pix2circ

Release 2022

Jon Augensen, Ole Benjamin Gauslaa & Lars Øvergård

Dec 02, 2022

CONTENT:

CHAPTER

ONE

INTRODUCTION

The topic we have chosen is **'pixels to circles'.** The assignment is as follows :

```
Pixels to circles

As an input, a two-dimensional image or landscape is given by an n x n square of black-
↪and-white
pixels.

Your code should create an approximation of this landscape by a set of superimposed␣
↪circular disks.
It should aim at the best possible accuracy for a given maximum number of circular disks␣
↪that can
be varied as part of the input. It should also be able to generate a lossless␣
↪representation of
the image/landscape using an unlimited number of circular disks.
```

Our **method** to deal with the task at hand was to create 3 main classes, namely Circle, Image and ImageConverter.

Flowchart and structure of our 3 main classes:

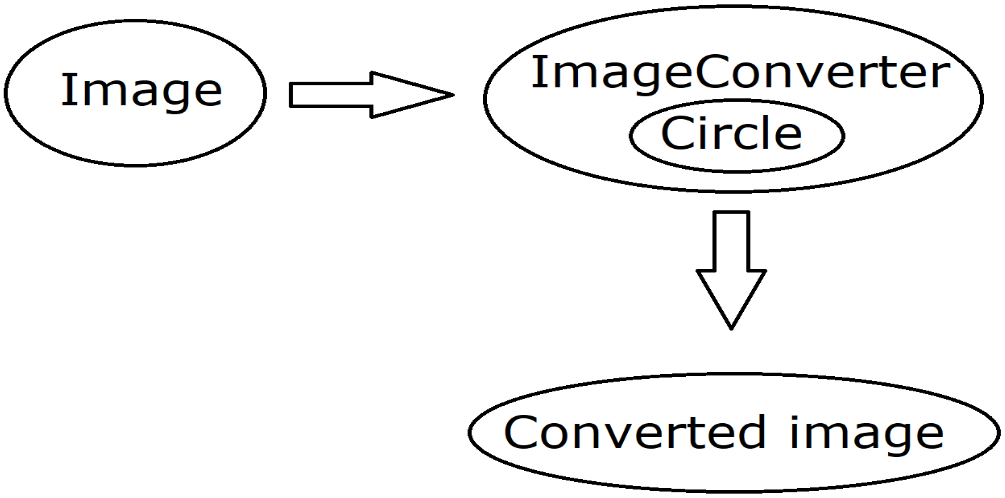


Image	Imageconverter	Circle
Imports picture	Imports picture	Receive values
Extracts length	Placing circles	Set values
Extracts width	Reads image info	
Calculates values	List of circles	

CHAPTER

TWO

ALGORITHMS TESTED

We wanted to test more than one algorithm. This section contains the algorithms used Algorithms that got tested was with both random aswell as systematic approaches.

2.1 ‘Bogo place’

This algorithm is loosely based on the poorly optimized random algorithm ‘Bogo sort’, hence the given name. ‘Bogo place’ works in the way that it places circles at random, with no clear structure. This algorithm was included as there was a need for a worst case scenario to compare with. It is expected that this algorithm will provide this - the worst case scenario.

2.2 Bogo feedback - Circle version

The same as ‘Bogo place’ but with a more distinct approach towards accuracy. The first version of the modified bogo place. This one work with somewhat the same random approach, except that it has more focus on accuracy. This version places a set amount of circles. The biggest issue with this algorithm is that it might use alot of time to finish the image if too many circles is set, as shown in results.

Note: The code for this version is not detailed in ‘Scripts’ or ‘Code documentation’ because it was a ‘first draft’ kind of thing. It is very similar to the second version - the accuracy one, which is the one we ended up using.

2.3 Bogo feedback - Accuracy version

This version of the modified bogo place algorithms is set up the same way as the previous one. The only difference is that in this version, one sets the accuracy instead of the amount of circles. The algorithm will then place out circles at random until the set accuracy is reached.

2.4 Directed random place

This algorithm is based on the bogo algorithm, but with addtional strategies for radius and color. The circles color is based on the highest improvement of accuracy. it bases the iteration on a condition if successive placement of circles. For each size of radius there will be n trials of placement. When this number is reached, radius is reduced by 1. When radius is 1 and n numbers of trials are done, the loop will be terminated.

CHAPTER

THREE

CODE DOCUMENTATION

3.1 Image

Header file: `image.h`

`Image::Image(std::string file_name)`

`void import_image(std::string file_name)`

Import image

Brief Imports the image based on the filename. It will use the "find dims()" to find and check the dimensions of the image, and then "image_make()" to set the pixels value to the img_array.

Parameters **`file_name`** – A string, contains the file name

`void print_dims()`

Print dimensions

Brief Prints the dimensions, if the dimension are not set, it will display an error message and return it.

`void print_image()`

Print image

Brief Prints the image and if the dimension are not set, it willl display an error message and return from it.

`int get_image_rows(){return dims[0]}`

Get image rows

Brief Returns the images row (height)

`int get_image_columns(){return dims[1]}`

Get image columns

Brief Returns the image columns (width)

`void set_dims(int rows, int columns)`

Set dimensions

Brief Sets the dimensions manually. To be used for custom size of image and/or buypass the control check of consistency.

Parameters

- **`rows`** – Holds the number of rows
- **`columns`** – Holds the number of columns

`void find dims(std::String file_name)`

Find dimensions

Brief Will find the first columns-length and ensure all other columns have the same length. Will abort if one column is inconsistent. Stores the dimensions if all checks out.

Parameters `file_name` – A string, contains the file name.

`void image_make(std::String file_name)`

Make image

Brief Imports the image based on the filename. It will check if the dimensions are set and if not, run "find_dims()" and then run through all the pixels in the image and store the values on a 2D array. This function and "set_dims()" can be used for custom size of an image.

Parameters `file_name` – A string, contains the file name.

`int check_pixel(int x, int y){return img_array[y-1][x-1]}`

Check pixels

Brief Returns the pixel/color value of a given position(input).

Parameters

- `x` – Holds the column position of the pixel.
- `y` – Holds the row position of the pixel.

Returns int, the colour value.

`bool is_image_imported()`

Image imported (should be expanded!)

Brief Checks if all the required values are set for operations for an image. Checks dimensions nad if atleast one pixel is given.

Returns bool, values are set.

Source code: *Related to Image*

3.2 ImageConverter

Header file: `imageconverter.h`

`ImageConverter::ImageConverter()`

`void print_circles()`

Print circles (might get changed to return or save values to file!)

Brief Will iterate through the circle_list[vector] and print the values of the circles in the terminal.

`int get_amount_circles(){return circle_list.size()}`

Get amount of circles

Brief Returns the number of circles in the circle list.

`int get_circle_x_pos(int i){return circle_list[i].get_x_pos()}`

Get x position

Brief Returns the x positions from the circle list.

Parameter **i**

Returns x position [int]

`int get_circle_y_pos(int i){return circle_list[i].get_y_pos()}`

Get y position

Brief Returns the y positions from the circle list.

Parameter i

Returns y position [int]

`int get_circle_radius(int i){return circle_list[i].get_radius()}`

Get circle radius

Brief Returns the circle radius from the circle list.

Parameter i

Returns radius [int]

`int get_circle_color(int i){return circle_list[i].get_color()}`

Get circle color

Brief Returns the circle color from the circle list.

Parameter i

Returns circle color [int]

`void print_approx_image()`

Print approximate image

Brief Prints the approximated image and if image is empty, it will display an error message and return from it.

`double accuracy()`

Accuracy

Brief Compare true predictions to total of predictions.

`double precision()`

Precision

Brief Compare true positive to total positive predictions

`double recall()`

recall

Brief Compare true positive with actual positive

`double f1_score()`

F1 score

Brief Compare true positive approximation with all positive approximation and all false approximation

`void evaluation_of_pixels(int &p, int &tn, int &fp, int &fn)`

Evaluation of pixels.

Brief Compare original image with approximate image. Calculate True positive (=1) pixels, True negative(=0) pixels, False negative pixel, false positive pixel.

