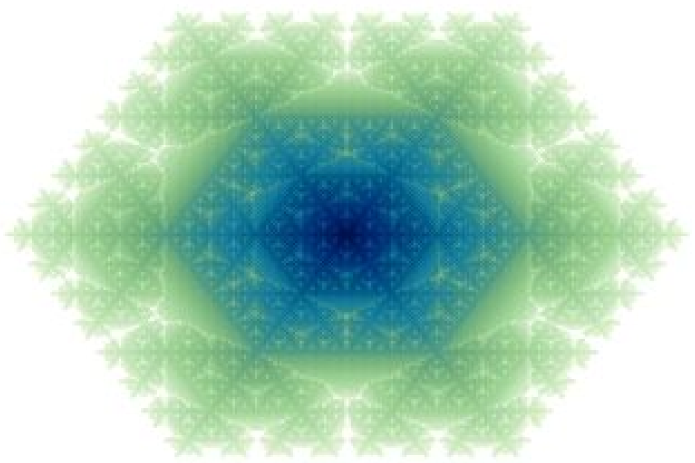
Aalto University
School of Science
PHYS - E0412
Computational Physics

Spring Semester 2019
Final Project:

SIMULATION OF THE GROWTH OF SNOWFLAKES

by Johanna Czech



Contents

List of Figures

1. Introduction and small physical Background

Inspired by my first experience of a 'real' winter this year in Finland, I wanted to understand how snowflakes actually grow and why their look like they do. Therefore, I decided to do a simulation of the growth of snowflakes as my final project in the programming language python.
During my research about that topic, I noticed that the growth of snowflakes is not totally understood yet, but a general idea of the growing process is developed. The one attribute the most of the snowflakes have in common is the hexagonal basis structure. The hexagonal crystal structure of ice causes that shape. There are some snowflakes with a non-hexagonal shape, like needles or triangular crystals [5], which are not taken into account in this project. Generally, snowflakes are unique. That no snowflake looks like the other is due to the fact that the growth is determined by the environment the snowflake experiences during its fall to the earth. Since every snowflake has taken a different path, the experienced environment is different and creates a different snowflake. In the laboratory, it was possible to create the same conditions for the environment such that 'twin'-snowflakes were created [3]. But since that not really happens in nature, it is reasonable to say that snowflakes are principally unique.
However, a snowflake is built when the water vapor in the air turns directly into ice without becoming liquid inbetween. Therefore, the growth of the crystal is mainly determined by the ratio of the humidity in the air and the temperature. [3]
The snowflake is in principal three dimensional, but the thickness, say in the z-direction, is for most of the snowflakes very thin. On these grounds, this simulation is only two-dimensional, but keeps track of the influence of the third dimension with a parameter. There are snowflakes which have remarkable z-components, but these are neglected in this simulation. If one is interested in more detail in the different types of snowflakes see for instance [5].

2. The Code

2.1. General Idea

The simulation code is based on the model of [2], 'A local cellular model for snow crystal growth' by Clifford A. Reiter. My code is available in the appendix. The explanation tries to follow the implementation order in the code.
The growth of snowflakes is simulated with the calculation of real values in a two-dimensional array. How the values are influenced is explained below. If the value of a site in the array is greater or equal to one, it is set to be part of the snowflake and set to be receptive. If a neighbour site of the considered site is part of the snowflake, it is set to be receptive as well. Otherwise, it is not part of the snowflake and non-receptive. This is based on the idea that

