**Laboratory 7: Modelling motion**

**Introduction:** Calculus is the study of change. Calculus is divided into two main areas, differentiation that calculates the slope of a curve and integration that calculates areas under a curve. The following is a brief reminder of what you probably already know.

θ

ΔX

ΔY

A1

A2

Integration

Differentiation

**Figure 1:** Calculus includes Integration and Differentiation as demonstrated on a parabola.

**Integration**

The area, A, (shown in green) under the parabola is typically calculated as follows,

Another (approximate) way to measure the area is to calculate the area of the rectangle and triangle that sit under the curve and add them together,

*A2*

*A1*

**Differentiation**

The slope, m, of the line (shown in red) is typically calculated as follows,

At point *x=0.5* the slope is *m=1*.

Another (approximate) way to measure the slope is to take the ratio of the height and base of the triangle shown coinciding with the tangent to the curve.

**Part 1:** Plotting some well-known mathematical functions and their derivatives.

Consider the following equation that we wish to plot on the interval shown,

The sine function will return values on the interval [-1,1]. We can start by creating a window that is 400x400 pixels organised such that the top left hand corner is at (0,0) and the bottom right hand corner is at (399,399). We will need to scale up the value passed to and returned by the sine function so we can plot it on the screen.

The *map()* method available in Processing provides a means to do this rescaling of values,

float new\_x=map(theta,0,2\*PI,0,width); // Calculate x position

float new\_sy=map(sin(theta),-1,1,height,0); // Calculate new y value given sine theta

The map() method has the following syntax,

outgoing value = map(incoming value, incoming lower bound, incoming upper bound,

outgoing lower bound, outgoing upper bound)

The following code creates a range of values for theta using a loop

*Runs once at start, creating a floating point variable named theta with the value of 0.*

*Before entering (or re-entering) the code block check if the theta is less than 2π.*

*At the end of the code block increase the value stored in the variable theta by*

*one fiftieth of two pi.*

for(float theta=0;theta<=2\*PI;theta+=(2\*PI)/50)

{

*Code grouped together by the brackets is known as a code block and runs each time you go around the loop.*

}

If we bring together our loop and the mapping for the screen co-ordinates, we can write the following program to plot 50 points representing a sine function.

Remember the points do not appear as you are drawing them, you are just loading them into a screen buffer. You are only shown this buffer at the end of *draw(){}* method. You could consider the “}” at the end of the method as equivalent to “replace what was on the screen with all the things I have asked to be shown in the current call to *draw()”*

**To Do:** Enter and run the following short program that captures the ideas contained in the discussion so far.

void **setup**()

{

  size(400, 400);

  background(255,255,255);

}

void **draw**()

{

  float step=(2\*PI)/50; // Step is 1/50 of interval [0,2PI]

  for(float theta=0;theta<=(2\*PI)+step;theta+=step) // Loop theta values on range [0,2PI]

  {

    float screen\_x=map(theta,0,2\*PI,0,width); // Calculate screen x position from theta

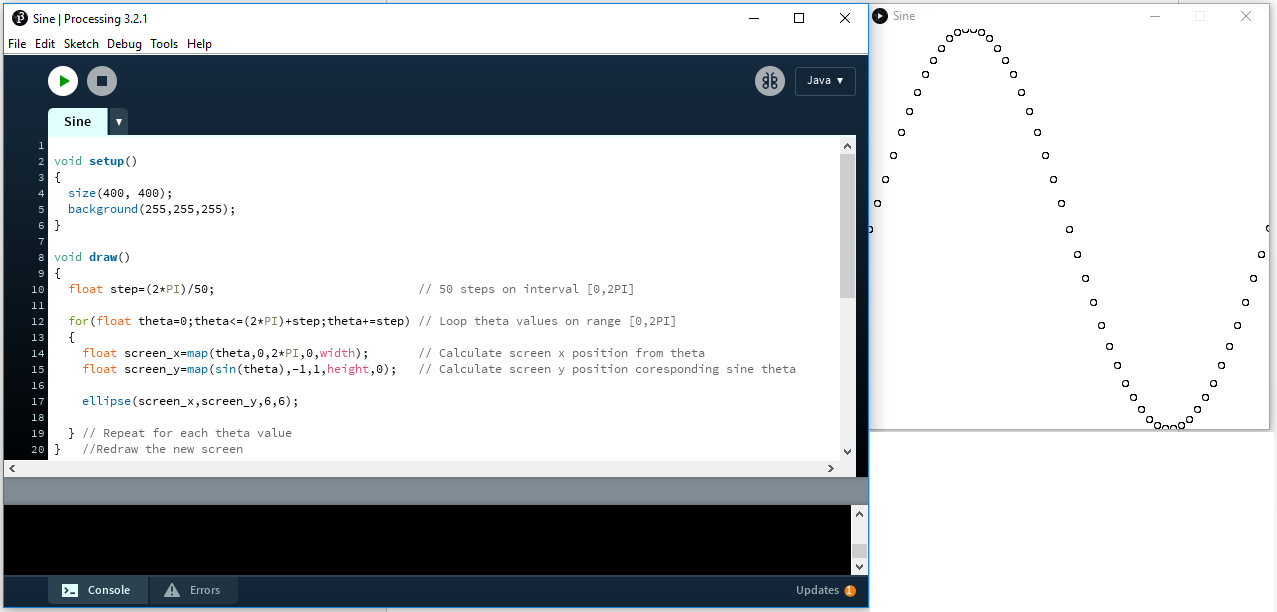
    float y=sin(theta);

    float screen\_y=map(y,-1,1,height,0); // Calculate screen y posn. corresponding sine theta

    ellipse(screen\_x,screen\_y,6,6);

