The ODEToJava Manual

Andrew Kroshko

1 Introduction

ODEToJava is a problem solving environment (PSE) designed to facilitate the development and prototyping of algorithms for solving Ordinary Differential Equations (ODE). ODEToJava is written in the object-oriented language JAVA, which allows a modular construction and portability across many platforms. Currently supported methods for solving ODEs include the classic explicit Runge–Kutta (ERK) methods [10] and linearly-implicit additive Runge–Kutta (ARK) methods [1, 3, 4, 13]. The modular construction of ODEToJava helps organize different numerical methods, facilitates the reuse of code, and allows a user to readily add new functionality such as integration formulas, error control methods, and different types of output.

For linear algebra, as well as other matrix and vector computations, ODEToJava uses the numerical library JScience [16], which is a performance-oriented, thread safe, cross-platform pure JAVA library that conforms to the generally accepted coding practices for the JAVA language [2]. In order to conform to JAVA coding standards while preserving good performance, JScience is based on the real-time JAVA specification [6] compliant library Javolution in order to allow object reuse through object pools. In addition, ODEToJava itself uses the high-performance data structures provided by Javolution to further enhance performance. JScience also includes the Float64 and Complex number types, which unlike the primitive data types in JAVA are true object-oriented data types. Although ODEToJava does not currently support complex numbers, it could readily be modified to do so.

2 Building ODEToJava

2.1 ODEToJava Prerequisites

Building ODEToJava requires the JAVA SDK 1.6 or later and ANT, which is a tool for JAVA similar to make. JAVA can be obtained either from its main download website http://www.oracle.com/technetwork/java/javase/downloads/index.html or by using a package management system in the case of GNU/Linux and similar systems. On OS X, JAVA and ANT are typically installed by default, however, more advanced users can install alternate versions of JAVA and ANT. To check for the correct version of JAVA, execute the following at the command prompt:

```
$ java -version
```

The output should be similar to:

```
java version "1.7.0" Java(TM) SE Runtime Environment (build 1.7.0-b147) Java
HotSpot(TM) 64-Bit Server VM (build 21.0-b17, mixed mode)
```

The build tool ANT can be obtained from the following sources:

- GNU/Linux users can find ANT through the package management system for their particular distribution.

- OS X users who wish to install a different version of Ant can download it from its website or use their preferred package manager, such as Fink or MacPorts.
- Windows users can downloaded Ant from its website.

See `http://ant.apache.org/manual/install.html` for the installation instructions if downloading from the Ant website. To check that Ant is installed properly, execute the following command at a command prompt:

```
$ ant -version
```

The output should be similar to:

```
Apache Ant(TM) version 1.8.2 compiled on December 20 2010
```

Typical sources of error include: not setting the `ANT_HOME` environment variable that specifies the Ant installation directory, not updating the `PATH` variable with the `ANT_HOME/bin` directory, or not setting the `JAVA_HOME` environment variable to the Java installation directory. After Ant is installed, if `ODEToJava` is in a zip file, unzip `ODEToJava` with the following commmand:

```
$ unzip odeToJava2.zip
```

Once the `ODEToJava` package is in the appropriate directory, go to the root directory of the `ODEToJava` package:

```
$ cd odeToJava
```

In this manual, all of the directories in this manual are referenced with respect to the root directory of the `ODEToJava` package and `ODEToJava` packages are referenced with respect to the `ca.usask.simlab` package. The API documentation, which is in the `Javadoc` format that is used almost universally by Java software, contains a great deal of information that is not in this manual and is built by the command:

```
$ ant javadoc
```

To view the `Javadoc`, point your web browser to the `javadoc/index.html` file.

In general, to try new numerical methods or experiments with `ODEToJava` it will be necessary to study the structure of the code including the `Javadoc` and the examples provided within. That provides a starting point to trying novel and original tasks with the `ODEToJava` framework.

2.2 Building and using `ODEToJava` on the Command Line

The `odeToJava.jar` file is both a library that can be used by external programs and an executable for the examples included along with `ODEToJava`. The different commands for building the `odeToJava.jar` file and the resulting executable are given in Table 1.

Regardless of the version of the `.jar` file is built, the desired executable can be run with the command:

```
$ ant run
```

The two examples described later in this manual, which are built with the `tutorial-jar` Ant command, take a total of about five minutes to run on a 2.6 GHz AMD Athlon64 X2 machine.

The compiled `*.class` files can be cleaned up using the command:

Command	**Description**	Main-Class **file**
`ant test-jar`	Build an executable that tests the basic functionality of `ODEToJava`.	`tests/FunctionalityTest.java`
`ant manual-controller-jar`	Build an executable that runs the code from Section 4.1, which demonstrates the `odeToJava.controller` package.	`tests/ManualIVPController.java`
`ant manual-testable-jar`	Build an executable that runs the code from Section 4.4, which demonstrates the `odeToJava.testsuite` package.	`tests/ManualTestable.java`
`ant orbit-jar`	Build an executable that solves examples of celestial mechanics problems [9, 10, 11, 17].	`tests/OrbitTest.java`
`ant cardiac-jar`	Build an executable that solves the Luo-Rudy [15] cardiac problem.	`tests/Cardiac.java`
`ant pollution-jar`	Build an executable that solves an air pollution problem [17].	`tests/PollutionTest.java`
`ant nonstiff-jar`	Build an executable that solves the non-stiff DE test set from [8, 12].	`tests/NonStiffDETestSet.java`
`ant stiff-jar`	Build an executable that solves the stiff DE test set from [8, 9].	`tests/StiffDETestSet.java`
`ant mol-jar`	Build an executable that solves a PDE from [1] that has been discretized by the method of lines and is additively split into advection and diffusion terms.	`tests/MOLTest.java`

