Initial Value Problems for Time-Dependent Differential Equations

Sarah Ellwein, Kate Davis, Olivia Hartnett, Connor Leipelt

Winter 2022

1 Introduction

This project focuses on solving initial value problems (IVP) for time-dependent ordinary differential equations (ODE). These problems take the form:

$$u'(t) = f(u(t), t),\ t > t_0$$

with the initial data $u(t_0) = \eta$ (where $t_0 = 0$ is often assumed). The above equation may represent a system of ODEs:

$$u'(t) = \begin{cases} f_1(u, t) \\ ... \\ f_n(u, t) \end{cases}$$

We solve ODEs numerically by computing the approximate $u'(t)$ for each forward time step, or in other words discretizing the problem. In this project, we will go over Euler's Method, Heun's Method, and the Runge-Kutta Method.

2 Euler's Method

2.1 Derivation

We want to discretize our problem given the following information:

$$\begin{cases} u'(t) = f(u, t),\ t > t_0 \\ u(t_0) = \eta \end{cases}$$

Euler's method is derived from the difference quotient

$$u' \approx \frac{\Delta u}{\Delta t}$$

We want to equally space our points into n time steps, so have $\Delta t = k$ and each time step represented as $t_j = j(k)$ where $j = 1, ..., n$. Then $\Delta u = u_{j+1} - uj$ where $u_j = u(t_j)$. It follows that

$$u' \approx \frac{u_{j+1} - u_j}{k} \Rightarrow u_{j+1} - u_j \approx ku' \Rightarrow u_{j+1} - u_j \approx kf(u_j, t_j) \Rightarrow u_{j+1} \approx u_j + kf(u_j, t_j)$$

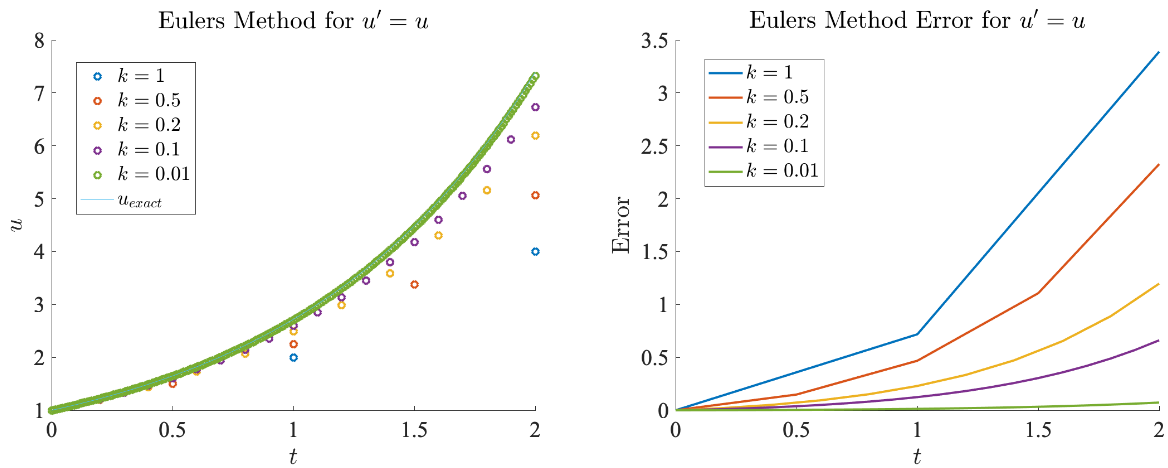
$$\boxed{u_{j+1} \approx u_j + kf(u_j, t_j)}$$

2.2 Example

Example: Let $u' = u$ and $u(0) = 1$. We can analytically solve this ODE using separation of variables.

$$u' = u \Rightarrow \frac{du}{dt} = u \Rightarrow \frac{1}{u}du = dt \Rightarrow \int \frac{1}{u}du = \int dt \Rightarrow \ln u + c_1 = t + c_2 \Rightarrow u = e^{t+c_3} = Ce^t$$

After solving for C, we get the solution $u(t) = e^t$. We can also solve this numerically by Euler's method using MATLAB for $k = 1, 0.5, 0.2, 0.1, 0.01$. Notice that the approximate points becomes closer to the exact function $u(t)$ as k decreases.



The left graph above shows the discretized points for each k and the right graph shows the error between the exact and approximate values of u as t increases a step. You may notice from the right figure that Euler's method becomes more accurate as k decreases. Unfortunately, there is still quite a bit of error as t increases even with low k values.

