Genetic Evolution of a Picture

Basic Approach

I tried six approaches:

- **v1-circles**

  Initial implementation based on circles with naïve fitness function

  - variable number of candidates kept
  - variable number of circles per candidate
  - mutation occurs by selecting random circles from each candidate and randomizing
  - children are made by combining both parents' shapes but at half visibility

- **v2-pixels**

  Pixel-based candidates with pixel mutation and more sensitive fitness function

  - mutates pixel array directly by changing random pixels

- **v3-generational**

  Pixel-based still, but each generation is created anew — parents are not kept

  - otherwise the same as v2
  - The 'sexiest' X candidates with a greater number of Y candidates. Similar to reality, the more 'fit' candidates reproduce more widely but still tend to reproduce with other 'fit' candidates

- **v4-reprovariety**

  Added more variety to the repository to avoid local maxima harming evolution

    - In addition to some X and Y of ‘sexy’ candidates reproducing, other candidates have a variable reproduction rate (probability directly proportional to their relative fitness)
- **v5-pixelCircles**

  Reverted to circle-based mutation, but with pixel-based candidates
    - Crashes readily — instead of bug-fixing, I reverted to circle-based candidates
- **v6-circlesImproved**

  Circle-based candidates and circle-based mutation (based on v1)
    - Fixed number of candidates per round
    - Atomic generation of each new round — mutations and children kept, but parents die
    - Static fitness calculation to speed processing
    - Duplicate candidate removal (to ease away from local maxima)
    - Candidate crossover through shape combining (half-way between shape attributes)
    - Circle comparison for sorting and fuzzy equality checking

Quantitative measurements

Keep in mind that the programs I wrote actually generate quantitative data about the candidates in their candidate pool. Unfortunately the file with this data was lost and I didn’t have time to re-collect the info. However, it would be interesting to see how the various algorithms / constant combinations compare as judged by a naïve fitness function.

Results — v6-circlesImproved

This is the most successful algorithm that I developed. As you can see, it quickly gets to a reasonable approximation of the shape. There are a few interesting variations in the algorithm's performance against the various images. It gets closer to a good result on the first and third because they have greater color locality. However, it more quickly gets to something recognizable in the first and second images because they have a more consistent color scheme (green and red, respectively). Below you will see various iterations with different content values.

Running the system with various parameters

If you have git, you can run different versions of the code by running one of:

```
git checkout v1-circles
git checkout v2-pixels
git checkout v3-generational
git checkout v4-reprovariety
git checkout v5-pixelCircles
git checkout v6-circlesImproved
```

The focus of this report is primarily on *v6-circlesImproved*, the most successful of my implementations. In this code, the constants can be varied at the top of the code.

These are the key variables:

- POPULATION_SIZE_INITIAL

    - Controls the number of pseudo-random candidates in the initial population
    - High numbers are useful for quickly getting to a good start point
    - Can be larger than SURVIVAL_MAX
- SURVIVAL_PROB_KEEP
    - Probability that a given candidate will survive from round to round
    - Lower numbers help fight off local maxima
- SURVIVAL_PROB_MUTATE
    - Probability that a given survivor will mutate from round to round
    - Lower numbers lead to increased stability; higher numbers introduce new features
- SURVIVAL_MAX
    - Maximum number of candidates to keep between rounds
    - Higher numbers will naturally lead to better results, but slow down computation
- BREEDING_ALPHA
    - The small selection of the sexiest ‘mates‘ that are permitted to initiate breeding
- BREEDING_BETA
    - The less small selection mates that the alphas will breed with
- NUM_SHAPES
    - The number of shapes in each candidate
    - More shapes provide greater accuracy, but at the expense of computational power

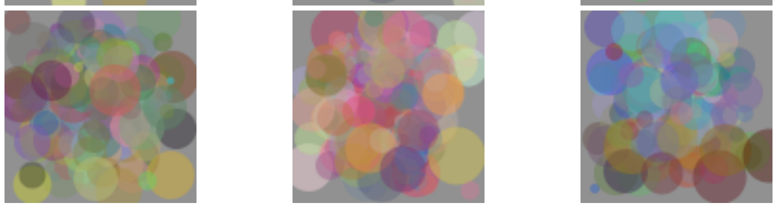
Constant set #1

```
POPULATION_SIZE_INITIAL = 1000;
SURVIVAL_PROB_KEEP = 0.9;  SURVIVAL_PROB_MUTATE = 0.9;  SURVIVAL_MAX = 100;
BREEDING_ALPHA = 10;       BREEDING_BETA = 20;
NUM_SHAPES = 250;
```

