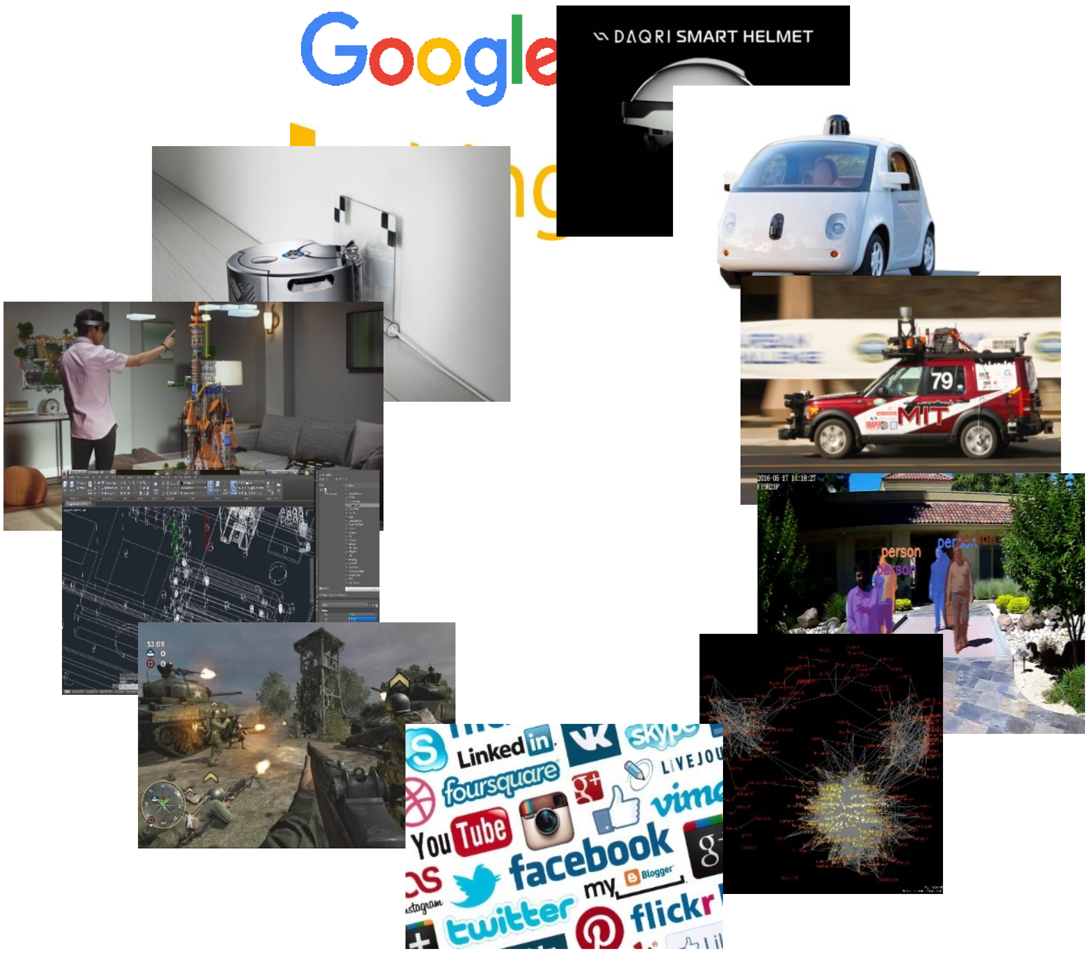
Laboratory 4: Linear Algebra in Action

Introduction:

We have seen in lectures, Linear Algebra provides us with a formal language for abstraction and modelling. As such, it provides an essential element in the development of computer systems and applications across areas ranging from robotics and artificial intelligence to Internet search and social networking.

In this lab we will see how to utilise some of the linear algebra concepts covered during lectures directly within Processing through a linear algebra library provided by Java.



If you have questions outside of the lab time whilst working through the handout you should feel free to post them in the Class Discussion Forum on the CS171 Moodle page. Remember, if you have a question on some aspect of the lab, then someone else most likely has the same question. So asking the question on the forum benefits the whole class! ☺

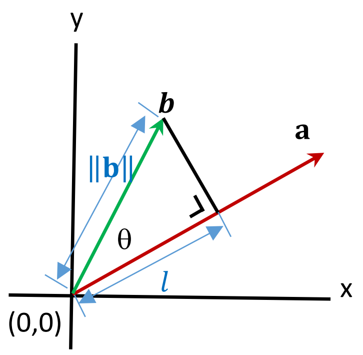
Part 1: A closer look at the dot product.

In a mathematics class, you may see the dot product explained in the following terms.

The dot product of two vectors $\mathbf{a} = \begin{bmatrix} a_x \\ a_y \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} b_x \\ b_y \end{bmatrix}$ is given by,

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y = \|\mathbf{a}\|\|\mathbf{b}\| cos\theta \qquad (1.1)$$

When you first meet the dot product it can seem like a very unusual operation, however it is at the centre of all linear algebra. This is because it has a very important geometric interpretation. To see this, consider the two vectors **a** and **b**, below. Now consider the *projection* of **b** onto **a** is calculated as follows



The scalar projection (sometimes called the component) of **b** onto **a** is the length, l.

$$l = \|\mathbf{b}\| cos\theta \qquad (1.2)$$

Aside: This observation is from the definition of cosine, where l is the length of the *adjacent* side of the triangle and $\|\mathbf{b}\|$ the hypotenuse.

$$sin\theta = \frac{opposite}{hypotenues}, cos\theta = \frac{adjacent}{hypotenues} = \frac{l}{\|\mathbf{b}\|}, tan\theta = \frac{opposite}{adjacent}$$

Combining equation (1.1) and (1.2) gives,

$$l = \frac{a_x b_x + a_y b_y}{\|\mathbf{a}\|} = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|} \qquad (1.3)$$

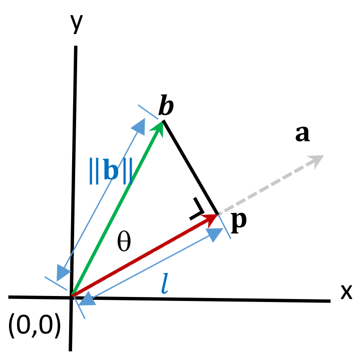
Note: remember that l here is a scalar, i.e. a length, and **not** a vector. In the figure on the top of the next page $\mathbf{p}$ is a vector (of length l) in the direction of **a**. To construct $\mathbf{p}$ we can start with a vector of unit length in the direction of **a** and then multiply it by l.