water molecules more likely connect, e.g. freeze together, if they are close to each other or not.
This simulation takes into account the ratio of temperature and humidity in the air, because it is probably the most dominant influence. Effects like noise or melting are not considered. The change of the values of the array is simulated in three different ways. First, there is a general background value of water vapor, parameter ρ in the code. This parameter is decided in the beginning and initially equal at every site in the array. Only one site in the centre, which is set equal to one, is set as an ice seed form which everything starts to grow. The vapor parameter ρ is between zero and one. The greater the value is, the more probable the growth.
Second, the value of the sites changes with respect to the equation of diffusion depending on the value of vapor in the neighbour sites. Hence, there is a diffusion parameter α. The diffusion is only happening for non-receptive sites. That is based on the physical background that the water particles in the air can freely move and the frozen water inside the snowflake cannot. At least not without the consideration of ice melting. How the diffusion works is explained later in the text.
Third, there is an additive parameter γ. γ is added on top of the value of the receptive sites, such that the mass of ice increases and the cells at the boundary of the snowflake become more likely part of the snowflake. It can be interpreted as an influence from the third dimension normal to the snow crystal surface. Even if the simulation itself does not demonstrate the behavior in three dimensions, it takes so account of the influences the three-dimensional surrounding can have. The parameter can also be set equal zero and thus one can ignore the influence of the third dimension.
Generally, the values of the receptive and non-receptive cells are saved in two different arrays, *rec* and *nonrec* respectively, which update the values of the main array *ice*.

2.2. Check for Neighbours

Because the influence of the environment is very important for the growth, the consideration of the neighbour sites of one site in the code is explained in more detail in the following.
Since the vapor and ice values are saved in a rectangular $mx \times my$ array, one has to check what are the neighbours of the site in the hexagonal lattice which also causes the simulated hexagonal growth. Therefore, the code uses six two-dimensional vectors. The values for the x and y components are collected in the lists *dx* and *dy*. The values are corresponding to the cartesian coordinate system and create a disorted and stretched hexagonal structure which will be corrected with the plot, see therefor section 2.4.3 Plotting. *dx* and *dy* are used to iterate over the indices of the array sites. For the iteration, a for-loop is used. In figure 1 the iteration is sketched. Therefore, the related values of *dx* and *dy* are added to i and j, respectively and thus the index is changed and one can scan the value of the neighbouring site.

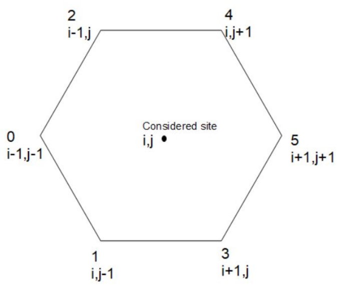


Figure 1: Neighbours

In this figure the iteration of the neighbours is sketched. The for-loop used in the code with the numbers k $\in$ {0,1,2,3,4,5}, while the i and j correspond to the considered site indices.

2.3. Diffusion

The diffusion equation is a partial differential equation which describes the motion of microparticles. It is given by $\frac{\partial u(x,t)}{\partial t} = \nabla^2 u(x,t)$. In the considered case of snowflakes, $u(x,t)$ is the value of water vapor in a cell at position x at iteration t. The following approximation for the Laplacian is based on [2] and can be derived similar like it was done for example in the lecture 7 in class.

$$\nabla^2 u(x_s,t) \approx= \frac{\alpha}{12}(-6u(x_s,t) + \sum_n u(x_n)) \tag{1}$$

x_s is here the position of the considered site and x_n the position of the corresponding neighbour n at time t (iteration t). α is the diffusion coefficient. The weight of the effect of the site itself is thus 50% and every neighbour site has a weight of $\frac{1}{12}$. The sum over the values of the neighbours gives hence the other 50% of the new vapor value. The resulting value for the next iteration t of the site with position x_s is hence

$$u(x_s,t+1) = u(x_s,t) + \frac{\alpha}{12}(-6u(x_s,t) + \sum_n u(x_n)) \tag{2}$$

2.4. Implementation

2.4.1. Set Up

In the code, one can set the parameters, the number of iterations and the size of the snowflake. Moreover, the vectors for the neighbours calculation are set with *dx* and *dy*. The arrays for saving the vapor values are for the actual snowflake *ice*, the values of the receptive sites are saved in *rec* and the non-receptive site values in *nonrec*. The values of *ice* and *nonrec* are initially set equal to ρ while *rec* is set equal zero, because one knows that all the points are

