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?

9.6 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.

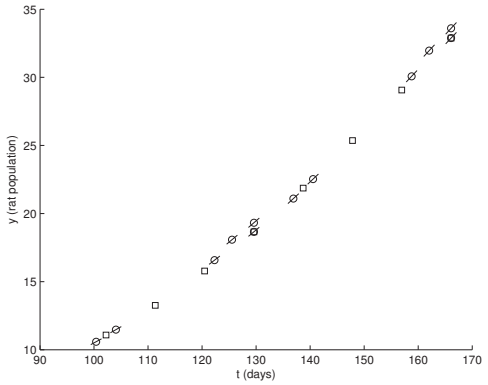
If you are curious to know more about how `ode45` works, you can modify `rats` to display the points, (t, y), where `ode45` evaluates g. Here is a simple version:

```
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).

9.7 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:

$$g : t, y \rightarrow ay$$

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).

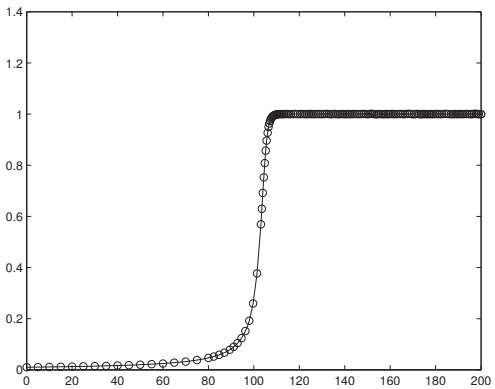
9.8 Stiffness

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:

$$f_t = f^2 - f^3$$

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.

[†]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`