UNIVERSIY OF GRONINGEN

In collaboration with:

CALIFORNIA INSTITUTE OF TECHNOLOGY

PHYSICS BACHELOR PROJECT

A Quasi Geostrophic Model to Simulate Vortices in the Jovian Atmosphere

Author
Carmen R. Hoek

Supervisors
Prof. dr. Andrew P. Ingersoll [1]
Prof. dr. ir. Patrick R. Onck [2]

July 14, 2020

[1] California Institute of Technology
[2] University of Groningen

Abstract

Looking at the Jovian atmosphere we find several patterns of cyclones, which are evolving over time. The goal of this project is to create a new model based on the Quasi Geostrophic Equations (QGE) to describe the time evolution of the vortex patterns. In order to describe the vortex patterns, the equations must be solved for the stream function. The QGE give an equation similar to the Poisson equation with some added terms, and from this equation the stream function is found using Fourier Transforms. In order to evolve in time, the strong stability Runge Kutta 3rd order method is used. The second part of the project consists of comparing the results of the new model with the ones obtained from the Shallow Water Model (SWM). By running the simulations with equal input as for the SWM the new algorithms can give more insight in the results of SWM.

Acknowledgements

First of all, I would like to thank prof. dr. Andrew P. Ingersoll of the Planatary Science Departement at the California Institute of Technology (Caltech) for the oppertunity to do a research project at Caltech during the summer of 2019 and for all the help from the beginning to the end of this project. I would also like to thank dr. Tricia Ewald and dr. Cheng Li of Caltech for their help with the programming part of the project. Lastly I would like to thank prof. dr. ir. Patrick Onck of the University of Groningen for his help in the last stages of the project.

Contents

1 Introduction

1.1 What are we looking for?

The goal of this project is to create a new model to describe the atmosphere of Jupiter in a more convenient an faster way. We want to approximate the physical situation in a first order model which has a short computational time. There are more sophisticated models available, but at the moment we see that the computation time of these models can take up to days. The general idea of this model is that we find a method which is able to approach the physics situation with a certain level of detail while having a significantly shorter computational time.

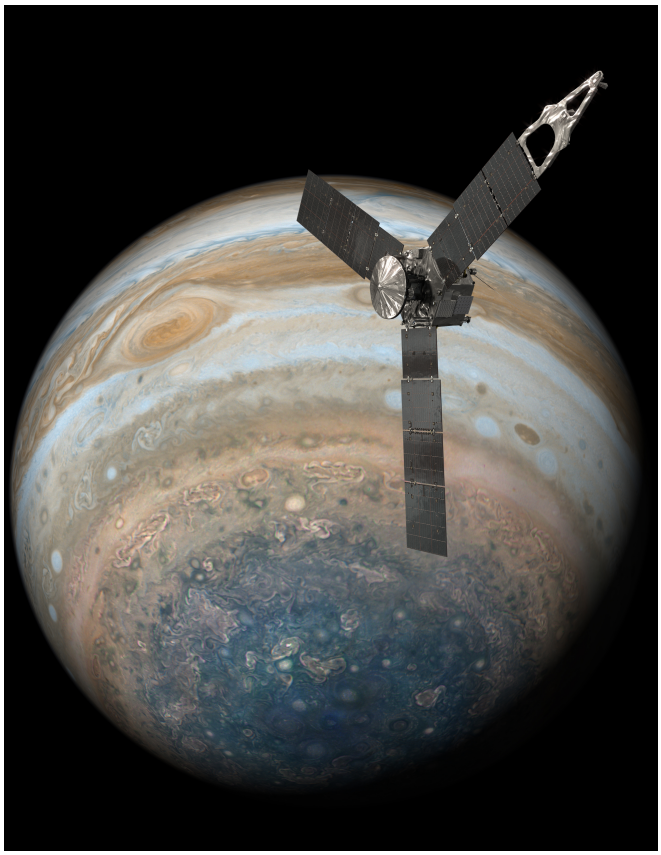

Figure 1: An illustration of Juno at the south-pole of Jupiter (1)

The method proposed is to use a quasi geostrophic model, to approximate the movement around the Polar region of Jupiter. The model will be written in Python and will be based on many already available numerical methods, mostly written in programming languages as Fortran. These methods will be converted into Python 3 code and there will be new code written to adjust to the situation we want. We will use an output in the form of 2D plots at certain time intervals and we will combine these visual outputs to movies describing the storms over a longer period. More

information about the way the model works and what kind of input we will use can be found in the chapter describing the code.

The region we want to describe in this model is the polar regions of the planet, more specifically the regions till about 70 degrees away from the geographical pole of Jupiter. We take an interest in this region because we find interesting storm patterns consisting of multiple cyclonic and anti-cyclonic movements which live way longer than we would expect. These storms are mainly studied using data collected by the Juno Space Mission, on which is elaborated in the next section.

1.2 Juno Space Mission

The Juno Space Mission is a NASA mission aiming to discover more about the biggest planet in our Solar System. The space craft was launched on August 5, 2011, as part of the New Frontiers program, which also contained the New Horizon Mission to Pluto and Beyond, and OSIRIS-REx, a spacecraft aiming to visit an astroid and collecting samples to bring back to Earth. Juno was the second mission in the program and was built by Lockheed Martin and operated by NASA's Jet Propulsion Laboratory managed by the California Institute of Technology (2).

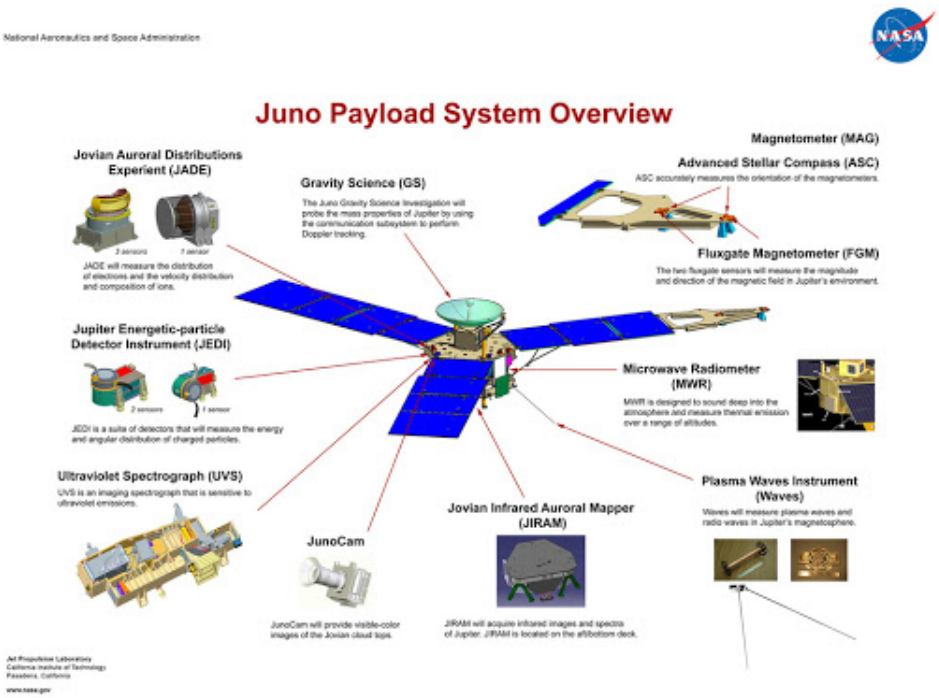

Figure 2: An overview of the instruments on Juno (3)

Juno reached the Jovian System on July 5, 2016 and started the scientific exploration of the planet. The main instruments on Juno are the microwave radiometer (MWR), the Jovian Infrared Auroral Mapper (JIRAM), the Magnetometer (MAG), gravity science (GRAV), the Jovian Auroral Distribution Experiment (JADE), the Jovian Energetic Particle Detector Instrument (JEDI), the Radio and Plasma Wave Sensor (Waves), the Ultraviolet Imaging Spectrograph (UVS) and last but not least the JunoCam (4).

The goals of the Juno mission are mainly to measure the gravity field, magnetic field and polar magnetosphere. It will also provide research data to find out how the planet is formed, how the core looks like and many more interesting things (5). Juno is the second spacecraft visiting Jupiter over a longer period. Before Juno, we had the Galileo Orbiter, which orbited Jupiter in the period between 1995 and 2003 (6).

Most probes send out to the outer planets of our Solar System are nuclear powered. This because at these distances the sun is very far away and is in most cases not enough to power the space craft. Juno is different, it works with three solar panels.

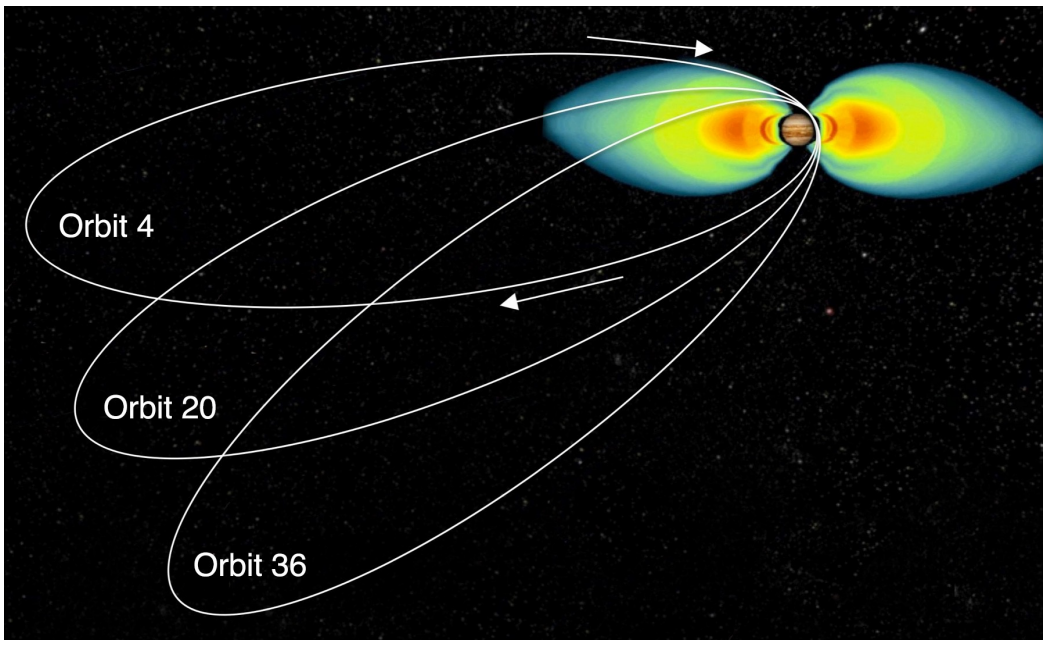

Figure 3: Juno's trajectory is elliptical in order to protect the instruments against the magnetic field of Jupiter. (7)

Unlike satellites which orbit the Earth in a circular orbit, Juno has a very elliptic orbit around Jupiter (8). This because the charged particles around the planet will fry Juno's instruments, so Juno is doing a polar orbit to avoid these regions. The orbits of Juno are shown in figure 2.