not part of the snowflake in the beginning. To actually cause a growth, the central site is set equal 1 and is thus called the ice seed.
Furthermore, there are three different functions defined for the final calculation and simulation. In the following part, they will be shortly explained.
inImage is a Boolean function. It gives the information if a considered site is inside the image or not. Therefore, it returns True if the indices i and j are in the limit between zero and the picture size mx ormy, respectively. Otherwise, the function returns False.
receptive tests if the considered site in the main array *ice* is receptive or not. Hence, it is a Boolean function as well as **inImage**. First, it inspects if the value of *ice* at the site with indices i and j is greater equal one. If not it tests if one of the neighbours has a value greater equal one. If one neighbour is part of the snowflake, the if loop breaks. In both of the listed cases, the function returns True, otherwise it returns False.
For the calculation of the diffusion, the function **average** is implemented. It calculates the value of the non-receptive site according to the diffusion rule, see equation 2. For that, it sums up all the values of the neighbour sites, if the neighbours are in the image (therefore it uses **inImage**). This value can then be used for the calculation according to the solution of the diffusion equation in equation 2. The function returns the new value for the site.

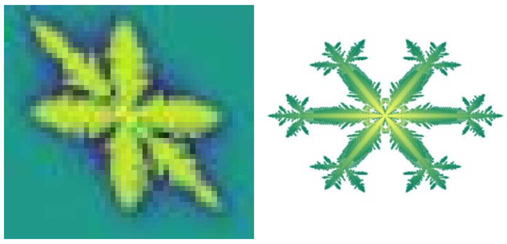
2.4.2. Computation

For the final computation of all the values of *ice* three for loops are used. The first one corresponds to the number of iterations and the two others to the position (indices) of the sites. That causes that all the points are tested and counted for the calculation. Generally, there are two main steps done for the computation of the values.
At first, **receptive** is called to check if the considered site is receptive or not.
In case the site in *ice* is receptive, the value of *rec* is changed to the value of ice plus the parameter γ, to simulate the growth of ice mass depending on the influence of the z-direction. In addition, the value of *nonrec* on this site is set equal zero.
In case the site in *ice* is non-receptive, the value of *nonrec* is updated with the value of *ice* while *rec*'s value stays zero.
Second, the new vapor value of *ice* due to diffusion is calculated. Hence, it iterates with again the for loops over all sites and uses the function **average**. The sum of that gained value and the value of *rec* from the first case calculation (that the site is part of the snowflake) is set equal to the element of *ice*. Then a new t-iteration begins.

2.4.3. Plotting

For plotting the values of a rectangular two-dimensional array in a hexagonal shape, one may turn the array with 45 degrees and stretch the distance of points according to the hexagonal shape by multiplying the rectangular array points by the basis vectors of a hexagonal lattice. Therefore, the first step is to implement an empty array, *hexagonal*, with three dimensions

$2 \times mx \times my$. To arrange now the corresponding values of the indices of a rectangular array to the hexagonal shape, the values of the hexagonal basis vectors $a_1 = (1, \frac{-1}{\sqrt{3}})$ and $a_2 = (1, \frac{1}{\sqrt{3}})$ are multiplied by the index numbers i or j in a for loop over all indices i and j of the rectangular lattice.



(a) Rectangular (b) Hexagonal

Figure 2: Rectangular and Hexagonal Plot
(a) Result without the respect to the stretching and turning due to the hexagonal lattice.
(b) Result with respect to the stretching and turning due to the hexagonal lattice.

The next step is to find all the entries of *ice* which are element of the snowflake, more precise check which sites have a value greater equal one. For that one can use the numpy function *np.where* which gives all indices of the entries of the considered array for which a implemented condition is true. These are saved in the array *indexsnow*. The condition is to check where the value of *ice* is greater equal to one. Next, an array with only the sites which are part of the snowflake is created by combining the *hexagonal* array with *indexsnow* such that the new array *snow* only contains the points which are part of the snow crystal and their position in the hexagonal lattice. The value of the ice mass is not saved, but used later for colouring the snowflake.
Finally, a figure can be plotted. This code uses scatter plot. All points of *snow* are plotted. With $plt.scatter(snow[0,:], snow[1,:])$ which are stretched from *hexagonal* and chosen by *indexsnow*. Additionally, the value of *ice* can be taken into account to have a colourful presentation of the snowflake and its mass distribution. Accordingly, it is possible to use colourmap. The value of the corresponding element in *ice* is set in to relation with the plotted site with the expression for c in the following line.

$$plt.scatter(snow[0,:], snow[1,:], c = ice[indexsnow[0], indexsnow[1]] - 1.,$$
$$s = 1., cmap =' cool'_r, marker =' H')$$

s gives the size of the points, *cmap* sets the colour corresponding to the value given by c and *marker* the shape of the points (in this case it is hexagonal). $_r$ reverses the colormap.