`void bogo_algorithm(int wanted_circles)`

Bogo algorithm

Brief Bogo algorithm tries to make the worst case scenario for placing circles, by randomly placing them, with a random size, only limited by the image diagonal.

Parameters `wanted_circles` – Value which specifies the number of circles to be placed by algorithm.

`void bogo_feedback(int wanted_circles)`

Bogo Feedback

Brief This algorithm is based on bogo algorithm , but checks if accuracy increases, and if it does, it will keep the circle. If many circles is needed, it will scale the circles down.

Parameters `wanted_circles` – Value which specifies the number of circles to be placed by algorithm

`directed_random_place`(*int wanted_circles*)

Directed random place

Brief This algorithm is based on bogo algorithm, but with additional strategies for radius and color. The circles color is based on the highest improvement of accuracy. It bases the iteration on a condition of succesful placement of circles. For each size of radius there will be n trials of placement. When this number is reached, radius is reduced by 1. When radius is 1 and n numbers of trials are done, the loop is terminated.

`private:`

Progress bar

Brief Prints the progress in the terminal, with a bar.

Parameter Where it is, in the algorithm [int]

Parameter The goal, user set parameter [int]

Source code: *Related to Image*

3.3 Circle

Header file: `imageconverter.h`

Note: `Circle` is a nested class within imageconverter.

`Circle::Circle(int x, int y, int r, int c)`

`int get_x_pos() const { return this->get_x_pos}`

Get x position

Brief An implenetation for returning the x-value of a circle.

Returns x position [int]

`int get_y_position() const { return this->get_y_pos}`

Get y position

Brief An implementation for returning the y-value of a circle.

Returns y position[int]

`int get_radius() const{ return this ->radius}`

Get radius:

Brief An implementation for returning the radius of a circle.

Returns radius [int]

`get_color() const {return this->color}`

Get color

Brief An implementation for returning the color-value of a circle.

Returns color[int]

`void set_x_pos(int x) { this->x_pos = x}`

Set x position

Brief An implementation for setting the x-value of a circle.

Parameters **`x`** – contains the x-position [int]

`void set_y_pos{int y) { this->y_pos = y}`

Set y position

Brief An implementation for setting the y-value of a circle.

Parameters **`y`** – contains the y-position [int]

`void set_radius(int r) {assert(r>=0) this->radius = r}`

Set radius

Brief An implementation for setting the radius of a circle.

Parameters **`r`** – contains the radius [int]

`void set_color(int c) {this->color = c}`

Set color

Brief An implementation for setting the color of a circle.

Parameters **`c`** – contains the color. [int]

`bool check_circle()`

Check circle

Brief Function for checking the area, which the circle is placed. Will store relevant data.

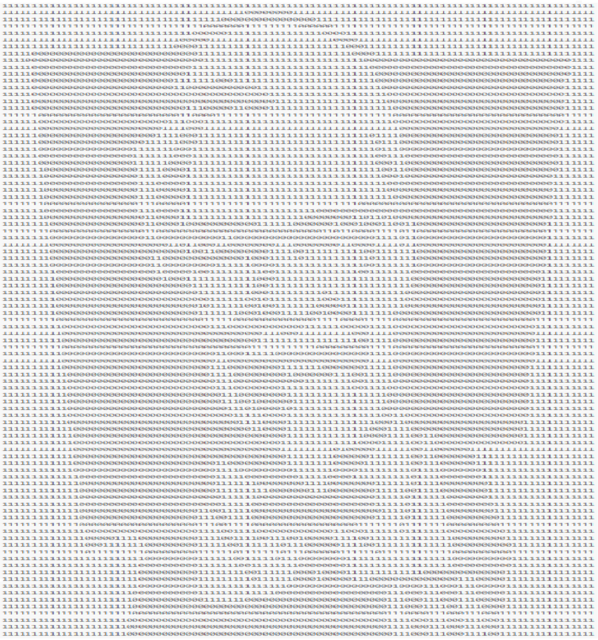
Return bool Tells if the circle is placed on a black pixel.

Source code: *Related to Circle*

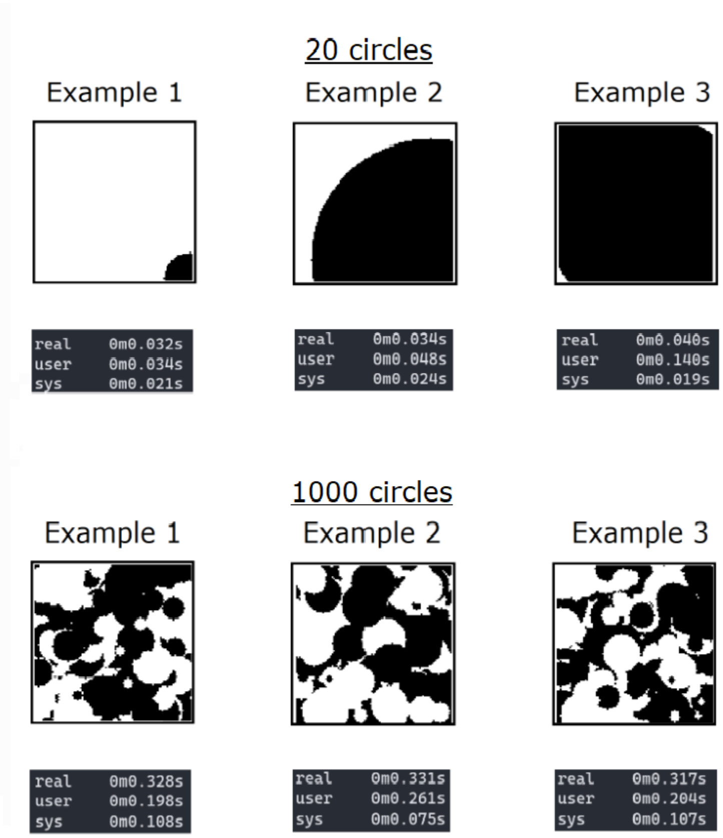
CHAPTER FOUR

RESULTS

To test the algorithms, a binary image of the KFC logo was converted to 1's and 0's.



Bogo place As this algorithm is a random, worst case scenario algorithm, the test results was as expected, totally random in the range from all white to all black. As shown, run with both 20 circles and 1000, the results makes no sense in comparison with the original picture. In turn, this makes alot of sense as this is an *all* random algorithm.



```
real    0m0.032s
user    0m0.034s
sys     0m0.021s
```

```
real    0m0.034s
user    0m0.048s
sys     0m0.024s
```

```
real    0m0.040s
user    0m0.140s
sys     0m0.019s
```

```
real    0m0.328s
user    0m0.198s
sys     0m0.108s
```

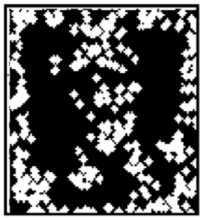
```
real    0m0.331s
user    0m0.261s
sys     0m0.075s
```

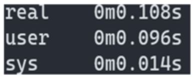
```
real    0m0.317s
user    0m0.204s
sys     0m0.107s
```

Bogo feedback - Circle version Even though the results shown is not subpar, the performance is. If this version is run with for example 1500 circles, it might not even finish running, as bogo place is still randomly placed. As shown in the examples, the runtime of 1000 circles is still under 1 second, but 1500 circles was aborted after 3 minutes and 25 seconds.

