Topic 1: Numerical Integration

1.1 Introduction

The aim of the document is to cover the practical aspects for numerical integration and how to use the local JAVA `integrator` classes to perform simple numerical integration. This document does not cover error analysis, accuracy of speed consideration or optimal choice of techniques. For this see and good text book on numerical computing.

1.2 The Basics

The aim of numerical integration is to numerically calculate an approximation solution to the

$$\frac{\partial y}{\partial x} = f(x,y)$$

where there is an analytical expression for $f(x,y)$, but it cannot be analytically integrated. More generally we have N couples equations of the form

$$\begin{aligned}
\frac{\partial y_1}{\partial x} &= f_1(x,y_1,y_2,\cdots,y_N) \\
\frac{\partial y_2}{\partial x} &= f_2(x,y_1,y_2,\cdots,y_N) \\
\cdots &= \cdots \\
\frac{\partial y_N}{\partial x} &= f_N(x,y_1,y_2,\cdots,y_N)
\end{aligned}$$

where each f_i is analytic, and we want a numerical solution for the $y_1,\cdots,y_N$ over the range $a \rightarrow b$, subject to an initial condition, being the values of $y_1,\cdots,y_N$ at $x = a$. In many occasions we will be dealing with the case that x is *time* and we will be starting with $x = 0$, but this is not always the case.

In many occasions we will actually have higher order differential equations, but these can always be written as coupled first order equations by using intermediate variables. We will see an example of this when we consider implementation of the second order forced harmonic oscillator equation in the programming examples. This means that the first order scheme in all that is needed for one-dimensional problems.

1.3 Integration Schemes

All the simple, linear, integration schemes have the same basis concept, so to solve,

$$\frac{\partial y}{\partial x} = f(x,y)$$

start at $x = a$, with initial condition of $y = y_0$, and calculate y at intervals Δx by the recurrence relation that

$$y_{i+1} = y_i + \Delta x \left[\frac{\partial y}{\partial x}\right]_{\text{Estimate}}$$

the two problems being (a) how to estimate the derivative and (b) how to set the step size. We will consider the derivative first, since this is actually the biggest problem.

1.3.1 Euler Integration

The simplest scheme to evaluate the derivative at the current x, y values, so the relation simply becomes,

$$y_{i+1} + y_i + \Delta x\, f(x_i, y_i)$$

where $x_i = a + i\Delta x$. This scheme is know an EULER INTEGRATION as is shown in figure 1. This scheme assumes that the derivative is constant over the interval Δx, which is almost always a very poor assumption. In fact for all non-trivial cases[1], the simple EULER scheme is totally useless since for most functions the errors e in each step add in one direction, so the integration goes totally wrong; making the step Δx smaller in an attempt improve things also often simply makes it *go wrong faster*. The simple first order EULER should only ever be used as as example of how *not* to do numerical integration.

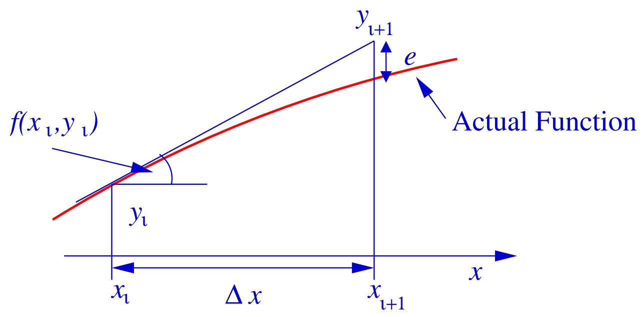

Figure 1: The Euler integration scheme.

1.3.2 Improved Euler Integration

The next level of approximation is to calculate the derivative as the *start* and *end* of the Δx interval and average. This gives the IMPROVED EULER integration scheme. At the start of the interval we have x_i, y_i, so the derivative is simple $f(x_i, y_i)$, which is exact. At the end of the interval we have that $x_{i+1} = x_i + \Delta x$, but we need as *estimate* for y_{i+1}, which we take to be the simple first order EULER estimate, given by,

$$y' + y_i + \Delta x\, f(x_i, y_i)$$

The *estimate* for the derivative at the end of the interval is now $f(x_{i+1}, y')$, so the IMPROVED EULER scheme becomes,

$$y_{i+1} = y_i + \frac{1}{2}\Delta x\, \left[f(x_i, y_i) + f(x_{i+1}, y')\right]$$

which is shown in figure 2. This scheme is much more stable, and is a viable integration scheme for simple fictions where the rate of change of the f_i are small. This is the simplest scheme that is likely to give sensible answers.

