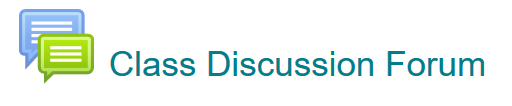
# Laboratory 7: Rigid Body Transformations

## Introduction:

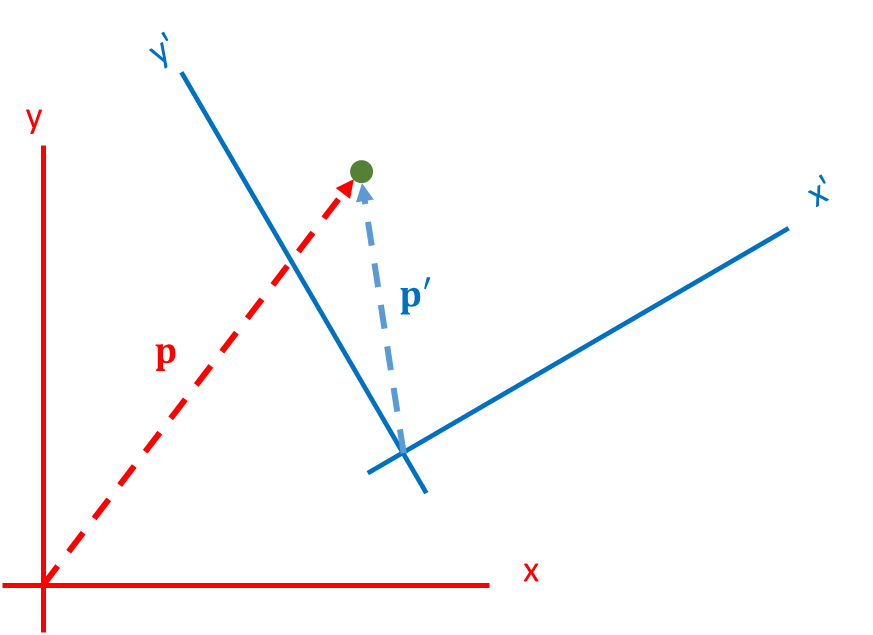
Last week’s lab introduced you to various linear algebra concepts and showed how they could be used in Processing using the JAMA library. We saw how using some basic operations from linear algebra we could compute the projection of an arbitrary vector onto other vectors. In this lab we will build on these ideas to see how these operations of vector projection are the fundamental components of an important class of *transformations* known as *Rigid Body Transformations.*

|  |  |
| --- | --- |
| Image result for atlas robot   1. ATLAS Humanoid Robot. Boston Dynamics Incorporated. | http://www2.djicdn.com/uploads/nav_link/cover/666/size_1000_540_a34ab5c95915a83b150d5157c21ae613.png   1. DJI Mavic Quadrotor. |
| Image result for HTC Vive   1. HTC Vive Virtual Reality Headset. |
| **Figure 1.** From humanoid robots to autonomous drones to virutal reality headsets, the ability to model the motion of computer systems and to understand how the world moves around them is critical to their operation. | |

Remember, if you have questions outside of the lab time whilst working through the handout you should feel free to post them in the  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! ☺