.Because of their importance in constructing other vectors, we have a special notation for unit vectors where we place a *hat* over the vector. For example the unit vector in the direction of **a** is denoted as $\hat{\mathbf{a}}$**.**

So with this notation we can say that

$$\mathbf{p} = l\hat{\boldsymbol{a}} = \left(\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|}\right)\hat{\mathbf{a}} \qquad (1.4)$$

where we have substituted the right hand side of (1.3) in for l.



Remember from lectures that we can create a unit vector from any vector by simply dividing that vector by its length. For example, the unit vector for $\mathbf{a}$ is given by,

$$\hat{\mathbf{a}} = \frac{\mathbf{a}}{\|\mathbf{a}\|} \qquad (1.5)$$

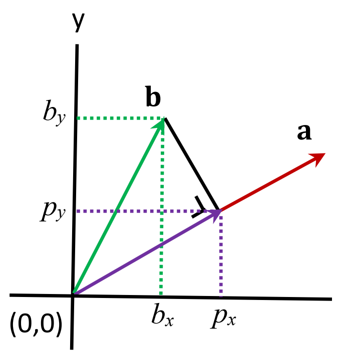
Substituting (1.5) into (1.4) give the project of $\mathbf{b}$ onto $\mathbf{a}$,

$$\mathbf{p} = \left(\frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|}\right)\hat{\mathbf{a}} = (\hat{\mathbf{a}}\cdot\mathbf{b})\hat{\mathbf{a}} \qquad (1.6)$$

where $\mathbf{p}$ is the projection. Again, note here that $\left(\frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|}\right)$ is a scalar, which we are multiplying with $\hat{\mathbf{a}}$ which is the unit vector pointing in the same direction as the original vector, $\mathbf{a}$. So from this you should see that $\mathbf{p}$ is therefore a vector of length $\frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|}$ pointing in the direction of $\mathbf{a}$.

At this point we need to step back from the algebra and reflect on the significance of equation (1.6). This equation provides us with a way of "changing our point of view" of a vector, mathematicians would say "changing our frame of reference". This ability to change our point of view is fundamental to many of the techniques used in robotics, graphics and computer vision today. Every matrix multiplication is a series of dot products.

At the start we described $\mathbf{b}$ in terms of the (X,Y) co-ordinate system, however it becomes possible to build it around new co-ordinate system orientated with $\mathbf{a}$.



The dot product is a mathematical operation at the heart of lots of computer programs and yet it can go un-noticed. Lets take time to look at by writting code to show it in operation.

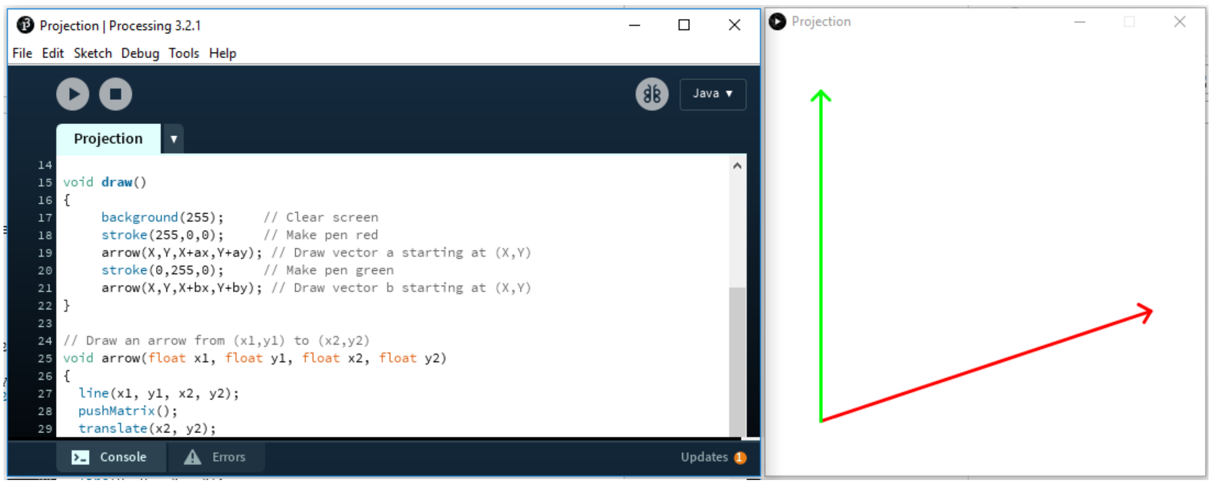
To do: Cut and paste (or enter) the code shown into Processing and run it. The code create variables to store to vectors representing **a** $= \begin{bmatrix} a_x \\ a_y \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} b_x \\ b_y \end{bmatrix}$. An additional pair of variable (X,Y) has been added to define the location of the origin (location of the "tails") of the vectors. The code then draws the vectors.

```
float X=50;     // Origin
float Y=350;

float ax=300;   // Vector a resolved into components
float ay=-100;
float bx=0;     // Vector b resolved into components
float by=-300;

void setup()
{
     size(400,400);         // Create a drawing window
     strokeWeight(3);       // Make pen 3 pixels wide for all lines
}

void draw()
{
     background(255);       // Clear screen
     stroke(255,0,0);       // Make pen red
     line(X,Y,X+ax,Y+ay);   // Draw vector a starting at (X,Y)
     stroke(0,255,0);       // Make pen green
     line(X,Y,X+bx,Y+by);   // Draw vector b starting at (X,Y)
}
```

When run, this code draws red and green lines representing the two vectors. However, vectors are normally drawn as arrows rather than lines (so we can see their direction). Processing does not have an arrow drawing method built in so will have to add it ourselves. Do not be put off by the code, it can just sit at the end of your program and be called upon whenever we need to draw an arrow. The code defines a new method called *arrow()* that can draw an arrow on the screen. Copy and paste the following code so that it sits after the } of the *draw()* method.

```
// Draw an arrow from (x1,y1) to (x2,y2)
void arrow(float x1, float y1, float x2, float y2)
{
  line(x1, y1, x2, y2);
  pushMatrix();
  translate(x2, y2);
  float a = atan2(x1-x2, y2-y1);
  rotate(a);
  line(0, 0, -8, -8);
  line(0, 0, 8, -8);
  popMatrix();
}
```

Replace the two references to the *line()* method with the *arrow()* method. Run your program again and verify that it is correctly displaying your two vectors.


Projection | Processing 3.2.1
File Edit Sketch Debug Tools Help
Java
Projection
Console
Errors
Updates 1
Projection

```

void draw()
{
     background(255);     // Clear screen
     stroke(255,0,0);     // Make pen red
     arrow(X,Y,X+ax,Y+ay); // Draw vector a starting at (X,Y)
     stroke(0,255,0);     // Make pen green
     arrow(X,Y,X+bx,Y+by); // Draw vector b starting at (X,Y)
}

// Draw an arrow from (x1,y1) to (x2,y2)
void arrow(float x1, float y1, float x2, float y2)
{
  line(x1, y1, x2, y2);
  pushMatrix();
  translate(x2, y2);
```

Figure 1: Code to draw two vectors on screen and the run time display.

