The swimmer Java library in clasJLib

D. Heddle, *Christopher Newport University*

Introduction

The swimmer library in the clas12 repository (and deployed in clasJLib) supports swimming charged particles through magnetic fields.

The input units are:

- Meters for any lengths
- Degrees for any angles
- GeV/c for momentum

State Vector

The swimmer library uses time (seconds) as the independent variable and a 6D state vector as the dependent variable. The 6D vector is:

x	The x coordinate (m)
y	The x coordinate (m)
z	The z coordinate (m)
θ_x	The momentum x direction cosine, p_x/p
θ_y	The momentum y direction cosine, p_y/p
θ_z	The momentum z direction cosine, p_z/p

This means that, for CLAS, all components are of the same order (unity), which facilitates stability.

The symbol *s* will be used throughout to indicate path length (in meters.)

A *trajectory* is nothing more than a collection of ordered state vectors.

Dependencies

The swimmer library depends on the magfield library (also in the repository and in also in clasJLib). The magfield library is standalone.

The swimmer library will integrate particles through a magnetic field. It is not CLAS specific—but at the moment the magfield library, which it uses to obtain field vectors, only has fieldmap files for the CLAS 12 torus and solenoid (which can be used separately or together).

The swimmer library uses the Lund particle designations. Support for the Lund format is provided in the cnuphys.lund package within the swimmer library. The swimmer doesn't need the Lund Id for

swimming, but some ancillary support, such as drawing trajectories (and *ced*), makes us of the Lund Id, if it is present. So a client can, optionally, set the Lund Id of a swum trajectory .

Modes of Operation

The `swimmer` library supports two types of integration advancing: *constant* (or *uniform*) step size and (the preferred) adaptive step size. The adaptive step size mode is recommended.

In addition, the `swimmer` library supports two modes of operation: the *trajectory* mode is when the swimming as carried out and the results (the 6D state vector at intermediate integration steps) are returned in a **SwimTrajectory** object. This is useful if you want to store the trajectory at a certain number of points. For example, *ced* uses this mode so that after the particle is swum *ced* can redraw the trajectory as needed without reswimming. The other mode of operation is the *listener* mode. In the listener mode, a listener is called at every advance of the integration, but no trajectory is cached and no **SwimTrajectory** object is returned. In the listener-mode the number of steps taken is returned.

The `swimmer` library also supports the use of **IStopper** objects that cause the integration to terminate when some specified condition is met, such as exceeding the maximum radial coordinate or precise integration to a value of *z*. Using a stopper is the preferred method to stop. The alternative (no stopper) is to integrate from $s_i = 0$ to s_f, where s_f is the final path length in meters.

The basic steps to swim a particle are:

1. Use the `magfield` library to create an **IField** object from one or more field maps.
2. Use the **IField** object to create a **Swimmer** object. The **Swimmer** can be used to swim any number of particles, and it is thread safe. *You only need a new **Swimmer** if you change magnetic fields.*
3. Obtain **the initial parameters** for the particle you want to swim. They consist of:
    a. The charge in units of *e*, e.g., -1 for an electron, +1 for a proton
    b. The starting position (x_0, y_0, z_0) in meters
    c. The starting polar angle θ and azimuthal angle ϕ in degrees.
    d. The momentum *p* in GeV/c.
4. (Optional) Create a listener (will be notified of each advance) if needed
5. (Optional) Create a stopper (will stop at a condition) if needed
6. Swim the particle using a call to `Swimmer.swim([args])`. The set of arguments will determine which underlying swimming algorithm is used.

Using the Swimmer Library

Note: there is a class `cnuphys.swim.Example` in the simmer library where you can find the examples here coded and tested.

Reading the magnetic fields

The **Example** class uses call to

```
IField field = MagneticFields.getIField(field)
```

to obtain a magnetic field in an `IField` object (which has methods to return the magnetic field at points in space) and which is then used by a swimmer. The single argument *field* is one of the following:

```
MagneticFields.FieldType.TORUS
MagneticFields.FieldType.SOLENOID
MagneticFields.FieldType.COMPOSITE
MagneticFields.FieldType.COMPOSITEROTATED
MagneticFields.FieldType.ZEROFIELD
```

Where "composite" means both the torus and the solenoid. The composite rotated field has been rotated by 25^{o} and is used in Veronique's tracking.

The library will look in various places for the CLAS field maps. If you have them anywhere within four levels of your home directory or four levels of the current working directory it should find them. As of now it searches—there is no provision for telling it the location.