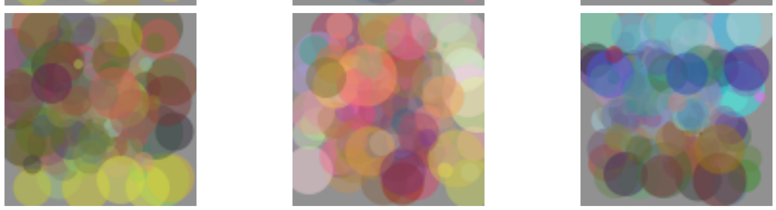
1 iteration

10 iterations

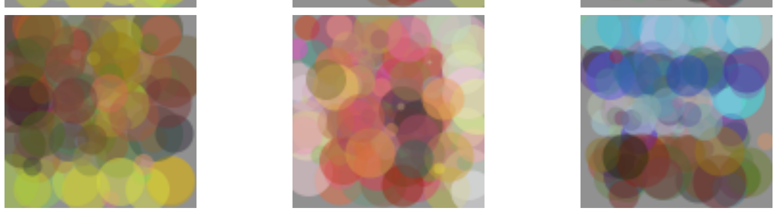
50 iterations



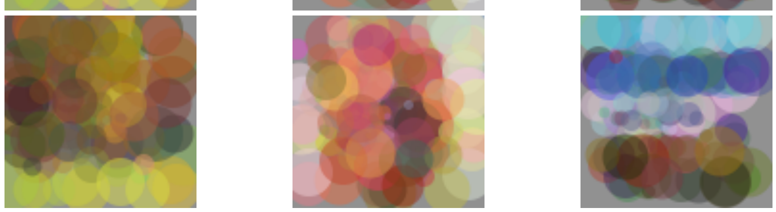
200 iterations



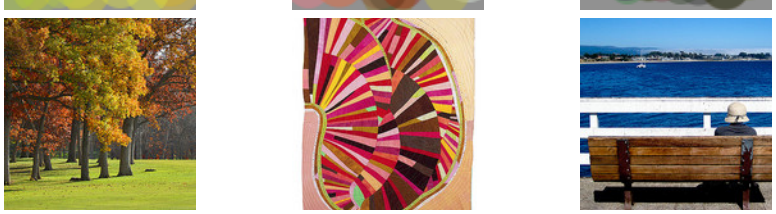
500 iterations



1000 iterations



Actual

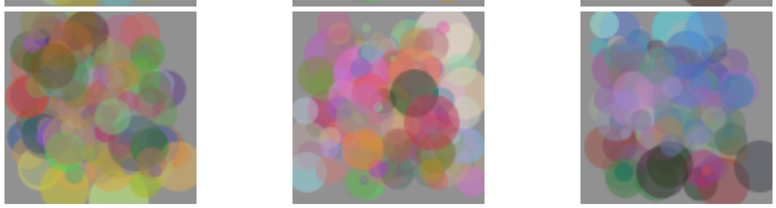


Constant set #2 — Population changed to 50

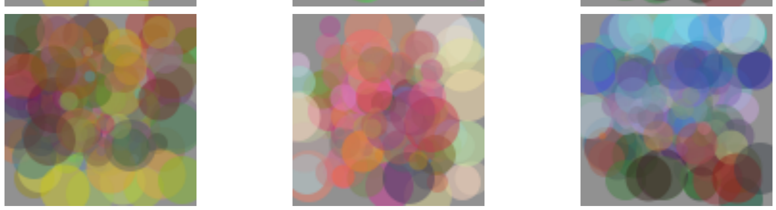
```
POPULATION_SIZE_INITIAL = 1000;
SURVIVAL_PROB_KEEP = 0.9;  SURVIVAL_PROB_MUTATE = 0.9; SURVIVAL_MAX = 50;
BREEDING_ALPHA = 10;       BREEDING_BETA = 20;
NUM_SHAPES = 250;
```