We can now add our code to evaluate equation (1.6) that calculates the projection of **b** onto **a**. Two lines have been added after the calculation to show the projection on screen. Add this code immediately after the clear screen provided by the *background()* method call in *draw()*.

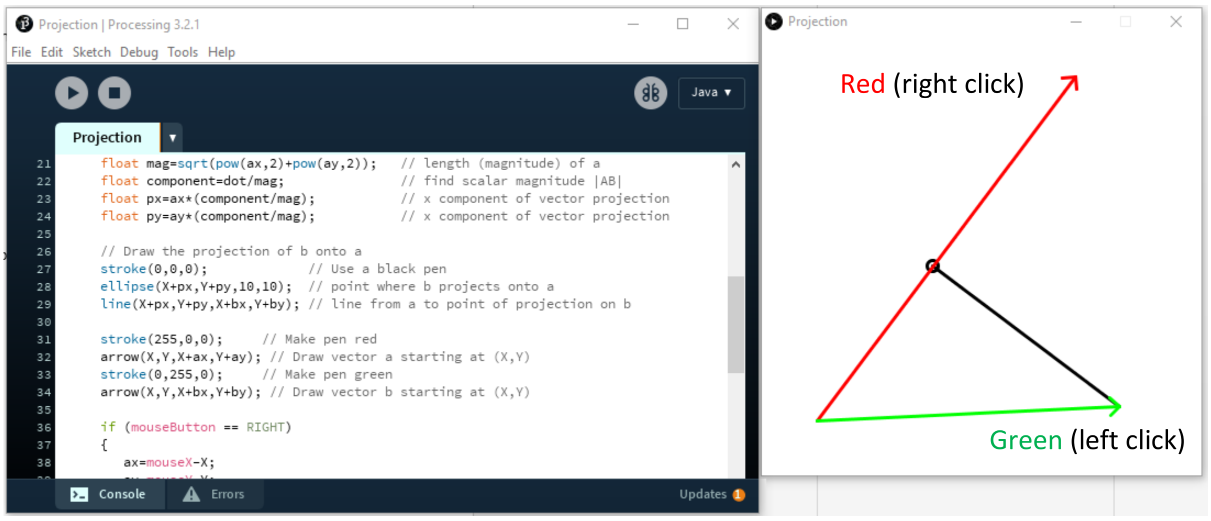
```
// Evaluate equation (1.5)
float dot=(ax*bx)+(ay*by);              // a.b (dot product a and b)
float mag=sqrt(pow(ax,2)+pow(ay,2));    // length (magnitude) of a
float component=dot/mag;                // find length of projection
float axu = ax / mag;                   // compute x component of unit vector a
float ayu = ay / mag;                   // compute y component of unit vector a
float px=component * axu;               // x component of vector projection
float py=component * ayu;               // y component of vector projection

// Draw the projection of b onto a
stroke(0,0,0);                // Use a black pen
ellipse(X+px,Y+py,10,10);    // point where b projects onto a
line(X+px,Y+py,X+bx,Y+by);  // line from a to point of projection on b
```

Run the code and verify that the program correctly draws vectors and projection.

Finally we can add the following code at the end of the *draw()* method just before the }. This will allow you to use the mouse to change position of the vectors using left and right clicks of the mouse.

```
if (mouseButton == RIGHT)
{
     ax=mouseX-X;
     ay=mouseY-Y;
}
if (mouseButton == LEFT)
{
     bx=mouseX-X;
     by=mouseY-Y;
}
```

Run the code and reflect on the significance of the dot product to the calculation.



```
    float mag=sqrt(pow(ax,2)+pow(ay,2));   // length (magnitude) of a
    float component=dot/mag;               // find scalar magnitude |AB|
    float px=ax*(component/mag);           // x component of vector projection
    float py=ay*(component/mag);           // x component of vector projection

    // Draw the projection of b onto a
    stroke(0,0,0);              // Use a black pen
    ellipse(X+px,Y+py,10,10);  // point where b projects onto a
    line(X+px,Y+py,X+bx,Y+by); // line from a to point of projection on b

    stroke(255,0,0);      // Make pen red
    arrow(X,Y,X+ax,Y+ay); // Draw vector a starting at (X,Y)
    stroke(0,255,0);      // Make pen green
    arrow(X,Y,X+bx,Y+by); // Draw vector b starting at (X,Y)

    if (mouseButton == RIGHT)
    {
       ax=mouseX-X;
```

Figure 2: Program with mouse interaction.

To Do: Add code to your program to print the angle between the red line and the length of the projection of the green line onto the red line.

The length of vector is given by

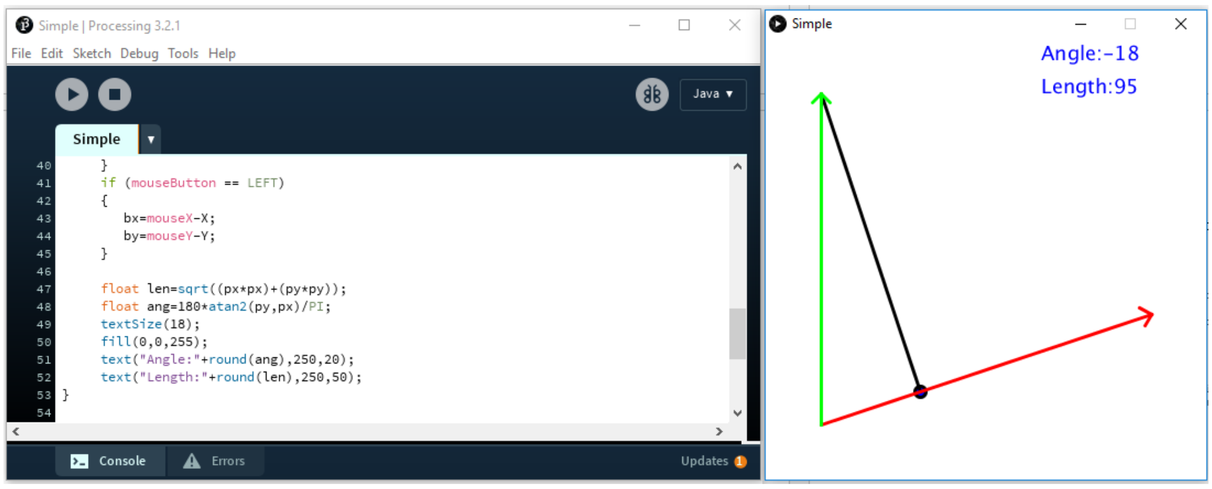
$$|\vec{p}| = \sqrt{px^2 + px^2}$$

and the angle is given by,

$$|\vec{p}| = arctan\left(\frac{py}{px}\right)$$

In processing:

```
    float len=sqrt((px*px)+(py*py));
    float ang=180*atan2(py,px)/PI;
    textSize(18);
    fill(0,0,255);
    text("Angle:"+round(ang),250,20);
    text("Length:"+round(len),250,50);
```



```
    }
    if (mouseButton == LEFT)
    {
       bx=mouseX-X;
       by=mouseY-Y;
    }

    float len=sqrt((px*px)+(py*py));
    float ang=180*atan2(py,px)/PI;
    textSize(18);
    fill(0,0,255);
    text("Angle:"+round(ang),250,20);
    text("Length:"+round(len),250,50);
}
```

Figure 3: Final code showing projection of one vector onto another.