Euler's method is fairly straightforward and easy to understand; however, it comes at a cost of accuracy. We first discuss Euler's method not to argue for it's efficiency, but rather set the foundation for it's successors we're about to explore.

3 Heun's Method

3.1 Derivation

We see that Euler's method fails to account for the predicted slope for each successive u_{k+1}. Heun's method (or updated Euler's method) succeeds in averaging the slope of the first point and slope of the predicted point.

To obtain a more accurate numerical approximation to the unique solution of the IVP listed above, Heun's method applies two steps instead of one: a "prediction step" and a "correction step." In the prediction step, we use Euler's method to compute a rough approximation of the solution u_{j+1}. Then, in the correction step we use Euler's method again to compute the slope at u_{j+1} as $f(u_{j+1}, t_{j+1})$ and take the average of the slopes $f(u_j, t_j)$ and $f(u_{j+1}, t_{j+1})$ to compute the slope where we find u_{j+1}.

$$\begin{cases} a_j = f(u_j, t_j) \\ b_j = f(u_{j+1}, t_{j+1}) \\ u_{j+1} = u_j + \frac{1}{2}(a_j + b_j)k \end{cases}$$

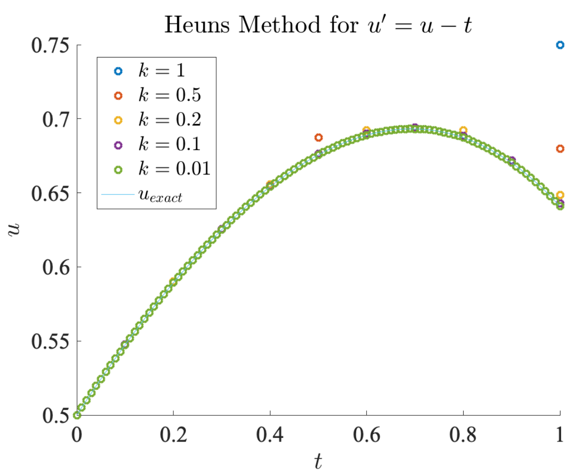
3.2 Example

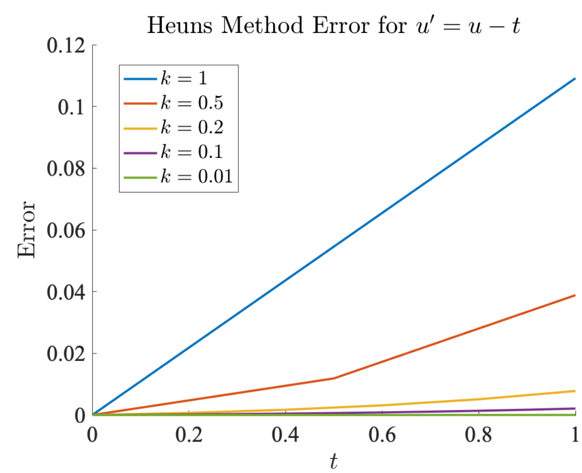
Example: Let $u' = u - t$ with initial value $u(0) = 1/2$. Here's an example of the first iteration of Heun's Method where $k = 0.1$.

Apply Heun's method to compute an approximation to the solution of the above IVP at t=1.
Taking $f(U^n) = u - t$, we have $U^* = U^n + 0.1(U_n - t_n)$ and $U^{n+1} = U^n + 0.05(U^n - t^n + U^* - t_{n+1})$
Then, $U^{n+1} = U^n + 0.05[U^n - t^n + [U^n + 0.1(U_n - t_n)] - t_{n+1}]$
$U^{n+1} = U^n + 0.05(2.1U^n - 1.1t_n - t_{n+1})$, for n=0,1,...,9
By applying the above equation, we get the results displayed in the table below.

n	x_n	y_n	Exact solution values	Error
1	0.1	0.5475	0.5474	0.00085
2	0.2	0.5895	0.5893	0.000189
3	0.3	0.6254	0.6251	0.000313
4	0.4	0.6545	0.6541	0.000461
5	0.5	0.6763	0.6756	0.000637
6	0.6	0.6898	0.6889	0.000845
7	0.7	0.6942	0.6931	0.00109
8	0.8	0.6886	0.6872	0.00138
9	0.9	0.6719	0.6702	0.00171
10	1.0	0.6430	0.6409	0.00210

We now compare the results for each $k = 1, 0.5, 0.2, 0.1, 0.01$ to the exact solution $u(t) = t + 1 - \frac{e^t}{2}$.