1 iteration

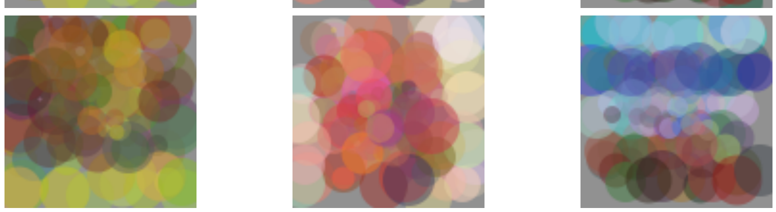
10 iterations



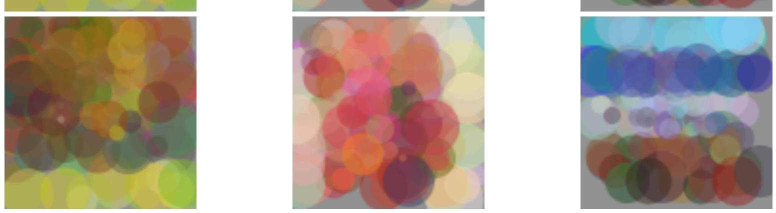
50 iterations



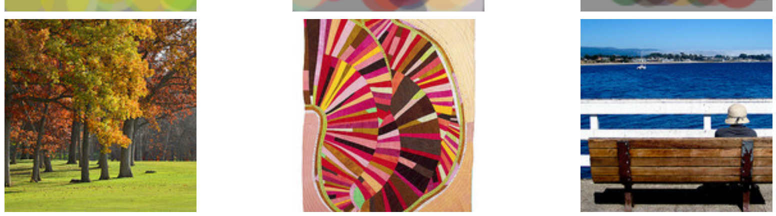
200 iterations



500 iterations



1000 iterations



Actual

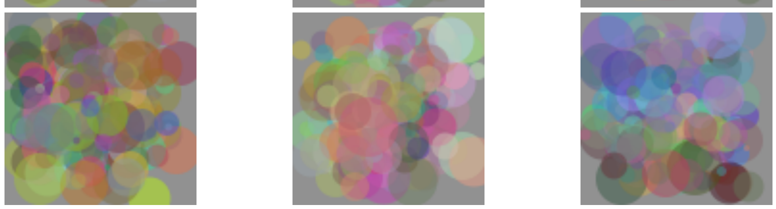
Constant set #3 — Mutation changed to 50%

```
POPULATION_SIZE_INITIAL = 1000;
SURVIVAL_PROB_KEEP = 0.9;  SURVIVAL_PROB_MUTATE = 0.5; SURVIVAL_MAX = 100;
BREEDING_ALPHA = 10;       BREEDING_BETA = 20;
NUM_SHAPES = 250;
```