## Coordinate Transformations:

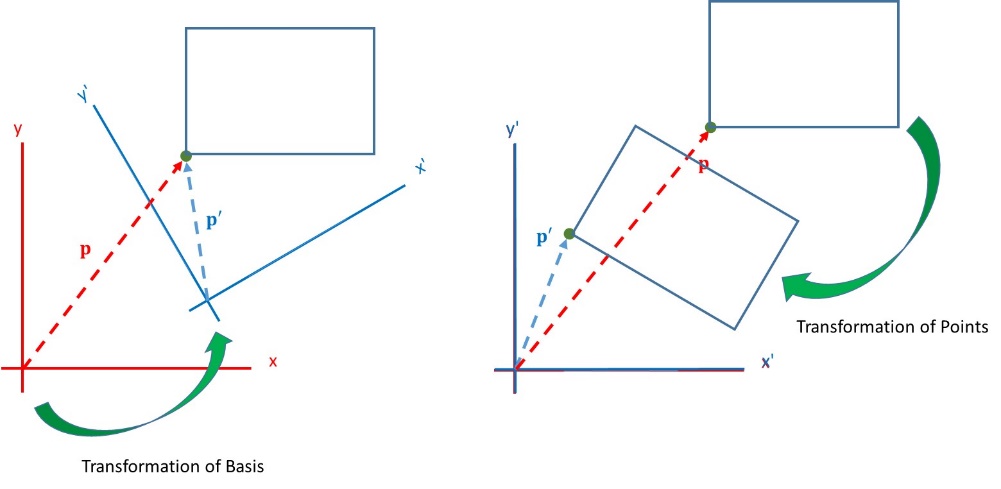
A coordinate transformation is a mathematical mapping or function that takes as input coordinates of a point relative to one coordinate system, and provides as output the corresponding coordinates of the point in a new coordinate system. For example, the figure below shows a point in the plane along with two different coordinate systems. Notice how depending on the particular coordinate system we use (i.e. the blue or the red), the point’s coordinates will be different. To make this explicit, the figure also shows the vectors that join the origin of each coordinate system to the point. In this example, in the green point will have different coordinates depending on whether the red coordinate system the or the blue coordinate system is used.



**Rigid Body Transformations**

As you will see in your linear algebra modules (and a number of the computer science modules in later years), there are a variety of types of coordinate transformations in terms of the types of mapping that they perform on the input space. A particularly important class of coordinate transformation are known as ***rigid body* transformations**. These transformations get their name from the fact that they preserve lengths i.e. object in the output space will be the same size and shape as they were in the input space. The reason that rigid body transformations are so important is that they capture how objects move around in the world. For example, when a car drives through the world, only it’s position and orientation, (also known its pose), change. Attributes such as the car size and shape do not change.

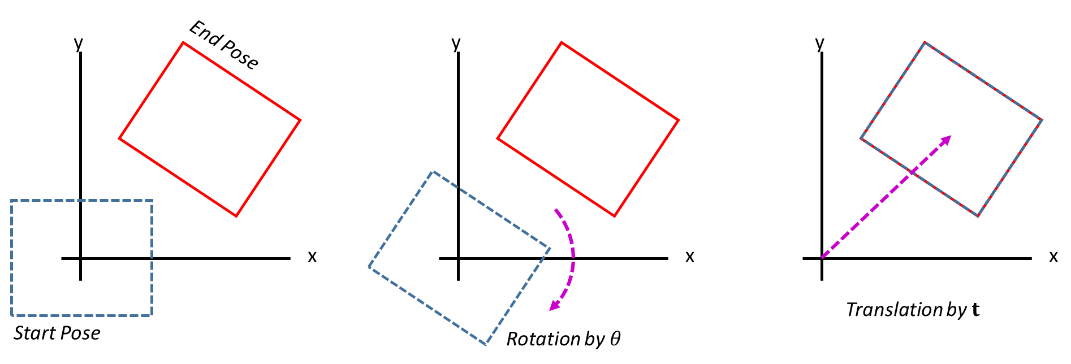
The image below shows two alternative approaches to interpreting the effect of a rigid body transformation: one where the world is fixed and the coordinate system moves, and the other where the coordinate system is fixed and the world moves relative to it. Depending on the application one or the other interpretation will be more appropriate.



For example, when modelling the motion of a mobile robot (i.e. it’s position and orientation), we can model the robot as a coordinate system attached to the robot which moves through the world. This would correspond to the figure on the left above. On the other hand, if modelling how the world would be perceived by a sensor attached to the robot we would use the interpretation on the right.