Table 1: The studies included with `ODEToJava` and the Ant commands that build them.

```
$ ant clean
```

To compile an appropriate Java source code file against a version of the `odeToJava.jar` use the following procedure. Assume the source code is named `Source.java` and it contains a class named `Source`. Copy the `jscience.jar` and `odeToJava.jar` into the same file folder as `Source.java`. To compile the file `Source.java` against `jscience.jar` and `odeToJava.jar` use the command:

```
javac -classpath ./jscience.jar:./odeToJava.jar:./ Source.java
```

On Windows machines, any classpath requires the `:` to be replaced by `;` and `/` to be replaced by `\` in the above command. On Windows compile with:

```
javac -classpath .\jscience.jar;.\odeToJava.jar;.\ Source.java
```

To execute the built class use the command:

```
java -classpath ./jscience.jar:./odeToJava.jar:./ Source
```

On Windows execute with:

```
java -classpath .\jscience.jar;.\odeToJava.jar;.\ Source
```

To increase the memory for better performance use the command:

```
java -classpath ./jscience.jar:./odeToJava.jar:./ -Xms512m -Xmx1024m -server
Source
```

For better performance on Windows execute with:

```
java -classpath .\jscience.jar;.\odeToJava.jar;.\ -Xms512m -Xmx1024m -server
Source
```

The arguments `-Xms512m` and `-Xmx1024m` set the initial and maximum Java virtual machine memory to 512 megabytes (the standard value = 2 megabytes) and 1024 megabytes (the standard value = 64 megabytes) respectively. The `-server` flag can be added to make the program run faster, although the program will take longer to compile.

2.3 Building The Manual

In order to build this manual, a TEX distribution that provides LATEX is required. Some examples of this include MikTEX for Windows, MacTEX for OS X, and TEXLive for GNU/Linux. Once the appropriate LATEX distribution is installed, run the following commands from the `ODEToJava` root directory to build the manual:

```
  cd manual
  pdflatex manual.tex
  bibtex manual
  pdflatex manual.tex
  pdflatex manual.tex
```

Note that `pdflatex` needs to be run multiple times in order to get the cross-references and citations correct. Many common IDEs for LATEX will perform this task automatically.

Location	Description	Reference
`nonstiffDETest/*.java`	The nonstiff DE test set is a set of standard nonstiff IVPs intended to test IVP solvers.	[7, 12]
`odes/Brusselator.java`	The Brusselator equation describes an oscillatory chemical reaction equation with two species.	[10]
`odes/BurgersMOL.java`	A semi-discretized advection-diffusion equation combining the Burgers equation for an advection term and a linear diffusion term with variable viscosity.	[1]
`odes/HiresODE.java`	An ODE from plant physiology that describes the 'High Irradiance RESponses' that model how light is involved in morphogenesis.	[17]
`odes/LuoRudyODE.java`	An IVP that describes the Luo-Rudy cardiac model that models the electrical activity in a cardiac cell.	[15]
`odes/OrbitArenstorfODE.java`	A planar restricted 3-body problem that describes a massless satellite in a periodic orbit, known as an *Arenstorf orbit*, along with the Earth and the Moon.	[10, 14]
`odes/PleiadesODE.java`	An IVP describing the motions of seven stars (reminding one of the famous Pleiades star cluster).	[10, 17]
`odes/PollutionODE.java`	An IVP describing the reaction of air pollution in the atmosphere.	[10, 17]
`stiffDETest/*.java`	The stiff DE test set is a set of standard stiff IVPs intended to test IVP solvers designed for stiff problems.	[7, 8, 18]

Table 2: The IVPs that are included with `ODEToJava`.

3 Implementing ODEs in `ODEToJava`

The ODEs that are included with `ODEToJava` and their locations in the `src/ca.usask/simlab/odeToJava` directory and important references are given in Table 2.

3.1 Implementing an ODE

ODEs are created by subclassing either `odeToJava.ode.ODE` in the case of a non-additive ODE or `odeToJava.ode.AdditiveOD` in the case of a 2-additive ODE that has a stiff linear part and nonstiff non-linear part. Both types of ODEs can be used with all categories of solvers in `ODEToJava`, however, some of the solvers may not take advantage of the special structure of additive ODEs. For stiff problems without a natural stiff/non-stiff additive splitting, `ODEToJava` uses the built-in IMEX methods as a linearly-implicit RK solver. Note that all JAVA classes are defined in their own file that shares the name of the class.

3.1.1 Creating a simple ODE

In order to demonstrate how to create an ODE in `ODEToJava` the example used is an *Arenstorf orbit*, which is a 3-body problem from celestial mechanics in which a satellite of negligible mass travels in a periodic orbit of a type known as an Arenstorf orbit. See Table 2 for the specific file defining this problem [10, 14].

In one variation of an Arenstorf orbit, the Earth and the Moon are put into circular orbits around their common center of mass using a coordinate system that is chosen in a such a way that both the Earth and the Moon remain fixed and only the satellite moves. The ODEs that describe this system are:

$$\begin{aligned}
\frac{d}{dt}y_1(t) &= y_1(t) + 2\frac{d}{dt}y_2(t) - \mu'\frac{y_1(t)+\mu}{D_1} - \mu\frac{y_1(t)-\mu'}{D_2},\\
\frac{d}{dt}y_2(t) &= y_2(t) - 2\frac{d}{dt}y_1(t) - \mu'\frac{y_2(t)}{D_1} - \mu\frac{y_2(t)}{D_2},\\
D_1 &= \left((y_1(t)+\mu)^2 + y_2(t)^2\right)^{\frac{3}{2}}, \quad D_2 = \left((y_1(t)-\mu')^2 + y_2(t)^2\right)^{\frac{3}{2}},\\
\mu &= 0.01277471, \quad \mu' = 1-\mu,
\end{aligned} \tag{1}$$