300 circles

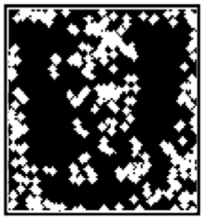
Example 1

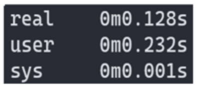




```
real	0m0.108s
user	0m0.096s
sys	0m0.014s
```

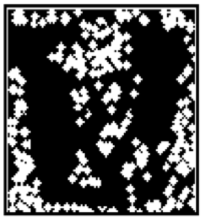
Example 2

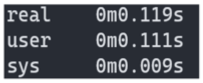




```
real	0m0.128s
user	0m0.232s
sys	0m0.001s
```

Example 3

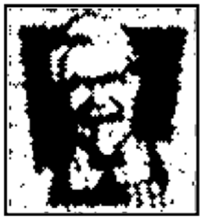


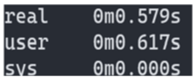


```
real	0m0.119s
user	0m0.111s
sys	0m0.009s
```

1000 circles

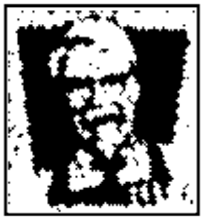
Example 1

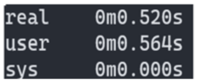




```
real	0m0.579s
user	0m0.617s
sys	0m0.000s
```

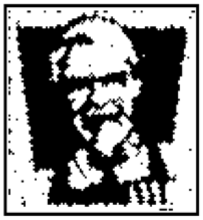
Example 2

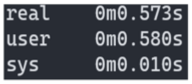




```
real	0m0.520s
user	0m0.564s
sys	0m0.000s
```

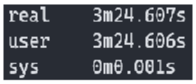
Example 3





```
real	0m0.573s
user	0m0.580s
sys	0m0.010s
```

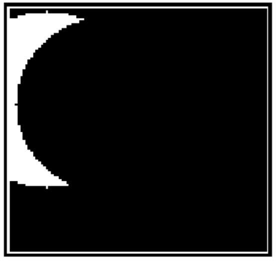
1500 circles
aborted

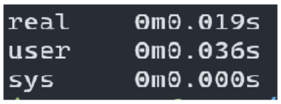


```
real	3m24.607s
user	3m24.606s
sys	0m0.001s
```

Bogo feedback - Accuracy version The second version of the modded bogo place. This algorithm place circles at random until the desired accuracy is hit. This algorithm gave more desirable results and is easier to handle as you can decide the accuracy yourself, instead of having to 'pick and place' with *x* amount of circles for subpar results with the Circle version.

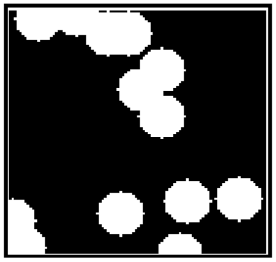
50 % Accuracy
3 circles

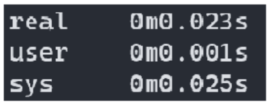




```
real    0m0.019s
user    0m0.036s
sys     0m0.000s
```

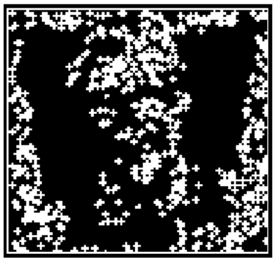
60 % Accuracy
15 circles

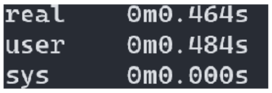




```
real    0m0.023s
user    0m0.001s
sys     0m0.025s
```

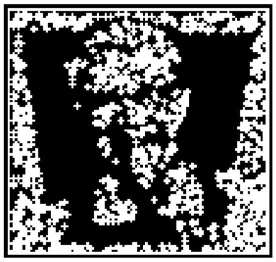
70% Accuracy
615 circles

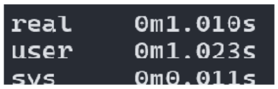




```
real    0m0.464s
user    0m0.484s
sys     0m0.000s
```

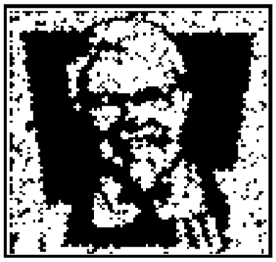
80 % Accuracy
1041 circles

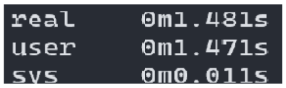




```
real    0m1.010s
user    0m1.023s
sys     0m0.011s
```

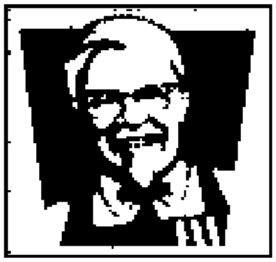
90 % Accuracy
1627 circles

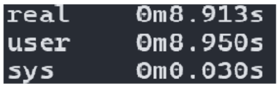




```
real    0m1.481s
user    0m1.471s
sys     0m0.011s
```

99 % Accuracy
2313 circles

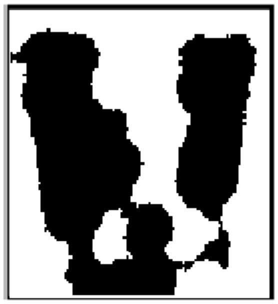


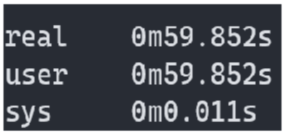


```
real    0m8.913s
user    0m8.950s
sys     0m0.030s
```

Directed random place This algorithm has two main changes from the Bogo feedback, for every circle, the algorithm will iterate through *n* number of circles of the biggest size. For every circle, it compares the difference by placing a white or black circle. When it places a circle, *n* will be set to 0, if it does not, it will change *n* to 1. When the radius is 1, it will iterate through until it cannot find any more circles to place after *n+1* attempts. This algorithm takes longer to compute than the other two algorithms, but it is a more thorough algorithm than the others. It breaks the process in the area of 820-900 circles as it seems like it is happy with the results that is achieved by then.

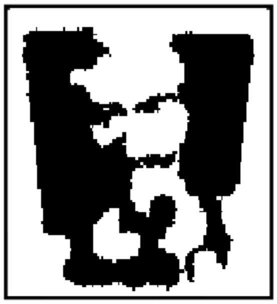
300 Circles

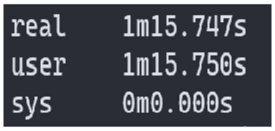




```
real	0m59.852s
user	0m59.852s
sys	0m0.011s
```

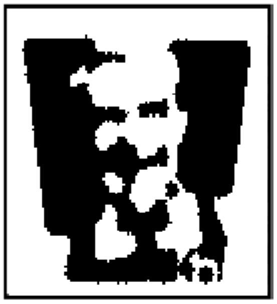
500 Circles

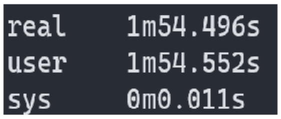




```
real	1m15.747s
user	1m15.750s
sys	0m0.000s
```

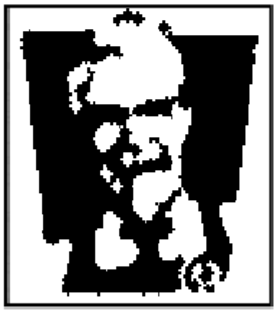
600 Circles

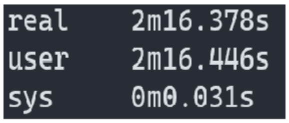




```
real	1m54.496s
user	1m54.552s
sys	0m0.011s
```

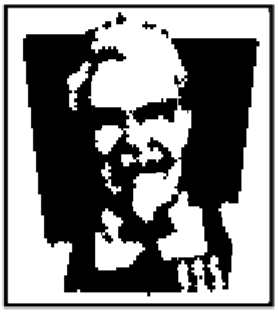
700 Circles

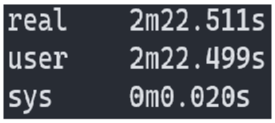




```
real	2m16.378s
user	2m16.446s
sys	0m0.031s
```

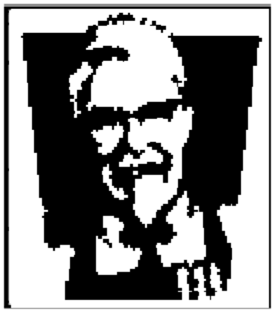
800 Circles

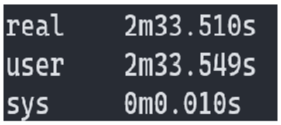




```
real	2m22.511s
user	2m22.499s
sys	0m0.020s
```

