CLimate Analysis using Digital Estimations Non-Offical Manual (CLAuDE NOM)

Sam "TechWizard" Baggen

January 18, 2022

Contents

0 Introduction

The CLimate Analysis using Digital Estimations model is a simplified planetary climate model. It will be used to educate people on how climate physics works and to experiment with different parameters and see how much influence a tiny change can have (like for instance the rotation rate of the planet around its axis). It is built to be accessible to and runnable by everyone, whether they have a super computer or a dated laptop. The model is written in Python and written during the weekly streams of Dr. Simon Clark [15]. There is a useful playlist on Simon's Twitch which has all the streams without ad breaks or interruptions [16].

The manual itself is split up into distinct sections, each explaining one particular part of the model. Each section will be treating one topic, like radiation, advection or the control panel. Although many concepts cannot be seen in isolation, as the wind has influence on how much temperature is distributed throughout the atmosphere, the calculations can be split up. The manual is cumulative, starting with the basics and slowly building up to the current form of the algorithm. All changes to the algorithms can therefore be found here. An important distinction needs to be made regarding the changes though. If the changes only change one part of the calculations, then it is considered an evolution, which will be added to the relevant section. However if the changes are significant and not based on the previous code then the old alghorithms will be relocated to Appendix B. Though the relevant theory will remain, as that is required to gain an understanding of what the algorithm does. Do note that the radiation section 3 is an exception for the first calculations as this forms the basis of the beginning of CLAuDE and the fundamentals of the theory which I deem important enough to be left in place even if the calculations end up significantly different.

This manual will provide an overview of the formulae used and will explain aspects of these formulae. For each equation each symbol will be explained what it is. In such an explanation, the units will be presented in SI units [13] between brackets like: T: The temperature of the planet (K). Which indicates that T is the temperature of the planet in degrees Kelvin. If you need to relate SI units to your preferred system of units, please refer to the internet for help with that. There are great calculators online where you only need to plug in a number and select the right units.

Within this manual we will not concern ourselves with plotting the data, instead we focus on the physics side of things and translating formulae into code. If you are interested in how the plotting of the data works, or how loading and saving data works, please refer to the relevant stream on Simon's Twitch page [15].

This manual is for the toy model, which is as of now still in development. One important thing to note is that the layout may change significantly when new sections are added. This is due to the amount of code that is added/changed. If a lot of code changes, a lot of so called 'algorithm' blocks are present which have different placement rules than just plain text. Therefore it may occur that an algorithm is referenced even though it is one or two pages later. This is a pain to fix and if something later on changes, the whole layout may be messed up again and is a pain to fix again. Hence I opt to let LaTeX(the software/typeset language used to create this manual) figure out the placement of the algorithm blocks, which may or may not be in the right places.

One last important thing, I try to make this manual as generic as possible regarding the code. This means that I try to steer away from using Python specific notation. However, this manual is also the documentation for the project written in Python. Therefore it is sometimes unavoidable to use Python notation. Furthermore, to make the model run efficiently, the code is vectorised. Vectorised code is however more difficult to understand (as a programmer without the required linear algebra) than the For loops that it replaces. Therefore, it is not uncommon to see (parts of) algorithms repeated in their relevant sections, once as a standard for loop and once as the vectorised version.

The manual is now up on the Planet Factory GitHub repository [18], together with all the source code. There is also a fork [9] that also contains the source code. The fork will usually be more up to date than the version on the Planet Factory repository as Simon needs to merge pull requests into the repository. However I can update the fork freely so if a particular stream is missing in the version on the Planet Factory repository, check the fork/Discord whether there is a newer version. If that is not the case, you just have to be a bit more patient, or you can start writing a part of the manual yourself! Don't forget to ping me in the Discord to notify me of any additions (GitHub refuses to send me emails so I have no other way of knowing).

1 Control Panel

Before we dive in an start modelling the planet, let us first set up a control panel that will influence how the model will behave and effectively decides what type of planet we model.

1.1 The Beginning

In the beginning there was nothing, and then there was "Hello World!" Or at least that is how many projects start. Why? you might ask, which is a perfectly valid question. In Computer Science, "Hello World!" is very simple code that we use to test whether all the tools we need to get coding works. This checks whether the computer compiles the code and is able to execute it and whether the code editor (IDE, Integrated Development Environment) starts the right processes to get the code compiled and executed. Oh right we were talking about CLAuDE, ahem.

Every project must have its beginning. And with CLAuDE I made the decision to start explaining the Control Panel first. This is to get you familiar with notation and to lay down some basics. To do that we start with the fixed part of the Control Panel, the physical constants. Many things vary from planet to planet, how much radiation they receive from their star, how strong their gravity is, how fast they spin around their axis and many many more. What does not change are the physical constants, well because they are constant. The Stefan-Boltzmann constant for instance does not change. Whether you are on Earth, in space or on Jupiter, the value of the Stefan-Boltzmann constant will remain the same.

The Stefan-Boltzmann constant is denoted by σ and has a value of $5.670373 \cdot 10^{-}8$ ($\mathrm{Wm^{-2}K^{-4}}$) [39]. The σ is a greek letter called sigma. Greek letters are often used in mathematics, as well as in physics or any other discipline that relies on maths (spoiler alert, quite a lot). Treat it like a normal letter in maths, representing a number that you either do not know yet or is too long or cumbersome to write down every time. The Stefan-Boltzmann constant is denoted in scientific notation, a number followed by the order of magnitude. It is denoted as a multiplication, because that is what you have to do to get the real number. An example: $4.3 \cdot 10^2 = 430$ and $4.3 \cdot 10^{-2} = 0.043$. The letters behind the numbers are units, how we give meaning to the numbers. If I say that I am 1.67 does not mean anything. Do I mean inches, centimeters, meters, miles? That is why we need units as they give meaning to the number. they tell us whether the number is talking about speed, distance, time, energy and many other things. In this manual we will use SI units. Behind all the letters you will find the following: [number]. This is a citation, a reference to an external source where you can check whether I can still read. If I pull a value out somewhere I will insert a citation to show that I am not making these numbers up. This is what scientists use to back up their claims if they do not want to redo the work that others have done. I mean what is the point of re-inventing the wheel if there is a tyre company next door? That is why scientists citate.

So with that out of the way, let us write down some constants. Why do I do this here? Because a lot of constants are used everywhere and I am too lazy to replicate them every time. If you see a letter or symbol that is not explicitly explained, then it is most likely a constant that we discuss here in the control panel.

1.2 Physical Constants

As mentioned before, physical constants do not change based on where you are in the universe. Below you will find an overview of all the relevant constants together with their units. And a short explanation where they are used or what they represent. To see them in action, consult the other sections of this manual, you will find them in equations all throughout this document.

1.2.1 The Gas Constant

The Gas constant, $R = 8.3144621$ ($\mathrm{JK^{-1}mol^{-1}}$) [40] is the constant used to relate the temperature of the gas to the pressure and the volume. One would expect this constant to be different per gas, but under high enough temperatures and low enough pressure the gas constant is the same for all gases.

1.2.2 The Specific Heat Capacity

The specific heat capacity c depicts how much energy is required to heat the object by one degree Kelvin per unit mass ($\text{Jkg}^{-1}\text{K}^{-1}$) [41]. This varies per material and is usually indicated by a subscript. The specific heat capacity for water for instance is $c_w = 4190$ $\text{Jkg}^{-1}\text{K}^{-1}$. Specific heat capacities also exist in the form of $\text{Jkg}^{-1}\text{K}^{-1}$, $\text{Jmol}^{-1}\text{K}^{-1}$ and $\text{Jcm}^{-3}\text{K}^{-1}$ which you can use in various circumstances, depending on what information you have.

1.2.3 Mole

Mole is the amount of particles ($6.02214076 \cdot 10^{23}$) in a substance, where the average weight of one mole of particles in grams is about the same as the weight of one particle in atomic mass units (u) [14]. This is not a physical constant perse, but more like a unit (mol). Though it is still important enough to be added here for future reference. All other units are way more intuitive and are assumed to be known.

1.2.4 The Stefan-Boltzmann Constant

The Stefan-Boltzmann constant, $\sigma = 5.670373 \cdot 10^{-}8$ $\text{Wm}^{-2}\text{K}^{-4}$) [39] is used in the Stefan-Boltzmann law (more on that in subsection 3.1).

1.3 Planet Specific Variables

The following set of variables vary per planet, that's why we call them variables since they vary. Makes sense right? We add them here as we will use them throughout the manual. The advantage of that is quite significant. If you want to test things for a different planet, you only need to change the values in one place, instead of all places where you use it. If there is one thing that we computer scientists hate is doing work, we like being lazy and defining things in one place means that we can be lazy if we need to change it. So we put in the extra work now, so we do not have to do the extra work in the future. That's actually a quite accurate description of computer scientists, doing hard work so that they can be lazy in the future.

1.3.1 The Passage of Time

On Earth we have various indications of how much time has passed. While most of them remain the same throughout the universe, like seconds, minutes and hours, others vary throughout the universe, like days, months and years. Here we specify how long the variable quantities of time are for the planet we want to consider as they are used in the code. This can be seen in algorithm 1. Here a $\leftarrow$ indicates that we assign a value to the variable name before it, so that we can use the variable name in the code instead of the value, which has the advantage I indicated before. // means that we start a comment, which is text that the code ignores and does not tell the cpu about. Not that the cpu would understand it, but that just means less work for the computer. Yes computers are lazy too.

Algorithm 1: Definition of how much time it takes for a day and a year on a planet and how much time on the planet passes before we start another calculation run

$day \leftarrow 60 * 60 * 24$; `//Length of one day in seconds (s)`
$year \leftarrow 365 * day$; `//Length of one year in seconds (s)`
$\delta t \leftarrow 60 * 9$; `//How much time is between each calculation run in seconds (s)`

1.3.2 The Planet Passport

Each planet is different, so why should they all have the same gravity? Oh wait, they don't. Just as they are not all the same size, tilted as much and their atmospheres differ. So here we define all the relevant variables that are unique to a planet, or well not necessarily unique but you get the idea. This can all be found in algorithm 2.

Algorithm 2: Defining the constants that are specific to a planet

$g \leftarrow 9.81$; //Magnitude of gravity on the planet in ms^{-2}
$\alpha \leftarrow -23.5$; //By how many degrees the planet is tilted with respect to the star's plane
$top \leftarrow 50 * 10^3$; //How high the top of the atmosphere is with respect to the planet surface in meters (m)
$ins \leftarrow 1370$; //Amount of energy from the star that reaches the planet per unit area (Jm^{-2})
$\epsilon \leftarrow 0.75$; //Absorbtivity of the atmosphere, fraction of how much of the total energy is absorbed (unitless)
$r \leftarrow 6.4 * 10^6$; //The radius of the planet in meters (m)
$p_z \leftarrow$
[100000, 95000, 90000, 80000, 70000, 60000, 50000, 55000, 40000, 5000, 30000, 25000, 20000, 15000, 10000, 7500, 5000, 2500, 1
; //The pressure of the different levels (Pa) noted in an array

1.3.3 Model Specific Parameters

These parameters cannot be found out in the wild, they only exist within our model. They control things like the size of a cell on the latitude longitude grid (more on that in later sections), how much time the model gets to spin up. We need the model to spin up in order to avoid numerical instability. Numerical instability occurs when you first run the model. This is due to the nature of the equations. Nearly all equations are continuous, which means that they are always at work. However when you start the model, the equations were not at work yet. It is as if you suddenly give a random meteor an atmosphere, place it in orbit around a star and don't touch it for a bit. You will see that the whole system oscilates wildly as it adjusts to the sudden changes and eventually it will stabilise. We define the amount of time it needs to stabilise as the spin up time. All definitions can be found in algorithm 3. What the *adv* boolean does is enabling or disabling advection, a process described in ¡section¿ which does not work yet.

Algorithm 3: Defining the paramters that only apply to the model

$resolution \leftarrow 3$; //The amount of degrees on the latitude longitude grid that each cell has, with this setting each cell is 3 degrees latitude high and 3 degrees longitude wide
$nlevels \leftarrow 10$; //The amount of layers in the atmosphere
$\delta t_s \leftarrow 60 * 137$; //The time between calculation rounds during the spin up period in seconds (s)
$t_s \leftarrow 5 * day$; //How long we let the planet spin up in seconds (s)
$adv \leftarrow$ FALSE ; //Whether we want to enable advection or not
$adv\_boun \leftarrow 8$; //How many cells away from the poles where we want to stop calculating the effects of advection
$C_a \leftarrow 287$; //Heat capacity of the atmosphere in $\mathrm{Jkg^{-1}K^{-1}}$
$C_p \leftarrow 1 \cdot 10^6$; //Heat capacity of the planet in $\mathrm{Jkg^{-1}K^{-1}}$
$\delta y \leftarrow \frac{2\pi r}{nlat}$; //How far apart the gridpoints in the y direction are (degrees latitude)
$\alpha_a \leftarrow 2 \cdot 10^{-5}$; //The diffusivity constant for the atmosphere
$\alpha_p \leftarrow 1.5 \cdot 10^{-6}$; //The diffusivity constant for the planet surface
$smooth_t \leftarrow 0.9$; //the smoothing parameter for the temperature
$smooth_u \leftarrow 0.8$; //The smoothing parameter for the u component of the velocity
$smooth_v \leftarrow 0.8$; //The smoothing parameter for the v component of the velocity
$smooth_w \leftarrow 0.3$; //The smoothing parameter for the w component of the velocity
$smooth_{vert} \leftarrow 0.3$; //The smoothing parameter for the vertical part of the velocities
$count \leftarrow 0$;
for $j \in [0, top]$ **do**
  $heights[j] \leftarrow count$; //The height of a layer (m)
  $count \leftarrow count + \frac{top}{nlevels}$;
end
for $i \in [0, nlat]$ **do**
  $\delta x[i] \leftarrow \delta y \cos(lat[i] \frac{\pi}{180})$; //How far apart the gridpoints in the x direction are (degrees longitude)
end
for $k \in [0, nlevels - 1]$ **do**
  $\delta z[k] \leftarrow p_z[k+1] - p_z[k]$; //How far apart the gridpoints in the z direction are (Pa)
end
$\Pi \leftarrow$ Array with the same amount of elements as p_z ; //Dimensionless pressure that is used to easily convert from potential temperature to absolute temperature and vice versa
$\kappa \leftarrow \frac{R}{C_p}$;
for $i \leftarrow 0$ **to** $\Pi.length$ **do**
  $\Pi[i] \leftarrow C_p(\frac{p_z[i]}{p_z[0]})^\kappa$;
end
$poleLowerLatLimit \leftarrow 4$; //Polar Plane Approximation Lower Grid Limit
$poleHigherLatLimit \leftarrow 4$; //Polar Plane Approximation Upper Grid Limit

2 Utility Functions

With the control panel defined and explained, let us move over to some utility functions. Functions that can be used in all kinds of calculations, which we might need more often. In general it concerns functions like calculating the gradient, the lacplacian or interpolation.

2.1 Gradients

Let us define the gradient in the x, y and z directions. The functions can be found in algorithm 4, algorithm 5 and algorithm 6. We use these functions in various other algorithms as the gradient (also known as derivative) is often used in physics. It denotes the rate of change, how much something changes over time. Velocity for instance denotes how far you move in a given time. Which is a rate of change, how much your distance to a given point changes over time.

In algorithm 6 *a.dimensions* is the attribute that tells us how deeply nested the array a is. If the result is 1 we have just a normal array, if it is 2 we have a double array (an array at each index of the array) which is also called a matrix and if it is 3 we have a triple array. We need this because we have a one-dimensional case, for when we do not use multiple layers and a three-dimensional case for when we do use multiple layers. This distinction is needed to avoid errors being thrown when running the model with one or multiple layers.

This same concept can be seen in algorithm 4 and algorithm 5, though here we check if k is defined or `NULL`. We do this as sometimes we want to use this function for matrices that does not have the third dimension. Hence we define a default value for k which is `NULL`. `NULL` is a special value in computer science. It represents nothing. This can be useful sometimes if you declare a variable to be something but it is referring to something that has been deleted or it is returned when some function fails. It usually indicates that something special is going on. So here we use it in the special case where we do not want to consider the third dimension in the gradient. We also use forward differencing (calculating the gradient by taking the difference of the cell and the next/previous cell, multiplied by 2 to keep it fair) in algorithm 5 as that gives better results for the calculations we will do later on.

Algorithm 4: Calculating the gradient in the x direction

Input : Matrix (double array) a, first index i, second index j, third index k with default value `NULL`
Output: Gradient in the x direction
if $k = NULL$ **then**
  $grad \leftarrow \frac{a[i,(j+1) \bmod nlon] - a[i,(j-1) \bmod nlon]}{\delta x[i]}$;
else
  $grad \leftarrow \frac{a[i,(j+1) \bmod nlon,k] - a[i,(j-1) \bmod nlon,k]}{\delta x[i]}$;
end
return $grad$;

2.2 Laplacian Operator

The Laplacian operator (∇^2, sometimes also seen as Δ) has two definitions, one for a vector field and one for a scalar field. The two concepts are not indpendent, a vector field is composed of scalar fields [3]. Let us define a vector field first. A vector field is a function whose domain and range are a subset of the Eucledian $\mathbb{R}^3$ space. A scalar field is then a function consisting out of several real variables (meaning that the variables can only take real numbers as valid values). So for instance the circle equation $x^2 + y^2 = r^2$ is a scalar field as x, y and r are only allowed to take real numbers as their values.

With the vector and scalar fields defined, let us take a look at the Laplacian operator. For a scalar field ϕ the laplacian operator is defined as the divergence of the gradient of ϕ [4]. But what are the divergence and gradient? The gradient is defined in Equation 1a and the divergence is defined in Equation 1b. Here ϕ is a vector with components x, y, z and Φ is a vector field with components x, y, z. Φ_1, Φ_2 and Φ_3 refer to the functions that result in the corresponding x, y and z values [3]. Also, i, j and k are the basis vectors of

Algorithm 5: Calculating the gradient in the y direction

Input : Matrix (double array) a, first index i, second index j, third index k with default value `NULL`
Output: Gradient in the y direction
if $k =$ *NULL* **then**
  if $i == 0$ **then**
    $grad \leftarrow 2\frac{a[i+1,j]-a[i,j]}{\delta y}$;
  else if $i = nlat - 1$ **then**
    $grad \leftarrow 2\frac{a[i,j]-a[i-1,j]}{\delta y}$;
  else
    $grad \leftarrow \frac{a[i+1,j]-a[i-1j]}{\delta y}$;
else
  if $i = 0$ **then**
    $grad \leftarrow 2\frac{a[i+1,j,k]-a[i,j,k]}{\delta y}$;
  else if $i = nlat - 1$ **then**
    $grad \leftarrow 2\frac{a[i,j,k]-a[i-1,j,k]}{\delta y}$;
  else
    $grad \leftarrow \frac{a[i+1,j]-a[i-1j]}{\delta y}$;
end
return $grad$;

Algorithm 6: Calculating the gradient in the z direction

Input : Array a, array p_z, index k
Output: Gradient in the z direction
$nlevels \leftarrow p_z.length$;
if $k = 0$ **then**
  $grad \leftarrow \frac{a[k+1]-a[k]}{p_z[k+1]-p_z[k]}$;
else if $k = nlevels - 1$ **then**
  $grad \leftarrow \frac{a[k]-a[k-1]}{p_z[k]-p_z[k-1]}$;
else
  $grad \leftarrow \frac{a[k+1]-a[k-1]}{p_z[k+1]-p_z[k-1]}$;
return $-grad$;

$\mathbb{R}^{\text{IK}}$, and the multiplication of each term with their basis vector results in Φ_1, Φ_2 and Φ_3 respectively. If we then combine the two we get the Laplacian operator, as in Equation 1c.

$$\text{grad } \phi = \nabla\phi = \frac{\delta\phi}{\delta x}i + \frac{\delta\phi}{\delta y}j + \frac{\delta\phi}{\delta z}k \tag{1a}$$

$$\text{div}\Phi = \nabla \cdot \Phi = \frac{\delta\Phi_1}{\delta x} + \frac{\delta\Phi_2}{\delta y} + \frac{\delta\Phi_3}{\delta z} \tag{1b}$$

$$\nabla^2\phi = \nabla \cdot \nabla\phi = \frac{\delta^2\phi}{\delta x^2} + \frac{\delta^2\phi}{\delta y^2} + \frac{\delta^2\phi}{\delta z^2} \tag{1c}$$

For a vector field Φ the Laplacian operator is defined as in Equation 2. Which essential boils down to taking the Laplacian operator of each function and multiply it by the basis vector.

$$\nabla^2\Phi = (\nabla^2\Phi_1)i + (\nabla^2\Phi_2)j + (\nabla^2\Phi_3)k \tag{2}$$

The code can be found in algorithm 7. Δ_x and Δ_y in algorithm 7 represents the calls to algorithm 4 and algorithm 5 respectively.