2 Background

2.1 The Quasi Geostrophic Model

The whole goal of this project is to find a method to describe the Jovian atmosphere and model the storms within the atmosphere. We can use a quasi geostrophic approach to create a programmable model (9) (10). In order to describe the system it is important to make some assumptions which make it possible to describe the situation with a simple model.

The first assumption is that the fluid is in-compressible, which means that the density (ρ) is constant. This is a reasonable assumption since the density does not significantly change compared to the scale of the planet and so for a first order approximation it is okay to work under the impression that the fluid which we are describing, in this case the Jovian atmosphere is in-compressible (11) (12).

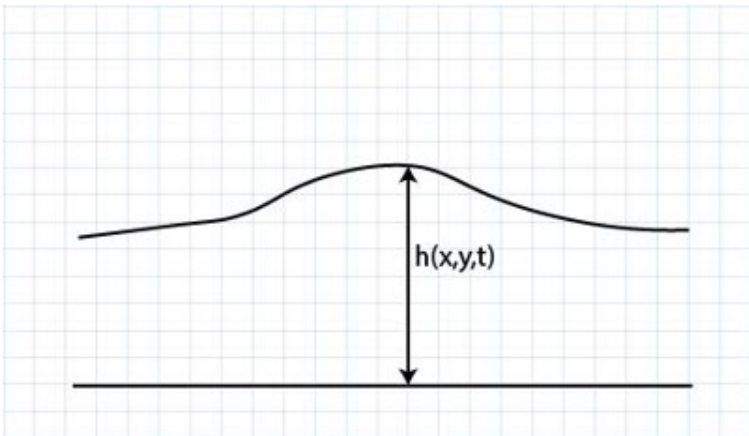

Figure 4: The 1 layer model

The next assumption is that the upper surface of the fluid is at $z = h(x, y, t)$ (see figure 3) and that the velocity is not dependent on this function, but only on (x, y, t)(13) (14). This assumption is a bit more ambiguous since we do not know how the atmosphere looks below the first layers. It is for now impossible to know how the fluids move blow these layers, we do not know if there is an up or down stream of material (15). But this model is supposed to be a first approximation of a very complex physical situation and so in this first approximation we only look at the most upper part of the atmosphere and in this region the up and down streams can be neglected and we can assume a flat layer below the layer we model (see figure 4) since the changes are very minor compared to the whole layer (16) (17).

For future research into this topic and the creation of more complicated models it is certainly a thing to look at and it will definitely improve the model, but the main goal of this model is a fast approximation of the real situation and adding extra layers will not improve the model much if you consider the extra computational time need to carry out the extra rounds of calculations.

For an in-compressible fluid the conservation of mass can be written as:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = \vec{\nabla} \cdot \vec{v} = 0 \tag{1}$$

Integrating this equation from the bottom of the system ($w = 0$), to the surface of the system ($w = \frac{Dz}{Dt} = \frac{Dh}{Dt}$ where $\frac{D}{Dt} = \frac{\partial}{\partial t} + \vec{v} \cdot \vec{\nabla}$ describes the material derivative) results in the following equation:

$$0 = \frac{Dh}{Dt} + h(\vec{\nabla} \cdot \vec{v}) \tag{2}$$

Assuming that the pressure is hydrostatic, the pressure gradient term in the horizontal momentum equation can be written as $-g\vec{\nabla}h$ resulting in the following equation:

$$\frac{D\vec{v}}{Dt} = -f\hat{k} \times \vec{v} - g\vec{\nabla}h \tag{3}$$

where f is the Coriolis term given by $f = 2\Omega \sin\phi$. Taking the curl of the momentum equation and defining the vertical component of the curl as $\zeta = \hat{k} \cdot (\vec{\nabla} \times \vec{v})$ results in:

$$\frac{D(\zeta + f)}{Dt} + (\zeta + f)\vec{\nabla} \cdot \vec{v} = 0 \tag{4}$$

Combining this with the mass conservation equation results in:

$$\frac{DQ}{Dt} = 0, Q = \frac{(\zeta + f)}{h} \tag{5}$$

Using the definition of ζ, assuming there are no derivatives in the vertical direction and v and u being given by the derivatives of the stream function (Ψ) there can also be written:

$$\zeta = \frac{\partial^2 \Psi}{\partial^2 x} - \frac{\partial^2 \Psi}{\partial^2 y} = \nabla^2 \Psi \tag{6}$$

The Coriolis parameter around the poles can be defined as:

$$f = f_0\left[1 - \frac{r^2}{2R^2}\right] \tag{7}$$

where $f_0 = 2\Omega$, with Ω being the rotation of Jupiter, $r^2 = x^2 + y^2$ and R is the radius of Jupiter. β is defined as $\frac{f_0}{2R^2}$ in order to simplify notation. Using the given definition, the following can be deduced:

$$\frac{\zeta + f}{H} = \frac{f_0 + \nabla^2\Psi + \beta r^2}{H} \tag{8}$$

H can be expanded as $H = H_0 + H'$, where $H' = \frac{f_0\Psi}{g}$, resulting in, using the binomial approximation:

$$\frac{\zeta + f}{H} = \frac{f_0}{H_0}\left[1 + \frac{\zeta}{f_0} - \frac{\beta r^2}{f_0}\right]\left[1 - \frac{f_0\Psi}{H_0 g}\right] \tag{9}$$

Ignoring the smallest terms, this can be written as:

$$\frac{\zeta + f}{H} = \left[f_0 + \nabla^2\Psi - \frac{\Psi}{L_D^2} - \beta r^2\right]\frac{1}{H_0} \tag{10}$$

Taking the material derivative and setting it to zero as a result of conservation results in:

$$\frac{\partial}{\partial t}\left[\nabla^2\Psi - \frac{\Psi}{L_D^2} - \beta r^2\right] = J(\Psi, q - \beta r^2) \tag{11}$$

which is defined as the Quasi Geostrophic Equation.

Now we can use this equation to make a model of the Jovian atmosphere. We can come up with a code which is able to calculate the Jacobian and Laplacian of the original wave function grid and use a method to step forward in time. The methods are discussed in the next chapter and a more qualitative description of the code is given in the chapter about the code.

2.2 Courant Friedrichs Lewy Condition

In order to determine the time step of a simulation we need to look at a specific condition called the Courant Friedrichs Lewy Condition (CFL-condition) named after Richard Courant, Kurt Friedrichs, and Hans Lewy, who first mentioned it in their paper in 1928 (18). This condition has

a mathematical basis and is the necessary convergence condition while solving partial differential equations. It comes forward in the numerical analysis of direct time integration schemes, which are in this project used to move the system forward in time.

As a consequence of this condition, the time step you take to move the system forward must be less than the time calculated using the CFL-condition. The CFL-condition is defined in one and two dimensions. Since this simulation is two dimensional, so we will define the condition in two dimensions. The condition is defined as:

$$C = \frac{u_x, \Delta t}{\Delta x} + \frac{u_y, \Delta t}{\Delta y} \leq C_{max} \tag{12}$$

In this equation C is defined as the Courant number, u is the magnitude of the velocity, Δt is the time step and Δx, Δy are the grid spacing of the simulation.

The maximal value for the Courant number depends on the method to solve for time. In the case of an explicit method, as used in this project the maximal value is 1. Equation 12 can be solved for the time step, which results in:

$$\Delta t \leq \frac{1}{\frac{u_x}{\Delta x} + \frac{u_y}{\Delta y}} \tag{13}$$

The following conditions can be imposed:

1. $u_{xmax} = u_{ymax} = u$
2. $\Delta x = \Delta y = d$

With these conditions equation 13 can be written as:

$$\Delta t \leq \frac{d}{2u} \tag{14}$$

3 Numerical Methods

In this chapter we will discuss the numerical methods used in the code. First we will look at the Fourier Transform, how we can do this numerically, and how does it help us to solve the problem. Then we are going to look how to calculate the Jacobian and finally we will look at the time integration method. The choice was made to use strong preserving Runge-Kutta third order, and we will discuss how to implement it in the code.

3.1 Fourier Transform

3.1.1 Introduction

The Fourier Transform is the result of the study of the Fourier Series. These series make it possible to write any periodic function as a sum of a combination of $\sin$ and $\cos$ terms. The Fourier Transform is a result of this method where the period is extended and allowed to approach infinity resulting in an integral term (19).

