```
/*
Program written by Hector Rodriguez "Doktor Hekkor".
The core mathematical functions are based on pseudo-code written by Felipe Cucker.

The idea here is to implement our own matrix functions, to avoid the memory overhead of using the Jama library.
Jama is used only once, to invert a matrix, since the matrix that has to be inverted is relatively small and so manageable.
Otherwise, we work with float[][] objects, to save memory space.
*/

class Projector {
 //Matrix A; //dictionary        float[pixPerFrame][frames]
 //Matrix M; //                  float[frames][pixPerFrame]
 //Matrix c; //coefficients

 float[][] A, M;  //A is the dictionary, M is the output of applying Moore-Penrose to A
 float[] c; //coefficients

int brightnessmode = 1;                    // which brightnesssfunction we use inside this programm
                                  // 1 - perceptive weights, 2 - plain average, 3 - processing max
int frames, pixPerFrame;  //How many frames in the dictionary, and how many pixels per frame
 PImage[] shadows;
 float magnification = 1.0f;

 //The constructor of this class receives an array of PImage objects,
 //which are the frames in the dictionary.
 //It then arranges them into a matrix object and computes the
 //Moore Penrose pseudo-inverse.
```

```
Projector ( PImage[] images  ){
  int i, j; //loop variables
  float br; //storage variable for brightness
  frames = images.length;
  pixPerFrame = images[0].pixels.length;

  println( pixPerFrame );

   //A = new Matrix( pixPerFrame, frames);
  A = new float[pixPerFrame][ frames ]; //new Matrix( pixPerFrame, frames);

  for (i = 0; i < frames; i++) {
    images[i].loadPixels();
    for (j = 0; j < pixPerFrame; j++ ) {
      br = brightnessFunction( images[i].pixels[j] );
        A[ j ][ i ] =  br ;
    }
  } //for loop ends here
  M = MoorePenroseLeft( A );


  //Now create the array of images that will serve to visualize the "shadows"
  shadows = new PImage[ frames ];
  for (i = 0; i < frames; i++) {
    shadows[i] = new PImage( images[0].width, images[0].height, ARGB);
  }
}//close class constructor
```

```
//This constructor is not so important, just in case we wish to specify our own brightnessmode...
//We should in general not use this.
Projector( PImage[] images, int brightnesmode) {
 this(images );
  this.brightnessmode = brightnessmode;
}

Projector (PImage[] images, float magnification ) {
 this(images);
this.magnification = magnification;
}

/*
This function computes the Moore-Penrose pseudo-inverse of the matrix that is received as a float[][].
The function gets called ONLY ONCE, in the initialization, and never again.
Once we comptue the matrix M in this function, we store that matrix and reuse it to obtain the projection coefficients.
Exeuction of this function is by far the most time-consuming part of the program.
*/
float[ ][ ] MoorePenroseLeft( float[][] A ) {
    int i, j, l; //loop variables
   float temp;
   //double temp; //just for convenience, to avoid writing very long lines of code

  float[][] R, M;
  Matrix S; //We need to use the JAMA library to handle the inversion...

  //Matrix R, S;
```

```
 //M is the return matrix, the others are used to store results of various matrix operations


 //Initialize the matrices that we are going to use.
 //Note that the JAMA library initializes all matrices so that every entry is zero.
 S = new Matrix( frames, frames);
 //R = new Matrix( frames, frames);
 //M = new Matrix( frames, pixPerFrame);


//S =  new float[ frames ][ frames ];
R = new float[ frames ][ frames ];
M = new float[ frames ][ pixPerFrame] ;
 println( "INITIALIZED MATRICES FOR MOORE PENROSE");




//STEP ONE: compute A.transpose times A
 for (i = 0; i < frames; i++) {
   for (j = 0; j < i + 1; j++) {
     R[ i ][ j ] = 0.0f; //unnecessary step, since Java initalizes float arrays with zeros...
    for ( l =  0; l < pixPerFrame; l++) {
      //temp = R.get(i,j) + ( A.get(l, i) * A.get(l,j) );
      //R.set(i, j, temp );
      R[i][j] =  R[i][j] + ( A[l][i] * A[l][j] );
    } //innermost for loop
    R[j][i] = R[i][j];
   } //for loop
 } //outermost for loop
 //We have now computed R = A.transpose times A
```

```
println( "FINISHED STEP ONE -- A.TRANS * A ");



//Step two: invert the result of the previous operation and let S refer to that inverse
//first copy the result into a Jama matrix
for (i = 0; i < frames; i++) {
  for (j = 0; j < frames; j++) {
     S.set(i, j, (double) R[ i ][j ] );
  }
}
//now invert the matrix.
