Epode: Reference Manual

Martin Jay McKee
martinjaymckee@gmail.com

May 21, 2018

Contents

Part I

Just the Basics

Chapter 1

Getting Started

1.1 Overview

Epode is a library for numeric integration of first-order systems of ordinary differential equations for the solution of inital value problems. That is, given a system of equations, an integration range, and an intial value of the form,

$$\dot{\vec{x}} = f(t, \vec{x}), t = [t_0, t_1], \vec{x}_0 = ... \tag{1.1}$$

Epode can calculate an estimate of the final value of the system variables $\vec{x}$. Some problems of this type can be solved analytically. The vast majority of this type are, however, analytically intractible and, as such require some form of numeric solution. This is where an ode solver like **Epode** comes in.

1.1.1 What's with the name anyway?

Epode is a C++14 compatible library for integration of ordinary differential equations (odes). The connection is simple, an ode is a lyric poem written to address a particular subject. This library was written to address the need for a clean, easy to use and efficient library for the solution of ordinary differental equations. The final link (as simply naming the library "ode" would be too boring) is that the word epode is virtually synonomous to ode and refers to a specific form of lyric poem typified by couplets composed of lines with, typically, varying length (not that that matters in any way to this library). The Epode library is an ode to the new features of C++ 11/14 which make such a library much easier to create and use. It doesn't hurt that it's a short name and contains the letters 'o', 'd', 'e', either!

1.2 Installation

Because of its header-only nature, there is no specific need to install **Epode**. One has the option to treat it as a local library and to simply include the files

in the project tree of a new application. This has the advantage that **Epode** version differences can easily be ignored. The two main disadvantage of this is that, first, it may necessitate writing code to different versions of the library and, second, as a result of keeping to the newest version of the library, bugfixes that are applied to the mainline code will not be available to an application. The suggestion is, then, to use keep a current version of **Epode** and use it for all development. While this may require API updates in code at the moment (the **Epode** API is not yet stable), it guarantees that the most effective code is available at all times.

1.3 A Boring Example

Let's take a look at what an **Epode** program might look like. It's important to realize that this example is not a particularly useful one. It is designed to demonstrate the simplified API – using the solve function. Additionally, the system of equations was not designed to solve any specific problem but, rather, to demonstrate a system of moderate complexity. A complete program may look like the following:

```
#include <iostream>
using namespace std;
#include <Epode/ode.h>
using namespace epode;
using solver_t = epode::integrator::Euler<double, 3>;
using state_t = typename solver_t::state_t;
auto f = [](auto v, auto y){
  return {
    y(0) + v*y(1),
    -y(1),
    y(0) * y(2)
  }
} -> state_t;
auto y0 = state_t{1, 2, 3};
auto solver = Solver(0.01);
int main(void) {
  auto results = solver(f, 0, {0.5, 1, 1.5, 2}, y0);
  for(auto result: results) {
    cout << 't = ' << result.t << '\ty = ' << result.y << '\n';
  }
  cout << "\tIteration Steps = " << results.back().stats.steps << "\n";
  cout << "\tFunction Evals = " << results.back().stats.evals << "\n";
}
```

Listing 1: This is how to create and run a solver, then print the results

Chapter 2

User API

TALK ABOUT THE PARTS OF THE LIBRARY... INTRODUCE THE SECTIONS

2.1 The "solve" Function

The function `solve(...)` is syntactic sugar for a number of steps to develop an Epode solver and run it. Still, for simple uses of **Epode**, it should be fully sufficient.

2.2 Integrator Engine

The integrator engine is the object that contains the implementation of the method "stepping", as well as storage of the integration results.

2.3 Methods

Epode provides a number of integration methods, all the way from the 1^{st} order Euler's method up to the ?? method with adaptive stepping. These methods can be selected explicitly by the user either when using the basic "solve" function or when constructing an integration flow directly.

