1 and 2 to compute values of T_i and F_i until T_i gets to the value of t we are interested in.

If the rate doesn't change too fast and the time step isn't too big, Euler's method is accurate enough for most purposes. One way to check is to run it once with time step Δt and then run it again with time step $\Delta t/2$. If the results are the same, they are probably accurate; otherwise, cut the time step again.

Euler's method is **first order**, which means that each time you cut the time step in half, you expect the estimation error to drop by half. With a second-order method, you expect the error to drop by a factor of 4; third-order drops by 8, etc. The price of higher order methods is that they have to evaluate g more times per time step.

3 Another note on notation

There's a lot of math notation in this chapter so I want to pause to review what we have so far. Here are the variables, their meanings, and their types:

Name	Meaning	Type
t	time	scalar variable
Δt	time step	scalar constant
y	population	scalar variable
r	rate of change	scalar variable
f	The unknown function specified, implicitly, by an ODE.	function $t \to y$
df/dt	The first time derivative of f	function $t \to r$
g	A "rate function," derived from the ODE, that computes rate of change for any t, y.	function $t, y \to r$
T	a sequence of times, t, where we estimate $f(t)$	sequence
F	a sequence of estimates for $f(t)$	sequence

So f is a function that computes the population as a function of time, $f(t)$ is the function evaluated at a particular time, and if we assign $f(t)$ to a variable, we usually call that variable y.

Similarly, g is a "rate function" that computes the rate of change as a function of time and population. If we assign $g(t, y)$ to a variable, we call it r.

df/dt is the first derivative of f, and it maps from t to a rate. If we assign $df/dt(t)$ to a variable, we call it r.

It is easy to get df/dt confused with g, but notice that they are not even the same type. g is more general: it can compute the rate of change for any (hypothetical) population at any time; df/dt is more specific: it is the actual rate of change at time t, given that the population is $f(t)$.

4 ode45

A limitation of Euler's method is that the time step is constant from one iteration to the next. But some parts of the solution are harder to estimate than others; if the time step is small enough to get the hard parts right, it is doing more work than necessary on the easy parts. The ideal solution is to adjust the time step as you go along. Methods that do that are called **adaptive**, and one of the best adaptive methods is the Dormand-Prince pair of Runge-Kutta formulas. You don't have to know what that means, because the nice people at Mathworks have implemented it in a function called `ode45`. The `ode` stands for "ordinary differential equation [solver];" the 45 indicates that it uses a combination of 4th and 5th order formulas.

In order to use `ode45`, you have to write a MATLAB function that evaluates g as a function of t and y.

As an example, suppose that the rate of population growth for rats depends on the current population and the availability of food, which varies over the course of the year. The governing equation might be something like

$$\frac{df}{dt}(t) = af(t)\,[1 + \sin(\omega t)]$$

where t is time in days and $f(t)$ is the population at time t.

a and ω are **parameters**. A parameter is a value that quantifies a physical aspect of the scenario being modeled.

In this example, a characterizes the reproductive rate, and ω is the frequency of a periodic function that describes the effect of varying food supply on reproduction.

This equation specifies a relationship between a function and its derivative. In order to estimate values of f numerically, we have to transform it into a rate function.

The first step is to introduce a variable, y, as another name for $f(t)$

$$\frac{df}{dt}(t) = ay\,[1 + \sin(\omega t)]$$

This equation means that if we are given t and y, we can compute $df/dt(t)$, which is the rate of change of f. The next step is to express that computation as a function called g:

$$g(t, y) = ay\,[1 + \sin(\omega t)]$$

Writing the function this way is useful because we can use it with Euler's method or `ode45` to estimate values of f. All we have to do is write a MATLAB function that evaluates g. Here's what that looks like using the values $a = 0.01$ and $\omega = 2\pi/365$ (one cycle per year):

```
function res = rats(t, y)
    a = 0.01;
    omega = 2 * pi / 365;
    res = a * y * (1 + sin(omega * t));
end
```