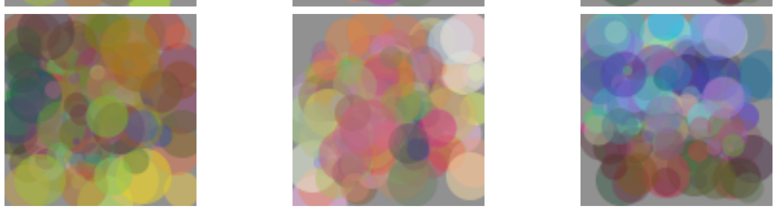
1 iteration

10 iterations

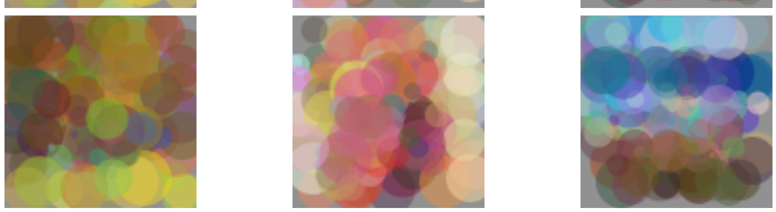
50 iterations



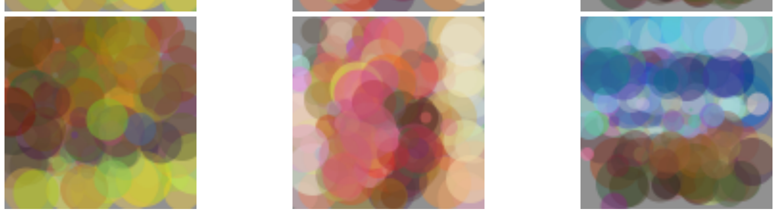
200 iterations



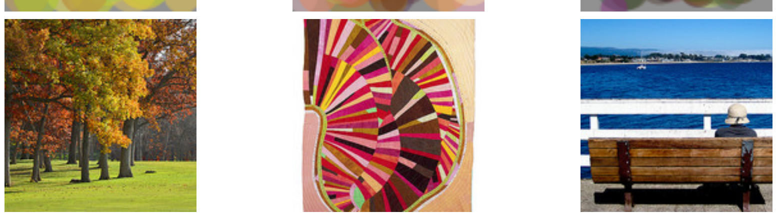
500 iterations



1000 iterations

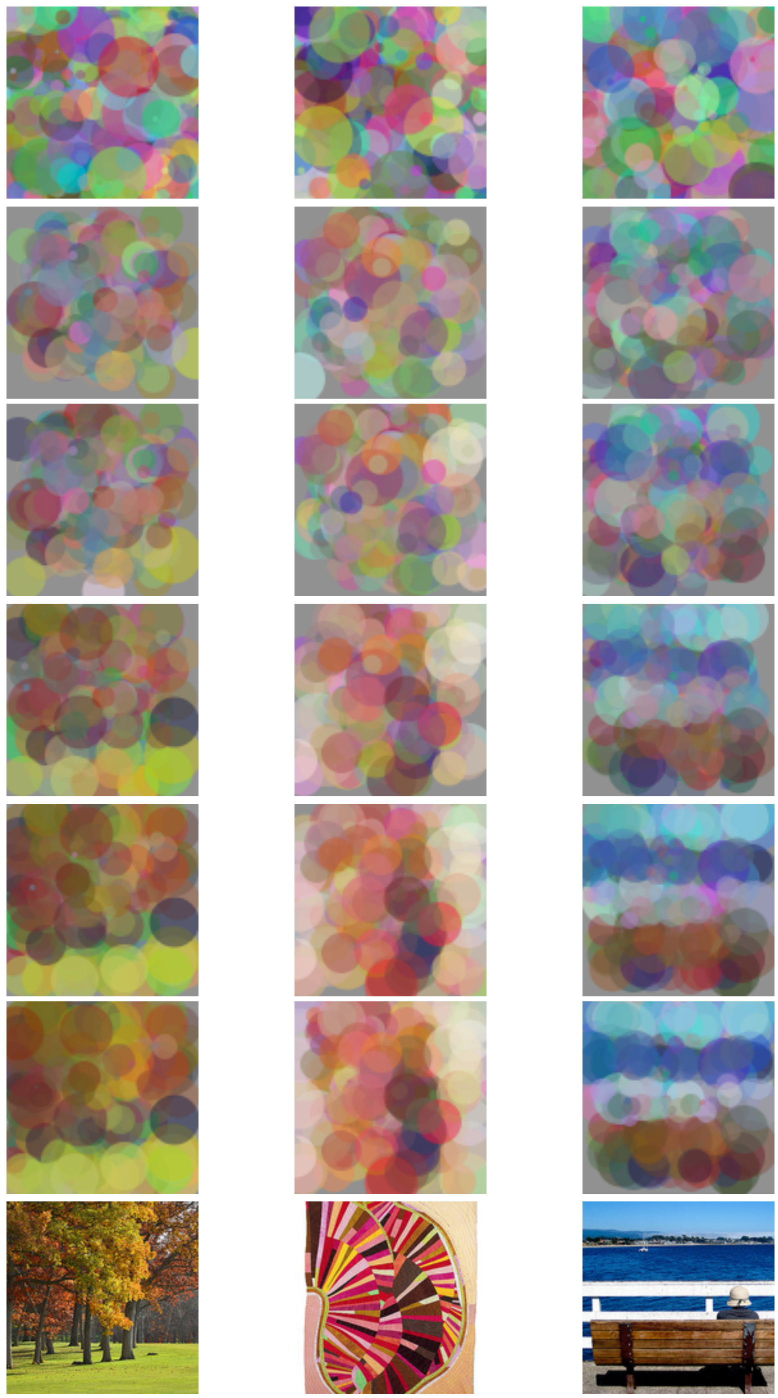


Actual



Constant set #4 — Shapes/Candidate changed to 500

```
POPULATION_SIZE_INITIAL = 1000;
SURVIVAL_PROB_KEEP = 0.9;  SURVIVAL_PROB_MUTATE = 0.9;  SURVIVAL_MAX = 100;
BREEDING_ALPHA = 10;       BREEDING_BETA = 20;
NUM_SHAPES = 500;
```