2.4 Trigger Objects

In the **Epode** library, Several functions of the integration are controlled by - so called - trigger objects.

IF THE "END" VALUE(S) (IS/ARE) BEFORE THE "START" VALUE, CONFIGURE THE TRIGGER OBJECT, LIMITER AND INTEGRATOR TO RUN BACKWARDS IN TIME.

Part II

The Library in Depth

Chapter 3

Solver Methods

3.1 Solver API

3.1.1 End/Save Syntax

There are multiple ways to denote the points where a solver either ends integration or saves the calculated values.

3.1.2 Save Transformers

ADD A WAY TO "TRANSFORM" THE STATE BEFORE IT IS OUTPUT. THIS SHOULD ALLOW FOR CHANGING THE SIZE OF THE STATE AS WELL AS DOING SOME NUMBER OF OPERATIONS ON THE STATE BEFORE SAVING IT.

SYNTAX – transform(stats, y_n, $\dot{v}_n$, v_n, y_{n-1})

3.1.3 Results Type

IT WOULD BE NICE TO MAKE IT POSSIBLE FOR THE RESULTS PACKAGE TO BE OF FIXED SIZE (I.E. NOT A STD::VECTOR) SO THAT DYNAMIC ALLOCATIONS CAN BE AVOIDED). THIS WILL REQUIRE DEFINING THE RETURN TYPE OF THE INTEGRATOR OPERATOR () BASED ON THE TYPE OF THE STORE TRIGGER. ONCE THAT WORKS, AS

Syntax	**End Trigger**	**Store Trigger**	**Limit Trigger**
$\langle num \rangle$	$v \geq num$	Always	$v = num$
$\{\langle num \rangle +\}$	$v \geq num_{last}$	Each num	Each num
step::uniform($\langle num \rangle$, $\langle N \rangle$)	$v \geq num$	N calculated points	N calculated points
step::until($\langle func \rangle$)	$func(\dots) = true$	Always	None
step::by_until($\langle step \rangle$, $\langle func \rangle$)	$func(\dots) = true$	Every $v_0 + n * step$	Every $v_0 + n * step$
trigger::custom($\langle end \rangle$, $\langle store \rangle$, ($\langle limit \rangle$))	$end(\dots()) = true$	$store(\dots) = true$	$limit(\dots) = true$

LONG AS THE RETURN TYPES ALL HAVE A SIMILAR API, IT COULD BE A SINGLE VALUE, ARRAY OR VECTOR.

3.2 Explicit Methods

The explicit methods currently implemented are generally of the Runge-Kutta type. For an overview of these types of methods read [1]. The Runge-Kutta methods are arranged roughly in increasing order. As such, later methods are – in general – more accurate than those listed earlier. For complex problems, this increased accuracy *may* or may not translate into a faster method as it is impossible to make general statements about the computational expense of a problem/method combination without further analysis. The default method of the "solve" function, however, is currently the BS32 method, pending verification of the correctness of the RKF45 method, or something similar. This method typically provides good accuracy while also providing reasonably fast computation.

3.2.1 Forward Euler

The Forward Euler method of integration is by far the simplest. It uses only the current state and the derivitive of the state at the current point to extrapolate the state at the next point. While this makes the method simple to implement and easy to understand, it does lead to an integration method which cannot be both accurate and fast. If the timestep is made so small that some level of accuracy is attained, the runtime becomes prohibitive. In any case, the low order of Euler's method makes truly accurate calculations impossible in any case. The butcher's tableau for Forward Euler is:

0	0
	1

Forward Euler is provided mostly because it can be used as an exceptionally quick to run comparison for a more usable method.

Available with: **include** ⟨*Euler*⟩

3.2.2 RKF1(2)

RKF 1(2) is a first-order accurate, adaptive step-size Runge-Kutta method, introduced by Fehlberg in [2]. The method requires an amortized three function evaluations per step with a butcher's tableau of:

0			
1/2	1/2		
1	1/256	255/256	
	1/256	255/256	0
	1/512	255/256	1/512

Epode implements the adaptive loop by removing the first evaluation from the loop, as it does not depend upon the timestep.