If we have a Fourier Series, it is possible to recover the amplitude of the original wave due to the properties of the $\sin$ and the $\cos$. In many cases, it is useful to write the Fourier Series using exponentials instead of a combination of $\cos$ and $\sin$ and we can rewrite this using Euler's Formula as stated in formula 15.

$$e^{2\pi i\phi} = \cos 2\pi\phi + i \sin 2\pi\phi \tag{15}$$

If we write the Fourier Series in terms of complex exponentials it is needed that the coefficients of the Fourier Transform are also complex. These complex values are often seen as the amplitude of the wave as well as the phase of the wave. This will become more clear in section 3.1.3 (20).

The best way to go from the Fourier Series to the Fourier Transform is looking at functions which are zero, except for a small domain. In this case we can calculate the Fourier Series for each domain of such a function, and when we increase the length of the domain we are going more and more towards a more general case.

3.1.2 Definition

The definition of the Fourier Transform is actually not that hard, and there are several ways to write it down. In this section I will go with the traditional mathematical way of writing, but in

other fields like electrical engineering it is more conventional to change the sign (21). The Fourier Transform is nothing more than the integral of the entire real domain, so from minus infinity till infinity of the original function times an exponential and is almost always indicated by a hat. The definition is given in formula 16 (22).

$$\hat{f}(\epsilon) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i x \epsilon} dx \tag{16}$$

It is now easy to see what the inverse Fourier Transform must be, since this transform needs to obtain the original function back, so the inverse Fourier Transform is given in formula 17 (23).

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\epsilon) e^{-2\pi i x \epsilon} d\epsilon \tag{17}$$

3.1.3 Example

In order to make it a little bit more understandable let us look at the visuals of the Fourier Transform. Suppose we have a function, like the one given in formula 18. If we plot this function on an interval from -3 till 3 we obtain the graph as shown in figure 5.

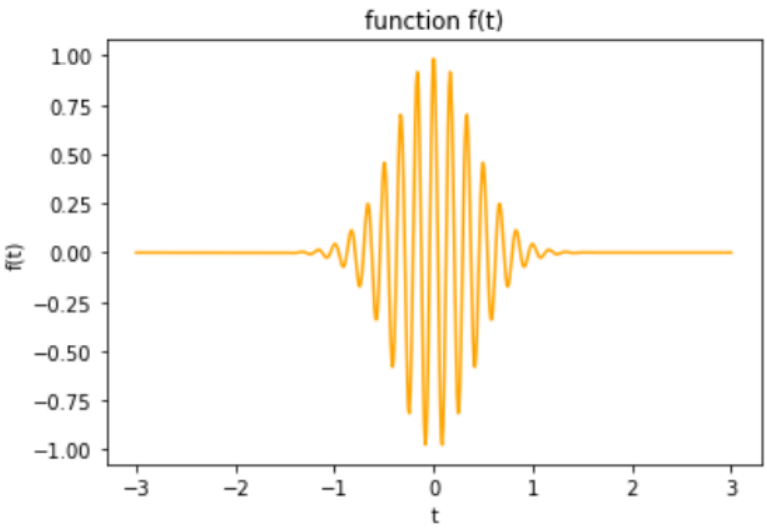

Figure 5: A function f(t)

$$f(t) = \cos 12\pi t e^{-\pi t^2} \tag{18}$$

If we take the Fourier Transform of formula 18, we get the formula given in equation 19. We can plot the real and the imaginary part separately, resulting in the graphs given in figure 6 and 7. As seen in these figures, the magnitude of the real part is much larger than the magnitude of the imaginary part.

$$\hat{f}(w) = \frac{1}{2}e^{-\pi(w+6)^2}(e^{24\pi w} + 1) \tag{19}$$

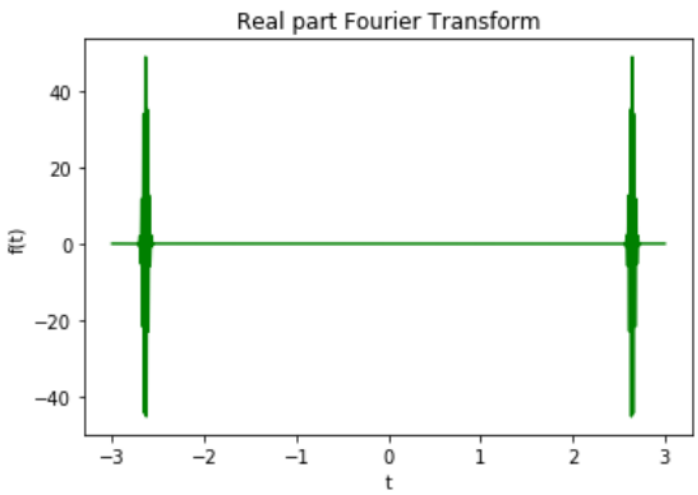

Figure 6: The real part of the Fourier Transform

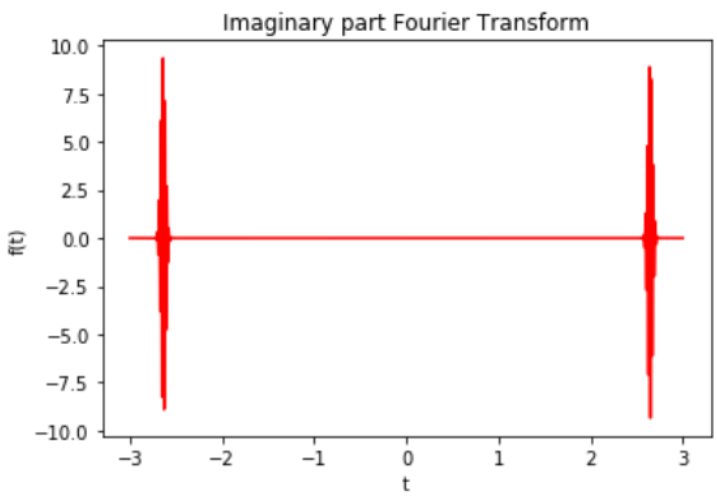

Figure 7: The imaginary part of the Fourier Transform

3.2 Fast Fourier Transform

The Fast Fourier Transform is an algorithm that computes the Discrete Fourier Transform of an array of numbers. The Discrete Fourier Transform is a discretized form of the Fourier Transform in order to be able to calculate the Fourier Transform numerically (24) (25). The algorithm makes it possible to do the Fourier Transform of large data sets much faster, which is help full since the data size of this project is enormous.

There are lots of different methods to do a Fast Fourier Transform, but the one which is most widly used is the Cooley-Tukey algorithm (26). The general idea of the method when applied to an array of length $N = N_1 N_2$ consists of three steps:

1. Preform N_1 Discrete Fourier Transform of size N_2
2. Multiply the result by twiddle factors (complex roots of unity)
3. Preform N_2 Discrete Fourier Transform of size N_1

A visual overview of the method is given in figure 8.

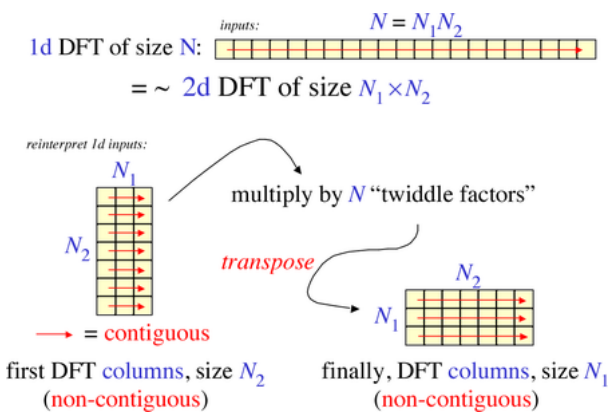

Figure 8: A visual overview of the Cooley-Tukey algorithm (27)

3.2.1 Multi-Dimensional

Everything discussed till now about the Fourier Transform was only one dimensional. Of course we can extend this till more dimensions, and we find that is results in a very simple expansion of the one dimensional case. The Fourier Transform in multiple dimensions is given by:

$$F(w_1, w_2, ..., w_n) = \Sigma_{t_1=-\infty}^{\infty} \Sigma_{t_2=-\infty}^{\infty} ... \Sigma_{t_n=-\infty}^{\infty} f(t_1, t_2, ..., t_n) e^{-iw_1 t_1 - iw_2 t_2 ... - iw_n t_n} \tag{20}$$

3.2.2 Numpy-Method

As discussed before, there are many ways to define the Fourier Transform. The numpy method used in this project uses the following definition:

$$A_k = \sum_{m=0}^{n-1} a_m \exp\left\{-2\pi i \frac{mk}{n}\right\} \qquad k = 0, \ldots, n-1 \tag{21}$$

The direct Fourier Transform is defined for complex and real inputs and outputs. A single frequency output will be represented by:

$$a_m = \exp\{2\pi i\, f m \Delta t\} \tag{22}$$

In this case Δt gives the sampling interval.

The results follow the "standard" order, this means that if we define $A = fft(a, n)$ we get $A[0]$ is the total sum of the signal, also called the zero-frequency. This number is always purely real for real inputs. The next terms contain the rest of the frequency terms, alternating between the positive and negative frequencies with the absolute value of the term always growing.

There are a couple of different algorithms within the numpy package. The routine np.fft.fftfreq(n) returns an array giving the frequencies of corresponding elements in the output. The routine np.fft.fftshift(A) shifts transforms and their frequencies to put the zero-frequency components in the middle, and np.fft.ifftshift(A) undoes that shift.

In the case where the input of the Fourier Transform is a time-signal domain the Fourier Transform will result in its amplitude-, power- and phases spectrum, using np.abs(A), np.abs(A)**2 and np.angle(A), respectively.

Within the numpy package we can divine the inverse DFT as:

$$a_m = \frac{1}{n} \sum_{k=0}^{n-1} A_k \exp\left\{2\pi i \frac{mk}{n}\right\} \qquad m = 0, \ldots, n-1. \tag{23}$$

It differs from the forward transform by the sign of the exponential argument and the default normalization by 1/n.