1 iteration

10 iterations

50 iterations

200 iterations

500 iterations

1000 iterations

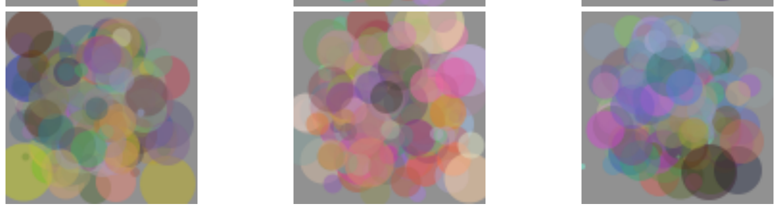
Actual

Constant set #5 — Breeding alpha changed to 20 (wider gene pool)

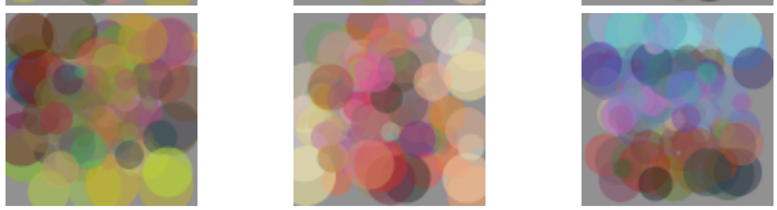
```
POPULATION_SIZE_INITIAL = 1000;
SURVIVAL_PROB_KEEP = 0.9;  SURVIVAL_PROB_MUTATE = 0.9;  SURVIVAL_MAX = 100;
BREEDING_ALPHA = 20;       BREEDING_BETA = 20;
NUM_SHAPES = 250;
```