  } // Repeat for each theta value

}   //Redraw the new screen

****

**Figure 2:** Simple plot of the sine function.

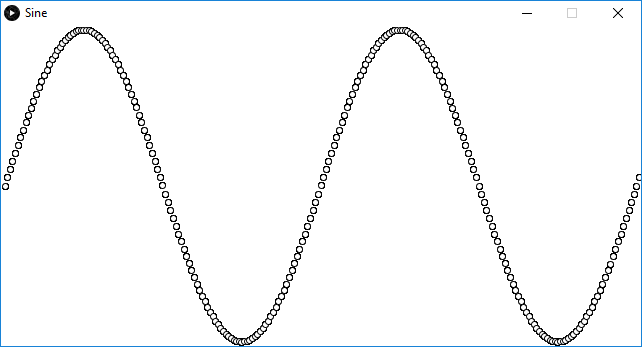
**To Do:** Modify the program so that

1/ The screen size is 640x320 pixels,

2/ two complete periods of the sinewave are shown on the graph,

3/ the points at the top and bottom of the screen are not clipped,

4/ that the graph includes 150 points.



**Figure 2:** Simple plot of the sine function, see if you can produce it (or similar).

If we want to replace the points with lines, we have to add code to remember the previous point as we go around the loop so that we can draw a line from the previous point to the current point, i.e. join the dots. We introduce two new variables to do this *old\_sx* (old screen x position) and *old\_sy*.

**To Do:** Replace the existing code with the following code.

void **setup**()

{

  size(400, 400);

  background(255,255,255);

}

void **draw**()

{

  float step=(2\*PI)/120; // 50 steps on interval [0,2PI]

  float theta\_start=0; // Starting theta value

  float old\_sx=map(theta\_start,0,2\*PI,4,width-4); // Calculate screen x position from theta

  float old\_sy=map(sin(theta\_start),-1,1,height-4,4); // Calculate screen y position corresponding to sine theta

  for(float theta=step;theta<=(2\*PI)+step;theta+=step) // Loop theta values on range [0+1step,2PI]

  {

    float screen\_x=map(theta,0,2\*PI,4,width-4); // Calculate screen x position from theta

    float screen\_y=map(sin(theta),-1,1,height-4,4); // Calculate screen y position coresponding sine theta

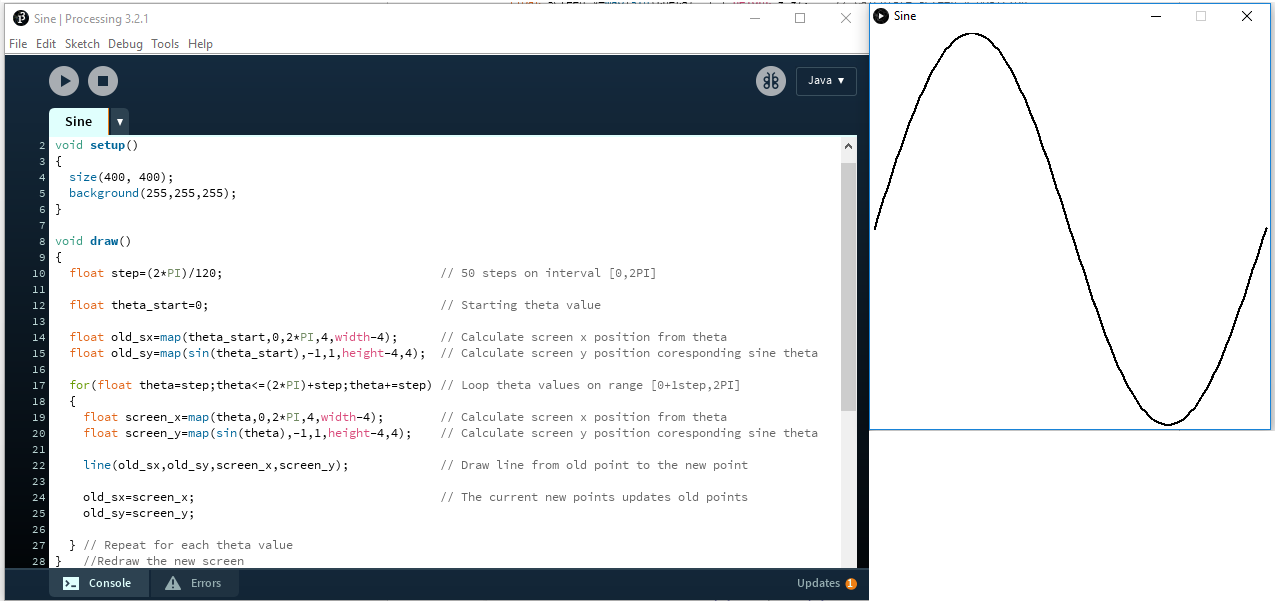
    line(old\_sx,old\_sy,screen\_x,screen\_y); // Draw line from old point to the new point

    old\_sx=screen\_x;                                   // The current new points updates old points

    old\_sy=screen\_y;