3.3 Application to the Project

When solving the equation found in section 2 in time, for every time step there is found a new value for q. From this value it is necessary to find the stream function and the Laplacian of the stream function in order to calculate the Jacobians at the new time. This equation is essentially a second order differential equation in 2 dimensions and can be solved using an adapted Poisson solver.

The first step of the algorithm is to use the 2 dimensional Fourier Transform to transform the matrix of values to Fourier Space. The next step is to divide all entries of the matrix by M, where M is defined as a value based on the point in Fourier space as derived below. From here the only thing left is the inverse 2 dimensional Fourier Transform in order to go back to real space.

In order to find M, it is important to look how the Laplacian is described in Fourier Space. Using the definition of Fourier Series the following can be written:

$$Q(x,y) = \Sigma_{k=0}^{N}\Sigma_{m=0}^{N}\hat{Q}(k,m)e^{ikx+imy} \tag{24}$$

Taking a Laplacian of this function results in:

$$\nabla^2 Q(x,y) = \Sigma_{k=0}^{N}\Sigma_{m=0}^{N}(k^2+m^2)\hat{Q}(k,m)e^{ikx+imy} \tag{25}$$

Which can be written for each component in Fourier Space as:

$$\nabla^2 \hat{Q}_{km} = (k^2+m^2)\hat{Q}_{km} \tag{26}$$

from which can be deduced that:

$$\hat{Q}_{km} = \frac{1}{k^2+m^2}\nabla^2\hat{Q}_{km} \tag{27}$$

Since q consists of an addition term, it is necessary to add an extra term to the multiplication factor in the following way:

$$\hat{\Psi}_{km} = \frac{1}{k^2+m^2-\frac{1}{L_D^2}}(\nabla^2\hat{\Psi}_{km} - \frac{1}{L_D^2}\hat{\Psi}_{km}) \tag{28}$$

3.4 Jacobian Calculations

One of the major problems within this project is the calculation of the Jacobian side of equation 11. It has been found that there is an instability when using the normal Jacobian Format for the long term numerical integration of the equations of fluid motion (28). The instability is the result of the formation of eddies at a few grid intervals (29).

Based on the constraints described in (29) a new method is developed. We have four basic second order finite difference analogues for a square grid:

$$\begin{aligned}\mathbb{J}^{++}_{i,j} = \frac{1}{4d^2}[&(\zeta_{i+1,j} - \zeta_{i-1,j}) \\ &(\psi_{i,j+1} - \psi_{i,j-1}) - \\ &(\zeta_{i,j+1} - \zeta_{i,j-1}) \\ &(\psi_{i+1,j} - \psi_{i+1,j})]\end{aligned} \tag{29}$$

$$\begin{aligned}\mathbb{J}^{+\times}_{i,j} = \frac{1}{4d^2}[&\zeta_{i+1,j}(\psi_{i+1,j+1} - \psi_{i+1,j-1}) - \\ &\zeta_{i-1,j}(\psi_{i-1,j+1} - \psi_{i-1,j-1}) - \\ &\zeta_{i,j+1}(\psi_{i+1,j+1} - \psi_{i-1,j+1}) + \\ &\zeta_{i,j-1}(\psi_{i+1,j+1} - \psi_{i-1,j-1})]\end{aligned} \tag{30}$$

$$\begin{aligned}\mathbb{J}^{\times+}_{i,j} = \frac{1}{4d^2}[&\zeta_{i+1,j+1}(\psi_{i,j+1} - \psi_{i+1,j}) - \\ &\zeta_{i-1,j-1}(\psi_{i-1,j} - \psi_{i,j-1}) - \\ &\zeta_{i-1,j+1}(\psi_{i,j+1} - \psi_{i-1,j}) + \\ &\zeta_{i+1,j-1}(\psi_{i+1,j} - \psi_{i,j-1})]\end{aligned} \tag{31}$$

$$\begin{aligned}\mathbb{J}^{\times\times}_{i,j} = \frac{1}{8d^2}[&(\zeta_{i+1,j+1} - \zeta_{i-1,j-1}) \\ &(\psi_{i-1,j+1} - \psi_{i+1,j-1}) - \\ &(\zeta_{i-1,j+1} - \zeta_{i+1,j-1}) \\ &(\psi_{i+1,j+1} - \psi_{i-1,j-1})]\end{aligned} \tag{32}$$

In these equations i is the finite difference grid index in x, j is the index in y and d is the grid

interval.
All four of these methods to calculate the Jacobian maintain the constraints discussed in the paper by Arakawa (29) and all have the same order of accuracy. More general Jacobian methods can be developed by taking linear combinations of the four different methods. We take now:

$$\mathbb{J}_{i,j}(\zeta,\psi) = \alpha\mathbb{J}^{++}_{i,j}(\zeta,\psi) + \beta\mathbb{J}^{+\times}_{i,j}(\zeta,\psi) + \gamma\mathbb{J}^{\times+}_{i,j}(\zeta,\psi) + \delta\mathbb{J}^{\times\times}_{i,j}(\zeta,\psi) \tag{33}$$

where $\alpha + \beta + \gamma + \delta = 1$.

The method which satisfies both the conservation of energy and the conservation of square vorticity is has the following coefficients:

$$\alpha = \beta = \gamma = \frac{1}{3}, \;\; \delta = 0 \tag{34}$$

Only the linear combination $[\mathbb{J}^{++}_{i,j}(\zeta,\psi) + \mathbb{J}^{+\times}_{i,j}(\zeta,\psi) + \mathbb{J}^{\times+}_{i,j}(\zeta,\psi)]/3$ will satisfy $\mathbb{J}(\zeta,\psi) = -\mathbb{J}(\zeta,\psi)$ and also conserve both the quadratic quantities. We obtain the following equation to do the Jacobian calculations:

$$\begin{aligned}
\mathbb{J}_{i,j}(\zeta,\psi) = -\frac{1}{12d^2}\,[&(\psi_{i,j-1} + \psi_{i+1,j-1} - \psi_{i,j+1} - \psi_{i+1,j+1}) \\
&(\zeta_{i+1,j} + \zeta_{i,j}) - (\psi_{i-1,j-1} + \psi_{i,j-1} - \psi_{i-1,j+1} - \psi_{i,j+1}) \\
&(\zeta_{i,j} + \zeta_{i-1,j}) + (\psi_{i+1,j} + \psi_{i+1,j+1} - \psi_{i-1,j} - \psi_{i-1,j+1}) \\
&(\zeta_{i,j+1} + \zeta_{i,j}) - (\psi_{i+1,j-1} + \psi_{i+1,j} - \psi_{i-1,j-1} - \psi_{i-1,j}) \\
&(\zeta_{i,j} + \zeta_{i,j-1}) + (\psi_{i+1,j} - \psi_{i,j+1})(\zeta_{i+1,j+1} + \zeta_{i,j}) \\
&- (\psi_{i,j-1} - \psi_{i-1,j})\,(\zeta_{i,j} + \zeta_{i-1,j-1}) + (\psi_{i,j+1} - \psi_{i-1,j}) \\
&(\zeta_{i-1,j+1} + \zeta_{i,j}) - (\psi_{i+1,j} - \psi_{i,j-1})\,(\zeta_{i,j} + \zeta_{i+1,j-1})].
\end{aligned}$$

This equation corresponds to the advective form and the flux form of the Jacobian (29).

3.5 Time Integration Method

In order to evolve the model forward it is needed to do a time integration. In this project there is made use of third order Strong Stability Preserving Runge Kutta (30). The method makes use of:

$$\mathcal{F}[f,t] = f + \Delta t\mathcal{L}[f,t] \tag{35}$$

which is also known as the first Euler update.

We use the following steps to go forward in time:

$$f^{(1)} = \mathcal{F}[f^n] \tag{36}$$

$$f^{(2)} = \frac{3}{4}f^n + \frac{1}{4}\mathcal{F}[f^{(1)}] \tag{37}$$

$$f^{n+1} = \frac{1}{3}f^n + \frac{2}{3}\mathcal{F}[f^{(2)}] \tag{38}$$

where $\mathcal{L}$ is the right hand side of the time differential equation, so in the case of this project, it is the Jacobian.

4 Methods and Formalism

This chapter describes the code which is used to run the simulations. First we will discuss the initial values, then the simulation itself, and lastly the output of the program.

4.1 Transformations and Initialization

In the transformations and initialization part of the code we set up the environment to run our simulations. We start by defining a grid and the calculation of the coordinates. We do that using some short functions to convert to pixels to Cartesian coordinates and to convert to latitude and longitude and back. The code is listed in chapter 8.1.

Next there are some functions to define the initial vortex patterns on the planet, which will be the starting point of the stimulation. First we have a function which sets up the background of the pole using the Coriolis force as described in chapter 2.1. Then the vortex pattern has to be set up in this background. There are two different models to set up the vortices. There is a Gaussian profile which is defined by:

$$-V_0 \frac{r}{R_0} \exp\left(\frac{1}{2} - \frac{r^2}{2R_0^2}\right) \tag{39}$$

In this equation we have V_0 defines as the initial velocity of the vortex, which is set to 80 m/s during the simulation. We have chosen this value to be able to compare the simulations to other projects. The other constant in this equation, R_0 represents the radius of Jupiter, which is used to adapt to different coordinates on the pole. The other profile is an exponential profile which is defined by:

$$-V_0(r + R_0) \exp\left(1 - \frac{r}{R_0}\right) - \frac{V_0^2}{4R_0 f_0}(2r + R_0) \exp\left(2 - \frac{2r}{R_0}\right) \tag{40}$$

The only new introduced constant in this profile is the Coriolis parameter which is denoted by f_0 and is defined to be the $3.48 \cdot 10^4$ s^{-1}.

The output from this code is saved into a npy file to save memory and be able to run different simulations at the same time. The initial vortex profile can be plotted, which can be found in figure 9.

The background of the profile which is defined by the Coriolis force is shown in figure 10.