Assuming a field has been read in successfully and a valid IField object (*field*) has been found, a **Swimmer** object can be constructed with the call:

```
Swimmer swimmer = new Swimmer(field)
```

This object is then used to swim any number of particles. You only need a new swimmer if you change magnetic fields.

The Swimmer Library

The Swimmer library is deployed in `clasJLib` in `clasJLib/swimmer`. The code is in the CLAS12 repository in `repos/clas12/swimmer`. The root package path is `cnuphys.rk4`.

There are three packages within the swimmer library:

1. **lund:** which is a support package for using particles identified by their Lund ID.
2. **rk4:** which contains general classes for Runge Kutta integration and is not specific for swimming charged particles through magnetic fields.
3. **swim:** which extends and specializes the rk4 classes to create objects that can swim particles through magnetic fields.

The IDerivative Interface

Problem specificity comes in supplying the derivative of the state variable. That is done through the **IDerivative** interface:

```java
/**
 * This interface is where the specificity of the problem enters. Any integrator such
 * as RungeKutta4 will solve 1st order ODEs of the form dy/dt = f(t). The vector y is
 * often a six-D vector [x, y, z, vx, vy, vx]. The specificity of the problem is implemented
 * by filling the dydt vector.
 * @author heddle
 */
public interface IDerivative {

	/**
	 * Compute the derivatives given the value of the independent variable and
	 * the values of the function. Think of the Differential Equation as being
	 * dydt = f[y,t].
	 *
	 * @param t the value of the independent variable (usually t) (input).
	 * @param y the values of the state vector (usually [x,y,z,vx,vy,vz]) at t (input).
	 * @param dydt will be filled with the values of the derivatives at t (output).
	 */
	public void derivative(double t, double y[], double dydt[]);
}
```

If you are just using the swimming then **you never have to worry about this**—because the derivative is common to all charged particle swimming through a magnetic field. In the swimmer library that is implemented in the **DefaultDerivative** class. This is done behind the scenes—each swim method in the Swimmer object will create the necessary **DefaultDerivative** object. The DefaultDerivative embodies the differential equation for charged particles in static magnetic fields.

Common Setup

There is some common setup regardless of what integration mode we choose. Suppose we want to integrate an electron with kinetic energy of 1 GeV from the nominal target position (0, 0, 0) with a theta of 30° and a phi of 0. And suppose we only want to use the torus.

```java
//create a swimmer for our magnetic field
Swimmer swimmer = new Swimmer(torus);

//OK lets integrate a negatively charged particle
int charge = -1;

//vertex position (m)
double xo = 0.0;
double yo = 0.0;
double zo = 0.0;

//initial angles in degrees
double theta = 30.0;
double phi = 0.0;

//these will be used to create a DefaultStopper
double rmax = 6.0; //m

//where the integration terminates if not stopped by a stopper
double maxPathLength = 8.0; //m

//initial step size in m
double stepSize = 5e-4; //m

//The  momentum, in GeV/c
double momentum = 1.0015
```

Creating a Stopper Object

A stopper object is any object that implements the **IStopper** interface:

```
/**
 * Given the current state of the integration, should we stop? This allows
 * the integration to stop, for example, if some distance from the origin
 * has been exceeded or if the independent variable passes some threshold.
 * It won't be precise, because the check may not happen
 * on every step, but it should be close.
 *
 * @param t the current value of the independent variable (typically time)
 * @param y the current state vector (typically [x, y, z, vx, vy, vz])
 * @return <code>true</code> if we should stop now.
 */
public boolean stopIntegration(double t, double y[]);
```

If the stopper is present, it will be called at every integration step. If it returns `true`, the integration is stopped. Note that precision is not guaranteed, in the sense that the following stopper:

```
IStopper stopper = new IStopper() {
     public boolean stopIntegration(double t, double u[]) {
         double x = u[0];
         double y = u[1];
         double z = u[2];
         double rsq = x*x + y*y + z*z;
         return   rsq > 49.0;
     }
};
```

Will stop the integration the first time a step takes us *past* r = 7.0m. It does not mean the last points will be at *exactly* r = 7.0m .

