Turing pattern low diaphony pointset

Chris van den Oetelaar

April 2021

Figure 1: Physicist and model Youri S. used as a seed for a Turing pattern.

Contents

1 Introduction

Quasi random pointsets offer faster error estimate behaviour than $\mathcal{O}(\frac{1}{\sqrt{N}})$, without producing uniform predictable patterns. Thus making them suitable for quasi Monte Carlo. In this project we will take inspiration from nature in the form of Turing patterns. Alan Turing (of computer fame) found that two diffusing interacting chemical products can produce stable patterns from almost uniform random starting conditions[8]. He proposed this is the basis for morphogenesis, how complex patterns in nature are formed from a homogeneous state. Two examples are shown in figure 2 and 3. As Turing patterns are evenly spaced and can be produced from almost any starting situations, they show potential to be turned into low diaphony quasi random pointsets.

Figure 2: Pattern on pufferfish[2]

Figure 3: Pattern on zebra[1]

2 Random pointsets

To integrate a function using the Monte Carlo method, a pointset is required to determine where the function will be sampled. If we look at a 2d function, we can create a 2d pointset by creating pairs of random numbers as x and y coordinates on our plane. A downside of true random numbers is that they are relatively slow to generate. A random process, such as quantum fluctuations need to be produced and recorded. This takes time and specialised equipment. There is also no way to repeat or verify results unless the pointset is saved in its entirety. Computer algorithms can create pseudo random numbers. Algorithms such as RCARRY, RANLUX, and the Mersenne twister can produce pointsets fast. They also offer repeatably where only the hyper parameters need to be saved instead of the entire pointset. They are also easier to implement, which is why we have chosen for RCARRY. A 2d pointset generated using RCARRY can be seen in figure 4.

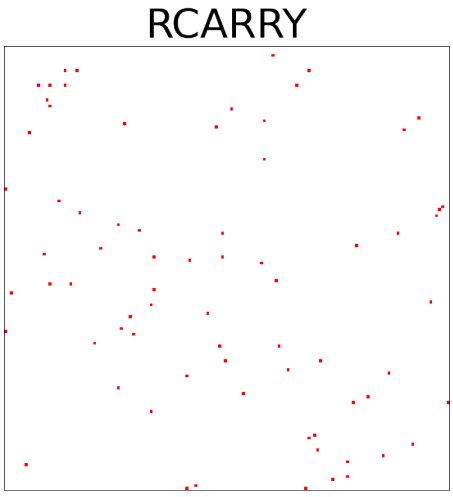

Figure 4: 73 points generated with RCARRY

3 Argument for quasi random pointsets

One thing to note about pointsets generated with random or pseudo random numbers is the nonuniformity of the set. Areas with over- and underdensities can clearly be seen in figure 4. As new points are generated independently from the locations of the previous points, over- and underdensities occur naturally. This leads to the relatively slow error estimate behaviour[11] when used in Monte Carlo integration of $\mathcal{O}(\frac{1}{\sqrt{N}})$, where N is the number of points in the pointset. This behaviour be seen in equation 1. Thus we might want to look at non truely random pointsets to increase error estimate behaviour with regards to N.

$$\frac{\sqrt{\hat{E}_2}}{E_1} = \frac{\sqrt{\frac{1}{N^3}(NS_2 - S_1^2)}}{\frac{1}{N}S_1} \propto \frac{1}{\sqrt{N}} \tag{1}$$

Using truly uniform pointsets does not lead to desirable results either. 100 uniform points are shown in figure 5. It is quite easy to imagine a periodic function which would align with such a uniform pointset. If the points line up such that they all lie within the peaks or valleys of a periodic function, the errors of the Monte Carlo integration will be rather large.

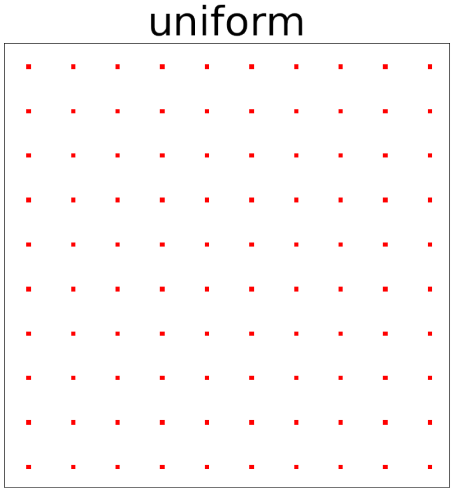

Figure 5: 100 uniform points

Quasi random pointsets are a middle ground between the truly random and uniform pointsets. It aims to produce relatively evenly spaced points in an unpredictable pattern. By using a quasi random pointset, Quasi Monte Carlo solves the over- and underdensity regions resulting in the $\mathcal{O}(\frac{1}{\sqrt{N}})$ error estimate behaviour of random pointsets. While not being as vulnerable to obvious patterns of uniform pointsets. In this project we will attempt to turn Turing patterns into a quasi random pointset.