## Using Linear Algebra to Create Rigid Body Transformations

As you will have seen in lectures last week, every rigid body transformation can be broken down into a **rotation** followed by a **translation**. To understand this idea, we can take a look at the figure on the left below. Here we see an object in two poses, a start pose and an end pose. Note that pose is just a word that people use to mean position + orientation. In the middle figure we see the first object rotated by some angle in an anti-clockwise direction around the origin. In the figure on the right we then translate the resulting object by some translation vector, . Notice how applying these two operations in the correct proportions we align the first object with the second one. Understanding this point is the key to understand rigid body transformations.



So let’s try to figure out what the corresponding linear algebra operations are for a rigid body transformation. To do this we first simplify our problem by realising that any object can be considered as a collection of points, and if we can compute how each individual point moves under the rigid body transformation then this implies that we can compute how the entire object moves.

To understand this point, we can again look at the example of apply a rigid body transformation to the rectangle on the left above. In order to compute the output rectangle, we only need to compute the output locations of the corner points (since the lines between them will stay as straight lines).

Starting with the rotation component of the transformation, we saw in lectures that the lab from last week showed us exactly how to achieve this. The figure below on the left shows an example taken from last week’s lab where we have shown the unit vectors in the direction of the a and c vectors. Note how these two vectors are at right angles to each other, and as such, form a new coordinate system which is rotated by an angle relative to the first (i.e. black) coordinate system. This rotation angle is illustrated in the figure on the right.

|  |  |
| --- | --- |
|  |  |

As we have seen in lectures the projection of b onto these unit vectors provides us with the coordinates of b in this new coordinate systems. These values are given by the inner product of b with both unit vectors. Both the dot products can be captured in a single matrix multiplication

(1)

For a complete explanation of this matrix multiplication you should review the slides from Lecture 5. The above matrix rotates the coordinate frame counter-clockwise by , which corresponds to a rotation of points clockwise by . More often the convention will be that a positive corresponds to a counter-clockwise rotation. With this second convention the sign of is reversed leading to the rotation matrix of the form,

(2)

The second component of the rigid body transformation is to translate the rotated point by a vector , resulting in the complete transformation,

where we have relabelled the input vector coordinates as x and y.

## Part 1: Rotations with JAMA in Processing

We will now look at some Processing code for implementing these types of transformations.

**To do:** Create a newProcessing project and drag-and-drop the Jama matrix library into the processing window as you did for last week’s lab. Now cut and paste (or enter) the code shown into Processing and run it. The code creates a vector, , and a rotation matrix for rotating points 45 degrees counter-clockwise. It then multiplies the point by this rotation matrix and prints out the resulting point to the console.

// Create a vector (1,0)  
double[][] Pvals = {{1},   
 {0}};  
Matrix P = new Matrix(Pvals);

//// Create rotation matrix through 45 degrees  
float angle=45\*(PI/180);  
double[][] Rvals = {{cos(angle), -sin(angle)}, {sin(angle), cos(angle)}};  
Matrix R = new Matrix(Rvals);  
  
// Apply rotation matrix to the points  
P=R.times(P);  
  
// Print points (column width, number of digits) to console  
P.print(5, 2);

**To Do:** Check the result of the program by hand on paper.

**To Do:** Alter the program to rotate the point by 90 degrees and check that you get the expected result.

## Part 2: Complete Rigid Body Transformations in Processing

Now let’s add a translation by to the above code.

**To Do:** Add the following code to the top of the program.

//// Create a 2D translation vector  
double tx = 10, ty = 5;  
double[][] Tvals = {{tx}, {ty}};  
Matrix T = new Matrix(Tvals);

And add the following code immediately after the multiplication of P by rotation matrix

// Apply translation to the points  
P=P.plus(T);

The function of these code snippets is to (i) create a translation vector, and, (ii) add the vector to the rotated points.

