Using Numerical ODE's to Approximate and AND Gate

Numerical Ordinary Differential Equations

Jonathon D'Arcy [s1607860]

December 7, 2021

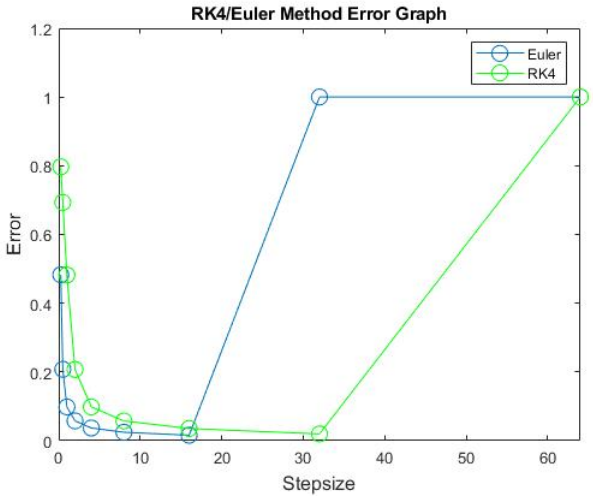

Figure 1: RK4 and Euler

Immediately we have numerous notable features of the plot most dramatically an atypical correlation with increased step size and decreased error, and sudden increase of error at about a step size of 15. These are both caused by the process we are using called gradient descent. This process is a way to navigate closer to a local minimum of a function by taking steps proportional to the negative gradient of the current position of the function. In this example 600 steps are being taken to move across the sigmoid function in order to create an approximation of the AND Logic gate. While this is an effective approach to the given problem it does require a starting step size which will affect all subsequent step sizes, this is where these phenomena originates from.

The first phenomena mentioned was an atypical correlation with increased step size and decreased error, so it makes sense to address this first. What is essentially happening here is that for small time steps the descent is not moving far enough to meaningfully approach the target, as for each time we move towards the target we are going to make an even smaller step, and thus a likely a small change in accuracy. This idea makes sense as imagine if we took a step size of 0, then we would never move and the output would for every step be the same as the initial guess, so by approaching zero we should be approaching our starting guess.

The second feature we can now explain is the sudden increase in the norm error to one. To do this it is important to observe the outputs of at least one of our two methods to get an idea of what is happening, below

are the outputs of the Euler method for each step size:

$$Stepsizes = \begin{bmatrix} 0.25 \\ 0.5 \\ 1 \\ 2 \\ 4 \\ 8 \\ 16 \\ 32 \\ 64 \end{bmatrix}, \quad Euler = \begin{bmatrix} 0.024257 & 0.28995 & 0.10687 & 0.63055 \\ 0.00366 & 0.13077 & 0.02645 & 0.84138 \\ 0.00161 & 0.06164 & 0.01087 & 0.92444 \\ 0.00094 & 0.03601 & 0.00612 & 0.95556 \\ 0.0006 & 0.02303 & 0.00384 & 0.97144 \\ 0.00044 & 0.01505 & 0.00269 & 0.98089 \\ 0.0004 & 0.00932 & 0.00216 & 0.98738 \\ 4.722e^{-9} & 9.3010e^{-9} & 5.4370e^{-9} & 1.3175e^{-8} \\ 2.0945e^{-12} & 1.12394e^{-11} & 2.55883e^{-12} & 1.79974e^{-11} \end{bmatrix}$$

Here we can see in the beginning the output is doing what we now expect approaching the AND gates true output as we choose larger and large step sizes. However suddenly somewhere between the step sizes of 16 and 32 we suddenly have a very fast approach to zero in all four columns, seemingly the method has broken down here. The explanation for this is the opposite problem to the first phenomena our steps are too large. This causes us to overshoot our target and thus creating large error by trying to get back. The other problem for this is not only is it passing the target constantly, to hit it and be caught by it, i.e. sufficiently close enough for the weights to not allow the next step to jump away, we must hit the target more directly as the step size is much more likely to overpower the weights. It is because of this last fact however that these large jumps can actually be the most accurate, because when you do have a step size that catches it must be very close to the target and the larger the initial step size the more accurate it must be in a catch. This can be seen if we increase the number of vectors we output up to 90 evenly distributed in the range 0 to 50 (Figure 2).

