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

Sam "TechWizard" Baggen

August 19, 2020

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 [10]. Each subsequent section starts with a number, this number indicates which coding stream corresponds to that section. This does not only make it easier to cross reference but if the explanation is unclear or you just want to watch the stream about that specific topic, you know which stream to watch. There is a useful playlist on Simon's Twitch which has all the streams without ad breaks or interruptions [11].

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 [8] 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 formular 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 [10].

This manual is for the toy model, which is as of now still in development. Therefore this manual will be in chronological order, explaining everything in the same order as it has been done. There are plans to eventually modularise the whole model into seperate parts that can be extended by the community. When that will hit development, a new manual for that version of the model will be made that treats things per topic instead of chronological order.

One important thing to note, 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.

1 The Beginning

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

The beginning of CLAuDE is based upon one of the most important laws of physics: "Energy is neither created nor destroyed, only changed from one form to another." In otherwords, if energy goes into an object it must equal the outflowing energy plus the change of internal energy. This is captured in Stefan-Boltzmann's law (Equation 1a) [25].

Here we assume that the planet is a black body, i.e. it absorbs all radiation (energy waves, some waves are visible like light, others are invisible like radio signals) on all wavelengths. In Equation 1a the symbols are:

- S: The energy that reaches the top of the atmosphere, coming from the sun or a similar star, per meters squared Jm^{-2}. This is also called the insolation.
- σ: The Stefan-Boltzmann constant, $5.670373 \cdot 10^{-}8$ ($Wm^{-2}K^{-4}$) [25].

- T: The temperature of the planet (K).

Technically speaking Equation 1a is incorrect, as there is a mismatch in units. However, that is corrected in Equation 2d so there is no need to worry about it just yet. The energy difference between the energu 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 1b. The symbols on the right hand side are:

- ΔU: The change of internal energy (J) [26].
- C: The specific heat capacity of the object, i.e. how much energy is required to heat the object by one degree Kelvin ($\frac{J}{K}$).
- ΔT: The change in temperature (K).

We want to know the change of temperature ΔT, so we rewrite the equation into Equation 1c. 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 needed to make the units match, and by adding this time step the units all match up perfectly.

$$S = SB = \sigma T^4 \tag{1a}$$

$$S - \sigma T^4 = \Delta U = C\Delta T \tag{1b}$$

$$\Delta T = \frac{\delta t(S - \sigma T^4)}{C} \tag{1c}$$

The set of equations in Equation 1 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 2. We view the energy reacing the atmosphere as a circular area of energy, with the equation for the are of a circle being Equation 2a [2]. The area of a sphere is in Equation 2b [3]. In both equations, r is the radius of the circle/sphere. By using Equation 2a and Equation 2b in Equation 1c we get Equation 2c 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 2d as the final equation we are going to use.

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

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

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

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

1.2 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 3. 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 2d 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 3b to Equation 2d in Equation 3c. In Equation 3a, ϵ is the absorbtivity of the atmosphere, the fraction of energy that the atmosphere absorbs. We divided Equation 3a by πR^2 as we did that with Equation 2d as well, so we needed to make it match that division.

$$4\pi R^2 \epsilon \sigma T_a^4 \tag{3a}$$

$$4\epsilon \sigma T_a^4 \tag{3b}$$

$$\Delta T_p = \frac{\delta t(S + 4\epsilon\sigma T_a^4 - 4\sigma T_p^4)}{4C_p} \tag{3c}$$

As you probably expected, the atmosphere can change in temperature as well. This is modelled by Equation 4, which is very similar to Equation 3c. 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.

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

1.3 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 [14]. 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 1 defines the main loop 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). 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.

Algorithm 1: The main loop of the temperature calculations

$\delta t \leftarrow 60 \cdot 5$;
$\sigma \leftarrow 5.67 \cdot 10^{-8}$;
$\epsilon \leftarrow 0.75$;
$C_p \leftarrow 10^7$;
$C_a \leftarrow 10^6$;
$S \leftarrow 1370$;
$R \leftarrow 6.4 \cdot 10^6$;
$t \leftarrow 0$;
while *TRUE* **do**
  for $lat \in [-90, 90]$ **do**
    for $lon \in [0, 360]$ **do**
      $T_p[lat, lon] \leftarrow T_p[lat, lon] + \frac{\delta t(S + 4\epsilon\sigma(T_a[lat,lon])^4 - 4\sigma(T_p[lat,lon])^4)}{C_p}$;
      $T_a[lat, lon] \leftarrow T_a[lat, lon] + \frac{\delta t(\sigma(T_p[lat,lon])^4 - 2\epsilon\sigma(T_a[lat,lon])^4)}{C_a}$;
      $t \leftarrow t + \delta t$;
    end
  end
end

1.4 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 the correct amount of energy that lands on that part of the planet surface. This is done in algorithm 2. 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) [19] 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. When we did the calculation specified in algorithm 2 we return the final value (which means that the function call is "replaced" [1] 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).

In the second stream, it was revealed that t mod d in algorithm 2 should be $-t$ mod d such that the sun moves in the right direction. In the first stream the sun would move to the right (west to east), however the sun moves to the left (east to west) and so the time must be flipped in order for the model to be correct.

[1] 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.

Algorithm 2: 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 2, algorithm 1 must be changed as well, as S is no longer constant for the whole planet surface. So let us do that in algorithm 3. Note that S is defined as the call to algorithm 2 (as is showcased by the text `solar`). In case you are unfamiliar with calls, defining a function is defining how it works and calling a function is actually using it.