**To Do:** Again check the result of the code by hand to ensure you understand the purpose and operation each line

**To Do:** Switch the values of tx and ty and predict the resulting output. Run the code to verify your understanding.

## Part 3: Transforming Multiple Points

Often we will want to apply the same transformation to multiple points. One way to do this is to apply the transform to each point in turn. If we were just rotating the points this would consist of,

,

,

etc.

It turns out that linear algebra allows us to package this up as a single matrix multiplication. To do this we *concatenate* all of the input vectors into a single *input matrix*. Multiplying this input vector by the rotation matrix is equivalent to multiplying each column independently. This results in a output matrix where each column is the output vector associated with the corresponding column in the input matrix. This all looks as follows:

**To Do:** Returning to the first program from this lab we now extend it to rotate the four points each at a unit distance out on the +/- x and y axes. Cut-and-paste the code below into a Processing window. Remember again to ensure that you have dragged and dropped the Jama jar file onto the Processing window. Note how the code starts by creating a matrix where each column corresponds to one point. Also note how here, P, is a matrix of output column vectors, all of which get printed by the last line of the program.

// Create a matrix of points to transform  
double[][] Pvals = {{1,0,-1,0},   
 {0,1,0,-1}};  
Matrix P = new Matrix(Pvals);

//// Create rotation matrix through 45 degrees  
float angle=45\*(PI/180);  
double[][] Rvals = {{cos(angle), -sin(angle)}, {sin(angle), cos(angle)}};  
Matrix R = new Matrix(Rvals);  
  
// Apply rotation matrix to the points  
P=R.times(P);  
  
// Print points (column width, number of digits) to console  
P.print(5, 2);

**To Do:** Check the result of the code by hand to ensure you understand the purpose and operation each line.

**To Do:** Alter the program to rotate the point by 90 degrees and check that you get the expected result.

To add the translation component when we are transforming multiple points we create a matrix of translation vectors (i.e. all of which are the same vector. Mathematically this is equivalent to:

Now let’s again add a translation by for each point to the above code.

**To Do:** Add the following code to the top of the program.

//// Create a \*matrix\* of four 2D translation vectors, 1 for each  
//// input point  
double tx = 10, ty = 5;  
double[][] Tvals = {{tx, tx, tx, tx},   
 {ty, ty, ty, ty}};  
Matrix T = new Matrix(Tvals);

And add the following code immediately after the multiplication of P by rotation matrix

// Apply translation to the points  
P=P.plus(T);

The function of these code snippets is to (i) create a matrix of four translation vectors, and, (ii) add the matrix of column vectors to the rotated points.

## **Part 4:** Visualising Rigid Body Transformations.

**To Do:** Create a new Processing project called Lab5a and drag and drop the Jama matrix library jar file into the windows as before. Cut-and-paste, or input, the following code. The code creates a 400x400 pixel window and draws a square of side length 40 pixels in the centre of the window. The corners of the square are stored in the matrix P. The draw\_quad function at the end of the program draws the square defined by the four points store in the columns of P.

float OriginX = 200, OriginY = 200;   
  
void setup() {  
 // Create a window 400x400  
 size(400, 400);  
}  
  
void draw() {  
 background(255);  
  
 // Create points on a 40x40 square centered on 0,0 (X's row 1, Y's row 2)  
 double[][] Pvals = {{-20, 20, 20, -20},   
 {-20, -20, 20, 20}};  
 Matrix P = new Matrix(Pvals);  
   
 // Draw the original square  
 draw\_quad(P);  
  
 // Print points (column width, number of digits) to console  
 P.print(5, 2);  
  