Useful link

http://math.oregonstate.edu/home/programs/undergrad/CalculusQuestStudyGuides/vcalc/dotprod/dotprod.html

Part 2: Importing a library for Matrix mathematics.

We could continue to write our own methods to do advanced mathematics, however this is considered bad practice. It is inefficient because the code already exists in libraries available online which will typically be more robust and have undergone far more testing and scrutiny since it is publicly available code. Code reuse is at the heart of software engineering and it is important that you do it properly. When you do use another programmer's code, it should, where possible be kept in a separate library. You should always acknowledge and document this imported code. Passing code off as yours that is not your own is plagiarism.

Consider the following pair of simultaneous equations. There is one value of *(x,y)* that makes both equations true.

$$3x + 2y = 3 \tag{2.1}$$

$$-2x - y = -1 \tag{2.2}$$

In the past, you may have solved this pair of equation, by solving one equation (2.1) for y and then substituting the result into the other equation.

Rewriting (2.1) gives,

$$y = \frac{3 - 3x}{2} \tag{2.3}$$

Substituting (2.3) into (2.2) gives,

$$-2x - \frac{3 - 3x}{2} = -1 \tag{2.4}$$

Multiplying (2.4) across by 2 gives,

$$-4x - 3 + 3x = -2 \tag{2.5}$$

Solving (2.5) for *x* gives,

$$x = -1 \tag{2.6}$$

Substituting (2.6) into (2.1) gives,

$$3(-1) + 2y = 3 \tag{2.7}$$

Solving (2.7) for y gives,

$$y = 3$$

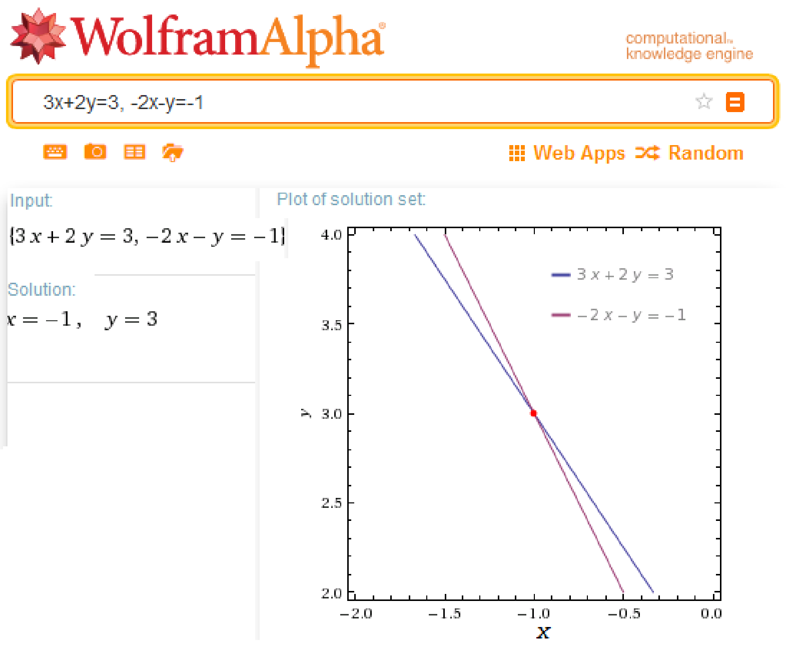
Check *(-1,3)* satisfies both (2.1) and (2.2) simultaneously,

$$3(-1) + 2(-3) = 3\; True$$

$$-2(-1) - (-3) = -1\; True$$

Q.E.D. (*quod erat demonstrandum*, *"which was to be demonstrated"*)

The approach we used to solve the equation involve symbolic manipulation. In practice, this is very difficult to implement on the computer. Knowledge engines such as *Wolfram Alpha* and mathematics software such as *Maple* can do it when requested. We can use Wolfram Alpha to check our answer.



3x+2y=3, -2x-y=-1

Input:

$\{3x + 2y = 3, -2x - y = -1\}$

Solution:

$x = -1, \quad y = 3$

Figure 4: Using Wolfram Alpha to check answer.

Using symbolic manipulation to solve equations works but it is not scalable. You would have to do a lot of work to solve four equations containing four unknowns.

At this point numerical matrix methods come to our rescue.

Writing equations (2.1) and (2.2) in matrix as follows,

$$\begin{bmatrix} 3 & 2 \\ -2 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 3 \\ -1 \end{bmatrix} \tag{2.8}$$

Making the observation that,

$$\mathbf{A} = \begin{bmatrix} 3 & 2 \\ -2 & -1 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} \tag{2.9}$$

Then

$$\mathbf{Ax} = \mathbf{b} \tag{2.10}$$

Solving for *X* gives,

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \tag{2.11}$$

Although algorithms to compute the inverse of a matrix are not covered in this course, in case you are interested, Appendix 1 at the end of this lab sheet provides some explanation on this point. You will cover this in far more detail in your linear algebra course. In both this and next weeks' labs we will use a Java matrix library called JAMA to compute the inverse (along with many other operations).

To Do: Add the Jama matrix library top your Processing project. First, create a new Processing project called Lab4a.pde. This project should be saved on your *X:\* drive, possibly in a sub folder of your making called *CS171*. Remember Processing will automatically put the *Lab4a.pde* in a folder called *Lab4a*. Then using the Firefox browser add the Jama matrix library as follows...