[1]where the solution is analytic anyway!

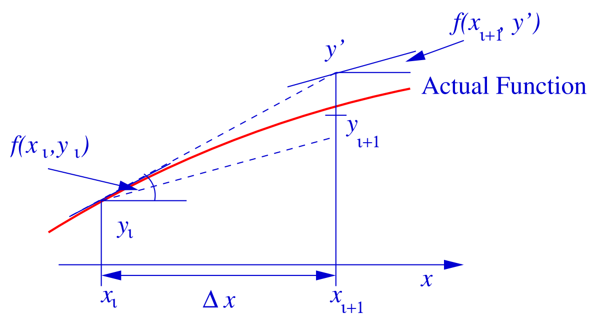

Figure 2: The Improved Euler integration scheme.

1.3.3 Runge Kutta Integration

The next level of sophistication is the four point RUNGE KUTTA scheme, often known as RK4. This is essentially a *double* IMPROVED EULER scheme where we estimate the derivative at *four* locations over the Δx internal, *one* at x_i, *two* at $x_i + \Delta x/2$ and *one* at x_{i+1}. The derivative used to step the y_i forward to y_{i+1} is then a weighted average of these four estimates.

If we write $x = x_i$, then at the start of the interval we have the exact derivative being,

$$k_1 = f(x, y)$$

we then use a simple EULER step a distance $\Delta x/2$, to get an *estimated* position at the middle of the interval being,

$$x_1 = x + \frac{\Delta x}{2} \quad \text{and} \quad y_1 = y + \frac{\Delta x}{2} k_1$$

The first *estimate* for the derivative at the centre is then given by the derivative at x_1, y_1, so being,

$$k_2 = f(x_1, y_1)$$

We now use this *estimate* for the derivative to to form a second estimate at x_1, being

$$y_2 = y + \frac{\Delta x}{2} k_2$$

The second *estimate* for the derivative at the centre is given by the derivative at x_1, y_2, so is given by

$$k_3 = f(x_1, y_2)$$

We now use this second *estimate* of the derivative to give an *estimate* for the point at the end of the internal, again using a simple EULER step, so giving at point,

$$x_2 = x + \Delta x \quad \text{and} \quad y_3 = y + \Delta x k_3$$

and finally we form an *estimate* for the derivative at the *end* of the interval, being,

$$k_4 = f(x_2, y_3)$$

giving us the required *four* estimates for the derivatives with their locations shown in figure 3.

The Runge Kutta step is then to form a weighed average of these four *estimates* with the *two* at the half step location weighted double, giving the scheme to be,

$$y_{i+1} = y_i + \Delta x \left[\frac{1}{6} k_1 + \frac{1}{3} k_2 + \frac{1}{3} k_3 + \frac{1}{6} k_4 \right]$$

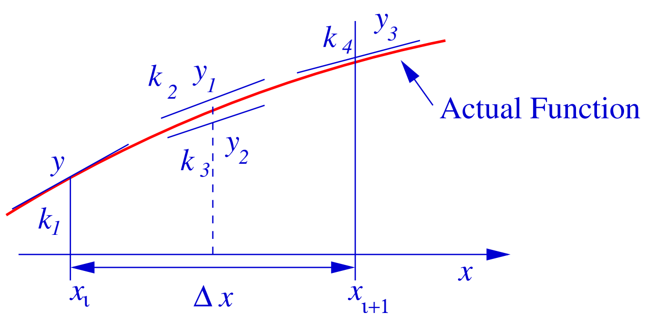

Figure 3: Location of derivative estimates in the four step Runge Kutta integration scheme.

This combination of point *can be shown*[2] to minimise the error, with the errors being of the order Δx^5. More importantly the error is independent of the form of the derivatives of the $f()$. This is particularly important in gravitational or eletrostatic orbits calcualtions, since this leads to conservation of total energy in the system.