where the functions $y_1(t), y_2(t) : \mathbb{R} \to \mathbb{R}$ give the spatial location of the satellite at time t, the Earth is located at $(-\mu, 0)$, and the Moon is located at $(\mu', 0)$. The masses of the Earth and Moon are $\mu', \mu \in \mathbb{R}$ respectively. When an IVP is used with the initial values of:

$$\begin{aligned}
y_1(0) = 0.994, &\quad \frac{d}{dt}y_1(0) = 0,\\
y_2(0) = 0, &\quad \frac{d}{dt}y_2(0) = -2.00158510637908252240537862224,\\
t_0 = 0, &\quad t_f = 17.0652165601579625588917206249,
\end{aligned} \tag{2}$$

where $y_1(0)$ and $y_2(0)$ are the coordinates of the satellite at the initial time t_0 [10], the satellite has a periodic orbit where it returns to the initial state at positive integer multiples of t_f. A difficulty in integrating this system is that it has two singular points at the location of the Earth and the Moon. The satellite passes near to the singular point representing the position of the Moon, which causes difficulty when the problem is solved with a constant-stepsize.

In order to set up an ODE, it is necessary to create a class in its own file that subclasses `odeToJava.ode.ODE`. The function evaluation is defined by overriding the `f(Float64 t, Float64Vector x)` method, as well as the `get_size()` method to indicate the size of the ODE. The first step is to declare the package and import the necessary classes. It is required to import the `Float64` and `Float64Vector` classes from `JScience`:

```java
import ca.usask.simlab.odeToJava.ode.ODE;

import org.jscience.mathematics.numbers.Float64;
import org.jscience.mathematics.vectors.Float64Vector;
```

To set up ODE itself, create a class that subclasses `odeToJava.ode.ODE`:

```java
/**
 * ODE that describes a 3-body problem with the Earth, Moon, and
 * a massless satellite.
 * <p>
 * Benedict Leimkuhler, Sebastian Reich. "Simulating Hamiltonian
 * dynamics", Cambridge University Press, pg. 161-163, 2004.
 */
public class OrbitArenstorfODE extends ODE {
```

The function get_size must be overridden with the size of the ODE system:

```
    public int get_size() {
        return 4;
    }
```

Since JScience uses the Float64 data type, and complex calculations are easier to implement using primitive datatypes such as double, a conversion from Float64 to double can be done at the beginning of each function evaluation using the Float64.doubleValue() method, such as when the problem time t is required for the calculation of the RHS. Use the method Float64Vector.getValue(int i) to extract the i$^{\text{th}}$ component of the Float64Vector data type. For more complex function evaluations, it may be necessary to keep the values as their JScience types in order to take full advantage of the numerical methods provided by JScience. The method describing the RHS is declared and the first lines typically extract the necessary quantities:

```
    public Float64Vector f(Float64 t,Float64Vector y) {
        double y0 = y.getValue(0);
        double y1 = y.getValue(1);
        double y2 = y.getValue(2);
        double y3 = y.getValue(3);
```

A typical procedure is to create a new array of double types to store the results of the function evaluation, after which the function evaluation can be computed:

```
        double[] yp = new double[y.getDimension()];

        double d1 = Math.pow(Math.pow((y0 + mu), 2) + y1*y1, 1.5);
        double d2 = Math.pow(Math.pow((y0 - muhat), 2) + y1*y1, 1.5);

        yp[0] = y2;
        yp[1] = y3;
        yp[2] = y0 + 2 * y3 - muhat * (y0 + mu) / d1 - mu*(y0 - muhat) / d2;
        yp[3] = y1 - 2 * y2 - muhat * y1 / d1 - mu*y1 / d2;
```

For the return value, it is necessary to convert from the array of double values back into a Float64Vector object using the Float64Vector.valueOf(double[] a) method. ODEToJava requires the RHS value to be returned as a Float64Vector:

```
        return Float64Vector.valueOf(yp);
    }
```

Additionally, this particular problem requires that two constants to be defined and this is done using instance variables:

```
    private final double mu = 0.012277471; // masses of planet and sun
    private final double muhat = 1.0 - mu; // respectively
}
```

3.1.2 Implementing a 2-Additive ODE

An advection-diffusion problem from [1], which has the well-known Burgers equation as the advection component and a constant-coefficient linear diffusion component, is used as an example of how to implement a

2-additive ODE. The IVP is given as:

$$\begin{aligned}\frac{d}{dt}u_i(t) &= u_i(t)\frac{u_{i-1}(t)-u_i(t)}{\Delta x} + \frac{u_{i-1}(t)-2u_i(t)+u_{i+1}(t)}{(\Delta x)^2},\\ u_i(0) &= \sin\left(\pi\, i\, \Delta x\right),\\ u_1(t) &= 0,\\ u_m(t) &= 0,\end{aligned} \tag{3}$$

where $u_i \in \mathbb{R}$ is the concentration of grid point $i \in [1, m]$ in the spatial interval $x \in [0, 1]$, Δx is the spatial grid spacing, and ν represents the diffusion coefficient or viscosity.

In order to set up a 2-additive ODE, it is necessary to create a class in its own file that subclasses `odeToJava.ode.AdditiveODE`. It is not necessary to explicitly split the ODE in order to use the IMEX methods provided with `ODEToJava`, however, doing so can be advantageous and it eliminates a costly calculation involving the Jacobian during the automatic splitting [5].

The methods that must be overridden are `f1(Float64 t, Float64Vector y)` and `f2(Float64 t, Float64Vector y)`. The convention used by IMEX methods in `ODEToJava` is that the method `f1` corresponds to the non-linear component of the ODE and the method `f2` corresponds to the linear component of the ODE.