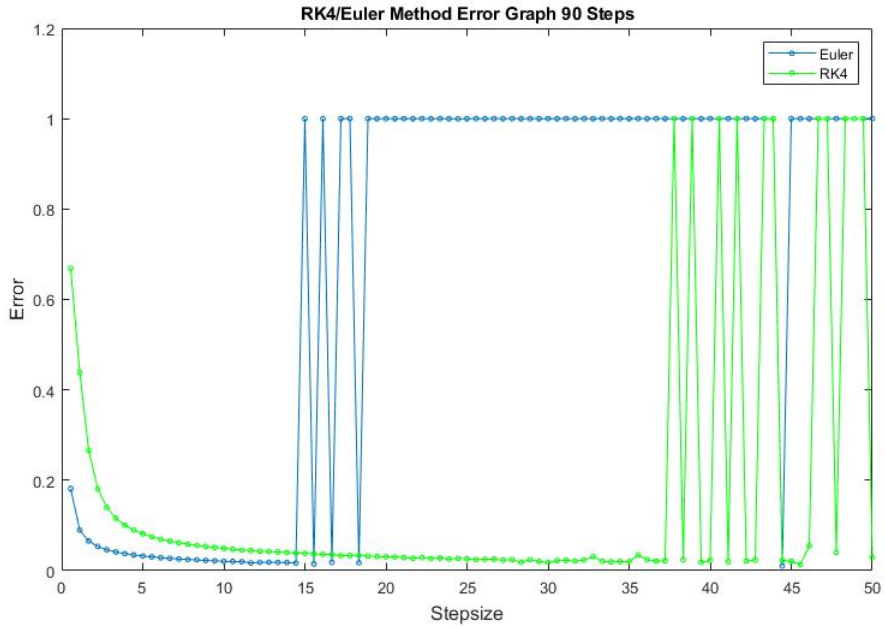

Figure 2: RK4 and Euler Increased Step Size

Here we see the same trends we noticed before but there is a very interesting point in the Euler method at the initial step size of $44.\overline{4}$. Here we see an output which drops from the line of one errors to suddenly being the most accurate point we have found so far with an error of 0.0104. This appears to be an example of a large step size point that managed to land in its catching zone.

So this leads us to the question of best choice in step size. At first glance an obvious thought is towards our earlier stated minimum error at step size $44.\overline{4}$. This however as a choice has a few major draw backs. Firstly, to find this point we have to test many points as we don't necessarily know where points like these will lie, so the

computation behind finding this sort of point is actually quite large as we just have to test a lot of points in the hopes of finding one, making it a bit of a needle in a haystack. Secondly, this is not replicable, by this I mean if we change the random seed of our weights and run the Euler method again at this point it may no longer hit its catching zone, making this point not only not a minimum error point but it will have among the highest error we've seen. So we can see that while we get some low error with these high step sizes they are going to be too computationally difficult to find, and if we were to have that extra computation it would be better to just up our iteration count and choose a smaller step size. So if we want to find a choose a best step size that is replicable in result and still gives a decently low error output we want to be as close to the last step size before we have a norm of 1 as possible, while still being a far enough distance to have it be unlikely that we will be engulfed by the overstepping if we vary our starting seed for this reason I would choose a step size of approximately 12. This wont give us the lowest error every time but it will give a very good one every time. Similarly for the RK4 method I would use the same strategy choosing a safe, and replicable step size of around $33.\bar{4}$ with each methods choice giving in this scenario an error of 0.0189 and 0.02 respectively. So in recap the optimal (in a given range) for each seed will appear late in the step sizes but will be very hard to find, but these aren't the best because they depend on the starting position of the weights. [1]