Clearly there is a computational cost in this method since the functions have to evaluated at *four* point to take a single step. This means that Runge Kutta tends to be slow, but very reliable, and provided that the step size is *small* enough, can be used to numerically solve a very wide variety of coupled differential equations. For most people this is the the only algorithm you *will ever need*, and should generally be the first scheme you should try, and then investigate more complex schemes if you need more speed or you get unreliable results.

1.4 Extending to coupled equations.

The above schemes can be easily extended to a set of N coupled differential equations by making the **y** and **f** to be a vector of length N, so the equation to solve can be written at

$$\frac{\partial \mathbf{y}}{\partial x} = \mathbf{f}(x, \mathbf{y})$$

and in the particular case of the four point Runge Kutta scheme, we have that

$$\mathbf{y}_{i+1} = \mathbf{y}_i + \Delta x \left[\frac{1}{6}\mathbf{k}_1 + \frac{1}{3}\mathbf{k}_2 + \frac{1}{3}\mathbf{k}_3 + \frac{1}{6}\mathbf{k}_4 \right]$$

where the $\mathbf{k}_i$ and also vector of length N that contains the derivatives at the particular $(x, \mathbf{y})$ locations. As we will see below, formulating the integration scheme in this general form allow us to write a single *integrator* class that can be applied to a range of differential equations.

1.5 The step size

Selecting the *correct* step size Δx for integration is a very problem dependent issue. If the step size is too large and there are rapid variations in $f_i()$, these will be *missed* and significant errors will occur, while if the step size is too small there will be excessive computational time, and in extreme cases, the very large number of calculations actually reduce the overall accuracy due to build-up of rounding errors in the calculation.

[2]see for example *Numerical Recipes* or any other text book on numerical methods.

The simplest scheme is to try running the algorithm with a range of fixed step sizes and *see what happens* by graphing the components of $\mathbf{y}_i$ against x. You are ideally looking for the largest step size that gives a consistent solution, and in particular when *halving* the step size make no difference to the final solution. You can then be fairly confident that you have found a useful step size that will give you a meaningful solution in the minimum of computer time. Clearly this is a somewhat *ad-hoc* scheme, but in many conditions where all you want is a single solution to a set of equations, this is a far as you need to go.

In some conditions we want the step size of *adapt* during the calculation, in particular becoming larger where the $f_i()$ equations are slow varying, and becoming smaller where they vary rapidly. The simplest scheme to do this is a *half/double* scheme shown in figure 4.

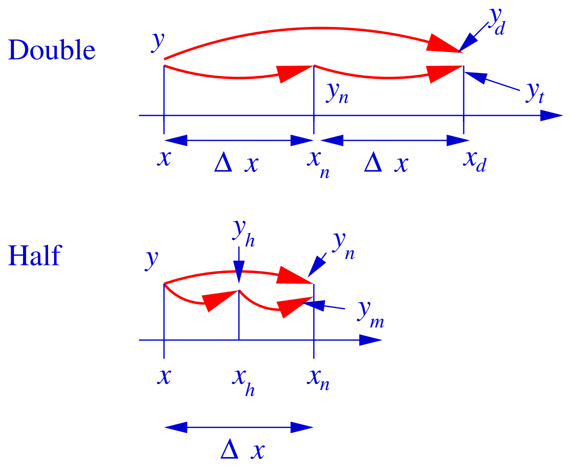

Figure 4: Layout of the double/half step scheme for adaptive step size.

Starting with a step size of Δx, then from a position (x, y) we form one forward step to (x_n, y_n), this is the reference position, we then,

1. Try in increase the step, by calculating

   (a) the next position using step size Δx, giving (x_d, y_t)

   (b) the *double step* starting at (x, y) with a step size of $2\Delta x$, giving (x_d, y_d).

   Now if y_t and y_d are sufficiently close, within a specified accuracy criteria, then we take (x_d, y_t) are being are current point and increase the step size of $2\Delta x$ for the next step.

2. if increasing the step size *fails*, then we have to check we done have to reduce it. Again taking (x_n, y_n), as the reference position, we calculate,

   (a) the half way position with step of $\Delta x/2$, being (x_h, y_h),

   (b) step the position forward again by $\Delta x/2$, to give a (x_n, y_m),

3. if y_n is sufficient close to y_m, then the current step size of Δx is valid,

4. if this test fails, then Δx is too large, since greater is obtained by using $\Delta x/2$, so reduce step size to $\Delta x/2$, and the current point becomes (x_h, y_h).