Algorithm 7: Calculate the laplacian operator over a matrix a

Input : A matrix (double array) a
Output: A matrix (double array) with results for the laplacian operator for each element
if $a.dimensions = 2$ **then**
  for $lat \leftarrow 1$ **to** $nlat - 1$ **do**
    for $lon \leftarrow 0$ **to** $nlon$ **do**
      $output[lat, lon] \leftarrow \frac{\Delta_x(a,lat,(lon+1) \bmod nlon) - \Delta_x(a,lat,(lon-1) \bmod nlon)}{\delta x[lat]} + \frac{\Delta_y(a,lat+1,lon) - \Delta_y(a,lat-1,lon)}{\delta y}$;
    end
  end
else
  for $lat \leftarrow 1$ **to** $nlat - 1$ **do**
    for $lon \leftarrow 0$ **to** $nlon$ **do**
      for $k \leftarrow 0$ **to** $nlevels - 1$ **do**
        $output[lat, lon, k] \leftarrow \frac{\Delta_x(a,lat,(lon+1) \bmod nlon,k) - \Delta_x(a,lat,(lon-1) \bmod nlon,k)}{\delta x[lat]} + \frac{\Delta_y(a,lat+1,lon,k) - \Delta_y(a,lat-1,lon,k)}{\delta y} + \frac{\Delta_z(a,lat,lon,k+1) - \Delta_z(a,lat,lon,k+1)}{2\delta z[k]}$;
      end
    end
  end
end
return *ouput* ;

2.3 Divergence

As we expect to use the divergence operator more often throughout our model, let us define a seperate function for it in algorithm 8. Δ_x and Δ_y in algorithm 8 represents the calls to algorithm 4 and algorithm 5 respectively. We do the multiplication with the velocity vectors u, v and w here already, as we expect that we might use it in combination with the divergence operator more frequently. What those vectors are and represent we will discuss in subsection 4.2.

Algorithm 8: Calculate the result of the divergence operator on a vector

Input : A matrix (triple array) a, pressure field p_z
Output: A matrix (triple array) containing the result of the divergence operator taken over that element
for $i \leftarrow 0$ **to** $a.length$ **do**
  for $j \leftarrow 0$ **to** $a[i].length$ **do**
    for $k \leftarrow 0$ **to** $a[i, j].length$ **do**
      $output[i, j, k] \leftarrow \Delta_x(au, i, j, k) + \Delta_y(av, i, j, k) + \Delta_z(aw[i, j, :], p_z, k)$;
    end
  end
end
return *output* ;

2.4 Interpolation

Interpolation is a form of estimation, where one has a set of data points and desires to know the values of other data points that are not in the original set of data points [10]. Based on the original data points, it is estimated what the values of the new data points will be. There are various forms of interpolation like linear interpolation, polynomial interpolation and spline interpolation. The CLAuDE model uses linear

interpolation which is specified in Equation 3. Here z is the point inbetween the known data points x and y. λ is the factor that tells us how close z is to y in the interval $[0, 1]$. If z is very close to y, λ will have the value on the larger end of the interval, like 0.9. Whereas if z is close to x then λ will have a value on the lower end of the interval, like 0.1.

$$z = (1 - \lambda)x + \lambda y \tag{3}$$

2.5 3D smoothing

As you can imagine the temperature, pressure and the like vary quite a lot over the whole planet. Which is something that we kind of want but not really. What we really want is to limit how much variety we allow to exist. For this we are going to use Fast Fourier Transforms, also known as FFTs. A Fourier Transform decomposes a wave into its frequences. The fast bit comes from the algorithm we use to calculate it. This is because doing it via the obvious way is very slow, in the order of $O(n^2)$ (for what that means, please visit subsection A.2). Whereas if we use the FFT, we reduce the running time to $O(n \log(n))$. There are various ways to calculate the FFT, but we use the Cooley–Tukey algorithm [22]. To explain it, let us first dive into a normal Fourier Transform.

The best way to explain what a Fourier Transform does is to apply it to a sound. Sound is vibrations travelling through the air that reach your air and make the inner part of your air vibrate. If you plot the air pressure reaching your air versus the time, the result will have the form of a sinoidal wave. However, it is only a straight forward sinoidal wave (as if you plotted the cos function) if the tone is pure. That is often not the case, and sounds are combinations of tones. This gives waves that are sinoidal but not very alike to the cos function. The FT will transform this "unpure" wave and splits them up into a set of waves that are all of pure tone. To do that we need complex numbers which are explained here subsection A.3.

With that explanation out of the way, we now know that with Euler's formula (Equation 4a) we can rotate on the complex plane. If we rotate one full circle per second, the formula changes to Equation 4b, as the circumference of the unit circle is 2π and t is in seconds. This rotates in the clock-wise direction, but we want to rotate in the clockwise direction, so we need to add a $-$ to the exponent. If we also want to control how fast the rotation happens (which is called the frequency) then we change the equation to Equation 4c. Note that the frequency unit is Hz which is defined as s^{-1}, which means that a frequency of $10Hz$ means 10 revolutions per second. Now we get our wave which we call $g(t)$ and plonk it in front of the equation up until now. Which results in Equation 4d. Visually, this means that we take the sound wave and wrap it around the origin. This might sound strange at first but bear with me. If you track the center of mass (the average of all the points that form the graph) you will notice that it hovers around the origin. If you now change the frequency of the rotation (f) you will see that the center of mass moves a bit, usually around the origin. However, if the frequency of the rotation matches a frequency of the wave, then the center of mass is suddenly a relatively long distance away from the origin. This indicates that we have found a frequency that composes the sound wave. Now how do we track the center of mass? That is done using integration, as in Equation 4e. Now to get to the final form, we forget about the fraction part. This means that the center of mass will still hover around the origin for the main part of the rotation, but has a huge value for when the rotation is at the same frequency as one of the waves in the sound wave. The larger the difference between t_2 and t_1, the larger the value of the Fourier Transform. The final equation is given in Equation 4f.

$$e^{ix} = \cos(x) + i\sin(x) \tag{4a}$$

$$e^{2\pi it} \tag{4b}$$

$$e^{-2\pi ift} \tag{4c}$$

$$g(t)e^{-2\pi ift} \tag{4d}$$

$$\frac{1}{t_2 - t_1}\int_{t_1}^{t_2} g(t)e^{-2\pi ift} \tag{4e}$$

$$\hat{g}(f) = \int_{t_1}^{t_2} g(t)e^{-2\pi ift} \tag{4f}$$

These Fourier Transforms have the great property that if you add them together you still have the relatively large distances of the center of mass to the origin at the original frequencies. It is this property that enables us to find the frequencies that compose a sound wave.

Before moving on we first need to discuss some important properties of FTs. Actually, these properties only apply to complex roots of unity (the $e^{2\pi ift}$ part is a complex root of unity). Due to the FT only being a coefficient $g(t)$ in front of a complex root of unity all these properties apply to FTs as well. Let us first start with the cancellation lemma as described in Lemma 1 [23]. Note that $\omega_n^k = e^{\frac{2\pi k}{n}}$ which is very similar to the form we discussed previously if you realise that $f = \frac{1}{T}$. So replace k by t and n by $T = \frac{1}{f}$ and you get the form we have discussed earlier.

Lemma 1 (Cancellation Lemma) *$\omega_{dn}^{dk} = \omega_n^k$, for all $k \geq 0, n \geq 0$ and $d > 0$.*

With that we also need to talk about the halving lemma (Lemma 2). This means that $(\omega_n^{k+\frac{n}{2}})^2 = (\omega_n^k)^2$.

Lemma 2 (Halving Lemma) *If $n > 0$ is even, then the squares of the n-complex n^{th} roots of unity are the $\frac{n}{2}$-complex $\frac{n}{2}^{th}$ roots of unity.*

Now that we know what a Fourier Transform is, we need to make it a Fast Fourier Transform, as you can imagine that calculating such a thing is quite difficult. Some smart people have thought about this and they came up with quite a fast algorithm, the Cooley-Tukey algorithm [22], named after the people that thought of it. They use something we know as a Discrete Fourier Transform which is described by **??**. Here N is the total amount of samples from the continuous sound wave. This means that $0 \leq k \leq N-1$ and $0 \leq n \leq N-1$.

Now with the DFT out of the way we can discuss the algorithm. Before we do, we assume that we have an input of n elements, where n is an exact power of 2. Meaning $n = 2^k$ for some integer k.

Algorithm 9: One dimensional Fast Fourier Transformation

Input : array A
Output: array B with length $A.length - 1$ containing the DFT
$n \leftarrow A.length$;
if $n = 1$ **then**
  return A ;
$\omega_n \leftarrow e^{\frac{2\pi i}{n}}$;
$\omega \leftarrow 1$;
$a^0 \leftarrow (A[0], A[2], \ldots, A[n-2])$;
$a^1 \leftarrow (A[1], A[3], \ldots, A[n-1])$;
$y^0 \leftarrow \texttt{FFT}(a^0)$;
$y^1 \leftarrow \texttt{FFT}(a^1)$;
for $k \leftarrow 0$ **to** $\frac{n}{2} - 1$ **do**
  $B[k] \leftarrow y^0[k] + \omega y^1[k]$;
  $B[k + \frac{n}{2}] \leftarrow y^0[k] - \omega y^1[k]$;
  $\omega \leftarrow \omega\omega_n$;
end
return B

There is just this one problem we have, the algorithm in algorithm 9 can only handle one dimension, and we need to support multidimensional arrays. So let us define a multidimensional FFT first in Equation 5. Here N and M are the amount of indices for the dimensions, where M are the amount of indices for the

first dimension and N are the amount of indices for the second dimension. This can of course be extended in the same way for a p-dimensional array.

$$X_{k,l} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{m,n} e^{-2i\pi(\frac{mk}{M} + \frac{nl}{N})} \tag{5}$$

It is at this point that the algorithm becomes very complicated. Therefore I would like to invite you to use a library for these kinds of calculations, like Numpy [38] for Python. If you really want to use your own made version, or if you want to understand how the libraries sort of do it, then you may continue. I still advise you to use a library, as other people have made the code for you.

We can calculate the result of such a multidimensional FFT by computing one-dimensional FFTs along each dimension in turn. Meaning we first calculate the result of the FFT along one dimension, then we do that for the second dimension and so forth. Now the order of calculating the FFTs along each dimension does not matter. So you can calculate the third dimension first, and the first dimension after that if you wish. This has the advantage that we can program a generic function without keeping track of the order of the dimensions. Now, the code is quite complicated but it boils down to this: calculate the FFT along one dimension, then repeat it for the second dimension and multiply the two together and keep repeating that for all dimensions.

With that out of the way, we need to create a smoothing operation out of it. We do this in algorithm 10. Keep in mind that `FFT` the call is to the multidimensional Fast Fourier Transform algorithm, `IFFT` the call to the inverse of the multidimensional Fast Fourier Transform algorithm (also on the TODO list) and that the $int()$ function ensures that the number in brackets is an integer. Also note that the inverse of the FFT might give complex answers, and we only want real answers which the *.real* ensures. We only take the real part and return that. v is an optional parameter with default value 0.5 which does absolutely nothing in this algorithm.

Algorithm 10: Smoothing function

Input : Array a, smoothing factor s, vertical smoothing factor $v \leftarrow 0.5$
Output: Array A with less variation
$nlat \leftarrow a.length$;
$nlon \leftarrow a[0].length$;
$nlevels \leftarrow a[0][0].length$;
$temp \leftarrow \texttt{FFT}(a)$;
$temp[int(nlats) : int(nlat(1-s)), :, :] \leftarrow 0$;
$temp[:, int(nlons) : int(nlon(1-s)), :] \leftarrow 0$;
$temp[:, :, int(nlevelsv) : int(nlevels(1-v))] \leftarrow 0$;
return $\texttt{IFFT}(temp).real$;

3 Radiation

Radiation is energy waves, some waves are visible like light, others are invisible like radio signals. As is the basis for physics, energy cannot be created nor destroyed, only changed from one form to another.

3.1 The First Law of Thermodynamics and the Stefan-Boltzmann Equation

If energy goes into an object it must equal the outflowing energy plus the change of internal energy. Which is exactly what happens with the atmosphere. Radiation from the sun comes in, and radiation from the atmosphere goes out. And along the way we heat the atmosphere and the planet which causes less radiation to be emitted than received. At least, that is the idea for Earth which may not apply to all planets. Let one thing be clear, more radiation cannot be emitted than is inserted, unless the planet and atmosphere are cooling. Anyway, we assume that the planet is a black body, i.e. it absorbs all radiation on all wavelengths. This is captured in Stefan-Boltzmann's law (Equation 6a) [39].

In Equation 6a the symbols are:

- S: The energy that reaches the top of the atmosphere, coming from the sun or a similar star, per second per meters squared (Wm^{-2}). This is also called the insolation.
- σ: The Stefan-Boltzmann constant, $5.670373 \cdot 10^{-}8$ ($\mathrm{Wm}^{-2}\mathrm{K}^{-4}$) [39].
- T: The temperature of the planet (K).

The energy difference between the energy that reaches the atmosphere and the temperature of the planet must be equal to the change in temperature of the planet, which is written in Equation 6b. The symbols on the right hand side are:

- ΔU: The change of internal energy (J) [42].
- C: The specific heat capacity of the object, i.e. how much energy is required to heat the object by one degree Kelvin (JK^{-1}).
- ΔT: The change in temperature (K).

We want to know the change of temperature ΔT, so we rewrite the equation into Equation 6c. Here we added the δt term to account for the time difference (or time step). This is needed as we need an interval to calculate the difference in temperature over. Also we need to get the energy that we get (J) and not the energy per second (W), and by adding this time step the units all match up perfectly.

$$S = SB = \sigma T^4 \tag{6a}$$

$$S - \sigma T^4 = \Delta U = C\Delta T \tag{6b}$$

$$\Delta T = \frac{\delta t(S - \sigma T^4)}{C} \tag{6c}$$

The set of equations in Equation 6 form the basis of the temperature exchange of the planet. However two crucial aspects are missing. Only half of the planet will be receiving light from the sun at once, and the planet is a sphere. So we need to account for both in our equation. We do that in Equation 7. We view the energy reacing the atmosphere as a circular area of energy, with the equation for the are of a circle being Equation 7a [5]. The area of a sphere is in Equation 7b [6]. In both equations, r is the radius of the circle/sphere. By using Equation 7a and Equation 7b in Equation 6c we get Equation 7c where r is replaced by R. It is common in physics literature to use capitals for large objects like planets. However we are not done yet since we can divide some stuff out. We end up with Equation 7d as the final equation we are going to use.

$$\pi r^2 \tag{7a}$$

$$4\pi r^2 \tag{7b}$$

$$\Delta T = \frac{\delta t(\pi R^2 S - 4\pi R^2 \sigma T^4)}{4\pi C R^2} \tag{7c}$$

$$\Delta T = \frac{\delta t(S - 4\sigma T^4)}{4C} \tag{7d}$$

3.2 Insolation

With the current equation we calculate the global average surface temperature of the planet itself. However, this planet does not have an atmosphere just yet. Basically we modelled the temperature of a rock floating in space, let's change that with Equation 8. Here we assume that the area of the atmosphere is equal to the area of the planet surface. Obviously this assumption is false, as the atmosphere is a sphere that is larger in radius than the planet, however the difference is not significant enough to account for it. We also define the atmosphere as a single layer. This is due to the accessibility of the model, we want to make it accessible, not university simulation grade. One thing to take into account for the atmosphere is that it only partially absorbs energy. The sun (or a similar star) is relatively hot and sends out energy waves (radiation) with relatively low wavelengths. The planet is relatively cold and sends out energy at long wavelengths. As a side note, all objects radiate energy. You can verify this by leaving something in the sun on a hot day for a while and almost touch it later. You can feel the heat radiating from the object. The planet is no exception and radiates heat as well, though at a different wavelength than the sun. The atmosphere absorbs longer wavelengths better than short wavelengths. For simplicity's sake we say that all of the sun's energy does not get absorbed by the atmosphere. The planet's radiation will be absorbed partially by the atmosphere. Some of the energy that the atmosphere absorbs is radiated into space and some of that energy is radiated back onto the planet's surface. We need to adjust Equation 7d to account for the energy being radiated from the atmosphere back at the planet surface.

So let us denote the temperature of the planet surface as T_p and the temperature of the atmosphere as T_a. Let us also write the specific heat capacity of the planet surface as C_p instead of C. We add the term in Equation 8b to Equation 7d in Equation 8c. In Equation 8a, ϵ is the absorbtivity of the atmosphere, the fraction of energy that the atmosphere absorbs. We divided Equation 8a by πR^2 as we did that with Equation 7d as well, so we needed to make it match that division.

$$4\pi R^2 \epsilon \sigma T_a^4 \tag{8a}$$

$$4\epsilon \sigma T_a^4 \tag{8b}$$

$$\Delta T_p = \frac{\delta t(S + 4\epsilon\sigma T_a^4 - 4\sigma T_p^4)}{4C_p} \tag{8c}$$

As you probably expected, the atmosphere can change in temperature as well. This is modelled by Equation 9, which is very similar to Equation 8c. There are some key differences though. Instead of subtracting the radiated heat of the atmosphere once we do it twice. This is because the atmosphere radiates heat into space and towards the surface of the planet, which are two outgoing streams of energy instead of one for the planet (as the planet obviously cannot radiate energy anywhere else than into the atmosphere). C_a is the specific heat capacity of the atmosphere.

$$\Delta T_a = \frac{\delta t(\sigma T_p^4 - 2\epsilon\sigma T_a^4)}{C_a} \tag{9}$$

3.3 The Latitude Longitude Grid

With the current model, we only calculate the global average temperature. To calculate the temperature change along the surface and atmosphere at different points, we are going to use a grid. Fortunately the world has already defined such a grid for us, the latitude longitude grid [26]. The latitude is the coordinate running from the south pole to the north pole, with -90 being the south pole and 90 being the north pole. The longitude runs parallel to the equator and runs from 0 to 360 which is the amount of degrees that an angle can take when calculating the angle of a circle. So 0 degrees longitude is the same place as 360 degrees longitude. To do this however we need to move on from mathematical formulae to code (or in this case pseudocode).

Pseudocode is a representation of real code. It is meant to be an abstraction of code such that it does not matter how you present it, but every coder should be able to read it and implement it in their language of preference. This is usually easier to read than normal code, but more difficult to read than mathematical formulae. If you are unfamiliar with code or coding, look up a tutorial online as there are numerous great ones.