4 Turing patterns

Turing patterns are stationary patterns of lines and dots that can be found in nature, as seen in figure 2 and 3. The basis of the pattern lies in chemistry. Two interacting substances, an autocatalytic activator and an inhibitor, with different diffusion rates. The starting condition is a mixed state, which in our case we will call the seed, as seen in figure 7. Self autocatalysation creates local patches of the activator, and the inhibitor blocks patch growth. There are multiple ways to create Turing patterns. Such as the FitzHugh-Nagumo[3] [4], Grierer-Meinhardt[5], and Brusselator[6] models. In this project, a new method will be used, to create the Turing patterns that only has 2 hyper parameters and is robust to their choices, always producing Turing patterns. This method follows several iterations of the chemical process to result in a stable pattern. The process can be followed in figure 6. It starts with a seed produced by the pseudo random number generation method (RCARRY). Then a high pass or sharpening filter acts as the activator autocatalysing, this creates high contrast regions. The values of the kernel are $\frac{-1}{distance\ to\ center\ pixel}$ with the center pixel having the positive value equal to the sum of all other pixels. The kernel of the high pass filter is shown in table 1, keep in mind that -0.71 = $\frac{-1}{\sqrt{2}}$. Thus a 3 by 3 kernel is depicted table 1. The kernel size used in the project is 9. A threshold is applied to solidify the activator and inhibitor boundaries. Followed by a Gaussian blur as a model for the diffusion between the substances. The kernel of the Gaussian blur is shown in table 2 The values of the kernel are $e^{-x^2-y^2}$ with [x,y]=[0,0] for the center pixel. The 3 by 3 kernel is depicted in table 2. The kernel size used in the project is 9, equal to the high pass kernel size.

Table 1: high pass kernel of size 3

-0.71	-1	-0.71
-1	6.83	-1
-0.71	-1	-0.71

Table 2: Gaussian blur kernel of size 3

0.04	0.12	0.04
0.12	0.33	0.12
0.04	0.12	0.04

Repeating these steps, the Turing pattern is slowly formed, as seen in figure 7. The iterations are stopped when the output of the current iteration matches the previous one, so no change has been made. The processed is then stopped at the threshold step, producing a Turing pattern with crips lines, only consisting of ones and zeros, as seen in figure 7 (right).

The advantage of using these kernels, is that they are able to wrap around the boundaries. Thus if applied on the left edge, they partly wrap to the right hand of the pattern. This produces a pattern which has periodic boundary conditions, or is perfectly tileable. The patterns continue from the left to the right side seamlessly, as well as from top to bottom. This is shown in figure 8.

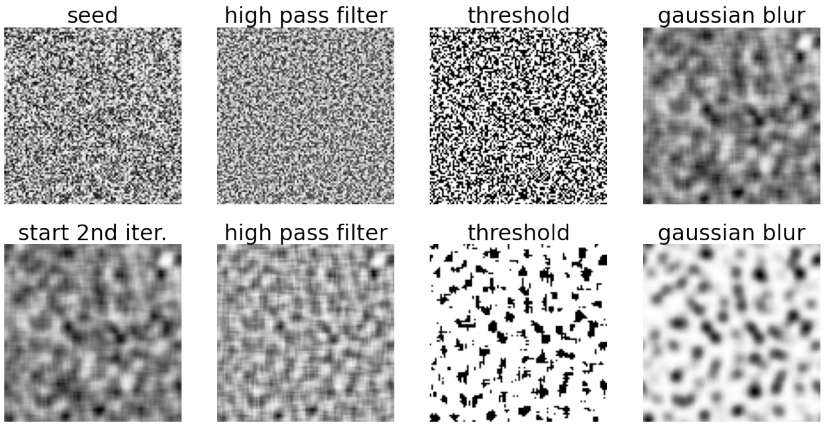

Figure 6: The iterative process of Turing pattern creation. Starting with a random seed (top left). Then a high pass filter, a threshold is applied and a Gaussian blur (top right). The process is then iterated (bottom left to bottom right)

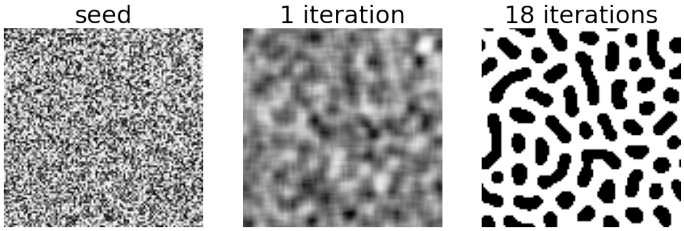

Figure 7: The iterative process of Turing pattern creation. Starting with a random seed (left). After the first iteration (middle). Until finally after several iterations a stable state has been reached (right).