This scheme allows the step size to vary during the calculation while retaining a specified accuracy.

This scheme has two practical problems, these being:

1. the computational time taken testing for the optimal step size often outweights the advantage,
2. for simple slow varying functions the step size can become unreasonable huge; this is a particular problem is simple orbit problems where the step size can become a sizable fraction of the orbit period.

Adaptive step-size is however a useful scheme especially when the rate of change of the $f(x, y_1, \ldots, y_N)$ varies significantly over the range of interest. A good example is a highly elliptical satellite orbit where you want large step sizes during most of thge orbit appart from when it is close to its attractor, where you need very small steps.

This scheme is build into the `integrator` class described in the code section.

Example of Simple Harmonic Oscillator

The above description is somewhat abstract, but lets consider the example of a forced damped simple harmonic oscillator with a mass m, spring constant k, damping coefficient b with a forcing frequency ω and amplitude a, giving the second order differential equation of

$$m\ddot{x} + b\dot{x} + kx = a\cos(\omega t)$$

where x is displacement and $\dot{x}$ the velocity. We first divide through by m, to get,

$$\ddot{x} + \gamma x + \omega_0^2 x = \alpha\cos(\omega t)$$

where $\gamma = b/m$, $\alpha = a/m$ and $\omega_0^2 = k/m$ where we recognise that ω_0 is the natrual frequency of the system. This system is easily anaytically solvable in the *steady state* condition but is analytically tough in the transient stage; it therefore makes a good experimental test for numerical integration since you clearly know what should happend as $t \to \infty$.

To numerically calcuate this we need for formulate it as a pair of coupled first order differential equations by defining

$$x = t \quad , \quad y_1 = x \quad , \quad y_2 = \dot{x}$$

so that x is the time varaible, y_1 the displacement, and y_2 the velocity. This gives the coupled equations as,

$$\begin{aligned}\frac{\partial y_1}{\partial x} &= y_2 \\ \frac{\partial y_2}{\partial x} &= \alpha\cos(\omega x) - \gamma y_2 - \omega_0^2 y_1\end{aligned}$$

which is in the exact form required. This type of re-arrangement is essentail to prior to any programming. How this is actually programmed in described in the next section.

Range of variables

A computer hold floating point number to a fixed precission, this being typically 12 significant figures when using JAVA `double` types. This means you have to take extreme care when combining *very large* and *very small* numbers. On all computers,

$$c + h = c$$

where c is the speed of light and h is Plank's constant in SI units!

To prevent these type of problems you need to use units appropriate to the task so *all* variables of are comparable numerical size. A good working rule is you want to $|x|, |y_i| < 1000$ throughout the calculation[3].

This is one of the most common problem areas in numerical integration and some of the steps you will require to do this are:

1. Combine very large or small constants in equations, for example in Boltzman distributions use
$$\alpha = \frac{h}{k}$$
or where possible scale units to that such constants become unity.

2. Use sensible distance, time and energy units, for example in atomic orbital calcualtions used Angstrons, femto or pecoseconds and eVs, in Earth satellite ortbits use km and kiloseconds, and in particle interactions used GeV (or TeV), and femtoseconds.

   In practice few probelms can be solved numerically in SI units.

3. Avoid unrealistic starting conditions, for example a "comit" started at many light years distance from the Sun with a time step of a second, may simple never move, or worse wander in a random direction since the its realtive change in position may be less than the numerical accuracy of the computer!

If when you run the calculations you get *huge* or *tiny* values for your solved y_i, it is unlikely they will be very accurate and you need to rethink your units and scaling.

Java Code

To use the JAVA code we need firstly to look at some of the support classes.

`DataDerivative` class

This class holds a single data-derivative with one x value and an array of $\partial y_i/\partial x$ for $i = 0, \ldots, N-1$. All the internal values are held as `doubles` and has supporting method to access and manipulate these values. This class and `DataPoint` definded below, are central to the `integrator` being used to define the differential equations, set starting conditions and return solved solutions.

Constructors: There are three constructors, being

[3]This may not always be possible but is a good working target.

1. `DataDerivative()` default constructor to form a blank DataDerivative with $x = 0$ but does not define $y[]$ array which is set to `null`. This is included for GURU's who may want to extend this class.