The source code for the following class can be found in `odeToJava.odes.BurgersMOLODE`. Like the case with a non-additive ODE, the package and imports must be defined first:

```java
import org.jscience.mathematics.numbers.Float64;
import org.jscience.mathematics.vectors.Float64Vector;
import org.jscience.mathematics.vectors.Float64Matrix;
import ca.usask.simlab.odeToJava.ode.AdditiveODE;
```

Create a class that subclasses the `AdditiveODE` class:

```java
/**
 * An ODE derived from the spatial discretization by finite differences
 * of the Burgers advection equation with diffusion.
 *
 * Uri Ascher, Steven Ruuth, Raymond Spiteri. "Implicit-explicit
 * Runge-Kutta methods for time-dependent partial differential equations",
 * Applied Numerical Mathematics, vol. 25, pg. 151-167, 1997.
 */
public class BurgersMOLODE extends AdditiveODE {
```

Set up any necessary member variables, in this case a constructor is used for the parameters:

```java
    private int n = 0;

    private double dx; // spatial stepsize
    private double nu; // the diffusion coefficient
    private double[][] jac;

    @Override
    public int get_size() {
        return n;
    }

    /**
     * The constructor for the ODE.
     */
    public BurgersMOLODE(int n, Float64 dx, Float64 nu) {
        this.dx = dx.doubleValue();
        this.n = n;
        this.nu = nu.doubleValue();
        jac = new double[n][n];
    }
```

Override the `f1` method for the non-linear part of the problem, advection in this case:

```
    @Override
    public Float64Vector f1(Float64 t, Float64Vector y) {
        double[] yp = new double[y.getDimension()];

        // apply upwind finite differences
        // the front point as a Dirchlet boundary condition
        yp[0] = 0.0;
        // the middle points
        for (int i = 1; i < y.getDimension() - 1; i++)
        {
            yp[i] = y.getValue(i)*(y.getValue(i-1) - y.getValue(i)) / dx;
        }
        // the end point as a Dirchlet boundary condition
        yp[n-1] = 0.0;

        return Float64Vector.valueOf(yp);
    }
```

Override the `f2` method for the linear part of the problem, linear constant-coefficient diffusion in this case:

```
    @Override
    public Float64Vector f2(Float64 t, Float64Vector y) {
        double[] yp = new double[y.getDimension()];

        // the front point as a Dirchlet boundary condition
        yp[0] =  0.0;
        // the middle points
        for (int i = 1; i < y.getDimension() - 1; i++)
        {
            yp[i] = nu*(y.getValue(i-1) - 2.0*y.getValue(i) + y.getValue(i+1)) / (
                dx*dx);
        }
        // the end point as a Dirchlet boundary condition
        yp[n-1] = 0.0;

        return Float64Vector.valueOf(yp);
    }
```

Define an analytical Jacobian, which avoids expensive numerical finite-differencing for finding the Jacobian:

```
    @Override
    public Float64Matrix jacobian(Float64 t, Float64Vector y) {

        double coeff = nu/(dx*dx);
        for (int i = 1; i < n-1; i++) {
            jac[i][i-1] = coeff;
            jac[i][i] = -2.0*coeff;
            jac[i][i+1] = coeff;
        }

        return Float64Matrix.valueOf(jac);
    }
}
```

The `f` method is not overridden and in the `AdditiveODE` class it is calculated internally by adding the `f1` and `f2` together.

4 The `IVPController`, `SolutionTester`, and `Testable`

The `IVPController`, `SolutionTester`, and `Testable` classes provide interfaces to the core `ODEToJava` solver, in order to allow the user to easily experiment with different numerical methods.

4.1 Basic Use

In order to allow the user to easily solve problems with many standard ODE methods and their commonly used parameters, the `IVPController` class is a wrapper around the core `ODEToJava` solver and its modules. This provides an interface for testing as well as providing a base on which interfaces such as GUIs for `ODEToJava` can be built.

An auxiliary class for `IVPController` is `IVP`, which stores an `ODE` object along with a set of initial conditions. The following shows how to set up a simple solver using `IVPController`, that solves the IVPs defined by the classes `OrbitArenstorfODE` and `BurgersMOLODE` from Section 3.1.1 and 3.1.2:

```
import org.jscience.mathematics.numbers.Float64;
import org.jscience.mathematics.vectors.Float64Vector;

import ca.usask.simlab.odeToJava.controller.IVP;
import ca.usask.simlab.odeToJava.controller.IVPController;
import ca.usask.simlab.odeToJava.modules.io.writers.DiskWriter;
import ca.usask.simlab.odeToJava.scheme.ERKButcherTableau;
import ca.usask.simlab.odeToJava.scheme.IMEXESDIRKButcherTableau;
```

Create a new class with the standard main method. In general, I/O exceptions raised by `ODEToJava` components should be handled individually, however, for the purpose of this tutorial the program is simply terminated.

```
public class ManualIVPController {
    public static void main(String[] args) throws Exception {
        IVP ivp;
        IVPController controller;
```

An `IVP` class is created that consists of an initial time, initial values and an `ODE` class:

```
        ivp = new IVP(new OrbitArenstorfODE(),
                      Float64.valueOf(0),
                      Float64Vector.valueOf(
                        0.994,
                        0.0,
                        0.0,
                        -2.00158510637908252240537862224));
```

Create an `IVPController` class by using the final time and the `IVP` class:

```
        controller = new IVPController(ivp, Float64.valueOf(17.1));
```

An excellent general-purpose nonstiff ERK method is the Dormand–Prince 5(4) method, however, any Butcher tableau provided with `ODEToJava` can be used. See the `Javadoc` for `odeToJava.scheme.ERKButcherTableau` to see the complete set of ERK methods:

```
controller.set_butcher_tableau(
  ERKButcherTableau.get_DormandPrince54_tableau());
```