1 iteration

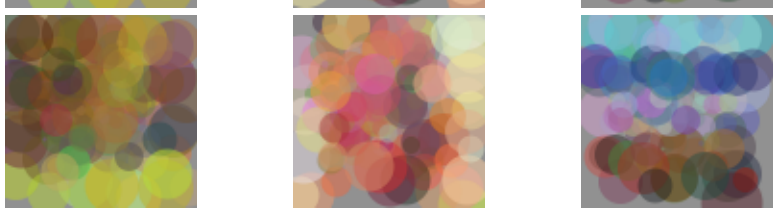
10 iterations



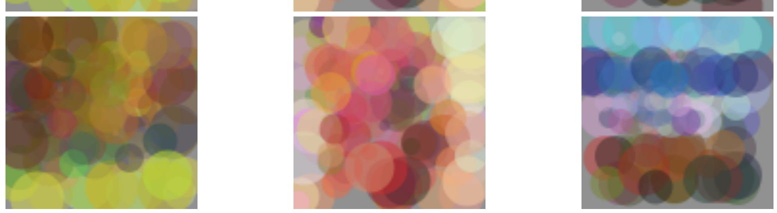
50 iterations



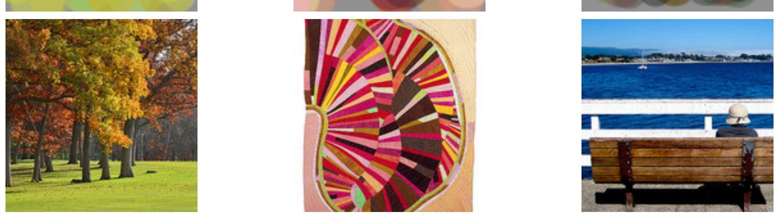
200 iterations



500 iterations



1000 iterations



Actual

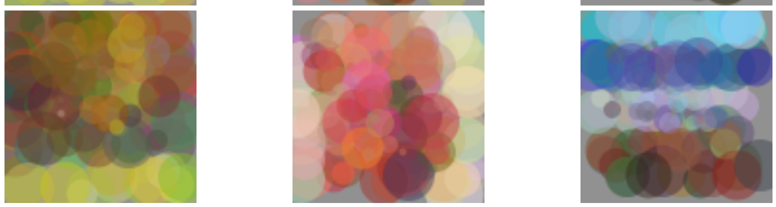
Comparison of constant sets

Data set #1 — All comparisons at 1000 iterations

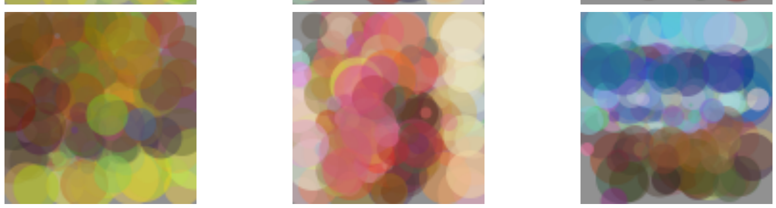
Actual

Constant set #1

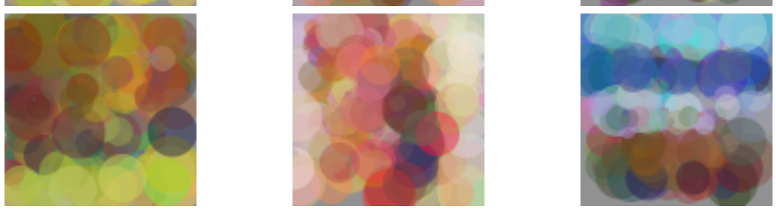
Constant set #2



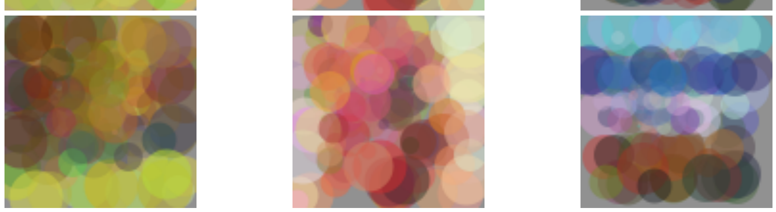
Constant set #3



Constant set #4



Constant set #5

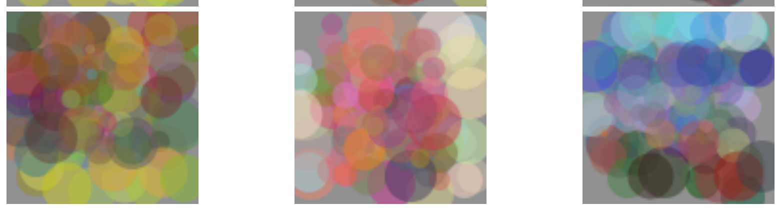


Data set #2 — All comparisons at 200 iterations

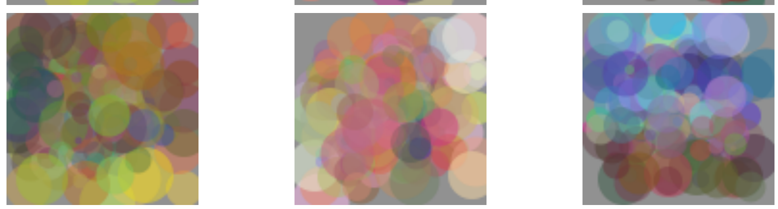
Actual

Constant set #1

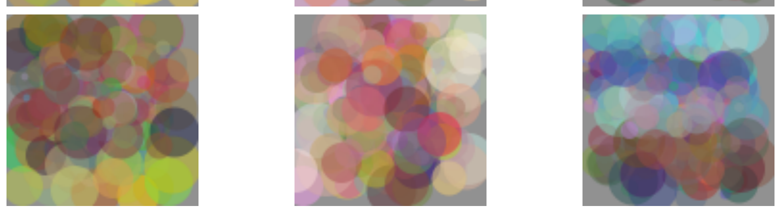
Constant set #2



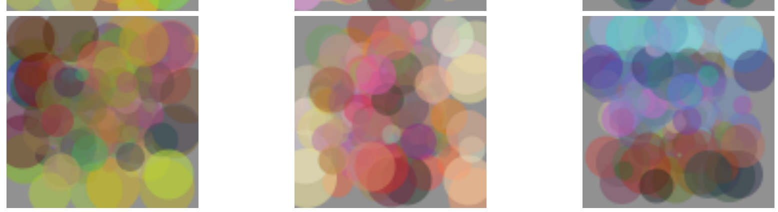
Constant set #3



Constant set #4



Constant set #5



Analysis of Data

Showing this data to 4 people, I was able to get multiple opinions about which approximations look 'best'. The data is included in the Excel file provided. Asking each participant to rank each column 1 to 5 (1 being best), we came up with a few interesting results:

- Algorithm 4 performs consistently well, both in the 200 and 1000 tests
- Algorithms 1 and 2 perform consistently poorly
- Algorithm 3 performs best in the 200 iteration test, but 'average' at 1000 iterations
- Algorithm 5 performs slightly erratically, with high performance at 1000 iterations but low performance at 200 iterations

Average of Rating	
Row Labels	Total
200	3
1	2.222222222
2	2.222222222
3	3.888888889
4	3.777777778
5	2.888888889
1000	3
1	2.222222222
2	2.666666667
3	3
4	3.555555556
5	3.555555556
Grand Total	3