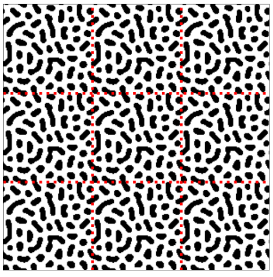

Figure 8: The resulting Turing patterns have periodic boundary conditions, meaning they are perfectly tileable. The dotted red lines show 9 tiles of the same Turing pattern as in figure 7.

5 Diaphony

Diaphony (T) is used as a measure of uniformity of point sets.

$$T(X) \;=\; \frac{1}{N}\sum_{n}^{\alpha}\sigma_n^2|\sum_{j=1}^{n} exp(2i\pi n x_j)|^2 \;=\; \frac{1}{N}\sum_{j,k=1}^{N}\beta(x_j - x_k) \tag{2}$$

$$\beta(z) \;=\; \sum_{n}^{*}\sigma_n^2 exp(2i\pi n z) \tag{3}$$

We can use this to determine how uniform our Turing pattern generated pointset is, compared to random pointsets (T=1) and a uniform pointset (T=0). The goal is to create low diaphony pointsets without the predictable patterns of the uniform pointset. There are different diaphonies that can be used to make the calculation of β in equation 3 easier. The Gulliver diaphony is used in this project, shown in equation 4. The hyper parameter $0 < s < 1$ is set to 0.5.

$$\beta_G(z) \;=\; ((\frac{1+s}{1-s})^d - 1)^{-1}(-1 + \prod_{\mu=1}^{d}\frac{1-s^2}{1-2s\ cos(2\pi x^\mu) + s^2}) \tag{4}$$

This now provides us with a method to consistently determine the uniformity of a pointset. As shown in figures 9, 10 , 11, The diaphony of N points in the same place (all at [0.5,0.5]) is equal to N. The diaphony of a random pointset is on average equal to 1. The diaphony of a uniform pointset is close to 0. The goal is to generate pointsets from Turing patterns with a low diaphony without obvious patterns. Which means it must lie between figures 10 and 11.

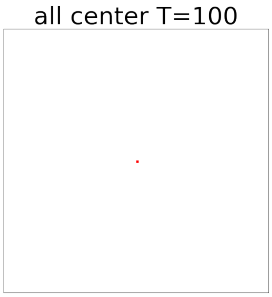

Figure 9:
100 points at [0.5,0.5]

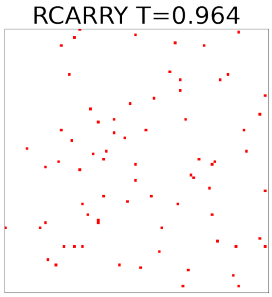

Figure 10:
100 random points

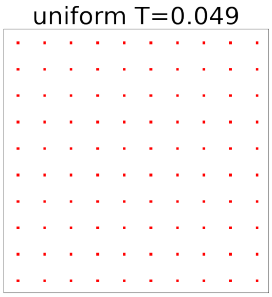

Figure 11:
100 uniform points

6 Pattern to pointset conversion

Now that the Turing pattern has been produced, it needs to be converted to a pointset. There are a few ways to go about this process. We will take a look at 5 of them: Shrinking, blob detection, kmeans, and a custom kmeans method. The first 4 are general approaches to create points from patterns. The custom kmeans combines the best aspects from the blob detection and the kmeans for Turing patterns.

6.1 Shrinking

The most simple solution is to take each blob of the turing pattern and shrink it to a single pixel, thus creating a pointset. This can be achieved through an iterative process. The blobs in the pattern have value 1 while the surrounding values are 0. For each pixel in the image that has value 1, we check if some but not all of the neighbours have value 0. If all neighbours have value 0 it would mean it is a lone pixel, thus does not reducing. If all neighbours have value 1 it would mean it is in the center of a blob. If some but not all of the neighbours have value 0 it means the pixel is at the edge of a blob and thus can be set to 0. This is iterated, thus shrinking the edges away until only lone pixels remain. The neighbours to look at can either be only directly adjacent, we will call this method square, resulting in figure 12. Or look at diagonal neighbours too, we will call this method round, resulting in figure 13.