859 Circles





```
real	2m33.510s
user	2m33.549s
sys	0m0.010s
```

CHAPTER FIVE

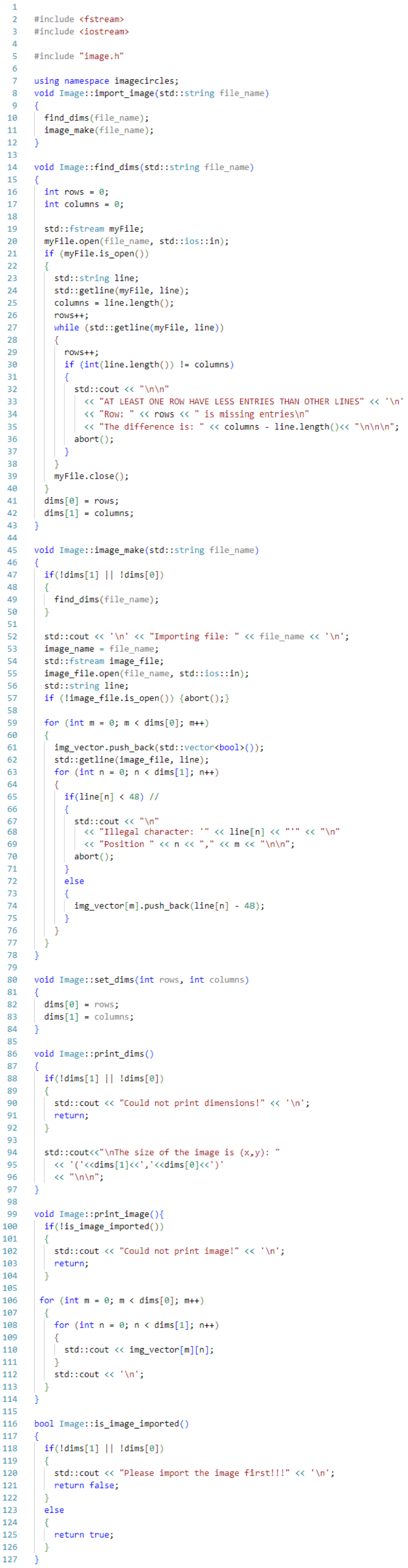
SCRIPTS

CHAPTER

SIX

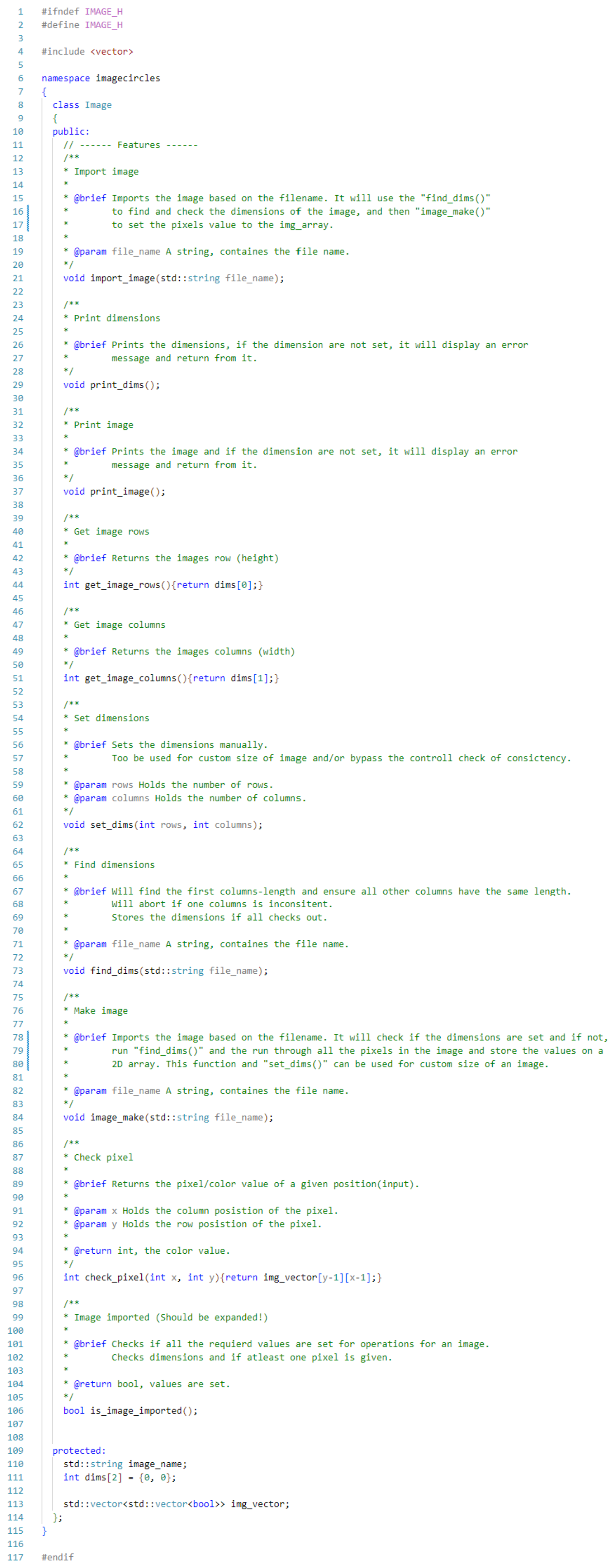
RELATED TO IMAGE

`Image.cpp`



```cpp

#include <fstream>
#include <iostream>

#include "image.h"

using namespace imagecircles;
void Image::import_image(std::string file_name)
{
  find_dims(file_name);
  image_make(file_name);
}

void Image::find_dims(std::string file_name)
{
  int rows = 0;
  int columns = 0;

  std::fstream myFile;
  myFile.open(file_name, std::ios::in);
  if (myFile.is_open())
  {
    std::string line;
    std::getline(myFile, line);
    columns = line.length();
    rows++;
    while (std::getline(myFile, line))
    {
      rows++;
      if (int(line.length()) != columns)
      {
        std::cout << "\n\n"
          << "AT LEAST ONE ROW HAVE LESS ENTRIES THAN OTHER LINES" << '\n'
          << "Row: " << rows << " is missing entries\n"
          << "The difference is: " << columns - line.length()<< "\n\n\n";
        abort();
      }
    }
    myFile.close();
  }
  dims[0] = rows;
  dims[1] = columns;
}

void Image::image_make(std::string file_name)
{
  if(!dims[1] || !dims[0])
  {
    find_dims(file_name);
  }

  std::cout << '\n' << "Importing file: " << file_name << '\n';
  image_name = file_name;
  std::fstream image_file;
  image_file.open(file_name, std::ios::in);
  std::string line;
  if (!image_file.is_open()) {abort();}

  for (int m = 0; m < dims[0]; m++)
  {
    img_vector.push_back(std::vector<bool>());
    std::getline(image_file, line);
    for (int n = 0; n < dims[1]; n++)
    {
      if(line[n] < 48) //
      {
        std::cout << "\n"
          << "Illegal character: '" << line[n] << "'" << "\n"
          << "Position " << n << "," << m << "\n\n";
        abort();
      }
      else
      {
        img_vector[m].push_back(line[n] - 48);
      }
    }
  }
}

void Image::set_dims(int rows, int columns)
{
  dims[0] = rows;
  dims[1] = columns;
}

void Image::print_dims()
{
  if(!dims[1] || !dims[0])
  {
    std::cout << "Could not print dimensions!" << '\n';
    return;
  }

  std::cout<<"\nThe size of the image is (x,y): "
    << '('<<dims[1]<<','<<dims[0]<<')'
    << "\n\n";
}

void Image::print_image(){
  if(!is_image_imported())
  {
    std::cout << "Could not print image!" << '\n';
    return;
  }

 for (int m = 0; m < dims[0]; m++)
  {
    for (int n = 0; n < dims[1]; n++)
    {
      std::cout << img_vector[m][n];
    }
    std::cout << '\n';
  }
}

bool Image::is_image_imported()
{
  if(!dims[1] || !dims[0])
  {
    std::cout << "Please import the image first!!!" << '\n';
    return false;
  }
  else
  {
    return true;
  }
}
```