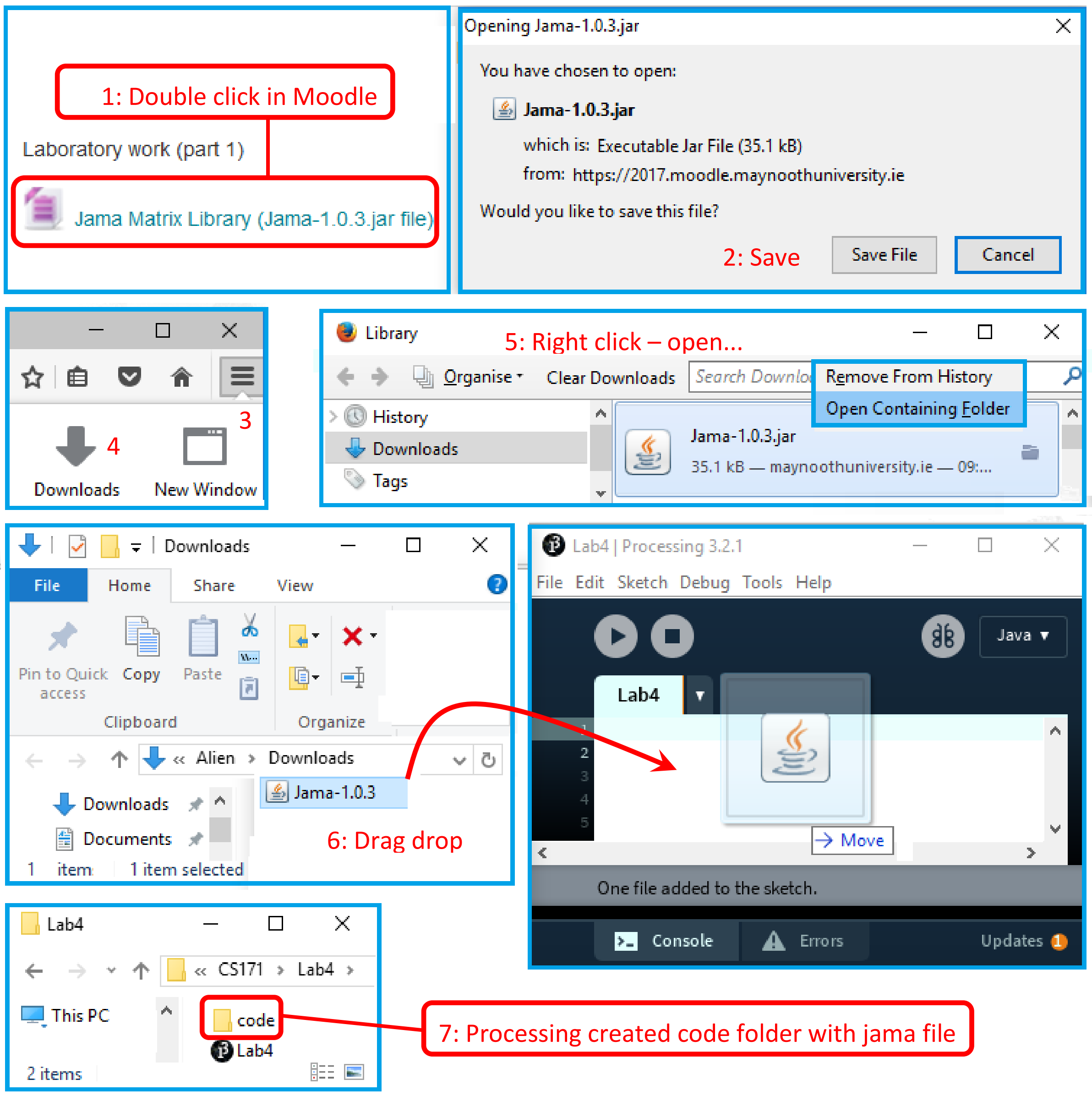
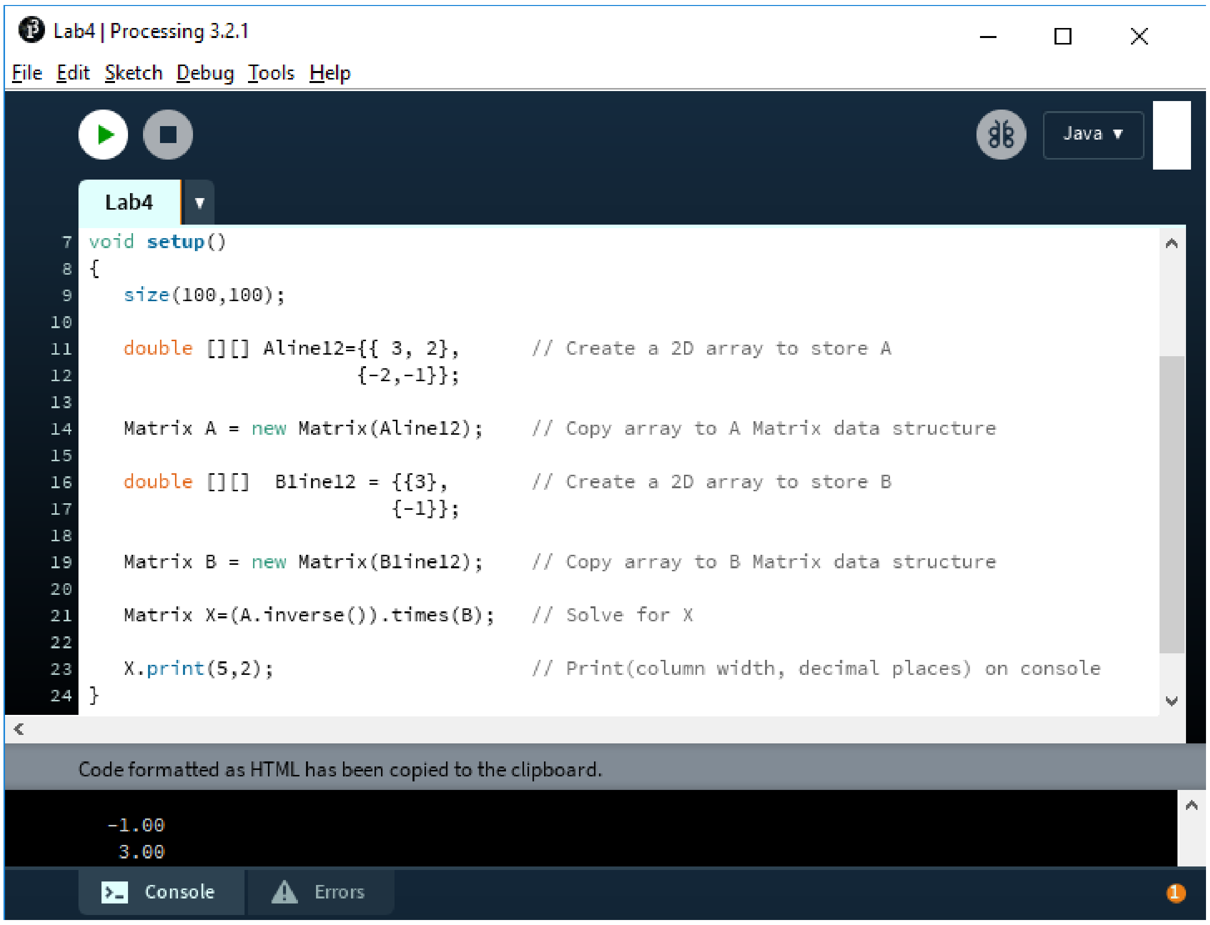


Figure 5: Sequence adding Jama jar file to your Processing project.