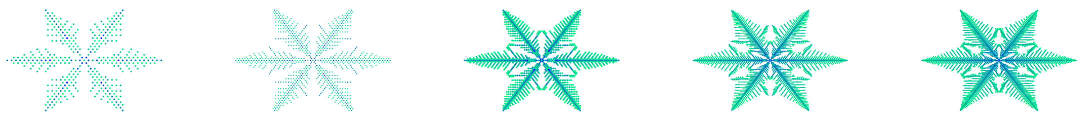
3. Results

3.1. Change of Parameter

In the appendix are three tables available. They demonstrate the change the parameters α, ρ and γ cause in the simulation. One parameter is always fixed while the others vary. The number of iterations differs from snowflake to snowflake, because the growth is diverging. The figure size is always the same and the used colormaps vary.
The dependence of α and ρ at $\gamma = 0.0001$ is demonstrated in figure 5. Figure 6 illustrates the correlation of α and γ with constant $\rho = 0.5$. In figure 7 the dependency of γ and ρ with constant $\alpha = 1.2$ is shown.
These tables are not as detailed as in [2], but qualitative comparable. One can see the similarities, which is not very surprisingly according to the fact that the coded simulation is based on the concept of [2].

Generally, one can observe, the greater the parameter γ is, the blurrier and the bigger the surface as a result gets. Moreover, the less γ is, the more delicate is the simulation. Furthermore, the comparison of the results for different α shows that the greater α, the more dominant the arms which start from the hexagonal basis structure get. The less the value of α is chosen, the more filigree results of the arm patterns appear and the overall surface of the snowflake seems to be greater. If ρ is varied, one can observe very different results. In general, the effect of α and γ dominates. ρ seems to distinct the properties of the other parameters. The iteration number must be higher the lower the value of the parameters is to achieve some acceptable result.

3.2. Change of Iteration Number - Growth and Mass



(a) 100 iterations (b) 175 iterations (c) 250 iterations (d) 425 iterations (e) 500 iterations

Figure 3: Growth due to iteration number
This figures demonstrate the growth of a snow crystal. Therefore, the values of the parameters were kept and just the number of iterations was changed. The used values for the parameters are $\alpha = 1.2$, $\gamma = 0.0001$ and $\rho = 0.6$.

Figure 3 illustrates the growth of a snowflake. Therefore, the number of iterations was changed. It is visible that the growth of the ends of the arms is faster than the growth of

the side arms. This could be interpreted as the stronger interaction with the environment for the arms. Moreover, the mass of ice increases and the structure of the snowflake gets more and more visible, the higher the iteration number is.
In nearly all of the created figures, see figure 5, 6 and 7 in the appendix, it is observable that the mass is greatest in the centre and decreases the more far the site is from the centre. Since the colormap was changed very often to give the whole report a literally colourful input, it might be a bit confusing to the mass weights.

4. Conclusion and Perspectives

The developed simulation works well, if one compares the results with different types of real snowflakes, e.g. in comparison with [3] or with the results of the simulation in [1] or [2]. But still, it does not guaranty that the results would be the same as for real snowflakes, if one actually changes the parameters in reality in a laboratory. That could be possible to compare if corresponding real physical results would be available from a laboratory.

The code could still be developed. For instance, melting could be implemented in a similar way like it was done in [1]. Moreover, it could be taken account to random noise like it is explained in [1] and [2].
Another improvement could be the implementation to break the calculation if either the limit of the figure is reached and the growth would act as it were reflected on the boundaries on the rectangular basis structure, or the number of iterations is higher than a limit number. See therefore [2]. That implementation would prevent computations with not snowflake-like results. An example of the problem of the reflection is demonstrated in figure 4.

