Estimating Error, ODE23

Software that implements modern numerical methods has two features that aren't present in codes like ODE4 and classical Runge-Kutta. The methods in the software can estimate error and provide automatic step size control. You don't specify the step size h. You specify an accuracy you want. And the method estimate the errors as they go along and adjust the step size accordingly. And they provide a fully accurate continuous interpolant. They don't just provide the solution at the discrete set of points. They provide a function that defines the solution everywhere in the interval. And so you can plot it, find zeroes of the function, provide a facility called event handling, and so on.

Let's see the ODE23. The basic method is order three And the error estimate is based on the difference between the order three method and then the underlying order two method. There are four slopes involved.

$$s_1 = f(t_n, y_n) \qquad (\text{First Same AS Last})$$

$$s_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2} s_1)$$

$$s_3 = f(t_n + \frac{3}{4}h, y_n + \frac{3}{4}h s_2)$$

$$t_{n+1} = t_n + h$$

$$y_{n+1} = y_n + \frac{h}{9}(2s_1 + 3s_2 + 4s_3)$$

$$s_4 = f(t_{n+1}, y_{n+1})$$

$$e_{n+1} = \frac{h}{72}(-5s_1 + 6s_2 + 8s_3 - 9s_4)$$

The first one is the value of the function at the start of the interval. But that is based on something called FSAL, where that slope is most likely left over from the previous step. If the previous step was successful, this function value is the same as the last function value from the previous step. The second slope is used to step into the middle of the interval, function is evaluated there. The third slope is used to step $3/4$ of the way across the interval and a third slope obtained there. Then these three values are used to take the step. y_{n+1} is a linear combination of these three function values. Then the function is evaluated to get a fourth slope at the end of the interval. And then, these four slopes are used to estimate the error. The error estimate here is the difference between y_{n+1} and another estimate of the solution that is obtained from a second order method that we don't actually evaluate. We just need the difference between that method and y_{n+1} to estimate the error. This estimated error is compared with a user-supplied tolerance. If the estimated error is less than a tolerance, then the step is successful. And the fourth slope s_4 becomes the s_1 of the next step. If the answer is bigger than the tolerance, then the error could be the basis for adjusting the step size. In either case, the error estimate is the basis for adjusting the step size for the next step. This is the Bogacki-Shampine Order 3(2) Method which is the basis for ODE 23.

Lets look at some very simple uses of ODE 23 just to get started.

We are going to take the differential equation.