`ODEToJava` supports several types of error control. For this example, we use the standard embedded error-estimation method along with the Dormand–Prince method:

```
controller.set_emb_error_control();
```

`ODEToJava` contains modules for producing solution arrays in memory and writing to a file, that facilitate interfacing `ODEToJava` with other programs. In this case, the output is to a file at a fixed interval using the special dense output method provided by the Dormand–Prince method [10]:

```
controller.write_at_interval(0.1);
controller.add_solution_writer(
  new DiskWriter("output/outputOrbitManual.txt"));
```

Finally with the ODE set up, it can be solved with the `run` method of the `IVPController` class:

```
controller.run();
```

Using the same methodology, an `IVPController` can be set up for the problem described by the `BurgersMOLODE` class that is given above. The method used is the order 4(3) method from [13]: See the `Javadoc` for `odeToJava.scheme.IMEXESDIRKButcherTableau` to see the complete set of IMEX methods:

```
ivp = new IVP(new BurgersMOLODE(127,Float64.valueOf(1./126.),Float64.
    valueOf(0.01)),
              Float64.valueOf(0),
              Float64Vector.valueOf(
                0.000000000000000000e+00, 2.493069173807287153e-02,
                4.984588566069715621e-02, 7.473009358642423994e-02,
                9.956784659581664754e-02, 1.243437046474851621e-01,
                1.490422661761744150e-01, 1.736481776669303312e-01,
                1.981461431993975508e-01, 2.225209339563143929e-01,
                2.467573976902936173e-01, 2.708404681430051086e-01,
                2.947551744109041527e-01, 3.184866502516844333e-01,
                3.420201433256687129e-01, 3.653410243663949841e-01,
                3.884347962746946825e-01, 4.112871031306115088e-01,
                4.338837391175581204e-01, 4.562106573531629627e-01,
                4.782539786213181876e-01, 4.999999999999999445e-01,
                5.214352033794980024e-01, 5.425462638657593262e-01,
                5.633200580636219534e-01, 5.837436722347897344e-01,
                6.038044103254773809e-01, 6.234898018587334834e-01,
                6.427876096865392519e-01, 6.616858375968593942e-01,
                6.801727377709193556e-01, 6.982368180860727414e-01,
                7.158668492597183297e-01, 7.330518718298262293e-01,
                7.497812029677340950e-01, 7.660444431189780135e-01,
                7.818314824680298036e-01, 7.971325072229223929e-01,
                8.119380057158565034e-01, 8.262387743159949061e-01,
                8.400259231507714031e-01, 8.532908816321554957e-01,
                8.660254037844385966e-01, 8.782215733702285476e-01,
                8.898718088114685454e-01, 9.009688679024190350e-01,
                9.115058523116731370e-01, 9.214762118704076244e-01,
                9.308737486442041353e-01, 9.396926207859083169e-01,
                9.479273461671317014e-01, 9.555728057861406777e-01,
                        9.626242469500120302e-01, 9.690772862290779610e-01,
                        9.749279121818236193e-01, 9.801724878485438275e-01,
                        9.848077530122080203e-01, 9.888308262251285230e-01,
                        9.922392066001720634e-01, 9.950307753654014098e-01,
                        9.972037971811801293e-01, 9.987569212189223444e-01,
                        9.996891820008162455e-01, 1.000000000000000000e+00,
                        9.996891820008162455e-01, 9.987569212189223444e-01,
                        9.972037971811801293e-01, 9.950307753654014098e-01,
                        9.922392066001720634e-01, 9.888308262251285230e-01,
                        9.848077530122080203e-01, 9.801724878485438275e-01,
                        9.749279121818236193e-01, 9.690772862290779610e-01,
                        9.626242469500121413e-01, 9.555728057861406777e-01,
                        9.479273461671318124e-01, 9.396926207859084279e-01,
                        9.308737486442042464e-01, 9.214762118704076244e-01,
                        9.115058523116732481e-01, 9.009688679024191460e-01,
                        8.898718088114687674e-01, 8.782215733702286586e-01,
                        8.660254037844387076e-01, 8.532908816321557177e-01,
                        8.400259231507715141e-01, 8.262387743159947950e-01,
                        8.119380057158566144e-01, 7.971325072229226150e-01,
                        7.818314824680301367e-01, 7.660444431189780135e-01,
                        7.497812029677344281e-01, 7.330518718298264513e-01,
                        7.158668492597185518e-01, 6.982368180860728524e-01,
                        6.801727377709196887e-01, 6.616858375968596162e-01,
                        6.427876096865394739e-01, 6.234898018587335944e-01,
                        6.038044103254777140e-01, 5.837436722347901785e-01,
                        5.633200580636222865e-01, 5.425462638657595482e-01,
                        5.214352033794981134e-01, 5.000000000000003331e-01,
                        4.782539786213184652e-01, 4.562106573531631293e-01,
                        4.338837391175582314e-01, 4.112871031306119529e-01,
                        3.884347962746946270e-01, 3.653410243663952617e-01,
                        3.420201433256688794e-01, 3.184866502516849329e-01,
                        2.947551744109045968e-01, 2.708404681430054417e-01,
                        2.467573976902938393e-01, 2.225209339563149480e-01,
                        1.981461431993976063e-01, 1.736481776669306920e-01,
                        1.490422661761747203e-01, 1.243437046474853425e-01,
                        9.956784659581717489e-02, 7.473009358642467015e-02,
                        4.984588566069748233e-02, 2.493069173807309705e-02,
                        1.224646799147353207e-16));
        controller = new IVPController(ivp, Float64.valueOf(2.0));
        controller.set_butcher_tableau(IMEXESDIRKButcherTableau.get_KC43_tableau()
            );
        controller.set_emb_error_control();
        controller.write_at_interval(0.1);
        controller.add_solution_writer(
          new DiskWriter("output/outputMOLManual.txt"));
        controller.run();
    }
}
```

The output array can now be found in the files `output/outputOrbitManual.txt` and `output/outputMOLManual.txt`.