DefaultSwimStopper

There is a **DefaultSwimStopper** object that will stop when a maximum radial coordinate is attained. The constructor for the **DefaultSwimStopper** is:

```
/**
 * A default swim stopper that will stop if
 * the radial coordinate is exceeded
 * @param maxR the max radial coordinate in meters.
*/
public DefaultSwimStopper(final double maxR)
```

Creating a Listener Object

A listener object is any object that implements the **IRk4Listener** listener interface:

```
        /**
         * The integration has advanced one step
         * @param newS the new value of the independent variable (e.g., path length)
         * @param newY the new value of the dependent variable, e.g.
         * often a vector with six elements (e.g., x, y, z, px/p, py/p, pz/p)
         * @param h the stepsize used for this advance
         */
        public void nextStep(double newS, double newY[], double h);
```

If the listener is present, it will be called at every integration step. Note that every call to swim can have zero or one listeners—not an arbitrary number of listeners as in the usual Java “Listener” pattern. If (for some reason) you want to notify multiple objects, then you need to rebroadcast the callback to the single listener to all interested objects.

Special Note: the `nextStep` call to the listener is on the ***same* thread** as the integration. Your listener should be fast. If it is not, then you should redirect the callback to a separate thread for processing.

DefaultListener

There is a **DefaultListener** object (with only a null constructor) that keeps track of the most recent (and ultimately, last) value of the time and the state vector (location and velocity). The **DefaultListener** supports four public methods that can be called, typically when the integration is finished:

```
/**
 * Call reset if you are going to use the same listener again.
 */
public void reset()

/**
 * Get the independent variable (e.g., path length) for the last step.
 * @return the independent variable (e.g., path length) at the last step
 */
public double getIndependentVariable()

/**
 * Get the last state vector entries (e.g., [x,y,z, px.p, py/p, pz/p])
 * @return the last state vector
 */
public double[] getLastStateVector()

/**
 * Get the number of steps taken
 * @return the count, i.e., the number of steps taken
 */
public int getCount()
```

The reset method is needed if you want to keep using the same **DefaultListener**. If you do not reset it before swimming another particle the count will not restart at 0. There is really not much need to reset—just create a new one.

Examples

In all the examples below, we assume that we have created a **Swimmer** object as described above, using an **IField** object from the `magfield` library.

The actually swimmimg is performed with various overloaded calls to the **Swimmer** object's `swim` method. That method will return a **SwimTrajectory** object if we make a trajectory-mode `swim` call, or the number of steps taken if we make a listener-mode call.

In all the examples below, we use the same initialization:

```
// lets swim an electron. Start by getting its LundId
private static final int charge = -1; //units of e

// create a swimmer for the torus field
private static final Swimmer swimmer = new Swimmer(
               MagneticFields.getIField(MagneticFields.FieldType.TORUS));

// starting position (m)
private static final double xo = 0.0;
private static final double yo = 0.0;
private static final double zo = 0.0;

// initial angles in degrees
private static final double theta = 30.0;
private static final double phi = 0.0;

// Used to create a DefaultStopper
private static final double rmax = 7.0; // m

// Integration limit
private static final double maxPathLength = 8.0; // m

// The momentum, if the KE = 1 GeV
private static final double momentum = 1.00051; GeV/c

// get some fit results
private static final double hdata[] = new double[3];
```

Uniform Step Size, Trajectory Mode

In our first example we use uniform step size integration in the trajectory mode to cache trajectory state vectors. Since this is a trajectory-mode call, the return from swim is a **SwimTrajectory** object. We use that object and its stored trajectory points (state vectors) to generate a very crude console plot just to see if we get something reasonable.

```
// step size in m
double stepSize = 5e-3; // m

               // save about every 20th step
double distanceBetweenSaves = 20 * stepSize;

               // swim the particle, and return the results in a SwimTrajectory object
SwimTrajectory traj = swimmer.swim(electron.getCharge(), xo, yo, zo,
               momentum, theta, phi, rmax, maxPathLength, stepSize,
               distanceBetweenSaves);

               // how many steps did we save:
printSummary("\nresult from uniform step stepsize method with storage",
               traj.size(), momentum, traj.lastElement(), null);

               //make a crude terminal plot
terminalPlot(traj, "z (horizontal, cm) vs. x (vertical, cm) [FIXED]");
```