2. `DataDerivative(int size)` constructor to form a DataDerivative with $x = 0$ and the $y[]$ array of length `size` which is set to zero. This is the most common and safest constructor.

3. `DataDerivative(double x, double y[])` constructor to form DataDerivative with specified x and $y[]$ double array. Note this does *not* take a local copy of the array.

In addition this class implements `Cloneable`, with method

- `DataDerivative clone()` which return a clone of the current DataDerivative where the y-array as also cloned.

Setters: The internal (private) variables can be set by the following methods:

1. `void setX(double x)` sets the x value.

2. `void setY(int i, double yValue)` sets the i^{th} component of the $y[]$ array. It assume thats $\mathtt{i} < \mathtt{size}()$, the length of the $y[]$ data array.

3. `void setY(double y[])` sets the $y[]$ data array, note does *not* take a local copy of the array.

Getters: The internal variables can be read by the following methods:

1. `double getX()` gets the x value.

2. `int size()` gets the size, or length, or the $y[]$ array.

3. `double getY(int i)` gets the i^{th} element of the $y[]$ array.

4. `double[] getY()` gets the $y[]$ data array.

5. `String toString()` gets the DataDerivative as a formatted String.

`DataPoint` class

This class holds a single data-point with one x value and an array of y_i for $i = 0, \ldots, N-1$. It *extends* `DataDerivative` defined above, with the same internal structure and methods, but adds additional control methods.

Constructors: There are three constructors, being

1. `DataPoint()` default constructor to form a blank DataPoint with $x = 0$ but does not define $y[]$ array which is set to `null`. This is included for GURU's who may want to extend this class.

2. `DataPoint(int size)` constructor to form a DataPoint with $x = 0$ and the $y[]$ array of length `size` which is set to zero. This is the most common and safest constructor.

3. `DataPoint(double x, double y[])` constructor to form DataPoint with specified *x* and *y*[] double array. Note this does *not* take a local copy of the array.

In addition this class implements `Cloneable`, with method

- `DataPoint clone()` which return a clone of the current DataPoint where the *y*-array as also cloned.

Additional Setters: The internal variables can also be incremented by,

1. `void addToX(double step)` add the value `step` to the current *x*.
2. `addToY(int i, double delta)` add the value `delta` to the i^{th} component of the *y*[] array.

Error Methods: The following methods for obtaining the error between the current DataPoint and a supplied DataPoint are available.

1. `double squareError(DataPoint p)` calculates the normalized square error between *y*[] array in the current DataPoint and the *y*[] array in the supplied DataPoint `p`.
2. `double maximumError(DataPoint p)` calcualte the maximum square error between *y*[] array of the current DataPoint and the *y*[] array of the supplied DataPoint `p`.
3. `error(DataPoint p)` the *default* calculated error between *y*[] array of the current DataPoint and the *y*[] array of the supplied DataPoint `p`. The *default* error, is `squareError`, but this can be changed by `setDefaultError()` method below.
4. `void setDefaultError(int type)` Method to set the meaning of *default* error returned by `error()` method. The current implemented values are:
   (a) `0` to give `squareError`, the default.
   (b) `1` to give `maximumError`.

These methods are called by the `Integrator` class when setting variable step size and are not normnally called by users except during DEBUGGING.
There are two other more advanced methods being

1. `DataPoint eulerStep(DataDerivative delta,double step)` with return a new DataPoint after a Euler forward step.
2. `DataPoint weightedStep(DataDerivative delta[], double w[], double step)` with rerturns a new DataPoint after a weighted step forward.

These can be overloaded in extending classes to allow `integrator` to be used with more advanced data structutures. Strictly GURU land, see the JAVADOC for details.

`DiffEquations` **class**

This is an *abstract* class that must be extended buy the user to implement the differential equation for the particular problem. There is **one** method that must be overloaded by the extending class that does the work being,

- `DataDerivative evaluate(DataPoint p)` which must calculate the differetials $\partial y_i/\partial x$ at the specify DataPoint `p` and return them in as DataDerivative.

This method is called by the `Integrator` class to actually do the integration.