4.2 Error Control

The basic parameters for a variable stepsize solver are the absolute tolerances and relative tolerances [10]. Absolute tolerance is set as follows:

```
controller.set_atol(1e-8);
```

Relative tolerance is set as follows:

```
controller.set_rtol(1e-6);
```

There are other stepsize-control heuristics that can be set for the standard error-controller methods that allow the user to fine tune the controller. `amax_normal` is the factor for the maximum increase in step size, and must be greater than one:

```
controller.set_amax_normal(3.0);
```

The `amax_rejected` parameter is the factor for the maximum increase in step size if the last step was rejected, and must be one or greater:

```
controller.set_amax_rejected(0.8);
```

The `amin` parameter is the maximum decrease in step size, and must be between zero and one:

```
controller.set_amin(0.2);
```

The safety factor is the fraction of the optimal step size to take, and must be less than one:

```
controller.set_safety(0.95);
```

4.3 Output

`IVPController` has several methods that produce output and several different output modules can be used simultaneously. The most basic method is to write every solution point, however, this is only possible for certain problems because large or long running problems will exceed the available disk space:

```
controller.write_all_points();
```

The output can be interpolate to a fixed time interval such as every 0.1:

```
controller.write_at_interval(0.1);
```

The output can be interpolated to a series of solution times given in a `Float64Vector` data type:

```
controller.write_at_array(array_of_times);
```

The standard output format consists of time in the first column of the file, then there are columns for each solution component of the ODE. Custom solution writers can be added as follows:

```
controller.add_solution_writer(new DiskWriter("output/outputManualOrbit.txt"));
```

Another possibility is to use the `odetojava.testsuite.SolutionCollector` class to store the solution times and solution values into arrays. The solution times and values can then be retrieved later from this object.

```
collector = new SolutionCollector();
controller.add_solution_writer(collector);
times = collector.get_times();
values collector.get_values();
```

4.4 The test suite

The classes `SolutionTester` and `Testable` provide an additional set of interfaces to allow the user to solve an IVP with a particular method: either with a set of stepsizes in the case of a constant-stepsize solver or a set of absolute and relative tolerances in the case of a variable-stepsize solver. In addition, the solution can be compared to a previously generated reference solution. This allows many methods and problems to be quickly evaluated with a range of parameters, by using the `SolutionTester` and `Testable` classes, without the user having to explicitly write a driver.

The `SolutionTester` class provides a more extensive interface for testing a variety of methods and parameters, while the `Testable` class provides an easier to use interface built on top of the `SolutionTester` class. Specific examples that use the `Testable` class can be found in most of the source code files referenced in Table 2.

4.4.1 Creating a `Testable` class

The `Testable` class lets the user run a variety of methods on a given problem and compare the result to a reference solution. An example demonstrating constant stepsize, embedded error control, and step-doubling error control on the ODEs defined in Section 3.1.1 and 3.1.2 is:

```java
import java.util.Vector;
import java.util.Arrays;

import org.jscience.mathematics.vectors.Float64Vector;
import org.jscience.mathematics.numbers.Float64;

import ca.usask.simlab.odeToJava.testSuite.Testable;
import ca.usask.simlab.odeToJava.scheme.IMEXESDIRKButcherTableau;


public class ManualTestable {
    public static void main(String[] args) throws Exception {
        Vector STEPSIZES = new Vector(Arrays.asList(1e-2, 3.16277e-3,
                                                    1e-3, 3.16277e-4,
                                                    1e-4));
        Vector ORBIT_RTOLS = new Vector(Arrays.asList(1e-4, 3.16277e-5,
                                                      1e-5, 3.16277e-6,
                                                      1e-6, 3.16277e-7,
                                                      1e-7, 3.16277e-8,
                                                      1e-8));
        Vector ORBIT_ATOLS = new Vector(Arrays.asList(1e-8, 3.16277e-9,
                                                      1e-9, 3.16277e-10,
                                                      1e-10, 3.16277e-11,
                                                      1e-11, 3.16277e-12,
                                                      1e-12));
        Vector MOL_RTOLS = new Vector(Arrays.asList(1e-2, 3.16277e-3,
                                                    1e-3, 3.16277e-4,
                                                    1e-4, 3.16277e-5,
                                                    1e-5));
        Vector MOL_ATOLS = new Vector(Arrays.asList(1e-2, 3.16277e-3,
                                                    1e-3, 3.16277e-4,
                                                    1e-4, 3.16277e-5,
                                                    1e-5));

        Testable orbit = new Testable(new OrbitArenstorfODE(),
                        "orbitReference.txt");
        Testable mol = new Testable(new BurgersMOLODE(127,Float64.valueOf(1./126.)
            ,Float64.valueOf(0.01)),
                      "burgersMOLReference.txt");

        orbit.test_const(ERKButcherTableau.get_DormandPrince54_tableau(),
                         STEPSIZES);
        orbit.test_embedded(ERKButcherTableau.get_DormandPrince54_tableau(),
                            ORBIT_RTOLS,ORBIT_ATOLS);
        orbit.test_sd(ERKButcherTableau.get_DormandPrince54_tableau(),
                      ORBIT_RTOLS,ORBIT_ATOLS);

        mol.test_embedded(IMEXESDIRKButcherTableau.get_KC32_tableau(),
                          MOL_RTOLS,MOL_ATOLS);
        mol.test_embedded(IMEXESDIRKButcherTableau.get_KC43_tableau(),
                          MOL_RTOLS,MOL_ATOLS);
        mol.test_embedded(IMEXESDIRKButcherTableau.get_KC54_tableau(),
                          MOL_RTOLS,MOL_ATOLS);
    }
}
```