You can test this function from the Command Window by calling it with different values of `t` and `y`; the result is the rate of change (in units of rats per day):

```
>> r = rats(0, 2)

r = 0.0200
```

So if there are two rats on January 1, we expect them to reproduce at a rate that would produce 2 more rats per hundred days. But if we come back in April, the rate has almost doubled:

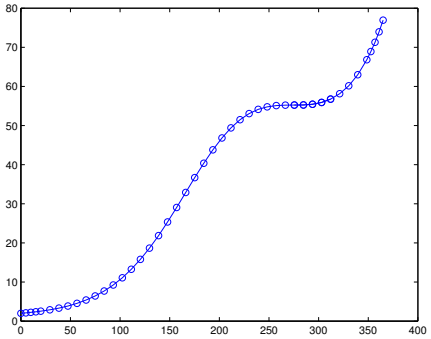
```
>> r = rats(120, 2)

r = 0.0376
```

Since the rate is constantly changing, it is not easy to predict the future rat population, but that is exactly what `ode45` does. Here's how you would use it:

```
>> ode45(@rats, [0, 365], 2)
```

The first argument is a handle for the function that computes g. The second argument is the interval we are interested in, one year. The third argument is the initial population, $f(0) = 2$.

When you call `ode45` without assigning the result to a variable, MATLAB displays the result in a figure:



The x-axis shows time from 0 to 365 days; the y-axis shows the rat population, which starts at 2 and grows to almost 80. The rate of growth is slow in the winter and summer, and faster in the spring and fall, but it also accelerates as the population grows.

5 Multiple output variables

`ode45` is one of many MATLAB functions that return more than one output variable. The syntax for calling it and saving the results is

```
>> [T, Y] = ode45(@rats, [0, 365], 2);
```

The first return value is assigned to `T`; the second is assigned to `Y`. Each element of `T` is a time, t, where `ode45` estimated the population; each element of `Y` is an estimate of $f(t)$.

If you assign the output values to variables, `ode45` doesn't draw the figure; you have to do it yourself:

```
>> plot(T, Y, 'bo-')
```

If you plot the elements of `T`, you'll see that the space between the points is not quite even. They are closer together at the beginning of the interval and farther apart at the end.

To see the population at the end of the year, you can display the last element from each vector:

```
>> [T(end), Y(end)]

ans = 365.0000    76.9530
```

`end` is a special word in MATLAB; when it appears as an index, it means "the index of the last element." You can use it in an expression, so `Y(end-1)` is the second-to-last element of `Y`.

How much does the final population change if you double the initial population? How much does it change if you double the interval to two years? How much does it change if you double the value of a?

6 Analytic or numerical?

When you solve an ODE analytically, the result is a function, f, that allows you to compute the population, $f(t)$, for any value of t. When you solve an ODE numerically, you get two vectors. You can think of these vectors as a discrete approximation of the continuous function f: "discrete" because it is only defined for certain values of t, and "approximate" because each value F_i is only an estimate of the true value $f(t)$.

So those are the limitations of numerical solutions. The primary advantage is that you can compute numerical solutions to ODEs that don't have analytic solutions, which is the vast majority of nonlinear ODEs.

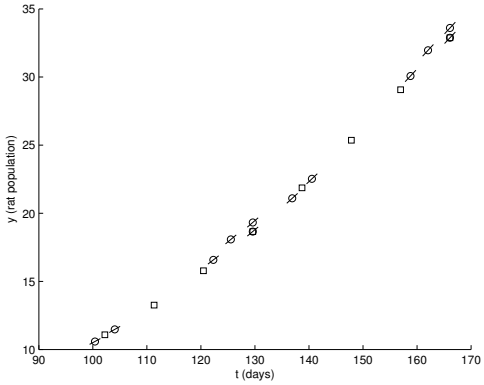
If you are curious to know more about how `ode45` works, you can modify `rats` to display the points, (t, y), where `ode45` evaluates g. Here is a simple version:

```
function res = rats(t, y)
    plot(t, y, 'bo')
    a = 0.01;
    omega = 2 * pi / 365;
    res = a * y * (1 + sin(omega * t));
end
```

