Exercise 1: Poisson Systems

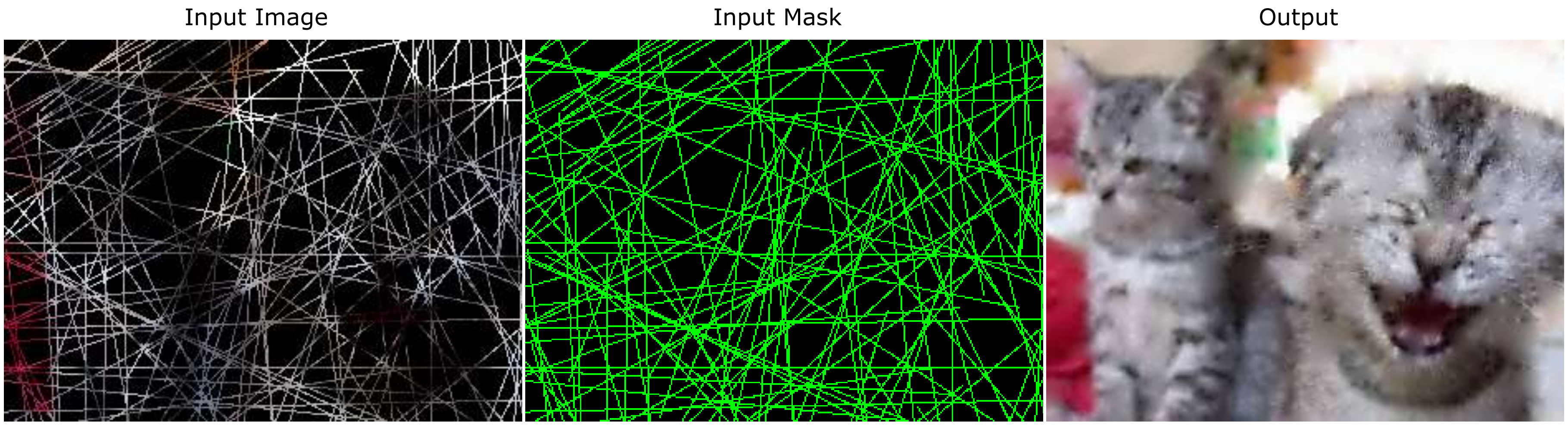
1. Objective

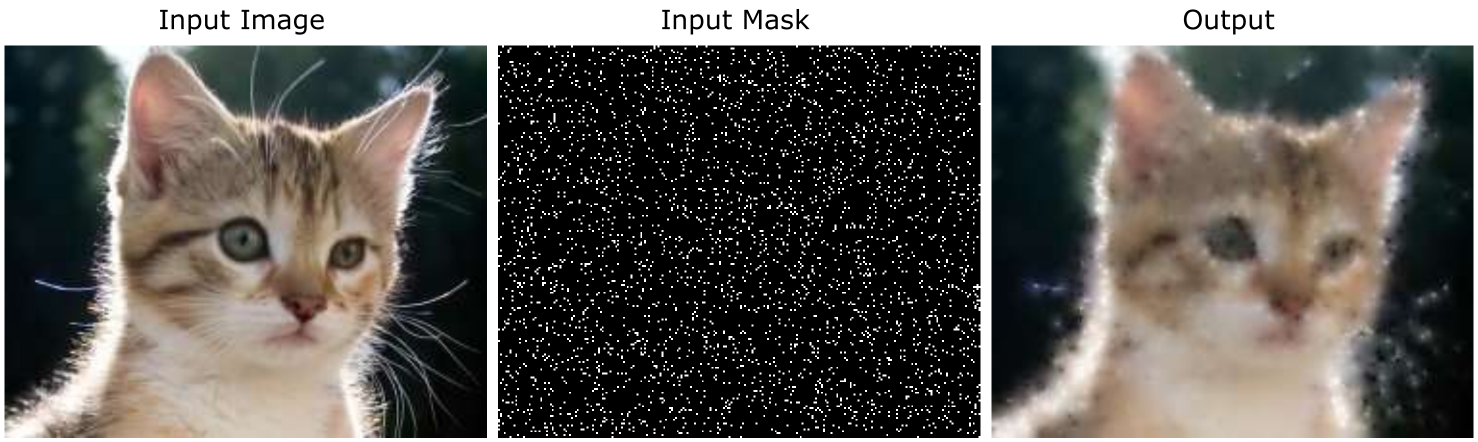
In this exercise you will implement image completion and image cloning using Poisson systems.

