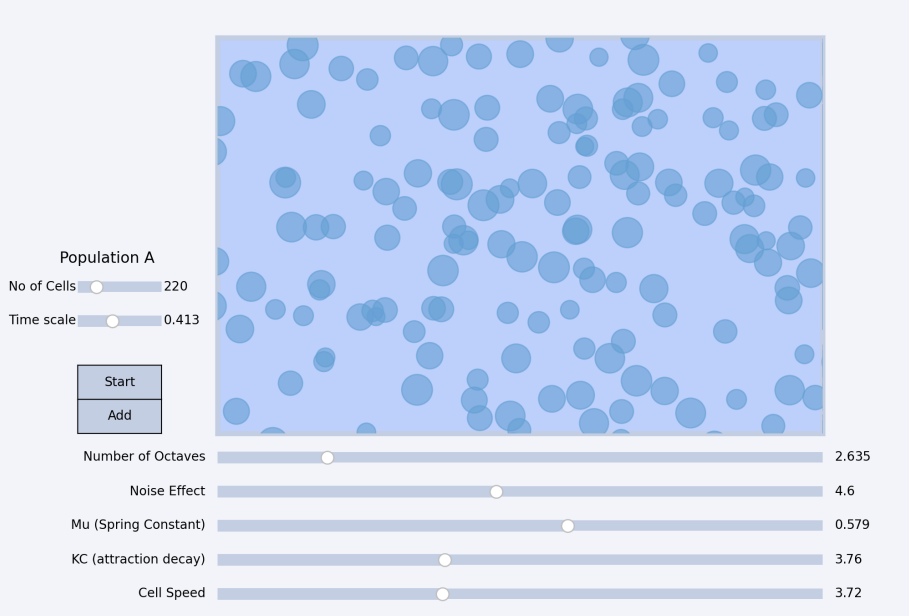
# Generating Spatially Correlated Noise for Computational Biological Simulations




Candidate Number: 1052113

University of Oxford

Trinity Term 2023

BA Computer Science

## Abstract

Biological simulations require spatially correlated, normally distributed random noise to be comparable to real life experiments. This noise must be computable in sensible computational time as simulations can require noise to be generated on every timestep. The issue with generating this kind of noise is that it's tricky to achieve all three. If a noise is heavily spatially distributed, it's generally too slow to generate for large numbers of cells, on every timestep of a simulation. In this project, two random noises: Perlin noise and a smoothed variant of Gaussian noise, are implemented and analysed in detail. The runtime of both with respect to their various parameters is studied, hypothesis testing is employed to explore to what extent different versions of these noises are normally distributed, and extensive work is done to figure out a way of extracting a length scale from a given noise. The aim is to use those techniques to be able to find out which type of noise should be used for a desired spatial behaviour. The behaviours of both noises depend heavily on the parameter values given, but the broad conclusions are that Perlin noise has an asymptotically faster runtime, but Smooth Gaussian tends to run faster, Perlin noise is more spatially correlated, and Smooth Gaussian more normally distributed. In the conclusions of this project a test simulation of a population of cells is implemented, both noises are added, and desirable results of random, but spatially correlated movement is observed.

# Contents

# 1 Introduction

The aim of this project is to generate random noise that is correlated on a specific length scale. Spatially correlated noise [1] is a type of noise which is present in many biological systems and simulations. It refers to noise which when applied to a spatial context, has a high probability of having a similar value over a small length scale, and is randomly distributed over a larger length scale. In a biological context, it relates to biological processes [2] or environmental conditions which aren't independent of each other but are influenced by nearby regions or neighbouring cells.

A specific example is, if you were modelling cells by multiple points on a cell membrane, and your random noise was simulating the addition of chemicals which agitated the movement of the cells in some way. If two nodes represent nearby points on the same cell membrane, it's clear biologically that any random noise acting on both nodes should be correlated, but on two different separated cells, the likelihood is that the noise should affect the nodes randomly.

## 1.1 Motivation

This work builds on work undertaken by my supervisor [3] which uses Gaussian Random Fields. This method yields noise that is independent over long distances but correlated over short distances. However, the runtime of this algorithm is cubic order and not feasible other than for toy simulations. This method was improved using various approximations, but the resulting algorithm still has quadratic order and is infeasible other than for small simulations. Generating spatially correlated noise quickly enough to be applicable to larger simulations will require a new approach.

## 1.2 Challenges

Creating fast, normally distributed, and spatially correlated noise creates many challenges including controlling the trade-off between spatial correlation and normal distribution. These two properties are related, and ideally our noise will exhibit both behaviours, but it's already clear to see that increasing the average spatial correlation to a length scale comparable to the size of your domain would affect the underlying distribution. The second challenge is how to measure the spatial correlation of the output noise; the noises are random, and the output will be different each run so there isn't a clear empirical method for analysis. Ultimately, the noise needs to be able to run in feasible computational time, or it becomes practically useless.

## 1.3 Requirements

The goal of this project is to create methods of implementing random noise which can be applied to computational biological frameworks. The requirements for this noise are:

1. To be useful in complex simulations, the noise should be able to run in sensible computational time, it should certainly be asymptotically better than the kind of simulations it will be applied to.

2. To be comparable to biological noise, it should be normally distributed to some extent.

3. Finally, it should be possible to choose a desired spatial correlation profile for the generated noise to have, i.e., a request for noise which is 100% correlated at 4 units, 75% correlated at 12 units and 0% correlated at 25 units should be able to be met.

# 2 Background

## 2.1 Context

The coordinated movement of cells in a population is instrumental in many biological processes including tissue growth during development. Many computational modelling frameworks have been developed to simulate cell populations, a selection of which have been implemented in the Chaste project [4] and applied to several prototypical biology problems by Osborne and colleagues [5].

An important implementation detail in the application of these methods is that random noise is often required. An example of this is cell sorting by differential adhesion investigated by Osbourne and colleagues: of the five simulation frameworks they consider, two intrinsically include random noise, and the remaining three require it to be added to recapitulate the observed biological behaviour. *[Figure 1]*

The implementation of noise chosen by Osbourne and colleagues is the simple addition of random numbers drawn from a normal distribution to each node in the simulation at each timestep. A node in this context is just a point in space that represents different things in different simulation frameworks. This implementation has the drawback, that the noise is not spatially correlated.

## 2.2 Literature Review

A literature review was undertaken, into existing methods of efficiently generating random noise. Exploration began into methods used in computer graphics [6] for generating natural looking textures including clouds, fire, and terrains [7]. There is a parallel between video game terrain [8] which need to be independent over long distances but correlated over short distances and the noise that would be needed for this project. The benefits of looking down this avenue rather than diving straight into random number generation was that computer graphics, typically used for video games, need to have a minimal amount of stored data and

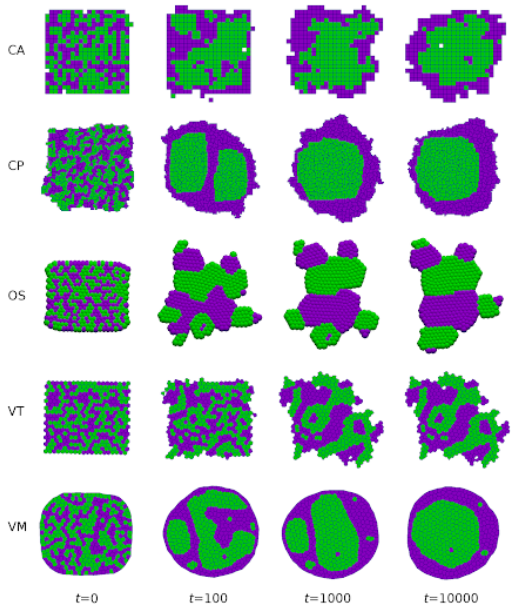



Figure 1: Simulations of cell sorting due to differential adhesion [4].

the graphics need to be computable rapidly as they need to be updated multiple times a second.

### 2.3 Perlin Noise

Perlin and Gaussian were the most frequently mentioned noises. Perlin [9] is a member of a set of noises called Gradient Noise [10]. Gradient Noise algorithms create a lattice of random gradients in the form of vectors of unit length. Then, you interpolate between the lattice points by employing dot products to calculate noise at specific locations. These algorithms are fundamentally different to methods used by Osbourne and colleagues of creating the same lattice but assigning them random scalar values rather than vectors. The scalar method creates distributions known as Value Noise which are much less smooth, especially at each of the lattice points.

## 2.4 Simplex Noise

Perlin Noise uses the Cartesian grid as its base lattice. There are other gradient noises, including Simplex noise [11] which divides the space into simplices, then chooses a random gradient for each lattice point, and interpolates by using the three surrounding points (assuming 2 dimensions) rather than four. Simplex noise has the benefit of being slightly computationally cheaper than Perlin over fixed grids, but on unstructured meshes, as would likely be optimal for biological simulations, the difference is slight, and Perlin noise is more comprehensible to implement and analyse.

## 2.5 Smooth Gaussian Noise

Gaussian noise is the simplest implementation. It involves sampling from the normal distribution at every point you want a noise value for. Clearly this won't meet the spatial correlation requirement laid out previously. However, in this area of research, there is another tool which utilises the normal distribution, called Gaussian Smoothing [12]. This involves calculating a Gaussian kernel, also called the normal probability density distribution. Applying a Gaussian kernel to an image using methods of convolution gives us a blurred effect on the original image. Convolution in this scenario is the method of combining two arrays of different sizes (but same dimensionality) to give an array of the size of the larger array. While this is mostly used for smoothing images, the fundamental result is that you are changing the underlying image slightly to give it spatial correlation, while preserving as much of the input as possible. The outcome is that it was discovered that using a Gaussian kernel overlaid over a normally distributed noise, with the correct parameter values, can create a noise which keeps the benefits of the original noise and is normally distributed, while having the benefits of the new spatial correlation. The immediate drawback is the trade-off between spatial correlation and normal distribution. If the noise is

highly spatially correlated, it will become less normally distributed.

### 2.6 Technology

It was obvious that implementation in Python was the best choice. This is because there are many open-source implementations of both Perlin noise and Image Smoothing [13] and the most readily available support and research is implemented in Python in this area. The most obvious benefit of implementation in Python is the number of useful libraries available, including: matplotlib [14], and perlin-noise [15] (a noise generation graphics library for Perlin noise). For a good proportion of the work, it's useful to be able to display it in an image, which made matplotlib the obvious choice, it also has a very simple API which allowed the focus to be kept on the actual implementation, keeping the programming streamlined.