`Image.h`



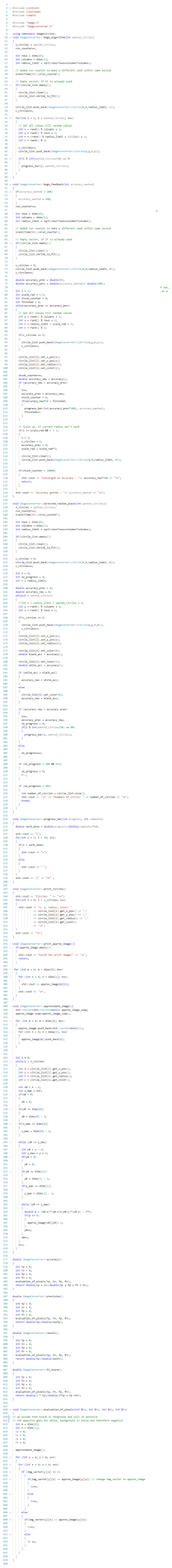
```
#ifndef IMAGE_H
#define IMAGE_H

#include <vector>

namespace imagecircles
{
  class Image
  {
  public:
    // ------ Features ------
    /**
    * Import image
    *
    * @brief Imports the image based on the filename. It will use the "find_dims()"
    *        to find and check the dimensions of the image, and then "image_make()"
    *        to set the pixels value to the img_array.
    *
    * @param file_name A string, containes the file name.
    */
    void import_image(std::string file_name);

    /**
    * Print dimensions
    *
    * @brief Prints the dimensions, if the dimension are not set, it will display an error
    *        message and return from it.
    */
    void print_dims();

    /**
    * Print image
    *
    * @brief Prints the image and if the dimension are not set, it will display an error
    *        message and return from it.
    */
    void print_image();

    /**
    * Get image rows
    *
    * @brief Returns the images row (height)
    */
    int get_image_rows(){return dims[0];}

    /**
    * Get image columns
    *
    * @brief Returns the images columns (width)
    */
    int get_image_columns(){return dims[1];}

    /**
    * Set dimensions
    *
    * @brief Sets the dimensions manually.
    *        Too be used for custom size of image and/or bypass the controll check of consictency.
    *
    * @param rows Holds the number of rows.
    * @param columns Holds the number of columns.
    */
    void set_dims(int rows, int columns);

    /**
    * Find dimensions
    *
    * @brief Will find the first columns-length and ensure all other columns have the same length.
    *        Will abort if one columns is inconsitent.
    *        Stores the dimensions if all checks out.
    *
    * @param file_name A string, containes the file name.
    */
    void find_dims(std::string file_name);

    /**
    * Make image
    *
    * @brief Imports the image based on the filename. It will check if the dimensions are set and if not,
    *        run "find_dims()" and the run through all the pixels in the image and store the values on a
    *        2D array. This function and "set_dims()" can be used for custom size of an image.
    *
    * @param file_name A string, containes the file name.
    */
    void image_make(std::string file_name);

    /**
    * Check pixel
    *
    * @brief Returns the pixel/color value of a given position(input).
    *
    * @param x Holds the column posistion of the pixel.
    * @param y Holds the row posistion of the pixel.
    *
    * @return int, the color value.
    */
    int check_pixel(int x, int y){return img_vector[y-1][x-1];}

    /**
    * Image imported (Should be expanded!)
    *
    * @brief Checks if all the requierd values are set for operations for an image.
    *        Checks dimensions and if atleast one pixel is given.
    *
    * @return bool, values are set.
    */
    bool is_image_imported();


  protected:
    std::string image_name;
    int dims[2] = {0, 0};

    std::vector<std::vector<bool>> img_vector;
  };
}

#endif
```

