

MTH3199 Applied Math for Engineers – Fall 2024

Assignment 3: Introduction to Numerical Integration

Assigned: Friday, October 4, 2024.

Lab Report Due: Thursday, October 17, 2024 (11:59 PM EST).

Online Resources

- Forward Euler: [Wikipedia](#)
- Explicit midpoint method: [Wikipedia](#)

Overview

In today's activity, we will begin to look at a few different [numerical integration techniques](#). Our goal is to find an approximate solution to the following initial value problem (IVP):

$$\dot{X} = f(t, X), \quad X(t_{start}) = X_0 \quad (1)$$

In this differential equation, the independent variable, t , is a scalar, and the independent variable, X , can either be a scalar or a vector. The function $f(t, X)$ describes the rate at which X changes over time. The solution to the IVP is a function of time, $X(t)$, that satisfies the IC's, $X(t_{start}) = X_0$, and the differential equation, $\dot{X} = f(t, X)$.

Broadly speaking, there are two categories of numerical methods for solving ODE's:

- **Explicit methods** directly compute the system state at the next time step, X_{n+1} , as a function of the the current state, X_n .
- **Implicit methods** set up a (possibly nonlinear) set of equations that constrain the state at the next time step, X_{n+1} . These equations must then be solved (often with Newton's method) to find X_{n+1} .

To get a feel for how numerical integration works, we will start by implementing and testing two basic [explicit](#) methods: [forward Euler](#) and [explicit midpoint](#):

- Forward Euler approximates the value of X_{n+1} at the next time step by using the tangent line approximation.

$$X(t+h) \approx X(t) + h\dot{X} \rightarrow X_{n+1} = X_n + hf(t_n, X_n) \quad (2)$$

where h is the length of the time step.

- The explicit midpoint method uses a forward Euler step to approximate the value of $X_{n+.5}$ a half-time step away (i.e. the **midpoint** between X_n and X_{n+1}). It then evaluates the slope at this midpoint to approximate X_{n+1} at the next time step.

$$X_{n+.5} = X_n + \frac{h}{2}f(t_n, X_n) \quad (3)$$

$$X_{n+1} = X_n + hf(t_n + \frac{h}{2}, X_{n+.5}) \quad (4)$$

where h is the length of the time step. It should be noted that this method is different than just computing two forward Euler steps using a time step of $\frac{h}{2}$.

One last thing before we get into the implementation section: using computers to solve differential equations is an entire field of applied mathematics. There's a whole zoo of integration algorithms out there, and the content presented in this module is just the tip of the iceberg.

Instructions

Day 10 (Friday, October 4th) activity. Please complete before class on Tuesday, October 8th.

Before you begin, you are encouraged to form groups of two or three. Groups will jointly submit a single set of deliverables on Canvas (and will be graded as a single unit). Once you have found teammates (or if you are choosing to work alone), add yourself (and your teammates if you have them) to the same **assignment 03** group on Canvas.

Test Functions

To test the performance of our numerical integration scheme, we'll consider two initial value problems (IVP's) that have known analytical/closed-form solutions. The first IVP describes a first order system driven by a sinusoid:

$$\dot{x} = -5x + 5 \cos(t) - \sin(t), \quad x(0) = 1 \quad (5)$$

The solution to this IVP is $x(t) = \cos(t)$. I have provided an implementation for both the rate function, $f(t, x)$, and the corresponding solution, $x(t)$, below:

```
function dXdt = rate_func01(t,X)
    dXdt = -5*X + 5*cos(t) - sin(t);
end

function X = solution01(t)
    X = cos(t);
end
```

The second IVP describes an undamped second order system:

$$\dot{X} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} X, \quad X(0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad (6)$$

and has the solution:

$$X(t) = \begin{bmatrix} \cos(t) \\ \sin(t) \end{bmatrix} \quad (7)$$

I have provided an implementation for both the rate function, $f(t, X)$, and solution, $X(t)$, below:

```
function dXdt = rate_func02(t,X)
    dXdt = [0, -1; 1, 0]*X;
end

function X = solution02(t)
    X = [cos(t); sin(t)];
end
```