## 3 Implementations

### 3.1 Perlin Noise Implementation

Conceptually, Perlin noise is simple: a space is divided into a grid of lattice points, and each lattice point is assigned a random unit vector. To get a noise value for any point not on this lattice, run an interpolation function on the four enclosing lattice points. Notation-wise, each square of this grid is referred to as an Octave, and the vital parameter here is the size of these Octaves. This implementation follows from code previously implemented [15], where the whole N-by-N grid is transposed down to a grid of OctSize-by-OctSize squares. On this new grid, the coordinates which are whole numbers are treated as the lattice points, and interpolation is used to sample from any non-whole number coordinates. The two-dimensional implementation is described in pseudocode based on Python here, the full code is available in the Appendix.

```
def perlinNoise(octSize,(x,y)):
  x:=x/OctSize
  y:=y/OctSize
  Gs = [(⌊x⌋,⌊y⌋),(⌈x⌉,⌊y⌋),(⌊x⌋,⌈y⌉),(⌈x⌉,⌈y⌉)]
  Cs = Gs.map(contribution(x,y))
  return sum(Cs)
def contribution((x,y),(gx,gy)):
  #gs: coords of a lattice point surrounding (x,y)
  if (gx,gy) not in C:    #C: global cache
    C(gx,gy) = generateRandomVec(gx,gy)
  vs,vy = C(gx,gy)
  dx,dy = (gx−x),(gy−y)
  w = fade(1−|dx|)*fade(1−|dx|)
  return w*(vx,vy)·(dx,dy)
```

Fade *[L13]*, *[Figure 2]* is a smoothing quintic equation $x^3(6x^2 - 15x + 10)$ which has a desirable behaviour when given $0 \leq x \leq 1$. It's the behaviour of this function that gives Perlin noise the smooth transitions around the lattice points.

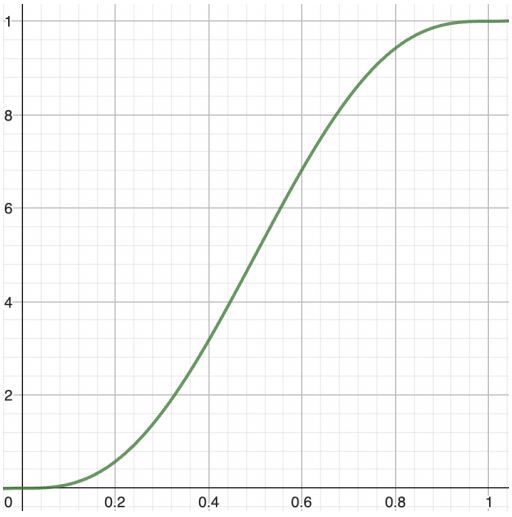



Figure 2: Fade Function: $0 \leq x \leq 1$.

### 3.2 Smooth Gaussian Noise Implementation

It's not immediately obvious how to implement Smooth Gaussian noise point by point as we did with Perlin noise. This implementation creates an N-by-N grid of noise, but later in this project there's another approach shown of generating blocks at a time, by utilising caching.

Initially, generate an N-by-N grid of normally distributed noise, drawing from the standard normal distribution, then create a Gaussian kernel, and combine them.

```
1  def smoothGaussianNoise(k)
2    noise[i,j] = normal(0,1)
3    K = createKernel(k)
4    smoothNoise[x,y] = ∑_{u=1}^{k} ∑_{v=1}^{k} noise[i−u, j−v] ∗ K[u,v]
5    return smoothNoise
6  def createKernel(k)
7    #Create two 2D arrays X,Y of size 2k∗2k such that:
8    X[i,j] = −k+j
9    Y[i,j] = −k+i
10   Z[i,j] = X[i,j]^2 + Y[i,j]^2
11   F[i,j] = exp (−Z[i,j])/k
12   return F
```

CreateKernel creates a grid Z which has very high values at the edges, and moving inwards shrinks the values increasingly slowly until you reach a zero in the middle. Using the negative and the exponential in the calculation of F gives a grid which has a one in the middle, and then decreases very quickly to zero as you move outwards *[Figure 3]*. Once this is overlaid onto the random noise, it adds that "blurred" which makes the noise more spatially correlated as desired.

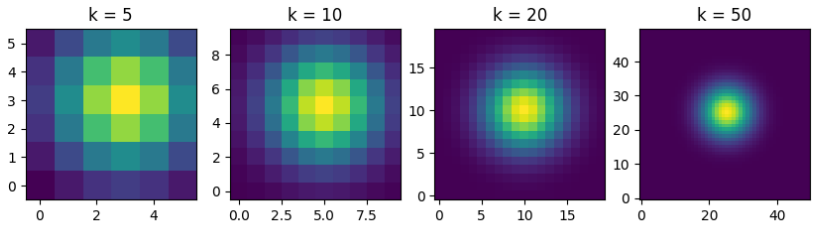



Figure 3: Gaussian Kernels with varying values of k.

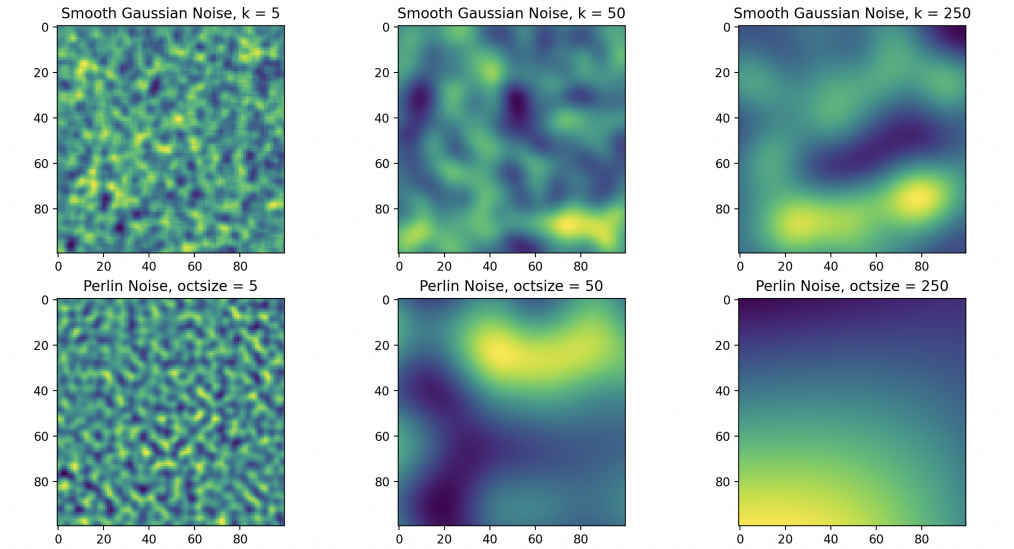



Figure 4: 100 by 100 samples of Perlin and Smooth Gaussian noises, ranging over their respective parameters.

### 3.3 Basic Runtime Analysis

Initial investigations were run to see how the parameters k and N would affect the time taken to generate an N-by-N grid of Smooth Gaussian noise. Analysing the algorithm line by line; generating the initial grid of random noise is $O(N^2)$, creating the kernel is $O(k^2)$, and the convolution step is $O(N^2k^2)$. This convolution step is the slowest and it characterises the overall runtime as $O(N^2k^2)$. As you can see from *[Figure 5]* where k and N were fixed respectively, if you fix one, the other causes quadratic time.

However, if k > N then the runtime ends up being $O(N^4)$, which will be slow for large N.

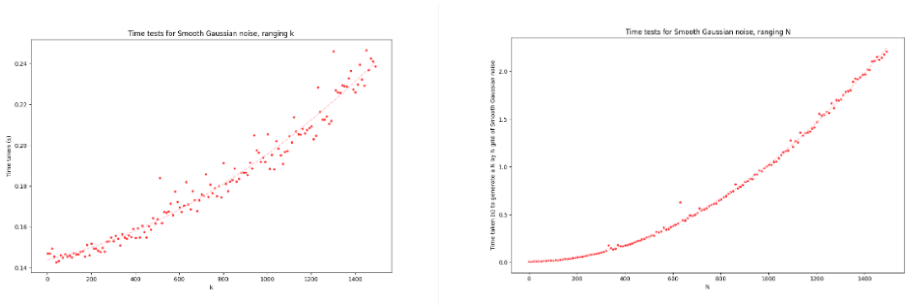



Figure 5: Time tests for Smooth Gaussian noise ranging k [left] and N [right].

Simply by looking at the algorithm for Perlin noise, it's clear that the octave size won't affect the runtime because it's only used once (when all coordinates get divided by Octave Size). The runtime will be $O(N^2)$ to create an N-by-N grid of Perlin noise, as you can see in *[Figure 6]*

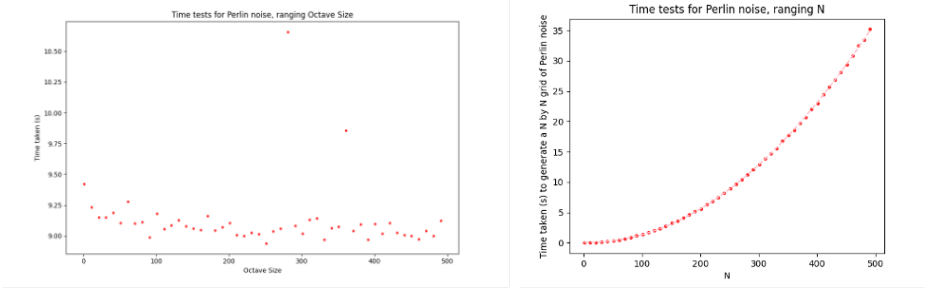



Figure 6: Time tests for Perlin noise ranging Octave Size [left] and N [right].

The expectation would be that Gaussian noise is slower for k>1, however it's faster to run for small N. For example, it takes ∼9 seconds to generate a 250 by 250 grid of Perlin noise (with octave value 100), whereas you can generate a 1500 by 1500 grid of Smooth Gaussian noise (with k = 500) in ∼2 seconds. This is because if you compare coefficients from the quadratic lines of best fit you can see that Gaussian is $O(9.6 * 10^{-7}N^2)$ and Perlin is $O(1.5 * 10^{-4}N^2)$.

# 4 Distribution Analysis

## 4.1 Initial Observations

The second requirement is that our noise is, to some extent, normally distributed. Initially, visual analysis was undertaken by plotting both noises against a best fit normal distribution, implemented using the "norm" module from the scipy.stats [17] library *[Figure 7]*.