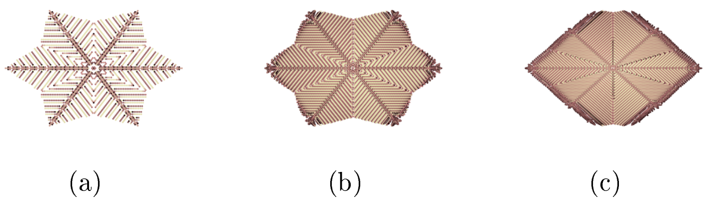


Figure 4: Change of the simulation shap due to too much iterations for speed of the growth
(a) shows a fitting amount of iterations, (b) shows an amount of iterations which is already a bit too much to keep the hexagonal shape and (c) demonstrates the transformation to the rectangular shape and the reflection of the growth due to too much iterations.

Moreover, the mass distribution of the snow crystals for different parameter could be analysed in more detail with, for instance, the same colormap and same number of iterations.
At the end, the code simulates snowflakes with a huge variety. The results are very acceptable, since they actually look like snowflakes. Unfortunately, it is not possible for me to

relate the influences of the simulation parameters to actual parameters in reality. But since, the growth of snow crystals is not fully understood yet, the simulation could be valued as successful.

5. References

[1] *http://psoup.math.wisc.edu/papers/h2l.pdf* 29.05.2019, 21:10

[2] *http://patarnott.com/pdf/SnowCrystalGrowth.pdf* 29.05.2019, 21:10

[3] *http://snowcrystals.com/* 29.05.2019, 21:10

[4] *https://www.its.caltech.edu/ atomic/snowcrystals/class/class-old.htm* 29.05.2019, 21:10