Image completion is the process of filling holes in an image with color that matches the existing data. The method you will implement in this exersice is the simplest method to complete an image since it only fills with smooth color.
Image cloning is the process of smoothly implanting part of one image into another in a specific place. The cloning

2. Theory

Image completion





Given an image with one missing pixel at (x0, y0) which you want to fill in, A good choice would be to set that pixel in a way that it's Laplacian equals zero.
Let **I(x,y)** be the value of the pixel in coordinates (x,y) from the input image and let $P(x_0,y_0)$ be the value we're looking for.
To fill in the pixel at (x0, y0) we solve the following single Laplacian equation:

$$4P(x_0,y_0) - I(x_0-1,y_0) - I(x_0+1,y_0) - I(x_0,y_0-1) - I(x_0,y_0+1) = 0$$

Moving members around:

$$P(x_0,y_0) = (I(x_0-1,y_0) + I(x_0+1,y_0) + I(x_0,y_0-1) + I(x_0,y_0+1)) / 4$$

Incidentally, we get that the value of (x_0,y_0) is the average of its neighbors.

More generally, Given an image with many missing pixels, an image completion algorithm enforces similar logic to fill in the holes by saying that eventually, every unknown pixel should receive a value according to the Laplacian equations.

A pixel <u>in the output</u> image at coordinates (x,y) is considers as a variable - $P_{x,y}$. x is in the range [0,m-1] and y in the range [0,n-1]. m,n are the width and height of the image respectivly.
As before $I_{x,y}$ is the value of the pixel in the <u>input</u> image at coordinates (x,y).
The second input of the algorithm is the <u>mask image</u>. A pixel that is black in the mask image should be considered as a hole. The mask image size is the same as input image and its values are $M_{x,y}$.
The objective is to find All the values for all $P_{x,y}$ variables. Instead of a single equation, we build a system of equations:

```
if $M_{x,y}$ is not black  *not a hole*
    $P_{x,y} = I_{x,y}$
      *A non-hole pixel gets the known value.*
else
    $4P_{x,y} - P_{x-1,y} - P_{x+1,y} - P_{x,y-1} - P_{x,y+1} = 0$
    *Laplacian equation for the missing pixel in (x,y)*
```

The number of equations is equal to the number of pixels in the image. Notice that without the equations of the first kind ($P_{x,y} = I_{x,y}$) the solution to the system is trivial: $P_{x,y}=0$ for all x,y.
We can formulate the problem as a linear system of the form AX = B, where:

- X is the vector of all variables ($P_{x,y}$) - This vector's size is N=m·n, every element corresponds to a pixel in the image.
- B is the vector of the values at the right hand side of the equations - its size is also N.
- A is the matrix of coefficients of the left hand side of the equations. its size is NxN

Solving the system will give us the vector X. From it we extract the values of all $P_{x,y}$ and draw the output image. The size of the output image is the same as the input image size.
The matrix A is 99% filled with zeros - a sparse matrix. every row has exactly 5 elements that are not 0.
This equation system is slightly different than the one discussed in class. The difference is that here we take all the pixels in the image whereas in the lecture Lior said that only those pixels that are missing and on the edge should be regarded.
This difference does not affect the solution of the system, only the size of the matrix and the running time of the algorithm. This simplified system is a larger matrix so it takes longer to solve.

A few more notes:
indexes: Don't confuse the coordinates of the image with the indexes of the matrix. The image is sized (m)x(n) and the matrix A is sized (m·n)x(m·n). Every pixel "owns" a complete row in the matrix A. To deal with this you'll need to convert its image coordinates (x,y) to the index of that row.
color: The above description considers every pixel as a single value. This is suitable for a gray-scale image. In color images every pixel has 3 values: red, green and blue. You can solve for each color channel separately, in a separate linear system and then combine the results into a single image.
edges: As with almost any other image processing algorithm, the edges of the image need to get a slightly special treatment. You need to figure out how the equations change. Think about how the Laplacian equation is built.