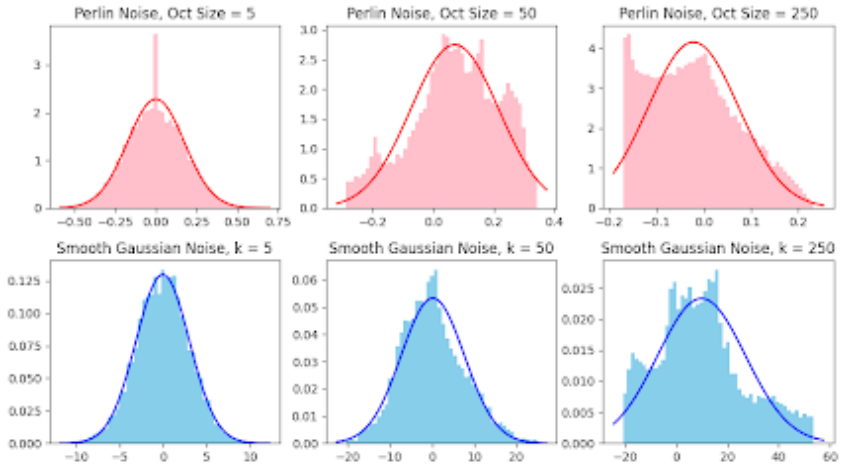



Figure 7: Visualising the extent to which Perlin and Smooth Gaussian noises are Normally Distributed.

For Perlin noise: if you fix the size of the region you're sampling from, when you increase the octave size, the number of lattice points decreases, meaning there are fewer points which are drawn from the normal distribution. It's clear that this means that the noise gets less and less normally distributed. For Smooth Gaussian noise: the initial values, before you lay a kernel onto them, are drawn from a normal distribution, so it makes sense that as you increase k, the "blurring" effect of the Gaussian kernel increases, meaning that you're getting further away from those initial values, so the noise becomes less normally distributed.

## 4.2 Shapiro Wilks Hypothesis Testing

The next analysis was employing the Shapiro Wilks hypothesis test for normality [18]. With varying significance levels (shown in yellow), tests against the null hypothesis that the data was not normally distributed were performed. The test returns a statistic called the p value: if this value is greater than the significance level, then there is insufficient data to reject the hypothesis, so we can assume that the data is normally distributed with that significance. The scatter plots *[Figure 8]* show multiple runs of this hypothesis test with different octave sizes and k values. Clearly the runs end up being slightly random, but they show the trend expected that increasing the parameters generally decreases the normality.

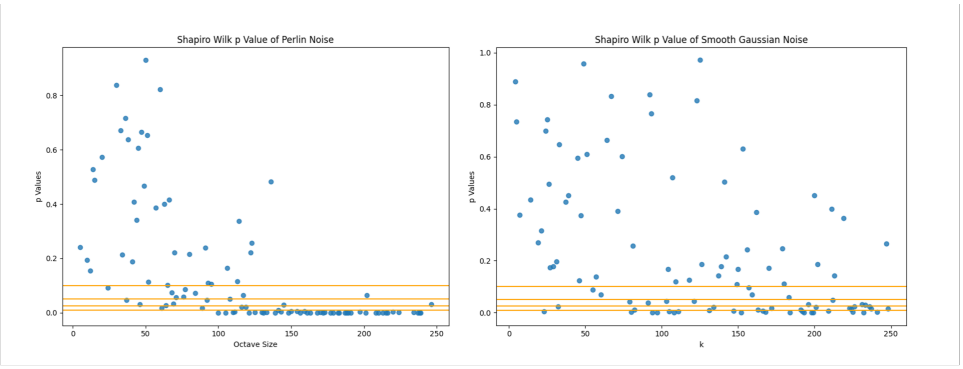



Figure 8: Shapiro Wilk p values of Perlin [left] and Smooth Gaussian [right] noises ranging their respective parameters.

To extract more information, tests were run with a fixed step length (10 units), testing multiple times for each step along the x axis *[Figure 9]*. The percentage of tests which accepted the null hypothesis were plotted. The orange line shows the result of running the same test, the same number of times to a sample of noise drawn from a normal distribution. This line appears at about 92-95% each time, which makes sense, because we're testing at a 5% significance level, especially considering sampling error.

Both noises exhibit similar behaviour; as you increase their respective parameters, the level of normal distribution decreases. Perlin noise 'drops' off much

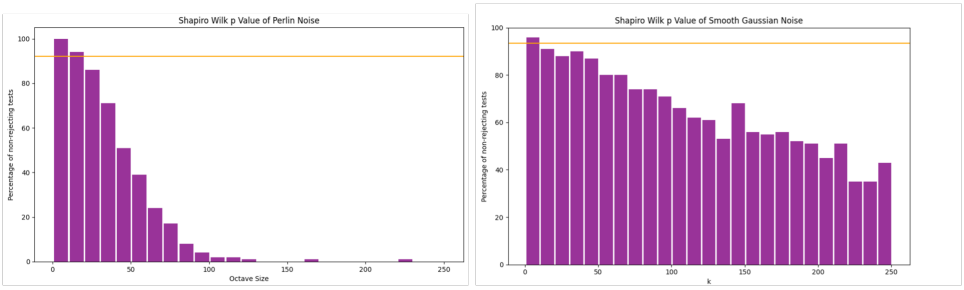



Figure 9: Percentage of noises which passed the Shapiro Wilk hypothesis test for both Perlin [left] and Smooth Gaussian noises [right], [step length = 10, run 100 times per step, 500 samples per test with a 5% significance level].

faster than Smooth Gaussian noise, which makes sense, as for Smooth Gaussian noises, the base graph is always sampled from the normal distribution, even with the additional blurring added by larger and larger kernels. However, for Perlin noise, when you're increasing the octave size, you're quadratically decreasing the number of lattice points (drawn from the normal distribution), so it makes sense that Perlin noise would drop off in a quadratic manner.

# 5 Spatial Correlation Analysis

## 5.1 Overview

It seems intuitively that both noises studied so far will be spatially correlated to some extent. For Smooth Gaussian noise, it seems like the correlation will be related to k, the size of the kernel, but it's not immediately obvious how directly k will affect the outcome, because of the overlapping which occurs when combining the kernel and base noise.

Initial tests were run *[Figure 10]*, to check that the noises did in fact exhibit the desired behaviour. The graphs below were drawn through sampling pairs of points and plotting the distance between them, against the absolute difference in their noise values.

For both noises, as you increase the values of their respective parameters, they become more spatially distributed, Perlin noise more so than Smooth Gaussian.

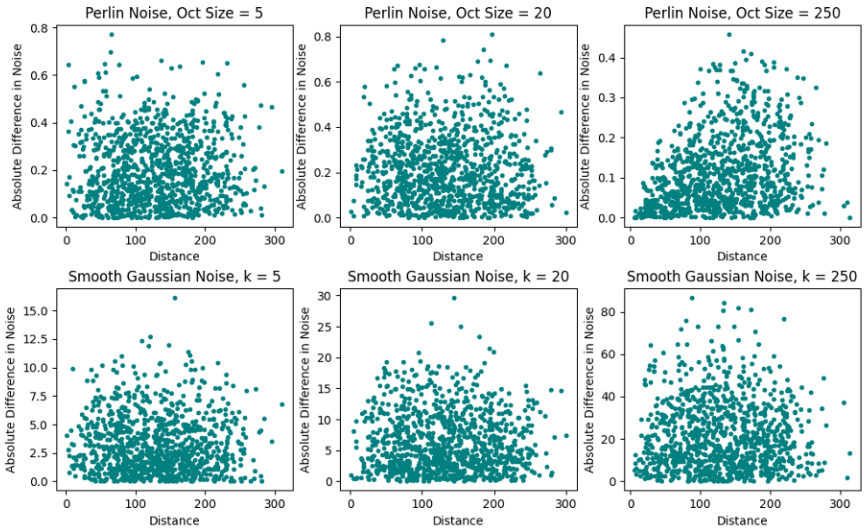



Figure 10: Initial Exploration into Spatial Correlation of Perlin and Smooth Gaussian noises.

## 5.2 Correlation Function

To extract quantitative values out of these qualitative results, correlation functions were investigated. Specifically, iteration through all possible distances between points ($0 \leq \delta \leq \sqrt{2n^2}$), and using the following correlation function:

$$F(\delta) = \frac{\sum_{x,y=0}^{N} P[x,y] \times P([x,y]+\delta)}{\sum_{x,y=0}^{N} P[x,y]^2}.$$

Where P(x,y) is the noise value at point (x,y), and the denominator is included simply to normalise the values down to between 0 and 1.

The results look as expected, examining the top graph *[Figure 11]* of Perlin noise with octave size = 100, when the distance is <5 units, the correlation is ∼1, and as you reach ∼65 units, the correlation hits 0. This makes sense, as examining the image representing this noise, the patches of the same colour (specifically looking at the yellow patch in the bottom left corner) are ∼65 units. However, then the graph dips below zero and returns at ∼175 (the length of the long blue patch in the middle). It's initially unclear which of these values we should take to be the point at which the graph is no longer correlated.

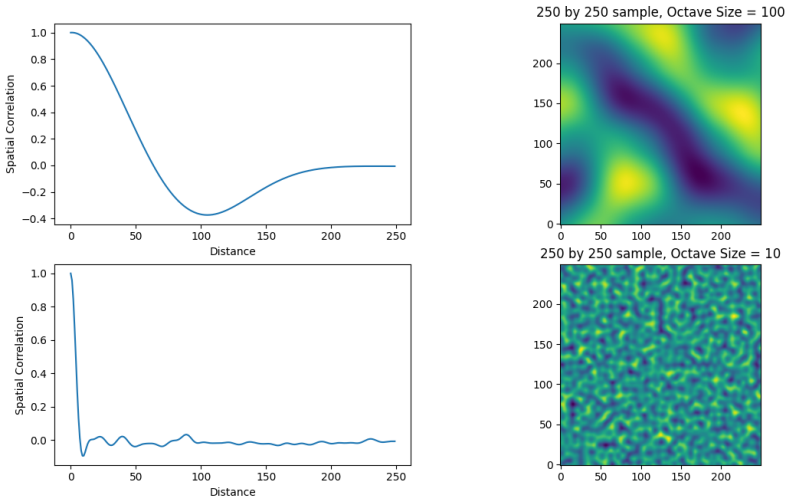



Figure 11: Spatial Correlation Function applied to a 250 by 250 sample of Perlin noise.