S = S.inverse( );

println( "FINISHED STEP TWO -- INVERTED OUTPUT OF STEP ONE ");



//Step three: compute M = S times A.transpose, to conclude the procedure
  for (i = 0; i < frames; i++) {
    for (j = 0; j < pixPerFrame; j++) {
      for (l = 0; l < frames; l++) {
        //temp = M.get(i, j) + ( S.get( i, l ) * A.get(j, l ) ) ;
        //M.set(i,j, temp);
        M[i][j] = M[i][j] + ( (float) S.get(i,l) * A[ j ][ l ] );
      } //innermost for loop
    } //for loop
  } //outermost for loop

  //We have now comptuerd M = S times A.transpose, so we are done.
```

```
  println( "DONE WITH STEP THREE");
  println( "M[" + M.length + "][" + M[0].length + "]" );
  return M; //we're outta here!

}

float[ ][ ] MoorePenroseLeft_Partial( float[][] A ) {
    int i, j, l; //loop variables
   float temp;
   //double temp; //just for convenience, to avoid writing very long lines of code

  float[][] R, M;
  Matrix S; //We need to use the JAMA library to handle the inversion...

  //Matrix R, S;
  //M is the return matrix, the others are used to store results of various matrix operations

  //Initialize the matrices that we are going to use.
  //Note that the JAMA library initializes all matrices so that every entry is zero.
  S = new Matrix( frames, frames);
  //R = new Matrix( frames, frames);
  //M = new Matrix( frames, pixPerFrame);

//S =  new float[ frames ][ frames ];
R = new float[ frames ][ frames ];
M = new float[ frames ][ pixPerFrame] ;
  println( "INITIALIZED MATRICES FOR MOORE PENROSE");
```

```
//STEP ONE: compute A.transpose times A
  for (i = 0; i < frames; i++) {
    for (j = 0; j < i + 1; j++) {
      R[ i ][ j ] = 0.0f; //unnecessary step, since Java initalizes float arrays with zeros...
     for ( l =  0; l < pixPerFrame; l++) {
       //temp = R.get(i,j) + ( A.get(l, i) * A.get(l,j) );
       //R.set(i, j, temp );
       R[i][j] =  R[i][j] + ( A[l][i] * A[l][j] );
     } //innermost for loop
     R[j][i] = R[i][j];
    } //for loop
  } //outermost for loop
  //We have now computed R = A.transpose times A

println( "FINISHED STEP ONE -- A.TRANS * A ");




//Step two: invert the result of the previous operation and let S refer to that inverse
//first copy the result into a Jama matrix
for (i = 0; i < frames; i++) {
  for (j = 0; j < frames; j++) {
     S.set(i, j, (double) R[ i ][j ] );
  }
}
//now invert the matrix.
S = S.inverse( );
```

```
println( "FINISHED STEP TWO -- INVERTED OUTPUT OF STEP ONE ");



//Step three: compute M = S times A.transpose, to conclude the procedure
  for (i = 0; i < frames; i++) {
    for (j = 0; j < pixPerFrame; j++) {
      for (l = 0; l < frames; l++) {
        //temp = M.get(i, j) + ( S.get( i, l ) * A.get(j, l ) ) ;
        //M.set(i,j, temp);
        M[i][j] = M[i][j] + ( (float) S.get(i,l) * A[ j ][ l ] );
      } //innermost for loop
    } //for loop
  } //outermost for loop

  //We have now comptuerd M = S times A.transpose, so we are done.

  println( "DONE WITH STEP THREE");

  return M; //we're outta here!

}



//This function returns the coefficients of the projection of the pixel array as
//a linear combination of frames in the dictionary.
float[] project_coefficients( float[] pix) {
  return matrixTimesVec( M, pix );
}
```

```
//This function returns the projection of the pixels
//over the space spanned by the frames in the dictionary
float[ ] projection( float[] c) {
 return matrixTimesVec(A, c );
}


//This function receives a frame as  PImage object and returns a PImage that
//is the reconstruction of the original frame using a linear combination
//of frames in the dictionary.
//This function gets called from the main thread of execution.
//Once called, it calls several functions.
//In particular, it calls the purely mathematical functions project_coefficients() and projection()
//It also calls the vectorToPimage() function in order to visualize the vector projections.