The pseudocode in algorithm 11 defines the main function of the radiation part of the model. All values are initialised beforehand, based on either estimations, trial and error or because they are what they are (like the Stefan-Boltzmann constant σ), which is all done in section 1. The total time t starts at 0 and increases by δt after every update of the temperature. This is to account for the total time that the model has simulated (and it is also used later). What you may notice is the $T_p[lan, lon]$ notation. This is to indicate that T_p saves a value for each lan and lon combination. It is initialised as all zeroes for each index pair, and the values is changed based on the calculations. You can view T_p like the whole latitude longitude grid, where $T_p[lat, lon]$ is an individual cell of that grid indexed by a specific latitude longitude combination. Do note that from here on most, if not all functions need to be called [1] from another file which I will call the master. The master file decides what parts of the model to use, what information it uses for plots and the like. We do it this way because we want to be able to switch out calculations. Say that I find a more efficient way, or more detailed way, to calculate the temperature change. If everything was in one file, then I need to edit the source code of the project. With the master file structure, I can just swap out the reference to the project's implementation with a reference to my own implementation. This makes the life of the user (in this case the programmer who has another implementation) easier and makes changing calculations in the future easier as well. Also note that what we pass on as parameters [2] in algorithm 11 are the things that change during the execution of the model or that are calculated beforehand and not constants. S for instance is not constant (well at this point it is but in subsection 3.4 we change that) amd the current time is obviously not constant. All constants can be found in section 1.

Algorithm 11: The main function for the temperature calculations

Input: time t, amount of energy that hits the planet S
Output: Temperature of the planet T_p, temperature of the atmosphere T_a
for $lat \leftarrow -90$ **to** 90 **do**
  for $lon \leftarrow 0$ **to** 360 **do**
    $T_p[lat, lon] \leftarrow T_p[lat, lon] + \frac{\delta t(S+4\epsilon\sigma(T_a[lat,lon])^4-4\sigma(T_p[lat,lon])^4)}{C_p}$;
    $T_a[lat, lon] \leftarrow T_a[lat, lon] + \frac{\delta t(\sigma(T_p[lat,lon])^4-2\epsilon\sigma(T_a[lat,lon])^4)}{C_a}$;
  end
end

3.4 Day/Night Cycle

As you can see, the amount of energy that reaches the atmopsphere is constant. However this varies based on the position of the sun relative to the planet. To fix this, we have to assign a function to S that gives

[1] In case you are unfamiliar with calls, defining a function is defining how it works and calling a function is actually using it.
[2] Parameters are variables that a function can use but are defined elsewhere. The real values of the variables are passed on to the funciton in the call.

the correct amount of energy that lands on that part of the planet surface. This is done in algorithm 12. In this algorithm the term insolation is mentioned, which is S used in the previous formulae if you recall. We use the cos function here to map the strength of the sun to a number between 0 and 1. The strength is dependent on the latitude, but since that is in degrees and we need it in radians we transform it to radians by multiplying it by $\frac{\pi}{180}$. This function assumes the sun is at the equinox (center of the sun is directly above the equator) [31] at at all times. The second cos is needed to simulate the longitude that the sun has moved over the longitude of the equator. For that we need the difference between the longitude of the point we want to calculate the energy for, and the longitude of the sun. The longitude of the sun is of course linked to the current time (as the sun is in a different position at 5:00 than at 15:00). So we need to map the current time in seconds to the interval [0, seconds in a day]. Therefore we need the mod function. The mod function works like this: x mod y means subtract all multiples of y from x such that $0 \leq x < y$. So to map the current time to a time within one day, we do $-t$ mod d where $-t$ is the current time and d is the amount of seconds in a day. We need $-t$ as this ensures that the sun moves in the right direction, with t the sun would move in the opposite direction in our model than how it would move in real life. When we did the calculation specified in algorithm 12 we return the final value (which means that the function call is "replaced" [3] by the value that the function calculates). If the final value is less than 0, we need to return 0 as the sun cannot suck energy out of the planet (that it does not radiate itself, which would happen if a negative value is returned).

Algorithm 12: Calculating the energy from the sun (or similar star) that reaches a part of the planet surface at a given latitude and time

Input: insolation ins, latitude lat, longitude lon, time t, time in a day d
Output: Amount of energy S that hits the planet surface at the given latitude-time combination.
$longitude \leftarrow 360 \cdot \frac{(-t \bmod d)}{d}$;
$S \leftarrow ins \cdot \cos(lat\frac{\pi}{180}) \cos((lon - longitude) \cdot \frac{\pi}{180})$;
if $S < 0$ **then**
  return 0
else
  return S
end

By implementing algorithm 12, algorithm 11 must be changed as well, as S is no longer constant for the whole planet surface. So let us do that in algorithm 13. Note that S is defined as the call to algorithm 12.

Algorithm 13: The main function for the temperature calculations

Input: amount of energy that hits the planet S
Output: Temperature of the planet T_p, temperature of the atmosphere T_a
for $lat \leftarrow -nlat$ **to** $nlat$ **do**
  for $lon \leftarrow 0$ **to** $nlot$ **do**
    $T_p[lat, lon] \leftarrow T_p[lat, lon] + \frac{\delta t(S+4\epsilon\sigma(T_a[lat,lon])^4-4\sigma(T_p[lat,lon])^4)}{C_p}$;
    $T_a[lat, lon] \leftarrow T_a[lat, lon] + \frac{\delta t(\sigma(T_p[lat,lon])^4-2\epsilon\sigma(T_a[lat,lon])^4)}{C_a}$;
  end
end

3.5 Albedo

Albedo is basically the reflectiveness of a material (in our case the planet's surface) [2]. The average albedo of the Earth is about 0.2. Do note that we change C_p from a constant to an array. We do this to allow

[3] Replaced is not necessarily the right word, it is more like a mathematical function $f(x)$ where $y = f(x)$. You give it an x and the value that correpsonds to that x is saved in y. So you can view the function call in pseudocode as a value that is calculated by a different function which is then used like a regular number.

adding in oceans or other terrain in the future. Same thing for the albedo, different terrain has different reflectiveness.

Algorithm 14: Defining albedo

$a \leftarrow 0.2$;
$C_p \leftarrow 10^6$;

Now that we have that defined, we need to adjust the main loop of the program (algorithm 13). We need to add albedo into the equation and change C_p from a constant to an array. The algorithm after these changes can be found in algorithm 15. We multiply by $1 - a$ since albedo represents how much energy is reflected instead of absorbed, where we need the amount that is absorbed which is exactly equal to 1 minus the amount that is reflected.

Algorithm 15: The main function for the temperature calculations

Input: amount of energy that hits the planet S
Output: Temperature of the planet T_p, temperature of the atmosphere T_a
for $lat \leftarrow -nlat$ **to** $nlat$ **do**
  for $lon \leftarrow 0$ **to** $nlot$ **do**
    $T_p[lat, lon] \leftarrow T_p[lat, lon] + \frac{\delta t((1-a[lat,lon])S+4\epsilon\sigma(T_a[lat,lon])^4-4\sigma(T_p[lat,lon])^4)}{C_p[lat,lon]}$;
    $T_a[lat, lon] \leftarrow T_a[lat, lon] + \frac{\delta t(\sigma(T_p[lat,lon])^4-2\epsilon\sigma(T_a[lat,lon])^4)}{C_a}$;
  end
end

3.6 Temperature with Varying Density

The air density is not at all points exactly the same. This may be due to the winds blowing, or due to height changes in the terrain. We need to account for that, which is done in algorithm 16.

Algorithm 16: The main function for the temperature calculations

Input: amount of energy that hits the planet S
Output: Temperature of the planet T_p, temperature of the atmosphere T_a
for $lat \leftarrow -nlat$ **to** $nlat$ **do**
  for $lon \leftarrow 0$ **to** $nlot$ **do**
    $T_p[lat, lon] \leftarrow T_p[lat, lon] + \frac{\delta t((1-a[lat,lon])S+4\epsilon\sigma(T_a[lat,lon])^4-4\sigma(T_p[lat,lon])^4)}{\rho[lat,lon]C_p[lat,lon]}$;
    $T_a[lat, lon] \leftarrow T_a[lat, lon] + \frac{\delta t(\sigma(T_p[lat,lon])^4-2\epsilon\sigma(T_a[lat,lon])^4)}{\rho[lat,lon]C_a}$;
  end
end

3.7 Adding Layers

To add layers, we need a vertical coordinate. You would think that height is the logical and obvious choice right? You are right, but also quite wrong. We instead are going to use pressure. The reason for this is quite simple: vertical advection (see section 5) when using height as vertical coordinate will not work, whereas using pressure allows vertical advection to work. Therefore we will use pressure as the vertical coordinate. This makes sense too, as there is less pressure at the top of the atmosphere than at the bottom of the atmosphere.

Using pressure as your vertical coordinate does have some effect on the temperature. Instead of using the temperature, we will need to use potential temperature (as explained in subsection 5.4). Therefore you

will need to convert from and back to temperature when reading in and outputting thermal data. A benefit is though that we do not need to keep track of the air density anymore, as that is incorporated into the potential temperature. To avoid confusion with the previously defined temperature of the planet T_p we will from now on refer to the potential temperature as T_{pot}. The initial temperature of the atmosphere (which is read in in subsection 7.4) is henceforth referred to as T_i.

3.8 Grey Radiation Scheme

Inspired by the Isca project [37] and a paper describing the grey radiation scheme [24].

A radiation scheme is a model for how energy is redistributed using light in a system. Such a model is a Grey radiation scheme if you split it into two parts, short and long wavelength radiation. So you have two redistribution systems, one for short wavelength light and one for long wavelength light. Another assumption we make when using the Grey radiation scheme, is that the atmosphere is transparent to short wavelength radiation, meaning it lets through light with short wavelengths. Additionally we use a two stream approximation, which means that we have a stream of radiation going up, and another stream of radiation going down. Note that these two streams are both long wavelength radiation, because we said earlier we assume the atmosphere completely ignores short wavelength radiation.

The two long wavelength radiation streams are described in Equation 10a and Equation 10b [24]. In those equations, the symbols are:

- U: Upward flux.
- D: Downward flux.
- B: The Stefan-Boltzmann equation (see Equation 6a).
- τ: Optical depth.

$$\frac{dU}{d\tau} = U - B \tag{10a}$$

$$\frac{dD}{d\tau} = B - D \tag{10b}$$

With Equation 10a and Equation 10b written down, we can discuss how they work. These equations need a boundary condition to work, a starting point if you like. For those equations the boundary conditions are that U is at the surface equal to B and that D at the top of the atmosphere is equal to 0. Meaning that in the beginning the top of the atmosphere has no downward flux as there is no heat there, and that the bottom of the atmosphere has a lot of upward flux as most if not all of the heat is located there. Then after the spin up time this should stabilise. We are interested in the change of the fluxes, so dU and dD, to get those we need to multiply the right hand side by $d\tau$. Then we have the flow of radiation that we need. However we cannot solely use these two equations to calculate the heat of a given layer. For that we need a few more components. These are described in Equation 11. Here Q_R is the amount of heat in a layer, c_p is the specific heat capacity of dry air (our atmosphere), ρ is the density of the air in that layer and δz is the change in height. $\delta U - D$ are the change in net radiation, meaning the amount of radiation that is left over after you transferred the upward and downward flux. See it as incoming and outgoing energy for a given layer, the net change (either cooling down or heating up) is what remains after you have subtracted the incoming energy from the outgoing energy. While this explanation is not entirely true (as flux is not entirely equivalent to energy), it explains the concept the best.

$$Q_R = \frac{1}{c_p \rho} \frac{\delta(U - D)}{\delta z} \tag{11}$$

Now only one question remains: what is optical depth? Optical depth is the amount of work a photon has had to do to get to a certain point. This might sound really vague, but bear with me. Optical depth describes how much stuff a certain photon has had to go through to get to a point. As you'd expect this is 0 at the top of the atmosphere as space is a big vacuum so no stuff to move through, so no work. Then the

further the photon moves into the atmosphere, the more work the photon has had to do to get there. This is because it now needs to move through gases, like air, water vapour and other gases. Hence the closer the photon gets to the surface of the planet, the larger the optical depth is because the photon has had to work more to get there. This phenomenon is described in Equation 12. The symbols in the equation mean:

- τ_0: Optical depth at the surface.
- p: Atmospheric pressure (Pa).
- p_s: Atmospheric pressure at the surface (Pa).
- f_l: The linear optical depth parameter, with a value of 0.1.

$$\tau = \tau_0[f_l(\frac{p}{p_s}) + (1 - f_l)(\frac{p}{p_s})^4] \tag{12}$$

As one can see, Equation 12 has two parts, a linear part and a quatric part (to the power 4). The quatric term approximates the structure of water vapour in the atmosphere, which roughly scales with $\frac{1}{4}$ with respect to the height. The linear term is present to fix numerical behaviour because this is an approximation which will not be completely correct (that's why it is an approximation) so we add this term to make it roughly right. The same thing holds for f_l which can be manually tuned to fix weird numerical behaviour.

With these equations in our mind, let's get coding. First we add the pressure profile, the pressure of all atmospheric layers at a lat lon point. We need this to accurately represent the optical depth per atmospheric layer. Then we need to use the pressure profile with regards to Equation 12. The resulting code can be found in algorithm 17. This algorithm replaces the temperature calculations we have done in algorithm 62, as this is basically a better version of the calculations done in that algorithm. f_l has a value of 0.1 and is located near all the other constants in the code, henceforth we will refer to this section in the code as the control panel, since most if not all of the constants can be tweaked here. τ_0 is a function that gives the surface optical depth for a given latitude. The corresponding equation can be found in Equation 13 [17]. Translating this into code is left as an exercise to the reader. $U[0]$ is the boundary condition discussed before (being the same as Equation 6a), just as $D[nlevels]$ is the boundary condition. S_z represents the call to algorithm 6. `solar` represents the call to algorithm 12. T_{trans} represents the call to algorithm 29.

$$\tau_0 = 3.75 + \cos(lat\frac{\pi}{90})\frac{4.5}{2} \tag{13}$$

3.9 Adding In Some Ozone (Or Something Else That Approximates It)

Adding in ozone in the stratosphere is hella complicated, so we leave that as an exercise to the reader as in true academic fashion. Just joking, if you want you can work on implementing ozone however we opt not to because it is quite complicated. Instead we approximate it, which is decent enough for our purpose. We need to do it in algorithm 17 as we need to adjust the Q. We add in a check to see if we are currently calculating the radiation in the stratosphere. If so we add some radiation extra to replicate the effect of ozone. As can be seen in algorithm 18, where we only focus on the Q part of algorithm 17, we add in some extra radiation based on how high the current layer calculation is, which scales with the height.

3.10 Tilting the Planet

In order to model a planet that has seasons, like Earth, we need to tilt the planet. This has as effect that the sun is not always directly above the equator but is above a certain band around the equator as the year moves on. This means that some hemispheres receive more/less sun based on what part of the year it is. Which corresponds to the various seasons we have on Earth. But in order to do that, we have to change the `solar` function. The new version as shown in algorithm 19 will replace algorithm 12. Here α is the tilt in degrees.

What the code in algorithm 19 does boils down to calculating the latitude and longitude of the sun and checking whether the planet receives any energy. If not return 0 immediately. If so we check if the difference

Algorithm 17: Main function for calculating the temperature using radiation

Input: amount of energy that hits the planet S
Output: Potential temperature T_{pot}, temperature of the planet surface T_p

$T_a \leftarrow T_{trans}(T_{pot}, p_z, \texttt{False})$;
for $lat \leftarrow -nlat$ **to** $nlat$ **do**
  $\tau = \tau_0(lat) f_l \frac{p_z}{p_z[0]} + (1 - f_l)(\frac{p_z}{p_z[0]})^4)$;
  for $lon \leftarrow 0$ **to** $nlot$ **do**
    $U[0] \leftarrow \sigma T_p[lat, lon]^4$;
    for $level \leftarrow 1$ **to** $nlevels$ **do**
      $U[level] \leftarrow U[level-1] - \frac{(\tau[level]-\tau[level-1])(\sigma \cdot (mean(T_a[:,:,level]))^4)}{1+(\tau[level-1]-\tau[level])}$;
    end
    $D[nlevels-1] \leftarrow 0$;
    for $level \leftarrow nleves - 1$ **to** 0 **do**
      $D[level] \leftarrow D[level+1] - \frac{(\tau[level+1]-\tau[level])(\sigma \cdot (mean(T_a[:,:,level]))^4)}{1+(\tau[level]-\tau[level+1])}$;
    end
    for $level \leftarrow 0$ **to** $nlevels$ **do**
      $Q[level] \leftarrow -287 T_a[lat, lon, level] \frac{S_z(U-D, p_z, level)}{10^3 \cdot p_z[level]}$;
    end
    $T_a[lat, lon, :] \leftarrow T_a[lat, lon, :] + Q$;
    $S \leftarrow \texttt{solar}(I, lat, lon, t)$;
    $T_p[lat, lon] \leftarrow T_p[lat, lon] \frac{\delta t((1-a[lat,lon])S + (D[0]-U[0]))}{C_p[lat,lon]}$;
  end
end
$T_{pot} \leftarrow T_{trans}(T_a, p_z, \texttt{True})$;
return T_p, T_{pot}

Algorithm 18: Replicating the effect of ozone

$inv \leftarrow \frac{1}{24*60*60}$;
for $level \leftarrow 0$ **to** $nlevels$ **do**
  $Q[level] \leftarrow -287 T_a \frac{S_z(U-D, 0, level)}{10^3 \cdot p_z[level]}$;
  if $p_z[level] < 40000$ **then**
    $Q[level] \leftarrow Q[level] + \texttt{solar}(5, lat, lon, t) inv(\frac{100}{p_z[level]})$;
end

Algorithm 19: Calculating the energy from the sun (or similar star) that reaches a part of the planet surface at a given latitude and time

Input: insolation ins, latitude lat, longitude lon, time t, time in a day d
Output: Amount of energy S that hits the planet surface at the given latitude, longitude and time combination.

$sun\_lon \leftarrow -t \bmod d$;
$sun\_lon \leftarrow sun\_lon \cdot \frac{360}{d}$;
$sun\_lat \leftarrow \alpha \cos(\frac{2t\pi}{year})$;
$S \leftarrow insolation \cos(\frac{\pi(lat - sun\_lat)}{180})$;
if $S < 0$ **then**
    return 0 ;
else
    $lon\_diff \leftarrow lon - sun\_lon$;
    $S \leftarrow S \cos(\frac{lon\_diff\pi}{180})$;
    if $S < 0$ **then**
        if $lat + sun\_lat > 90$ *or* $lat + sun\_lat < -90$ **then**
            return $insolation \cos(\frac{\pi(lat+sun\_lat)}{180}) \cos(\frac{lon\_diff\pi}{180})$;
        else
            return 0 ;
    else
        return S ;

between the sun's longitude and the planet's longitude and calculate how much energy would hit the planet given that the sun is not straight above the equator. We do this by multiplying the energy it would receive from the sun if it were above the equator S by the cosine of the difference in longitudes, which represents the tilt. Then we check again if the planet is receiving energy, if not we check if it happens around the poles. We do this because due to the tilt it can be the case that at certain points in the year the pole is in constant sunlight, i.e. the sun does not go down. This creates a sort of overshoot which needs to be accounted for. If it does this then we add the latitudes of the sun and the planet together and use that to calculate the energy that would hit that spot. If it is not the case that we are around the poles and we do not receive energy, then we return 0. If it happens to be that we do receive energy (so no negative values) then we return S.

4 Air Velocity

Did you ever feel the wind blow? Most probably. That's what we will be calculating here. How hard the wind will blow. This is noted as velocity, how fast something moves.

4.1 Equation of State and the Incompressible Atmosphere

The equation of state relates one or more variables in a dynamical system (like the atmosphere) to another. The most common equation of state in the atmosphere is the ideal gas equation as described by Equation 14a [40]. The symbols in that equation represent:

- p: The gas pressure (Pa).
- V: The volume of the gas (m^3).
- n: The amount of moles in the gas (mol).
- R: The Gas constant as defined in subsubsection 1.2.1 ($JK^{-1}mol^{-1}$) [40].
- T: The temperature opf the gas (K).