[5] *http://www.snowcrystals.com/guide/guide.html* 31.05.2019, 16:30

A. Appendix

A.1. Results of Changing the Parameters

A.1.1. Change of α and ρ with constant γ

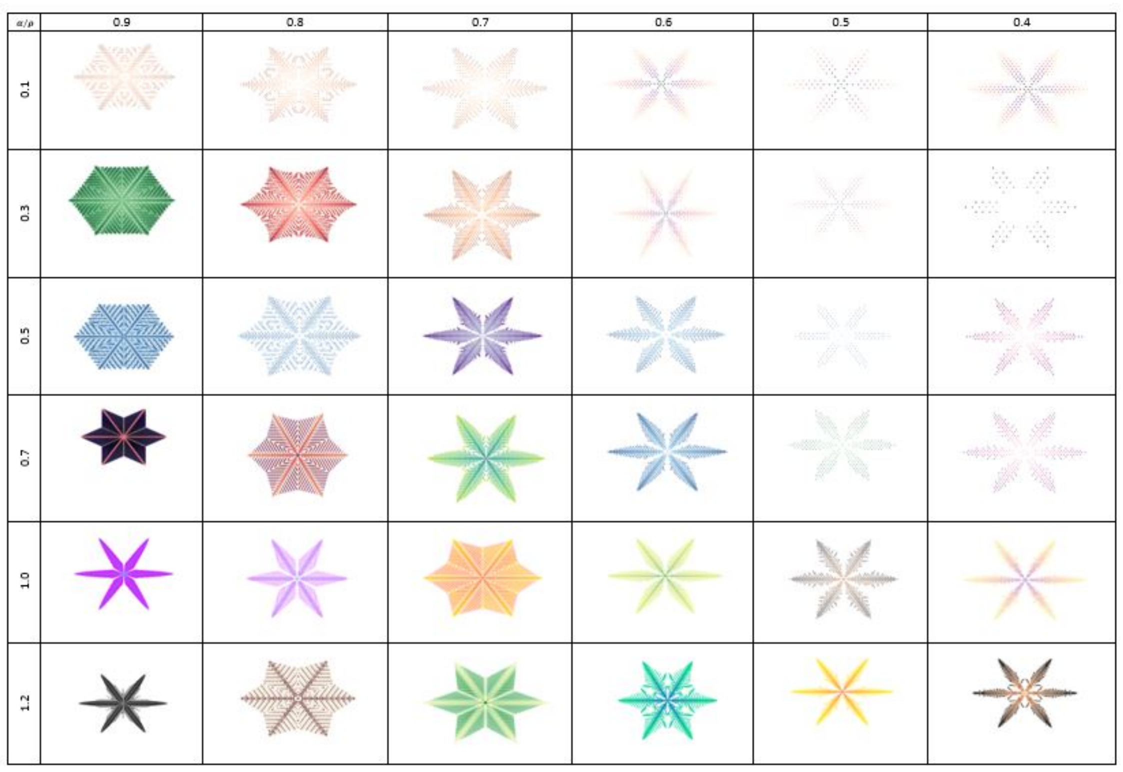


Figure 5: Change of α and ρ with constant γ

In this figure is shown how the snowflake shapes change with varying the parameters α in y-direction and ρ in the x-direction for a constant value of $\gamma = 0.0001$.

A.1.2. Change of α and γ with constant ρ

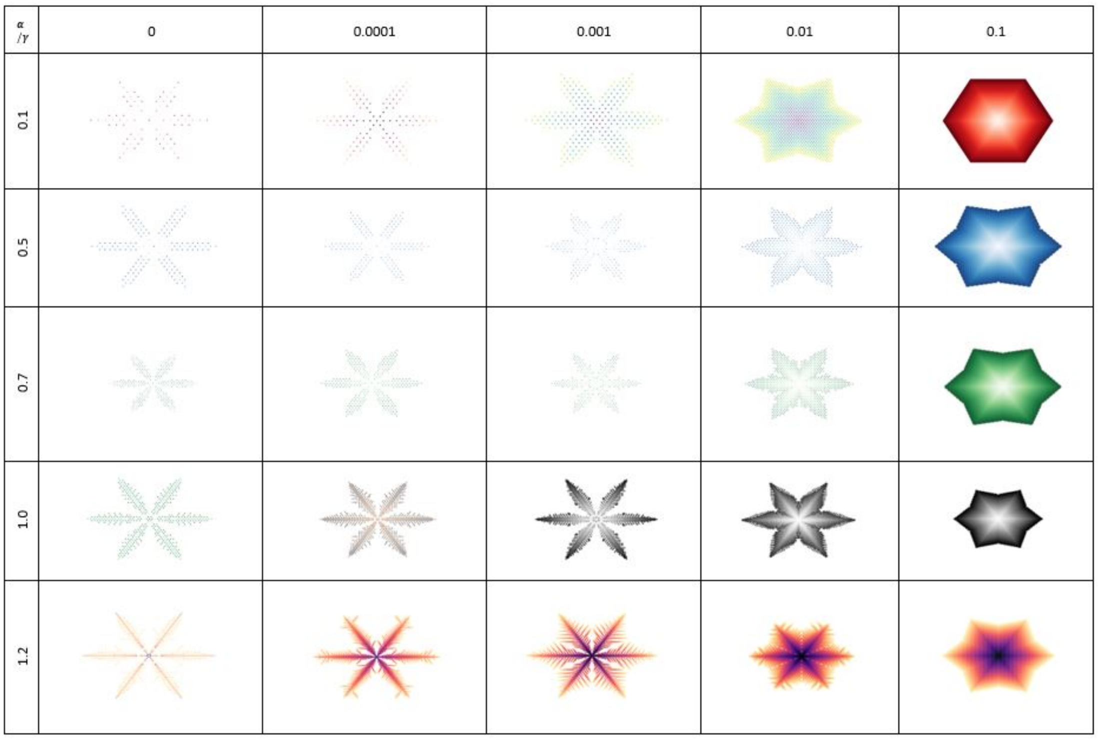


Figure 6: Change of α and γ with constant ρ

In this figure is shown how the snowflake shapes change with varying the parameters α in y-direction and γ in the x-direction for a constant value of $\rho = 0.5$.