The lower graph *[Figure 11]* with octave size 10 is harder to analyse in the same way, either we take distances 5 or 12 when the graph first dips below 0 and returns to 0. But the argument could also be made that the graph hasn't "settled down" to zero until distance = 100. However, the graphs still safely show us at which distance the noise is x% spatially correlated, for $x > 5$.

After the data has finished its initial descent, it oscillates at ∼0, demonstrating that this is the "natural level" of spatial correlation which the noise sits at, i.e., the amount of correlation which is caused by just generating random noise. If you run the same function on uniform random noise *[Figure 12]*, the correlation factor is 0 for all values apart from distance = 0.

The same function can be run with Smooth Gaussian noise, with similar results. *[Figure 13]*.

## 5.3 R Values

Next, work was done investigating getting a few different values out of these graphs and plotting these values against different sizes of octaves and different k values respectively. Each value was the distance when the spatial correlation

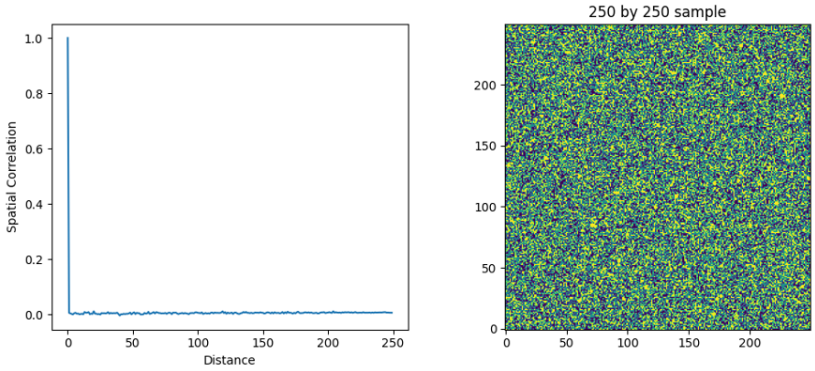



Figure 12: Spatial Correlation Function applied to a 250 by 250 sample of Random noise.

dropped below X% and will be referred to as rX values from now on. *[Figure 14]* shows an example on a 250 by 250 sample of Smooth Gaussian noise with k = 100. Here the r0 value is taken to be when the noise initially drops below 0, which seems feasible for this specific graph, but it's not obvious that it's the optimal approach in all situations.

Another consideration was to take areas under the graph, using numeric methods, instead of simply the distance values. After experimentation, this ended up giving similar results, and didn't provide much extra precision.

These values give information about the profile of the spatial correlation of a specific noise. For example, it can show at which distance this specific sample of noise drops below a 50% correlation level, and at what point it first hits 0. But if we wanted a noise which was 75% spatially correlated at ∼10 units, and 50% correlated at ∼25, it doesn't tell us what kind of noise or parameter to choose. The next step is to plot these values against octave size and k value for Perlin and Gaussian noises respectively, so that given a specification of what kind of spatial correlation we need in our noise, an informed choice can be made about what type of noise and which parameter would be best.

Plotting different r values against octave size for Perlin noise *[Figure 15]* gives more information. Studying the pink line specifically, it shows that if a noise was desired which was 50% correlated at ∼25, you should choose an octave

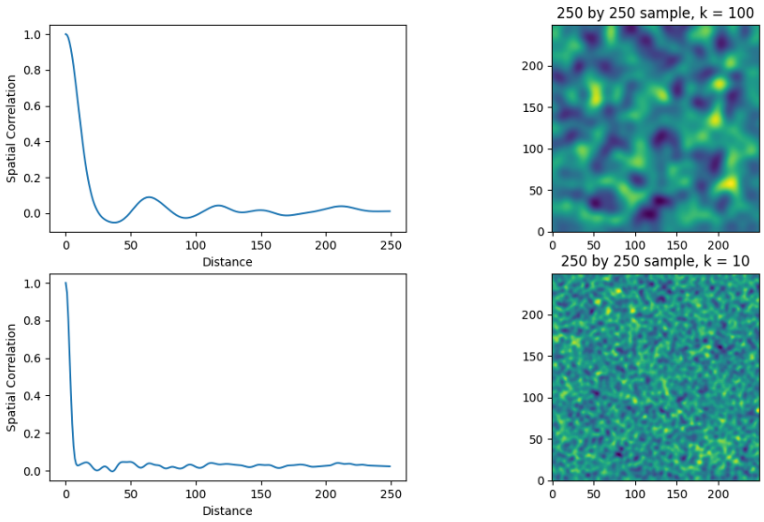



Figure 13: Spatial Correlation Function applied to a 250 by 250 sample of Smooth Gaussian noise.

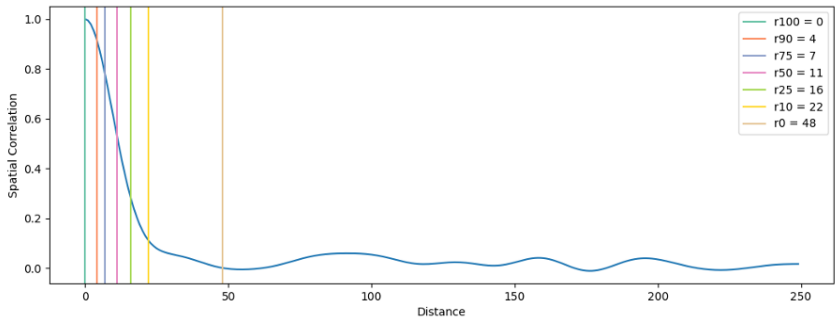



Figure 14: R Values of Smooth Gaussian Noise (k = 100).

size of ∼65. You would then also know that your noise would be 25% correlated at ∼34 units and 90% correlated at ∼7 units.

The same can be done with Smooth Gaussian Noise *[Figure 16]*, for a similar graph.

This comparison shows how Perlin noise is more spatially correlated for k = octave size, almost 5 times so. It seems that the distribution of where it's spatially correlated is similar for both noises, i.e., the ratio of the distances between lines is similar.

The issue with these graphs is how irregular the top lines are: specifically, the

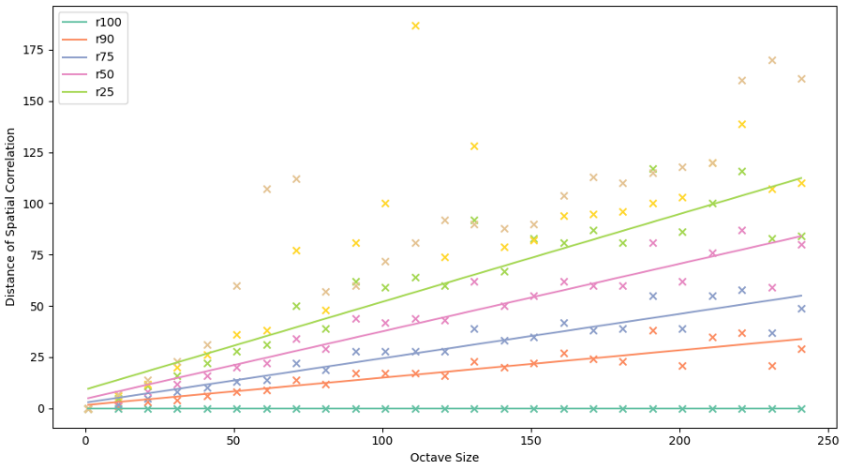



Figure 15: Perlin noise: Octave Size vs Different R Values [250 by 250 sample].

r0 line in both noises. They're so irregular, it's not possible to draw a sensible line of best fit for them. This is because of the problem mentioned earlier about how it's just not clear when to measure this r0 value. Should it be when the data first crosses the zero-correlation line, or when it finally levels out? It makes sense to be the latter, but it's not obvious how to define "fully levelled out" in this situation. The benefit of the former method is that it underestimates the spatial correlation which is less dangerous than overestimating.

# 6 Unstructured Grids

Focus has been kept on methods of generating noise in N-by-N grids. Smooth Gaussian noise was significantly faster than Perlin noise at this, but there may be a problem when generating random noise on an unstructured grid [19], which will be needed in most of the biological simulations discussed already. It's inefficient to generate an entire N-by-N grid, just to get the noise values of n cells if $n < N^2$, which it may be if the cell population is sparse. A second implementation was explored, which allows more pipelining by caching in a similar way as the initial Perlin noise algorithm.

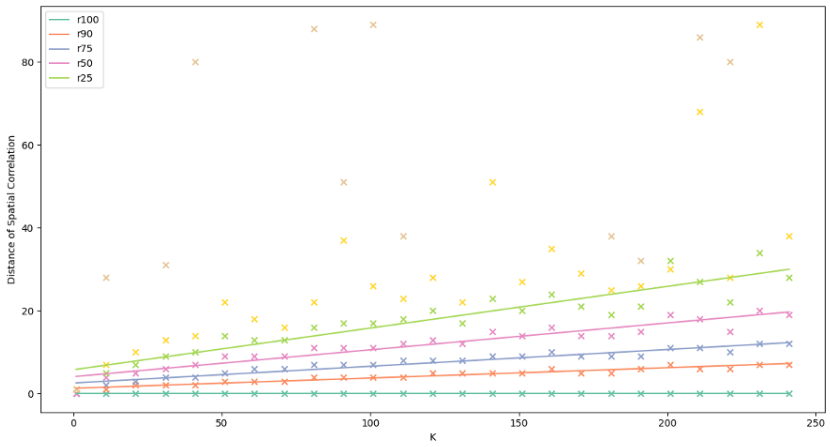



Figure 16: Smooth Gaussian noise: K vs Different R Values
[250 by 250 sample].

The idea of this algorithm is that to generate a single point of noise, only the $k^2$ points which surround it are needed. We can then cache these points to be used later.

```
smoothGaussianNoiseBlocks(k,K,(x,y))
  if K == []
    K = createKernel(k)   #this will only be run on the
  first point
    v = 0
  Foreach (u,v) pair: 0<=u,v<k
    pt = (x-u,y-v)
    if pt not in C    #C: global cache
      C(pt) = normal(0,1)
    v += C(pt)*K[u,v]
  return v
```

The runtime of this algorithm is $O(k^2)$ for each point. If k is small, this isn't too bad, but if a larger k is chosen, then it can end up being much slower than Perlin Noise.