5 Building the example code

The `ODEToJava` example that demonstrates `IVPController` from Section 4.1 can be built on the command line by following the general procedure in Section 2.2:

```
javac -classpath ./jscience.jar:./odeToJava.jar:./ BurgersMOLODE.java OrbitArenstorfODE.java ManualIVPController.java
```

On Windows compile with:

```
javac -classpath .\jscience.jar;.\odeToJava.jar;.\ BurgersMOLODE.java OrbitArenstorfODE.java ManualIVPController.java
```

To execute this example use the command:

```
java -classpath ./jscience.jar:./odeToJava.jar:./ ManualIVPController
```

On Windows execute with:

```
java -classpath .\jscience.jar;.\odeToJava.jar;.\ ManualIVPController
```

The `ODEToJava` example that demonstrates `Testable` from Section 4.4 can be built on the command line by the same general procedure:

```
javac -classpath ./jscience.jar:./odeToJava.jar:./ BurgersMOLODE.java OrbitArenstorfODE.java ManualTestable.java
```

On Windows compile with:

```
javac -classpath .\jscience.jar;.\odeToJava.jar;.\ BurgersMOLODE.java OrbitArenstorfODE.java ManualTestable.java
```

To execute this example use the command:

```
java -classpath ./jscience.jar:./odeToJava.jar:./ ManualTestable
```

On Windows execute with:

```
java -classpath .\jscience.jar;./odeToJava.jar;.\ ManualTestable
```

Sample output from the example code in 4.4.

```
run:
 Reference Solution: referenceSolutions/orbitReference.txt
 Method: Dormand-Prince, order 5, embedded order 4
         Solver   InitStepSize         AbsTol         RelTol       Time (s)       AbsError       RelError     MinSigFigs     AvgSigFigs
              0    1.00000E-02            N/A            N/A    1.15000E-01     8.38704E00     6.41626E03              0              0
              0    3.16277E-03            N/A            N/A    6.20000E-02     1.59046E00     4.75556E01              0              1
              0    1.00000E-03            N/A            N/A    1.64000E-01    6.91354E-04    1.37997E-02              1              4
              0    3.16277E-04            N/A            N/A    4.16000E-01    9.62481E-07    2.47659E-05              4              7
              0    1.00000E-04            N/A            N/A     1.30500E00    2.81928E-09    6.93880E-08              6              9

 Reference Solution: referenceSolutions/orbitReference.txt
 Method: Dormand-Prince, order 5, embedded order 4
         Solver   InitStepSize         AbsTol         RelTol       Time (s)       AbsError       RelError     MinSigFigs     AvgSigFigs
              0      Automatic    1.00000E-03    1.00000E-03    5.00000E-03     1.66181E00     3.36707E02              0              0
              0      Automatic    3.16277E-04    3.16277E-04    5.00000E-03    2.47467E-01     1.84915E01              0              1
              0      Automatic    1.00000E-04    1.00000E-04    6.00000E-03    1.55806E-01     2.37683E00              0              2
              0      Automatic    3.16277E-05    3.16277E-05    5.00000E-03    1.07646E-01     2.31076E00              0              2
              0      Automatic    1.00000E-05    1.00000E-05    9.00000E-03    2.78595E-02    5.85305E-01              0              2
              0      Automatic    3.16277E-06    3.16277E-06    5.00000E-03    1.55772E-02    3.02917E-01              0              3
              0      Automatic    1.00000E-06    1.00000E-06    7.00000E-03    3.53955E-03    6.70049E-02              0              3

 Reference Solution: referenceSolutions/orbitReference.txt
 Method: Dormand-Prince, order 5, embedded order 4
         Solver   InitStepSize         AbsTol         RelTol       Time (s)       AbsError       RelError     MinSigFigs     AvgSigFigs
              0      Automatic    1.00000E-03    1.00000E-03    7.00000E-03     1.36394E00     9.03075E01              0              0
              0      Automatic    3.16277E-04    3.16277E-04    5.00000E-03     1.09466E00     6.75989E01              0              0
              0      Automatic    1.00000E-04    1.00000E-04    9.00000E-03    5.39139E-01     4.87502E01              0              0
              0      Automatic    3.16277E-05    3.16277E-05    5.00000E-03    5.70689E-01     5.17525E01              0              0
              0      Automatic    1.00000E-05    1.00000E-05    5.00000E-03    3.74536E-01     4.59539E01              0              0
              0      Automatic    3.16277E-06    3.16277E-06    5.00000E-03    2.25659E-01     3.71225E01              0              0
              0      Automatic    1.00000E-06    1.00000E-06    6.00000E-03    1.28615E-01     1.63580E01              0              1

 Reference Solution: referenceSolutions/burgersMOLReference.txt
 Method: Additive Runge-Kutta, order 3, embedded order 2
         Solver   InitStepSize         AbsTol         RelTol       Time (s)       AbsError       RelError     MinSigFigs     AvgSigFigs
              0      Automatic    1.00000E-02    1.00000E-02     3.85400E00    4.58293E-02    2.09496E-01              0              6
              0      Automatic    3.16277E-03    3.16277E-03     3.07500E00    2.05077E-02    7.23931E-02              0              6
              0      Automatic    1.00000E-03    1.00000E-03     3.08400E00    8.66080E-03    4.74005E-02              0              6
              0      Automatic    3.16277E-04    3.16277E-04     3.77900E00    2.00303E-03    1.45888E-02              0              7
              0      Automatic    1.00000E-04    1.00000E-04     5.06600E00    8.09423E-04    5.22141E-03              0              7
              0      Automatic    3.16277E-05    3.16277E-05     7.13600E00    3.75510E-04    2.83356E-03              0              7
              0      Automatic    1.00000E-05    1.00000E-05     1.08770E01    1.65611E-04    1.47075E-03              0              7

 Reference Solution: referenceSolutions/burgersMOLReference.txt
 Method: Additive Runge-Kutta, order 4, embedded order 3
         Solver   InitStepSize         AbsTol         RelTol       Time (s)       AbsError       RelError     MinSigFigs     AvgSigFigs
              0      Automatic    1.00000E-02    1.00000E-02     7.90000E00    1.42296E-01    5.89973E-01              0              6
              0      Automatic    3.16277E-03    3.16277E-03     6.02800E00    1.53152E-02    4.64341E-02              0              6
              0      Automatic    1.00000E-03    1.00000E-03     4.21400E00    3.71687E-03    6.02579E-02              0              6
              0      Automatic    3.16277E-04    3.16277E-04     4.43200E00    1.49303E-03    2.35756E-02              0              6
              0      Automatic    1.00000E-04    1.00000E-04     5.80300E00    1.78820E-04    1.41493E-03              0              7
              0      Automatic    3.16277E-05    3.16277E-05     7.91700E00    1.29822E-04    2.10467E-03              0              7
              0      Automatic    1.00000E-05    1.00000E-05     1.11790E01    1.81329E-05    2.93970E-04              0              7

 Reference Solution: referenceSolutions/burgersMOLReference.txt
 Method: Additive Runge-Kutta, order 5, embedded order 4
         Solver   InitStepSize         AbsTol         RelTol       Time (s)       AbsError       RelError     MinSigFigs     AvgSigFigs
              0      Automatic    1.00000E-02    1.00000E-02     2.11880E01    3.62782E-02    1.82765E-01              0              6
              0      Automatic    3.16277E-03    3.16277E-03     1.89950E01    8.43747E-04    7.02058E-03              0              6
              0      Automatic    1.00000E-03    1.00000E-03     1.33330E01    3.36043E-04    3.82084E-03              0              7
              0      Automatic    3.16277E-04    3.16277E-04     1.64570E01    6.91675E-05    3.27950E-04              0              7
              0      Automatic    1.00000E-04    1.00000E-04     1.67390E01    4.01507E-05    3.09726E-04              0              7
              0      Automatic    3.16277E-05    3.16277E-05     1.77060E01    3.04821E-05    2.95253E-04              0              8
              0      Automatic    1.00000E-05    1.00000E-05     1.86570E01    2.40085E-05    2.80605E-04              0              8


BUILD SUCCESSFUL
Total time: 3 minutes 31 seconds
```