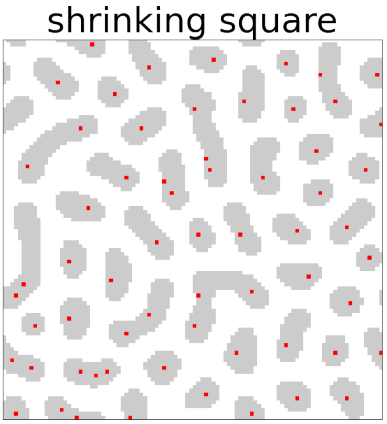

Figure 12: Pointset generated from Turing pattern using the square shrinking method.

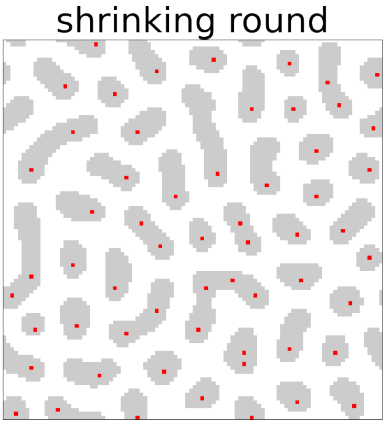

Figure 13: Pointset generated from Turing pattern using the round shrinking method.

The square shrinking method seems to put points more often too close together, when compared to the round shrinking method. This is due to it creating multiple points for some small blobs. However, both methods do not handle large blobs well, sometimes reducing them to single points and thus leaving un-

derdensities in the pointset. An advantage of this method is the that it handles the blobs on the edges well, due to the algorithm that looks for neighbours wrapping around the edges. This means that blobs that continue around the borders are seen as single blobs and not as separate blobs. This is a problem we will come across with other algorithms.

6.2 Kmeans

Kmeans[9] is a clustering algorithm from the scikit-learn package in python, classifying as a regression machine learning method. Which means it minimises a loss function. The Turing pattern needs to be converted to points representing the blobs on the pattern first. This is done by creating a point at each pixel with value equal to 1. This transforms the blobs in usable points for the Kmeans algorithm. It attempts to create k centers such that the mean quadratic distance between each point and the closes center is smallest. The loss function that is to be minimised is shown in equation 5, where x_i are the coordinates of the points representing the blobs in the Turing pattern and $x_{closest\ center}$ is the closest center point determined by the Kmeans alorithm.

$$Loss \ = \ \sum_{i=0}^{all\ points} (x_i - x_{closest\ center})^2 \tag{5}$$

It begins by distributing k centers all around the space and then moves them each iteration to decrease the loss function. This continues until no improvement has been made anymore in an iteration. Due to the tendency of the algorithm to settle in local minima, this process is repeated 10 times and the best distribution of centers is kept. Because the algorithm requires the k amount of centers to be set beforehand, an estimate for k has to be made. This depends on the size and scale of the Turing pattern generated. The estimate used for k is shown in equation 6, where n is the width and height of the square Turing pattern in pixels and κ is the size of the high pass kernel and Gaussian blur kernel in pixels. This is due to the kernel sizes determining the thickness or pixel size of the blobs in the Turing pattern. The factor 2 is an empirical value, it is adjusted if the packing efficiency or the distance from one blob to another changes due to differences in Turing pattern generation, such as the value of the threshold (which set to 0.5 in this project).

$$k \ = \ \frac{n^2}{2 * \kappa^2} \tag{6}$$

We do need to keep the edges in mind. The Kmeans algorithm does not support periodic boundary conditions. We could sacrifice performance by tiling the input image as shown in figure 8, thus making the blobs on the boundary truely visible for the algorithm. Then the centers found within the middle tile are selected as the pointset. However, the Kmeans method is already the most expensive method to run. The tiling method would mean the input is even 9 times larger, as well as the k. The average complexity of Kmeans already scales

with $\mathcal{O}(n_{pix}kT)$ where n_{pix} is the points in the dataset and k the amount of centers, and T the number of iterations before a minimum is found. If we tile the input, n increases by a factor of 9. and k does as well. Thus decreasing the speed by a factor of 91 before taking into account the increased number of iterations needed to find a minimum with an input 9 times as large. We will cut our losses with the Kmeans algorithm by just inputing the Turing pattern without tiling and accepting the algorithms inability to handle blobs at the edges properly. This results in the pointset in figure 14.