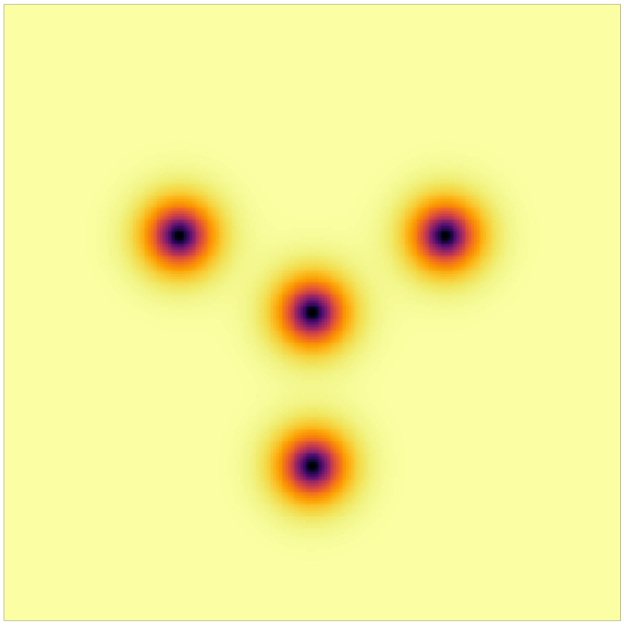

Figure 9: The vortex profile set up using the code listed in chapter 8.1.

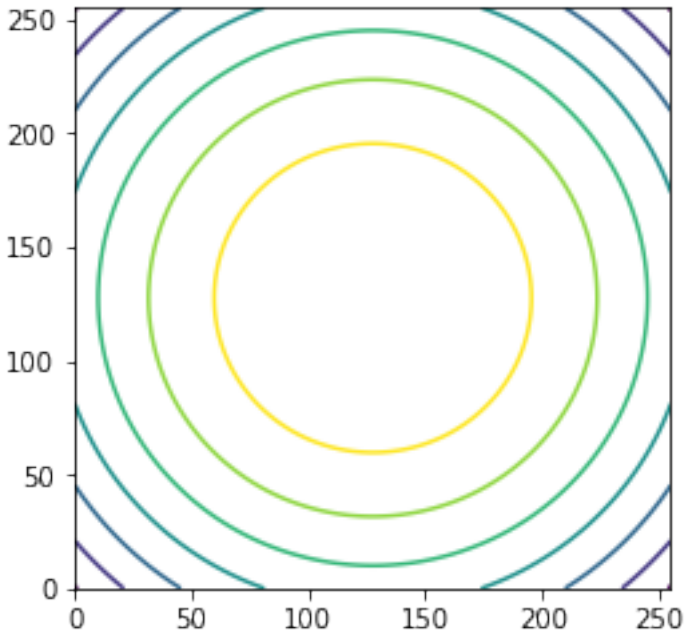

Figure 10: The background defined by the Coriolis force. The axis labels are the pixels, and they do not have a physical significance.

4.2 Simulation

The simulation code is based on a couple of simple functions and can be found in chapter 8.2. First there is need for a function which can convert the latitude and longitude to Cartesian coordinates. This function is equivalent to the one mentioned in the previous chapter and is only used to make sure that the input can be read.

The next functions are needed for the simulation to preform Fourier Transforms. This code is based on the method described in chapters 3.1, 3.2 and 3.3. The code takes the Laplacian of the stream function as input and transforms it into the stream-function itself which can be used for further calculations. The second function is a helper function for the Fourier function.

The next thing needed is the function to calculate the Jacobian. This function is using the method described in chapter 3.4 and is based on an older Fortran code available within the research group. The main improvement of the python function is the computational time and the compatibility with the rest of the program.

The last calculating function is the time-step. This function is based on the algorithm described in chapter 3.5. The function is also a combining function and has to recalculate every value after every small time step.

Because of the many calculations which have to be preformed the choice was made to use a third order time approximation instead of a fourth order because this will add significantly to the computational time, where the accuracy of the results are not much improved. This choice can be justified because the goal of this project is to improve the computational time by using an approximation of the more advanced methods.

The remaining function within the simulation code is the function which preforms the simulation itself. This function is a recursion algorithm which preforms the calculations using the previous described function. The algorithm is found in chapter 8.3.

4.3 Output

The output of the recursion algorithm is in principle very simple. It will return portable network graphics, and save them in the folder where the algorithm is running. But in order to able to say something about the physical process the output needs to be processed. The processing of the images is done using After Effects, part of Adobe creative studio (31).

Essentially all images are stacked together to create a small movie. Next to the movie application of the images, they are also used to given an overview of what is happening in the simulation. The average output of the simulation consists of about 1800 image at different times.

5 Results

In this chapter we will discuss the results of the previous described simulation algorithm. All simulations are done at a 256 $\times$ 256 grid, the plots are made around the pole towards 70 $^\circ$ outwards. The timestep is determined by the Courand Friedrichs Lewy condition as described in chapter 2.2. The simulation is done with a couple of different parameters. In table 1 the simulations are shown with their different parameters.

Simulation number	number of vortices	polar vortex present	longitude	time (years)
1	6	no	85°	3
2	5	no	80°	3
3	6	yes	80°	3
4	3	no	80°	3

Table 1: Some information about the simulations

5.1 Simulation 1

As described in the table above this simulation consist of 5 vortices, there is no polar vortex present and they are set at 85 $^\circ$ and the simulation was done for 3 years. The plots at an interval of 4 months can be found in figure 11. A movie, containing all plots can be found at (32).

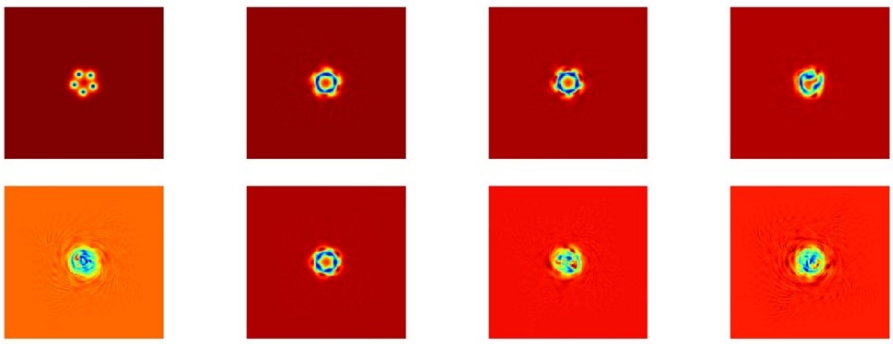

Figure 11: Plots of simulation 1 at 4 months time interval.

Looking at the plots in figure 11 we can see that the vortices are unstable. They merge together within the first year and form a really unstable region on the pole.

5.2 Simulation 2

As described in the table above this simulation consist of 5 vortices, there is no polar vortex present and they are set at 80 ° and the simulation was done for 3 years. The plots at an interval of 4 months can be found in figure 12. A movie, containing all plots can be found at (32).

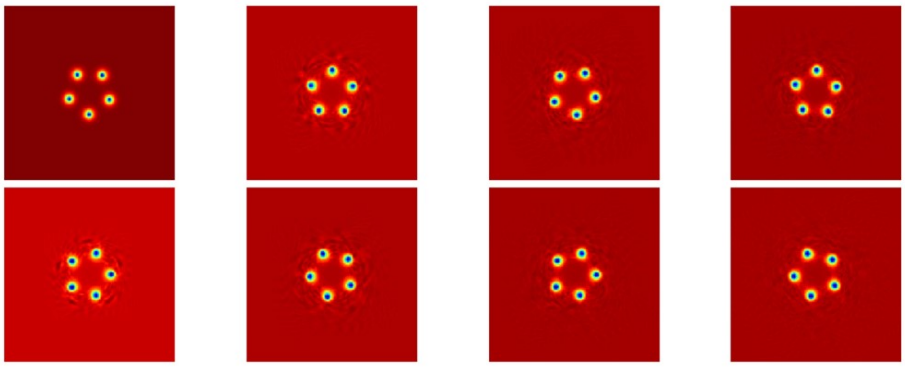

Figure 12: Plots of simulation 2 at 4 months time interval.

Looking at the plots in figure 12 we can see that the vortices are stable. They have still the same distance to the pole after 3 year and they still have the same intensity.

5.3 Simulation 3

As described in the table above this simulation consist of 6 vortices, there is a polar vortex present and they are set at 80 ° and the simulation was done for 3 years. The plots at an interval of 4 months can be found in figure 13. A movie, containing all plots can be found at (32).

Looking at the plots in figure 13 we can see that they are stable over a short period, but they move a bit towards the pole over time and the vortices around the polar vortex loose some of

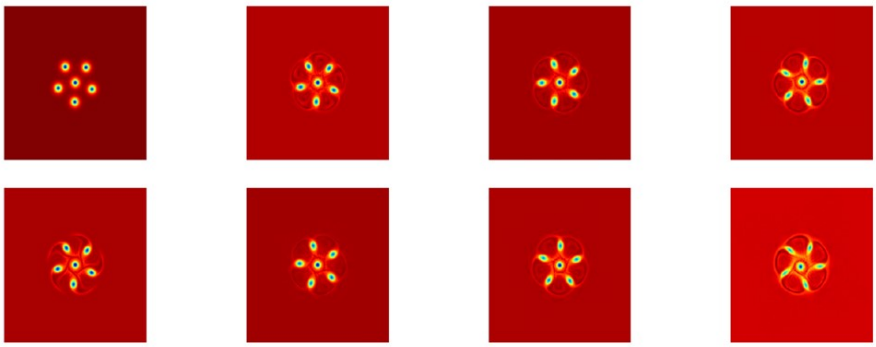

Figure 13: Plots of simulation 3 at 4 months time interval.

there vorticity to the polar vortex. Over time they are moving completely inwards.