The results:

```
result from uniform step stepsize method with storage
Number of steps: 61
R = [ 2.39125,  0.00000,  5.45976] |R| =  5.9605 m
P = [2.2813e-01, 2.2815e-15, 9.7415e-01] |P| =  1.000511e+00 GeV/c
norm (should be 1): 1.000000000000002
----------------------------------------

                        z (horizontal, cm) vs. x (vertical, cm) [FIXED]

2.39e+02  |+---------+---------+---------+---------+---------+---------+---------+---------|
          |                                                                              *|
          |                                                                   * ** **    |
          |                                                             * ** **          |
          |                                                       * ** **                |
          |                                                 ** **                        |
          |                                            * ** *                            |
          |                                         * **                                 |
          |                                      * **                                    |
          |                                   ***                                        |
1.26e+02  +                                **                                           +
          |                             * **                                             |
          |                           **                                                 |
          |                       ** *                                                   |
          |                     **                                                       |
          |                  ***                                                         |
          |              * *                                                             |
          |           ***                                                                |
          |        **                                                                    |
          |    * **                                                                      |
0.00e+00  +***                                                                           +
          |+---------+---------+---------+---------+---------+---------+---------+---------|
       0.00e+00  6.82e+01  1.36e+02  2.05e+02  2.73e+02  3.41e+02  4.09e+02  4.78e+02  5.46e+02
```

Note that the number of trajectory points is 61, and stored about every 20th, so this integration used about 1200 steps.

Adaptive Step Size Examples

We now turn to adaptive step size examples. There are actually a variety of ways to do this—in addition to trajectory vs. listener mode there are different ways to specify the tolerances. The example here is for the simplest modes, which so-far seem to work well for CLAS. But in all honesty there is no "best" adaptive method—it is very problem specific. There is art involved.

It is worth remembering that there are two advantages to adaptive method:

1. In general the number of steps is reduced (although each step is more expensive)
2. There is better confidence in the results because they adapt to satisfy a tolerance/error requirement.

Our algorithm works like most adaptive algorithm: it computes the difference between 4^{th} order and 5^{th} order steps, and if that difference is greater than some specified tolerance the step size is reduced. If it is within the tolerance the step size is increased.

A detail: the Runge Kutta is implemented using what are called *Butcher Tableau*. The particular one that we are using, at least for now, is a Fehlberg tableau.

hdata

All the `swim` calls that use adaptive step size are distinguished by having, as the last argument, an array of three doubles (called `hdata` in all examples). Upon return from an adaptive step size call to swim, the values in the hdata array are:

- hdata[0] is the minimum step size used (m)
- hdata[1] is the minimum step size used (m)
- hdata[2] is the maximum step size used (m)

(The first time we ran the adaptive step algorithm we discovered that the minimum step size was greater than the step size we were using for the uniform step size calls, so we changed that value—greatly reducing the number of steps taken.)

Adaptive Step Size: `eps` and `yscale`

Before we demonstrate the adaptive step size calls we need to explain two more parameters. The parameter `eps` is the overall relative tolerance. It is the "accurate to one part in a whatever" parameter.

`yscale` converts eps to an absolute error for each element of the state vector. It can be thought of as the "order" of the maximum value of the state vector. For CLAS the x, y and z maximum values are on the order of 1 m. The velocity components are on the order of 10^8 m/s.

In the simplest form of the adaptive step size the user only provides the `eps` tolerance parameter, and the `yscale` vector is created (behind the scenes) to be {1, 1, 1, 3e8, 3e8, 3e8}.

In fact, and as shown in the adaptive steps size examples below, the user can just provide the predefined tolerance vector **`Swimmer.CLAS_Tolerance`**, which appears to be suitable for CLAS.

Adaptive Step Size, Trajectory Mode

Here we show a call to the adaptive step size algorithm in trajectory mode.

```
DefaultSwimStopper stopper = new DefaultSwimStopper(rmax);
// step size in m
double stepSize = 5e-4; // m
try {
        SwimTrajectory traj = swimmer.swim(electron.getCharge(), xo, yo, zo, momentum,
                theta, phi, stopper, maxPathLength, stepSize, Swimmer.CLAS_Tolerance, hdata);
        double[] lastY = traj.lastElement();
        printSummary("\nresult from adaptive stepsize method with storage and err vector",
                        traj.size(), momentum, lastY, hdata);

        terminalPlot(traj,
                        "z (horizontal, cm) vs. x (vertical, cm) [ADAPTIVE]");

} catch (RungeKuttaException e) {
        e.printStackTrace();
}
```