Available with: **include** ⟨*RKF*⟩

3.2.3 Generic 2nd-Order Runge-Kutta

There are a number of parametrically related second-order Runge-Kutta methods. This fact is used to create a generic second-order method. The Butcher's tableau of this generic method is:

$$\begin{array}{c|cc} 0 & & \\ \eta & \eta & \\ \hline & 1 - 1/2\eta & 1/2\eta \end{array}$$

There is a single parameter η which ranges between zero and one. Any of these parameter values represents a valid second-order method. Some of the specific methods have previously been defined and are included below as subclasses of the generic method. At construction of the method, the parameter η is required to be passed to the constructor. This parameter can simply follow the initial step-size parameter in the solver constructor.

Available with: **include** ⟨*RK2*⟩

3.2.4 Heun

Heun's method is a specialization of the generic second-order Runge-Kutta method.

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \end{array}$$

Available with: **include** ⟨*RK2*⟩

3.2.5 Midpoint

The Midpoint method is a specialization of the generic second-order Runge-Kutta method.

$$\begin{array}{c|cc} 0 & & \\ 1/2 & 1/2 & \\ \hline & 0 & 1 \end{array}$$

Available with: **include** ⟨*RK2*⟩

3.2.6 Ralston's

Ralston's method is a specialization of the generic second-order Runge-Kutta method.

$$\begin{array}{c|cc} 0 & & \\ 2/3 & 2/3 & \\ \hline & 1/4 & 3/4 \end{array}$$

Available with: **include** ⟨*RK2*⟩

3.2.7 Euler/Heun

Available with: **include** ⟨*Euler*⟩

3.2.8 RK3

Available with: **include** ⟨*RK3*⟩

3.2.9 Bogacki–Shampine 3(2)

0				
1/2	1/2			
3/4	0	3/4		
1	2/9	1/3	4/9	
	2/9	1/3	4/9	0
	7/24	1/4	1/3	1/8

Available with: **include** ⟨*BS32*⟩

3.2.10 RK4

Available with: **include** ⟨*RK4*⟩

3.2.11 RKF4(5)

Introduced by Fehlberg in [2], the RKF 4(5) method has been used as the default method[1] in a variety of ODE solvers since it's introduction in the late sixties. The method has an amortiezed cost of six function evaluations per step and an error of order four. The method is described by the Butcher's tableau,

0				
1/2	1/2			
3/4	0	3/4		
1	2/9	1/3	4/9	
	2/9	1/3	4/9	0
	7/24	1/4	1/3	1/8

Because it does not depend upon the timestep, the first evaluation is removed from the adaptation loop and, therefore, additional adaptation steps require only five function evaluations.

Available with: **include** ⟨*RKF*⟩

[1] In some cases, related $4^{th}/5^{th}$ order methods like the Cash-Karp 4(5) or the Dormand-Prince 4(5) method has been used. In any case, the RKF 4(5) method is important as representing a very good blend of accuracy and speed in problem-space of ode integration.

3.2.12 Butcher's 5th

Available with: **include** ⟨*Butchers*⟩

Chapter 4

The Integrator Process

4.1 Triggers

The ODE integrator works in response to only a few triggers: end, storage and limiting. In general, all of these triggers are based off of a single end-trigger identifier, though the system does allow for the creation and use of triggers seperately (MAKE THIS ACTUALLY BE TRUE!). Each of the triggers control a certain action within the integrator and together they allow for control of integration stepping, value storage and integration completion.