5.4 Simulation 4

As described in the table above this simulation consist of 3 vortices, there is no polar vortex present and they are set at 80 ° and the simulation was done for 3 years. The plots at an interval of 4 months can be found in figure 14. A movie, containing all plots can be found at (32).

As seen in the plots in figure 14 and even more in the video (32) we can see that the 3 vortices are mostly stable, they only form very small anti-vortices on the edge of the original ones.

5.5 Computational Time

In order to compare the proposed model to the previous model in terms of computational time we need to look at the the time it took to do the simulations. The data per simulation can be found in table 2 and in figure 15. All simulations were running on the supercomputers owned by the research group of Andrew Ingersoll at the Geological and Planetary Sciences Institute of the California Institute of Technology. The simulations were running under similar circumstances.

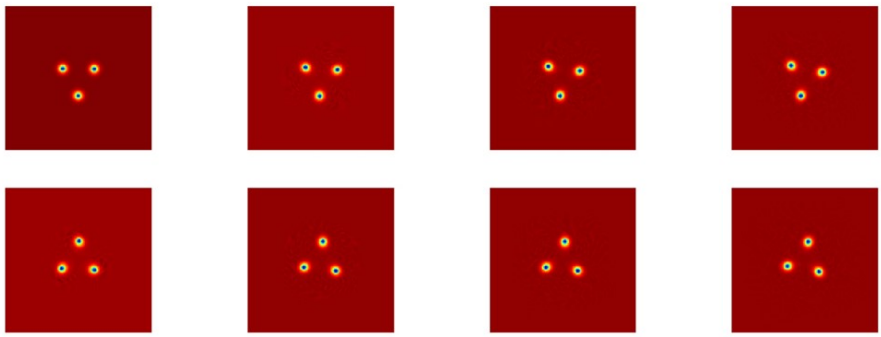

Figure 14: Plots of simulation 4 at 4 months time interval.

Simulation number	Computational Time Previous Model	Computational Time Proposed Model	Improvement Factor
1	37h 18m 12s	22h 7m 47s	1.69
2	38h 7m 5s	21h 58m 2s	1.74
3	37h 23m 18s	22h 4m 38s	1.70
4	37h 55m 45s	22h 11m 19s	1.71

Table 2: The computational times of the previous model and the proposed model

Table 2 has also a column called Improvement Factor. This factor can be defined in the following way:

$$IF = \frac{\text{Computational Time Previous Model}}{\text{Computational Time Proposed Model}} \tag{41}$$

So there can be concluded that the improvement factor defines how much faster the proposed model is compared to the previous model.

Using the data from table 2, the average improvement factor is found to be 1.71 with a standard deviation of 0.019. So there can be conclude on the basis of this very limited data set that the proposed model is 1.71 ± 0.019 times faster.

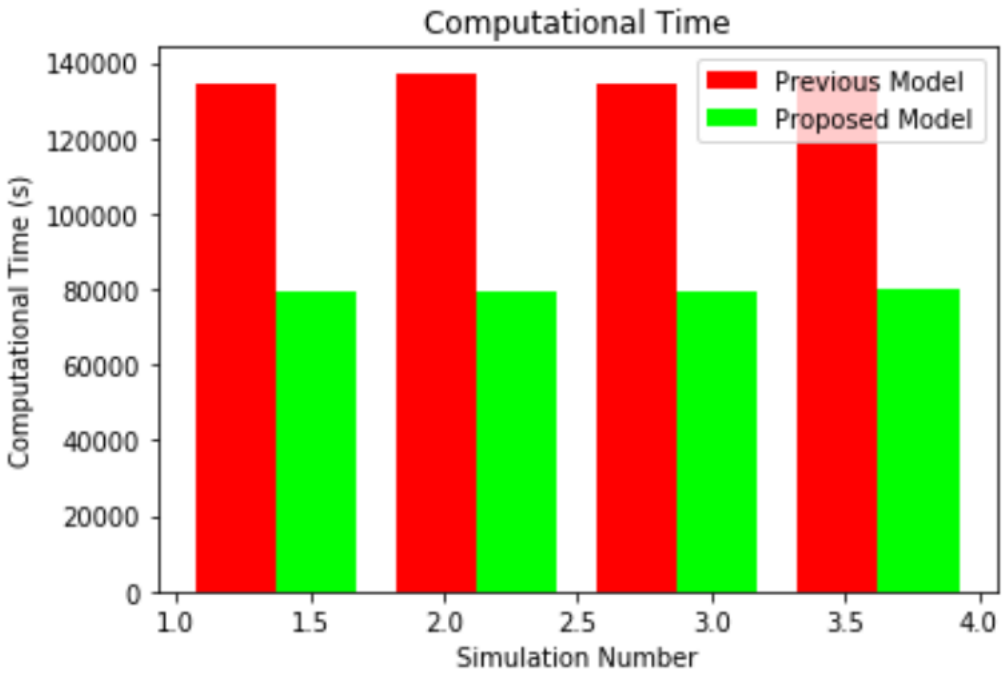

Figure 15: A comparison between the previous model and the proposed model in terms of computational time

6 Conclusions and Summary

6.1 A short summary

Looking at the Jovian atmosphere we find several patterns of cyclones, which are evolving over time. The goal of this project was to create a new model based on the Quasi Geostrophic Equations to describe the time evolution of the vortex patterns.

The Quasi Geostrophic Equation is derived in chapter 2.1 and is given below:

$$\frac{\partial}{\partial t}\left[\nabla^2\Psi - \frac{\Psi}{L_D^2} - \beta r^2\right] = J(\Psi, q - \beta r^2) \tag{42}$$

This equation looks like a Poisson equation and is solved in a similar way using Fourier Transforms ending up with the following formula which is derived in chapter 3.3:

$$\hat{\Psi}_{km} = \frac{1}{k^2 + m^2 - \frac{1}{L_D^2}}(\nabla^2\hat{\Psi}_{km} - \frac{1}{L_D^2}\hat{\Psi}_{km}) \tag{43}$$

Using this equation combined with the method to calculate the Jacobian as described in chapter 3.4 we have all the tools to step forward in time. This is done using the strong preserving third order Runge Kutta method as described in chapter 3.5. All code based on these numerical methods is listed in chapter 8.

6.2 Time evolution of Vortex patterns

Using the simulations discussed in chapter 5 we can come to some conclusions. There are a couple of parameters of which we have measured the influence on the time evolution of the vortex patterns. We find that:

1. The closer the vortices are located to the pole, the more likely they are destroyed and merge at the pole.
2. In the absence of a polar vortex the remaining vortices are less likely to merge.
3. The number of vortices influences the rotational velocity of the vortex pattern.

Point 1 is found by comparing simulation 1 and 2. We see that in simulation 1 the vortices move quickly (within a year) inside towards the pole and merge together to form a chaotic vortex at

the pole. In simulation 2 we see that the pattern is mostly stable within the time frame of 3 years. So the vortices at a longitude of 85° will move inside and merge, where the vortices at 80° are stable and do not merge.

Point 2 is found by comparing simulation 2 and simulation 3. We see in simulation 3 that the vortices move inwards in the direction of the polar vortex and become more elliptical. Towards the end of the simulation they start to merge with the polar vortex. So we can conclude that the presence of a polar vortex destabilizes the system, and will cause the system to merge at the pole.

The last point is found by comparing simulation 2 and simulation 4. The difference between these simulations are the number of vortices. We find that the 5 vortices rotate much faster than the 3 vortices. From this we can conclude that the number of vortices influences the rotational velocity of the vortex pattern.

6.3 Computational Time

From the data provided in chapter 5.5 we can conclude that the proposed model in this project is on average 1.71 ± 0.019 times faster. This improvement is mainly caused by a new way to describe the atmosphere. The Quasi Geostrophic approach is a much simpler way to describe the flows within the atmosphere compared to the shallow water equations. The main advantage of the Quasi Geostrophic method is the much larger Courant number which means that the time step can be much larger than when running the Shallow Water Model.

7 Improvements and the Future

This project resulted in two main conclusions, one about the time evolution of the vortex patterns and one about the computational time of this model.

The main improvement in the time evolution section is simulating more situations to confirm the described effects and improve the understanding of the evolution. It would be very interesting to run the simulations at different longitudes, or change the number of vortices. A combination of the changed parameters would also be interesting and crucial to a good understanding of the evolution.

It would also be interesting to expand the grid from 256×256 to 512×512 to be able to see more detailed effects during the simulation. This would increase the computational time significantly.

The second conclusion of this project is a bit hand waving. In the absence of more data to compare the two methods the improvement of the computational time is based on only four data points. In order to get a better view of the total improvement it would be good to obtain more data about the computational time.

Further more, the simulations were running on the super computer under similar circumstances, but the other tasks of the computer and the network could have an influence on the total computational time. Lastly, the Shallow Water model is running a parallel computing process where the Quasi Geostrophic model is only doing one calculation at the time.

One addition which could be made to this project is the expansion of the code and make it possible to do parallel computing processes, which would make the simulations much faster and make it possible to expand to a 512×512 grid without increasing the computational time by a large factor.

Another addition could be the introduction of a two or more layer model to describe the Jovian atmosphere more accurately and learn more about what is beneath the clouds of Jupiter.

References

[1] "Juno's nieuwe Jupiter | Unknown Universe." [Online]. Available: http://unknownuniverse.be/910/juno_jupiter/

[2] S. Bolton *et al.*, "The juno mission," *Proceedings of the International Astronomical Union*, vol. 6, no. S269, pp. 92–100, 2010.

[3] "It's Official: Juno is Going to Jupiter," Nov. 2008. [Online]. Available: https://www.universetoday.com/21451/its-official-juno-is-going-to-jupiter/

[4] S. Bolton, J. Lunine, D. Stevenson, J. Connerney, S. Levin, T. Owen, F. Bagenal, D. Gautier, A. Ingersoll, G. Orton *et al.*, "The juno mission," *Space Science Reviews*, vol. 213, no. 1-4, pp. 5–37, 2017.