6 References

References

[1] Uri M. Ascher, Steven J. Ruuth, and Raymond J. Spiteri. Implicit-explicit Runge–Kutta methods for time-dependent partial differential equations. *Appl Numer Math*, 25(2-3):151–167, 1997. Special issue on time integration (Amsterdam, 1996).

[2] Joshua Bloch. *Effective Java*. Addison–Wesley Professional, second edition, 2008.

[3] Sebastiano Boscarino and Giovanni Russo. On a class of uniformly accurate IMEX Runge–Kutta schemes and applications to hyperbolic systems with relaxation. *SIAM J Sci Comput*, 31(3):1926–1945, 2009.

[4] M. P. Calvo, J. de Frutos, and J. Novo. Linearly implicit Runge–Kutta methods for advection-reaction-diffusion equations. *Appl Numer Math*, 37(4):535–549, June 2001.

[5] Graham J. Cooper and Ali Sayfy. Additive Runge–Kutta methods for stiff ordinary differential equations. *Math Comp*, 40(161):207–218, 1983.

[6] Peter Dibble, Rudy Belliardi, Ben Brosgol, et al. *The Real-time Specification for Java.* Addison–Wesley Professional, Boston, MA, USA, 2006.

[7] W. H. Enright and J. D. Pryce. Two FORTRAN packages for assessing initial value methods. *ACM Trans Math Softw*, 13(1):1–27, 1987.

[8] Wayne H. Enright and Thomas E. Hull. Test results on initial value methods for non-stiff ordinary differential equations. *SIAM J Num Anal*, 13(6):944–961, 1976.

[9] Wayne H. Enright, Thomas E. Hull, and Bengt Lindenberg. Comparing numerical methods for stiff systems of ODEs. *BIT*, 15(2):10–48, 1975.

[10] Ernst Hairer, Syvert Paul Nørsett, and Gerhard Wanner. *Solving ordinary differential equations I: Nonstiff problems.*, volume 8 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, second edition, 1993.

[11] Ernst Hairer and Gustaf Söderlind. Explicit, time reversible, adaptive step size control. *SIAM J Sci Comput*, 26(6):1838–1851, 2005.

[12] Thomas E. Hull, Wayne H. Enright, B.M. Fellen, and Arthur E. Sedgwick. Comparing numerical methods for ordinary differential equations. *SIAM J Num Anal*, 9(4):603–607, 1972.

[13] Christopher A. Kennedy and Mark H. Carpenter. Additive Runge–Kutta schemes for convection-diffusion-reaction equations. *Appl Numer Math*, 44(1–2):139–181, 2003.

[14] Benedict Leimkuhler and Sebastian Reich. *Simulating Hamiltonian dynamics.* Cambridge University Press, 2004.

[15] C. Luo and Y. Rudy. A model of ventricular cardiac action potential. *Circ Res*, 68(6):1501–1526, 1991.

[16] Silvëre Martin-Michiellot. JScience, a general scientific API in java, 2008.

[17] Francesca Mazzia and Cecilia Magherini. Test set for initial value problem solvers, release 2.4. Technical Report 4, Department of Mathematics, University of Bari, Italy, 2008.

[18] Lawrence F. Shampine. Evaluation of a test set for stiff ODE solvers. *ACM Trans Math Software*, 7(4):409–420, 1981.