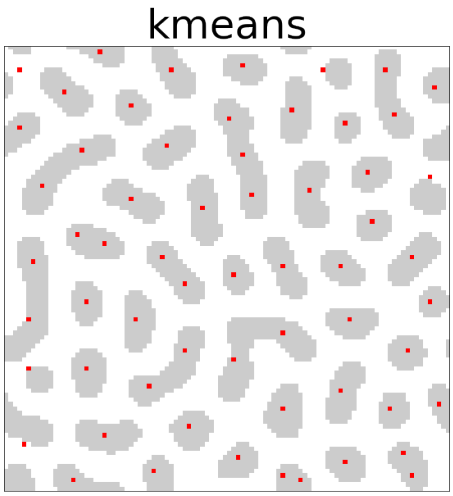

Figure 14: Pointset generated from Turing pattern using the Kmeans method.

We can see some small blobs having multiple centers, and some centers lying between two or three blobs, especially at the edges and corners of the image. This is due to having to guess the number of centers to produce (k) and the algorithm not being able to handle the periodic boundary conditions.

6.3 Blob detection

The OpenCV package in python has a built in SimpleBlobDetector[10] function. It can convert the Turing patterns into a point set by taking the center of each blob as a point. The algorithm works in 4 steps:
1. Thresholds. Create binary images where each pixel is 0 if below and 1 if above a certain thresholds. This way a binary image is created for each threshold, which each helps detect blobs at certain brightness levels. If the input image is binary (only ones and zeros) we only need one threshold.
2. Grouping. All connected 1 value pixels in each threshold image are clustered into blobs. In the case of the binary Turing pattern, this would put each blob in a seperate group. There are some constraints on if a cluster of pixels is considered a group. These will be discussed later.

3. Merging. The centers of each blob are calculated. If centers of 2 groups are too close together, the two groups are merged into one.
4. Center calculation. The mean position of the pixels of the group is calculated with equation 7, where c is the center coordinate for the group of n pixels, and x_i the coordinate of a pixel in the group.

$$c = \frac{1}{n}\sum_{i=0}^{n} x_i \tag{7}$$

The input Turing pattern can either be a binary image, as we did with the other methods. Or we can put a Gaussian blur over the image first. The Gaussian blur can help to break up the larger blobs, which would only create a single point for the binary input, into multiple blobs.

As mentioned before, there are some restrictions on pixel clusters being classified as groups or not. The 5 main constraints are shown in figure 15.

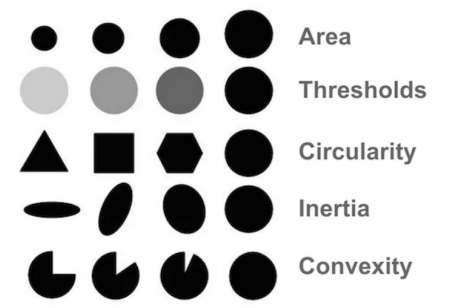

Figure 15: Main constraints of the SimpleBlobDetector from OpenCV[7].

Due to the non circular shapes of the blobs in the Turing pattern, this method works best with all the constraints except for the area disabled. Blobs at the border of the image are ignored by the algorithm, which only groups them if there are edges of empty space around them (or zeros). We can apply the tiling solution also proposed for the Kmeans method at the end of the previous section. But a less computing power consuming method is adding a 1 pixel thin border of zeros around the entire perimeter of the pattern. Then the big blobs cut off at the center are detected, while the small blob edges are ignored. This does however induce a small bias towards the center of the space, just as with the previous method. When we compare the binary input shown in figure 16 with the blurred input in figure 17, Only marginal gain can be seen by the Gaussian bur dividing one larger blob into multiple centers.

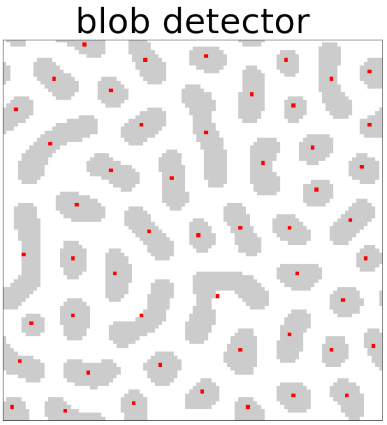

Figure 16: Pointset generated from Turing pattern using the SimpleBlobDetector with a binary input