[5] S. Matousek, "The juno new frontiers mission," *Acta Astronautica*, vol. 61, no. 10, pp. 932–939, 2007.

[6] T. Johnson, C. Yeates, and R. Young, "Space science reviews volume on galileo mission overview," in *The Galileo Mission*. Springer, 1992, pp. 3–21.

[7] "Juno Mission & Trajectory Design Juno." [Online]. Available: https://spaceflight101.com/juno/juno-mission-trajectory-design/

[8] R. S. Grammier, "A look inside the juno mission to jupiter," in *2009 IEEE Aerospace conference*. IEEE, 2009, pp. 1–10.

[9] G. P. Williams, "Planetary circulations: 2. the jovian quasi-geostrophic regime," *Journal of the Atmospheric Sciences*, vol. 36, no. 5, pp. 932–969, 1979.

[10] R. Salmon, G. Holloway, and M. C. Hendershott, "The equilibrium statistical mechanics of simple quasi-geostrophic models," *Journal of Fluid Mechanics*, vol. 75, no. 4, pp. 691–703, 1976.

[11] A. R. Vasavada and A. P. Showman, "Jovian atmospheric dynamics: An update after galileo and cassini," *Reports on Progress in Physics*, vol. 68, no. 8, p. 1935, 2005.

[12] A. P. Ingersoll, "Atmospheric dynamics of the outer planets," *Science*, vol. 248, no. 4953, pp. 308–315, 1990.

[13] D. Bresch and B. Desjardins, "Existence of global weak solutions for a 2d viscous shallow water equations and convergence to the quasi-geostrophic model," *Communications in mathematical physics*, vol. 238, no. 1-2, pp. 211–223, 2003.

[14] J. C. McWilliams, “A note on a consistent quasigeostrophic model in a multiply connected domain,” *Dynamics of Atmospheres and Oceans*, vol. 1, no. 5, pp. 427–441, 1977.

[15] X. Carton and J. McWilliams, “Barotropic and baroclinic instabilities of axisymmetric vortices in a quasigeostrophic model,” in *Elsevier oceanography series*. Elsevier, 1989, vol. 50, pp. 225–244.

[16] I. M. Held, R. T. Pierrehumbert, S. T. Garner, and K. L. Swanson, “Surface quasi-geostrophic dynamics,” *Journal of Fluid Mechanics*, vol. 282, pp. 1–20, 1995.

[17] A. P. Ingersoll, R. F. Beebe, J. L. Mitchell, G. W. Garneau, G. M. Yagi, and J.-P. Müller, “Interaction of eddies and mean zonal flow on jupiter as inferred from voyager 1 and 2 images,” *Journal of Geophysical Research: Space Physics*, vol. 86, no. A10, pp. 8733–8743, 1981.

[18] R. Courant, K. Friedrichs, and H. Lewy, “On the partial difference equations of mathematical physics,” *IBM Journal of Research and Development*, vol. 11, no. 2, pp. 215–234, Mar. 1967. [Online]. Available: http://ieeexplore.ieee.org/document/5391985/

[19] G. Kaiser and L. H. Hudgins, “*A Friendly Guide to Wavelets*,” *Physics Today*, vol. 48, no. 7, pp. 57–58, Jul. 1995. [Online]. Available: http://physicstoday.scitation.org/doi/10.1063/1.2808105

[20] E. M. Stein and R. Shakarchi, *Fourier analysis: an introduction*, ser. Princeton lectures in analysis. Princeton: Princeton University Press, 2003, no. 1.

[21] M. Rahman, *Applications of Fourier transforms to generalized functions*. Southampton, U.K. ; Boston, Mass: WIT Press, 2011, oCLC: ocn752818778.

[22] E. Kreyszig, H. Kreyszig, and E. J. Norminton, *Advanced engineering mathematics*, 10th ed. Hoboken, NJ: John Wiley, 2011.

[23] G. B. Folland, *Harmonic analysis in phase space*, ser. The Annals of mathematics studies. Princeton, N.J: Princeton University Press, 1989, no. no. 122.

[24] S. G. Johnson and M. Frigo, “A Modified Split-Radix FFT With Fewer Arithmetic Operations,” *IEEE Transactions on Signal Processing*, vol. 55, no. 1, pp. 111–119, Jan. 2007. [Online]. Available: http://ieeexplore.ieee.org/document/4034175/

[25] J. Cooley, P. Lewis, and P. Welch, “Historical notes on the fast Fourier transform,” *IEEE Transactions on Audio and Electroacoustics*, vol. 15, no. 2, pp. 76–79, Jun. 1967. [Online]. Available: http://ieeexplore.ieee.org/document/1161903/

[26] P. Duhamel and M. Vetterli, “Fast fourier transforms: A tutorial review and a state of the art,” *Signal Processing*, vol. 19, no. 4, pp. 259–299, Apr. 1990. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/016516849090158U

[27] “Cooley Tukey Algorithm.” [Online]. Available: https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm#/media/File:Cooley-tukey-general.png

[28] N. A. Phillips, “The general circulation of the atmosphere: A numerical experiment,” *Quarterly Journal of the Royal Meteorological Society*, vol. 82, no. 352, pp. 123–164, Apr. 1956. [Online]. Available: http://doi.wiley.com/10.1002/qj.49708235202

[29] A. Arakawa, “Computational design for long-term numerical integration of the equations of fluid motion: Two-dimensional incompressible flow. Part I,” *Journal of Computational Physics*, vol. 1, no. 1, pp. 119–143, Aug. 1966. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0021999166900155

[30] A. Hakim. [Online]. Available: https://gkeyll.readthedocs.io/en/latest/dev/ssp-rk.html#id1

[31] “Adobe After Effects.” [Online]. Available: https://www.adobe.com/nl/products/aftereffects.html

[32] C. Hoek, “Movies Bachelor Project Physics.” [Online]. Available: https://www.astro.rug.nl/∼hoek/movies.html

8 Appendix

8.1 Transformation and Initialization

```
import numpy as np

def pix_car(n,m, LenX):
    x = np.array(m) - np.ones(LenX)*(LenX/2-1)
    y = np.array(m) - np.ones(LenX)*(LenX/2-1)
    return x,y


def Lat_Lon(x,y,radius):
    dist = (x*x+y*y)**0.5
    lat = np.pi/2 - dist/radius
    lon = np.arcsin(x/dist)
    if x>0 and y>0:
        lon = np.pi - lon
    if x<0 and y>0:
        lon = -np.pi - lon
    return lat, lon


def XY(lat,lon,radius):
    dist = radius*(np.pi/2 - lat)
    x = dist*np.sin(lon)
    y = dist*np.cos(lon)
    return x,y


def Vortex(r):
    v0 = 80 #m/s
    R0 = 1e6 #m
    f0 = 2*(1.74e-4) #1/s
    return -v0*(r+R0)*np.exp(1-r/R0) - v0*v0/4/R0/f0*(2*r+R0)*np.exp(2-2*r/R0)
#-v0/R0*r*np.exp(1/2-r*r/(2*R0*R0))
#
```

```
def Vgen(where, N, LenX):
    k = 360/N
    Xmid = [0]*(N+1)
    Ymid = [0]*(N+1)
    radius = 7.1e7 #m (Radius Jupiter)
    rad1,rad = XY(70*np.pi/180,0,radius)
    print(rad,rad1)

    for i in range(N): #Create midpoints of all vortices
        Xmid[i],Ymid[i] = XY(where*np.pi/180,i*k*np.pi/180, radius)
        print(XY(where*np.pi/180,i*k*np.pi/180, radius))

    N = N+1

    VORTEX = [[0]]

    for i in range(LenX-1):
        VORTEX.append([0])

    for j in range(LenX):
        for i in range(LenX-1):
            VORTEX[j].append(0)

    XX = np.linspace(-int(LenX/2), int(LenX/2), LenX)*rad/(LenX/2)
    YY = np.linspace(-int(LenX/2), int(LenX/2), LenX)*rad/(LenX/2)

    phi0 = -8.75e4
    f = 2*1.74e-4
    Ld = (phi0/f/f)

    xv,yv = np.meshgrid(XX,YY)
    for i in range(N):
        xmid = Xmid[i]
        ymid = Ymid[i]

        for i in range(LenX):
            for j in range(LenX):
                x = XX[i]
```

```
                y = YY[j]
                dx = x - xmid
                dy = y - ymid
                r =(dx*dx+dy*dy)**0.5
                VORTEX[j][i] = VORTEX[j][i] + Vortex(r)
        print("")
    for i in range(LenX):
        for j in range(LenX):
            VORTEX[j][i] = VORTEX[j][i]
    return VORTEX, Xmid, Ymid

LenX = 256
V, Xmid, Ymid = Vgen(80,3,LenX)
radius = 7.1e7
XX = np.linspace(-int(LenX/2), int(LenX/2), LenX)
YY = np.linspace(-int(LenX/2), int(LenX/2), LenX)
xv,yv = np.meshgrid(XX,YY)

np.save("vortex",V)
%matplotlib notebook
import matplotlib.pyplot as plt
n_bins = LenX

weights = np.array(V)

# We'll also create a grey background into which the pixels will fade
greys = np.full((*weights.shape, 3), 70, dtype=np.uint8)

# First we'll plot these blobs using only ``imshow``.
vmax = np.abs(weights).max()
vmin = -vmax
cmap = plt.cm.inferno

fig, ax = plt.subplots()
ax.imshow(greys)
ax.imshow(weights,  cmap=cmap)
ax.set_axis_off()
plt.show()
```

```
        from matplotlib.pyplot import figure, show
        import matplotlib.pyplot as plt


        rad,rad1 = XY(20*np.pi/180,np.pi,radius)
        XX = np.linspace(-int(LenX/2), int(LenX/2), LenX)*rad/(LenX/2)
        YY = np.linspace(-int(LenX/2), int(LenX/2), LenX)*rad/(LenX/2)

        xv,yv = np.meshgrid(XX,YY)

        fig = figure(figsize=(4,4))
        frame = fig.add_subplot(1,1,1)
        cset1 = frame.contour(xv,yv,weights)
        plt.show()
        #fig.colorbar(cset1)

        figg = figure(figsize=(4,4))
        framee = figg.add_subplot(1,1,1)
        framee.plot(Xmid,Ymid, 'o', )

        plt.show()

rad1,rad = XY(70*np.pi/180,0,radius)
        rad = rad
        XX = np.linspace(-int(LenX/2), int(LenX/2), LenX)*rad/(LenX/2)
        YY = np.linspace(-int(LenX/2), int(LenX/2), LenX)*rad/(LenX/2)

        f0 = 2*1.74e-4
        beta = 1/(2*radius*radius)*f0

        other_terms = [[0]]

        for i in range(LenX-1):
            other_terms.append([0])

        for j in range(LenX):
            for i in range(LenX-1):
```

```
            other_terms[j].append(0)

    for i in range(LenX):
        for j in range(LenX):
            r2 = XX[i]**2 + YY[j]**2
            other_terms[j][i] = (f0 - beta*r2)

    np.save("other_terms", other_terms)


    %matplotlib inline

    fig = figure(figsize=(4,4))
    frame = fig.add_subplot(1,1,1)
    cset1 = frame.contour(other_terms)
    plt.show()
```