If we divide everything in Equation 14a by V and set it to be unit (in this case, set it to be exactly 1 m^3) we can add in the molar mass in both the top and bottom parts of the division as show in Equation 14b. We can then replace $\frac{nm}{V}$ by ρ the density of the gas (kgm^{-3}) and $\frac{R}{m}$ by R_s the specific gas constant (gas constant that varies per gas in $JK^{-1}mol^{-1}$) as shown in Equation 14c. The resulting equation is the equation of state that you get that most atmospheric physicists use when talking about the atmosphere [17].

$$pV = nRT \tag{14a}$$

$$p = \frac{nR}{V}T = \frac{nmR}{Vm}T \tag{14b}$$

$$p = \rho R_s T \tag{14c}$$

The pressure is quite important, as air moves from a high pressure point to a low pressure point. So if we know the density and the temperature, then we know the pressure and we can work out where the air will be moving to (i.e. how the wind will blow). In our current model, we know the atmospheric temperature but we do not know the density. For simplicities sake, we will now assume that the atmosphere is Incompressible, meaning that we have a constant density. Obviously we know that air can be compressed and hence our atmosphere can be compressed too but that is not important enough to account for yet, especially considering the current complexity of our model.

The code that corresponds to this is quite simple, the only change that we need to make in Equation 14c is that we need to replace T by T_a, the temperature of the atmosphere. As T_a is a matrix (known to programmers as a double array), p will be a matrix as well. Now we only need to fill in some values. $\rho = 1.2$ [43], $R_s = 287$ [32].

4.2 The Momentum Equations

The momentum equations are a set of equations that describe the flow of a fluid on the surface of a rotating body. For our model we will use the f-plane approximation. The equations corresponding to the f-plane approximation are given in Equation 15a and Equation 15b [33]. Note that we are ignoring vertical movement, as this does not have a significant effect on the whole flow. All the symbols in Equation 15a and Equation 15b mean:

- u: The east to west velocity (ms^{-1}).
- t: The time (s).

- f: The coriolis parameter as in Equation 31a.
- v: The north to south velocity (ms^{-1}).
- ρ: The density of the atmosphere (kgm^{-3}).
- p: The atmospheric pressure (Pa).
- x: The local longitude coordinate (m).
- y: The local latitude coordinate (m).

If we then define a vector $\bar{u}$ as $(u, v, 0)$, we can rewrite both Equation 15a as Equation 15c. Here ∇u is the gradient of u in both x and y directions. Then if we write out ∇u we get Equation 15e. Similarly, if we want to get ∂v instead of ∂u we rewrite Equation 15b to get Equation 15d and Equation 15f.

$$\frac{Du}{Dt} - fv = -\frac{1}{\rho}\frac{\partial p}{\partial x} \tag{15a}$$

$$\frac{Dv}{Dt} - fu = -\frac{1}{\rho}\frac{\partial p}{\partial y} \tag{15b}$$

$$\frac{\partial u}{\partial t} + \bar{u} \cdot \nabla u - fv = -\frac{1}{\rho}\frac{\partial p}{\partial x} \tag{15c}$$

$$\frac{\partial v}{\partial t} + \bar{u} \cdot \nabla v - fu = -\frac{1}{\rho}\frac{\partial p}{\partial y} \tag{15d}$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - fv = -\frac{1}{\rho}\frac{\partial p}{\partial x} \tag{15e}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} - fu = -\frac{1}{\rho}\frac{\partial p}{\partial y} \tag{15f}$$

With the gradient functions defined in algorithm 4 and algorithm 5, we can move on to the main code for the momentum equations. The main loop is shown in algorithm 20. Do note that this loop replaces the one in algorithm 66 as these calculate the same thing, but the new algorithm does it better.

Algorithm 20: Calculating the flow of the atmosphere (wind)

$S_{xu} \leftarrow$ `gradient_x`(u, lat, lon) ;
$S_{yu} \leftarrow$ `gradient_y`(u, lat, lon) ;
$S_{xv} \leftarrow$ `gradient_x`(v, lat, lon) ;
$S_{yv} \leftarrow$ `gradient_y`(v, lat, lon) ;
$S_{px} \leftarrow$ `gradient_x`(p, lat, lon) ;
$S_{py} \leftarrow$ `gradient_x`(p, lat, lon) ;
for $lat \leftarrow 1$ **to** $nlat - 1$ **do**
  for $lon \leftarrow 0$ **to** $nlon$ **do**
    $u[lat, lon] \leftarrow u[lat, lon] + \delta t \frac{-u[lat,lon]S_{xu} - v[lat,lon]S_{yu} + f[lat]v[lat,lon] - S_{px}}{\rho}$;
    $v[lat, lon] \leftarrow v[lat, lon] + \delta t \frac{-u[lat,lon]S_{xv} - v[lat,lon]S_{yv} - f[lat]u[lat,lon] - S_{py}}{\rho}$;
  end
end

4.3 Improving the Coriolis Parameter

Another change introduced is in the coriolis parameter. Up until now it has been a constant, however we know that it varies along the latitude. So let's make it vary over the latitude. Recall Equation 31a, where Θ is the latitude. Coriolis (f) is currently defined in algorithm 65, so let's replace it with algorithm 21.

Algorithm 21: Calculating the coriolis force

$\Omega \leftarrow 7.2921 \cdot 10^{-5}$;
for $lat \leftarrow -nlat$ **to** $nlat$ **do**
  $f[lat] \leftarrow 2\Omega \sin(lat\frac{\pi}{180})$;
end

4.4 Adding Friction

In order to simulate friction, we multiply the speeds u and v by 0.99. Of course there are equations for friction but that gets complicated very fast, so instead we just assume that we have a constant friction factor. This multiplication is done directly after algorithm 20 in algorithm 54.

4.5 Adding in Layers

With adding in atmospheric layers we need to add vertical winds, or in other words add the w component of the velocity vectors. We do that by editing algorithm 20. We change it to algorithm 23. Here we use gravity (g) instead of the coriolis force (f) and calculate the change in pressure. Therefore we need to store a copy of the pressure before we do any calculations. This needs to be a copy due to aliasing [4]. Since we use pressure as the vertical coordinate, we must be able to convert that into meters (why we opted for pressure is explained in subsection 3.7) in order to be able to say something sensible about it. To do that we need the concept of geopotential height.

4.5.1 Dimensionless Pressure

Geopotential height is similar to geometric height, except that it also accounts for the variation in gravity over the planet [1]. One could say that geopotential height is the "gravity adjusted" height. That means that it is similar to the height, but not exactly the same. Height is a linear function, whereas the geopotential height is not, though it is very similar to a linear function if you would plot it. Now to convert easily to and from potential temperature into temperature, we need another function which is known as the Exner function. The Exner function is a dimensionless [5] pressure. The Exner function is shown in Equation 16a [7]. The symbols in the equation are:

- c_p: The specific heat capacity of the atmosphere.
- p: Pressure (Pa).
- p_0: Reference pressure to define the potential temperature (Pa).
- R: The gas constant 8.3144621 (J(mol)$^{-1}$K).
- T: The absolute temperature (K).
- θ: the potential temperature (K).

Since the right hand side contains what we want to convert to and from, we can do some basic rewriting, which tells us what we need to code to convert potential temperature in absolute temperature and vice versa. This is shown in Equation 16b and Equation 16c respectively.

$$\Pi = c_p(\frac{p}{p_0})^{\frac{R}{c_p}} = \frac{T}{\theta} \tag{16a}$$

[4] Aliasing is assigning a different name to a variable, while it remains the same variable. Take for instance that we declare a variable x and set it to be 4. Then we say $y \leftarrow x$, which you might think is the same as saying they $y \leftarrow 4$ but behind the screen it is pointing to x. So if x changes, then so does y.

[5] Being dimensionless means that there is no dimension (unit) attached to the number. This is useful for many applications and is even used in daily life. For instance when comparing price rises of different products, it is way more useful to talk about percentages (who are unitless) instead of how much you physically pay more (with your favourite currency as the unit).

$$T = \Pi\theta \tag{16b}$$

$$\theta = \frac{T}{\Pi} \tag{16c}$$

Now onto some code. Let us initialise Π before we do any other calculations. This code is already present in the control panel section (section 1) as that is where it belongs, so for further details please have a look at the code there. Now onto the geopotential height.

4.5.2 Geopotential Height

As stated before, geopotential height is similar to geometric height, except that it also accounts for the variation in gravity over the planet. One could say that geopotential height is the "gravity adjusted" height. That means that it is similar to the height, but not exactly the same. Height is a linear function, whereas the geopotential height is not, though it is very similar to a linear function if you would plot it. Now one could ask why we would discuss dimensionless pressure before geopotential height. The answer is quite simple, in order to define geopotential height, we need the Exner function to define it. Or rather, we need that function to convert potential temperature into geopotential height. How those three are related is shown in Equation 17a. Then with a little transformation we can define how the geopotential height will look like, as shown in Equation 17b. The symbols in both equations are:

- Π: The Exner function.
- θ: Potential temperature (K).

$$\theta + \frac{\delta\Phi}{\delta\Pi} = 0 \tag{17a}$$

$$\delta\Phi = -\theta\delta\Pi \tag{17b}$$

Now to turn this into code we need to be careful about a few things. First we are talking about a change in geopotential height here, so defining one level of geopotential height means that it is dependent on the level below it. Second this calculation needs potential temperature and therefore it should be passed along to the velocity calculations function. With those two things out of the way, we get the code as shown in algorithm 22. Note that `Smooth3D` refers to algorithm 10.

Algorithm 22: Calculating the geopotential height

for $level \leftarrow 1$ **to** $nlevels$ **do**
  $\Phi[:,:,level] \leftarrow \Phi[:,:,level-1] - T_{pot}[:,:,level](\Pi[level] - \Pi[level-1])$;
end
$\Phi \leftarrow$ `Smooth3D`$(\Phi, smooth_t)$;

4.5.3 Finally Adding in the Layers

Now with the geopotential height and dimensionless pressure out of the way, we need to use those two concepts to add layers to the velocity calculations. Before we dive into the code however, there are slight changes that we need to discuss. The equation shown in Equation 18a is the primitive equation (as discussed in subsubsection B.2.1). The momentum equations are gesostrphic momentum, which are a special form of the primitive equation. Since this whole system must remain in equilibrium, we need to set the right hand side to 0 as shown in Equation 18b. Now let us rewrite Equation 18a into Equation 18c. We replaze z with pressure as that is our vertical coordinate. ω is the velocity of the pressure field, as defined in Equation 18d. Note that p_k is the pressure for layer k and p_0 is the pressure at the planet surface. Now we need to turn the

velocity of the pressure field into the velocity of a packet of air (see it as a box of air being moved), which is done in Equation 18e. Here ρ is the density and g is gravity (ms^{-2}).

$$\frac{\delta T}{\delta x} + \frac{\delta T}{\delta y} + \frac{\delta T}{\delta z} = \nabla T \tag{18a}$$

$$\nabla T = 0 \tag{18b}$$

$$\frac{\delta u}{\delta x} + \frac{\delta v}{\delta y} + \frac{\delta \omega}{\delta p} = 0 \tag{18c}$$

$$\omega_k = -\int_{p_0}^{p_k} \frac{\delta u}{\delta x} + \frac{\delta v}{\delta y} dp \tag{18d}$$

$$w = \omega \rho g \tag{18e}$$

But I hear you say, what is the density? Since we have moved to pressure coordinates we can actually calculate the density rather than store it. This is done in Equation 19, where each symbol means:

- ρ: The density of the atmosphere.
- p: The pressure of the atmosphere (Pa).
- c: Specific heat capacity of the atmosphere ($\mathrm{JKg}^{-1}\mathrm{K}^{-1}$).
- T: Temperature of the atmosphere (K).

$$\rho = \frac{p}{cT} \tag{19}$$

Finally, let us convert Equation 18e to code in algorithm 23. Here T_{trans} is a call to the algorithm as described in algorithm 29. `gradient_x`, `gradient_y` and `gradient_z` are calls to algorithm 4, algorithm 5 and algorithm 6 respectively.

Algorithm 23: Calculating the flow of the atmosphere (wind)

//`The following variables are function calls to algorithms and their shorthand notations are used in the loops`
$S_{xu} \leftarrow$ `gradient_x`$(u, lat, lon, layer)$;
$S_{yu} \leftarrow$ `gradient_y`$(u, lat, lon, layer)$;
$S_{xv} \leftarrow$ `gradient_x`$(v, lat, lon, layer)$;
$S_{yv} \leftarrow$ `gradient_y`$(v, lat, lon, layer)$;
$S_{zu} \leftarrow$ `gradient_z`$(u[lat, lon], p_z, layer)$;
$S_{zv} \leftarrow$ `gradient_z`$(v[lat, lon], p_z, layer)$;
$S_{px} \leftarrow$ `gradient_x`$(geopot, lat, lon, layer)$;
$S_{py} \leftarrow$ `gradient_y`$(geopot, lat, lon, layer)$;
//`The following variables are real variables`
$nlat \leftarrow \Phi.length$;
$nlon \leftarrow \Phi[0].length$;
$u_t \leftarrow$ array like u ;
$v_t \leftarrow$ array like v ;
$w_t \leftarrow$ array like w ;
for $lat \leftarrow 1$ **to** $nlat - 1$ **do**
  for $lon \leftarrow 0$ **to** $nlon$ **do**
    for $layer \leftarrow 0$ **to** $nlevels$ **do**
      $u_t[lat, lon, layer] \leftarrow u[lat, lon, layer] + \delta t \frac{-u[lat,lon,layer]S_{xu} - v[lat,lon,layer]S_{yu} + f[lat]v[lat,lon,layer] - S_{px}}{10^5 u[lat,lon,layer]}$;
      $v_t[lat, lon, layer] \leftarrow v[lat, lon, layer] + \delta t \frac{-u[lat,lon,layer]S_{xv} - v[lat,lon,layer]S_{yv} - f[lat]u[lat,lon,layer] - S_{py}}{10^5 v[lat,lon,layer]}$;
    end
  end
end
$T_a \leftarrow T_{trans}(T_{pot}, p_z,$ `False`$)$;
for $lat \leftarrow 2$ **to** $nlat - 2$ **do**
  for $lon \leftarrow 0$ **to** $nlon$ **do**
    for $level \leftarrow 1$ **to** $nlevels$ **do**
      $w_t[lat, lon, level] \leftarrow w_t[lat, lon, level - 1] - \frac{(p_z[level] - p_z[level-1])p_z[level]g(S_{xu} + S_{yv})}{C_p T_a[lat,lon,layer]}$;
    end
  end
end
$u \leftarrow u + u_t$;
$v \leftarrow v + v_t$;
$w \leftarrow w + w_t$;
$p_0 \leftarrow copy(p)$;

5 Advection

Advection is a fluid flow transporting something with it as it flows. This can be temperature, gas, solids or other fluids. In our case we will be looking at temperature.

5.1 Thermal Diffusion

As of this time, what you notice if you run the model is that the winds only get stronger and stronger (and the model is hence blowing up, which means that the numbers increase so dramatically that it is no longer realistic). This is because there is no link yet between the velocities of the atmosphere and the temperature. Currently, any air movement does not affect the temperature in the atmosphere of our model while it does in reality. So we need to change some calculations to account for that. Thermal diffusion helps with spreading out the temperatures and tempering the winds a bit.

The diffusion equation, as written in Equation 20, describes how the temperature spreads out over time [34]. The symbols in the equation represent:

- u: A vector consisting out of 4 elements: x, y, z, t. x, y, z are the local coordinates and t is time.
- α: The thermal diffusivity constant.
- ∇^2: The Laplace operator, more information in subsection 2.2.
- $\bar{u}$: The time derivative of u, or in symbols $\frac{\delta u}{\delta t}$.

$$\bar{u} = \alpha \nabla^2 u \tag{20}$$

Now to get this into code we need the following algorithms algorithm 7 and algorithm 24. algorithm 7 implements the laplacian operator, whereas algorithm 24 implements the diffusion calculations. ∇^2 in algorithm 24 represents the call to algorithm 7.

Algorithm 24: The main calculations for calculating the effects of diffusion

$T_a \leftarrow T_a + \delta t \alpha_a \nabla^2(T_a)$;
$T_p \leftarrow T_p + \delta t \alpha_p \nabla^2(T_p)$;

5.2 Adding in Advection

With thermal diffusion in place, the temperature will spread out a bit, however air is not transported yet. This means that the winds we simulate are not actually moving any air. Advection is going to change that. The advection equation is shown in Equation 21. The symbols are:

- ψ: What is carried along (in our case temperature, K).
- t: The time (s).
- u: The fluid velocity vector (ms^{-1}).
- ∇: The divergence operator (as explained in subsection 2.2).

$$\frac{\delta \psi}{\delta t} + \nabla \cdot (\psi u) = 0 \tag{21}$$

With the divergence functon defined in algorithm 8, we now need to adjust algorithm 24 to incorporate this effect. The resulting algorithm can be found in algorithm 25. Here ∇ represents the function call to algorithm 8.

Algorithm 25: The main calculations for calculating the effects of advection

$T_{add} \leftarrow T_a + \delta t \alpha_a \nabla^2(T_a) + \nabla(T_a)$;
$T_a \leftarrow T_a + T_{add}[5 : -5, :]$ //Only add T_{add} to T_a for indices in the interval $[-nlat + 5, nlat - 5]$. ;
$T_p \leftarrow T_p + \delta t \alpha_p \nabla^2(T_p)$;

Now that we have the air moving, we also need to account for the moving of the density. This is because moving air to a certain place will change the air density at that place if the air at that place does not move away at the same rate. Say we are moving air to x at y ms^{-1}. If air at x moves at a rate z ms^{-1} and $z \neq y$ then the air density at x will change. The equation we will need for that is the mass continuity equation as shown in Equation 22 [35].

$$\frac{\delta \rho}{\delta t} + \nabla \cdot (\rho v) = 0 \tag{22}$$

Using this equation means that we will no longer assume that the atmosphere is incompressible. Therefore we need to change a few things in the code. First we need to change the ρ in algorithm 20. Since ρ is no longer constant we need to access the right value of ρ by specifying the indices. So ρ will change to $\rho[lat, lon]$. Furthermore we need to calculate ρ after the movement of air has taken place, so we need to change algorithm 25 as well to include the calculations for ρ. The new version can be found in algorithm 26. Again the ∇ represents the call to algorithm 8.

Algorithm 26: The main calculations for calculating the effects of advection

$T_{add} \leftarrow T_a + \delta t \alpha_a \nabla^2(T_a) + \nabla(T_a)$;
$T_a \leftarrow T_a + T_{add}[5 : -5, :]$ //Only add T_{add} to T_a for indices in the interval $[-nlat + 5, nlat - 5]$. ;
$\rho \leftarrow \rho + \delta t \nabla \rho$;
$T_p \leftarrow T_p + \delta t \alpha_p \nabla^2(T_p)$;

Currently the advection does not work like it should. This is probably due to boundary issues, where we get too close to the poles and it starts freaking out there [17]. So to fix this we are going to define boundaries and assume that the advection only works within those boundaries. We only let it change by half of the values. The changes are incorporated in algorithm 27. The reason why we mention this seperately, in contrast to the other fixes that we have incorporated throughout the manual already, is the accompanying change with the boundary.

Algorithm 27: The main calculations for calculating the effects of advection

$T_{add} \leftarrow T_a + \delta t \alpha_a \nabla^2(T_a) + \nabla(T_a)$;
$T_a \leftarrow T_a - 0.5 T_{add}[adv\_bound : -adv\_boun, :$
$]$ //Only subtract T_{add} to T_a for indices in the interval $[-nlat + adv\_boun, nlat - adv\_boun]$. ;
$\rho[adv\_boun : -adv\_boun, :] \leftarrow \rho -$
$0.5(\delta t \nabla \rho)$ //Only change the density for indices in the interval $[-nlat + adv\_boun, nlat - adv\_boun]$
;
$T_p \leftarrow T_p + \delta t \alpha_p \nabla^2(T_p)$;

5.3 Layers, layers and layers

With the atmospheric layers, and all matrices that have an extra dimension to account for it, we need to add the correct indices to the advection algorithm algorithm 27. Let us add it, with algorithm 28 as a result. Here the ':' means all indices of the 3 dimensional matrix. Also keep in mind that the potential temperature is described and discussed in subsection 5.4.