Implementing Forward Euler

Your first task is to implement forward Euler:

$$X_{n+1} = X_n + hf(t_n, X_n) \quad (8)$$

We'll do this in two steps. First, write a MATLAB function that computes X_{n+1} for the next time step, given the rate function $f(t, X)$ the current values of t_n and X_n , and the size of the time step, h . For the purpose of performance experiments later in this activity, this function should also return the number of times it called the rate function, $f(t, X)$. I have provided a template below:

```
%This function computes the value of X at the next time step
%using the Forward Euler approximation
%INPUTS:
%rate_func_in: the function used to compute dXdt. rate_func_in will
```

```

%           have the form: dXdt = rate_func_in(t,X)   (t is before X)
%t: the value of time at the current step
%XA: the value of X(t)
%h: the time increment for a single step i.e. delta_t = t_{n+1} - t_{n}
%OUTPUTS:
%XB: the approximate value for X(t+h) (the next step)
%   formula depends on the integration method used
%num_evals: A count of the number of times that you called
%           rate_func_in when computing the next step
function [XB,num_evals] = forward_euler_step(rate_func_in,t,XA,h)
    %your code here
end

```

Next, use your single step function write a MATLAB function that runs the numerical integration over some time interval, $t \in [t_0, t_f]$, given the rate function $f(t, X)$, the initial condition X_0 , and some target step size, h_{ref} . Note that, since we would like the step sizes to be equal, it won't usually be possible to set the step size to be exactly h_{ref} . Instead, just find the smallest **number of steps**, N , such that the step size $h = \frac{t_f - t_0}{N}$ is still than h_{ref} . This function should keep track of the number of times the rate function was called.

```

%Runs numerical integration using forward Euler approximation
%INPUTS:
%rate_func_in: the function used to compute dXdt. rate_func_in will
%           have the form: dXdt = rate_func_in(t,X)   (t is before X)
%tspan: a two element vector [t_start,t_end] that denotes the integration endpoints
%X0: the vector describing the initial conditions, X(t_start)
%h_ref: the desired value of the average step size (not the actual value)
%OUTPUTS:
%t_list: the vector of times, [t_start;t_1;t_2;...;t_end] that X is approximated at
%X_list: the vector of X, [X0';X1';X2';...;(X_end)'] at each time step
%h_avg: the average step size
%num_evals: total number of calls made to rate_func_in during the integration
function [t_list,X_list,h_avg, num_evals] = ...
    forward_euler_fixed_step_integration(rate_func_in,tspan,X0,h_ref)
    %your code here
end

```

To verify that your implementation is working, use it to solve the IVP described by equation 5. Plot the closed-form solution for $X(t)$ and a few numerical approximations (with different time steps) on the same axes.

Explicit Midpoint Method

Your next task is to implement the explicit midpoint method:

$$X_{n+.5} = X_n + \frac{h}{2} f(t_n, X_n) \quad (9)$$

$$X_{n+1} = X_n + h f(t_n + \frac{h}{2}, X_{n+.5}) \quad (10)$$

As with forward Euler, we'll do this in two steps. First, write a MATLAB function that computes X_{n+1} for the next time step, given the rate function $f(t, X)$ the current values of t_n and X_n , and the size of the time step, h . For the purpose of performance experiments later in this activity, this function should also return the number of times it called the rate function, $f(t, X)$. I have provided a template below:

```

%This function computes the value of X at the next time step
%using the explicit midpoint approximation
%INPUTS:
%rate_func_in: the function used to compute dXdt. rate_func_in will
%           have the form: dXdt = rate_func_in(t,X)   (t is before X)
%t: the value of time at the current step
%XA: the value of X(t)