Image cloning.

The procedure for solving the image cloning problem is almost exactly the same as Image completion.

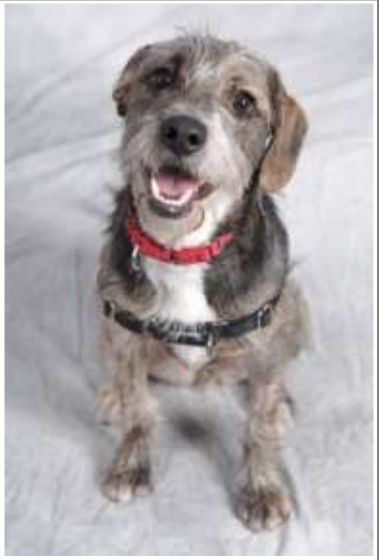
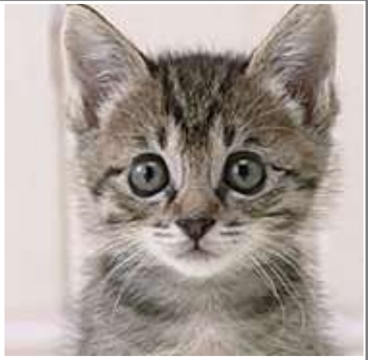
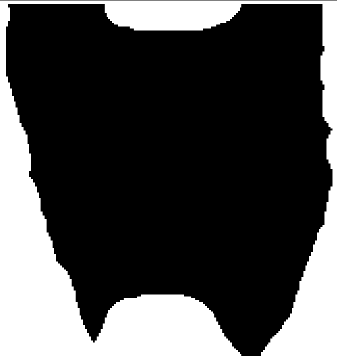
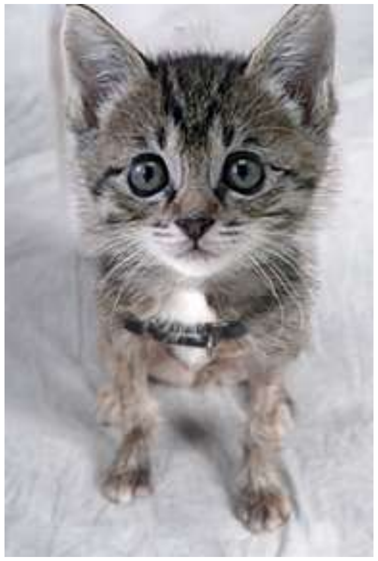
Background Image	Cloned Image	Mask	Result of cloning to (0,0)
			

Image cloning gets the following inputs:

- The background image - onto which we implant the second image. value at (x,y) is $I_{x,y}$
- The cloned image - the image we implant onto the background. Value at (x,y) is $C_{x,y}$
- The mask image - Only pixels that are black in the mask should transfer from the cloned image to the background image. The value at (x,y) is $M_{x,y}$ and it corrsponds to the pixel $C_{x,y}$ from the cloned image.
- Coordinates x_0,y_0 - A point in the background image where the upper-left corner of the cloned image should end up.

The size of the background image can be different from the size of the cloned image. The mask the cloned image are the same size.

In cloning, instead of equating the Laplacian to 0, we set it to be equal to the Laplacian from the cloned image. The modified equation system is as follows -
As before, the output pixels are represented by variables $P_{x,y}$

if $M_{x,y}$ is not black: *The mask says we should not clone this pixel.*

$P_{x,y} = I_{x,y}$

Set it to the value of the background.

else

$4P_{x,y}-P_{x-1,y}-P_{x+1,y}-P_{x,y-1}-P_{x,y+1} = 4C_{x,y}-C_{x-1,y}-C_{x+1,y}-C_{x,y-1}-C_{x,y+1}$

The Laplacian of the output should be the same as the Laplacian of the cloned image.

Notice that $(4C_{x,y}-C_{x-1,y}-C_{x+1,y}-C_{x,y-1}-C_{x,y+1})$ is a constant that depends only in the cloned image.