Algorithm 28: The main calculations for calculating the effects of advection

$T_{add} \leftarrow T_a + \delta t \alpha_a \nabla^2(T_{pot}) + \nabla(T_{pot})$;
$T_{add} \leftarrow 0.5 T_{add}[adv\_boun : -adv\_boun, :, :$
] //Only halve T_{add} for indices in the interval $[-nlat + adv\_boun, nlat - adv\_boun]$. ;
$T_{add}[-adv\_boun :, :, :] \leftarrow 0$//Only replace by 0 in the interval [0, adv_boun] ;
$T_{add}[: adv\_boun, :, :] \leftarrow 0$//Only replace by 0 in the interval [nlat - adv_boun, nlat] ;
$T_{pot} \leftarrow T_{pot} - T_{add}[adv\_boun : -adv\_boun, :, :$
] //Only subtract T_{add} from T_{pot} for indices in the interval $[-nlat + adv\_boun, nlat - adv\_boun]$. ;

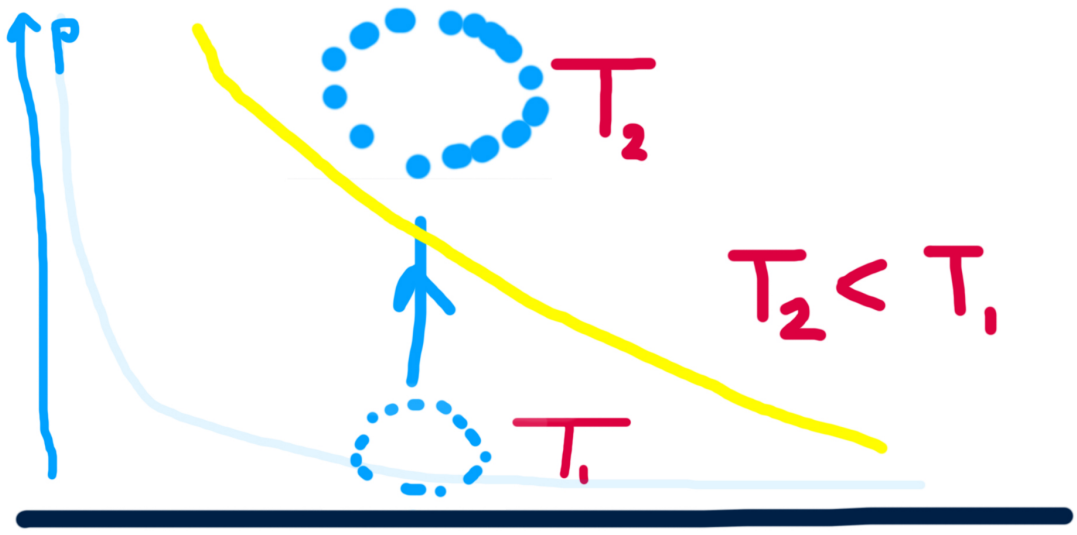

Figure 1: Visual representation of why we need Thermal Potential [17].

5.4 Adiabatic Motion

Up until now, we have been moving air and the density of the atmosphere. However we have not transported any temperature yet. What this means is that if we have a packet of air (see it as a box filled with air, but the box is invisible) P which is at the planet surface. There P has temperature T_1. If we then move P to a layer higher up in the atmosphere, P will still have the same temperature, which is wrong because the density differs. Due to this difference, there is either more or less pressure applied to the box of air which means that P will contract or expand. This obviously changes it's temperature as the air molecules are closer together/further away from each other. Therefore the energy spread is different which affects the temperature. For a visual representation, please consult subsection 5.4.

As seen in subsection 5.4, the packet of air (next to T_1) moves up into a higher layer of the atmosphere and ends up at the top (next to T_2). The packet has grown bigger, as the density in that atmospheric layer is lower and hence the air expands. Because the same energy is in that packet, but it has expanded, the temperature of that packet drops which gives us $T_2 < T_1$. The light blue graph in the background shows how the density behaves the higher you go, meaning at the far right (when we are at the highest point) the density is quite low and at the far left (when we are at the planet surface) the density is quite high. The yellow graph shows the temperature of the air at that height, which behaves similarly to the density graph.

Thermal potential or potential temperature, is the temperature that the packet of air would be if it is at the planet surface. So take the packet at T_2 in subsection 5.4. If we move that down to the surface, then it would be temperature T which is then equal to the thermal potential. The equation corresponding to thermal potential is shown in Equation 23a [36]. The symbols in Equation 23a mean:

- T: The temperature of the packet of air (K).
- p_0: Reference pressure, usually the pressure at the planet surface (Pa).
- p: Pressure of the packet of air (Pa).
- R: Gas constant as defined in subsubsection 1.2.1 ($JK^{-1}mol^{-1}$).
- C_a: Specific heat capacity of air at a constant pressure ($Jkg^{-1}K^{-1}$).

$$\theta = T(\frac{p_0}{p})^{\frac{R}{C_a}} \tag{23a}$$

$$T = \theta(\frac{p}{p_0})^{\frac{R}{C_a}} \tag{23b}$$

If we now re-arrange Equation 23a so that we have T on one side and the rest on the other side, we get Equation 23b. With this we can convert temperature into potential temperature and vice versa. The whole process of moving temperature around is called adiabatic motion. Now it is time to get this into code. For this to work we need to translate both Equation 23a and Equation 23b into code and create an overarching function that calls the previously mentioned equations. Let us start with Equation 23a which is described in algorithm 29. Note that $\frac{R}{C_a}$ does not change as they are constants, therefore we can precompute them which saves a bit of time (namely $O(n)$ divisions, where n is the length of p). Also note that we can inverse the process by inserting a minus in the exponent and swapping T_a and θ, which can easily be done in the call to algorithm 29.

Algorithm 29: Converting temperature into potential temperature

Input : temperature of the atmosphere (or potential temperature) T_a, air pressure p, boolean *back*
Output: potential temperature θ
if *back* **then**
  $\kappa \leftarrow \frac{R}{C_a}$;
else
  $\kappa \leftarrow -\frac{R}{C_a}$;
for $k \leftarrow 0$ **to** $p.length$ **do**
  $\theta[:,:,k] \leftarrow T_a[:,:,k](\frac{p_z[k]}{p_z[0]})^{\kappa}$
end
return θ

Now we only need to do one more thing, replace the algorithm that calculates T_{pot} as a result of advection. Let us do that in algorithm 30, where `ThermalAdv` is the call to algorithm 8.

Algorithm 30: The main calculations for calculating the effects of advection

$T_{add} \leftarrow T_{pot} + \nabla(T_{pot}, p_z)$;
$T_{add} \leftarrow 0.5T_{add}[adv\_boun : -adv\_boun, :, :$
] //Only halve T_{add} for indices in the interval $[-nlat + adv\_boun, nlat - adv\_boun]$. ;
$T_{add}[-adv\_boun :, :, :] \leftarrow 0$//Only replace by 0 in the interval [0, adv_boun] ;
$T_{add}[: adv\_boun, :, :] \leftarrow 0$//Only replace by 0 in the interval [nlat - adv_boun, nlat] ;
$T_{pot} \leftarrow T_{pot} - T_{add}[adv\_boun : -adv\_boun, :, :$
] //Only subtract T_{add} from T_{pot} for indices in the interval $[-nlat + adv\_boun, nlat - adv\_boun]$. ;

6 Planar Approximation

It is time to deal with the pole situation. The north and south poles that is, not the lovely people over in Poland. We run into problems because the latitude longitude grid cells become to small near the poles. Therefore, the magnitudes no longer fit into one cell and overflow into other cells which makes everything kind of funky. So we need to fix that, and we do that by a planar approximation.

6.1 The Initial Theory

As said earlier, the grid cells on the latitude longitude grid get closer together the closer you get to the poles which poses problems. To fix this, we will be using a planar approximation of the poles. What this means is that we will map the 3D grid near the poles onto a 2D plane parallel to the poles, as if we put a giant flat plane in the exact center of the poles and draw lines from the grid directly upwards to the plane. For a visual representation, please consult the stream with timestamp 1:38:25 [19], which includes some explanation. In the streamm we use r to indicate the radius of the planet (which we assume is a sphere), θ for the longitude and λ for the latitude. So we have spherical coordinates, which we need to transform into x and y coordinates on the plane. We also need the distance between the center point (the point where the plane touches the planet which is the center of the pole) and the projected point on the plane from the grid (the location on the plane where a line from the gird upwards to the plane hits it). This distance is denoted by a (Simon chose this one, not me). We then get the following equations as shown in Equation 24a, Equation 24b and Equation 24c.

$$a = r\cos(\theta) \tag{24a}$$

$$x = a\sin(\lambda) \tag{24b}$$

$$y = a\cos(\lambda) \tag{24c}$$

But what if we know x and y and want to know θ and λ? Pythagoras' Theorem then comes into play [11]. We know that (due to Pythagoras) Equation 25 must always be true. Then if we substitue a by $\sqrt{x^2+y^2}$ in Equation 24a we get Equation 26a. Then we transform that equation such that we only have θ on one side and the rest on the other side (since we want to know θ) and we get Equation 26c.

$$x^2 + y^2 = a^2 \tag{25}$$

$$\sqrt{x^2+y^2} = r\cos(\theta) \tag{26a}$$

$$\frac{\sqrt{x^2+y^2}}{r} = \cos(\theta) \tag{26b}$$

$$\arccos(\frac{\sqrt{x^2+y^2}}{r}) = \theta \tag{26c}$$

For λ we need another trigonometric function which is the tangent (tan). The tangent is defined in Equation 27. If we then take a look at Equation 24b and Equation 24c, we see that λ is present in both equations. So we need to use both to get λ [6]. So let's combine Equation 24b and Equation 24c in Equation 28a, transform it such that we end up with only λ on one side and the rest on the other side and we end up with Equation 28c.

$$\tan(\alpha) = \frac{\sin(\alpha)}{\cos(\alpha)} \tag{27}$$

[6] Yes you could only use one but since we both know x and y it is a bit easier to use both than to only use one as you need to know θ at that point as well which may or may not be the case.

$$\frac{x}{y} = \frac{a\sin(\lambda)}{a\cos(\lambda)} = \frac{\sin(\lambda)}{\cos(\lambda)} \tag{28a}$$

$$\frac{x}{y} = \tan(\lambda) \tag{28b}$$

$$\lambda = \arctan(\frac{x}{y}) \tag{28c}$$

6.2 The Grid Code

To start the planar approximation, we first need to create a grid. One for the north pole, and one for the south pole. Now since the project is made in Python, Simon uses a function to generate a grid from two coordinate vectors (lists with a coordinate as elements). Since the documentation tries to not be language specific, I instead opt to use words instead of function calls. What that comes down to is use your favourite language and import the libraries that do this for you, or start coding your own after finding out how to do that (your mileage may vary). To implement the grid function in the exact same way as the numpy packages does, please refer to the following two references [20] [25]. Anyway, the code for the grid can be found in algorithm 31. To convert x, y coordinates into lat, lon coordinates, we make use of Equation 26c and Equation 28c. Keep in mind that the equations themselves assume that the angles are in radians, whereas the model uses the angles in degrees so they need to be converted. To convert from lat, lon back into x, y we need to combine Equation 24a, Equation 24b and Equation 24c. $gridPad$ refers to how many cells for padding we add to the grid. We add padding so that the calculations later on all go smoothly and don't require complex and specific code to deal with them. The more general an algorithm is, the better, as you can reuse it for other things while not adjusting the algorithm for that particular problem. That way, it is easier and better to adjust the input then to alter the algorithm.

To make the process of generating the grid faster, we can vectorise the first for loop as shown in algorithm 32. Here instead of going over each value one by one (and because Python is an interpreter based language, the interpreter cannot optimise it out as it does not know what is coming next) we treat everything as a vector or a scalar. By treating the arrays as a vector, Python can efficiently (and most likely in parallel) perform the same calculations as it would do by using the classic for loop. We include the `flatten()` method as it forces everything to be a one dimensional array (vector).

We need to do a similar thing for the north pole and insert a few changes to some of the equations to correct for different angles and similar things. The code can be found in algorithm 33. Again, see the references at the south pole explanation on how to generate the grid itself.

As was the case with the south pole, we can optimise the first for loop by vectorising it. A similar snippet to algorithm 32 is the result, though there are small changes. These changes can be found in algorithm 34.

In both algorithms it is important to make a distinction between $gridLatCoords$ and $polarXCoords$ and their respective variants. $gridLatCoords$ are the latitudinal coordinates on the lat, lon grid corresponding to the x and y coordinates on the polar grid. Whereas $polarXCoords$ are the x coordinates on the polar grid corresponding to the latitude and longitude on the lat, lon grid. Those are different to the $gridXValues$ as they represent the values on the x axis as integers which may or may not directly correspond to the $polarXCoords$. So we need a way of mapping the $polarXCoords$ to $gridXValues$ and their respective values and vice versa. This is done in subsection 6.3.

6.3 Switching between grids

Now that we have defined the polar plane grid, we need code to convert the values from the lat, lon grid to the polar plane grid and vice versa. Let's start with converting to the polar grid. We need 2 versions of the algorithm for that. One that converts in 2 dimensions, and one that converts in 3 dimensions. The code for the 2 dimensional case is shown in algorithm 35. Here we use bivariate spline interpolation [21] which is a different form of linear interpolation than discussed in subsection 2.4 though the same principle applies.

The 3 dimensional algorithm is quite similar to the 2 dimensional algorithm, which can be found in algorithm 36

Algorithm 31: Generating the grid for polar approximation of the south pole

$poleLowIndexS \leftarrow$ find first index where $lat > poleLowerLatLimit$;
$poleHighIndexS \leftarrow$ find first index where $lat > poleHigherLatLimit$;
$polarGridResolution \leftarrow dx[poleLowIndexS]$; `//Will be reused for the north pole`
$gridSize \leftarrow r\cos(lat[poleLowIndexS + gridPad]\frac{\pi}{180})$; `//Will be reused for the north pole`
$gridXAxisS \leftarrow$ array going from $-gridSize$ to $gridSize$ with steps of $polarGridResolution$;
$gridYAxisS \leftarrow$ array going from $-gridSize$ to $gridSize$ with steps of $polarGridResolution$;
$gridXValuesS, gridYValuesS \leftarrow$ generate the grid from the two axis vectors $gridXAxissS$ and $gridYAxissS$;
$gridSideLength \leftarrow gridXValuesS.length$; `//Is globally available...`
$gridLatCoordsS \leftarrow$ empty list ;
$gridLonCoordsS \leftarrow$ empty list ;
for $i \leftarrow 0$ **to** $gridXValuesS.length$ **do**
  for $j \leftarrow 0$ **to** $gridXValuesS[i].length$ **do**
    $x \leftarrow gridXValuesS[i, j]$;
    $y \leftarrow gridYValuesS[i, j]$;
    $latPoint \leftarrow -\arccos(\frac{\sqrt{x^2+y^2}}{r})\frac{180}{\pi}$;
    $lonPoint \leftarrow 180 - \arctan(\frac{x}{y})\frac{180}{\pi}$;
    $gridLatCoordsS.append(latPoint)$;
    $gridLonCoordsS.append(lonPoint)$;
  end
end
$polarXCoordsS \leftarrow$ empty list ;
$polarYCoordsS \leftarrow$ empty list ;
for $i \leftarrow 0$ **to** $poleLowIndexS$ **do**
  for $j \leftarrow 0$ **to** $nlon$ **do**
    $polarXCoordsS.append(r\cos(lat[i]\frac{\pi}{180})\sin(lon[j]\frac{\pi}{180}))$;
    $polarYCoordsS.append(-r\cos(lat[i]\frac{\pi}{180})\cos(lon[j]\frac{\pi}{180}))$;
  end
end

Having dealt with converting to the polar grid, we now also need to deal with converting from the polar grid. In contrast to converting to the polar griod, this is only done in 3 dimensions so we do not need a 2 dimensional algorithm. How we convert from the polar grid can be found in algorithm 37.

6.4 Creating the Other Required Vectors and Matrices (Planes)

So now that we have the grids, we need some more vectors and matrices. We need the coriolis plane (the coriolis force on the polar plane) and the velocity vectors. We need these in order for the velocity calculations to go correctly. We also need these at this level and not locally as both velocity and advection need the velocity vectors and it is easier to adjust the coriolis plane for different planets on the highest level than when it is embedded into the calculation algorithms. How we create the plane and the velocity vectors is shown in algorithm 38.

Algorithm 32: Snippet for generating the grid for polar approximation of the south pole

$gridLatCoordsS \leftarrow (-\arccos(\frac{\sqrt{gridXValuesS^2+gridYValuesS^2}}{r})\frac{180}{\pi})$`.flatten()` ;
$gridLonCoordsS \leftarrow (180 - \arctan(\frac{gridXValuesS}{gridYValuesS})\frac{180}{\pi})$`.flatten()` ;

Algorithm 33: Generating the grid for polar approximation of the north pole

$poleLowIndexN \leftarrow$ find last index where $lat < -poleLowerLatLimit$;
$poleHighIndexN \leftarrow$ find last index where $lat < -poleHigherLatLimit$;

$gridXAxisN \leftarrow$ array going from $-gridSize$ to $gridSize$ with steps of $polarGridResolution$;
$gridYAxisN \leftarrow$ array going from $-gridSize$ to $gridSize$ with steps of $polarGridResolution$;
$gridXValuesN, gridYValuesN \leftarrow$ generate the grid from the two axis vectors $gridXAxisN$ and $gridYAxisN$;

$gridLatCoordsN \leftarrow$ empty list ;
$gridLonCoordsN \leftarrow$ empty list ;
for $i \leftarrow 0$ **to** $gridXValuesN.length$ **do**
  for $j \leftarrow 0$ **to** $gridXValuesN[i].length$ **do**
    $x \leftarrow gridXValuesN[i, j]$;
    $y \leftarrow gridYValuesN[i, j]$;
    $latPoint \leftarrow \arccos(\frac{\sqrt{x^2+y^2}}{r})\frac{180}{\pi}$;
    $lonPoint \leftarrow 180 - \arctan(\frac{x}{y})\frac{180}{\pi}$;
    $gridLatCoordsN.append(latPoint)$;
    $gridLonCoordsN.append(lonPoint)$;
  end
end

$polarXCoordsN \leftarrow$ empty list ;
$polarYCoordsN \leftarrow$ empty list ;
for $i \leftarrow 0$ **to** $poleLowIndexN$ **do**
  for $j \leftarrow 0$ **to** $nlon$ **do**
    $polarXCoordsN.append(r\cos(lat[i]\frac{\pi}{180})\sin(lon[j]\frac{\pi}{180}))$;
    $polarYCoordsN.append(-r\cos(lat[i]\frac{\pi}{180})\cos(lon[j]\frac{\pi}{180}))$;
  end
end

Algorithm 34: Snippet for generating the grid for polar approximation of the north pole

$gridLatCoordsS \leftarrow (\arccos(\frac{\sqrt{gridXValuesN^2+gridYValuesN^2}}{r})\frac{180}{\pi})$.`flatten()` ;
$gridLonCoordsS \leftarrow (180 - \arctan(\frac{gridXValuesN}{gridYValuesN})\frac{180}{\pi})$.`flatten()` ;

6.5 Gradually Changing Grids

Now that we can convert between grids we also need a way to do so gradually. Otherwise we would move the hard border we had previously around the poles further down the lat, lon grid. Instead we have to do some interpolation between the two grids in order to ensure a smooth transition in the final output, so that there are no hard borders. This interpolation is done in algorithm 39, using the linear interpolation technique as discussed in subsection 2.4.