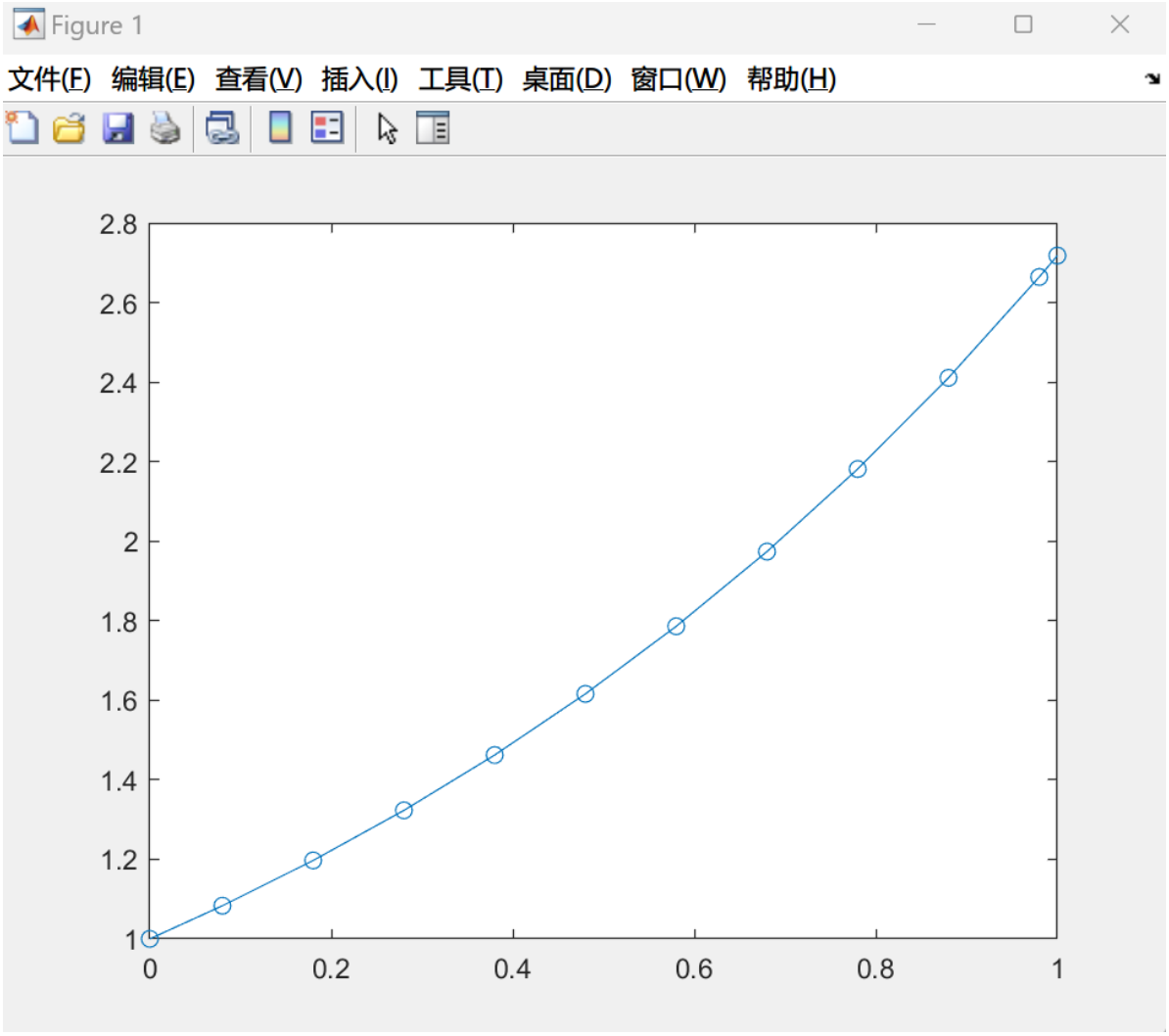
$$y' = y$$

```
>> F=@(t,y) y
```

And just call ODE 23 on the interval from 0 to 1 with initial value 1.

```
>> ode23(F,[0,1],1)
```

No output arguments. If we call it ODE23, it just plots the solution.



Here it is. It just produces a plot. It picks a step size, goes from 0 to 1 and here it gets the final value of e --2.7..

If we do supply output arguments.

```
>> [t,y]=ode23(F,[0,1],1)
```

It comes back with the values of t and y. ODE 23 picks the values of t it wants. This is a trivial problem. It ends up picking a step size of 0.1. After it gets started, it chooses can initial step size of 0.08 for whatever error tolerances. And the final value of y is 2.7182, which is the values of e. So these are the two simple uses of ODE23. If you don't supply any output arguments, it draws a graph. If you do supply output arguments, t and y, it comes back with the values of t and y choosing the values of t to meet the error. The default error tolerances is 10^{-3}.

Now let's try something a little more challenging to see the automatic error-controlled step size choice in action.

```
>> a = 0.25
```

```
>> y0 = 15.9
```