A.1.3. Change of γ and ρ with constant α

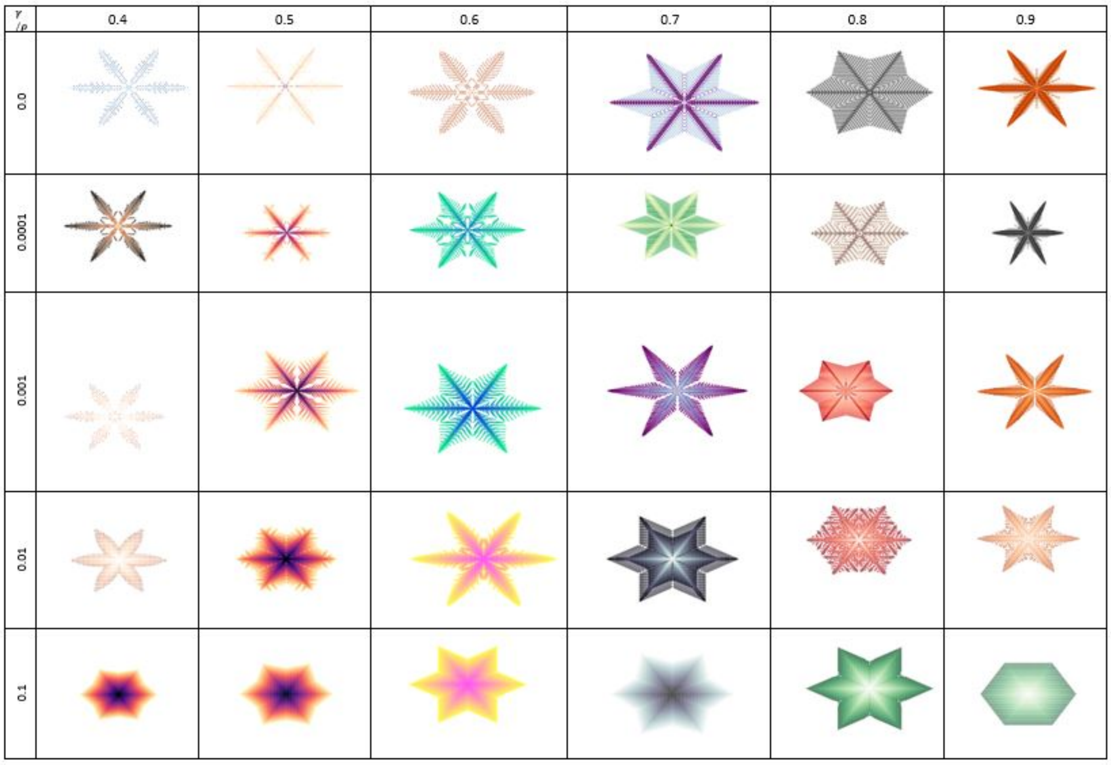


Figure 7: Change of γ and ρ with constant α
In this figure is shown how the snowflake shapes change with varying the parameters γ in y-direction and ρ in the x-direction for a constant value of $\alpha = 1.2$.