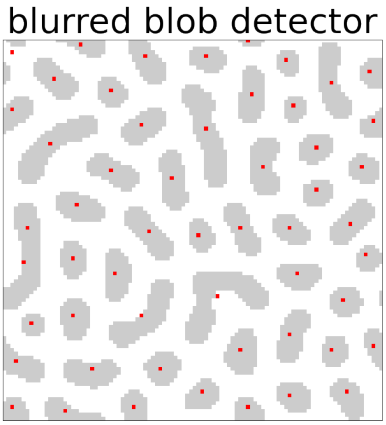

Figure 17: Pointset generated from Turing pattern using the SimpleBlobDetector with a blurred input

6.4 Periodic custom Kmeans

Lastly, we can create an entirely custom clustering algorithm. It will be partly based on the OpenCV SimpleBlobDetection method and partly on the Kmeans algorithm. It will also use periodic boundary conditions, to not split up the blobs at the edges. And has to be able to produce multiple points for the large blobs. We will now take a look at the steps also used in the SimpleBlobDetection method, and adjust them for our needs.
1. Thresholds. The threshold step was only needed for the Gaussian input, which was meant to create multiple centers for large blobs. This algorithm will create multiple centers out of the larger blobs in the last step, so this step can be skipped.
2. Grouping. The grouping now has to be done with periodic boundary conditions in mind. The blobs that are cut in pieces by the image border are now correctly seen as one. This is due to the breath first algorithm that looks for connecting pixels being able to wrap around the borders of the pattern. We do need to keep in mind to use the 'pre-wrapped' coordinates for the distance calculation between each point. This means we keep coordinates such as [-1,n] that falls outside of the image as such and not translate them to [n-1,0] before distance calculations, where n is the pixel width of the image.
3. Merging. This step can be skipped as the creation of the Turing patterns already ensures that no blobs can end up too close to one another. Thus no blobs will have to be merged.
4. Number of centers calculation. This is an additional step necessary to determine how many centers (k) each blob should have individually. Where as with the Kmeans method before, the total number of centers (k) had to be estimated beforehand for the entire pattern, we can now determine k for each isolated blob

individually. We can determine the individual k's from the amount of pixels in the blob with equation 8. Where k is the amount of centers for a blob containing n pixels. The κ represents the kernel size, as it does in equation 6. The 1.0 term can be adjusted, if increased, bigger blobs produce less centers. And if the term is lower, bigger blobs produce more centers. The value of 1.0 produced the best result in our observations.
5. Center calculation. The isolated blobs can then individually be fed into the Kmeans algorithm, with their k determined in the previous step with equation 8. The Kmeans algorithm then minimises the squared distances loss function shown in equation 5. We do not have to apply tiling due to the pixel coordinates saved in the 'pre-wrapped' coordinates in step 2. The resulting centers do have to be translated or wrapped back into coordinates within the image. This means that a center at [-1,n] that falls outside of the image boundaries is translated to [n-1,0].

$$k = \frac{1+n}{1.0 \cdot \kappa^2} \tag{8}$$

The resulting pointset can be seen in figure 18. It achieved the goals of evaluating the blobs cut by the boundaries as continuous blobs and producing multiple points for the larger blobs. This makes it the best method evaluated for both of these requirements.

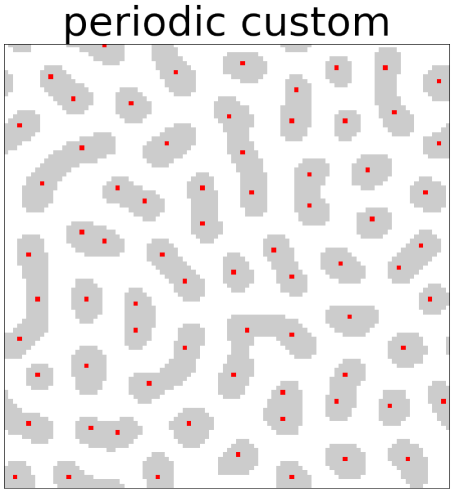

Figure 18: Pointset generated from Turing pattern using the periodic custom Kmeans method.

The computational strain however is higher than the shrinking and blob detection method due to the Kmeans method used on the blobs. It is however faster than the pure Kmeans method as average computational time for Kmean scales with $\mathcal{O}(n_{pix}kT)$. Where the number of pixels (n_{pix}) and number of centers (k) are now lower. Let us assume they are lower by a factor α, we need to run

the algorithm α more times. This still leads to a speed increase as shown in equation 9. This is without taking into account the speed increase resulting from decreased number of iterations (T) needed with smaller n_{pix}.

$$\mathcal{O}(\alpha \cdot \frac{n_{pix}}{\alpha} \frac{k}{\alpha}) \;=\; \mathcal{O}(\frac{1}{\alpha} n_{pix} k) < \mathcal{O}(n_{pix} k) \tag{9}$$

A different way to calculate the points blob centers could be used for the single point blobs to speed up the process. The mean distance in equation 7 from the SimpleBlobDetector could be used, but only for the small blobs containing a single point. This method would not work for the larger blobs which need to be converted into multiple points. This would then lead to a mix of methods used in determining the centers which leads to inconsistencies in the pointset. With the larger blobs taking the most time for the Kmeans fit anyhow, the slight time saved is not worth the inconsistency in the pattern by using a different method for the small blobs calculation method. As such, all blobs in this method are evaluated through Kmeans.

