Main class:

- **ColorTextureSynthesis.java** for color sample images
- **TextureSynthesis.java** for gray sample images

1. Test your function using the checkerboard function that we provide. Execute the call: `my_SSD(checkerboard(3,3),[[0 1 1; 1 0 -1; 1 0 0], [1 1 1; 1 1 0; 1 1 1]])` and turn in a printout of the results, along with your code.

   
   $\begin{bmatrix}
   0.5 & 0.5 & 0.625 & 0.5 & 0.5 & 0.375 & 0.5 \\
   0.625 & 0.375 & 0.25 & 0.375 & 0.625 & 0.75 & 0.625 \\
   0.625 & 0.375 & 0.0 & 0.375 & 0.625 & 1.0 & 0.625 \\
   0.5 & 0.5 & 0.375 & 0.5 & 0.5 & 0.625 & 0.5 \\
   0.375 & 0.625 & 0.75 & 0.625 & 0.375 & 0.25 & 0.375 \\
   0.375 & 0.625 & 1.0 & 0.625 & 0.375 & 0.0 & 0.375 \\
   0.5 & 0.5 & 0.625 & 0.5 & 0.5 & 0.375 & 0.5 \\
   \end{bmatrix}$

   7x7 output SSD result (skip the boundaries)

2. Test your program using the checkerboard function to generate a sample checkerboard pattern, and using the bricks image. Hand in printouts of the results.

Sample image: 60x60
Output image : 200x200
Seed: 3x3
Neighborhood kernel: 20x20
Gaussian Kernel: not using
Gray level image
Total time: 2990703 ms
(2990s = 49 mins)
3. Use two more images of your own choosing as sample textures and generate new textures using these samples. Turn in printouts of the original images and of the textures you generate. You can modify your program so that it will work with color images, and generate color textures.

Sample image: 108x106
Output image: 200x200
Seed: 3x3
Neighborhood kernel: 24x24
Gaussian Kernel: not using
Gray level image
Total time: 12702485 ms
(12702s = 3 hours and 31 mins)

Color Texture
Sample image: 64x64
Output image: 200x200
Seed: 3x3
Neighborhood kernel: 13x13
Gaussian Kernel: not using
Color image
Total time: 7939671 ms
(7939s = 2 hours and 12 mins)
5. Implement Gaussian smoothing described in the paper.

Sample image: 72x62
Output image: 100x100
Seed: 3x3
Neighborhood kernel: 23x23
Gaussian Kernel: using gray level image
Total time: 3707047ms
(3707s = 1 hours and 2 mins)
Notes

1. One problem of this algorithm is its tendency for some textures to occasionally slip into a wrong part of the search space and start growing garbage or get locked onto one place in the sample image and produce verbatim copies of the original. These problems occur when the texture sample contains too many different types of texels (or the same texels but differently illuminated) making it hard to find close matches for the neighborhood context window. These problems can usually be eliminated by providing a bigger sample image.

2. Too small neighborhood window size will produce images with incorrect features.

<table>
<thead>
<tr>
<th>Sample</th>
<th>Result Image (Seed:3x3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60x60</td>
<td>Neighborhood size: 9x9</td>
</tr>
<tr>
<td></td>
<td>Total time: 48172 ms</td>
</tr>
<tr>
<td></td>
<td>(incorrect)</td>
</tr>
<tr>
<td></td>
<td>Neighborhood size: 20x20</td>
</tr>
<tr>
<td></td>
<td>Total time: 116063 ms</td>
</tr>
<tr>
<td></td>
<td>(correct)</td>
</tr>
</tbody>
</table>

3. Texture synthesizing with large size of sample images or neighborhood window size is time consuming.

<table>
<thead>
<tr>
<th>Sample</th>
<th>Result (Seed:3x3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size: 140*96</td>
<td>Size: 50*50</td>
</tr>
<tr>
<td></td>
<td>Total time: 2032641 ms</td>
</tr>
<tr>
<td>Size: 80*69</td>
<td>Size: 50*50</td>
</tr>
<tr>
<td></td>
<td>Total time: 708141 ms</td>
</tr>
</tbody>
</table>