}

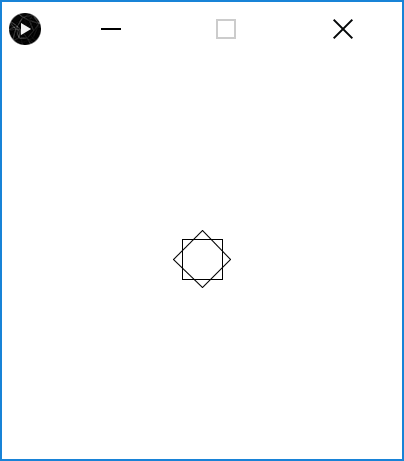
// This function draws a quadrilateral on the screen defined  
// by the four points in the columns of P  
void draw\_quad(Matrix P){  
 // Extract point infromation from matrix   
 float X1=(float)P.get(0, 0)+OriginX;  
 float Y1=(float)P.get(1, 0)+OriginY;  
 float X2=(float)P.get(0, 1)+OriginX;  
 float Y2=(float)P.get(1, 1)+OriginY;   
 float X3=(float)P.get(0, 2)+OriginX;  
 float Y3=(float)P.get(1, 2)+OriginY;  
 float X4=(float)P.get(0, 3)+OriginX;  
 float Y4=(float)P.get(1, 3)+OriginY;   
  
 // Draw rectangle  
 line(X1, Y1, X2, Y2);  
 line(X2, Y2, X3, Y3);  
 line(X3, Y3, X4, Y4);  
 line(X1, Y1, X4, Y4);   
}

**To Do:** Add the following code immediately after the call to draw\_quad(P). This code defines a rotation matrix where , applies it to the points in P, and then draws the resulting quad.

//// Create rotation matrix through 45 degrees  
 float angle=45\*(PI/180);  
 double[][] Rvals = {{cos(angle), -sin(angle)}, {sin(angle), cos(angle)}};  
 Matrix R = new Matrix(Rvals);  
  
 // Apply rotation matrix to the points  
 Matrix P1=R.times(P);

draw\_quad(P1);

This should result in an output as follows:



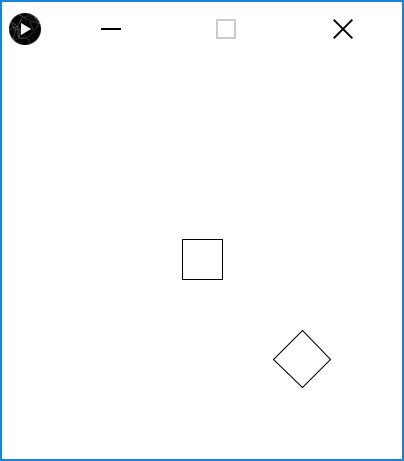
where we can see the rotated square on top of the original.

Now let’s add a translation so that we can see each square separately.

**To Do:** Add the following code immediately after the line that applies the rotation to P1. The code will translate the rotate square by 100 pixels in both the x and y directions.

//// Create a 2D translation matrix to move 50,100  
 double tx = 100, ty = 100;  
 double[][] Tvals = {{tx, tx, tx, tx}, {ty, ty, ty, ty}};  
 Matrix T = new Matrix(Tvals);  
  
 // Apply translation to the points  
 P1=P1.plus(T);

Run the resulting code and ensure you understand what each line does. You should see a window like the one below when you run the code.



**To Do:** Change the program such that the second square is rotated by and translated by

## Part 5: Making the First Square Spin

We will now add a button to our processing window which, when we click and hold it, will make the first square spin. To do this we will first make angle a global variable to store the current rotation angle. We will also add some code for the button functionality.

To Do: To make angle a global variable and the following line to the top of the program

//// Add rotation angle as a global variable  
float angle=0;

Now add the following code immediately after the call to background(255) in the draw function

// Draw button  
 // Fill colour RGB  
 fill(0, 0, 0);  
 // Rectangle  
 rect(10, 10, 50, 20);  
 // Add text "Rotate"  
 textSize(12);  
 fill(255, 255, 255);  
 text("Rotate", 16, 24);   
  
 if (mousePressed)  
 {  
 if (mouseX>10 && mouseX <60 && mouseY>10 && mouseY <30)  
 {  
 angle+=3.0\*(PI/180);  
 }  
 }