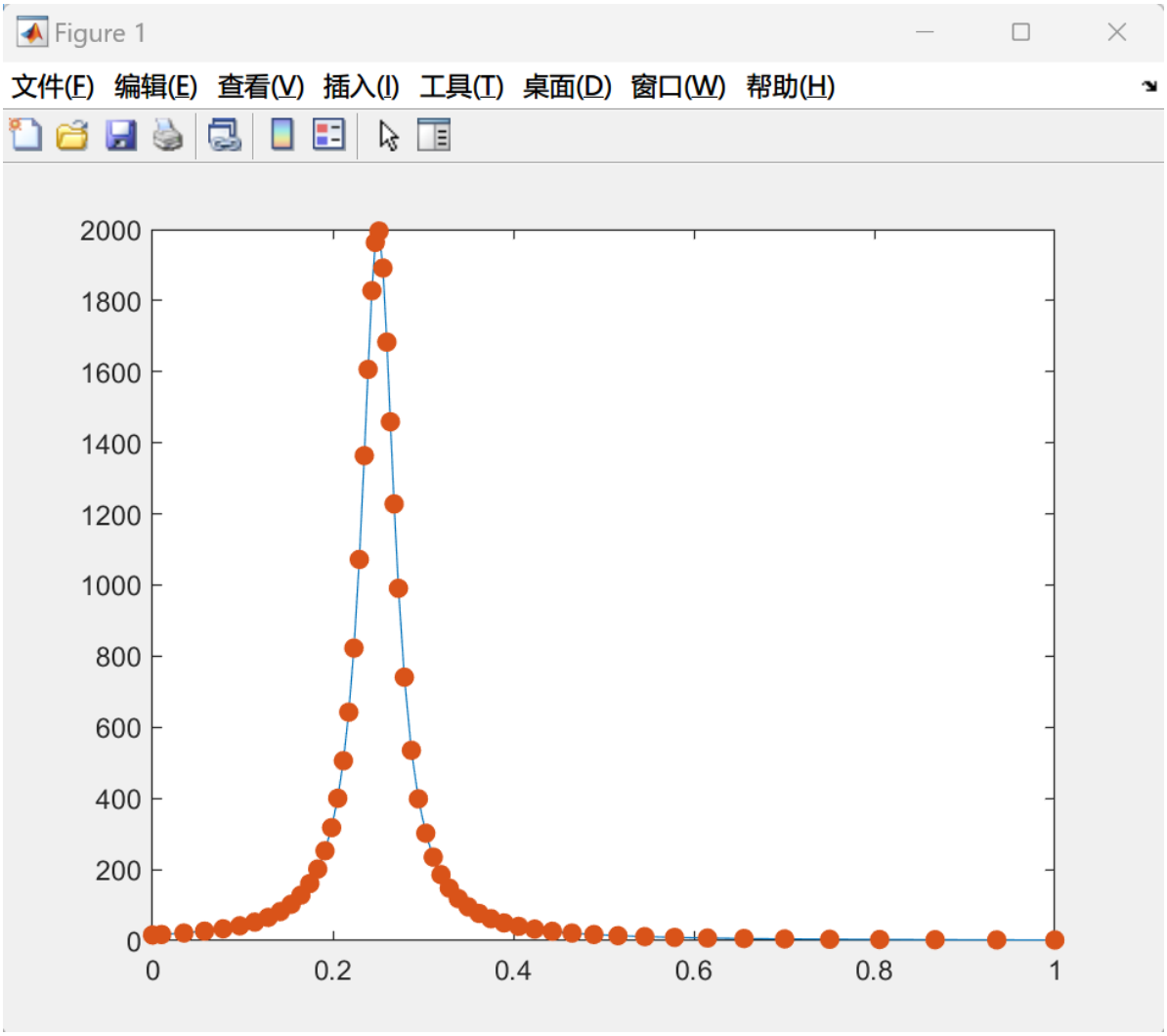
Now the differential equation is $y' = 2(a-t)y^2$

```
>> F = @(t,y) 2*(a-t)*y^2
```

We are going to integrate this with the ODE 23 on the interval from 0 to 1 staring at y0, and saving the results in t and y, and then plotting them

```
>> [t,y] = ode23(F,[0,1],y0);
```

```
plot(t,y,'-',t,y,'.','markersize',24)
```



So there is a near singularity. It nearly blows up. And then it settles back down. So the points are bunched together as you go up to the singularity and come back down, but then get farther apart as the solution settles down. And the ODE solver is able to take bigger steps.

To see what steps were actually taken, let's compute the difference of t, and then plot that.

```
>> h = diff(t);
   plot(t(2:end),h,'.','markersize',24)
```

And we see that a small step size was taken near the almost singularity at that 0.25. And then as we get towards the end of the interval, a larger step size is taken. And then, finally, the step size just to reach the end of the interval is taken as the last step.

So that's the automatic step size choice of ODE23.

BS23 has a nice natural interpolant that goes along with it that's actually been known for over 100 years. It's called **Hermite Cubic Interpolation**. We know that two points determine a straight line.