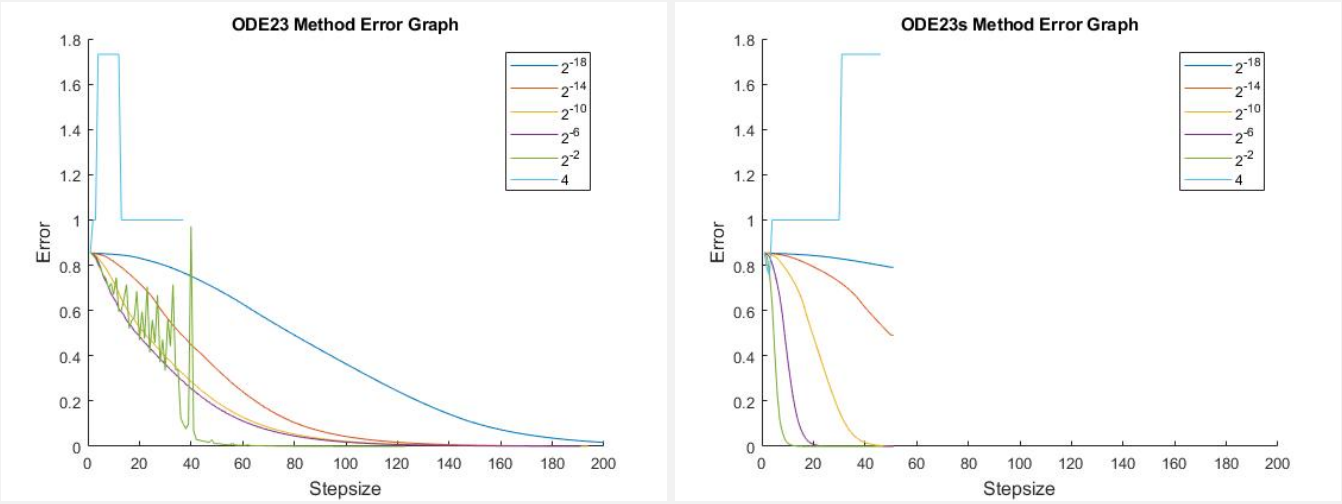

Figure 3 and 4

Here we begin looking at the Shampine-Bogacki methods (ode23 and ode23s) and how it approximates the AND gate. These graph outputs are a bit different and need to be explained, here we have a computational budget and can use our steps to try and approach it, once we use our budget however we have to stop of method. That is why we see some trajectories with sudden stopping points. There are some similarities to our previous graphs however that we can see like the explosive error and the best approximations being close to the high error. These approximations are both better than the RK4 and Euler variants seen before as the convergence on the target is much faster as we can see from observing the steps size of 0.8 in these method to the corresponding area in Figure 2. One of the major problems with ones of these methods is the non-monotinicity property in ODE23 which means that as we vary weight we dont know if we actually choosing a sharp increase or decrease if we dont keep the outputs of earlier iterations. We note however that this is not in ODE23s and so it appears to give the lowest replicable error which is an error of $1.38818 * 10^{-10}$ at with a starting step size of 0.25.

[1] Question to the Marker: I've got three questions on this that I can't find the answer to but seem logical to me that I was wondering if you could answer. One, for these large catching points are there infinitely many of them but they just become less likely? I believe so but only have conjecture. Secondly, could there not be points which perfectly hit the target and thus force themselves to stop? I again believe there are some but can't find them. Finally are there also inverse catching points which will give error greater than one, these I also can't locate but think they exist