8.2 Simulation

```
import numpy as np
     def XY(lat,lon,radius):
         dist = radius*(np.pi/2 - lat)
         x = dist*np.sin(lon)
         y = dist*np.cos(lon)
         return x,y

     def fou_der2(f,Ld):
         #Length of vector f
         LenX = int(np.size(f)**0.5)
         #Create matrix of Ld^2
         LD = np.ones((LenX,LenX))*1/Ld/Ld
         #Create list of k values
         t = np.array(list(np.linspace(0,LenX-2,LenX/2))+
            list(np.linspace(-LenX,0-2,LenX/2)))*np.pi/LenX
         X = []
         for i in range(LenX):
             X.append(t)
         X = np.array(X) #create 2D array for k values
```

```
    Y = np.transpose(X) #transpose array for k values to obtain l values
    XY = -(X*X+Y*Y+LD) #Calculate sum of squares of k and l values and Ld^2
    for i in range(LenX):
        for j in range(LenX):
            if i==0 and j==0:
                continue
            XY[j][i] = 1/XY[j][i] #find inverse at each location
    # Fourier derivative
    ff = np.fft.fft2(f) #calculate Fourier Transform of input array
    ff = XY*ff
    # multiply by k,l value matrix to obtain the derivative in Fourier Space
    df_num = np.real(np.fft.ifft2(ff))
    #Calculate the inverse Fourier Transform to go back to real space
    return df_num

def psi(F,LenX,Ld):

    psi = fou_der2(F,Ld) #Calulate psi from Jacobian

    Lap_psi = F + 1/Ld/Ld*np.array(psi)
    #Calculate Lap_psi using the geostrophic equation

    return psi, Lap_psi

def subJac(a,b,c,d,e,f,Jacobi,q,p,h):
    "Calculate the value for the Jacobian index"
    B1 = (1/2/h*(p[e][c]-p[e][a])*1/2/h*(q[f][b]-q[d][b])) -
       (1/2/h*(p[f][b]-p[d][b])*1/2/h*(q[e][c]-q[e][a]))
    B2 = (1/2/h*(p[e][c]*1/2/h*(q[f][c]-q[d][c]) -
       p[d][a]*1/2/h*(q[f][a]-q[d][a]))-1/2/h*(p[f][b]*1/2/h*(q[f][c]-q[f][a])-
       p[d][b]*1/2/h*(q[d][c]-q[d][a])))
    B3 = -(1/2/h*(q[e][c]*1/2/h*(p[f][c]-p[d][c]) -
       q[d][a]*1/2/h*(p[f][a]-p[d][a]))-1/2/h*(q[f][b]*1/2/h*(p[f][c]-p[f][a])-
       q[d][b]*1/2/h*(p[d][c]-p[d][a])))
    Jacobi[b][e] = (B1+B2+B3)/3
    return Jacobi

def Jacobian(LenX, LenY, psi, Lap_psi, other_terms):
```

```
"Calculate the Jacobian following Arakawa's paper"
#Create arrays with correct dimensions to store values
radius = 7.1e7
rad1,rad = XY(70*np.pi/180,0,radius)
rad = rad
h = rad/128
Jacobi = []
g = []

for i in range(LenY):
    Jacobi.append([0])
    g.append([0])

for i in range(LenY):
    for j in range(LenX-1):
        Jacobi[i].append(0)
        g[i].append(0)

#Create a matrix for g
for i in range(LenX):
    for j in range(LenY):
        g[j][i]= Lap_psi[j][i] + other_terms[j][i]

#Calculate the Jacobian at each point
for j in range(1,LenY-1):
    for i in range(1,LenX-1):
        Jacobi = subJac(j-1,j,j+1,i-1,i,i+1,Jacobi,psi,g,h)

for i in range(1, LenX-1):
    Jacobi = subJac(LenY-1,0,1,i-1,i,i+1,Jacobi,psi,g,h)
    Jacobi = subJac(LenY-2,LenY-1,0,i-1,i,i+1,Jacobi,psi,g,h)

for j in range(1, LenY-1):
    Jacobi = subJac(j-1,j,j+1,LenX-1,0,1,Jacobi,psi,g,h)
    Jacobi = subJac(j-1,j,j+1,LenX-2,LenX-1,0,Jacobi,psi,g,h)

Jacobi = subJac(LenY-1,0,1,LenX-1,0,1,Jacobi,psi,g,h)
Jacobi = subJac(LenY-2,LenY-1,0,LenX-1,0,1,Jacobi,psi,g,h)
```

```
    Jacobi = subJac(LenY-1,0,1,LenX-2,LenX-1,0,Jacobi,psi,g,h)
    Jacobi = subJac(LenY-2,LenY-1,0,LenX-2,LenX-1,0,Jacobi,psi,g,h)

    return Jacobi

def timestep(F, LenX, Ld, dt, other_terms):

    F = np.array(F) #transform list into array

    Psi, Lap_psi = psi(F,LenX,Ld)
    #Calculate psi and lap_psi at starting time
    jacob = Jacobian(LenX, LenX, Psi, Lap_psi, other_terms)
    #Calculate the jacobian form psi and lap_psi

    F1 = F - dt*np.array(jacob) #Do first timestep

    Psi, Lap_psi = psi(F1,LenX,Ld) #Recalculate everything
    jacob = Jacobian(LenX, LenX, Psi, Lap_psi, other_terms)

    F2 = 3/4*F+1/4*(F1 + dt*np.array(jacob))#Second timestep

    Psi, Lap_psi = psi(F2,LenX,Ld)#Recalculate everything
    jacob = Jacobian(LenX, LenX, Psi, Lap_psi, other_terms)

    Fn = 1/3*F + 2/3*(F2 + dt*np.array(jacob)) #Final result for new time

    return Fn

def TimeStepRK4(F, LenX, Ld, dt, other_terms):

    F = np.array(F) #transform list into array

    Psi, Lap_psi = psi(F,LenX,Ld)
    #Calculate psi and lap_psi at starting time
    jacob = Jacobian(LenX, LenX, Psi, Lap_psi, other_terms)
    #Calculate the jacobian form psi and lap_psi

    K1 = dt*np.array(jacob) #Do first timestep
```

```
        Psi, Lap_psi = psi(F+K1/2,LenX,Ld) #Recalculate everything
        jacob = Jacobian(LenX, LenX, Psi, Lap_psi, other_terms)

        K2 = dt*np.array(jacob)

        Psi, Lap_psi = psi(F+K2/2,LenX,Ld) #Recalculate everything
        jacob = Jacobian(LenX, LenX, Psi, Lap_psi, other_terms)

        K3 = dt*np.array(jacob)

        Psi, Lap_psi = psi(F+K3,LenX,Ld) #Recalculate everything
        jacob = Jacobian(LenX, LenX, Psi, Lap_psi, other_terms)

        K4 = dt*np.array(jacob)

        Fn = F + K1/6 + K2/3 + K3/3 + K4/6

        return Fn
```

8.3 Recursion Algorithm

```
import matplotlib.pyplot as plt
    def recu(F,i, other_terms,ld,dt):
        K = TimeStepRK4(F,256,ld,dt,other_terms)
        i = i+1
        # Generate the space in which the blobs will live
        xmin, xmax, ymin, ymax = (0,LenX, 0,LenX)
        n_bins = LenX
        xx = np.linspace(xmin, xmax, n_bins)
        yy = np.linspace(ymin, ymax, n_bins)

        xv, yv = np.meshgrid(xx, yy)

        # We'll also create a grey background into which the pixels will fade
        greys = np.full((*K.shape, 3), 70, dtype=np.uint8)

        # First we'll plot these blobs using only ``imshow``.
```

```
    vmax = np.abs(K).max()
    vmin = -vmax
    cmap = plt.cm.jet

    fig, ax = plt.subplots()
    ax.imshow(greys)
    ax.imshow(K, cmap=cmap)
    ax.set_axis_off()

    name = 'plot'+str(i)
    plt.savefig(name)
    if i>30:
        return 0
    else:
        print(i)
        recu(K,i,other_terms,Ld,dt)

import numpy as np
dt = 20
other_terms = np.load('other_terms.npy')
F = np.load('vortex.npy')
phi0 = 8.75e4
f = 2*1.74e-4
Ld = ((phi0/f/f)**0.5)
LenX = 256

recu(F,0,other_terms,Ld,dt)
```