6.6 Gradients on the Grid

With our new found ability to convert to and from the polar grid, while also gradually transitioning, we now get to the part we are doing it all for. Calculations on the polar grid. In order for that to work, we need some utility functions specifically for the polar grid first. We will need gradients in all 3 dimensions, those being the x dimension, the y dimension and the p dimension (pressure). All of the gradients will be quite similar to algorithm 4, algorithm 5 and algorithm 6 though some small tweaks are required as we are not differentiating over a spehere but over a plane. These changes are reflected in algorithm 40, algorithm 41 and algorithm 42.

Algorithm 35: Converting from lat, lon grid to polar plane grid in 2 dimensions

Input : Latitude coordinates lat, Longitude coordinates lon, Values of the lat, lon grid $data$, Length of one axis of the polar grid? $gridSize$, Latitude coordinates on the polar grid $gridLatCoords$, Longitude coordinates on the polar grid $gridLonCoords$

Output: Double array representing the values of the lat, lon grid on the polar grid

$f \leftarrow$ `BivariateSpline`$(lat, lon, data)$; `//Do the interpolation`

$polarPlane \leftarrow f(gridLatCoords, gridLonCoords)$.`reshape`$((gridSize, gridSize))$; `//Check the values of the interpolation at the specified coordinates and force them to align to the polar grid`

return $polarPlane$

Algorithm 36: Converting from lat, lon grid to polar plane grid in 3 dimensions

Input : Latitude coordinates lat, Longitude coordinates lon, Values of the lat, lon grid $data$, Length of one axis of the polar grid? $gridSize$, Latitude coordinates on the polar grid $gridLatCoords$, Longitude coordinates on the polar grid $gridLonCoords$

Output: Triple array representing the values of the $lat, lon, layer$ grid on the polar grid

$polarPlane \leftarrow$ 3 Dimensional array where the 1^{st} and 2^{nd} dimensions have length $gridSize$ and the 3^{rd} dimension has length $data[0][0].length$;

for $k \leftarrow 0$ **to** $data[0][0].length$ **do**

  $f \leftarrow$ `BivariateSpline`$(lat, lon, data[:, :, k])$; `//Do the interpolation on this layer`

  $polarPlane[:, :, k] \leftarrow f(gridLatCoords, gridLonCoords)$.`reshape`$((gridSize, gridSize))$; `//Check the values of the interpolation at the specified coordinates and force them to align to the polar grid`

end

return $polarPlane$

Now that we have seen the normal versions, let us vectorise them for optimal performance. The vectorised versions do exactly the same, however instead of returning a single value (and the method needing to be called for each and every element) we return a matrix of gradients. By using a matrix, we can apply the gradient to multiple elements at once and efficiently. The vectorised versions can be found in algorithm 43, algorithm 44 and algorithm 45 respectively.

6.7 Calculating Velocities on the Polar Grid

With all the utility functions out of the way, we now come to the physics of it. Let us start with the velocity calculations on the polar grid. These work very similarly to the ones on the lat, lon grid, though not exactly. For the explanation of the equations, please refer to section 4. The algorithm is given (in its non-vectorised form) in algorithm 46. An important thing to note is that this algorithm does not calculate the new velocities, but rather what needs to be changed with regards to the velocities. The `gridXGradient` refers to algorithm 40 and `gridYGradient` refers to algorithm 41.

One velocity calculation is still missing. That one being the vertical velocity, which is described in algorithm 47.

Now it is time to optimise (and thus vectorise)! The new versions of the algorithms can be found in algorithm 48 and algorithm 49. Here, `gridXGradient` refers to algorithm 43 and `gridYGradient` refers to algorithm 44.

6.8 Projecting the Velocities

Now that we have methods that can calculate the velocities on the polar plane and on the normal grid, we now need to combine methods to convert from the polar plane to the normal grid and vice versa. Due to the polar grid having different axes than the normal grid, we cannot simply project them down and be done with it. We actually need to correct for the differences in axes. After projecting the polar plane velocity

Algorithm 37: Converting from the polar plane grid to the lat, lon grid in 3 dimensions

Input : Longitude coordinates lon, Values of the lat, lon grid $data$, Polar x value indices $gridXValues$, Polar y value indices $gridYValues$, Polar x coordinates $polarXCoords$, Polar y coordinates $polarYCoords$

Output: Triple array representing the values of the polar grid on the $lat, lon, layer$ grid

$resample \leftarrow$3 Dimensional array where the 1st dimension has length $\lfloor \frac{polarXCoords}{lon.length} \rfloor$, the 2nd dimension has length $lon.length$ and the 3rd dimension has length $data[0][0].length$;

for $k \leftarrow 0$ **to** $data[0][0].length$ **do**

  $f \leftarrow$ `BivariateSpline`$)(gridXValues, gridYValues, data[:, :, k])$; `//Do the interpolation on this layer`

  $resample[:, :, k] \leftarrow f(polarXCoords, polarYCoords)$.`reshape`$((\lfloor \frac{polarXCoords}{lon.length} \rfloor, lon.length))$;

  `//Check the values of the interpolation at the specified coordinates and force them to align to the polar grid`

end

return $resample$

Algorithm 38: Generating the coriolis planes and the velocity vectors

$data \leftarrow$ matrix with dimensions $nlat - poleLowIndexN + gridPad \times nlon$;

for $i \leftarrow poleLowIndexN - gridPad$ **to** $nlat - 1$ **do**

  $data[i - poleLowIndexN, :] \leftarrow coriolis[i]$;

end

$coriolisPlaneN \leftarrow$ `BeamMeUp2D`$(lat[(poleLowIndexN - gridPad) :], lon, data, gridSideLength, gridLatCoordsN, gridLonCoordsN)$;

$data \leftarrow$ matrix with dimensions $poleLowIndexS + gridPad \times nlon$;

for $i \leftarrow 0$ **to** $poleLowIndexS + gridPad - 1$ **do**

  $data[i, :] \leftarrow coriolis[i]$;

end

$coriolisPlaneS \leftarrow$ `BeamMeUp2D`$(lat[: (poleLowIndexS + gridPad)], lon, data, gridSideLength, gridLatCoordsS, gridLonCoordsS)$;

vectors down as is, we correct for the mismatch in axes by converting everything back to the lat, lon grid. As that grid uses spherical coordinates, we know that each pointon that grid is uniquely identified by a combination of the sin and cos functions. This fact we use in realigning the velocity vectors. The whole process is shown in algorithm 50 and algorithm 51. `BeamMeDown` refers to algorithm 37. Again, the north and south methods are quite similar. Apart from a few different variable names the only real difference is reversing the reprojected vector values along the 2nd dimension for the north polar plane. This has to do with the difference in axis alignment between the two polar planes. Now you might be wondering what $[\texttt{None}, :, \texttt{None}]$ means when appended to lon. It turns the lon 1-dimensional array into a 3-dimensional array. It does this by creating small arrays that contain all a single element. So $lon[\texttt{None}, :, \texttt{None}]$ will become something like: $[[[x], [y], \dots]]$ where x and y are elements of the original lon.

The previous algorithms show how to convert from the polar plane to the lat, lon grid. However we also need the other direction. That process is shown in algorithm 52. We use similar information to re-align the velocity vectors to the polar plane grid as we used to re-align the velocity vectors to the lat, lon grid. `BeamMeUp` corresponds to the algorithm 36.

6.9 Advection on the Polar Planes

Like we have done with the velocities, we also need to make a calculation for the advection on the polar planes themselves. We do this in algorithm 53. For more details about advection itself, please refer to section 5. The interesting bit of magic happens with the $\dot{y}[i, j, :] + |\dot{y}[i, j, :]|$ term. If $\dot{y}[i, j, :]$ is negative, then this whole term will be 0 and hence will not be considered, whereas if it is positive it will be considered

Algorithm 39: Gradually transition from the lat, lon grid to the polar grid

Input : Index when to start using the polar grid $poleLowIndex$, Index when to only use the polar grid $poleHighIndex$, Polar data $polarData$, lat, lon data $sphericalData$

Output: 3 Dimensional array representing the values on the polar grid with the part that gradually transitions to the lat, lon grid

$output \leftarrow$ 3 Dimensional array with the exact same structure as $polarData$;

$overlap \leftarrow |poleLowIndex - poleHighIndex|$;

if $lat[poleLowIndex] < 0$ **then** `//Determine whether we are talking about the north or south pole`

  `//South pole`

  for $k \leftarrow 0$ **to** $output[0][0].length$ **do**

    for $i \leftarrow 0$ **to** $poleLowIndex$ **do**

      if $i < poleHighIndex$ **then**

        $\lambda \leftarrow 0$;

      else

        $\lambda \leftarrow \frac{i - poleHighIndex}{overlap}$;

      $output[i, :, k] \leftarrow (1 - \lambda) sphericalData[i, :, k] + \lambda polarData[i, :, k]$;

    end

  end

else

  `//North pole`

  for $k \leftarrow 0$ **to** $output[0][0].length$ **do**

    for $i \leftarrow 0$ **to** $nlat - poleLowIndex$ **do**

      if $i + poleLowIndex + 1 > poleHighIndex$ **then**

        $\lambda \leftarrow 0$;

      else

        $\lambda \leftarrow \frac{i}{overlap}$;

      $output[i, :, k] \leftarrow (1 - \lambda) sphericalData[i, :, k] + \lambda polarData[i, :, k]$;

    end

  end

return $output$

(that's also why we divide by $2 * polarGridResolution$, to correct for adding $\dot{y}[i, j, :]$ twice). Doing it this way saves an if statement and allows everything to be calculated faster along the last dimension. We have also added specific assignments for $j = 0$ and for $j = \dot{x}[i].length$ as these indicate the boundaries of the $\dot{x}$ vector and saves us 2 more if statements.

Algorithm 40: Gradient in the x dimension on the polar grid

Input : Triple array of data *data*, first index i, second index j and third index k, resolution of the polar grid *polarGridResolution*

Output: Gradient in the x dimension for the value of the grid point at the specified coordinates

if $i = 0$ **then**

  $value \leftarrow \frac{data[i,j+1,k]-data[i,j,k]}{polarGridResolution}$;

else if $j = gridSideLength - 1$ **then**

  $value \leftarrow \frac{data[i,j,k]-data[i,j-1,k]}{polarGridResolution}$;

else

  $vale \leftarrow \frac{data[i,j+1,k]-data[i,j-1,k]}{2polarGridResolution}$;

return *value*

Algorithm 41: Gradient in the y dimension on the polar grid

Input : Triple array of data *data*, first index i, second index j and third index k, resolution of the polar grid *polarGridResolution*

Output: Gradient in the y dimension for the value of the grid point at the specified coordinates

if $i = 0$ **then**

  $value \leftarrow \frac{data[i+1,j,k]-data[i,j,k]}{polarGridResolution}$;

else if $i = gridSideLength - 1$ **then**

  $value \leftarrow \frac{data[i,j,k]-data[i-1,j,k]}{polarGridResolution}$;

else

  $vale \leftarrow \frac{data[i+1,j,k]-data[i-1,j,k]}{2polarGridResolution}$;

return *value*

Algorithm 42: Gradient in the p dimension on the polar grid

Input : Triple array of data *data*, first index i, second index j and third index k, pressure levels *pressureLevels*

Output: Gradient in the p dimension for the value of the grid point at the specified coordinates

if $i = 0$ **then**

  $value \leftarrow \frac{data[i,j,k+1]-data[i,j,k]}{pressureLevels[k+1]-pressureLevels[k]}$;

else if $k = nlevels - 1$ **then**

  $value \leftarrow \frac{data[i,j,k]-data[i,j,k-1]}{pressureLevels[k]-pressureLevels[k-1]}$;

else

  $vale \leftarrow \frac{data[i,j,k+1]-data[i,j,k-1]}{pressureLevels[k+1]-pressureLevels[k-1]}$;

return *value*

Algorithm 43: Gradient in the x dimension on the polar grid

Input : Triple array of data *data*, resolution of the polar grid *polarGridResolution*

Output: Triple array containing the gradients for each cell on the polar grid

$shiftEast \leftarrow$ shift all x coordinates in *data* one cell to the east (positive x) ;

$shiftWest \leftarrow$ shift all x coordinates in *data* one cell to the west (negative x) ;

return $\frac{shiftWest-shiftEast}{2polarGridResolution}$

Algorithm 44: Gradient in the y dimension on the polar grid

Input : Triple array of data *data*, resolution of the polar grid *polarGridResolution*
Output: Triple array containing the gradients for each cell on the polar grid
$shiftSouth \leftarrow$ shift all y coordinates in *data* one cell to the south (negative y) ;
$shiftNorth \leftarrow$ shift all y coordinates in *data* one cell to the north (positive y) ;
return $\frac{shiftNorth - shiftSouth}{2polarGridResolution}$

Algorithm 45: Gradient in the p dimension on the polar grid

Input : Triple array of data *data*, pressure levels *pressureLevels*
Output: Triple array containing the gradients for each cell on the polar grid
$shiftUp \leftarrow$ shift all p coordinates in *data* one cell upwards (positive p) ;
$shiftDown \leftarrow$ shift all p coordinates in *data* one cell downwards (negative p) ;
$shiftPressureUp \leftarrow$ shift the pressure in *pressureLevels* up by one cell ;
$shiftPressureDown \leftarrow$ shift the pressure in *pressureLevels* down by one cell ;
return $\frac{shiftDown - shiftUp}{shiftPressureDown - shiftPressureUp}$

Algorithm 46: Velocity calculations on the polar plane

Input : Polar plane *polarPlane*, Length of one axis of the grid *polarGridLength*, Coriolis force on the plane (with the same size and resolution as the *polarPlane*) *coriolisPlane*, East-west velocity vector on the polar plane $\dot{x}$, North-south velocity vector on the polar plane $\dot{y}$, Resolution of the polar grid *polarGridResolution*
Output: Additions to the east-west velocity vector on the polar plane $\dot{x}_{add}$, Additions to the north-south velocity vector on the polar plane $\dot{y}_{add}$

for $i \leftarrow 0$ **to** *polarGridLength* **do**
  for $j \leftarrow 0$ **to** *polarGridLength* **do**
    for $k \leftarrow 0$ **to** $polarPlane[0,0].length$ **do**
      if $k \geq 17$ **then**
        $\dot{x}_{add}[i,j,k] \leftarrow -\dot{x}[i,j,k]\texttt{gridXGradient}(\dot{x},i,j,k) - \dot{y}[i,j,k]\texttt{gridYGradient}(\dot{x},i,j,k) +$
        $coriolisPlane[i,j]\dot{y}[i,j,k] - \texttt{gridXGradient}(polarPlane,i,j,k) - \dot{x}[i,j,k] \cdot 10^{-5}$;
        $\dot{y}_{add}[i,j,k] \leftarrow -\dot{x}[i,j,k]\texttt{gridXGradient}(\dot{y},i,j,k) - \dot{y}[i,j,k]\texttt{gridYGradient}(\dot{y},i,j,k) -$
        $coriolisPlane[i,j]\dot{x}[i,j,k] - \texttt{gridXGradient}(polarPlane,i,j,k) - \dot{y}[i,j,k] \cdot 10^{-5}$;
      else
        $\dot{x}_{add}[i,j,k] \leftarrow -\dot{x}[i,j,k]\texttt{gridXGradient}(\dot{x},i,j,k) - \dot{y}[i,j,k]\texttt{gridYGradient}(\dot{x},i,j,k) -$
        $\dot{x}[i,j,k] \cdot 10^{-3}$;
        $\dot{y}_{add}[i,j,k] \leftarrow -\dot{x}[i,j,k]\texttt{gridXGradient}(\dot{y},i,j,k) - \dot{y}[i,j,k]\texttt{gridYGradient}(\dot{y},i,j,k) -$
        $\dot{y}[i,j,k] \cdot 10^{-3}$;
    end
  end
end
return $\dot{x}_{add}, \dot{y}_{add}$

Algorithm 47: Vertical velocity calculations for the polar plane

Input : East-west velocity vector on the polar plane $\dot{x}$, North-south velocity vector on the polar plane $\dot{y}$, Pressure levels $pressureLevels$, Temperature on the polar plane $temp$, Resolution of the polar grid $polarGridResolution$

Output: Vertical velocity on the polar plane

$output \leftarrow$ array like $\dot{x}$;

for $i \leftarrow 0$ **to** $output.length$ **do**

  for $j \leftarrow 0$ **to** $output[i].length$ **do**

    for $k \leftarrow 0$ **to** $output[i,j].length$ **do**

      $output[i,j,k] = -\frac{(pressureLevels[k]-pressureLevels[k-1])*pressureLevels[k]*g*(\texttt{gridXGradient}(\dot{x},i,j,k)+\texttt{gridYGradient}(\dot{y},i,j,k))}{287*temp[i,j,k]}$;

    end

  end

end

return $output$

Algorithm 48: Velocity calculations on the polar plane

Input : Polar plane $polarPlane$, Length of one axis of the grid $polarGridLength$, Coriolis force on the plane (with the same size and resolution as the $polarPlane$) $coriolisPlane$, East-west velocity vector on the polar plane $\dot{x}$, North-south velocity vector on the polar plane $\dot{y}$, Resolution of the polar grid $polarGridResolution$

Output: Additions to the east-west velocity vector on the polar plane $\dot{x}_{add}$, Additions to the north-south velocity vector on the polar plane $\dot{y}_{add}$

$\dot{x}_{add} \leftarrow -\dot{x}\texttt{gridXGradient}(\dot{x}, polarGridResolution) - \dot{y}\texttt{gridYGradient}(\dot{x}, polarGridResolution) + coriolisPlane[:,:,None]\dot{y} - \texttt{gridXGradient}(polarPlane, polarGridResolution) - \dot{x} \cdot 10^{-5}$;

$\dot{y}_{add} \leftarrow -\dot{x}\texttt{gridXGradient}(\dot{y}, polarGridResolution) - \dot{y}\texttt{gridYGradient}(\dot{y}, polarGridResolution) - coriolisPlane[:,:,None]\dot{x} - \texttt{gridXGradient}(polarPlane, polarGridResolution) - \dot{y} \cdot 10^{-5}$;

$\dot{x}_{add}[:,:,17:] \leftarrow -\dot{x}[:,:,17:]\texttt{gridXGradient}(\dot{x}, polarGridResolution) - \dot{y}[:,:,17:]\texttt{gridYGradient}(\dot{x}, polarGridResolution) - \dot{x}[:,:,17:] \cdot 10^{-3}$;

$\dot{y}_{add}[:,:,17:] \leftarrow -\dot{x}[:,:,17:]\texttt{gridXGradient}(\dot{y}, polarGridResolution) - \dot{y}[:,:,17:]\texttt{gridYGradient}(\dot{y}, polarGridResolution) - \dot{y}[:,:,17:] \cdot 10^{-3}$;

return $\dot{x}_{add}, \dot{y}_{add}$

Algorithm 49: Vertical velocity calculations for the polar plane

Input : East-west velocity vector on the polar plane $\dot{x}$, North-south velocity vector on the polar plane $\dot{y}$, Pressure levels $pressureLevels$, Temperature on the polar plane $temp$, Resolution of the polar grid $polarGridResolution$

Output: Vertical velocity on the polar plane

$shiftUp \leftarrow$ shift all cells in pressureLevels up by one cell (wrap around if needed) ;

return $-\frac{(pressureLevels-shiftUp)*pressureLevels*g*(\texttt{gridXGradient}(\dot{x},polarGridResolution)+\texttt{gridYGradient}(\dot{y},polarGridResolution))}{287*temp}$

Algorithm 50: Projecting velocities from the northern polar plane down to the normal grid

Input : Longitude coordinates lon, East-west velocity vector on the polar plane $\dot{x}$, North-south velocity vector on the polar plane $\dot{y}$, Lower limit of northern polar plane $poleLowIndexN$, Polar x value indices $gridXValues$, Polar y value indices $gridYValues$, Polar x coordinates $polarXCoords$, Polar y coordinates $polarYCoords$