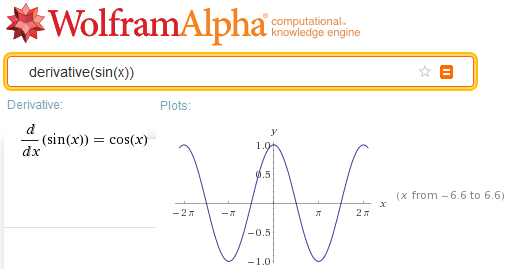
  } // Repeat for each theta value

}   //Redraw the new screen

****

**Figure 3:** Line plot of the sine function.

We can find the derivative of the sine function using our knowledge of mathematics (or for speed) use Wolfram Alpha.

****

**Figure 4:** Wolfram alpha used to show derivative of *sin(x)* is *cos(x).*

To over plot the *cosine()* function (derivative) we will need to introduce an additional, y, variable.

**To Do:** Complete the following code to produce the plot shown in the next figure.

void **setup**()

{

  size(400, 400);

  background(255,255,255);

}

void **draw**()

{

  float step=(2\*PI)/120; // 50 steps on interval [0,2PI]

  float theta\_start=0; // Starting theta value

  float old\_sx=map(theta\_start,0,2\*PI,4,width-4); // Calculate screen x position from theta

  float old\_sy1=map(sin(theta\_start),-1,1,height-4,4); // Calculate screen y position coresponding sine theta

  float old\_sy2= // ADD YOUR CODE Calculate screen y position coresponding cosine theta

  for(float theta=step;theta<=(2\*PI)+step;theta+=step) // Loop theta values on range [0+1step,2PI]

  {

    float screen\_x=map(theta,0,2\*PI,4,width-4); // Calculate screen x position from theta

    float screen\_y1=map(sin(theta),-1,1,height-4,4); // Calculate screen y position coresponding sine theta

    float screen\_y2=// ADD YOUR CODE

    stroke(255,0,0);

    line(old\_sx,old\_sy1,screen\_x,screen\_y1); // Draw line from old point to the new point

    stroke(0,255,0);

    line(old\_sx,old\_sy2,screen\_x,screen\_y2); // Draw line from old point to the new point

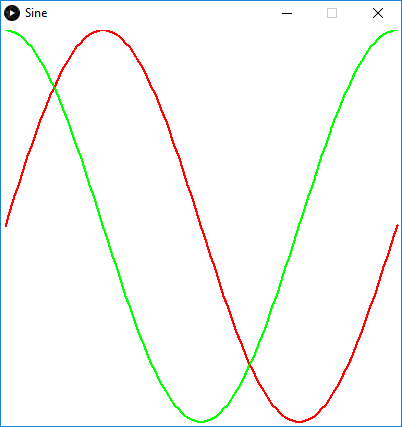
    old\_sx=screen\_x;                                   // The current new points updates old points

    old\_sy1=screen\_y1;

    old\_sy2=screen\_y2;

  } // Repeat for each theta value

}   //Redraw the new screen



**Figure 4:** Plot of *sine()* and *cosine()* functions as a function of theta.

**To Do:** plot the sine value (x-crd) vs the cosine value (y-crd) so as to produce a circle. Make the circle magenta/purple. Take a screen shot of the image produced and add it to a word document. The word document should be called <your\_surname.doc>. The first few lines of the document should contain your name and date. Each image should be titled.

Add comment to put out of action.

 //stroke(255,0,0);

 //line(old\_sx,old\_sy1,screen\_x,screen\_y1);

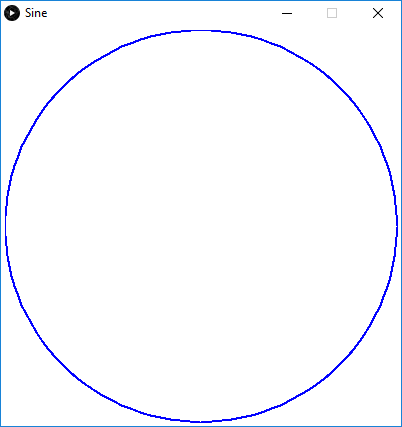
 //stroke(0,255,0);

 //line(old\_sx,old\_sy2,screen\_x,screen\_y2);

Add this.

  stroke(0,0,255);

   line(old\_sy1,old\_sy2,screen\_y1,screen\_y2); // sine vs cosine



**Figure 5:** Plot of *cosine()* vs *sine()* producing an image of the unit circle.

**Part 2:** Plotting data from a disk file using library code.

So far, we have plotted graphs using drawing primitives such as line and ellipse. Processing includes a graph-drawing library that simplifies the plotting of graphs. The graph produced is an instance of a graph object. Using an object-oriented approach allows you to change the properties of the graph and then use the associated methods to draw the graph to the screen. Each graph you produce is a new instance of the graph class. The methods and variables belonging to an object are all accessed using the “.” <dot> operator.