CHAPTER

SEVEN

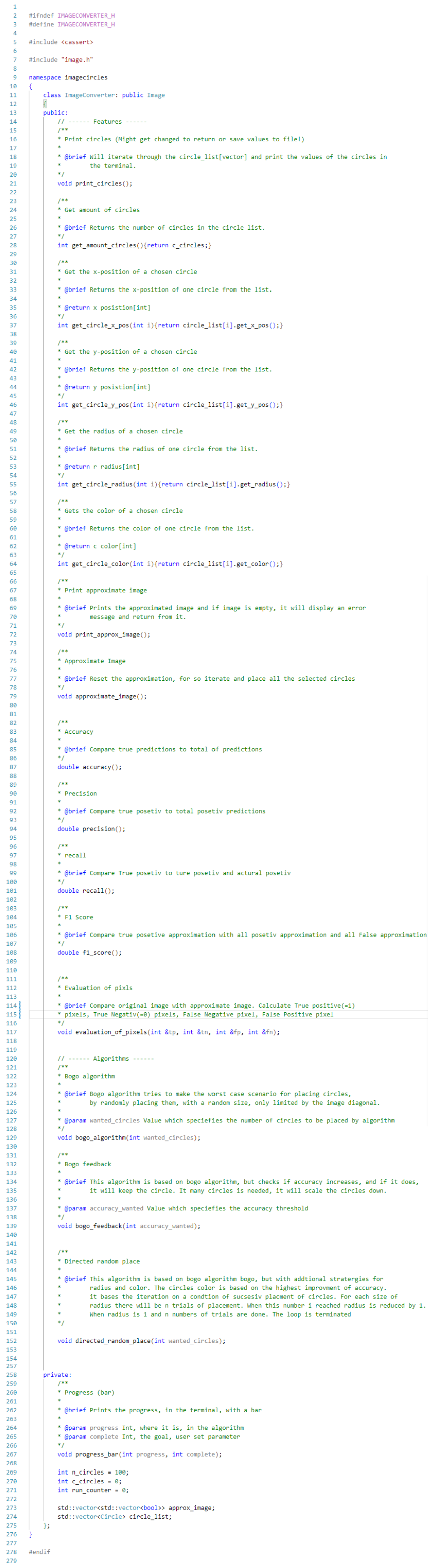
RELATED TO IMAGECONVERTER

`imageconverter.cpp`



```cpp

#include <cstdlib>
#include <iostream>
#include <cmath>

#include "image.h"
#include "imageconverter.h"

using namespace imagecircles;
void ImageConverter::bogo_algorithm(int wanted_circles)
{
  n_circles = wanted_circles;
  run_counter++;

  int rows = dims[0];
  int columns = dims[1];
  int radius_limit = sqrt(rows*rows+columns*columns);

  // Added run counter to make a different seed within same second
  srand(time(NULL)+run_counter);

  // Empty vector, if it is already used
  if(!circle_list.empty())
  {
    circle_list.clear();
    circle_list.shrink_to_fit();
  }

  circle_list.push_back(ImageConverter::Circle(0,0,radius_limit, 1));
  c_circles++;

  for(int i = 1; i < wanted_circles; i++)
  {
    // Set all values till random values
    int x = rand() % columns + 1;
    int y = rand() % rows + 1;
    int r = (rand() % radius_limit + 1)/(i+1) + 1;
    int c = rand() % 2;

    c_circles++;
    circle_list.push_back(ImageConverter::Circle(x,y,r,c));

    if(i % int(wanted_circles/10) == 0)
    {
      progress_bar(i, wanted_circles);
    }
  }
}

void ImageConverter::bogo_feedback(int accuracy_wanted)
{
  if(accuracy_wanted > 100)
  {
    accuracy_wanted = 100;
  }
  run_counter++;
                                                                                y.
  int rows = dims[0];
  int columns = dims[1];
  int radius_limit = sqrt(rows*rows+columns*columns);

  // Added run counter to make a different seed within same second
  srand(time(NULL)+run_counter);

  // Empty vector, if it is already used
  if(!circle_list.empty())
  {
    circle_list.clear();
    circle_list.shrink_to_fit();
  }

  c_circles = 0;
  circle_list.push_back(ImageConverter::Circle(0,0,radius_limit, 0));
  c_circles++;

  double accuracy_prev = double(0);
  double accuracy_perc = double(accuracy_wanted)/ double(100);
                                                                                  f not,
  int i = 1;                                                                      on a
  int scale_rad = 1.5;
  int stuck_counter = 0;
  int finished = 0;
  while(accuracy_prev <= accuracy_perc)
  {
    // Set all values till random values
    int y = rand() % columns + 1;
    int x = rand() % rows + 1;
    int r = radius_limit / scale_rad + 1;
    int c = rand() % 2;

    if(c_circles == i)
    {
      circle_list.push_back(ImageConverter::Circle(x,y,r,c));
      c_circles++;
    }

    circle_list[i].set_x_pos(x);
    circle_list[i].set_y_pos(y);
    circle_list[i].set_radius(r);
    circle_list[i].set_color(c);

    stuck_counter++;
    double accuracy_new = accuracy();
    if (accuracy_new > accuracy_prev)
    {
      i++;
      accuracy_prev = accuracy_new;
      stuck_counter = 0;
      if(accuracy_new*10 > finished)
      {
        progress_bar(int(accuracy_prev*100), accuracy_wanted);
        finished++;
      }
    }

    // Scale up, if current radius don't work
    if(i >= scale_rad && r > 1)
    {
      i = 1;
      c_circles = 1;
      accuracy_prev = 0;
      scale_rad = scale_rad*2;

      circle_list.clear();
      circle_list.push_back(ImageConverter::Circle(0,0,radius_limit, 0));
    }

    if(stuck_counter > 10000)
    {
      std::cout << "Converged on accuracy : "<< accuracy_new*100 << "\n";
      return;
    }
  }
  std::cout << "Accuracy gotten : "<< accuracy_wanted << "\n";
}

void ImageConverter::directed_random_place(int wanted_circles){
  n_circles = wanted_circles;
  run_counter++;
  srand(time(NULL)+run_counter);

  int rows = dims[0];
  int columns = dims[1];
  int radius_limit = sqrt(rows*rows+columns*columns);

  if(!circle_list.empty())
  {
    circle_list.clear();
    circle_list.shrink_to_fit();
  }

  c_circles = 0;
  circle_list.push_back(ImageConverter::Circle(0,0,radius_limit, 0));
  c_circles++;

  int i = 0;
  int no_progress = 0;
  int r = radius_limit;

  double accuracy_prev = 0;
  double accuracy_new = 0;
  while(i < wanted_circles)
  {
    //int r = radius_limit / wanted_circles + 2;
    int y = rand() % columns + 1;
    int x = rand() % rows + 1;

    if(c_circles == i)
    {
      circle_list.push_back(ImageConverter::Circle(x,y,r,0));
      c_circles++;
    }
    circle_list[i].set_x_pos(x);
    circle_list[i].set_y_pos(y);
    circle_list[i].set_radius(r);

    circle_list[i].set_color(0);
    double black_acc = accuracy();

    circle_list[i].set_color(1);
    double white_acc = accuracy();

    if (white_acc > black_acc)
    {
      accuracy_new = white_acc;
    }
    else
    {
      circle_list[i].set_color(0);
      accuracy_new = black_acc;
    }

    if (accuracy_new > accuracy_prev)
    {
      i++;
      accuracy_prev = accuracy_new;
      no_progress = 0;
      if(i % int(wanted_circles/10) == 00)
      {
        progress_bar(i, wanted_circles);
      }
    }
    else
    {
      no_progress++;
    }

    if (no_progress > 500 && r>2)
    {
      no_progress = 0;
      r--;
    }

    if (no_progress > 501)
    {
      int number_of_circles = circle_list.size();
      std::cout << '\n' << "Numbers of circls: " << number_of_circles << '\n';
      break;
    }
  }
}

void ImageConverter::progress_bar(int progress, int complete)
{
  double work_done = double(progress)/double(complete)*10;

  std::cout << "[";
  for(int i = 1; i < 10; i++)
  {
    if(i < work_done)
    {
      std::cout << "=";
    }
    else
    {
      std::cout << " ";
    }
  }
  std::cout << "]" << "\n" ;
}

void ImageConverter::print_circles()
{
  std::cout << "Circles: " << "\n";
  for(int i = 0; i < c_circles; i++)
  {
    std::cout << "x, y, radius, color: "
              << circle_list[i].get_x_pos() << ","
              << circle_list[i].get_y_pos() << ","
              << circle_list[i].get_radius() << ","
              << circle_list[i].get_color()
              << '\n';
  }
  std::cout << "\n";
}

void ImageConverter::print_approx_image(){
  if(approx_image.empty())
  {
    std::cout << "Could not print image!" << '\n';
    return;
  }

 for (int m = 0; m < dims[0]; m++)
  {
    for (int n = 0; n < dims[1]; n++)
    {
      std::cout << approx_image[m][n];
    }
    std::cout << '\n';
  }
}

void ImageConverter::approximate_image(){
  std::vector<std::vector<bool>> approx_image_swap;
  approx_image.swap(approx_image_swap);

  for (int m = 0; m < dims[0]; m++)
  {
    approx_image.push_back(std::vector<bool>());
    for (int n = 0; n < dims[1]; n++)
    {
      approx_image[m].push_back(0);
    }
  }


  int i = 0;
  while(i < c_circles)
  {
    int x = circle_list[i].get_x_pos();
    int y = circle_list[i].get_y_pos();
    int r = circle_list[i].get_radius();
    int c = circle_list[i].get_color();

    int x0 = x - r;
    int x_max = x+r;
    if(x0 < 0)
    {
      x0 = 0;
    }
    if(x0 >= dims[0])
    {
      x0 = dims[0] - 1;
    }
    if(x_max >= dims[0])
    {
      x_max = dims[0] - 1;
    }

    while (x0 <= x_max)
    {
      int y0 = y - r;
      int y_max = y + r;
      if(y0 < 0)
      {
        y0 = 0;
      }
      if(y0 >= dims[1])
      {
        y0 = dims[1] - 1;
      }
      if(y_max >= dims[1])
      {
        y_max = dims[1] - 1;
      }

      while (y0 <= y_max)
      {
        double p = (x0-x)*(x0-x)+(y0-y)*(y0-y) - r*r;
        if(p <= 0)
        {
          approx_image[x0][y0]= c;
        }
        y0++;
      }
      x0++;
    }
    i++;
  }
}

double ImageConverter::accuracy()
{
  int tp = 0;
  int tn = 0;
  int fp = 0;
  int fn = 0;
  evaluation_of_pixels(tp, tn, fp, fn);
  return double(tp + tn)/double(tp + fp + fn + tn);
}

double ImageConverter::precision()
{
  int tp = 0;
  int tn = 0;
  int fp = 0;
  int fn = 0;
  evaluation_of_pixels(tp, tn, fp, fn);
  return double(tp)/double(tp+fp);
}

double ImageConverter::recall()
{
  int tp = 0;
  int tn = 0;
  int fp = 0;
  int fn = 0;
  evaluation_of_pixels(tp, tn, fp, fn);
  return double(tp)/double(tp+fn);
}

double ImageConverter::f1_score()
{
  int tp = 0;
  int tn = 0;
  int fp = 0;
  int fn = 0;
  evaluation_of_pixels(tp, tn, fp, fn);
  return double(2 * tp)/double(2*tp + fp +fn);
}


void ImageConverter::evaluation_of_pixels(int &tp, int &tn, int &fp, int &fn)
{
// we assume that black is forground and call it positive
// the opposite goes for white, background is white and therefore negative
  int m = dims[0];
  int n = dims[1];
  tp = 0;
  tn = 0;
  fp = 0;
  fn = 0;

  approximate_image();

  for (int y = 0; y < m; y++)
  {
    for (int x = 0; x < n; x++)
    {
      if (img_vector[y][x] == 1)
      {
        if(img_vector[y][x] == approx_image[y][x]) // change img_vector to approx_image
        {
          tp++;
        }
        else
        {
          fn++;
        }
      }
      else
      {
        if(img_vector[y][x] == approx_image[y][x])
        {
          tn++;
        }
        else
        {
          fp ++;
        }
      }
    }
  }
}