# 7 Application to a Simulation

The final stage of this project is applying these random noises to a test simulation, to see if it gives the desired results. The implementation used was an Object-Oriented approach [22], each cell is an object which has attributes: position vector, id, and radius. The simulation is controlled step-by-step. During each step, the cells are given their new position vector based on how close they are to the cells around them. The interaction between cells is given by the following equation:

For two cells i and j, interact(i,j) is as follows:

```
def interact(i,j):
  #cells are 'far away', there should be no interaction
   if |r_ij| > maxRadius:
     return 0

   #cells are 'too close together', they should repel
  each other
   if |r_ij| < s_ij:
     return r̂_ij μ s_ij ln(1+(|r_ij|−s_ij)/s_ij)

   #cells are 'close but not too close', they should
  attract gently
   else:
     return r̂_ij μ(|r_ij| − s_ij) exp(−kc (|r_ij|−s_ij)/s_ij)

#Where:
#r_ij: distance vector between cell i and j
#maxRadius: maximum radius at which the cells interact
```

```
17 #s_ij: sum of the radii of cell i and j, ie their natural
    separation
18 #and â denotes the unit vector of a
```

This is a standard implementation of cell-cell behaviour called the overlapping spheres model [23]. If two cells are far apart, they don't have any effect on each other's motion and speed. If they're very close together, they repel like springs, and if they're reasonably close together, but not too close, they attract gently. The two parameters are $\mu$ and kc also called the spring, and attraction decay constants respectively. A high $\mu$ value, ($> 1$) results in cells bouncing off each other more dramatically. A low $\mu$ value allows cells to get very close together to each other before springing apart. A high kc value ($> 1$) causes cells to be highly attractive, and a low kc means cells must be very close together before being pulled towards one another.

Once the new displacement vectors were calculated, Perlin noise was generated and added to each cell. This re-enforces the point made earlier about how important it is that the noise can be calculated in sensible computational time. We must re-calculate these noise values at each time step, for each cell, so it's infeasible to run simulations with any useful number of data points unless this noise is calculable in constant time per point. The same application can be implemented with Smooth Gaussian Noise instead.

The left images *[Figure 17]* show 2 different runs of the simulation after the cells had been created and allowed to move about and interact with each other for $\sim$10 seconds. The right images show the cells another 10 seconds after adding random (top) and Perlin (bottom) noises respectively. The random noise causes the cells to move about independently of each other, but with the spatially correlated noise, the cells close together move in similar patterns.

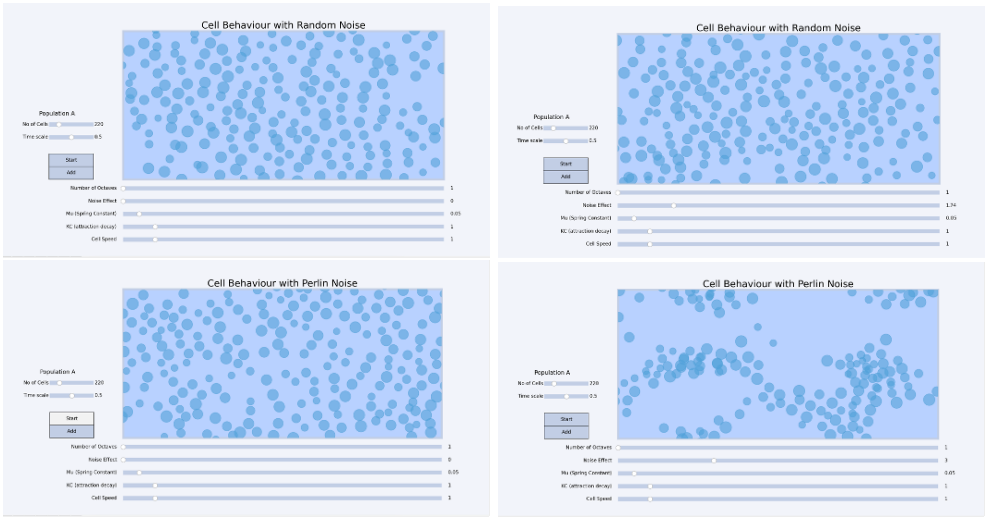



Figure 17: Simulations run before [left] and after [right] the addition of Random [upper] and Perlin [lower] noises.

# 8 Conclusions

## 8.1 Final Comparison

Theoretically Perlin noise is asymptotically faster than Smooth Gaussian, but Gaussian is significantly faster, especially for small k, for both N-by-N grids and unstructured grids. Both run in sensible computational time, are feasible for small biological simulations, and are asymptotically better than the kind of simulations they will be applied to, but Gaussian may be considered "better" in this aspect. Both noises are normally distributed to some extent, and Gaussian noise remains normally distributed for longer than Perlin as you increase both parameters. Both noises display different levels of spatial correlation at different parameter values, and with analysis, it's possible to choose a parameter which can give you the desired spatial correlation profile.

As parameters increase: Gaussian noise gets slower to run, and both noises become less normally distributed and more spatially distributed.

## 8.2 Summary

This project has achieved its aim of discussing and analysing different kinds of noise, two different normally distributed, spatially correlated random noises have been given which can both be run in varying degrees of sensible computational time. Comparisons were made between both noises, and it was found that both could be useful for differing applications.

Methods have been explored for measuring the spatial correlation of a given noise, which could be applied to other noises in future research.

## 8.3 Reflections

Upon starting this project, I thought the hardest element would be making the noise run fast enough to be useful for non-toy simulations. But it became evident that the hardest part was the work on spatial correlation. I thought it would be simple to get a value of spatial correlation out of a specific noise and to find a relationship between this and the input parameter. However, it was not clear how to get this value out, and a lot of the routes I went down, not discussed here, didn't lead me anywhere. The implementation to the simulation, was particularly successful however.

I ended up writing significantly more code than I expected. There are 20+ pages of successful code appended to this work, but at least 10 times as many were written in the process. I wrote many files of code for testing and analysing the presented work.

This project gave me a chance to solidify many of the courses I've taken during my degree including PoPL, Imperative Programming, ADS, Computational Complexity and even Concurrent Programming [20] (discussed below).

## 8.4 Concurrency

Perlin noise has no shared 'read' variables, and so is intrinsically parallelisable, and could be implemented concurrently. Smooth Gaussian noise has the problem that the base grid values are shared, meaning that neither of the algorithms shown so far can be directly implemented 100% concurrently. The initial grid algorithm can be partly parallelised, as the base grid and kernel could be calculated by the first thread, and then the rest of the algorithm is easy to parallelise as the accesses to the kernel and shared variables are read only. The block-based algorithm could be implemented by having a dictionary of which points of the base grid had been calculated already. If the point was already in the dictionary, then you can safely access it, as all threads trying to access it will be read only. The only issue will be if two threads need the same point and it's not in the dictionary, and they both calculate it themselves and write back at different times. This data inconsistency could cause issues, but it's an unlikely situation especially if the data is sparse, and the problems it would cause would likely not be noticeable. A solution would be to split the coordinate space up into contiguous blocks of size 2k-by-2k, and to create a lock [21] for each block. If a thread wanted to create a base point, it would have to obtain that lock first, and it would only be released after the base point was written back to the dictionary. To get around the problem of potentially having an infinite coordinate space and finite locks, you could have each lock responsible for multiple blocks. I.e. lock $(i, j)$ is responsible for any blocks with top left-hand coordinates of the form $(\alpha i, \alpha \mathrm{j})$ for some fixed (large) alpha.

## 8.5 Future Work

The work done in this project on spatial analysis of random noise is a good starting point, but more could be done to come up with additional results. Specifically, I would like to do further exploration into measuring our r0 value.

I would start by investigating sampling to see if I could make the analytical methods faster while maintaining accuracy, to see if there are any relationships which were missed in this research due to runtimes being too slow (10-20 minutes for some). It would be beneficial to investigate ways to measure when the graph "levels off" at that zero spatial correlation, and interesting to plot that value against Octave Size and K to see if the relationship becomes more obvious.

I would also like to explore ways to get the simulation to run faster: currently the deficiency is that the algorithm checks all $N^2$ pairs each timestep. Initial pathways to look down include KD-Trees for quick nearest neighbour lookup [24].

Additionally, more analysis could be done on the simulation itself, to see how the spatial correlation of the cells compares to the predicted length scales evaluated in the static theoretical noise.

# 9 Bibliography

## 9.1 References

# References

[1] Cepel, R. (2007). Statisical Analysis and Computer Generation of Spatially Correlated Acoustic Noise. pp.31–62. DOI: 10.32469/10355/5054.

[2] Simpson, M.L. (2009). Noise in Biological Circuits. WIREs Nanomedicine and Nanobiotechnology, 1(2), pp.214–255. DOI: 10.1002/wnan.22.

[3] Cooper, F.R. (2018). A mathematical and computational framework for modelling epithelial cell morphodynamics. pp.85–106.

[4] Pitt-Francis, J., Pathmanathan, P., Bernabeu, M.O. and Bordas, R. (2009). Chaste: A test-driven approach to software development for biological mod-

elling. Computer Physics Communications, 180(12), pp.2452–2471. DOI: 10.1016/j.cpc.2009.07.019.

[5] Osborne, J.M., Fletcher, A.G., Pitt-Francis, J.M., Maini, P.K. and Gavaghan, D.J. (2017). Comparing individual-based approaches to modelling the self-organization of multicellular tissues. PLOS Computational Biology. DOI: 10.1371/journal.pcbi.1005387.

[6] Nishita, T. and Dobashi, Y. (2001). Modelling and rendering of various natural phenomena consisting of particles. In: Proceedings. Computer Graphics International 2001. [online] pp.149–156.DOI: 10.1109/CGI.2001.934669.

[7] Archer, T., 2011, April. Procedurally generating terrain. In 44th annual midwest instruction and computing symposium, Duluth (pp. 378-393).

[8] Frade, M., de Vega, F.F. and Cotta, C. (2012). Automatic evolution of programs for procedural generation of terrains for video games. In: Soft Comput. DOI: 10.1007/s00500-012-0863-z.

[9] Green, S., 2005. Implementing improved perlin noise. GPU Gems, 2, pp.409-416.

[10] Parberry, I., 2014. Designer worlds: Procedural generation of infinite terrain from real-world elevation data. Journal of Computer Graphics Techniques, 3(1), DOI: 10.1109/estream.2019.8732171.

[11] Shuai, W.A.N.G. and Yu-bo, J.I., 2009. Method of Rendering Clouds Based on Simplex Noise. Journal of Liaoning University of Petroleum & Chemical Technology, 29(3), p.58.

[12] Chung, M.K. (2020). Gaussian kernel smoothing. DOI:10.48550/arXiv.2007.09539.