PImage projectNewFrame( PImage frame ) {
  int i; //loop variable
  PImage reconstruction;  //output image

   float[] projected; //the final values of the pixels as floats
   float[] imageVector = new float[ pixPerFrame ];


 //before we can resize the image,
 //we need to make a copy of the image because of some issues in processing...
 //see: http://forum.processing.org/two/discussion/2323/copying-and-resizing-pimages-get-in-processing-2-1
 frame = getCopy( frame);
```

```
//Now resize image
frame.resize( displayW, displayH );



 //prepare the pixels for processing
 frame.loadPixels();
   //println( "RECEIVED NEW FRAME" );

 for (i = 0; i < pixPerFrame; i++) {
   //imageVector.set( i, 0, brightnessFunction( frame.pixels[i ] ) );
  imageVector[ i ] = brightnessFunction( frame.pixels[i ] );
 } //for loop

 //println( "Constructed the image vector");
 c = project_coefficients( imageVector );
 //println( "PROJECTED THE COEFFICEINTS =");

 projected = projection( c );
 //println( "Done with the final projection");

 reconstruction = vectorToPImage( projected );

 reconstruction.resize( imgW, imgH );

 return reconstruction;
}
```

```
PImage vectorToPImage( float[] vector ) {
  int i;
 int col;
 PImage output = new PImage( displayW, displayH, ARGB);
 output.loadPixels();
 for (i = 0; i < pixPerFrame; i++) {
  col = (int) vector[ i ];
  //print (col + "  " );
  if (col < 0 ) col = 0;
  else if (col > 255) col = 255;
  output.pixels[i] = color( col );
 }
 return output;
}



//return the adjusted frames in the dictionary.
PImage[ ] showShadows(  ) {
  int i, j;  //loop variables
  int br; //temporary storage variable
  float coef; //storage variable for coefficients
  //output array
 //PImage[] shadows = new PImage [ frames ];

 for (i = 0; i < frames; i++) {
  //create a new image and add it to the output array
  //shadows[i] = new PImage( imgW, imgH, ARGB);

  shadows[i].loadPixels();
```

```
 for (j = 0; j < pixPerFrame; j++) {
  //Get the coefficient.
  coef = c[ i ];
  // if the coefficient is zero, just make the frame black
 if (coef == 0.0f ) br = 0;
  //If the coeffiicent is positive, multiply by the dictionary entry.
  else if (coef > 0.0f ) br =  (int) ( A[j][ i ] * coef );
 //if the coefficient is negative, multiply the neg coefficient times the neg image
  else  br = (int) ( ( 255.0f - A[j][ i]  )* ( - coef ) );



//   else  br = 2 * (int) ( Neg.get(j, i) * ( -1 *  c.get(i, 0 ) ) );


  //here we apply the magnification factor, but only if the coefficients are not close to 1.0f.
 //if (coef < 0.95f) br *= magnification;
 br = (int) ( 255.0f * pow((float) br / 255.0f, 0.85f) );


  //just for safety:
  //We should probably comment these lines out.
  if (br > 255) br = 255;
  else if (br < 0) br = 0;


  //Now we set the output image to the correct value that we want
  shadows[i].pixels[j ] = color( br );
 } //inner for
 shadows[i].updatePixels ( );


//In this trial, I use another formula to adjust the brightness.
//The result is poor, so I have not used it in the final version.
```

```
//  logarithmicEnhancedImage( shadows[ i ] );


 } //outer for
 //We are done, let's get outta here now!!!!