```

```
imageconverter.h
```



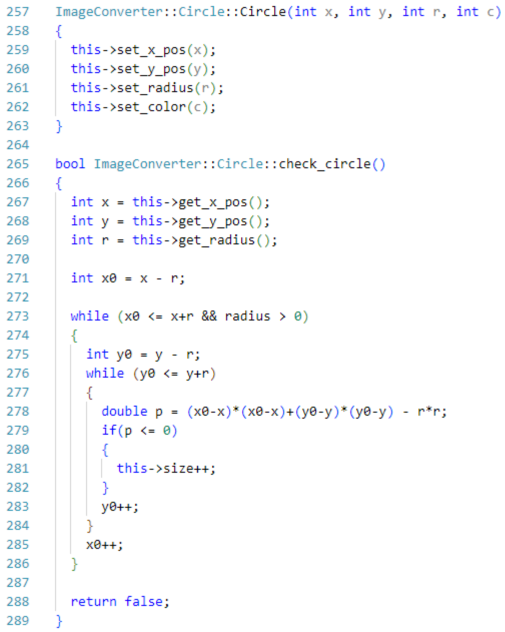
```
#ifndef IMAGECONVERTER_H
#define IMAGECONVERTER_H

#include <cassert>

#include "image.h"

namespace imagecircles
{
    class ImageConverter: public Image
    {
    public:
        // ------ Features ------
        /**
        * Print circles (Might get changed to return or save values to file!)
        *
        * @brief Will iterate through the circle_list[vector] and print the values of the circles in
        *        the terminal.
        */
        void print_circles();

        /**
        * Get amount of circles
        *
        * @brief Returns the number of circles in the circle list.
        */
        int get_amount_circles(){return c_circles;}

        /**
        * Get the x-position of a chosen circle
        *
        * @brief Returns the x-position of one circle from the list.
        *
        * @return x posistion[int]
        */
        int get_circle_x_pos(int i){return circle_list[i].get_x_pos();}

        /**
        * Get the y-position of a chosen circle
        *
        * @brief Returns the y-position of one circle from the list.
        *
        * @return y posistion[int]
        */
        int get_circle_y_pos(int i){return circle_list[i].get_y_pos();}

        /**
        * Get the radius of a chosen circle
        *
        * @brief Returns the radius of one circle from the list.
        *
        * @return r radius[int]
        */
        int get_circle_radius(int i){return circle_list[i].get_radius();}

        /**
        * Gets the color of a chosen circle
        *
        * @brief Returns the color of one circle from the list.
        *
        * @return c color[int]
        */
        int get_circle_color(int i){return circle_list[i].get_color();}

        /**
        * Print approximate image
        *
        * @brief Prints the approximated image and if image is empty, it will display an error
        *        message and return from it.
        */
        void print_approx_image();

        /**
        * Approximate Image
        *
        * @brief Reset the approximation, for so iterate and place all the selected circles
        */
        void approximate_image();


        /**
        * Accuracy
        *
        * @brief Compare true predictions to total of predictions
        */
        double accuracy();

        /**
        * Precision
        *
        * @brief Compare true posetiv to total posetiv predictions
        */
        double precision();

        /**
        * recall
        *
        * @brief Compare True posetiv to ture posetiv and actural posetiv
        */
        double recall();

        /**
        * F1 Score
        *
        * @brief Compare true posetive approximation with all posetiv approximation and all False approximation
        */
        double f1_score();


        /**
        * Evaluation of pixls
        *
        * @brief Compare original image with approximate image. Calculate True positive(=1)
        * pixels, True Negativ(=0) pixels, False Negative pixel, False Positive pixel
        */
        void evaluation_of_pixels(int &tp, int &tn, int &fp, int &fn);


        // ------ Algorithms ------
        /**
        * Bogo algorithm
        *
        * @brief Bogo algorithm tries to make the worst case scenario for placing circles,
        *        by randomly placing them, with a random size, only limited by the image diagonal.
        *
        * @param wanted_circles Value which speciefies the number of circles to be placed by algorithm
        */
        void bogo_algorithm(int wanted_circles);

        /**
        * Bogo feedback
        *
        * @brief This algorithm is based on bogo algorithm, but checks if accuracy increases, and if it does,
        *        it will keep the circle. It many circles is needed, it will scale the circles down.
        *
        * @param accuracy_wanted Value which speciefies the accuracy threshold
        */
        void bogo_feedback(int accuracy_wanted);


        /**
        * Directed random place
        *
        * @brief This algorithm is based on bogo algorithm bogo, but with addtional stratergies for
        *        radius and color. The circles color is based on the highest improvment of accuracy.
        *        it bases the iteration on a condtion of sucsesiv placment of circles. For each size of
        *        radius there will be n trials of placement. When this number i reached radius is reduced by 1.
        *        When radius is 1 and n numbers of trials are done. The loop is terminated
        */

        void directed_random_place(int wanted_circles);



    private:
        /**
        * Progress (bar)
        *
        * @brief Prints the progress, in the terminal, with a bar
        *
        * @param progress Int, where it is, in the algorithm
        * @param complete Int, the goal, user set parameter
        */
        void progress_bar(int progress, int complete);

        int n_circles = 100;
        int c_circles = 0;
        int run_counter = 0;

        std::vector<std::vector<bool>> approx_image;
        std::vector<Circle> circle_list;
    };
}

#endif
```