[13] Pei-Yung Hsiao, Shin-Shian Chou and Feng-Cheng Huang, "Generic 2-D gaussian smoothing filter for noisy image processing," TENCON 2007

- 2007 IEEE Region 10 Conference, Taipei, Taiwan, 2007, pp. 1-4, DOI: 10.1109/TENCON.2007.4428941.Perlin_noise.

[14] Hunter, J.D. (2007). Matplotlib: A 2D graphics environment. Computing in Science & Engineering, 9(3), pp.90–95. DOI:org/10.1109/MCSE.2007.55.

[15] Ildar, S. (2022). Python implementation for Perlin Noise with unlimited coordinates space. https://pypi.org/project/perlin-noise/.

[16] J. Babaud, A. P. Witkin, M. Baudin and R. O. Duda, "Uniqueness of the Gaussian Kernel for Scale-Space Filtering," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-8, no. 1, pp. 26-33, Jan. 1986, DOI: 10.1109/TPAMI.1986.4767749.

[17] Virtanen, P., Gommers, R., Oliphant, T.E. and Haberland, M. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods, 17(3), pp.261–272. DOI: 10.1038/s41592-019-0686-2.

[18] Keya Rani Das, A. H. M. Rahmatullah Imon. A Brief Review of Tests for Normality. American Journal of Theoretical and Applied Statistics. Vol. 5, No. 1, 2016, pp. 5-12. DOI: 10.11648/j.ajtas.20160501.12.

[19] Sales-Mayor, F. and Wyatt, R.E. (n.d.). A two-stage filter for smoothing multivariate noisy data on unstructured grids. Computers & Mathematics with Applications, 47(6-7), pp.877–891. DOI:10.1016/S0898-1221(04)90072-7.

[20] Lowe, G. (2022). Concurrent Programming. [online] Lecture Notes, University of Oxford. Available at: https://www.cs.ox.ac.uk/teaching/materials22-23/concurrentprogramming/.

[21] Andrews, G.R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley.

[22] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). Design Patterns: elements of reusable object-oriented software.

[23] Pathmanathan, P., Cooper, J., Fletcher, A., Mirams, G. and Pitt-Francis, J. (2009). A computational study of discrete mechanical tissue models. Physical Biology, 6(3). DOI 10.1088/1478-3975/6/3/036001.

[24] Virtanen, P., Gommers, R., Oliphant, T.E. and Haberland, M. (2020). SciPy. [online] scipy.spatial.KDTree. Available at: https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.KDTree.html.

## 9.2 Appendices

### 9.2.1 Perlin Implementation

```
import math
import random
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats
from scipy.stats import norm

#datatype having x and y values
class Coord:
    def __init__(self, x:float, y:float):
        self.x = x
        self.y = y

    def map(self, f):
        self.x = f(self.x)
        self.y = f(self.y)
```

```
17
18      def get_bounds(self):
19          lowerX_bound = math.floor(self.x)
20          upperX_bound = math.floor(self.x + 1)
21          lowerY_bound = math.floor(self.y)
22          upperY_bound = math.floor(self.y + 1)
23          return [lowerX_bound, upperX_bound, lowerY_bound,
    upperY_bound]
24
25      def print(self):
26          print("(", end = '')
27          print(round(self.x,2), end = '')
28          print(",", end = '')
29          print(round(self.y,2), end = '')
30          print(") ", end = '')
31
32      def value(self):
33          return self.x,self.y
34
35      def getweight(self) -> float:
36          return fade(1-abs(self.x))*fade(1-abs(self.y))
37
38  class Vec:
39      def __init__(self, x:float, y:float):
40          self.x = x
41          self.y = y
42
43      def print(self):
```

```
44          print("(", end = '')
45          print(round(self.x,2), end = '')
46          print(",", end = '')
47          print(round(self.y,2), end = '')
48          print(") ", end = '')
49
50      def value(self):
51          return self.x,self.y
52  def halfzip(list:list[int]) -> list[Coord]:
53      a,b,c,d = list
54      return [Coord(a,c), Coord(a,d), Coord(b,c), Coord(b,d
        )]
55  #returns c1-c2
56  def subtract(c1:Coord, v2:Vec) -> Vec:
57      x1,y1 = c1.value()
58      x2,y2 = v2.value()
59      return Coord(x1-x2, y1-y2)
60  def dot(v1:Vec, v2:Vec) -> float:
61      x1,y1 = v1.value()
62      x2,y2 = v2.value()
63      return x1*x2 + y1*y2
64  #smoothes [0,1] value
65  def fade(given_value: float) -> float:
66      if given_value < -0.1 or given_value > 1.1:
67          raise ValueError('expected to have value in
        [-0.1, 1.1]')
68      return 6 * math.pow(given_value, 5) - 15 * math.pow(
        given_value, 4) + 10 * math.pow(given_value, 3)
```

```
69 #assume seed = 1, octaves = 1
70
71 class MyPerlinNoise:
72     def __init__(self, octSize: float = 1, seed: int = 1)
   :
73         self.octSize: int = octSize
74         self.seed: int = random.randint(1,100)
75         self.cache: dict[Coord,Vec] = {}
76
77     #returns noise values for a pair of coordinates
78     def __call__(self, xy) -> float:
79         x,y = xy
80         coords = Coord(x,y)
81         #working coordinates we're trying to find noise
   for: Coord
82         coords.map(lambda x: x/self.octSize)
83
84         #bounding box around working coordinates: List[
   Int]
85         bounds = coords.get_bounds()
86         #grid points surrounding working coordinates:
   List[Coord]
87         gridpts = halfzip(bounds)
88         #vector values corresponding to those gridpoints:
    List[Vec], vector distances between each grid point
   and the working coordinates: List[Vec]
89         gridvecs = []
90         dists = []
```

```
91          for gridpt in gridpts:
92              gridvecs.append(self.getvector(gridpt))
93              dists.append(subtract(coords,gridpt))
94          #vector weights between each grid point and the
    working coordinates: List[Float], ie how much this
    should count for, points further away get a smaller
    weight and v.v.
95          weights = []
96          for dist in dists:
97              weights.append(dist.getweight())
98          #weighted contributions of each grid point: List[
    Float]
99          contributions = []
100         for i in range(0,4):
101             contributions.append(weights[i] * dot(
    gridvecs[i], dists[i]))
102         return sum(contributions)
103     def getvector(self, pt:Coord) -> Vec:
104         if pt not in self.cache:
105             self.cache[pt] = randVec(self.seed + hasher(
    pt))
106         return self.cache[pt]
107 def hasher(coors: Coord) -> int:
108     x,y = coors.value()
109     return max(1,int(abs(dot(coors, Coord(10**x, 10**y))
    + 1,))     )
110 #returns normalised vector, always same for a given seed
111 def randVec(seed:int) -> Vec:
```

```
112     st = random.getstate()
113     random.seed(seed)
114     vec = Vec(random.uniform(-1,1), random.uniform(-1,1))
115     random.setstate(st)
116     #this line has to be included so that it always
        creates same random vector for same input
117     return vec
118 #plots a noise against the normal distribution on a given
        set of axes
119
120 def perlin_noise(octSize, N):
121     A = np.zeros([N,N])
122     pn = MyPerlinNoise(octSize)
123     for i in range(N):
124         for j in range(N):
125             A[i,j] = pn((i,j))
126     return A
```

### 9.2.2 Smooth Gaussian Implementation

```
1 import numpy as np
2 import scipy.signal
3 import matplotlib.pyplot as plt
4 import math
5 import random
6 from MyPerlin2 import *
7 from scipy.stats import norm, entropy
8 from scipy import stats
9
```

```
10 #-------------------------------------------------------------
11 #PARAMETERS FOR EDITING
12 mean, std_dev = 0, 1
13 #k   = 1
14 #n = 100
15 #-------------------------------------------------------------
16
17 class Gaussian:
18     def __call__(self,m,s) -> float:
19         s = self.normal_dist(m,s)
20         return s
21
22     def normal_dist(self, m,s):
23         #s = random.uniform(-1,1)
24         s = np.random.normal(m,s)
25         return s
26
27 def create_filter(k):
28     scale = math.ceil(k/2)
29     x = np.arange(-scale, scale)
30     y = np.arange(-scale, scale)
31     X,Y = np.meshgrid(x,y)
32     dist = np.sqrt(X**2 + Y**2)
33     filter = np.exp(-dist**2/(2*scale))
34     return filter
35
36 def smooth(noise, k):
37     filter_kernel = create_filter(k)
```

```
38      return np.array(scipy.signal.fftconvolve(noise,
    filter_kernel, mode = 'same'))
39
40  def smooth_gaussian_noise(k,n):
41      noise = Gaussian()
42      A = np.zeros([n,n])
43      for x in range(n):
44          for y in range(n):
45              A[x,y] = noise(mean, std_dev)
46
47      A = smooth(A, k)
48
49      return A
```

### 9.2.3 Runtime Analysis

```
1  import time
2  from MyPerlin2 import *
3  from SmoothGaussian import *
4  import matplotlib.pyplot as plt
5
6  def p_noise(octSize,n):
7      noise = MyPerlinNoise(octSize)
8      A = np.zeros([n,n])
9      for x in range(n):
10         for y in range(n):
11             A[x,y] = noise((x,y))
12     return A
13
```

```
14 def g_noise(k,n):
15     noise = Gaussian()
16     A = np.zeros([n,n])
17     for x in range(n):
18         for y in range(n):
19             A[x,y] = noise(mean, std_dev)
20     A = smooth(A, k)
21     return A
22 
23 #octave size and k [0-N] on x axis and time on y
24 def time_test1(stepSize, N, maxparam, noisetype):
25     if noisetype == "P":
26         name = "Perlin"
27         parameterName = "Octave Size"
28         get_noise = p_noise
29     else:
30         name = "Smooth Gaussian"
31         parameterName = "k"
32         get_noise = g_noise
33     xs = np.arange(1,maxparam,stepSize)
34 
35     times = []
36     for x in xs:
37         print(x)
38 
39         start = time.time()
40         get_noise(x,N)
41         end = time.time()
```

```
42          times.append(end-start)
43
44      a,b,c = np.polyfit(xs,times,2)
45      print(a,b,c)
46      plt.scatter(xs,times, color = "red", alpha = 0.75,
    marker = '.')
47      #plt.plot(xs, a*xs*xs+b*xs+c, linestyle = '--', color
    = "pink")
48      #plt.legend(loc = "upper left")
49      plt.title("Time tests for {} noise, ranging {}".
    format(name,parameterName))
50      plt.ylabel("Time taken (s)")
51      plt.xlabel(parameterName)
52
53      plt.show()
54
55 #n on x axis [0-500] and time on y
56 def time_test2(stepSize, defParam, maxN, noisetype ):
57      if noisetype == "P":
58          name = "Perlin"
59          parameterName = "Octave Size"
60          get_noise = p_noise
61      else:
62          name = "Smooth Gaussian"
63          parameterName = "k"
64          get_noise = g_noise
65      ns = np.arange(1,maxN,stepSize)
66      times = []
```

```
67      for n in ns:
68          print(n)
69          start = time.time()
70          get_noise(defParam,n)
71          end = time.time()
72          times.append(end-start)
73
74      a,b,c = np.polyfit(ns,times,2)
75      print(a,b,c)
76      plt.plot(ns,a*ns*ns+b*ns+c, linestyle = '--', color =
        "pink")
77
78      plt.scatter(ns,times, color = 'red', alpha = 0.75,
       marker = ".")
79
80      #plt.legend(loc = "upper left")
81      plt.title("Time tests for {} noise, ranging N".format
       (name))
82      plt.ylabel("Time taken (s) to generate a N-by-N grid
       of {} noise".format(name))
83      plt.xlabel("N")
84
85      plt.show()
86
87  stepSize = 10
88  defN = 500
89  maxparam = 1500
90
```

```
defParam = 500
maxN = 1500

time_test1(stepSize, defN, maxparam, "G")
time_test2(stepSize,defParam,maxN,"G")
```