This produces the output:

```
result from adaptive stepsize method with storage and err vector
Number of steps: 112
min stepsize: 5.0E-4
avg stepsize: 0.05455502560181045
max stepsize: 0.20679515313825692
R = [ 2.40295,  0.00000,  5.50977] |R| =  6.0110 m
P = [2.2799e-01, 2.2980e-15, 9.7419e-01] |P| =  1.000511e+00 GeV/c
norm (should be 1): 0.9999999996172916
---------------------------------------

                        z (horizontal, cm) vs. x (vertical, cm) [ADAPTIVE]

2.40e+02  |+---------+---------+---------+---------+---------+---------+---------+---------|
          |                                                                              *|
          |                                                                     *******  |
          |                                                            *** **            |
          |                                                      *** **                  |
          |                                                 ******                       |
          |                                            ******                            |
          |                                        *****                                 |
          |                                     ****                                     |
          |                                  *****                                       |
1.26e+02  +                               ****                                          +
          |                            ****                                              |
          |                         ***                                                  |
          |                       *                                                      |
          |                   * *                                                        |
          |                * *                                                           |
          |             ***                                                              |
          |          *                                                                   |
          |       * *                                                                    |
          |    ***                                                                       |
0.00e+00  +***                                                                           +
          |+---------+---------+---------+---------+---------+---------+---------+---------|
        0.00e+00  6.89e+01  1.38e+02  2.07e+02  2.75e+02  3.44e+02  4.13e+02  4.82e+02  5.51e+02
```

From which you can see that the “plot” looks the same as in the uniform step size example but the number of steps has been reduced from about 1200 to about 112—although each step is more expensive.

Adaptive Step Size, Listener Mode

Here we demonstrate the adaptive step size in listener mode, using the same listener and stopper as before.

```
System.out.println("\n=== EXAMPLE 5 ===");
DefaultListener listener = new DefaultListener();
DefaultSwimStopper stopper = new DefaultSwimStopper(rmax);
double eps = 1.0e-08;
double stepSize = 5e-4; // m
try {
        int nstep = swimmer.swim(electron.getCharge(), xo, yo, zo,
                momentum, theta, phi, stopper, listener, maxPathLength, stepSize, eps, hdata);

        double[] lastY = listener.getLastStateVector();
        printSummary("\n\n================\nresult from adaptive stepsize method and yscale",
                        nstep, momentum, lastY, hdata);

} catch (RungeKuttaException e) {
        e.printStackTrace();
}
```

This results in the output:

```
================
result from adaptive stepsize method and yscale
Number of steps: 110
min stepsize: 5.0E-4
avg stepsize: 0.05500994375102368
max stepsize: 0.3125
R = [ 2.40205,  0.00000,  5.50586] |R| =  6.0070 m
P = [2.2801e-01, 1.1569e-15, 9.7418e-01] |P| =  1.000511e+00 GeV/c
norm (should be 1): 0.9999999997565097
```

On the next page is our final example: a fixed *z* cutoff.

Adaptive Stepsize, Fixed Z Cutoff

In this final example, we demonstrate swimming that stops at a fixed *z*. Rather than writing a stopper, there is a special swim call that handles this. That is because at the moment there is no general way to swim to an arbitrary plane (we anticipate adding such a feature) and so as a proof of principle (and at Veronique's request) we first implemented swimming to a fixed *z*. In the example below, the swimming is set to terminate within 20 microns (*accuracy*) of z = 2.75m (*ztarget*).

```
double ztarget = 2.75;  //where integration should stop
double accuracy = 20e-6; //20 microns
double stepSize = 5e-4; // m
try {
    SwimTrajectory traj = swimmer.swim(electron.getCharge(), xo, yo, zo, momentum, theta,
        phi, ztarget, accuracy, rmax, maxPathLength, stepSize, Swimmer.CLAS_Tolerance, hdata);
double lastY[] = traj.lastElement();
printSummary(
        "\nresult from adaptive stepsize method with storage and Z cutoff at " + ztarget,
        traj.size(), momentum, lastY, hdata);
terminalPlot(traj, "z (horizontal, cm) vs. x (vertical, cm) [ADAPTIVE] {STOP at Z=275}");
} catch (RungeKuttaException e) {
        e.printStackTrace();
}
```

This produces the output:

```
result from adaptive stepsize method with storage and Z cutoff at 2.75
Number of steps: 86
min stepsize: 5.0E-6
avg stepsize: 0.03797442853785494
max stepsize: 0.20679515313825692
R = [ 1.56105,  0.00096,  2.75001] |R| =  3.1622 m
P = [4.4160e-01, -2.7383e-04, 8.9778e-01] |P| =  1.000511e+00 GeV/c
norm (should be 1): 0.9999999999999999
----------------------------------------
```