An object (or class) is described by a list of the variables (data) and methods (actions) associated with it. This approach of combining data with the methods is a very popular approach to program design. The class is only the description of an object not an instance of the object itself (just as “float” defines a type of number but it is not an instance until you use it to create a variable e.g. “float x=0;”). You can create an instance (or multiple instances) of an object once, you have created the class. For example, you could define a class to describe a “ball”, and then create multiple instances of the ball class with different properties but common functionality (Tennis ball, Football, Golf ball etc).

You could consider the class as the “cookie cutter” and the instance the “cookie”.

In this section, we will plot population data contained in a CSV (Comma Separated Value) file obtained from the Irish Central Statistics Office.

**To Do:** Create a new Processing project and the code shown below. Download the CSV file from Moodle and copy it to the same folder as your Processing code. Technically, it should be in a subfolder of the project folder called ***data*** but it seems to work alongside the project pde file.

import grafica.\*;

Glpot is the class “cookie cutter”, plot1 an instance of the class “the cookie”

GPlot plot1;

Table table;

void **setup**()

{

  size(540,350);

  table = loadTable("IrishPop.csv", "csv");

  GPointsArray points = new GPointsArray(table.getRowCount());

  for (TableRow row : table.rows())

  {

    float year = row.getFloat(0); // (0) refers to the first column (year)

    float population = row.getFloat(1)/1000000; // (1) is second column (population) millions

    points.add(year,population); // Add new point to the list

  }

  plot1 = new GPlot(this); // Create an instance of the Gplot class called plot1

  plot1.setPos(25, 25);         // Set the plot1 position in window to 25,25

  plot1.setDim(350, 200);       // Set plot size to 350x200

  plot1.setMar(60, 70, 40, 70); // Set margins (bottom,left,top,right)

  plot1.setPointSize(5);        // Sets diameter of points on screen

  plot1.setTitleText("Population change in Ireland");

  plot1.getXAxis().setAxisLabelText("Year");

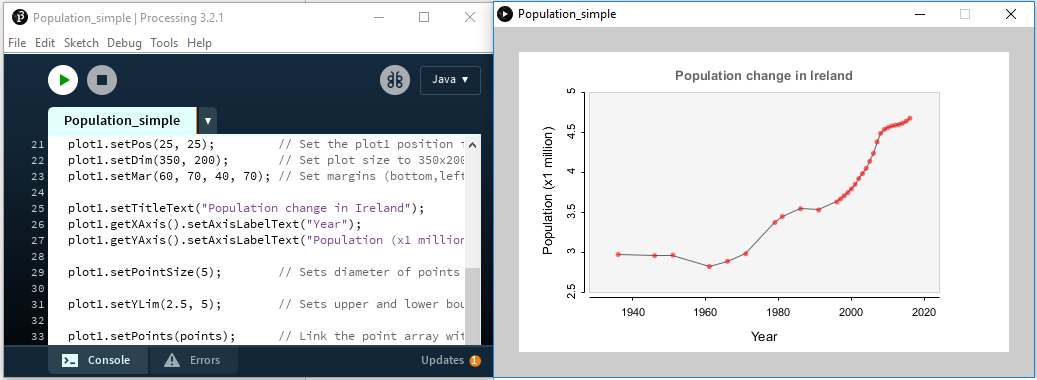
  plot1.getYAxis().setAxisLabelText("Population (x1 million)");