A.2. The Simulation Code of the Snowflake Growth in Python

```
# -*- coding: utf-8 -*-
"""
Created on Tue May 28 22:26:40 2019

@author: Paulina
"""

import numpy as np
import matplotlib.pyplot as plt
#import random

maxIt = 1000 #maximal iteration of growth steps
mx = 251 #width
my = 251 #and height of the snowflake arrays

rho = 0.5 #water vapor
gamma = 0.0005 #constant influence from the third dim. which can be understood
               #similar as the humidity,
               #but from the normal dimension of the snowflake
alpha = 1. #diffusion constant

#implement arrays for the ice mass value, receptive and
#non-receptive sites and set all elements equal to the inital
#humidity value rho
ice = np.empty((mx,my)); ice[:] = rho #values of ice mass
rec = np.zeros((mx,my)) #receptive site
nonrec = np.zeros((mx,my)) ; nonrec[:] = rho #non-receptive site

#initial condition: set an ice seed at the origin
x0 = int((mx-1)/2) ; y0 = int((my-1)/2)
ice[x0][y0] = 1.0

#initialize the vectors for the neighbors
dx = [-1, 0, -1, 1, 0, 1]
dy = [-1, -1, 0, 0, 1, 1]

#create function which checks if the considered site is inside the
#snowflake or not. use therefore a boolean function
def inImage(i,j,mx,my):
    inside = False
    if i >= 0 and i < mx and j >= 0 and j < my:
        inside = True
    return inside

#define boolean funuciton which checks which sites
#are receptive and which not
def receptive(i,j,dx,dy):
    recep = False
    if ice[i,j] >= 1.: #if the value of the site in ice is greater equal 1,
                        #its part of the snowflake
        recep = True
    else: #if the value of one neighbour site is greater equal 1,
            #it could be part of the snowflake
        for k in range(6): # check the 6 neighbours
            kx = i + dx[k] ; ky = j + dy[k]
            if inImage(kx,ky,mx,my):
                if ice[kx,ky] >= 1.: #check value of k-th neighbour site
                    recep = True
                    break #if one neighbour is receptive, break, bc then site
                          #is receptive as well
    return recep

#implement the average rule of the nonrec sites it simulates the
#diffusion if the water in the air according to the diffusion equ.
def average(i,j,dx,dy,alpha):
    sum_neigh_nonrec = 0. # sum of value of all neighbours of considered site
    for k in range(6):# check neighbors
        kx = dx[k] + i ; ky = dy[k] + j
        if inImage(kx,ky,mx,my): #if neighbour is inImage, add value to sum
            sum_neigh_nonrec += nonrec[kx,ky]
    summy = nonrec[i,j] + (alpha/12.) * \
            (sum_neigh_nonrec - 6.*nonrec[i,j]) #implement solution of the
                                                 #diffusion equation
    return summy

#Now we apply the functions to create a snowflake
#depending on if the considered site is part of the snowflake
#the code handles the sites as receptive or non-receptive with if conditions
#for loops over all the sites (i and j) and the number of iteration

for g in range(maxIt):
    print("step " + str(g+1) + " out of " + str(maxIt))

    for i in range(mx):
        for j in range(my):
            #check if site is part of the snowflake or not
            RECEPT_ij = receptive(i,j,dx,dy)
            if RECEPT_ij: #if part of the snowflake
                rec[i,j] = ice[i,j] + gamma #add influence from 3rd dim.
                nonrec[i,j] = 0. #set the value in nonrec zero,
                                  #bc it's part of the snowflake
            else: #if not part of snowflake, it's non-receptive:
                nonrec[i,j] = ice[i,j] #set value of ice equal nonrec

    #calculate the new value of the nonrec sites due to diffusion
    for i in range(mx):
        for j in range(my):
            new_hum_value = average(i,j,dx,dy,alpha) #value after diffusion
            ice[i,j] = rec[i,j] + new_hum_value #overwrite old ice value

#prepare plot
#therefore we have to rearrange the site distance in the plot due to the
#hexagonal lattice

#array which contains at the end hex. lattice points
hexagonal = np.empty((2,mx,my))
#hexagonal basis vectors
a1 = np.array([1., -1./np.sqrt(3.)])
a2 = np.array([1., 1./np.sqrt(3.)])
#create the position of the lattice points due to the basis vectors
for i in range(mx):
    for j in range(my):
        hexagonal[:,i,j] = int(i)*a1 + int(j)*a2

#create an array which contains only values which are part of the snowflake
#e.g., if the value is greater equal 1
#the np.where function saves the indeces of the corresponding elements in
#two tupels
index_snow = np.where(ice >= 1.)
#snow is array with all thevalues of the sites which are in the snowflake
snow = hexagonal[:,index_snow[0],index_snow[1]]
fig = plt.figure()
plt.scatter(snow[0,:], snow[1,:], c =ice[index_snow[0],\
                index_snow[1]]-1., s = 1., \
                    cmap = 'cubehelix', marker = 'H')
plt.axis("off")
plt.show()
fig.savefig('rho_' + str(rho)+ '_gamma_' + str(gamma) + \
            '_alp_' + str(alpha) + '_iter_' + str(maxIt)+\
            '_mxmy_' + str(mx) + '.png')
```