```

```

: the time increment for a single step i.e.  $\Delta t = t_{n+1} - t_n$ 
%OUTPUTS:
%XB: the approximate value for  $X(t+h)$  (the next step)
%    formula depends on the integration method used
%num_evals: A count of the number of times that you called
%           rate_func_in when computing the next step
function [XB,num_evals] = explicit_midpoint_step(rate_func_in,t,XA,h)
    %your code here
end

```

Next, use your single step function write a MATLAB function that runs the numerical integration over some time interval, $t \in [t_0, t_f]$, given the rate function $f(t, X)$, the initial condition X_0 , and some target step size, h_{ref} . Note that, since we would like the step sizes to be equal, it won't usually be possible to set the step size to be exactly h_{ref} . Instead, just find the smallest **number of steps**, N , such that the step size $h = \frac{t_f - t_0}{N}$ is still than h_{ref} . This function should keep track of the number of times the rate function was called.

```

%Runs numerical integration using explicit midpoint approximation
%INPUTS:
%rate_func_in: the function used to compute  $dX/dt$ . rate_func_in will
%             have the form:  $dX/dt = \text{rate\_func\_in}(t,X)$  ( $t$  is before  $X$ )
%tspan: a two element vector  $[t_{start}, t_{end}]$  that denotes the integration endpoints
%X0: the vector describing the initial conditions,  $X(t_{start})$ 
%h_ref: the desired value of the average step size (not the actual value)
%OUTPUTS:
%t_list: the vector of times,  $[t_{start}; t_1; t_2; \dots; t_{end}]$  that  $X$  is approximated at
%X_list: the vector of  $X$ ,  $[X_0'; X_1'; X_2'; \dots; (X_{end})']$  at each time step
%h_avg: the average step size
%num_evals: total number of calls made to rate_func_in during the integration
function [t_list,X_list,h_avg, num_evals] = ...
    explicit_midpoint_fixed_step_integration(rate_func_in,tspan,X0,h_ref)
    %your code here
end

```

To verify that your implementation is working, use it to solve the IVP described by equation 5. Plot the closed-form solution for $X(t)$ and a few numerical approximations (with different time steps) on the same axes.

Experiment: Local Truncation Error

Truncation error is the error that originates from the discrete nature of the numerical integration method (the number of time steps are finite). This is different from **round-off error** which is a consequence of computers not being able to represent real numbers with infinite precision. There are two types of truncation error:

- **Local:** the error for a single time step.
- **Global:** the error across the entire integration interval.

Decreasing the size of the time step will cause both of these truncation errors to shrink. Unfortunately, smaller time steps means more steps, which means more computation time. The rate at which the truncation error, ϵ scales with the time step, h varies from method to method. We can describe this relationship using **big O notation**:

$$\epsilon = O(h^p) \quad (11)$$

Big O notation is used to describe the asymptotic behavior of a function. In this case, as $h \rightarrow 0$, the error, ϵ is become proportional to h^p :

$$\epsilon \approx kh^p \quad (12)$$

The proportionality constant, k , is not relevant to big O notation, all that matters is that ϵ and h^p are proportional to one another when h is sufficiently small.

Your task is to run a set of experiments to measure how the local truncation error scales with the size of the time step (in other words, find p). It should be noted that the theoretical value of p is a whole number. Let $X_{n+1} =$

$G(t, X_n, h)$ be the function that computes next value of X_{n+1} . If $X(t)$ is the analytical solution to the IVP, then the local truncation error is given by:

$$\epsilon = |G(t, X(t), h) - X(t + h)| \quad (13)$$

Note that $|X|$ is the norm of X when X is a vector. Choose one of the test functions and some reference time t_{ref} . For each algorithm, compute the local truncation error at $t = t_{ref}$ for different values of h (I'd like to see around 100 data points ranging from 10^{-5} to 10^1 that are distributed evenly in log-space). Fit a line to this data in log space to estimate the value of p . I have uploaded a MATLAB function to Canvas (called log-log regression) to compute the fit for you. On the same axes, plot the data you gathered for both algorithms, as well as the corresponding fit lines (this should be a log-log plot). What were the estimated values of p that you computed for each algorithm?