### 9.2.4 Distribution Analysis

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm
from MyPerlin2 import *
from SmoothGaussian import *

N = 100
f, axes = plt.subplots(2,3)
params = [5,50,250]
mus = np.zeros([2,len(params)])
stds = np.zeros([2,len(params)])

plt.suptitle("Visualising the Extent to which Perlin and
   Smooth Gaussian noises are Normally Distributed")
bins = 50

for i in range(len(params)):
    print(i)
    p_noise = perlin_noise(params[i], N).flatten()
    g_noise = smooth_gaussian_noise(params[i],N).flatten
   ()
```

```
20
21      axes[0,i].set_title("Perlin Noise, Oct Size = {}".
    format(params[i]))
22
23      axes[1,i].set_title("Smooth Gaussian Noise, k = {}".
    format(params[i]))
24
25      #fit
26      pmu, pstd = norm.fit(p_noise)
27      gmu, gstd = norm.fit(g_noise)
28
29      #plot histograms
30      axes[0,i].hist(p_noise, bins = bins, density = True,
    color = "pink")
31      axes[1,i].hist(g_noise, bins = bins, density = True,
    color = "skyblue")
32
33      pxmin,pxmax = axes[0,i].get_xlim()
34      gxmin,gxmax = axes[1,i].get_xlim()
35
36      px = np.linspace(pxmin,pxmax,100)
37      gx = np.linspace(gxmin,gxmax,100)
38
39      pp = norm.pdf(px,pmu, pstd)
40      gp = norm.pdf(gx,gmu,gstd)
41
42      axes[0,i].plot(px,pp,color = "red")
43      axes[1,i].plot(gx,gp, color = "blue")
```

```
44
45 plt.show()
```

### 9.2.5 Spatial Correlation Analysis

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from SmoothGaussian import *
4  from MyPerlin import *
5  import random
6
7  n = 100
8  rs = [100,90,75,50,25,10,0]
9  colors = ['#66c2a5','#fc8d62','#8d9fca','#e789c2','#
       a5d853','#ffd82f','#e5c494']
10
11 def FastCorrelationFactor(noise):
12     N = len(noise)
13     def f(d):
14         total,sumSq = 0,0
15         for x in range(N):
16             for y in range(N):
17                 xD = min(N-1,x+d)
18                 yD = min(N-1,y+d)
19                 dNoise = noise[x,yD] + noise[xD,y]
20                 total += noise[x,y]*dNoise
21                 sumSq += noise[x,y]**2
22         f_d = total/(2*sumSq)
23         return f_d
```

```
24
25      f_ds = []
26      ds = np.arange(1,N)
27      f_ds.append(f(0))
28      vals = []
29      founds = np.zeros((len(rs)), dtype=bool)
30      for i in range(len(rs)):
31          vals.append(None)
32
33      vals[0] = 0
34
35      for d in ds:
36          f_d = f(d)
37          for i in range(1,len(rs)):
38              if (not founds[i] and f_d < f_ds[0]*rs[i
    ]/100):
39                  vals[i] = d-1
40                  founds[i] = True
41          f_ds.append(f_d)
42      ds = np.concatenate(([0],ds))
43
44      plt.plot(ds, f_ds)
45      return vals
46
47 #noisetype = "perlin"|"gaussian", N = size of output, ok
    = #octaves|k value
48 #output: return 2D array of perlin noise
49 def get_noise(noisetype, N, ok = 1):
```

```
50      if noisetype == "gaussian":
51          return smooth_gaussian_noise(ok, N)
52      elif noisetype == "perlin":
53          gen = MyPerlinNoise(ok)
54          noise = np.zeros([N,N])
55          for i in range(N):
56              for j in range(N):
57                  noise[i,j] = gen((i/N,j/N))
58          return noise
59      elif noisetype == "random":
60          noise = np.zeros([N,N])
61          for i in range(N):
62              for j in range(N):
63                  noise[i,j] = random.uniform(-1,1)
64          return noise
65      else:
66          print("ERROR, noisetype != perlin|gaussian")
67
68  N = 250
69  oct = 1.25
70  k = 100
71
72  A = get_noise("perlin", N,oct)
73  B = get_noise("gaussian", N, k)
74  C = get_noise("random", N)
75  #plt.imshow(C)
76  plt.show()
77
```

```
78 rvalues = FastCorrelationFactor(B)
79 for i in range(len(rs)):
80     if rvalues[i] != None:
81         plt.axvline(rvalues[i], color = colors[i], label
   = "r{} = {}".format(rs[i], rvalues[i]))
82     print("r value {} = {}".format(rs[i], rvalues[i]))
83
84
85 #plt.title("R Values of Random Noise")
86 plt.title("R Values of Smooth Gaussian Noise (k = {})".
   format(k))
87 plt.xlabel("Distance")
88 plt.ylabel("Spatial Correlation")
89
90 plt.legend(loc="upper right")
91 plt.show()
92
93 import numpy as np
94 import matplotlib.pyplot as plt
95 from SmoothGaussian import *
96 from MyPerlin2 import *
97
98 colors = ['#66c2a5','#fc8d62','#8d9fca','#e789c2','#
   a5d853','#ffd82f','#e5c494']
99
100 def FastCorrelationFactor(noise):
101     N = len(noise)
102     def f(d):
```

```
103         total,sumSq = 0,0
104         for x in range(N):
105             for y in range(N):
106                 xD = min(N-1,x+d)
107                 yD = min(N-1,y+d)
108                 dNoise = noise[x,yD] + noise[xD,y]
109                 total += noise[x,y]*dNoise
110                 sumSq += noise[x,y]**2
111         f_d = total/(2*sumSq)
112         return f_d
113 
114     f_ds = []
115     f_ds.append(f(0))
116     vals = []
117     founds = np.zeros((len(rs)), dtype=bool)
118     for i in range(len(rs)):
119         vals.append(None)
120 
121     vals[0] = 0
122 
123     d = 0
124     while(d < N and not founds[len(rs)-1]):
125         f_d = f(d)
126         i = 1
127         for i in range(1,len(rs)):
128             if (not founds[i] and f_d < f_ds[0]*rs[i
    ]/100):
129                 vals[i] = d-1
```

```
130                     founds[i] = True
131             f_ds.append(f_d)
132             d+=1
133 
134     return vals
135 
136 def get_noise(noisetype, N, ok):
137     if noisetype == "g":
138         return smooth_gaussian_noise(ok, N)
139     elif noisetype == "p":
140         return perlin_noise(ok,N)
141     else:
142         print("ERROR, noisetype != perlin|gaussian")
143 
144 
145 nt = "p"
146 N = 100
147 stepsize = 10
148 t = 5
149 params = np.arange(1,N,stepsize)
150 rs = [100,90,75,50,25,10,0]
151 
152 rvals = np.zeros([len(params),len(rs)])
153 
154 if nt == "g":
155     paramtype = "Octave Size"
156     noisename = "Perlin"
157 else:
```

```
158     paramtype = "K"
159     noisename = "Gaussian"
160
161 for i in range(len(params)):
162     print(params[i])
163     noise = get_noise(nt, N, params[i])
164     vs = []
165     for j in range(t):
166         print(t)
167         vs += FastCorrelationFactor(noise)x
168
169     rvals[i] = vs
170
171 rvals = np.rot90(np.fliplr(rvals))
172 print(rvals)
173
174 for i in range(len(rvals)):
175     plt.scatter(params, rvals[i], color = colors[i],
    marker = "x")
176     a,b = np.polyfit(params,rvals[i],1)
177     plt.plot(params, a*params+b, color = colors[i],label
    = "r{}".format(rs[i]))
```