Heun's method clearly succeeds in accuracy compared to its predecessor, Euler's method. However, like Euler's method, Heun's method paved way for its successor following the same concept.

4 Runge-Kutta Method

4.1 Derivation

Using a concept from Heun's method, the Runge-Kutta method accounts for not only the slope of its current and iterated point, but also some points in between. Doing so gives a far more accurate result as k decreases.

4.2 Example

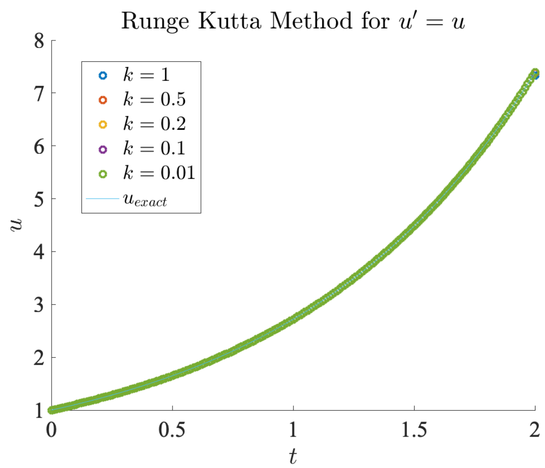
We'll use our first example $u' = u$ with initial value $u(0) = 1$. Here's an example of the first iteration of the Runge Kutta Method.

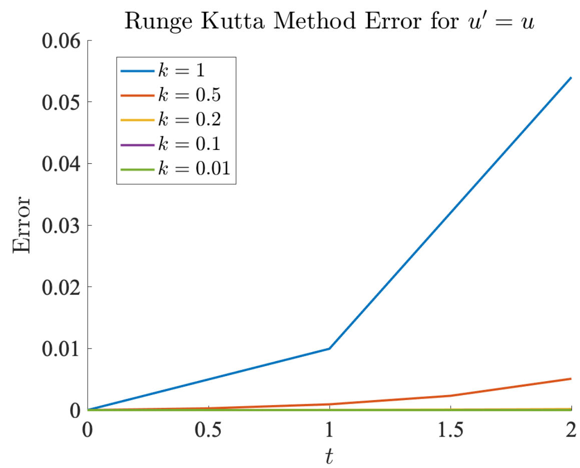
Given the ordinary differential equation, $u' = u$, $u(0) = 1$, and $\Delta t = 0.25$, we can use the 4-stage Runge-Kutta method:
We take four slopes.
A "half step" is used for better precision.

When we take the average of the 6 slopes, there is more emphasis on the middle two slopes (hence the middle two slopes are multiplied by two.)

$$k_1 = f(u(0), 0) = f(1, 0) = 1 \tag{1}$$
$$u_1(0 + \frac{0.25}{2}) = u_1(0.125) = u(0) + k_1 * \frac{\Delta t}{2} = 1 + 1 * \frac{0.25}{2} = 1.125 \tag{2}$$
$$k_2 = f(u_1(0.125), 0.125) = f(1.125, 0.125) = 1.125 \tag{3}$$
$$u_2(0 + \frac{0.25}{2}) = u_2(0.125) = u(0) + k_2 * \frac{\Delta t}{2} = 1 + 1.25 * \frac{0.25}{2} = 1.15625 \tag{4}$$
$$k_3 = f(u_2(0.125), 0.125) = f(1.15625, 0.125) = 1.15625 \tag{5}$$
$$u_3(0 + 0.25) = u_3(0.25) = u(0) + k_3 * \Delta t = 1 + 1.15625 * 0.25 = 1.2890625 \tag{6}$$
$$k_4 = f(u_3(0.25), 0.25) = f(1.2890625, 0.25) = 1.2890625 \tag{7}$$
$$m = \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} = 1.141927083 \tag{8}$$
$$u_1 = u_0 + km = 1 + 0.25(1.141927083) = 1.28548177075$$

Similar to Euler's Method, we solve this numerically for $k = 1, 0.5, 0.2, 0.1, 0.01$. We see that the results is a vast improvement compared to Euler's Method.