  plot1.setYLim(2.5, 5);        // Sets upper and lower bound of Y-axis

  plot1.setPoints(points);      // Link the point array with the plot

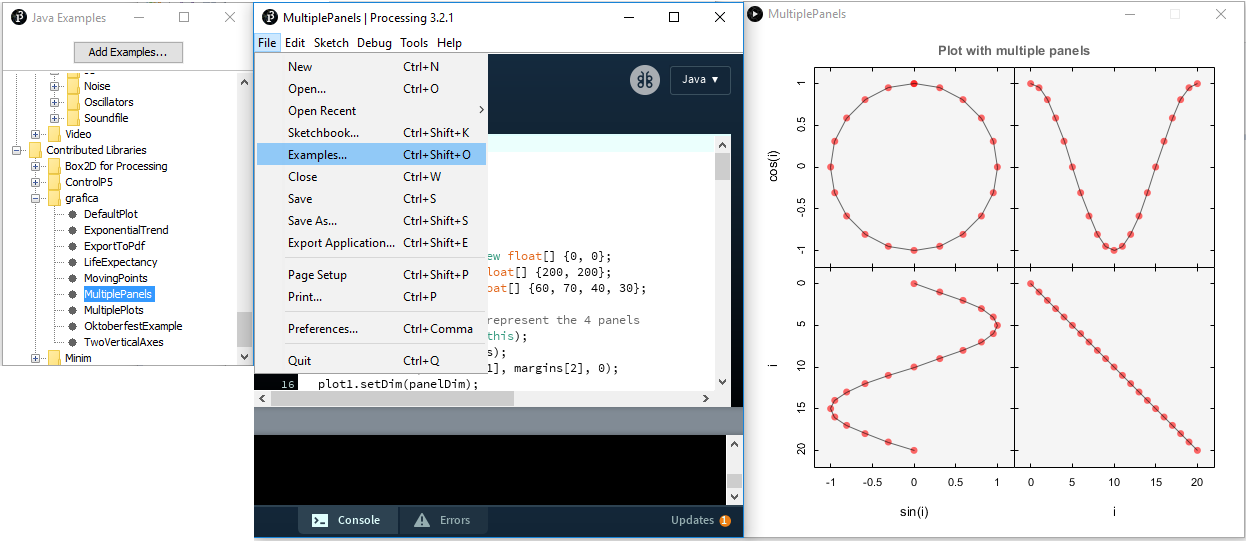
  plot1.defaultDraw();          // Draw the graph

}

****

**Figure 5:** Graph of Irish population produced in Processing using the Grafica library.

**Aside:** When you add a library to Processing it normally comes with some good exampleS to get you started. You could look at the “grafica” examples to see how to do different graphs and to learn some of the extra commands you could use to finesse a graph.

****

**Figure 6:** Exploring the examples provided with a new library.

We now need to add code to plot a second graph over the first showing the rate of change of population (at last some more calculus).

import grafica.\*;

Create a second plot for rate graph

GPlot plot1,plot2;

Table table;

void **setup**()

{

  size(540,350);

Open the csv data file

  table = loadTable("IrishPop.csv", "csv");

  GPointsArray points = new GPointsArray(table.getRowCount());

Create a second array to store rate data.

  GPointsArray rate = new GPointsArray(table.getRowCount());

  for (TableRow row : table.rows())

  {

Create a second array to store rate data.

    float year = row.getFloat(0);

    float population = row.getFloat(1)/1000000;

Add the new point to the list of points.

    points.add(year,population);

  }

  for (int i=1;i<points.getNPoints();i++)

  {

    float year=points.getX(i-1);

Use previous year

Change in years, change in population

    float dyear=points.getX(i)-points.getX(i-1);

    float dpopulation=points.getY(i)-points.getY(i-1);

Calculate rate *d*(population)/*d*(year)

    rate.add(year,1000\*dpopulation/dyear);

  }

  plot1 = new GPlot(this);

  plot1.setPos(25, 25);

  plot1.setDim(350, 200);

  plot1.setMar(60, 70, 40, 70);

  plot1.setTitleText("Population change in Ireland");

  plot1.getXAxis().setAxisLabelText("Year");

  plot1.setYLim(2.5, 5);

  plot1.getYAxis().setAxisLabelText("Population (x1 million)");

  plot1.setPointSize(5);

  plot1.setPoints(points);

  plot1.defaultDraw();

Create a second Gplot instance for the over plot graph.

  plot2 = new GPlot(this);

  plot2.setPos(25, 25);

  plot2.setDim(350, 200);

  plot2.setYLim(-20, 150);

  plot2.setPoints(rate);

  plot2.setLineColor(color(0 ,0, 0));

  plot2.setPointSize(5);

  plot2.setPointColor(color(0,0,255));

  plot2.getRightAxis().setAxisLabelText("Rate 1000/year");

  plot2.getRightAxis().setDrawTickLabels(true); // Make the right axis visible

  plot2.beginDraw();

  plot2.drawRightAxis();

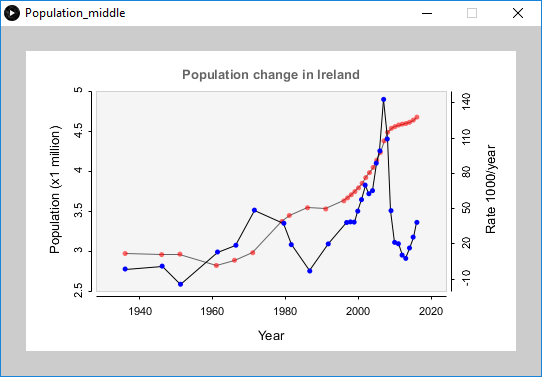
Only draw points, lines and right axis (plot2 over-plots plot1).

  plot2.drawLines();

  plot2.drawPoints();

  plot2.endDraw();

}



**Figure 7:** Grafica used to plot rate of population change in Ireland.

The blue trace in the above graph is a little noisy. It is possible to smooth this data by taking a running average across the rate data set.

rate\_avg[i] = (rate[i-1]+rate[i]+rate[i+1])/3

**To Do:** Add the following code (and modifications) to achieve a 3-point running average, clearly you will need to add the suggested code in the correct place. Add the image to your word document.