### 9.2.6 Simulation

```
1 import math
2 import numpy as np
3 import random
4
```

```
5  def size(coord):
6      x,y = coord
7      return math.sqrt(x*x + y*y)
8
9  class Cell:
10     def __init__(self, x:float, y:float, id:int, r:float)
       :
11         self.x = x
12         self.y = y
13         self.id = id
14         self.r = r
15
16 class Pair:
17     def __init__(self, x:float, y:float):
18         self.x = x
19         self.y = y
20     def printPair(self):
21         return ("({},{})".format(self.x,self.y))
22
23 def create_Cells(n, N, minRadius, maxRadius):
24     pairs = []
25     while(len(pairs)!=n):
26         x = np.random.randint(0,N)
27         y = np.random.randint(0,N)
28         if (x,y) not in pairs:
29             pairs.append((x,y))
30     i = 0
31     C = []
```

```
32      for x,y in pairs:
33          r = np.random.uniform(minRadius,maxRadius)
34          c = Cell(x,y, i, r)
35
36          C.append(c)
37          i+=1
38      return C
39  def unit(coord):
40      if coord == (0,0):
41          return (0,0)
42      x,y = coord
43      sz = size(coord)
44      return (x/sz,y/sz)
45
46  def F(I, J, maxRadius, mu, kc):
47      #print("{},{}: ".format(I.id,J.id), end  ='')
48      if (I == J):
49          #print("same node, force = (0,0)")
50          return Pair(0,0)
51      ri = (I.x,I.y)
52      rj = (J.x,J.y)
53      sij = I.r+J.r       #natural separation between cell
    I and J
54      rij = (I.x-J.x, I.y-J.y)          #distance between
    cells
55
56
57      #too far away, no attraction or repulsion
```

```
58      if size(rij) > maxRadius:
59          #print("far away, force = (0,0)")
60          return Pair(0,0)
61
62      urij = unit(rij)
63      #if too close together, repel
64      if size(rij) < sij:
65          frac = (size(rij) - sij)/sij
66          const = mu*sij*math.log(1+frac)
67          urijx,urijy = urij
68          ans = Pair(urijx*const, urijy*const)
69          #print("too close, force = {}".format(ans.
    printPair()))
70
71          return ans
72
73      #if apart but not too far, attract
74      frac = (size(rij)-sij)/sij
75      exponential = math.exp(-kc*frac)
76      const = mu*(size(rij)-sij)*exponential
77      urijx,urijy = urij
78      ans = Pair(urijx*const,urijy*const)
79      #print("far, force = {}".format(ans.printPair()))
80
81      return ans
82
83  def Cells_to_Coords(C, n):
84      xs = []
```

```
85      ys = []
86      rs = []
87      for i in range(n):
88          xs.append(C[i].x)
89          ys.append(C[i].y)
90          rs.append(C[i].r*100)
91      return xs,ys,rs
92
93  def update_cells(forces,C,n,t, cellSpeed):
94      xForce = 0
95      yForce = 0
96      for i in range(n):
97          for j in range(n):
98              xForce += forces[i,j].x
99              yForce += forces[i,j].y
100         C[i].x += xForce*t*cellSpeed
101         C[i].y += yForce*t*cellSpeed
102         xForce = 0
103         yForce = 0
104     return C
105
106 def add_noise(C, PNx, PNy, n,N, noiseSize,cellSpeed):
107     for i in range(n):
108         xNoise = PNx((C[i].x/N, C[i].y/N))
109         yNoise = PNy((C[i].x/N, C[i].y/N))
110         C[i].x += random.uniform(-1,1)
111         C[i].y += random.uniform(-1,1)
112         #C[i].x += xNoise*noiseSize*cellSpeed
```

```
113         #C[i].y += yNoise*noiseSize*cellSpeed
114     return C
115 
116 import numpy as np
117 import matplotlib.pyplot as plt
118 import matplotlib.animation as animation
119 import math
120 from scipy.interpolate import LinearNDInterpolator
121 from scipy import interpolate
122 from matplotlib.widgets import Slider, Button, TextBox
123 from MyPerlin import *
124 from SimHelper import *
125 
126 global c
127 global C
128 global n
129 global PNx
130 global PNy
131 global newn
132 global mu
133 global kc
134 global t
135 global noiseSize
136 global cellSpeed
137 
138 N = 100 #size of grid
139 init_n = 50 #number of points
140 n = init_n
```

```
141 init_cellSpeed = 1
142 cellSpeed = init_cellSpeed
143 init_noiseSize = 1
144 noiseSize = init_noiseSize
145 newn = n
146 C = []
147 maxRadius = 6
148 minRadius = 2
149 init_mu = 0.05
150 mu = init_mu #spring constant, controls size of force
151 init_kc = 1
152 kc = init_kc #defines decay of attractive force
153 init_t = 0.5
154 t = init_t #percentage of a second each timestep takes
155 init_octaves = 4
156 octaves = init_octaves
157 #light to dark
158 colorpallete = ['#f2f4f9','#b8d1ff','#c2cee5','#51a2da']
159 COL = colorpallete[3]
160 COL2 = colorpallete[2]
161 COLinnerbg = colorpallete[1]
162 COLbg = colorpallete[0]
163
164 PNx = MyPerlinNoise(init_octaves,1)
165 PNy = MyPerlinNoise(init_octaves,2)
166
167 fig, ax = plt.subplots()
168 fig.subplots_adjust(left = 0.25, bottom = 0.3)
```

```
169 ax.set_xlim(0,N)
170 ax.set_ylim(0,N)
171 #ax.set_title("Cell Behaviour with Random Noise",
        fontsize = 20)
172 ax.set_xticks([])
173 ax.set_yticks([])
174 ax.set_facecolor(COLinnerbg)
175 fig.set_facecolor(COLbg)
176 plt.setp(ax.spines.values(), color = COL2, linewidth = 4)
177
178
179 #octave changing
180 axOctaves = fig.add_axes([0.25,0.25,0.65,0.03])
181 OctSlider = Slider(
182     ax = axOctaves,
183     label = "Number of Octaves",
184     valmin = 1,
185     valmax = 10,
186     valinit = init_octaves,
187     initcolor = 'none',
188     color = COL2,
189     track_color = COL2,
190 )
191 def updateOctaves(val):
192     octaves = OctSlider.val
193     global PNx
194     global PNy
195     PNx = MyPerlinNoise(octaves, 1)
```

```
196     PNy = MyPerlinNoise(octaves,2)
197 OctSlider.on_changed(updateOctaves)
198 
199 #noise size
200 axNoiseSize = fig.add_axes([0.25,0.2,0.65,0.03])
201 NoiseSlider = Slider(
202     ax = axNoiseSize,
203     label = "Noise Effect",
204     valmin = 0,
205     valmax = 10,
206     valinit = init_noiseSize,
207     initcolor = 'none',
208     color = COL2,
209     track_color = COL2,
210 )
211 def updateNoiseSize(val):
212     global noiseSize
213     noiseSize = val
214 NoiseSlider.on_changed(updateNoiseSize)
215 
216 #mu
217 axMu = fig.add_axes([0.25,0.15, 0.65,0.03])
218 MuSlider = Slider(
219     ax = axMu,
220     label = "Mu (Spring Constant)",
221     valmin = 0,
222     valmax = 1,#play around
223     valinit = init_mu,
```

```
224     initcolor = 'none',
225     color = COL2,
226     track_color = COL2,
227 )
228 def updateMu(val):
229     global mu
230     mu = MuSlider.val
231 MuSlider.on_changed(updateMu)
232
233 #kc
234 axKC = fig.add_axes([0.25,0.1,0.65,0.03])
235 KCSlider = Slider(
236     ax = axKC,
237     label = "KC (attraction decay)",
238     valmin = 0,
239     valmax = 10, #play around
240     valinit = init_kc,
241     initcolor = 'none',
242     color = COL2,
243     track_color = COL2,
244 )
245 def updateKC(val):
246     global kc
247     kc = KCSlider.val
248 KCSlider.on_changed(updateKC)
249
250 #cell speed
251 axCellSpeed = fig.add_axes([0.25,0.05,0.65,0.03])
```

```
252 cellSpeedSlider = Slider(
253     ax = axCellSpeed,
254     label = "Cell Speed",
255     valmin = 0,
256     valmax = 10, #play around
257     valinit = init_cellSpeed,
258     initcolor = 'none',
259     color = COL2,
260     track_color = COL2,
261 )
262 def updateCellSpeed(val):
263     global cellSpeed
264     cellSpeed = cellSpeedSlider.val
265 cellSpeedSlider.on_changed(updateCellSpeed)
266
267
268
269 #adding new cells
270 axNewCell = fig.add_axes([0.1,0.3, 0.09, 0.05])
271 bNewCell = Button(axNewCell, "Add", color = COL2)
272 def add_cell(val):
273     global newn
274     xs,ys,rs = Cells_to_Coords(C,n)
275     pairs = []
276     for i in range(n):
277         pairs.append((xs[i],ys[i]))
278     added = False
279     while not added:
```

```
280             x = np.random.randint(0,N)
281             y = np.random.randint(0,N)
282             r = np.random.uniform(minRadius,maxRadius)
283             if (x,y) not in pairs:
284                 newCell = Cell(x,y,n,r)
285                 C.append(newCell)
286                 newn += 1
287                 added = True
288 bNewCell.on_clicked(add_cell)
289
290 def update_n():
291     global n
292     n = newn
293
294
295
296
297 def animate(i):
298     global C
299     update_n()
300     forces = []
301     for x in range(n):
302         for y in range(n):
303             forces.append(F(C[x], C[y], maxRadius, mu, kc
    ))
304
305     forces = np.reshape(forces, (n,n))
306
```

```
307     update_cells(forces,C,n,t, cellSpeed)
308     PNx = MyPerlinNoise(octaves, 1)
309     PNy = MyPerlinNoise(octaves,2)
310     add_noise(C,PNx, PNy, n,N, noiseSize, cellSpeed)
311     xs,ys,rs = Cells_to_Coords(C,n)
312 
313     line = ax.scatter(xs,ys, s = rs, c = COL, alpha =
        0.6)
314     return line,
315 
316 def init():
317     global C
318     update_n()
319 
320     C = create_Cells(n,N, minRadius, maxRadius)
321 
322     xs,ys,rs = Cells_to_Coords(C,n)
323     line = ax.scatter(xs,ys ,s= rs, c = COL, alpha = 0.8)
324     return line,
325 
326 #start button
327 axStart = fig.add_axes([0.1,0.35, 0.09, 0.05])
328 bStart = Button(axStart, "Start", color = COL2)
329 def start(val):
330     ani = animation.FuncAnimation(fig,animate,np.arange
        (1,200), init_func = init, interval = 1000*t, blit =
        True)
331     plt.show()
```

```
332     bStart.set_active(False)
333 bStart.on_clicked(start)
334 
335 #get input values for time and number of circles
336 #time changing
337 axTime = fig.add_axes([0.1,0.45,0.09,0.03])
338 TimeSlider = Slider(
339     ax = axTime,
340     label = "Time scale",
341     valmin = 0,
342     valmax = 1,
343     valinit = init_t,
344     initcolor = 'none',
345     color = COL2,
346     track_color = COL2,
347 )
348 def updateTime(val):
349     global t
350     t = val
351 TimeSlider.on_changed(updateTime)
352 
353 
354 fig.text(0.08,0.55, "Population A", fontsize = 12)
355 
356 #n changing
357 axn = fig.add_axes([0.1,0.5,0.09,0.03])
358 nSlider = Slider(
359     ax = axn,
```

```
360     label = "No of Cells",
361     valmin = 0,
362     valmax = 1000,
363     valstep = 10,
364     valinit = init_n,
365     initcolor = 'none',
366     color = COL2,
367     track_color = COL2,
368 )
369 def updaten(val):
370     global newn
371     newn = val
372 nSlider.on_changed(updaten)
373
374
375 plt.show()
```