4.1.1 End Trigger

The most basic trigger (and the most visible) in **Epode** is the end trigger. This is a trigger which defines the circumstances under which integration will end. In its most basic form, this trigger will end an integration when the variable of integration has reached a certain value. If one wishes to integrate a system of ODEs from $t = 0$ to $t = 10$, creating the correct end trigger can be as simple as passing in the number 10. Internally, this will do the right thing. Coincidentially, it also will create a limit trigger which constrains the integrator to never go beyond the end point and a storage trigger which will store every integrator step. A method of specifying the end point which is just as direct is to provide a bracket-delimited initializer list of points. This list contains points at which the integrator should save the state values. As well as creating a multi-point storage trigger, this method creates a multi-point limit trigger which ensures that the integrator will estimate as close as possible to the choosen

4.1.2 Store Trigger

Storage triggers define when an iteration's state will be stored as part of the results set. In most cases, the default is to store all of the iterations. Using the simplified API, there is only one case in whtheich a subset of samples are stored – when an initializer list of points is passed into the iterator (or solve)

function. When this is done the store trigger will store only the iterations which correspond to the steps nearest to the listed points. Store triggers provide a flexible API which allows for determination of storage based on a variety of model characteristics. The API looks like:

```
bool store = storer(dv, v, y, stats)
```

A storage trigger can respond to any combination of: the current iteration's variable step, the iteration variable value at the end of the step, the state at the end of the step, or the system statistics (which includes number of method invocations and number of system function executions.

Store triggers are designed, primarily, to allow for reasonable subsets of values to be selected at the time of system calculation and, therefore, to save memory in the system. What this means, of course, is that not all of the calculated information will be returned. As part of saving the memory and optimizing execution time, a storage transformer may be used to transform the iterator state (dv, v, y, stat) into an alternative state vector which is returned instead. the advantage of using such a transformer is not only that the state size may be expanded or reduced, but also that functions between multiple state vectors are possible. The Ballistic Modeling example (7.1) uses a store transformer to convert from a state which consists of the projectile's velocity and position components into a state containing potential, kinetic and total energy, as well fired distance.

NEED TO DEFINE A GOOD WAY TO ADD STORE TRIGGERS SEPERATELY FROM THE END TRIGGER AND LIMIT TRIGGER.

4.1.3 Limit Trigger

Limit triggers cretate the actual step-size limits. Even in a fixed step-size method, some of the steps can be altered to ensure that the integration limits are respected. Step-size limits are defined as being in a minimum to maximum step range. The limit trigger takes iteration state as input and returns an object containing those upper and lower limits. Specifically, the limit trigger is an executable object with the following interface:

```
auto limits = limiter(dv, v);
```

That is, the limit trigger takes both the last step size and the current value of the variable of integration.

NOTE: IT MAY MAKE SENSE TO PASS THE STATE VARIABLE INTO THE LIMIT TRIGGER, FOR ADDED FLEXIBILITY (ESPECIALLY WITH CUSTOM TRIGGERS)

4.2 Logging

WHILE LOGGING WOULD BE A NICE FEATURE, IT CURRENTLY ISN'T IMPLEMENTED. FIGURE OUT HOW TO DO THAT AND THEN ADD IT

Chapter 5

Extension API

5.1 Extension Methods

It is always possible to implement new methods for use in **Epode**. The method API is fairly rich (and, potentially, complicated), but it provides for all of the available functionality being accessible to any new methods.

DESCRIBE HOW METHODS ARE CREATED. SHOW AN EXAMPLE – MAKE IT CLEAR THAT IT'S NOT NECESSARILY A GOOD METHOD

Part III

Usage Examples

Chapter 6

Toy Problems

SHOULD THE TOY PROBLEMS ALL BE SOLVABLE ANALYTICALLY?

6.1 Capacitor Discharge

The dynamics of capacitors in charging and discharging is very simple. It is also a question which we can easily solve analytically. This provides us an interesting option to test **Epode**. We pose a question. How long does it take for a resistor-capacitor system to discharge from an initial voltage to some proportion of that initial value. Analytically this problem can be solved as,

$$t_{discharge} = RCln(p) \tag{6.1}$$

where $t_{discharge}$ is the time required to discharge, R is the circuit resistance in ohms, C is the circuit capacitance in farads and p is the ratio of final voltage to initial voltage. We will use this equation to verify our numerical solution to the problem.