 return shadows;
}









//INPUTS: MATRIX AS 2D FLOAT ARRAY AND VECTOR AS 1D FLOAT ARRAY
//OUTPUT: MATRIX TIMES VECTOR AS 1D FLOAT ARRAY
public float[] matrixTimesVec(float[][] matrix, float[] vec) {
  int i, j;  //loop variables
  float temp; //store the computations here


  //newV is the output vector -- it has the same components as rows in the matrix
  float[] newV = new float[ matrix.length];



  //Make sure we can multiply the matrix times this vector
  if (matrix[0].length != vec.length) {
   println( "INNER DIMENSIONS DON'T MATCH. YOU ARE DOING SOMETHING WRONG!");
  return null;
  }
```

```
  for (i = 0; i < matrix.length; i++) {
    temp = 0.0f;
    for (j = 0; j < vec.length; j++) {
      temp += matrix[ i ][ j ] * vec[ j ];
    }
    newV[ i ] = temp;
  }
  return newV;
} //matrixTimesVec




//INPUTS: MATRIX AS 2D FLOAT ARRAY, STARTING COLUMN AND ENDING COLUMN OF DESIRED SUBMATRIX
//OUTPUT: SUBMATRIX AS 2D FLOAT ARRAY
public float[][] subMatrix(float[][] input, int startCol, int endCol){
  if ( endCol <= startCol ) {
    println("REQUESTED SUBMATRIX DIMENSION ERROR!! MAKE SURE NO. OF COL > 0");
    return null;
  }
  int outputMatrixWidth = endCol - startCol + 1;
  float[][] outputSubMatrix = new float[pixPerFrame][outputMatrixWidth];
  // i == frames in SubMatrix, j == pixels per frame
  for(int i = 0; i < pixPerFrame; i++){
    for(int j = 0; j < outputMatrixWidth; j++){
      outputSubMatrix[i][j] = input[i][j + startCol];
    }//inner for loop
  }//outer for loop
  //println("submatrix [" + outputSubMatrix.length + "][" + outputSubMatrix[0].length + "]");
```

```
  return outputSubMatrix;
} //subMatrix


//INPUTS: VECTOR AS 1D FLOAT ARRAY, STARTING AND ENDING CROPPING POINT
//OUTPUT: CROPPED 1D VECTOR ARRAY
public float[] croppingArray(float[] input, int start, int end){
  if ( end <= start ){
    println("REQUESTED ARRAY DIMENSION ERROR!! MAKE SURE ARRAY LENGTH > 0");
    return null;
  }
  int outputLength = end - start + 1;
  float[] outputArray = new float[outputLength];
  for(int i = 0; i < outputLength; i++){
    outputArray[i] = input[i + start];
  }//for loop
  //println("cropped coefficient [" + outputArray.length + "]");
  return outputArray;
} //croppingArray




//The rest can be ignored. These are utility image processing functions.
//I should move them out of this class and put them into a separate ImageUtils class.
//When I have time... Life is short...

//This function is not being used at the moment.
//It does not work well here.
void logarithmicEnhancedImage( PImage img ) {
  //scaling factor
```

```
  //see:  http://homepages.inf.ed.ac.uk/rbf/HIPR2/pixlog.htm
  float c = 255.0f / log( 1.0f + 255.0f );
  //Compute the maximum value of the data (assuming the minimum to be zero).
  //We will use this max later to scale the data to the interval [0,255]
  float max = Float.MIN_VALUE;
  img.loadPixels();
  for (int i = 0; i < img.pixels.length; i++) {
    float br = red(img.pixels[i] );
    if (max < br ) max = br;
  } //for loop

  //we now scale each piece of data from [0, max] to [0,255]
  for (int i = 0; i < img.pixels.length; i++) {
    //Apply the formula from: http://homepages.inf.ed.ac.uk/rbf/HIPR2/pixlog.htm
    img.pixels[i] = color( c * log( red( img.pixels[i]   ) ) + 1.0f  );
  }
}




  // BRIGHTNESS FUNCTION SWITCH ::::::::::::::::::::::::::::::::::::::::::::::::::::::::
// central brightness function for all other classes to conveniently switch
 float brightnessFunction( color col ){

    int r,g,b;
    float ret=0.0;
```

```
switch( brightnessmode ){

  // 1 - preceptive option -----------------------------------------------------
  case 1:
   r = (col >> 16) & 0xFF;  // Faster way of getting red(argb)
   g = (col >> 8) & 0xFF;   // Faster way of getting green(argb)
   b = col & 0xFF;          // Faster way of getting blue(argb)

  ret =  ( 0.299*((float) r)) +  ( 0.587*((float) g)) +  ( 0.114*((float) b) );
  break;

 // 2 - plain average --------------------------------------------------------
 case 2:

   r = (col >> 16) & 0xFF;  // Faster way of getting red(argb)
   g = (col >> 8) & 0xFF;   // Faster way of getting green(argb)
   b = col & 0xFF;          // Faster way of getting blue(argb)

  ret = ( (float) (r + g + b)) / 3.0;
  break;

 // 3 - processing function : highest value ---------
 case 3:
   ret =  brightness( col );
   break;
}


return ret;
```

```
  }// brightnessFunction .....................................................................




  /*
  This code is from:
  http://forum.processing.org/two/discussion/2323/copying-and-resizing-pimages-get-in-processing-2-1
  */
  public PImage getCopy(PImage image) {
    PImage newImage = createImage(image.width, image.height, image.format);
    newImage.loadPixels();
    System.arraycopy(image.pixels, 0, newImage.pixels, 0, image.pixels.length);
    newImage.updatePixels();
    return newImage;
}




}
```