Part of the reason why the local truncation error is shrinking is that both $X(t + h)$ and $G(t, X(t), h)$ are approaching $X(t)$ as the step size, h , gets smaller and smaller. To verify that the local truncation error is shrinking in a meaningful way (and not just for the aforementioned reason), plot the difference $|X(t + h) - X(t)|$ as a function of h (for the same values of h that you already used) on the same axes, as well as a fit line.

Run the same experiment using the other test function. You do not need to generate additional plots. However, please include the new estimates of p in your lab report. Based on your results, does the local truncation error shrink in a meaningful way (does it shrink faster than $|X(t + h) - X(t)|$)? Does it shrink faster for the forward Euler step or the explicit midpoint step?

Experiment: Global Truncation Error

Now, let's see how the global truncation error scales with the size of the time step, h . If we use our numerical integration scheme to approximate X over the interval $t \in [t_0, t_f]$ with initial condition $X(t_0) = X_0$, and $X(t)$ is the closed-form solution, then the global truncation error is given by:

$$\epsilon = |X_f - X(t_f)| \quad (14)$$

where X_f is the numerical approximation of $X(t)$ after the last time step. Choose one of the test functions and some time interval $t \in [t_0, t_f]$. For each algorithm, compute the global truncation error for different values of h (I would like to see values that are evenly spaced across several orders of magnitude). Fit a line to this data in log space to estimate the value of p . On the same axes, plot the data you gathered for both algorithms, as well as the corresponding fit lines (this should be a log-log plot). What were the estimated values of p that you computed for each algorithm?

Your implementations should keep track of how times the rate function $f(t, X)$ was called. How does the global truncation error scale with the number of rate function calls (i.e. what are the corresponding values of p)? On a separate figure, generate a plot comparing this scaling for the two algorithms (this plot should include both the experimental data and their fit lines).

Run the same set of experiments using the other test function. You do not need to generate additional plots. However, in your lab report, please include the new estimates of p indicating how the global truncation error scales with both the step size, and the number of rate function calls.

Deliverables and Submission Guidelines

This exercise is the first part of the larger numerical integration assignment. The lab report for the numerical integration assignment will be due on Thursday, October 17th at 11:59 PM EST. In this lab report, make sure to document how you went about implementing the forward Euler and the explicit midpoint method. As a reminder, I want to see the following plots and tables:

- Plots comparing the closed-form solution to equation 5 with the numerical approximation (for a few different time step sizes). Each algorithm should get its own plot.
- A table that shows how the **local** truncation error scales with step size (the values of p). This table should have entries for each algorithm/test function combination.

- A log-log plot that compares the **local** truncation error scaling for the two algorithms. This plot should also include the difference $|X(t+h) - X(t)|$ for comparison. Step sizes should be across a range of magnitudes. Please include both the data points and fit lines. You only need to generate a plot for a single test function.
- A table that shows how the **global** truncation error scales with step size (the values of p). This table should have entries for each algorithm/test function combination.
- A short discussion on which local truncation error scaling is preferable (forward Euler or explicit midpoint).
- A log-log plot that compares the **global** truncation error scaling for the two algorithms. Step sizes should be across a range of magnitudes. Please include both the data points and fit lines. You only need to generate a plot for a single test function.
- A table that shows how the **global** truncation error scales with the **number of rate function calls** (the values of p). This table should have entries for each algorithm/test function combination.
- A log-log plot that compares the **global** truncation error scaling (with the **number of rate function calls**) for the two algorithms. Please include both the data points and fit lines. You only need to generate a plot for a single test function.
- A short discussion on which algorithm you would prefer to use for integrating these test functions and why.