To Do: Enter the flowing code that solves the same matrix problem as outlined earlier. Run the program.

```
// import Jama.*;
// Solve 3x+2y=3
//      -2x-y=-1
// AX=B
// X=InvA B

void setup()
{
   size(100,100);

   double [][] Aline12={{ 3, 2},       // Create a 2D array to store A
                        {-2,-1}};

   Matrix A = new Matrix(Aline12);     // Copy array to A Matrix data structure

   double [][]  B1ine12 = {{3},        // Create a 2D array to store B
                           {-1}};

   Matrix B = new Matrix(B1ine12);     // Copy array to B Matrix data structure

   Matrix X=(A.inverse()).times(B);    // Solve for X

   X.print(5,2);                       // Print(column width, decimal places) on console
}
```

Your run time display should look similar to that shown.



```
void setup()
{
   size(100,100);

   double [][] Aline12={{ 3, 2},       // Create a 2D array to store A
                        {-2,-1}};

   Matrix A = new Matrix(Aline12);     // Copy array to A Matrix data structure

   double [][]  B1ine12 = {{3},        // Create a 2D array to store B
                           {-1}};

   Matrix B = new Matrix(B1ine12);     // Copy array to B Matrix data structure

   Matrix X=(A.inverse()).times(B);    // Solve for X

   X.print(5,2);                       // Print(column width, decimal places) on console
}
```

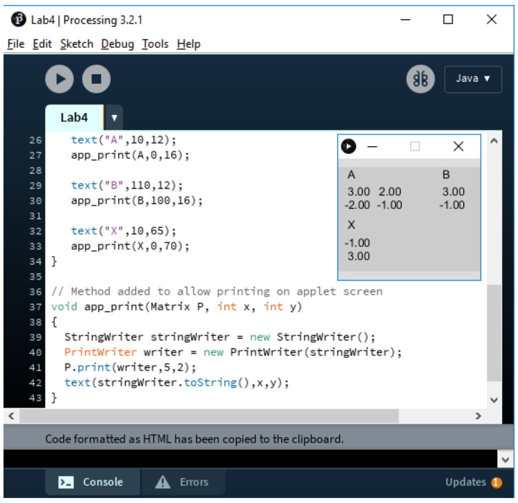
Code formatted as HTML has been copied to the clipboard.

```
   -1.00
    3.00
```

Figure 6: Answer displayed on console screen.

If you would like to see the matrices displayed in the applet window then you could add the following method to do this.

```
// import Jama.*;
// Solve 3x+2y=3
//      -2x-y=-1
// AX=B
// X=InvA B

import java.io.StringWriter;

void setup()
{
   size(150,110);
   fill(0,0,0);

   double [][] Aline12={{ 3, 2},       // Create a 2D array to store A
                        {-2,-1}};

   Matrix A = new Matrix(Aline12);    // Copy array to A Matrix data structure

   double [][]  Bline12 = {{3},       // Create a 2D array to store B
                           {-1}};

   Matrix B = new Matrix(Bline12);    // Copy array to B Matrix data structure

   Matrix X=(A.inverse()).times(B);   // Solve for X

   text("A",10,12);
   app_print(A,0,16);

   text("B",110,12);
   app_print(B,100,16);

   text("X",10,65);
   app_print(X,0,70);
}

// Method added to allow printing on applet screen at (x,y)
void app_print(Matrix P, int x, int y)
{
  StringWriter stringWriter = new StringWriter();
  PrintWriter writer = new PrintWriter(stringWriter);
  P.print(writer,5,2);
  text(stringWriter.toString(),x,y);
}
```



```
   text("A",10,12);
   app_print(A,0,16);

   text("B",110,12);
   app_print(B,100,16);

   text("X",10,65);
   app_print(X,0,70);
}

// Method added to allow printing on applet screen
void app_print(Matrix P, int x, int y)
{
  StringWriter stringWriter = new StringWriter();
  PrintWriter writer = new PrintWriter(stringWriter);
  P.print(writer,5,2);
  text(stringWriter.toString(),x,y);
}
```

```
A                B
 3.00  2.00       3.00
-2.00 -1.00      -1.00
X
-1.00
 3.00
```

Code formatted as HTML has been copied to the clipboard.

Figure 7: Full information on an applet window.

Exercise 1 : Solving a 3 equation system

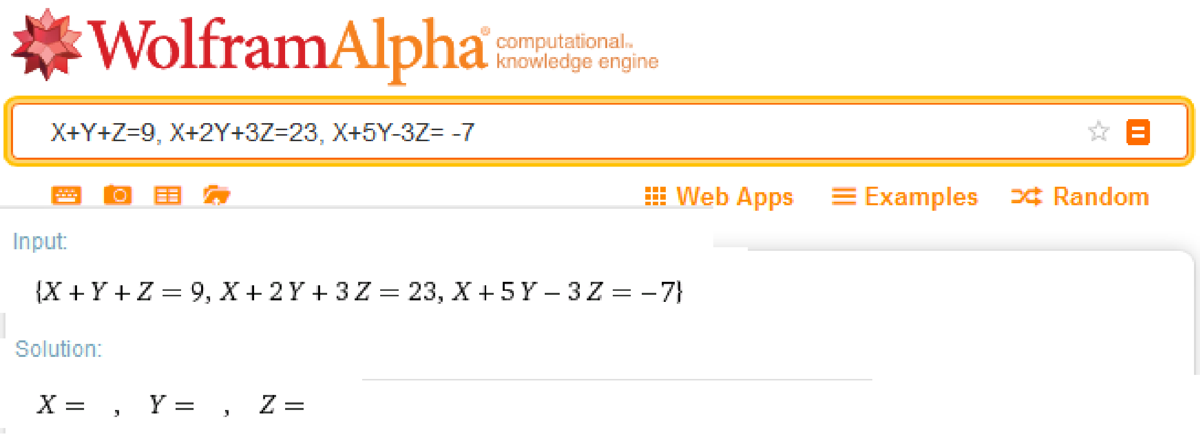
Modify the code you have developed so far to solve the following simultaneous equations.

$$x + y + z = 9$$

$$x + 2y + 3z = 23$$

$$x + 5y - 3z = -7$$

You should upload your resulting code using the “Exercise 1” submission link in the “Linear Algebra in Action” section on the CS171 page.