Each time `rats` is called, it plots one data point; in order to see all of the data points, you have to use `hold on`.

```
>> clf; hold on
>> [T, Y] = ode45(@rats, [0, 10], 2);
```

This figure shows part of the output, zoomed in on the range from Day 100 to 170:



The circles show the points where `ode45` called `rats`. The lines through the circles show the slope (rate of change) calculated at each point. The rectangles show the locations of the estimates (T_i, F_i). Notice that `ode45` typically evaluates g several times for each estimate. This allows it to improve the estimates, for one thing, but also to detect places where the errors are increasing so it can decrease the time step (or the other way around).

7 What can go wrong?

Don't forget the `@` on the function handle. If you leave it out, MATLAB treats the first argument as a function call, and calls `rats` without providing arguments.

```
>> ode45(rats, [0,365], 2)
??? Input argument "y" is undefined.

Error in ==> rats at 4
    res = a * y * (1 + sin(omega * t));
```

Again, the error message is confusing, because it looks like the problem is in `rats`. You've been warned!

Also, remember that the function you write will be called by `ode45`, which means it has to have the signature `ode45` expects: it should take two input variables, `t` and `y`, in that order, and return one output variable, `r`.

If you are working with a rate function like this:

$$g(t, y) = ay$$

You might be tempted to write this:

```
function res = rate_func(y)          % WRONG
    a = 0.1
    res = a * y
end
```

But that would be wrong. So very wrong. Why? Because when `ode45` calls `rate_func`, it provides two arguments. If you only take one input variable, you'll get an error. So you have to write a function that takes `t` as an input variable, even if you don't use it.

```
function res = rate_func(t, y)       % RIGHT
    a = 0.1
    res = a * y
end
```

Another common error is to write a function that doesn't make an assignment to the output variable. If you write something like this:

```
function res = rats(t, y)
    a = 0.01;
    omega = 2 * pi / 365;
    r = a * y * (1 + sin(omega * t))     % WRONG
end
```

And then call it from `ode45`, you get

```
>> ode45(@rats, [0,365], 2)
??? Output argument "res" (and maybe others) not assigned during
call to "/home/downey/rats.m (rats)".

Error in ==> rats at 2
    a = 0.01;

Error in ==> funfun/private/odearguments at 110
f0 = feval(ode,t0,y0,args{:});   % ODE15I sets args{1} to yp0.

Error in ==> ode45 at 173
[neq, tspan, ntspan, next, t0, tfinal, tdir, y0, f0, odeArgs,
odeFcn, ...
```

This might be a scary message, but if you read the first line and ignore the rest, you'll get the idea.

Yet another mistake that people make with `ode45` is leaving out the brackets on the second argument. In that case, MATLAB thinks there are four arguments, and you get

```
>> ode45(@rats, 0, 365, 2)
??? Error using ==> funfun/private/odearguments
When the first argument to ode45 is a function handle, the
tspan argument must have at least two elements.

Error in ==> ode45 at 173
[neq, tspan, ntspan, next, t0, tfinal, tdir, y0, f0, odeArgs,
odeFcn, ...
```

Again, if you read the first line, you should be able to figure out the problem (`tspan` stands for "time span", which we have been calling the interval).

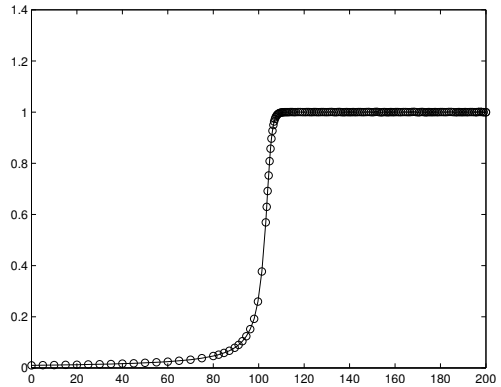
8 Stiffness

There is yet another problem you might encounter, but if it makes you feel better, it might not be your fault: the problem you are trying to solve might be **stiff**[*].