7 Results

The pointsets resulting from all methods have already been shown visually. But to quantify their uniformity all diaphonies of the previously displayed pointsets are shown in table 3. The table also includes the random pointset from the RCARRY algorithm from figure 4 and the uniform dataset from figure 5. To see the reliability of the different methods, the process was repeated 100 times with Turing patterns generated from different seeds. The means and standard deviations of those runs are shown in table 4. The Kmeans method was not included in the 100 runs at it is far slower than the other methods.
The amount of points generated by the method from the Turing pattern (k) is shown in the first column (k points). Of all the methods, the periodic custom Kmeans method was able to extract the most amount of points successfully from the pattern. We need to acknowledge that the pure Kmeans can of course produce any amount of points by adjusting its k factor. But increasing it past its current estimate reduces the quality of pointset generated by producing points between blobs or clustering multiple points too close together.
As seen from the second column, the diaphony, the custom method was able to produce the most uniform pointset out of the Turing pattern.

The third column shows the order at which the method runtime scales with the amount of points generated (k). We are using an n by n pixel size image for the Turing pattern, so $n_{pix} = n^2$. Keep in mind that the image pixel width (n) needed scales with kernelsize (κ) and k. as $n^2 \propto k \cdot \kappa^2$ as shown in equation 6. As the kernel size is a property of the Turing pattern method, and does not change between methods or with k, it is kept out of the runtime analysis. Thus we can say that $k \propto n^2$. We can then evaluate each method individually.

Table 3: Results from all the methods of converting Turing patterns into pointsets, and the random and uniform pointsets.

method	k points	diaphony	runtime	E(eq 10)	E(eq 11)	E(eq 12)
shrink square	63	0.351	$\mathcal{O}(k)$	0.048	0.045	0.081
shrink round	59	0.244	$\mathcal{O}(k)$	0.054	0.091	0.085
Kmeans	61	0.162	$\mathcal{O}(k^2)$	0.014	0.077	0.058
blob detect	51	0.231	$\mathcal{O}(k)$	0.048	0.071	0.105
blurred blob detect	54	0.310	$\mathcal{O}(k)$	0.021	0.083	0.051
periodic custom	73	0.101	$\mathcal{O}(k)$	0.002	0.032	0.021
random	73	1.143	$\mathcal{O}(k)$	0.029	0.031	0.052
uniform	100	0.049	$\mathcal{O}(k)$	0.002	0.328	0.007

Table 4: Results from all methods except for Kmean, with the means and standard deviation of 100 runs.

method x100	n points	Diaphony	E(eq 10)	E(eq 11)	E(eq 12)
shrink square	62±4	0.259±0.061	0.029±0.022	0.041±0.029	0.070±0.056
shrink round	63±3	0.244± 0.048	0.028±0.020	0.042±0.033	0.071±0.053
blob detect	48±3	0.278±0.069	0.025±0.018	0.041±0.030	0.060±0.048
blurred blob detect	52±3	0.296±0.067	0.020±0.015	0.042±0.028	0.051±0.040
custom Kmeans	67±2	0.099±0.012	0.016±0.012	0.039±0.029	0.038± 0.028
random	67±2	0.980±0.228	0.043±0.029	0.037±0.025	0.104± 0.071
uniform	100±0	0.049±0	0.002±0	0.319±0	0.007±0