The class is best explained by considering programming the simple harmonic problem, with a class `HarmonicEquations` detailed below:

```
import uk.ac.ed.ph.integrator.*;                    // (1)

public class HarmonicEquations extends DiffEquations { // (2)
    private double omegaSqr;                          // (3)
    private double gammaValue;                        // (4)
    private double omegaForce;                        // (5)
    private double ampForce;                          // (6)

    public HarmonicEquations(double omegaZero,
                             double gamma,
                             double omega, double a) { // (7)
        omegaSqr = omegaZero*omegaZero;               // (8)
        gammaValue = gamma;                           // (9)
        omegaForce = omega;                           // (10)
        ampForce = a;                                 // (11)
    }

    public DataDerivative evaluate(DataPoint p) {     // (12)
        double x = p.getX();                          // (13)
        double y[] = p.getY();                        // (14)
        DataDerivative d =
                new DataDerivative(p.size());         // (15)
        d.setX(x);                                    // (16)
        d.setY(0,y[1]);                               // (17)
        d.setY(1, ampForce*Math.cos(omegaForce*x) -
             (gammaValue*y[1] + omegaSqr*y[0]));      // (18)
        return d;                                     // (19)

    }
}
```

Look at the code line of the above class line at a time.

(1) Include the `integrator` classes.

(2) Class must extend the abstarct `Diffequations` class.

(2)→(6) internal variables.

(7) Constructor for class with four parameters.

(8)→(11) set the internal varaibles with parameters.

(12) Start of `DataDerivative evaluate` method with `DataPoint` parameter.

(13)→(14) get x and $y[]$ to local variables. Note $y[0]$ holds displacement and $y[1]$ the velocity.

(14) Create new `DataDerivative` of right size to hold derivatives.

(16) Set the x value to be same at supplied DataPoint.

(17) Set $\partial y[0]/\partial x$ to supplied velocity.

(18) Set $\partial y[1]/\partial x$ to acceleration.

(19) return `DataDerivative` holding derivatives.

`Integrator` class

The `Integrator` is the main class that performs the integration. The `Integerator` class itself is *abstract* with the actual integration performed by one of the three extending classes. All these classes share a common set of methods allowing a whole range of control and outputs.

The `Integrator` class is extended from `Vector<DataPoint>` so inherits the methods from the `Vector` class which is part of the `java.util` package.

Constructors: There are three constructor that implement three type of integration, there being:

1. `Euler(DiffEquations eqns)` class to implment the simple Euler integration scheme using the supplied `DiffEquations`. This is not practical useful scheme, and should only be used to see what goes wrong!

2. `ImprovedEuler(DiffEquations eqns)` class to implment the improved Euler integration scheme using the supplied `DiffEquations`.

3. `RungeKutta(DiffEquations eqns)` class to implment the RungeKutta integration scheme using the supplied `DiffEquations`.

Startup Methods:

1. `setStartConditions(DataPoint p)` method to set the starting conditions to the specified `DataPoint`. This point becomes the first solution point and the *current point*.

   If this method is called a second time *after* performing a `solve`, then all previoulsy stored solved `DataPoint` are lost and the integration is re-started.

2. `setStep(double step)` set the integration step size. If the *fixed step* scheme is used this will be constant throughout the integration, while if the *adaptive step* scheme is used this will be the initial step size. If not called the step size defaults to 0.01.

3. `setMaxStep(double step)` sets the *maximum* allowed step size when using the adaptive step scheme, defaults to `Double.MAX_VALUE`.

4. `setAccuracy(double acc)` set the error accuracy criteria for adaptive step size, default value is 10^{-7}.

5. `setMinStep(double step)` sets the *minumum* allowed step size when using the adaptive step scheme, defaults to 0.0.

6. `setAdaptiveTestInternal(int n)` set the number of steps between adaptive tests. If not called, defaults to 1, so adaptive test is done every step.

7. `setVerbose(boolean b)` set verbose mode, where in adaptice step size mode, changes in step size are printed to `System.out`.

Solving Methods: There is only one `solve` method being:

1. `DataPoint solve(double xEnd, boolean adaptive)` solves from the *current point*, usually set by `setStartConditions()` until x-coordinate of the solution $\geq$ `xEnd`. If `adaptive` is `true` then *double/half* adaptive step size is used, otherwise *fixed* step size is used.

   At the end of the integration the method returns the *current* DataPoint, being the last point solved for.

   On completion, the *current point* will be set to the last point solved for, so subsequent calls to `solve` with a larger `xEnd` will result in continuting the solution to the new `xEnd` value.