Output: Reprojected east-west velocity vector on the lat, lon grid u, Reprojected north-south velocity vector on the lat, lon grid v

$repX \leftarrow$
`BeamMeDown`$(lon, \dot{x}, poleLowIndexN, gridXValues, gridYValues, polarXCoords, polarYCoords)$;
$repY \leftarrow$
`BeamMeDown`$(lon, \dot{y}, poleLowIndexN, gridXValues, gridYValues, polarXCoords, polarYCoords)$;
$u \leftarrow repX * \sin(lon[\texttt{None}, :, \texttt{None}] * \frac{\pi}{180}) + repY * \cos(lon[\texttt{None}, :, \texttt{None}] * \frac{\pi}{180})$;
$v \leftarrow repX * \cos(lon[\texttt{None}, :, \texttt{None}] * \frac{\pi}{180}) - repY * \sin(lon[\texttt{None}, :, \texttt{None}] * \frac{\pi}{180})$;
$u \leftarrow$ reverse all values of u along the 2nd dimension ;
$v \leftarrow$ reverse all values of v along the 2nd dimension ;
return u, v

Algorithm 51: Projecting velocities from the southern polar plane down to the normal grid

Input : Longitude coordinates lon, East-west velocity vector on the polar plane $\dot{x}$, North-south velocity vector on the polar plane $\dot{y}$, Lower limit of southern polar plane $poleLowIndexS$, Polar x value indices $gridXValues$, Polar y value indices $gridYValues$, Polar x coordinates $polarXCoords$, Polar y coordinates $polarYCoords$

Output: Reprojected east-west velocity vector on the lat, lon grid u, Reprojected north-south velocity vector on the lat, lon grid v

$repX \leftarrow$
`BeamMeDown`$(lon, \dot{x}, poleLowIndexS, gridXValues, gridYValues, polarXCoords, polarYCoords)$;
$repY \leftarrow$
`BeamMeDown`$(lon, \dot{y}, poleLowIndexS, gridXValues, gridYValues, polarXCoords, polarYCoords)$;
$u \leftarrow repX * \sin(lon[\texttt{None}, :, \texttt{None}] * \frac{\pi}{180}) + repY * \cos(lon[\texttt{None}, :, \texttt{None}] * \frac{\pi}{180})$;
$v \leftarrow -repX * \cos(lon[\texttt{None}, :, \texttt{None}] * \frac{\pi}{180}) + repY * \sin(lon[\texttt{None}, :, \texttt{None}] * \frac{\pi}{180})$;
return u, v

Algorithm 52: Projecting velocities from the normal grid to the polar plane

Input : Latitude coordinates lat, Longitude coordinates lon, East-west velocity vector u, North-south velocity vector v, Size of the polar grid $gridSize$, Latitude coordinates corresponding to the polar plane grid $gridLatCoords$, Longitude coordinates correpsonding to the polar plane grid $gridLonCoords$

Output: East-west velocity vector on the polar plane $\dot{x}$, North-south velocity vector on the polar planes $\dot{y}$

$gridU \leftarrow$ `BeamMeUp`$(lat, lon, u, gridSize, gridLatCoords, gridLonCoords)$;

$gridV \leftarrow$ `BeamMeUp`$(lat, lon, v, gridSize, gridLatCoords, gridLonCoords)$;

$nlevels \leftarrow u[0,0].length$; `//Does not replace the global, is only of effect here in this method`

$\dot{x} \leftarrow$ matrix with shape $gridSize \times gridSize \times nlevels$;

$\dot{y} \leftarrow$ matrix with shape $gridSize \times gridSize \times nlevels$;

$gridLonCoords \leftarrow gridLonCoords$.`reshape`$(gridSize, gridSize)$;

if $lat[0] < 0$ **then** //Check to see with which pole we are dealing

  for $k \leftarrow 0$ **to** $nlevels - 1$ **do**

    $\dot{x}[:,:,k] \leftarrow gridU[:,:,k] * \sin(gridLonCoords * \frac{\pi}{180}) - gridV[:,:,k] * \cos(gridLonCoords * \frac{\pi}{180})$;

    $\dot{y}[:,:,k] \leftarrow gridU[:,:,k] * \cos(gridLonCoords * \frac{\pi}{180}) + gridV[:,:,k] * \sin(gridLonCoords * \frac{\pi}{180})$;

  end

else

  for $k \leftarrow 0$ **to** $nlevels - 1$ **do**

    $\dot{x}[:,:,k] \leftarrow -gridU[:,:,k] * \sin(gridLonCoords * \frac{\pi}{180}) + gridV[:,:,k] * \cos(gridLonCoords * \frac{\pi}{180})$;

    $\dot{y}[:,:,k] \leftarrow -gridU[:,:,k] * \cos(gridLonCoords * \frac{\pi}{180}) - gridV[:,:,k] * \sin(gridLonCoords * \frac{\pi}{180})$;

  end

return $\dot{x}, \dot{y}$

Algorithm 53: Performing the advection calculations on the polar plane

Input : Data matrix $data$, East-west velocity vector on the polar plane $\dot{x}$, North-south velocity vector on the polar plane $\dot{y}$, Resolution of the polar grid $polarGridResolution$

Output: 3-dimensional matrix with the advection data on the polar plane adv

$adv \leftarrow$ matrix with the same size as $data$;

for $i \leftarrow 1$ **to** $\dot{x}.length - 1$ **do**

  $j \leftarrow 0$;

  $adv[i,j,:] \leftarrow adv + \frac{(\dot{y}[i,j,:]+|\dot{y}[i,j,:]|)*(data[i,j,:]-data[i-1,j,:])}{2*polarGridResolution} + \frac{(\dot{y}[i,j,:]+|\dot{y}[i,j,:]|)*(data[i+1,j,:]-data[i,j,:])}{2*polarGridResolution}$;

  $adv[i,j,:] \leftarrow adv + \frac{(\dot{x}[i,j,:]+|\dot{x}[i,j,:]|)*(data[i,j+1,:]-data[i,j,:])}{2*polarGridResolution}$;

  for $j \leftarrow 1$ **to** $\dot{x}[i].length - 1$ **do**

    $adv[i,j,:] \leftarrow adv + \frac{(\dot{y}[i,j,:]+|\dot{y}[i,j,:]|)*(data[i,j,:]-data[i-1,j,:])}{2*polarGridResolution} + \frac{(\dot{y}[i,j,:]+|\dot{y}[i,j,:]|)*(data[i+1,j,:]-data[i,j,:])}{2*polarGridResolution}$;

    $adv[i,j,:] \leftarrow adv + \frac{(\dot{x}[i,j,:]+|\dot{x}[i,j,:]|)*(data[i,j,:]-data[i,j-1,:])}{2*polarGridResolution} + \frac{(\dot{x}[i,j,:]+|\dot{x}[i,j,:]|)*(data[i,j+1,:]-data[i,j,:])}{2*polarGridResolution}$;

  end

  $j \leftarrow \dot{x}[i].length - 1$;

  $adv[i,j,:] \leftarrow adv + \frac{(\dot{y}[i,j,:]+|\dot{y}[i,j,:]|)*(data[i,j,:]-data[i-1,j,:])}{2*polarGridResolution} + \frac{(\dot{y}[i,j,:]+|\dot{y}[i,j,:]|)*(data[i+1,j,:]-data[i,j,:])}{2*polarGridResolution}$;

  $adv[i,j,:] \leftarrow adv + \frac{(\dot{x}[i,j,:]+|\dot{x}[i,j,:]|)*(data[i,j,:]-data[i,j-1,:])}{2*polarGridResolution}$;

end

return adv

7 The Master File

The master file is the file that controls the model calculation. This file decides what calculations are used and what is done with the calculations (which is not the scope of this manual). In other words, the master file combines all the calculations and theory from the previous sections and puts it all together to form a model. As mentioned earlier, this structure enables the user to create their own version of the model. If one has their own calculations, or wants to use an older version of the calculations in this manual, then the user can define it themselves and call it instead of the calls that we use. The model is meant to be customisable, which this structure enables.

7.1 Adding a Spin-Up Time

Instead of having a static start (having the planet start from rest, so no rotations allowed) we will have the model start up for some time before we start simulating the climate extensively. To accomodate for this, we have to make some changes in the code. First we need to add two booleans (variables that can only take two values, either `TRUE` or `FALSE`) that we use to indicate to the model whether we want to simulate the wind and whether we want to simulate advection. This means that the main loop will have some changes made to it. After performing the radiation calculations we would calculate the velocities and afterwards we would calculate the advection. Instead let us change it to what is shown in algorithm 54.

Algorithm 54: Main loop that can simulate velocities and advection conditionally

```
while TRUE do
    Do the radiation calculations ;
    if velocity then
        Do the velocity calculations ;
        if advection then
            Do the advection calculations ;
        end
    end
end
```

Now to dynamically enable/disable the simulation of flow and advection we need to add the spin-up calculations to the main loop. So in algorithm 54, before algorithm 16 we add algorithm 55. What it does is it changes the timestep when spinnning up and disables flow simulation, and when a week has passed it reduces the timestep and enables flow simulation. At this point in time, the advection is not dynamically enabled/disabled but it is done by the programmer.

Algorithm 55: The spin-up block dynamically enabling or disabling flow simulation

```
if t < 7day then
    δt ← 60 · 47 ;
    velocity ← FALSE ;
else
    δt ← 60 · 9 ;
    velocity ← TRUE ;
end
```

7.2 Varying the Albedo

The albdeo (reflectiveness of the planet's surface) is of course not the same over the whole planet. To account for this, we instead vary the albedo slightly for each point in the latitude longitude grid. The algorithm that

does this is shown in algorithm 56. The uniform distribution basically says that each allowed value in the interval has an equal chance of being picked [30].

Algorithm 56: Varying the albedo of the planet

$V_a \leftarrow 0.02$;
for $lat \in [-nlat, nlat]$ **do**
  for $lon \in [0, nlon]$ **do**
    $R \leftarrow$ Pick a random number (from the uniform distribution) in the interval $[-V_a, V_a]$;
    $a[lat, lon] \leftarrow a[lat, lon] + V_a \cdot R$;
  end
end

7.3 Non-uniform air density

While air density on the surface is in general consistent, this does not hold if you move up through the atmosphere. The planet will pull air down due to gravity, which means that more air is at the planet surface than at the top of the atmosphere. Hence the air density changes throughout the atmosphere and we need to account for that. This is done in algorithm 57. Because this is used in radiation, velocity and advection, we initialise this in the master file. Though one could argue it could be part of the control panel, we opt not to include any code other than variable declarations in the control panel for greater clarity. This also means that we give the user the option to have only one layer (by skipping implementing this algorithm). Note that the $\rho[:, : i]$ notation means that for every index in the first and second dimension, only change the value for the index i in the third dimension.

Algorithm 57: Initialisation of the air density ρ

$\rho[:, :, 0] \leftarrow 1.3$;
for $i \in [1, nlevels - 1]$ **do**
  $\rho[:, :, i] \leftarrow 0.1\rho[:, :, i - 1]$
end

7.4 Interpolating the Air Density

In order to interpolate (see subsection 2.4) the air density, to have a better estimation at each grid cell, we need data. However currently we are just guessing the air density at higher levels, instead of taking real values. So let us change that. For that we are going to use the U.S. Standard Atmosphere, an industry standard measure of the atmosphere on Earth [8]. This data was provided in a text (`TXT`) file which of course needs to be read in order for the data to be used in the model. Here we only care for the density and the temperature at a specific height. So the text file only contains those two columns of the data (and the height in km of course as that is the index of the row, the property that uniquely identifies a row).

With that in mind, let's get coding and importing the data. We do this in algorithm 58. As one can see we do not specify how to open the file or how to split the read line, as this is language specific and not interesting to describe in detail. I refer you to the internet to search for how to open a text file in the language you are working in. Keep in mind in which magnitude you are working and in which magnitude the data is. If you work with km for height and the data is in m, you need to account for that somewhere by either transforming the imported data or work in the other magnitude.

Note that the function `interpolate` takes three arguments, the first one being the data points that we want to have values for, the second one is the data points that we know and the third one is the values for the data points that we know. This function may or may not exist in your programming language of choice, which might mean that you have to write it yourself. The formula that we use for interpolation can be found in Equation 3, though you still need to figure out what value you need for λ (see subsection 2.4). This is left as an exercise for the reader.

Algorithm 58: Loading in the U.S. Standard Atmosphere

$data \leftarrow$ open text file containing the us standard atmosphere data ;
foreach $line \in data$ **do**
  Split $line$ into three components, sh, st and sd, representing the height, temperature and density respectively ;
  $standardHeight.add(sh)$;
  $standardTemperature.add(st)$;
end
$temperatureProfile \leftarrow$ `interpolate`$(p_z, standardTemperature)$;
for $alt \in [0, nlevels]$ **do**
  $\rho[:, :, alt] \leftarrow densityProfile[alt]$;
  $T_i[:, :, alt] \leftarrow temperatureProfile[alt]$;
end

7.5 Clamping the Velocities

Due to the boundaries in the advection calculations (see section 5) we get weird instabilities as the velocity calculations are executed on more cells. Which means that air is trying to displace temperature (advect it) by flowing faster to those cells, but actually don't carry any temperature because we turned it off for those cells. This is something that we need to fix to get rid of weirdness around the edges. This is done in algorithm 59. Here the *bla* : means from *bla* to the last valid index, if the : is in front of *bla* then it means from the first valid index to *bla*.

Algorithm 59: Clamping the Velocities

algorithm 23 $u[(adv\_boun, -adv\_boun - 1), :, :] \leftarrow 0.5u[(adv\_boun, -adv\_boun - 1), :, :]$;
$v[(adv\_boun, -adv\_boun - 1), :, :] \leftarrow 0.5v[(adv\_boun, -adv\_boun - 1), :, :]$;
$w[(adv\_boun, -adv\_boun - 1), :, :] \leftarrow 0.5w[(adv\_boun, -adv\_boun - 1), :, :]$;
$u[: adv\_boun, :, :] \leftarrow 0$;
$v[: adv\_boun, :, :] \leftarrow 0$;
$w[: adv\_boun, :, :] \leftarrow 0$;
$u[-adv\_boun :, :, :] \leftarrow 0$;
$v[-adv\_boun :, :, :] \leftarrow 0$;
$w[-adv\_boun :, :, :] \leftarrow 0$;

7.6 Smoothing all the things

On a planet wide scale, you have a lot of variety in the data. To counteract that we filter out the high frequency data. Which means that we filter out the data that occurs sporadically. So we do not consider the data that occurs so infrequently that it means nothing. We do this for the radiation (temperature) and the velocity which is shown in algorithm 60 and algorithm 61 respectively. It is worth mentioning that algorithm 60 is executed after we do the calculations for T_{pot} (shown in algorithm 17). algorithm 61 is done after algorithm 23 but before algorithm 59.

Algorithm 60: Smoothing the atmospheric temperature

$T_{pot} \leftarrow$ `Smooth`$(T_{pot}, smooth_t)$;

Algorithm 61: Smoothing the velocity

$u \leftarrow \texttt{Smooth}(u, smooth_u)$;
$v \leftarrow \texttt{Smooth}(v, smooth_v)$;
$w \leftarrow \texttt{Smooth}(w, smooth_w, smooth_{vert})$;

Appendices

A Terms That Need More Explanation Then A Footnote

A.1 Potential

Potential is the energy change that occurs when the position of an object changes [44]. There are many potentials, like electric potential, gravitational potential and elastic potential. Let me explain the concept with an example. Say you are walking on a set of stairs in the upwards direction. As your muscles move to bring you one step upwards, energy that is used by your muscles is converted into gravitational potential. Now imagine you turn around and go downwards instead. Notice how that is easier? That is due to the gravitational potential being converted back into energy so your muscles have to deliver less energy to get you down. The potential is usually tied to a force, like the gravitational force.

A.2 Asymptotic Runtime

Asymptotic runtime is what we use in computer science to indicate how fast an algorithm works. We do it this way because concrete time indications (seconds, minutes, hours) are very machine dependent. It matters a lot if your CPU, RAM and GPU are fast or not for the runtime. Therefore, we needed something to compare algorithms by which is machine independent. That is what asymptotic runtime is. We have 3 notations for asymptotic runtime, Ω which is the lower bound of the runtime: not faster than; O which is the upperbound of the runtime: not slower than; and we have Θ which is the tight bound: not slower but also not faster than. After these 3 notations we usually denote the runtime in algebraic letters which stand for the input size. $O(n)$ for instance means that for an input of size n the algorithm will not run slower than n operations. Whereas $\Omega(n^3)$ means that the algorithm needs for an input of size n at least n^3 operations. Now this is not an exact match, as there are constants and other terms in the real runtime, but for asymptotic runtime we look at the most dominant factor, as that outgrows all the other factors if the input size increases. You can compare this by plotting the functions $y = x$ and $z = x^2$ on a graphical calculator. No matter which constant a you put in front of the x, $z = x^2$ will at some point (note we don't specify when or where) be larger than $y = ax$. What you need to remember for all this is that polynomials are faster than exponentials ($n^2 < 2^n$) and logarithms are faster than polynomials ($\log(n) < n$). $n!$ is very slow, $\log(n)$ is very fast.

A.3 Complex Numbers

As you all know in the real numbers ($\mathbb{R}$) negative roots are not allowed as they do not exist. But what would happen if we would allow them to exist? Then we move into the area of complex numbers. A complex number consists out of two parts, a real part and an imaginary part in the form $a + bi$ where $i = \sqrt{-1}$. Complex numbers have all kinds of properties, but what we need them for are rotations. This is captured in Euler's formula $e^{it} = \cos(x) + i\sin(x)$ [12]. Which means that for time t we rotate around the origin (of the complex plane) forming a circle with radius one (the unit circle). Now if you would set $t = \pi$ then the result is 0. What this means is that we have come full circle (hah) when $t = 2\pi$.

B History of the Algorithms

Back when I was a young naive programmer, I made a thing. Now a few years down the line I made the thing again, but infinitely better. So I have no use for the old thing anymore. But fear not, old algorithms (used by CLAuDE) will be collected here. This is just for historical purposes.

B.1 Radiation

B.1.1 Adding Layers

Remember Equation 9? We need this equation for every layer in the atmosphere. This also means that we have to adjust the main calculation of the code, which is described in algorithm 16. The T_a needs to change, we need to either add a dimension (to indicate which layer of the atmosphere we are talking about) or we need to add different matrices for each atmosphere layer. We opt for adding a dimension as that costs less memory than defining new arrays [7]. So T_a, and all other matrices that have to do with the atmosphere (so not T_p for instance) are no longer indexed by lat, lon but are indexed by $lat, lon, layer$. We need to account for one more thing, the absorbtion of energy from another layer. The new equation is shown in Equation 29a. Here k is the layer of the atmosphere, $k = -1$ means that you use T_p and $k = nlevels$ means that $T_{a_{nlevels}} = 0$ as that is space. Also, let us rewrite the equation a bit such that the variables that are repeated are only written once and stuff that is divided out is removed, which is done in Equation 29b. Let us also clean up the equation for the change in the surface temperature (Equation 8c) in Equation 29c.

$$\Delta T_{a_k} = \frac{\delta t(\sigma\epsilon_{k-1}T_{a_{k-1}}^4 + \sigma\epsilon_{k+1}T_{a_{k+1}}^4 - 2\epsilon_k\sigma T_{a_k}^4)}{C_a} \tag{29a}$$

$$\Delta T_{a_k} = \frac{\delta t\sigma(\epsilon_{k-1}T_{a_{k-1}}^4 + \epsilon_{k+1}T_{a_{k+1}}^4 - 2\epsilon_k T_{a_k}^4)}{C_a} \tag{29b}$$

$$\Delta T_p = \frac{\delta t(S + \sigma(4\epsilon_p T_a^4 - 4T_p^4))}{4C_p} \tag{29c}$$

With the changes made to the equation, we need to make those changes in the code as well. We need to add the new dimension to all matrices except T_p and a as they are unaffected (with regards to the storage of the values) by the addition of multiple atmospheric layers. Every other matrix is affected. The new code can be found in algorithm 62. δz

We also need to initialise the ϵ value for each layer. We do that in algorithm 63.