X+Y+Z=9, X+2Y+3Z=23, X+5Y-3Z= -7

Input:

$\{X + Y + Z = 9, X + 2Y + 3Z = 23, X + 5Y - 3Z = -7\}$

Solution:

$X =$, $Y =$, $Z =$

Figure 8: Check your answer using Wolfram Alpha. The result has been removed in figure.

JAMA Library Cheatsheet

The CS171 moodle page provides a link to the documentation for the JAMA library that will give you full details of all of the functionality that it provides. Although this site is very comprehensive it may be a little overwhelming. The list below provides a more condensed explanation of the functionality that is most relevant to the topics covered during lectures.

Operation & notes	Mathematical Example	JAMA commands

Creating a matrix
 	
Note: Creating a `Matrix` is a two step process. First we specify the numbers that will make up the matrix (i.e. `aelems`), and then we create a `Matrix` around those numbers
	$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$	`double [][] aelems = {{1,2},`
		
		`{3,4}};`
		
		`Matrix A = new`
		`Matrix(aelems);`

Creating a vector
 	
Note: The `Matrix` type is used to represent both vectors and matrices, where a vector is simply a matrix with just one column.
	$\mathbf{v} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$	`double [][] aelems = {{1},`
		`{2}};`
		
		`Matrix v = new`
		`Matrix(aelems);`

Scalar multiplication	$\mathbf{u} = s\mathbf{v}$	`Matrix u = v.times(s);`
Transpose	$\mathbf{v} = \mathbf{u}^T$	`Matrix v = u.transpose();`
Matrix multiplication	$\mathbf{b} = \mathbf{A}\mathbf{x}$	`Matrix b = A.times(x);`

Element access
 	
Note: To access individual elements of a matrix or vector we use the `get` operation, passing in the indices of the row and column of the element we wish to access. ***Remember row and column indices start at 0 (not 1).***
	$x = \mathbf{A}_{i,j}$
	
	Here $\mathbf{A}_{i,j}$ refers to the element in the i^{th} row and the j^{th} column.
	
	Example:
	To access the highlighted element
	$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ \mathbf{3} & 4 \end{bmatrix}$
	$x = \mathbf{A}_{1,0}$
		`double x = A.get(i,j);`
		
		`Example:`
		
		`// x will now store the`
		`value 3`
		`double x = A.get(1,0);`

Inner product
Note: A slightly tricky bit here is that `ut.times(v)` returns a `Matrix` with one element in it. This is why we need the last line of code i.e. to extract the element.
	$\mathbf{u} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$
	
	$a = \mathbf{u}^T\mathbf{v}$
		`// See "Creating a vector"`
		`above to`
		`// see how to create u and v`
		
		`Matrix ut = u.transpose();`
		`Matrix amat = ut.times(v);`
		`double a = amat.get(0,0);`

Matrix Inverse	$\mathbf{B} = \mathbf{A}^{-1}$	`Matrix B = A.inverse();`
Vector Length **Note:** To compute the length of a vector we use the `norm2()` method from the JAMA library. The reason for the name "`norm2`" is that it is the formal name that mathematicians use for vector length.	$\|\|\mathbf{u}\|\|$	`double length = u.norm2();`

IMPORTANT NOTE ABOUT double vs. float

If you take a look at the vector length code above, you might have notices the length is of is a variable of type double:

```
double length = u.norm2();
```

The reason for this is that the people that wrote the JAMA library implemented all of their methods that return scalars to have that type. Both double and float can be used to store floating point numbers in Java, however the difference is that double allows for much larger numbers to be stored. This is why the JAMA designers chose double.

What this means is if try to assign a variable of type double to a float like this:

```
// ERROR: norm2 returns a double
float length = u.norm2();
```

you will get an error. The reason for the error is that Processing knows that a double might not *fit* in a float so it tries to stop you doing this.

In some situations however you will want to assign a double value to a float. To do this we just need to let Processing know that we are aware of what we are doing. In programming we call this *casting* which we can do as follows:

```
// No error. We have explicitly asked for the double returned by norm2 to
// be converted to a float. Everyone's happy! ☺
float length = (float) u.norm2();
```

Part 3: Porting our first program to use JAMA

Given what you know about the JAMA library and how it can be used to directly implement linear algebra in code, we will now return to our first program to see if we can make JAMA do some of the heavy lifting for us! If we look at the code that computed the projection of **b** onto **a**, we notice that we had to write all of the vector operations (e.g. dot product, length, etc.) ourselves:

```
// Evaluate equation (1.5)
float dot=(ax*bx)+(ay*by);               // a.b (dot product a and b)
float mag=sqrt(pow(ax,2)+pow(ay,2));     // length (magnitude) of a
float component=dot/mag;                 // find scalar magnitude |AB|
float px=ax*(component/mag);             // x component of vector projection
float py=ay*(component/mag);             // y component of vector projection

// Draw the projection of b onto a
stroke(0,0,0);              // Use a black pen
ellipse(X+px,Y+py,10,10);   // point where b projects onto a
line(X+px,Y+py,X+bx,Y+by);  // line from a to point of projection on b
```

Exercise 2: Alter the dot product program to use JAMA.

Exercise 2.1

The code listing on the next page provides a starting point for using the JAMA library to achieve the same functionality as the first program in part 1 of the lab. Your task is to replace the comments labelled “// STEP 1”, “// STEP 2”, and “// STEP 3”, in the draw function with the necessary code for computing the projection of b onto a **using the JAMA library**.

First, create a new Processing project called Lab4b.pde. This project should be saved on your *X:\* drive, possibly in a sub folder of your making called *CS171*. Remember Processing will automatically put the *Lab4b.pde* in a folder called *Lab4a*. Then using the Firefox browser add the Jama matrix library in the same manner as in Exercise 1.

Exercise 2.2

Now that you have the program implemented using the JAMA library, extend the functionality of the program such that it draws a third vector, to **c**, of equal length to **a**, but at 90 degrees counter-clockwise to **a.** You should add this code by replacing the comment labelled // STEP 4

Hint: In order to compute a vector at 90 degrees to a given vector, think about some obvious examples. For example if we start with the vector $\begin{bmatrix}1\\0\end{bmatrix}$ we know that it maps to $\begin{bmatrix}0\\1\end{bmatrix}$. If you continue with this i.e. rotating by 90 degrees each time you get the sequence:

$$\begin{bmatrix}1\\0\end{bmatrix} \to \begin{bmatrix}0\\1\end{bmatrix} \to \begin{bmatrix}-1\\0\end{bmatrix} \to \begin{bmatrix}0\\-1\end{bmatrix} \to \begin{bmatrix}1\\0\end{bmatrix}$$

The question you need to answer is, what is the general rule here in terms of what is the mapping that should be applied to $\begin{bmatrix}x\\y\end{bmatrix}$ to rotate them by 90 degrees. We will learn more about these types of transformations for next week’s lab!