  GPointsArray rate\_avg = new GPointsArray(table.getRowCount());

  rate\_avg.add(rate.get(0));

  for (int i=1;i<rate.getNPoints()-2;i++)

  {

    float year=points.getX(i);

    float three\_year\_avg=(rate.getY(i-1)+rate.getY(i)+rate.getY(i+1))/3;

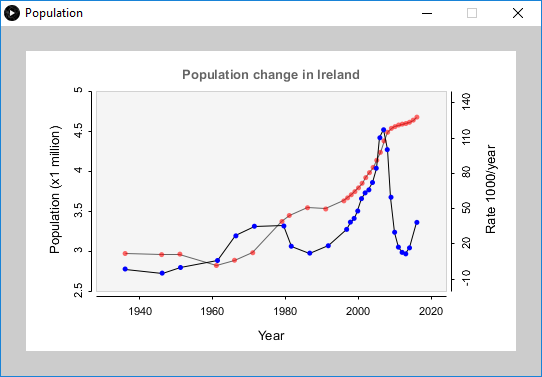
    rate\_avg.add(year,three\_year\_avg);

  }

  rate\_avg.add(rate.get(rate.getNPoints()-1));

  //plot2.setPoints(rate);

  plot2.setPoints(rate\_avg);



**Figure 8:** Three-point average used to smooth rate graph.

**To Do:** Write your own program to produce a graph showing birth and death rates as a function of time for Ireland. The data provided was obtained from the CSO website and the file is available on Moodle (IrishBthDthrate.csv). Add a screen image to your word document.

**Note:** The CSV file has headings on each of three columns. You will need to edit the file and remove these headings, or better refer to them in the code instead of using an Index.

For example, the code to read the first column “Year” would change to read as follows.

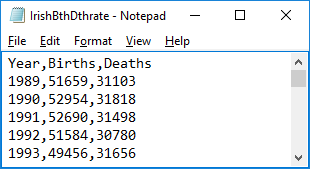
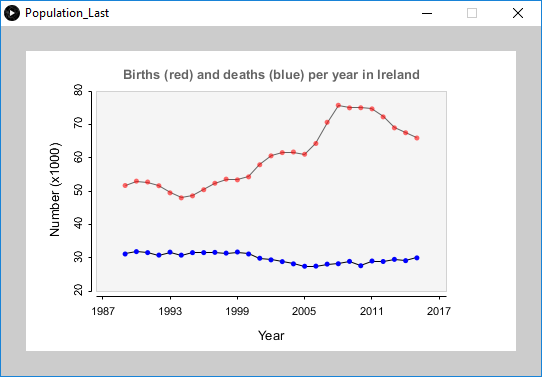
// table = loadTable("IrishBthDthrate.csv", "csv");

   table = loadTable("IrishBthDthrate.csv", "header");

// float year = row.getFloat(0);

   float year = row.getFloat("Year");



**Figure 9:** Sample plot of birth and death rate data.

**Part 3:** Rabbits and Foxes, simple equations with complicated behaviour.

The logistic equation is a simple model that was developed to explain how a population of animals changes with time.

The population of rabbits this year, rnew, depends on the number of rabbits last year, rold, multiplied by a “starvation factor”, where *β* is the susceptibility to starvation and responds negatively to increasing population.

The rabbit population is contained within the range [0,1].

**To Do:** Enter the following program and run it. The programs displays the value of rnew for 20 iterations after an initial 200 iterations to allow the value to settle down.

**Aside:** I have used some Java to show you one method to enter a number from the keyboard in Processing. Processing does not as part of its design like to use console input.

Import java library to allow keyboard input.

import javax.swing.JOptionPane;

void **setup**()

{

size(200,500);

}

void **draw**()

{

 float b,r;

 background(255);

Black writing on a white background.

 fill(0);

Use our keyboard method to read in a value for b the starvation factor.

 b=readFloat("B=");

 r=0.5;

Run through 200 iterations to allow sequence to settle down.

 for(int i=0;i<200;i++)

 {

     r=r\*b\*(1.0-r);

 }

 for(int i=0;i<20;i++)

 {

Evaluate *rnew* and display value (20 iterations)

     r=r\*b\*(1.0-r);

     text(r,10,i\*20);

 }

}

// Routine to read from keyboard

float readFloat(String text)

{

 float f = 0;

 String s = JOptionPane.showInputDialog(null,text, "Enter float:", JOptionPane.QUESTION\_MESSAGE);

“Boiler plate” method to read input from the keyboard.

 try {

      f = Float.parseFloat(s);

     }

 catch(NumberFormatException e)

 {

  println("you did not enter a number!");

 }

 return f;

}

Enter values for b from the keyboard (try 2, 3, 3.5 and 3.8 amongst other values) and convince yourself that you can see period doubling bifurcation.