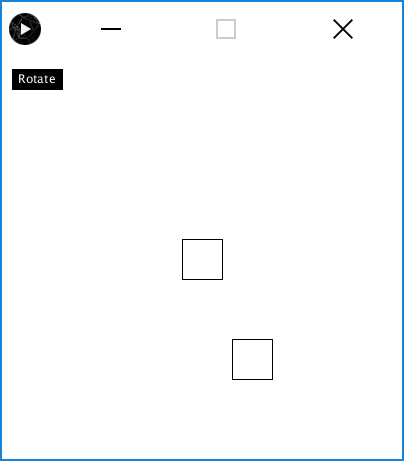
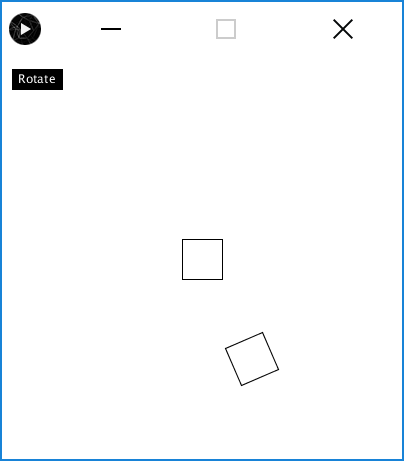
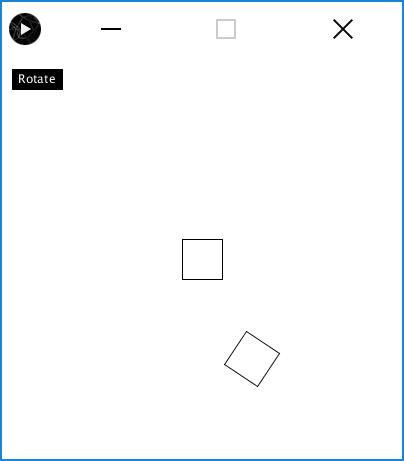
This code will create a black button in the left hand corner of the window which when clicked will add 3 degrees to the current value stored in angle.

Finally you will need to **delete the following line:**

float angle=75\*(PI/180);

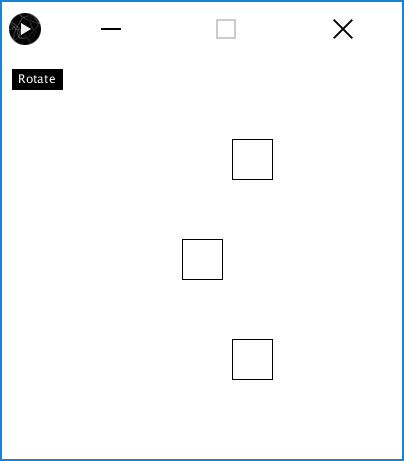
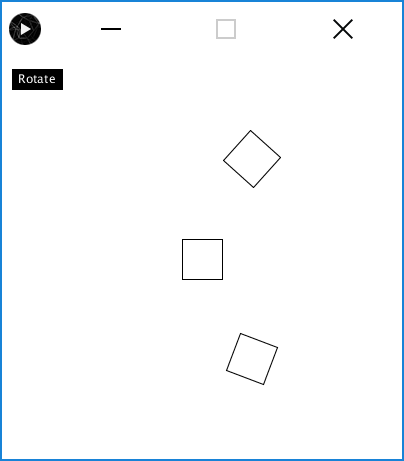
since this line will conflict with the new dynamic angle code. Note the full code for this example is provided in Appendix 1 at the end of this handout.

The figure below shows the program at start (on the left), and then two further screenshots where the rotation of the second square is visible.

## Exercise 1. Adding an extra spinning square

In this exercise you will extend the above program to include a second square as shown in the screenshot below. The square is positioned based on a *translation vector of (50,-100)* and *should spin at twice the speed* of the current spinning square.