Exercise 2.3

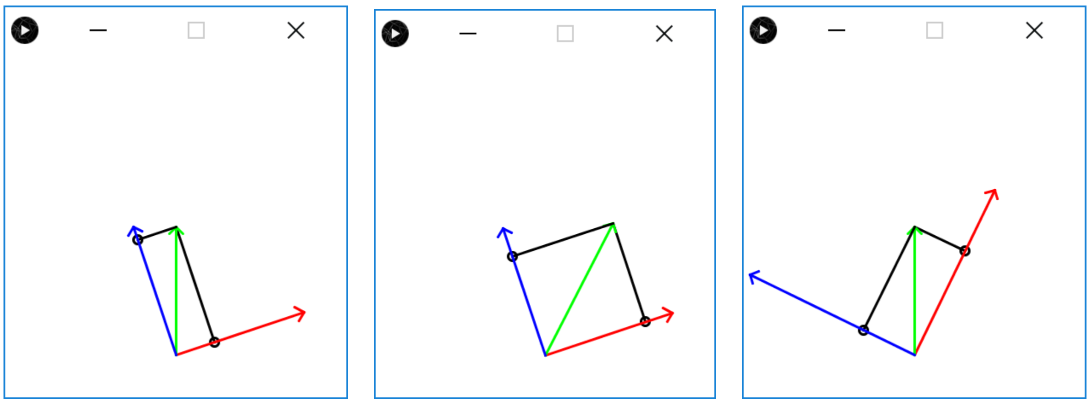
Finally, add the necessary code to also draw the projection of **b** onto the vector introduced in Part 2.

Some example screenshots of the final program are show after the source code below.

You should upload your resulting code using the “Exercise 2” submission link in the “Linear Algebra in Action” section on the CS171 page.

```
float X=200;    // Origin : Note we have now centred the origin in the X-direction
float Y=350;

float ax=300;  // Vector a resolved into components
float ay=-100;
float bx=0;    // Vector b resolved into components
float by=-300;

Matrix a;
Matrix b;

void setup()
{
     size(400,400);      // Create a drawing window
     strokeWeight(3);    // Make pen 3 pixels wide for all lines

     double [][] anums = {{ax},
                          {ay}};

     double [][] bnums = {{bx},
                          {by}};

     a = new Matrix(anums);
     b = new Matrix(bnums);
}


void draw()
{
     background(255);     // Clear screen


     // Evaluate equation (1.5)
     // STEP1: Insert code here that computes a_unit (i.e. the unit vector in the
     // direction of a

     // STEP2: Insert code here to compute the dot product of b and a_unit

     // STEP3: Insert code here to compute the vector p using equation 1.5 above


     float px = (float)p.get(0,0);
     float py = (float)p.get(1,0);
     float ax = (float)a.get(0,0);
     float ay = (float)a.get(1,0);
     float bx = (float)b.get(0,0);
     float by = (float)b.get(1,0);

     // Draw the projection of b onto a
     stroke(0,0,0);              // Use a black pen
     ellipse(X+px,Y+py,10,10);   // point where b projects onto a
     line(X+px,Y+py,X+bx,Y+by);  // line from a to point of projection on b

     stroke(255,0,0);     // Make pen red
     arrow(X,Y,X+ax,Y+ay); // Draw vector a starting at (X,Y)
     stroke(0,255,0);     // Make pen green
     arrow(X,Y,X+bx,Y+by); // Draw vector b starting at (X,Y)

     // STEP 4. Insert code here to add a new vector at 90 degrees to the vector a

     // STEP 5. Insert code here to compute and draw the projection of b onto c

     if (mouseButton == RIGHT)
     {
        a.set(0,0,(double)mouseX-X);
        a.set(1,0,(double)mouseY-Y);
     }
     if (mouseButton == LEFT)
     {
        b.set(0,0,(double)mouseX-X);
        b.set(1,0,(double)mouseY-Y);
     }
}

// Draw an arrow from (x1,y1) to (x2,y2)
void arrow(float x1, float y1, float x2, float y2)
{
  line(x1, y1, x2, y2);
  pushMatrix();
  translate(x2, y2);
  float a = atan2(x1-x2, y2-y1);
  rotate(a);
  line(0, 0, -8, -8);
  line(0, 0, 8, -8);
  popMatrix();
}
```

Example Screenshots:



Appendix 1: How are inverses calculated?

There are standard algorithms for finding the inverse of a matrix, for example

Making the observation that,

$$\mathbf{AA}^{-1} = \mathbf{I} \tag{2.12}$$

$$\mathbf{IA}^{-1} = \mathbf{A}$$

Using matrix row operations we can add/subtract any lines, or multiply across by any number and attempt to get the identity on the left hand side.

$$\left[\begin{array}{cc:cc} 3 & 2 & 1 & 0 \\ -2 & -1 & 0 & 1 \end{array}\right]$$

row 2=(2 x row 1)+(3 x row2)

$$\left[\begin{array}{cc:cc} 3 & 2 & 1 & 0 \\ 0 & 1 & 2 & 3 \end{array}\right]$$

*row1=row1-(2*row2)*

$$\left[\begin{array}{cc:cc} 3 & 0 & -3 & -6 \\ 0 & 1 & 2 & 3 \end{array}\right]$$

Divide row1 by 3,

$$\left[\begin{array}{cc:cc} 1 & 0 & -1 & -2 \\ 0 & 1 & 2 & 3 \end{array}\right]$$

So

$$\boldsymbol{A}^{-1} = \begin{bmatrix} -1 & -2 \\ 2 & 3 \end{bmatrix}$$

Substituting in (2.11) gives,

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} = \begin{bmatrix} -1 & -2 \\ 2 & 3 \end{bmatrix}\begin{bmatrix} 3 \\ -1 \end{bmatrix} = \begin{bmatrix} (-1 * 3) + (-2 * -1) \\ (2 * 3) + (3 * -1) \end{bmatrix} = \begin{bmatrix} -1 \\ 3 \end{bmatrix}$$

QED...

Aside: another approach that works with 2x2 matrices is to use the determinant as follows,

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

where,

$$determinant(\mathbf{A}) = ad - bc$$

and,

$$adjoint(\mathbf{A}) = \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

$$A^{-1} = \begin{bmatrix} 3 & 2 \\ -2 & -1 \end{bmatrix}^{-1} = \frac{1}{-3+4}\begin{bmatrix} -1 & -2 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ -2 & -3 \end{bmatrix}$$