What is important to realize with this problem, however, is that the problem specification identifies an end condition which is not based upon the variable of integration – in this case, time. As such, we are going to need to use a custom end trigger. In this case, integration should end when the capacitor voltage drops below $V_{initial}p$, that is the initial voltage times the initial-to-final ratio.

The dynamic equation of our RC system can be defined simply as,

$$\dot{V} = \frac{V}{RC} \tag{6.2}$$

here we see that there is no direct connection between the time variable and the state of the system. This makes sense as we would not expect the discharge of a capacitor to be dependant on the time of day or phase of moon. The start time does not matter. Because there is a time related rate of voltage change, however, the relative time does matter.

THIS SHOULD PROBABLY BE EXTENDED SOME. IN PARTICULAR, THIS IS NOT DESCRIBING ANYTHING ABOUT THE ACTUAL CODE.

6.1.1 Results

HERE THERE IS A NEED TO GRAPH THE RESULTS AND COMPARE THE **Epode** ESTIMATE WITH THE ANALYTIC SOLUTION. IN PARTICULAR, IT WOULD BE NICE TO SHOW NUMEROUS METHODS AND COMPARE THE ERRORS IN CALCULATION.

6.2 A Random Complex Valued IVP

JUST SOMETHING SIMPLE LIKE $\dot{z} = (z - t)^2$

6.3 Van der Pol Oscillator

THIS CAN SHOW A SIMPLE TWO VARIABLE, MODERATELY STIFF SYSTEM

Chapter 7

Physical Simulation

7.1 Ballistic Modeling

Some of the earliest mechanical and electronic computers were created to assist in the calculation of ballistics. Why not do the same here? We are going to use **Epode** to model the possible range of a cannon and refer to [3] for inspiration. To begin with, it is important to outline precisely what we are expecting to achieve with this simulation. In the absence of aerodynamic drag, the equations of motion are very simple. Drag, as always seems to be the case, complicates the whole situation. Drag is a dynamic phenomenon and is difficult to estimate accurately. Cannon balls, however, have one saving grace. They are spherical. And spheres are one of the best understood of shapes aerodynamically. To calculate the drag of cannon balls we will depend upon code found in [4] which introduces drag coefficient estimates for a variety of body shapes.

Being a dynamic property, a cannon ball's coeficient of drag is based upon the atmospheric conditions (air density, temperature, etc.) as well as the velocity and surface finish of the cannon ball. There are a number of methods by which this drag coefficient can be estimated with the technical report mentioned above presenting two distinct methods. For this simulation we will use the 1972 Krumins method. This method is based upon both the Reynolds number and Mach number. This is not the place to get into what these values really refer to, however, Reynolds number describes the relative velocity of a body in relation to the boundary layer size of a fluid, while mach number is the ratio of a body's speed to the speed of sound in a fluid. Together, the two provide a very general description of dynamic environment of the body. These two parmeters will be estimated at each time step in order to estimate the coefficient of drag.

With coefficeint of drag and a physical description of the cannon ball (radius and mass) it is possible to calculate the acceleration of the ball through free space. This leads us to consideration of the equations of motion for our simultion. CONTINUE WITH INTRODUCING THE EQUATIONS OF MOTION

MUCH MORE HERE....

So, even with all this complication, are there things we are forgetting (ignoring)? Well, of course. To begin, our simulation will not include consideration of the Magnus force which can create a lifting force on balls which could substantially skew the results we are getting. As mentioned earlier, there is the surface finish of the cannon ball in flight to consider as well. As it stands, there are no dials or knobs to twiddle which will give differing results for smooth or rough surfaces. Similarly, non-ideal spheres are not simulated in any way. And there is more. Physical modeling is tricky business.