To assist you with the task, Appendix 2 provide a full listing of the first program where comments have been added to direct you the new code that is required. Each comment has the format:

**/// STEP N: DIRECTIONS ON CODE REQUIRED**

There are 7 such comments in total where each comment is highlight in red as above.

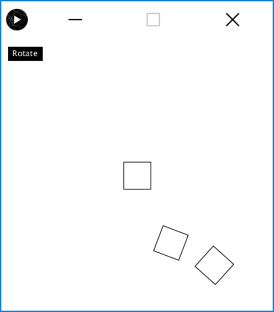
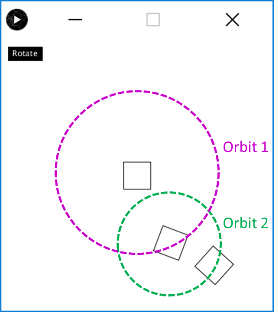
**Once you have completed you solution you should upload your code to the Lab 7 Exercise 1 link in the Linear Algebra in Action section on the CS171 Moodle Webpage.**

## Open Challenge: OPTIONAL!!

If you have finished the lab early and would like to really test your understanding then take the code that you have developed so far and implement a three-body planetary system using “squares as planets”. An example screenshot is shown below on the left, where on the right we have superimposed the orbits to show how each square should move relative to the other squares (i.e. the orbits and text are not required, they are simply shown in the figure here to help you understand the problem).

Here the center is analogous to the Sun and should remain static, whereas the next square should orbit the center square. Finally the outer most square should orbit the second square.

**Note that this is an open challenge for you, but is not marked. However if you do solve the problem we would love to see your work, so feel free to submit your code via the Open Challenge link on the CS171 course page.**

** **

**Appendix 1: Complete code for the rotating square example:**

// Drag and drop Jama.Jar onto IDE/PDE

// http://math.nist.gov/javanumerics/jama/

// http://math.nist.gov/javanumerics/jama/doc/

float OriginX = 200, OriginY = 200;

float angle=0;

void setup() {

// Create a window 200x200

size(400, 400);

}

void draw() {

background(255);

// Draw button

// Fill colour RGB

fill(0, 0, 0);

// Rectangle

rect(10, 10, 50, 20);

// Add text "Rotate"

textSize(12);

fill(255, 255, 255);

text("Rotate", 16, 24);

if (mousePressed)

{

if (mouseX>10 && mouseX <60 && mouseY>10 && mouseY <30)

{

angle+=3.0\*(PI/180);

}

}

// Create points on a 40x40 square centered on 0,0 (X's row 1, Y's row 2)

double[][] Pvals = {{-20, 20, 20, -20},

{-20, -20, 20, 20}};

Matrix P = new Matrix(Pvals);

// Draw the original square

draw\_quad(P);

//// Create rotation matrix through 45 degrees

double[][] Rvals = {{cos(angle), -sin(angle)}, {sin(angle), cos(angle)}};

Matrix R = new Matrix(Rvals);

// Apply rotation matrix to the points

Matrix P1=R.times(P);

//// Create a 2D translation matrix to move 50,100

double tx = 50, ty = 100;

double[][] Tvals = {{tx, tx, tx, tx}, {ty, ty, ty, ty}};

Matrix T = new Matrix(Tvals);

// Apply translation to the points

P1=P1.plus(T);

// Print points (column width, number of digits) to console

P1.print(5, 2);

// Draw Second Square

draw\_quad(P1);

}

void draw\_quad(Matrix P){

// Extract point infromation from matrix

float X1=(float)P.get(0, 0)+OriginX;

float Y1=(float)P.get(1, 0)+OriginY;

float X2=(float)P.get(0, 1)+OriginX;

float Y2=(float)P.get(1, 1)+OriginY;

float X3=(float)P.get(0, 2)+OriginX;

float Y3=(float)P.get(1, 2)+OriginY;

float X4=(float)P.get(0, 3)+OriginX;

float Y4=(float)P.get(1, 3)+OriginY;