5 Higher-Order Example

5.1 Using Euler's Method for Higher-Order

It is possible to solve higher-order differential equations using Euler's Method. An example of this would be to use Euler's Method for solving a second order differential equation.
When solving a second order differential equation we are given $x'' = f(t, x, x')$; $x(t_0) = x_0$; $x'(t_0) = x'_0$; and h which is our change in t.
When doing this, it is common to replace x' with u and x'' with u'.
So in final we have,
$u' = f(t, x, u)$; $x(t_0) = x_0$, $u(t_0) = u_0$; h

From this we can use Euler's method to compute x_1 and u_1.
We know that $x_1 = x_0 + h \cdot u_0$
We are already given u_0 so we can instantly compute x_1.
To compute x_2 we need to compute u_1, since $x_2 = x_1 + h(u_1)$.
To compute u_1 we do the same process that we use for solving for x_1 but instead of using u_0, our slope at

x_0, we will use u'_0 which is the slope at u_0.
From above we use the originally given equation(since $x'' = u'$):
$u'_0 = f(t_0, x_0, u_0)$
From this we get u'_0 and solve for u_1 using:
$u_1 = u_0 + h(u'_0)$
After computing u_1, we now quickly get t_1 from adding h to t_0.
$t_1 = t_0 + h$
Now, we have computed t_1,x_1, and u_1. Since we have all three of these along with a given function to find u'_1, we can repeat this action to find more and more points as t increases.
This method can be used for nth ordered differential equations as long as we are given:
$x^{(n)} = f(t, x, x', ..., x^{(n-1)}$; $x(t_0) = x_0$; $x'(t_0) = x'_0$;...; $x^{(n-1)}(t_0) = x_0^{(n-1)}$

MATLAB Codes Used

Euler's Method

```
function [t, u] = forward_euler(f, ti, ui, k, tf)
% FORWARD_EULER Discretize a first order differential equation using
% Forward Euler's Method
%   INPUT: function f(u,t) = u'(t), initial step ti, initial value ui, time
%   step size k, final time step tf
%
%   OUTPUT: a vector of time steps [ti ; ... ; tf] and a
%   vector of values [ui ; ... ; uf]

% Initialize vector outputs and populate with initial values
t(1) = ti; u(1) = ui;

% Initial u value
uj = ui; tj = ti; j = 0;

% Iterate all time steps and populate time and value vectors
while tj < tf
    uj = uj + k*f(tj,uj);
    tj = ti + (j+1)*k;
    t(j+2) = tj;
    u(j+2) = uj;
    j = j + 1;
end
end
```

Heun's Method

```
function [t, u] = heun(f, ti, ui, k, tf)
% HEUN Discretize a first order differential equation using
% Heun's Method
%   INPUT: function f(u,t) = u'(t), initial step ti, initial value ui, time
%   step size k, final time step tf
%
%   OUTPUT: a vector of time steps [ti ; ... ; tf] and a
%   vector of values [ui ; ... ; uf]

% Initialize vector outputs and populate with initial values
t(1) = ti; u(1) = ui;

% Initial u value
uj = ui; tj = ti; j = 0;

% Iterate all time steps and populate time and value vectors
while tj < tf
    aj = f(tj,uj);
    bj = f(tj+k, uj+k*aj);
    uj = uj + k*(aj+bj)/2;
    tj = ti + (j+1)*k;
    t(j+2) = tj;
    u(j+2) = uj;
    j = j + 1;
end
end
```

Runge Kutta Method

```
function [t, u] = rungekutta(f, ti, ui, k, tf)
% RUNGEKUTTA Discretize a first order differential equation using
% Runge Kutta Method
%   INPUT: function f(u,t) = u'(t), initial step ti, initial value ui, time
```

```
%   step size k, final time step tf
%
%   OUTPUT: a vector of time steps [ti ; ... ; tf] and a
%   vector of values [ui ; ... ; uf]

% Initialize vector outputs and populate with initial values
t(1) = ti; u(1) = ui;

% Initial u value
uj = ui; tj = ti; j = 0;

% Iterate all time steps and populate time and value vectors
while tj < tf
    y1 = f(tj,uj);
    y2 = f(tj+k/2, uj+k*y1/2);
    y3 = f(tj+k/2, uj+k*y2/2);
    y4 = f(tj+k, uj+k*y3);
    uj = uj + k*(y1+2*y2+2*y3+y4)/6;
    tj = ti + (j+1)*k;
    t(j+2) = tj;
    u(j+2) = uj;
    j = j + 1;
end
end
```