   By default, the `DataPoint` at *every step* is stored in the underlying `Vector` class, and can be accessed as descibed below. This behaviour can be controlled, see GURU methods at the end.

Reading out solved DataPoints: The default action is to store all solved DataPoints is the underlying `Vector` with can be accessed as follows:

1. `int size()` get the number of stored `DataPoints` (inherited from `Vector`).

2. `DataPoint get(int i)` get the i^{th} `DataPoint`. (inherited from `Vector`).

3. All other `Vector` methods, for example `firstElement()`, `lastElement()` all work.

Note also that last DataPoint solved for is returned by the `solve` methods.
For most application, this is all that is needed.

Guru Features

The `Integrator` class has a series of other methods and features to make it more flexible or return more information about the operation.

1. `setSaveInternal(int interval)` changes the interval between saves of the current `DataPoint` to the underlying `Vector`. This does not affect the accuracy of the calculation just how often it is saved. The deafult is 1, so every point is saved.

   Note: setting is 0 results in **no** `DataPoint` being saved, so only the last solved for DataPoint returned by `solve` is available. This is sulaully conbined with the `SolveMonitor` classes below.

2. `double getStep()` return the last step sized used.

3. `int getStepNumber()` gets the number of steps in the integration. This does not depend on the save interval set via `setStepInterval`, so may differ from the `size()` of the `Vector` of solutions.

`SolveMonitor` Class

This is an class that that allows a monitor method to be called periodically by *Integrator* during the solve process. This is an alternative method for obtaining solved DataPoint, but is mainly added to allow `integrator` to act as the driver for graphical animations.

`SolveMonitor` is declared as *abstract* and has one method that must be overloaded in an extending class, this being

- `void updateMonitor(DataPoint p)` which is called by the `Integrator` with the current DataPoint as its parameter.

This is then controlled with the `Integrator` methods,

1. `void addMonitor(SolveMonitor m, double interval)` which will result in the `SolveMonitor` method `updateMonitor()` being called at x intervals of `interval`.

   This is *not* related to the step size, and will be called at equal[4] x intervals even if the adaptive scheme is used.

2. `void setMonitorInterval(double interval)` set (or resets) the monitor interval for attached `SolveMonitor`.

3. `void setMonitorTimeInternal(int t)` sets the *minimum* time interval in msecs between calls to `updateMonitor()`. Default is 0.

   If set to > 0 the `solve` thread will be put into a `sleep` state until the next update is due. This methods used the software system clock in msecs to is likely to have ± 1 msec variability.

4. `void removeMonitor()` removes any attached `SolveMonitor`.

There is one predefined `SolveMonitor` that gives simple formatted output, that being `PrintMonitor`, constructors

1. `PrintMonitor()` default constructor to give formatted output to `System.out` which is normally the terminal screen.

[4]it will actually be called at the step when $x \geq$ monitor interval.

2. `PrintMonitor(File file)` gives formated output to file specified by `file`.

3. `PrintMonitor(String fileName)` gives formatted output to a file specified by `fileName`.

If the file is not available for writting, then it will default to `System.out`.

The format of the DataPoint is performed by the `DataPoint` method `formatPoint` which is currently set to format $x\ y_0\ y_1\ \ldots$ on a single line with space between each number and no other format. This being the easiest format to read into other graphical packages.

The real aim of this class it to drive graphical animations where you are required to write your own class, extending `SolveMonitor`, to do the graphical update. You would then typically:

1. Attach your monitor class via `addMonitor()` setting the x interval for updated.

2. Optionally set the minumum time interval for updates using `setMonitorTimeInterval()`. This will typically be needed to give a smooth animation and to stop the animation speed varying depending on processor, garphics, network speeds.

   Also for simple equations, stopping it *going too fast*.

3. Set `setSaveInternal(0)`, which will stop interdediate DataPoint being stored in the underlying `Vector`.

4. start the `solve` with a *very large* `xEnd`, for example `Double.MAX_VALUE` which will noramally cause it to loop *for ever*. It is expected that the application will the terminated by one of its other threads, like a STOP button on the graphical panel!

Getting animations to work well is rather tricky, and in many cases the much of the programming effort and complexity is likely to be in the monitor class which has to do *all the work.*