Shrink: Scales with he amount of pixels to check and the amount of iterations need, which scales with blob radius which depends on κ. So as mentioned before, we will ignore κ in the analysis, so the runtime scales as $\mathcal{O}(n^2) = \mathcal{O}(k)$.
KMeans: As discussed before, Kmeans follows $\mathcal{O}(n_{pix}kT)$, where you can set a limit to T in the algorithm and there is no hard relationship between the amount of iterations T and k, so we will take it out of the runtime analysis. As $n_{pix} \propto n^2$, we get $\mathcal{O}(n^2k) = \mathcal{O}(k^2)$.
Blob detector: All steps depend on the amount of pixels so $\mathcal{O}(n^2) = \mathcal{O}(k)$
Periodic custom Kmeans: All steps except for the Kmeans step scale with $\mathcal{O}(k)$, from the blob detector. The last step of finding the KMeans for each blob scales with the blob size. Fortunately, the blob sizes do not scale with the image size. a 1000x1000 Turing pattern has similar sized blobs as a 100x100 Turing pattern, just 100 times more of them. And such the runtime of the last step does not scale with the image size n and therefor also not with k. This means the total runtime scaling for the custom method is the same as for the blob detector, $\mathcal{O}(k)$.
The random pointset produced using RCARRY and the uniform pointset are both $\mathcal{O}(k)$ as well.

$$\int_0^1 \int_0^1 x^2 + y^2 \; dxdy \tag{10}$$

$$\int_0^1 \int_0^1 x \cdot sin(20.5\pi y) \; dxdy \tag{11}$$

$$\int_0^1 \int_0^1 e^{-x^2} e^{-y^2} dxdy \tag{12}$$

In the last 3 columns of table 3, the errors are displayed of the Monte Carlo integrals of equation 10, 11, and 12 with the pointsets. Where equation 11 is chosen to specifically show the vulnerability of the uniform pointset which performs well at the other two integrals. We can see that the custom method performs well in comparison to the other Turing pattern methods.

8 Conclusion

We must conclude that of the Turing pattern methods, the periodic custom Kmeans performs the best in all metrics. That is, number of points generated from each image, diaphony, and the evaluation of the integrals. It also performs better than the random dataset when calculating the integrals. Even though it is slower due to the Kmeans step, it has the same runtime scaling as the other algorithms.
The problem with the Turing pattern method lies in the memory scaling. As the amount of pixels in the pattern needed per final point scales with the dimensionality of the pointset. And this is where the problem with the Turing pattern lies, in the Turing pattern generation memory cost. This scales with the hard with the dimensionality as $\mathcal{O}(2 \cdot \kappa^{dims})$ from equation 6. This represents the number of pixels needed to form enough space for a blob big enough to form 1 point in our pointset. Meaning that with the speed of the Turing pattern generation and the custom method, the computer will have run out of memory before getting a speed benefit from the better error estimate behaviour than $\frac{1}{\sqrt{N}}$ of a random number pointset with a diaphony of 1. Especially considering the fact that a random number stream does not need to be saved before being usable for Monte Carlo integration, meaning the memory requirement does not scale with the size of the pointset for the random number stream.

Unfortunately, we will have to put the idea of Turing pattern low diaphony pointset generation away for now, until infinite memory has been made available. Or until we are in need of a social media profile picture that can beat facial recognition by looking like figure 1.

The code used for the project is availabe at:
https://github.com/C-van-den-Oetelaar/Turing-pattern-low-diaphony-pointset

9 References

References

[1] E. Siasoco, "Z is for zebra", https://www.flickr.com/ photos/scfiasco/99659676/, visited 30-04-2021

[2] C. Chap, "Giant pufferfish skin detail", https://commons.wikimedia.org/ wiki/File:Giant_Pufferfish_skin_pattern_detail.jpg, visited 30-04-2021

[3] R. FitzHugh, "Impulses and Physiological States in Theoretical Models ofNerve Membrane", Biophysical Journal 1 (6) (1961)

[4] J. Nagumo, S. Arimoto, S. Yoshizawa, "An Active Pulse Transmission LineSimulating Nerve Axon", Proceedings of the IRE 50 (10) (1962)

[5] A. Gierer, H. Meinhardt, "A theory of biological pattern formation", Kyber-netik 12 (1) (1972)

[6] I. Prigogine, R. Lefever, "Symmetry breaking instabilities in dissipativesys-tems", The Journal of Chemical Physics 48 (4) (1968)

[7] S. Mallick, "Blob Detection Using OpenCV", https://learnopencv.com/ blob-detection-using-opencv-python-c/, visited 30-04-2021

[8] A. Turing, "The Chemical Basis of Morphogenesis", Philosophical Trans-actions of the Royal Society of London B. 237 (641) (1952)

[9] scikit-learn, "Kmeans", https://scikit-learn.org /sta-ble/modules/generated/ sklearn.cluster.KMeans.html, visitied 02-05-2021

[10] OpenCV, "SimpleBlobDetector", https://docs.opencv.org /mas-ter/d0/d7a/ classcv_1_1SimpleBlobDetector.html, visited 02-05-2021

[11] R. Kleiss, "Monte Carlo:Techniques and Theory", (2020)