4. Implementation

You are to implement a command line program that implements image completion and cloning described above. The usage of the program should be as follows:

Image completion:
```
    java ex1.jar -complete input.jpg mask.png output.jpg
```

Image cloning:
```
    java ex1.jar -clone background.jpg cloned.jpg mask.png x0 y0 output.jpg
```

The size of the output image need to be same as the size of the input or background image.

- Your program must support at least the **PNG** and **JPG** image formats for formats. Don't use JPG images for the mask images since JPG does not maintain exact color. For the masks use only PNG.
- Outputs should be **PNG**.
- The program should return within a few seconds for small images and up to a few minutes for large images.
- You should write all your code using **Java 6**.

MTJ
You are expected to implement the algorithms yourself without use of image processing utilities from the Java library. You are not however expected to write a solver for linear systems. To solve the linear system needed use the MTJ library. MTJ contains a variety of matrix classes and iterative solvers them. In order to get adequate results you should familiarize yourself with the library. Specifically read in this page:
http://ressim.berlios.de/overview.html
The section titled **"Using the iterative solvers and preconditioners"**
- Make sure you understand every line of the example there. (read the docs)
- Select a Sparse matrix class (either FlexCompRowMatrix or CompRowMatrix)
- Set the accuracy of the solution by adjusting the variables of the IterationMonitor
- Try out different combinations of Preconditioners and IterativeSolvers

Notes and Hints

- Work incrementally. start with a small and simple example and work your way to the full functionality. Start with gray-scale and once that works move to color.
- Use ImageIO and BufferedImage to load and save image files.
- As a representation of an image you can use BufferedImage directly or you can write your own image class based on arrays or buffers of int the internal container. Using an ArrayList of *class MyNicePixel* (where *MyNicePixel* is a class you wrote to represent a single pixel) is a bad idea. It will cause your program to be slow and consume alot of memory.
- Avoid copying large chunks of data as much as possible.

Code quality
Your code should be well designed in a reasonable Object Oriented way. Classes should have defined areas of resposibility. An example of a bad design is a program made entierly of a single class with nothing but static methods.
The code should be internally documented and easy to read. Don't write long functions or duplicate code. Avoid over-design and code bloat .

If you really want you may implement the exercise using **C**, **C#** or **C++** but nothing other than these. Specifically not matlab or C++/CLI. If you choose to not use Java you are on your own as far as libraries for loading images and solving linear systems are concerned.

5. References

Linear equations systems (wikipedia)
Matrix Toolkit for Java (MTJ)
JDK 6 documentation

6. Bonus points

The matrix of the Poisson system described above is sized $O(N^2)$ where N=m·n is the total number of pixels in the output image. For bonus points, reduce the size of the matrix to $O(p^2)$ where p is the number of pixels in the mask image, thus increasing the performance of the process.
The systems described in section 2 contains pixels that are not needed for the soluion. The only variables that need to be solved for are the ones which belong to the pixels in the mask and the pixels at the boundary of the mask.

The end result of the the new system should be the same the the result of the full sysetm.
Notice: For receiving the bonus you need to have completed all other features of the exercise.

7. Submission

- Submission must in **Pairs** (unless approved by Lior).

- You need to make the submission by 23:59 of the submission date.
- On submission date, send a zip file containing all of the following items to the email address listed below
    - Complete source code
    - A compiled JAR file. The JAR file needs to have its extension renamed so it won't be filtered by the email server.
    - Documentation: A 1-3 pages document (doc/docx/pdf) explaining the structure of your code, how to operate the application, bonuses implemented, and anything else needed to make it work.
    - At least two original examples which demonstrate that your program is working. For each example include the input image and the two output images from the two image completion methods. The examples should not be the images given with this exercise
- After sending the submission email make sure you receive confirmation that your submission was received. A confirmation will be sent to you within 24 hours.

8. Grading

- 40% - Image completion
- 50% - Image cloning
- 5% - Code quality
- 5% - Original examples of both algorithms - inputs and outputs.
- 5 points - bonus.

9. Contact

Submission and questions regarding the exercise:
Shy Shalom - shoosh.cg@gmail.com