IT WOULD BE NICE TO DO A BALLISTIC MODEL THAT ALLOWS FOR TESTING WITH AND WITHOUT DRAG. IS THERE A WAY TO DO THIS IN THE ODE SOLVER, OR IS THAT PART OF THE SYSTEM DESCRIPTION?

7.2 Pendulum

Here we will solve a non-linear second-order equation which defines a pendulum with friction[1]. The equation,

$$m\ddot{\theta} + \lambda\dot{\theta} + \frac{mg}{L}sin(\theta) = 0 \tag{7.1}$$

defines how the angle of the pendulum, θ, changes with time given the angle (θ), a frictional factor (λ), the pendulum arm length (L), pendulum bob mass (m) and gravitational acceleration (g). We first need to convert this second-order equation to a first-order system so that it can be processed by **Epode**. We can use a change of variables to define the state variables as $y_0 = \theta$ and $y_1 = \dot{\theta}$. By substitution the equation becomes,

$$m\dot{y_1} + \lambda y_1 + \frac{mg}{L}sin(y_0) = 0 \tag{7.2}$$

and following a second-order to first order transformation,

$$\dot{y_0} = y_1 \tag{7.3}$$

$$\dot{y_1} = -\frac{\lambda y_1 + \frac{mg}{L}sin(y_0)}{m} \tag{7.4}$$

This is the system that we will implement.

`https://nrich.maths.org/6478`

Another Option[5]

7.3 Predator/Prey

New Zealand contains a triad of species the Kiwi, Rabbit and Stoats which form a linked predator-prey-prey system. An interesting article [6] demonstrates

[1] the `https://nrich.maths.org/content/id/6478/Paul-not so simple pendulum 2.pdf`

that a simple three-dimensional model of predator-prey behavior leads to the conclusion that one of the prey species will always approach extinction in such a system within a finite time even if there is no direct competition between prey species. The article approaches this question analytically. Here we will explore the same question numerically.

Chapter 8

Chaotic Attractors

8.1 Lorenz System

8.2 Rossler Attractor

8.3 Chua's Circuit

Chua's circuit was intended to show that a physical electronic circuit could demonstrate chaotic behavior, as documented by Chua himself in [7]. The circuit, like the Lorenz system and the Rossler system, is described by an ODE with three state variables. For this example, we shall use the implementation of the circuit described in [8].

Part IV

Appendicies

Chapter 9

Basics of ODEs

Chapter 10

Analysis of Numeric Methods

Chapter 11

Troubleshooting

LIST POSSIBLE ERRORS WITH THEIR CAUSE AND SOLUTION, ETC.

Chapter 12

Bibliography

[1] John Charles Butcher. A history of runge-kutta methods. *Applied numerical mathematics*, 20(3):247–260, 1996.

[2] Erwin Fehlberg. Low-order classical runge-kutta formulas with stepsize control and their application to some heat transfer problems. 1969.

[3] Amanda Wade. Going ballistic: bullet trajectories. *Undergraduate Journal of Mathematical Modeling: One+ Two*, 4(1):5, 2011.

[4] Robert J Yager. Calculating drag coefficients for spheres and other shapes using c++. Technical report, ARMY RESEARCH LAB ABERDEEN PROVING GROUND MD WEAPONS AND MATERIALS RESEARCH DIRECTORATE, 2014.

[5] Robert A Nelson and MG Olsson. The pendulum—rich physics from a simple system. *American Journal of Physics*, 54(2):112–121, 1986.

[6] Andrei Korobeinikov and Graeme C Wake. Global properties of the three-dimensional predator-prey lotka-volterra systems. *Advances in Decision Sciences*, 3(2):155–162, 1999.

[7] Leon O Chua. *The genesis of Chua's circuit.*

[8] Michael Peter Kennedy. Robust op amp realization of chua's circuit. *Frequenz*, 46(3-4):66–80, 1992.

Index