I won't give a technical explanation of stiffness here, except to say that for some problems (over some intervals with some initial conditions) the time step needed to control the error is very small, which means that the computation takes a long time. Here's one example:

$$\frac{df}{dt} = f^2 - f^3$$

If you solve this ODE with the initial condition $f(0) = \delta$ over the interval from 0 to $2/\delta$, with $\delta = 0.01$, you should see something like this:



After the transition from 0 to 1, the time step is very small and the computation goes slowly. For smaller values of δ, the situation is even worse.

In this case, the problem is easy to fix: instead of `ode45` you can use `ode23s`, an ODE solver that tends to perform well on stiff problems (that's what the "s" stands for).

In general, if you find that `ode45` is taking a long time, you might want to try one of the stiff solvers. It won't always solve the problem, but if the problem is stiffness, the improvement can be striking.

Exercise 8.1. Write a rate function for this ODE and use `ode45` to solve it with the given initial condition and interval. Start with $\delta = 0.1$ and decrease it by multiples of 10. If you get tired of waiting for a computation to complete, you can press the Stop button in the Figure window or press Control-C in the Command Window.

Now replace `ode45` with `ode23s` and try again!

9 Celestial mechanics

Modeling celestial mechanics is a good opportunity to compute with spatial vectors. Imagine a star with mass m_1 at a point in space described by the vector P_1, and a planet with mass m_2 at point P_2. The magnitude of the gravitational force[†] between them is

$$f_g = G\frac{m_1 m_2}{r^2}$$

where r is the distance between them and G is the universal gravitational constant, which is about $6.67 \times 10^{-11} Nm^2/kg^2$. Remember that this is the appropriate value of G only if the masses are in kilograms, distances in meters, and forces in Newtons.

[*]The following discussion is based partly on an article from Mathworks available at `http://www.mathworks.com/company/newsletters/news_notes/clevescorner/may03_cleve.html`

[†]See `http://en.wikipedia.org/wiki/Gravity`

The direction of the force on the star at P_1 is in the direction toward P_2. We can compute relative direction by subtracting vectors; if we compute `R = P2 - P1`, then the direction of `R` is from `P1` to `P2`.

The distance between the planet and star is the length of R:

```
r = norm(R)
```

The direction of the force on the star is $\hat{R}$:

```
rhat = R / r
```

Exercise 9.1. Write a sequence of MATLAB statements that computes `F12`, a vector that represents the force on the star due to the planet, and `F21`, the force on the planet due to the star.

Exercise 9.2. Encapsulate these statements in a function named `gravity_force_func` that takes `P1`, `m1`, `P2`, and `m2` as input variables and returns `F12`.

Exercise 9.3. Write a simulation of the orbit of Jupiter around the Sun. The mass of the Sun is about 2.0×10^{30} kg. You can get the mass of Jupiter, its distance from the Sun and orbital velocity from `http://en.wikipedia.org/wiki/Jupiter`. Confirm that it takes about 4332 days for Jupiter to orbit the Sun.

10 Animation

Animation is a useful tool for checking the results of a physical model. If something is wrong, animation can make it obvious. There are two ways to do animation in MATLAB. One is to use `getframe` to capture a series of images and `movie` to play them back. The more informal way is to draw a series of plots. Here is an example I wrote for Exercise 9.3:

```
function animate_func(T,M)
    % animate the positions of the planets, assuming that the
    % columns of M are x1, y1, x2, y2.
    X1 = M(:,1);
    Y1 = M(:,2);
    X2 = M(:,3);
    Y2 = M(:,4);

    minmax = [min([X1;X2]), max([X1;X2]), min([Y1;Y2]), max([Y1;Y2])];

    for i=1:length(T)
        clf;
        axis(minmax);
        hold on;
        draw_func(X1(i), Y1(i), X2(i), Y2(i));
        drawnow;
    end
end
```

The input variables are the output from `ode45`, a vector `T` and a matrix `M`. The columns of `M` are the positions and velocities of the Sun and Jupiter, so `X1` and `Y1` get the coordinates of the Sun; `X2` and `Y2` get the coordinates of Jupiter.