An easier way to visualise what is happening is to plot the *r* values (Y-axis) against the *b* values (X-axis). Enter the following program and convince yourself you can see the period doubling occurring.

  size(800,500);

  float b,r;

Cycle through b values on interval [1,4]

  for(b=1.0;b<=4.0;b+=0.003)

  {

    // Run forward for 100 stps to allow system to settle

    r=0.5;

    for(int i=0;i<100;i++) r=r\*b\*(1.0-r);

    // Plot r value(s) as a function of b

    for(int i=0;i<600;i++)

Scale r values on interval [0,1] and b values on interval [1,4] to fit on screen.

    {

     r=r\*b\*(1.0-r);

     float screenx=map(b,1.0,4.0,0,width);

     float screeny=map(r,0.0,1.0,height,0);

     point(screenx,screeny);

    }

  }

At what value of *β* does chaotic behaviour begin?

**To Do:** Add a picture of your version of the logistic map to your word document.

**Aside:** A good account of the different regions of behaviour can be found by looking at the Wikipedia page on the “Logistic Map”.

An extension to the Logistic Equation is to consider the effect or predation (foxes) on the animal population. In this case, the rabbit population is governed by the two parameters, α is the susceptibility to starvation and β the fox success rate.

The code to visualise the effect (α,β) has on the rabbit and fox population is shown. The program is seeded using a mouse click with values for α and β on the interval [1,4]. The program plots points on the screen on the interval [0,1] corresponding to *r* (X coordinate) and *f* (Y coordinate). Perhaps surprisingly the values that f and r produced do not always converge on a single value or vary between all possible values but instead are constrained. The figure produced by the code is an attractor showing all the possible values of *f* given an r value.

For now, it is enough to know that very simple equations can be very sensitive to their starting conditions and that they can exhibit very complicated behaviour to model and predict.



Repeat the following 40,000 times.

Print value in a and b on the screen.

Initialise the values for the rabbit and fox success factors.

Method runs once at the start.

Method runs each time the screen is redrawn,

Clear screen white.

Line/point width (2pixels), text size 24, black.

Create an applet window 500x500.

Set a and b values.

float a=3.11,b=3.88;

 void **setup**()

 {

    size(500,500);

    strokeWeight(2);

    textSize(24);

    fill(0);

 }

 void **draw**()

 {

  background(255);

  float r=0.3,f=0.15,rt;

  if(mousePressed)

  {

Use a mouse click to change (a and b) values based on location of mouse pointer in window [1,4].

   a=map(mouseX,0,width, 1,4);

   b=map(mouseY,0,height,1,4);

  }

Convert the values in a and b to a string , to 3 decimal places.

  String sa = nf(a, 1, 3);

  String sb = nf(b, 1, 3);

  text("a:"+sa+" b:"+sb,width-200,20);

  for(int i=0;i<40000;i++)

  {

Calculate new value for r and f using current values, note use of temporary variable rt.

     rt=a\*r\*(1.0-r-f);

     f=b\*r\*f;

     r=rt;

Rescale r and f values to fit on screen.

     float screenx=map(r,0.0,1.0,0,width);

     float screeny=map(f,0.0,1.0,height,0);

Plot a point into screen buffer.

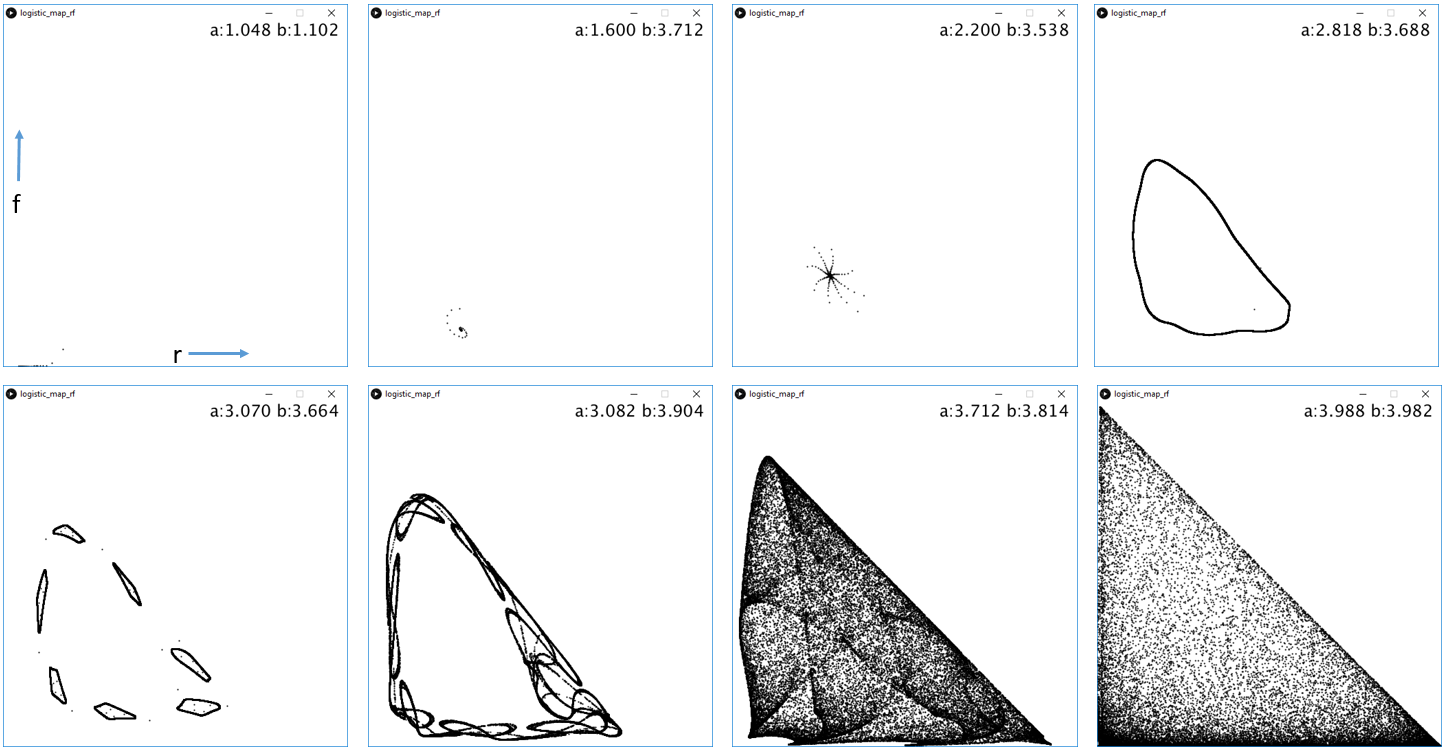
     point(screenx,screeny);