Matlab Code

```
%%%%%AND_RK4%%%%%%%5
function [weights, output] = AND_RK4(h,iters)
%
%   h= input stepsize
%   iters = number of steps
%
global AND
AND_Init();
weights = AND.initialWeights;
%
for i=1:iters
    wdot = AND_NegGradient((i-1)*h,weights);
    weights = RK4step((i-1)*h, weights, h, @AND_NegGradient);
end
output = AND.out;
end
%%%%%%%%MainScript%%%%%%%%
global AND
h_vect=[];
Err_vect=[];
outputs_vect=[];
Targ_out=[0;0;0;1];
Time=zeros(1,90);
for i=1:90
    tic;
    h=50*i/90;
    h_vect=horzcat(h_vect,h);
    iters=600;
    [weights,outputs]=AND_Euler(h,iters);
    Time(i)=toc;
    outputs_vect=horzcat(outputs_vect,outputs);
    Err=norm(outputs-Targ_out);
    Err_vect=horzcat(Err_vect,Err);
end
figure(1)
clf(1);
figure(1)
hold on
plot(h_vect,Err_vect,'-o','MarkerSize',10)

LowErr=1000;
LowErrPlace=-1;
LowErrTime=1;
for j=1:length(Err_vect)
        if Err_vect(j)<LowErr
            LowErr=Err_vect(j);
            LowErrPlace=j;
            LowErrTime=Time(j);
        end
end
LowErr
LowErrPlace
LowErrTime
outputs_vect
```

```
h_vect_2=[];
Err_vect_2=[];
Targ_out_2=[0;0;0;1];
outputs_vect_2=[];
Time_2=zeros(1,9);
for i_2=1:9
    tic;
    h_2=2^(i_2-3);
    h_vect_2=horzcat(h_vect_2,h_2);
    iters_2=150;
    [weights_2,outputs_2]=AND_RK4(h_2,iters_2);
    Time_2(i_2)=toc;
    outputs_vect_2=horzcat(outputs_vect_2,outputs_2);
    Err_2=norm(outputs_2-Targ_out_2);
    Err_vect_2=horzcat(Err_vect_2,Err_2);
end
plot(h_vect_2,Err_vect_2,'-o','MarkerSize',10,'Color', 'g')
ylabel('Error','FontSize',12)
xlabel('Stepsize','FontSize',12)
title('RK4/Euler Method Error Graph')
legend('Euler','RK4')
axis([0 64 0 1.2])
hold off

outputs_vect_2
Time_2

figure(2)
clf(2);
h=[2^(-18); 2^(-14); 2^(-10); 2^(-6); 2^(-2); 2^2];
Targ_out=[0;0;0;1];
Errors_vect=[];
N_vect=[];
T_vect=[];
Weights_vect=[];
Outputs_vect=[];
figure(2);
hold on
for i=1:6
    tic;
    iters=600;
    [N, T, Weights, Outputs, Errors]=AND_ode23(h(i),iters);
    Time(i)=toc;
    N_vect=horzcat(N_vect,N);
    T_vect=vertcat(T_vect,T);
    Weights_vect=vertcat(Weights_vect,Weights);
    Outputs_vect=vertcat(Outputs_vect,Outputs);
    Errors_vect=vertcat(Errors_vect,Errors);
    plot(Errors,'-','MarkerSize',10)
end
ylabel('Error','FontSize',12)
xlabel('Stepsize','FontSize',12)
title('ODE23 Method Error Graph')
legend('2^{-18}','2^{-14}','2^{-10}','2^{-6}','2^{-2}','4')
hold off
```

```
figure(3)
clf(3);
h=[2^(-18); 2^(-14); 2^(-10); 2^(-6); 2^(-2); 2^2];
Targ_out=[0;0;0;1];
Errors_vect=[];
N_vect=[];
T_vect=[];
Weights_vect=[];
Outputs_vect=[];
figure(3);
hold on
for i=1:6
    tic;
    iters=600;
    [N, T, Weights, Outputs, Errors]=AND_ode23s(h(i),iters);
    Time(i)=toc;
    N_vect=horzcat(N_vect,N);
    T_vect=vertcat(T_vect,T);
    Weights_vect=vertcat(Weights_vect,Weights);
    Outputs_vect=vertcat(Outputs_vect,Outputs);
    Errors_vect=vertcat(Errors_vect,Errors);
    plot(Errors,'-','MarkerSize',10)
end
ylabel('Error','FontSize',12)
xlabel('Stepsize','FontSize',12)
title('ODE23s Method Error Graph')
legend('2^{-18}','2^{-14}','2^{-10}','2^{-6}','2^{-2}','4')
axis([0 200 0 1.8])
hold off
```