CHAPTER

EIGHT

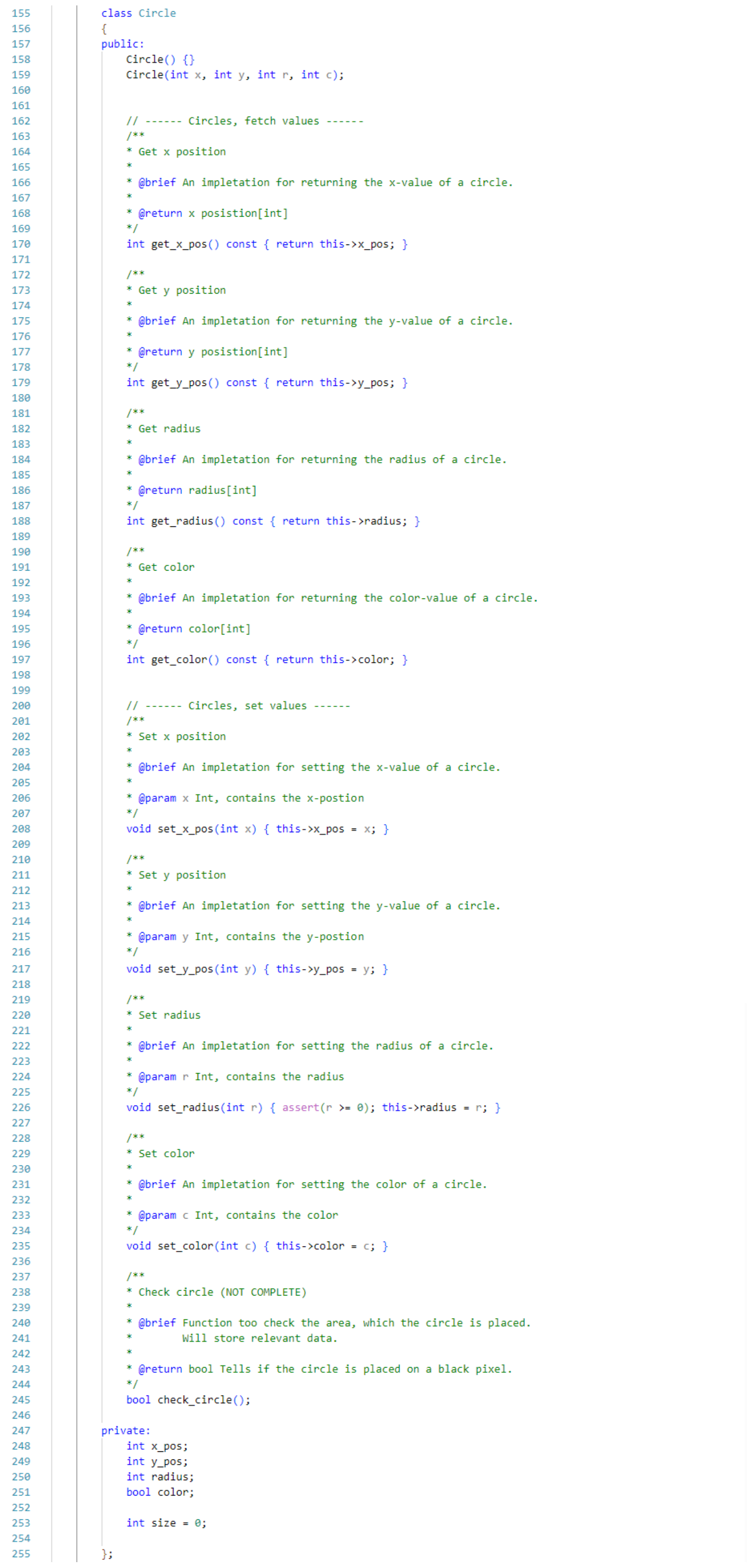
RELATED TO CIRCLE

`imageconverter.cpp` Circle class



```
ImageConverter::Circle::Circle(int x, int y, int r, int c)
{
  this->set_x_pos(x);
  this->set_y_pos(y);
  this->set_radius(r);
  this->set_color(c);
}

bool ImageConverter::Circle::check_circle()
{
  int x = this->get_x_pos();
  int y = this->get_y_pos();
  int r = this->get_radius();

  int x0 = x - r;

  while (x0 <= x+r && radius > 0)
  {
    int y0 = y - r;
    while (y0 <= y+r)
    {
      double p = (x0-x)*(x0-x)+(y0-y)*(y0-y) - r*r;
      if(p <= 0)
      {
        this->size++;
      }
      y0++;
    }
    x0++;
  }

  return false;
}
```

`imageconverter.h` Circle Header



```
class Circle
{
public:
    Circle() {}
    Circle(int x, int y, int r, int c);


    // ------ Circles, fetch values ------
    /**
    * Get x position
    *
    * @brief An impletation for returning the x-value of a circle.
    *
    * @return x posistion[int]
    */
    int get_x_pos() const { return this->x_pos; }

    /**
    * Get y position
    *
    * @brief An impletation for returning the y-value of a circle.
    *
    * @return y posistion[int]
    */
    int get_y_pos() const { return this->y_pos; }

    /**
    * Get radius
    *
    * @brief An impletation for returning the radius of a circle.
    *
    * @return radius[int]
    */
    int get_radius() const { return this->radius; }

    /**
    * Get color
    *
    * @brief An impletation for returning the color-value of a circle.
    *
    * @return color[int]
    */
    int get_color() const { return this->color; }


    // ------ Circles, set values ------
    /**
    * Set x position
    *
    * @brief An impletation for setting the x-value of a circle.
    *
    * @param x Int, contains the x-postion
    */
    void set_x_pos(int x) { this->x_pos = x; }

    /**
    * Set y position
    *
    * @brief An impletation for setting the y-value of a circle.
    *
    * @param y Int, contains the y-postion
    */
    void set_y_pos(int y) { this->y_pos = y; }

    /**
    * Set radius
    *
    * @brief An impletation for setting the radius of a circle.
    *
    * @param r Int, contains the radius
    */
    void set_radius(int r) { assert(r >= 0); this->radius = r; }

    /**
    * Set color
    *
    * @brief An impletation for setting the color of a circle.
    *
    * @param c Int, contains the color
    */
    void set_color(int c) { this->color = c; }

    /**
    * Check circle (NOT COMPLETE)
    *
    * @brief Function too check the area, which the circle is placed.
    *        Will store relevant data.
    *
    * @return bool Tells if the circle is placed on a black pixel.
    */
    bool check_circle();

private:
    int x_pos;
    int y_pos;
    int radius;
    bool color;

    int size = 0;

};
```

CHAPTER

NINE

IMPROVEMENTS & IDEAS

Implementation of parameters 1. stride of pixel for evaluation When we evaluate each circle we do *not* need to evaluate every pixel. This can also be dependent on the current status of the progress. When we initialize an algorithm we can assume that it is not neccessary to check every pixel to achieve a placement of increasing score. An alternative to the current solution could therefore be changed to evaluate a portion of the selected circe then gradually increasing this portion. By some experimenation this could yield results much like what was already achieved.

2. number of no-improvements The number of interations of no improvement will wary with the content of the image and the shape. It would be interesting to give the user the ability to change this as a parameter. By some experimentation we could also define a standard analsys of the image to find some parameters to ensure that the algorithm running smoothly.

3. increments of radius Most of the same ideas from above apply to radius as well. Instad of the numbers you could implement different functions for reducing the length.

Imageconverter 4. Backwards evaluation of circle relevance We can assume that the models already made will place circles that over time will be over written by other circles. This can be solved by simply iterate over the circles backwards. The strategy can be summed up to: Define an emtpy image of 0. Place the circles one by one as the same colors. Every time this places an circle and none of the pixels change any value in the image, we can remove this circle.

5. Improve evaluation method The current implementation of the evaluation method reproduces the approximation of the image based on every circle in the list every time. Over the course of the iterations, this will result in performance of O (**number of circles** $*$ **number of placement attempts**). This can be rewritten to *deep copy* the last approximation and placing only one circle.

6. Algorithms Based on the strcture we can freely add other Algorithms: One idea we had is to make an algorithm based on dividing the image into subimages, this enables us to implement other algorithms like:

a: at a chosen condition split the image based on the longest axis into two sub images. This will result in 2^k subimages for *k* numbers of splitting. b: implement a procedure of placing circles, for example placing *n* numbers of circles filling the majority of the sub image c: repeat a and b until we run out of circles to place or a specified size of subimage.

CHAPTER TEN

README

```
Pix2circ Group23
term assignment by Jon Augensen, Ole Benjamin Gauslaa and Lars Øvergård!
```

Steps to run our code:

Change directory to our pix2circ directory and build it by using *make*:

```
cd pix2circ/
make
```

We have 3 working algorithms you can choose from. We have assigned numbers to chose which algorithm you want to use. We have added two images, the batman logo and the kfc logo, converted them into 0's and 1's and exported them to a .txt file. Namely batman.txt and kfc.txt, the batman logo is a bit small, the results are shown better with the kfc image.

Bogo place :1 Directed random place :2 Bogo feedback :3

The different algorithms work in different ways, for Bogo place and the *Directed random place* algorithms, you have to assign how many circles you want to run with. For the *Bogo feedback - accuracy*, you assign the accuracy that you want it to run with, with a number between 1-99, the higher the number, the better accuracy you will get. After you have run the program, it will generate a .png image in the pix2circ folder.

Example for running Bogo place:

```
./pix2circ kfc.txt 1 500
```

This will run the *bogo place* algorithm, on the kfc image, with 500 circles.

Example for running Directed random place:

```
./pix2circ batman.txt 2 1000
```

This will run the *directed random place* algorithm, on the batman logo, with 1000 circles.|br|

Example for running Bogo feedback:

```
./pix2circ kfc.txt 3 99
```

This will run the *bogo feedback* algorithm, on the kfc image, with 99% accuracy.

Documentation The work is documented using Sphinx, it is expected and required that Sphinx is set up to view the documentation. change directory to the *docs* directory and make the documentation, followed by changing directory to *html* via the *_build* directory and open the index.html file.

```
cd docs/
make html
cd _build/html
```

CHAPTER ELEVEN

ABOUT US

Jon Augensen, 4th year applied robotics student. 34 years old. E-mail address: jon.eirik.augensen@nmbu.no

Ole Benjamin, 4th year data scientist student. 24 years old E-mail address: ole.benjamin.gauslaa@nmbu.no

Lars Øvergård, 4th year applied robotics student. 24 years old. E-mail address: lars.overgard@hotmail.com

CHAPTER TWELVE

COPYRIGHTS & LICENSES

Martin Horsch - our lecturer who has provided us with **charmap.cpp**, **disk-vector.cpp** with corresponding header files **charmap.h** and **disk-vector.h**. We have chosen to not include these in our code documentation because of the fact that this is not our code. We do not own rights to this code, the copyright is held by Martin Horsch himself.

Our program is written under the **CC BY-NC-SA 4.0** license which means that the user may use, share and adapt it to your liking. However, if used in this way you need to give appropriate credit, and indicate which parts of the material has been modified. Using the material for commercial purposes is a violation of the license. The licence is also added to our material.

INDEX