   }

End of draw code segment, causes screen buffer to be displayed.

 }

**To Do:** Add a screen image of an attractor (you liked) produced by your program to the Word document.



**Figure 10:** Some attractors that may be of interest, r (x-axis), f(y-axis) .

**Part 4:** Bouncing a ball (and projectile motion)

In lectures, we considered the forces acting on a ball that act to explain its motion. In summary, these forces were gravity pulling the ball down, inertia (mass) resisting change in the velocity of the ball and drag (air resistance) that resists movement.



*Fgravity=mg* (gravity pulls ball down)

*Fdrag=-kv* (Friction resists motion, *k* constant, *v* velocity)

*FInertia=ma* (Inertia, Newton’s second law, *m* mass, *a* acceleration)

*y* (position of ball at time *t*)

Making the observation that acceleration is rate of change of the rate of change of position,

and substituting gives,

rearranging gives a 2nd order differential equation that fully describes the motion of the ball.

This is a second order equation that we need to rewrite as two first order equations. Observe that velocity is change in position divided by change in time.

1st First order

taking the derivative gives

substituting in the differential equation gives,

2nd First order

Solving the first order equations for *dy* and *dv* gives,

and

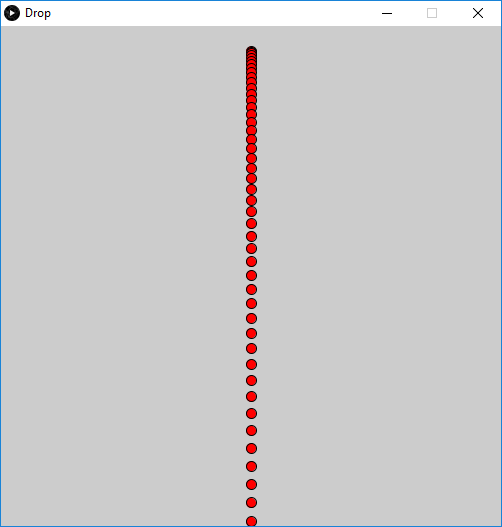
these equations tell us how to change the position and velocity of the ball over a time interval *dt*.

We can provide starting values for the position y=10 and velocity v=0, these are known as the “boundary conditions” and then iterate over time steps of dt.

Starting conditions:

Method to advance ball over time interval *dt*:

Putting these four equations into code can be achieved as follows.

float g=-9.81;

float k=0.2; // Spring constant

float m=0.25; // Mass kg

float dt=0.01; // Time step 50ms

float vy=0; // Initial velocity

float y=0.95; // Initial position

float t=0; // Initial time

void **setup**()

{

  size(500, 500);

}

void **draw**()

{

  vy=vy+(g-((k/m)\*vy))\*dt;

  y=y+(vy\*dt);

  t=t+dt;

  float sx=map(0.5,0,1,0,width);

  float sy=map(y,0,1,height-1,0);

  fill(255,0,0);

  ellipse(sx,sy,10,10);

}

**To Do:** The program as it stands just models a ball falling. Your goal is to realise the following extensions in code (we did it in the lecture). Cross them off one at a time.

1/ Clear the screen each frame so you only see the ball (not its path).

2/ Make the ball bounce by reversing the velocity when the ball is at the bottom of the screen.

3/ Extend the code to include horizontal motion (introduce *x* and *vx*), same equations as for *y* except *g=0*.

4/ Choose boundary conditions so the ball starts with a non-zero horizontal velocity and a zero vertical velocity.

5/ Make the ball bounce on the sides of the window.

Submit your finished ball bounce code via Moodle/VPL. Submit also the Word document you created during the laboratory work.