`minmax` is a vector of four elements which is used inside the loop to set the axes of the figure. This is necessary because otherwise MATLAB scales the figure each time through the loop, so the axes keep changing, which makes the animation hard to watch.

Each time through the loop, `animate_func` uses `clf` to clear the figure and `axis` to reset the axes. `hold on` makes it possible to put more than one plot onto the same axes (otherwise MATLAB clears the figure each time you call `plot`).

Each time through the loop, we have to call `drawnow` so that MATLAB actually displays each plot. Otherwise it waits until you finish drawing all the figures and *then* updates the display.

`draw_func` is the function that actually makes the plot:

```
function draw_func(x1, y1, x2, y2)
    plot(x1, y1, 'r.', 'MarkerSize', 50);
    plot(x2, y2, 'b.', 'MarkerSize', 20);
end
```

The input variables are the position of the Sun and Jupiter. `draw_func` uses `plot` to draw the Sun as a large red marker and Jupiter as a smaller blue one.

Exercise 10.1. To make sure you understand how `animate_func` works, try commenting out some of the lines to see what happens.

One limitation of this kind of animation is that the speed of the animation depends on how fast your computer can generate the plots. Since the results from `ode45` are usually not equally spaced in time, your animation might slow down where `ode45` takes small time steps and speed up where the time step is larger.

There are two ways to fix this problem:

1. When you call `ode45` you can give it a vector of points in time where it should generate estimates. Here is an example:

   ```
   end_time = 1000;
   step = end_time/200;
   [T, M] = ode45(@rate_func, [0:step:end_time], W);
   ```

   The second argument is a range vector that goes from 0 to 1000 with a step size determined by `step`. Since `step` is `end_time/200`, there will be about 200 rows in `T` and `M` (201 to be precise).

   This option does not affect the accuracy of the results; `ode45` still uses variable time steps to generate the estimates, but then it interpolates them before returning the results.

2. You can use `pause` to play the animation in real time. After drawing each frame and calling `drawnow`, you can compute the time until the next frame and use `pause` to wait:

   ```
   dt = T(i+1) - T(i);
   pause(dt);
   ```

   A limitation of this method is that it ignores the time required to draw the figure, so it tends to run slow, especially if the figure is complex or the time step is small.

Exercise 10.2. Use `animate_func` and `draw_func` to vizualize your simulation of Jupiter. Modify it so it shows one day of simulated time in 0.001 seconds of real time—one revolution should take about 4.3 seconds.

11 Conservation of Energy

A useful way to check the accuracy of an ODE solver is to see whether it conserves energy. For planetary motion, it turns out that `ode45` does not.

The kinetic energy of a moving body is $mv^2/2$; the kinetic energy of a solar system is the total kinetic energy of the planets and sun. The potential energy of a sun with mass m_1 and a planet with mass m_2 and a distance r between them is

$$U = -G\frac{m_1 m_2}{r}$$

Exercise 11.1. Write a function called `energy_func` that takes the output of your Jupiter simulation, `T` and `M`, and computes the total energy (kinetic and potential) of the system for each estimated position and velocity. Plot the result as a function of time and confirm that it decreases over the course of the simulation. Your function should also compute the relative change in energy, the difference between the energy at the beginning and end, as a percentage of the starting energy.

You can reduce the rate of energy loss by decreasing `ode45`'s tolerance option using `odeset`:

```
options = odeset('RelTol', 1e-5);
[T, M] = ode45(@rate_func, [0:step:end_time], W, options);
```

The name of the option is `RelTol` for "relative tolerance." The default value is `1e-3` or 0.001. Smaller values make `ode45` less "tolerant," so it does more work to make the errors smaller.

Exercise 11.2. Run `ode45` with a range of values for `RelTol` and confirm that as the tolerance gets smaller, the rate of energy loss decreases.

Exercise 11.3. Run your simulation with one of the other ODE solvers MATLAB provides and see if any of them conserve energy.