// Draw rectangle

line(X1, Y1, X2, Y2);

line(X2, Y2, X3, Y3);

line(X3, Y3, X4, Y4);

line(X1, Y1, X4, Y4);

}

**Appendix 2: Code with comments for Exercise 1**

// Drag and drop Jama.Jar onto IDE/PDE

// http://math.nist.gov/javanumerics/jama/

// http://math.nist.gov/javanumerics/jama/doc/

float OriginX = 200, OriginY = 200;

float angle=0;

**/// STEP 1: ADD A SECOND ANGLE VARIABLE HERE CALLED angle1**

void setup() {

// Create a window 200x200

size(400, 400);

}

void draw() {

background(255);

// Draw button

// Fill colour RGB

fill(0, 0, 0);

// Rectangle

rect(10, 10, 50, 20);

// Add text "Rotate"

textSize(12);

fill(255, 255, 255);

text("Rotate", 16, 24);

if (mousePressed)

{

if (mouseX>10 && mouseX <60 && mouseY>10 && mouseY <30)

{

angle+=3.0\*(PI/180);

**/// STEP 2: ADD CODE HERE TO UPDATE angle1 AT TWICE THE RATE OF angle**

}

}

// Create points on a 40x40 square centered on 0,0 (X's row 1, Y's row 2)

double[][] Pvals = {{-20, 20, 20, -20},

{-20, -20, 20, 20}};

Matrix P = new Matrix(Pvals);

// Draw the original square

draw\_quad(P);

//// Create rotation matrix through 45 degrees

double[][] Rvals = {{cos(angle), -sin(angle)}, {sin(angle), cos(angle)}};

Matrix R = new Matrix(Rvals);

// Apply rotation matrix to the points

Matrix P1=R.times(P);

**/// STEP 3: ADD CODE HERE TO CREATE A SECOND ROTATION MATRIX CALLED R1**

**/// BASED ON THE VALUE OF angle1**

**/// STEP 4: ADD CODE HERE TO CREATE A THIRD SQUARE CALLED P2 WHICH SHOULD**

**/// BE COMPUTED IN A SIMILAR MANNER TO P1 BUT USING R1 (NOT R)**

//// Create a 2D translation matrix to move 50,100

double tx = 50, ty = 100;

double[][] Tvals = {{tx, tx, tx, tx}, {ty, ty, ty, ty}};

Matrix T = new Matrix(Tvals);

// Apply translation to the points

P1=P1.plus(T);

**/// STEP 5: ADD CODE HERE TO CREATE A SECOND 2D TRANSLATION MATRIX**

**/// CALLED T2 using the values tx2 = 50, ty2 = -100**

**/// STEP 6: ADD CODE TO APPLY THE TRANSLATION IN T2 TO THE VALUES**

**/// IN P2**

// Print points (column width, number of digits) to console

P1.print(5, 2);

// Draw Second Square

draw\_quad(P1);

**/// STEP 7: ADD CODE TO CALL draw\_quad WITH P2**

}

void draw\_quad(Matrix P){

// Extract point infromation from matrix

float X1=(float)P.get(0, 0)+OriginX;

float Y1=(float)P.get(1, 0)+OriginY;

float X2=(float)P.get(0, 1)+OriginX;

float Y2=(float)P.get(1, 1)+OriginY;

float X3=(float)P.get(0, 2)+OriginX;

float Y3=(float)P.get(1, 2)+OriginY;

float X4=(float)P.get(0, 3)+OriginX;

float Y4=(float)P.get(1, 3)+OriginY;

// Draw rectangle

line(X1, Y1, X2, Y2);

line(X2, Y2, X3, Y3);

line(X3, Y3, X4, Y4);

line(X1, Y1, X4, Y4);

}