B.2 Velocity

B.2.1 The Primitive Equations and Geostrophy

The primitive equations (also known as the momentum equations) is what makes the air move. It is actually kind of an injoke between physicists as they are called the primitive equations but actually look quite complicated (and it says fu at the end! [17]). The primitive equations are a set of equations dictating the direction in the u and v directions as shown in Equation 30a and Equation 30b. We can make the equations simpler by using and approximation called geostrophy which means that we have no vertical motion, such that the terms with ω in Equation 30a and Equation 30b become 0. We also assume that we are in a steady state, i.e. there is no acceleration which in turn means that the whole middle part of the equations are 0. Hence we are left with Equation 30c and Equation 30d.

$$\frac{du}{dt} = \frac{\delta u}{\delta t} + u\frac{\delta u}{\delta x} + v\frac{\delta u}{\delta v} + \omega\frac{\delta u}{\delta p} = -\frac{\delta\Phi}{\delta x} + fv \tag{30a}$$

[7] This has to do with pointers, creating a new object always costs a bit more space than adding a dimension as we need a pointer to the object and what type of object it is whereas with adding a dimension we do not need this additional information as it has already been defined

Algorithm 62: The main function for the temperature calculations

Input: amount of energy that hits the planet S
Output: Temperature of the planet T_p, temperature of the atmosphere T_a
for $lat \leftarrow -nlat$ **to** $nlat$ **do**
  for $lon \leftarrow 0$ **to** $nlot$ **do**
    for $layer \leftarrow 0$ **to** $nlevels$ **do**
      $T_p[lat, lon] \leftarrow T_p[lat, lon] + \frac{\delta t((1-a[lat,lon])S+\sigma(4\epsilon[0](T_a[lat,lon,0])^4-4(T_p[lat,lon])^4))}{4C_p[lat,lon]}$;
      if $layer = 0$ **then**
        $T_a[lat, lon, layer] \leftarrow T_a[lat, lon, layer] + \frac{\delta t\sigma((T_p[lat,lon])^4-2\epsilon[layer](T_a[lat,lon,layer])^4)}{\rho[lat,lon,layer]C_a\delta z[layer]}$;
      else if $layer = nlevels - 1$ **then**
        $T_a[lat, lon, layer] \leftarrow$
        $T_a[lat, lon, layer] + \frac{\delta t\sigma(\epsilon[layer-1](T_a[lat,lon,layer-1])^4-2\epsilon[layer](T_a[lat,lon,layer])^4)}{\rho[lat,lon,layer]C_a\delta z[layer]}$;
      else
        $T_a[lat, lon, layer] \leftarrow T_a[lat, lon, layer] +$
        $\frac{\delta t\sigma(\epsilon[layer-1](T_a[lat,lon,layer-1])^4+\epsilon[layer+1]T_a[lat,lon,layer+1]-2\epsilon[layer](T_a[lat,lon,layer])^4)}{\rho[lat,lon,layer]C_a\delta z[layer]}$;
    end
  end
end

Algorithm 63: Intialisation of the insulation of each layer (also known as ϵ)

$\epsilon[0] \leftarrow 0.75$;
for $i \leftarrow 1$ **to** $nlevels$ **do**
  $\epsilon[i] \leftarrow 0.5\epsilon[i-1]$
end

$$\frac{dv}{dt} = \frac{\delta v}{\delta t} + u\frac{\delta v}{\delta x} + v\frac{\delta v}{\delta v} + \omega\frac{\delta v}{\delta p} = -\frac{\delta \Phi}{\delta y} - fu \tag{30b}$$

$$0 = -\frac{\delta \Phi}{\delta x} + fv \tag{30c}$$

$$0 = -\frac{\delta \Phi}{\delta y} - fu \tag{30d}$$

Equation 30c can be split up into to parts, the $\frac{\delta \Phi}{\delta x}$ part (the gradient force) and the fv part (the coriolis force). The same applies to Equation 30d. Effectively we have a balance between the gradient and the coriolis force as shown in Equation 31b and Equation 31c. The symbols in both of these equations are:

- Φ: The geopotential, potential (more explanation in subsection A.1) of the planet's gravity field (Jkg^{-1}).
- x: The change in the East direction along the planet surface (m).
- y: The change in the North direction along the planet surface (m).
- f: The coriolis parameter as described by Equation 31a, where Ω is the rotation rate of the planet (for Earth $7.2921 \cdot 10^{-5}$) ($rad\ s^{-1}$) and θ is the latitude [28].
- u: The velocity in the latitude (ms^{-1}).
- v: The velocity in the longitude (ms^{-1}).

$$f = 2\Omega \sin(\theta) \tag{31a}$$

$$\frac{\delta \Phi}{\delta x} = fv \tag{31b}$$

$$\frac{\delta \Phi}{\delta y} = -fu \tag{31c}$$

$$\frac{\delta p}{\rho \delta x} = fv \tag{31d}$$

$$\frac{\delta p}{\rho \delta y} = -fu \tag{31e}$$

Since we want to know how the atmosphere moves, we want to get the v and u components of the velocity vector (since v and u are the veolicites in longitude and latitude, if we combine them in a vector we get the direction of the overall velocity). So it is time to start coding and calculating! If we look back at algorithm 13, we can see that we already have a double for loop. In computer science, having multiple loops is generally considered a bad coding practice as you usually can just reuse the indices of the already existing loop, so you do not need to create a new one. However this is a special case, since we are calculating new temperatures in the double for loop. If we then also would start to calculate the velocities then we would use new information and old information at the same time. Since at index $i-1$ the new temperature has already been calculated, but at the index $i+1$ the old one is still there. So in order to fix that we need a second double for loop to ensure that we always use the new temperatures. We display this specific loop in algorithm 64. Do note that everything in algorithm 13 is still defined and can still be used, but since we want to focus on the new code, we leave out the old code to keep it concise and to prevent clutter.

Algorithm 64: The main loop of the velocity of the atmosphere calculations

for $lat \in [-nlat, nlat]$ **do**
  for $lon \in [0, nlon]$ **do**
    $u[lat, lon] \leftarrow -\frac{p[lat+1,lon]-p[lat-1,lon]}{\delta y} \cdot \frac{1}{f[lat]\rho}$;
    $v[lat, lon] \leftarrow \frac{p[lat,lon+1]-p[lat,lon-1]}{\delta x[lat]} \cdot \frac{1}{f[lat]\rho}$;
  end
end

The gradient calculation is done in algorithm 65. For this to work, we need the circumference of the planet. Herefore we need to assume that the planet is a sphere. While that is not technically true, it makes little difference in practice and is good enough for our model. The equation for the circumference can be found in Equation 32 [29], where r is the radius of the planet. Here we also use the f-plane approximation, where the coriolis paramter has one value for the northern hemisphere and one value for the southern hemisphere [27].

$$2\pi r \tag{32}$$

Because of the geometry of the planet and the construction of the longitude latitude grid, we run into some problems when calculating the gradient. Since the planet is not flat ("controversial I know" [17]) whenever we reach the end of the longitude we need to loop around to get to the right spot to calculate the gradients (as the planet does not stop at the end of the longitude line but loops around). So to fix that we use the modulus (mod) function which does the looping for us if we exceed the grid's boundaries. We do haveanother problem though, the poles. As the latitude grows closer to the poles, they are converging on the center point of the pole. Looping around there is much more difficult so to fix it, we just do not consider that center point in the main loop. The changed algorithm can be found in algorithm 66

Algorithm 65: Calculating the gradient δx (note that this algorithm is obsolete)

$C \leftarrow 2\pi R$;
$\delta y \leftarrow \frac{C}{nlat}$;
for $lat \in [-nlat, nlat]$ **do**
  $\delta x[lat] \leftarrow \delta y \cos(lat \cdot \frac{\pi}{180})$;
  if $lat < 0$ **then**
    $f[lat] \leftarrow -10^{-4}$;
  else
    $f[lat] \leftarrow 10^{-4}$;
  end
end

Algorithm 66: The main loop of the velocity of the atmosphere calculations

for $lat \in [-nlat + 1, nlat - 1]$ **do**
  for $lon \in [0, nlon]$ **do**
    $u[lat, lon] \leftarrow -\frac{p[(lat+1) \bmod nlat, lon] - p[(lat-1) \bmod nlat, lon]}{\delta y} \cdot \frac{1}{f[lat]\rho}$;
    $v[lat, lon] \leftarrow \frac{p[lat, (lon+1) \bmod nlon] - p[lat, (lon-1) \bmod nlon]}{\delta x[lat]} \cdot \frac{1}{f[lat]\rho}$;
  end
end

Do note that the pressure calculation is done between the temperature calculation in algorithm 13 and the u, v calculations in algorithm 66. At this point our model shows a symmetric vortex around the sun that moves with the sun. This is not very realistic as you usually have convection and air flowing from warm to cold, but we do not have that complexity yet (due to our single layer atmosphere).

C List of Variables

Are you ever confused about what something is? Do you ever forget what a variable represents? Then I got the solution for you. The following overview will explain what each variable is and represents. I will try to not use one variable for the same thing, though that is sometimes very difficult to do. I'll do my best. In the meantime, enjoy this exstensive list. Note that this only applies to variables in code, every symbol in equations are explained at the equations themselves.

- R: The Gas Constant with value 8.3144621 $(\text{J(mol)}^{-1}\text{K})$.
- *day*: Length of one day in seconds (s).
- *year*: Length of one year in seconds (s).
- δt: How much time is between each calculation run in seconds (s).
- g : Magnitude of gravity on the planet in ms^{-2}.
- α: By how many degrees the planet is tilted with respect to the star's plane, also called axial tilt.
- *top*: How high the top of the atmosphere is with respect to the planet surface in meters (m).
- *ins*: Amount of energy from the star that reaches the planet per unit area (Jm^{-2}).
- ϵ: Absorbtivity of the atmosphere, fraction of how much of the total energy is absorbed (unitless).
- *resolution*: The amount of degrees on the latitude longitude grid that each cell has, with this setting each cell is 3 degrees latitude high and 3 degrees longitude wide.
- $nlevels \leftarrow 10$: The amount of layers in the atmosphere.
- δt_s: The time between calculation rounds during the spin up period in seconds (s).
- t_s: How long we let the planet spin up in seconds (s).
- *adv*: Whether we want to enable advection or not.
- *velocity*: Whether we want to calculate the air velocity or not.
- *adv_boun*: How many cells away from the poles where we want to stop calculating the effects of advection.
- *nlon*: The amount of longitude gridpoints that we use, which depends on the resolution.
- *nlat*: The amount of latitude gridpoints that we use, which depends on the resolution.
- T_p: The temperature of the planet, a 2D array representing a latitude, longitude grid cell.
- T_a: The temperature of the atmosphere, a 3D array representing a grid cell on the latitude, longitude, atmospheric layer grid.
- σ: The Stefan-Boltzmann constant equal to $5.670373 \cdot 10^{-8}$ $(\text{Wm}^{-2}\text{K}^{-4})$.
- C_a: Specific heat capacity of the air, equal to 1.0035 $\text{Jg}^{-1}\text{K}^{-1}$.
- C_p: Specific heat capacity of the planet, equal to $1.0 \cdot 10^6$ $\text{Jg}^{-1}\text{K}^{-1}$.
- a: Albedo, the reflectiveness of a substance. Note that a is used in general functions as an array that is supplied as input. If that is the case it can be read at the top of the algorithm.
- ρ: The density of the atmosphere, a 3D array representing a grid cell on the latitude, longitude, atmospheric layer grid.

- δx: How far apart the gridpoints are in the x direction in degrees longitude.
- δy: How far apart the gridpoints are in the y direction in degrees latitude.
- p_z : The vertical pressure coordinate in Pascals (Pa).
- τ: The optical depth for an atmospheric layer.
- τ_0: The optical depth at the planet surface.
- f_l: The optical depth parameter.
- $pressureProfile$: The average pressure taken over all atmospheric layers in a latitude, longitude gridcell in Pascals (Pa).
- $densityProfile$:The average density taken over all atmospheric layers in a latitude, longitude gridcell.
- $temperatureProfile$: The average temperature taken over all atmospheric layers in a latitude, longitude gridcell in degrees Kelvin (K).
- U: Upward flux of radiation, 1D array representing an atmospheric layer.
- D: Downward flux of radiation, 1D array representing an atmospheric layer.
- u: The east to west air velocity in meters per second (ms^{-1}).
- v: The north to south air velocity in meters per second (ms^{-1}).
- w: The bottom to top air velocity in ms^{-1}.
- f: The coriolis parameter.
- Ω: The rotation rate of the planet in rads^{-1}.
- p: The pressure of a latitude, longitude, atmospheric layer gridcell in Pascals (Pa).
- p_0: The pressure of a latitude, longitude, atmospheric layer gridcell from the previous calculation round in Pascals (Pa).
- α_a: The thermal diffusivity constant for air.
- α_p: The thermal diffusivity constant for the planet surface.
- $smooth_t$: The smoothing parameter for the temperature.
- $smooth_u$: The smoothing parameter for the u component of the velocity.
- $smooth_v$: The smoothing parameter for the v component of the velocity.
- $smooth_w$: The smoothing parameter for the w component of the velocity.
- $smooth_{vert}$: The smoothing parameter for the vertical part of the velocities.
- r: Radius of the planet in m.

References

[1] 145.254.33.233. Geopotential height. https://en.wikipedia.org/w/index.php?title=Geopotential_height&oldid=985531215, Jan 2003.

[2] 155.42.27.xxx (Anonymous Internet User). Albedo. https://en.wikipedia.org/wiki/Albedo, Jun 2020.

[3] Robert Alexander Adams and Christopher Essex. *Calculus a complete course*, chapter 15, page 867. Pearson, 9th edition, 2018.

[4] Robert Alexander Adams and Christopher Essex. *Calculus a complete course*, chapter 16, page 923. Pearson, 9th edition, 2018.

[5] Robert Alexander Adams and Christopher Essex. *Calculus a complete course*, chapter 1, page 62. Pearson, 9th edition, 2018.

[6] Robert Alexander Adams and Christopher Essex. *Calculus a complete course*, chapter 7, page 412. Pearson, 9th edition, 2018.

[7] Alistair Adcroft. Vertical coordinates, 2004.

[8] National Aeronautics and Space Administration. *U.S. standard atmosphere*. National Oceanic and Atmospheric Administration, 1976.

[9] Sam Baggen. Techwizzart/claude. https://github.com/TechWizzart/claude.

[10] Dick Beldin. Interpolation. https://en.wikipedia.org/wiki/Interpolation, May 2020.

[11] Axel Boldt. Pythagorean theorem. https://en.wikipedia.org/wiki/Pythagorean_theorem, Sep 2001.

[12] Axel Boldt. Equinox. https://en.wikipedia.org/wiki/Euler%27s_formula, September 2020.

[13] Isabel Chavez. Si units. https://www.nist.gov/pml/weights-and-measures/metric-si/si-units, Nov 2019.

[14] Isabel Chavez. Si units - amount of substance. https://www.nist.gov/pml/weights-and-measures/si-units-amount-substance, Nov 2019.

[15] Simon Clark. https://www.twitch.tv/drsimonclark.

[16] Simon Clark. https://www.twitch.tv/collections/ewOcca6DExadGg.

[17] Simon Clark.

[18] Simon Clark. Planet-factory/claude. https://github.com/Planet-Factory/claude.

[19] Simon Clark. https://www.twitch.tv/videos/715183053, Aug 2020.

[20] The SciPy community. numpy.meshgrid¶, Jun 2020.

[21] The SciPy community. scipy.interpolate.rectbivariatespline¶, Nov 2020.

[22] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–297, 1965.

[23] Thomas H. Cormen, Charles E, Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 30, page 907. MIT Press, 3rd edition, 2009.

[24] Dargan M. W. Frierson, Isaac M. Held, and Pablo Zurita-Gotor. A gray-radiation aquaplanet moist gcm. part i: Static stability and eddy scale. *Journal of the Atmospheric Sciences*, 63(10):2548–2566, 2006.

[25] Arijit Kumar Gayen. Numpy meshgrid function, Apr 2019.

[26] Tobias Hoevekamp. Geographic coordinate system. https://en.wikipedia.org/wiki/Geographic_coordinate_system#Latitude_and_longitude, Jul 2020.

[27] Nathan Johnson. F-plane. https://en.wikipedia.org/wiki/F-plane, Apr 2020.

[28] Kjkolb. Coriolis frequency. https://en.wikipedia.org/wiki/Coriolis_frequency, Jun 2020.

[29] MBManie. Circumference. https://en.wikipedia.org/wiki/Circumference, Jun 2020.

[30] Douglas C. Montgomery and George C. Runger. *Applied statistics and probability for engineers*, chapter 3, page 49. Wiley, 7th edition, 2018.

[31] Karl Palmen. Equinox. https://en.wikipedia.org/wiki/Equinox, Jun 2020.

[32] 192.2.69.128 (Anonymous Internet User). Gas constant. https://en.wikipedia.org/wiki/Gas_constant#Specific_gas_constant, Jun 2020.

[33] Geoffrey K. Vallis. *Atmospheric and Oceanic Fluid Dynamics: Fundamentals and Large-Scale Circulation*, chapter 2, page 69. Cambridge University Press, 2nd edition, 2017.

[34] Geoffrey K. Vallis. *Atmospheric and Oceanic Fluid Dynamics: Fundamentals and Large-Scale Circulation*, chapter 13, page 473. Cambridge University Press, 2nd edition, 2017.

[35] Geoffrey K. Vallis. *Atmospheric and Oceanic Fluid Dynamics: Fundamentals and Large-Scale Circulation*, chapter 1, page 8. Cambridge University Press, 2nd edition, 2017.

[36] Geoffrey K. Vallis. *Atmospheric and Oceanic Fluid Dynamics: Fundamentals and Large-Scale Circulation*, chapter 1, page 24. Cambridge University Press, 2nd edition, 2017.

[37] Geoffrey K. Vallis, Greg Colyer, Ruth Geen, Edwin Gerber, Martin Jucker, Penelope Maher, Alexander Paterson, Marianne Pietschnig, James Penn, Stephen I. Thomson, and et al. Isca, v1.0: a framework for the global modelling of the atmospheres of earth and other planets at varying levels of complexity. *Geoscientific Model Development*, 11(3):843–859, 2018.

[38] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[39] Hugh D. Young, Roger A. Freedman, and A. Lewis Ford. *Sears and Zemanskys University physics with modern physics*, chapter 39, page 1328. Pearson Education, 14th global edition, 2016.

[40] Hugh D. Young, Roger A. Freedman, and A. Lewis Ford. *Sears and Zemanskys University physics with modern physics*, chapter 18, page 610. Pearson Education, 14th global edition, 2016.

[41] Hugh D. Young, Roger A. Freedman, and A. Lewis Ford. *Sears and Zemanskys University physics with modern physics*, chapter 17, page 581. Pearson Education, 14th global edition, 2016.

[42] Hugh D. Young, Roger A. Freedman, and A. Lewis Ford. *Sears and Zemanskys University physics with modern physics*, chapter 19, page 648. Pearson Education, 14th global edition, 2016.

[43] Hugh D. Young, Roger A. Freedman, and A. Lewis Ford. *Sears and Zemanskys University physics with modern physics*, chapter 12, page 394. Pearson Education, 14th global edition, 2016.

[44] Hugh D. Young, Roger A. Freedman, and A. Lewis Ford. *Sears and Zemanskys University physics with modern physics*, chapter 7, pages 227–247. Pearson Education, 14th global edition, 2016.