Algorithm 3: The main loop of the temperature calculations

$\delta t \leftarrow 60 \cdot 5$;
$\sigma \leftarrow 5.67 \cdot 10^{-8}$;
$\epsilon \leftarrow 0.75$;
$C_p \leftarrow 10^7$;
$C_a \leftarrow 10^7$;
$I \leftarrow 1370$;
$R \leftarrow 6.4 \cdot 10^6$;
$t \leftarrow 0$;
$day \leftarrow 60 \cdot 60 \cdot 24$;
$S \leftarrow$ `solar(`I`,` lat`,` lon`,` t`,` day`)` ;
$nlat$ is the amount of latitude points in the interval $[0, 90]$, how you divide them is your own choice. ;
$nlot$ is the amount of longitude points in the interval $[0, 360]$, how you divide them is your own choice. ;
while *TRUE* **do**
  for $lat \in [-nlat, nlat]$ **do**
    for $lon \in [0, nlot]$ **do**
      $T_p[lat, lon] \leftarrow T_p[lat, lon] + \frac{\delta t(S+4\epsilon\sigma(T_a[lat,lon])^4-4\sigma(T_p[lat,lon])^4)}{C_p}$;
      $T_a[lat, lon] \leftarrow T_a[lat, lon] + \frac{\delta t(\sigma(T_p[lat,lon])^4-2\epsilon\sigma(T_a[lat,lon])^4)}{C_a}$;
      $t \leftarrow t + \delta t$;
    end
  end
end

algorithm 3 calculates the values that are plotted (which is not discussed here as that is Python specific). Due to the `WHILE(TRUE)` loop, this calculation never finishes and allows us to simulate days, weeks, months and even years of heat exchange all conveniently plotted in a graph. In Simon's implementation, the graphs update in realtime, meaning that whenever a round of calculations has finished, they are immediately processed to be displayed in the graph.

However other forms of looking at the calculated data can be implemented, like writing a table to a txt file, saving the generated grpahs at a certain interval or spewing all the data into a csv dataset. The possibilities are endless, and the whole goal of the model is for it to be modular. Meaning that if you want to do something with it (like have a multi-layered atmosphere instead of a single layer atmosphere) you can

just write some lines of code and run the model and it should still work. Therefore you can write your own extensions of the model to fit it to your needs and requirements.

2 Let's Get the Atmosphere Moving

In its current state, CLaUDE has a static planet. This means that the planet remains in place and does not move. However we know that planets move in orbit and more importantly, spin around themselves. But before we start adding layers, let's talk about a term you will hear more often: 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 i for a bit. You will see that the whole system oscilates wildly as it adjusts to the sudden changes and eventually it will stabilise. Another term you might encounter is blow up, this occurs when when the model suddenly no longer behaves like it should. This is most likely caused by mistakes in the code or incorrect paramter initialisation. Be wary of the existence of both factors, and do not dismiss a model if it behaves weirdly as it has just started up.

2.1 Equation of State and the Incompressible Atmosphere

The equation of state relates one or more variables in a dynamical system (like the atmosphee) to another. The most common equation of state in the atmosphere is the ideal gas equation as described by Equation 5a [27]. The symbols in that equation represent:

- p: The gas pressure (Pa).
- V: The volume of the gas (m^3).
- n: The amount of moles[2] in the gas.
- R: The Gas constant, 8.3144621 ($J(mol)^{-1}K$) [27].
- T: The temperature opf the gas (K).

If we divide everything in Equation 5a by V and set it to be unit (in this case, set it to be exactly $1m^3$) we can add in the molar mass in both the top and bottom parts of the division as show in Equation 5b. 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 $J(mol)^{-1}K$) as shown in Equation 5c. the resulting equation is the equation of state that you get that most atmospheric physicists use when talking about the atmosphere [12].

$$pV = nRT \tag{5a}$$

$$p = \frac{nR}{V}T = \frac{nmR}{Vm}T \tag{5b}$$

$$p = \rho R_s T \tag{5c}$$

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.

[2]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) [9]

The code that corresponds to this is quite simple, the only change that we need to make in Equation 5c 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. $\rho = 1.2$ [28], $R_s = 287$ [20].

2.2 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! [12]). The primitive equations are a set of equations dictating the direction in the u and v directions as shown in Equation 6a and Equation 6b. 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 6a and Equation 6b 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 6c and Equation 6d.

$$\frac{du}{dt} = \frac{\delta u}{\delta t} + u\frac{\delta u}{\delta x} + v\frac{\delta u}{\delta v} + \omega\frac{\delta u}{\delta p} = -\frac{\delta \Phi}{\delta x} + fv \tag{6a}$$

$$\frac{dv}{dt} = \frac{\delta v}{\delta t} + u\frac{\delta v}{\delta x} + v\frac{\delta v}{\delta v} + \omega\frac{\delta v}{\delta p} = -\frac{\delta \Phi}{\delta y} - fu \tag{6b}$$

$$0 = -\frac{\delta \Phi}{\delta x} + fv \tag{6c}$$

$$0 = -\frac{\delta \Phi}{\delta y} - fu \tag{6d}$$

Equation 6c 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 6d. Effectively we have a balance between the gradient and the coriolis force as shown in Equation 7b and Equation 7c. 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 7a, where Ω is the rotation rate of the planet (for Earth $7.2921 \cdot 10^{-5}$) ($rad\ s^{-1}$) and θ is the latitude [16].
- u: The velocity in the latitude (ms^{-1}).
- v: The velocity in the longitude (ms^{-1}).

$$f = 2\Omega \sin(\theta) \tag{7a}$$

$$\frac{\delta \Phi}{\delta x} = fv \tag{7b}$$

$$\frac{\delta \Phi}{\delta y} = -fu \tag{7c}$$

$$\frac{\delta p}{\rho \delta x} = fv \tag{7d}$$

$$\frac{\delta p}{\rho \delta y} = -fu \tag{7e}$$

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 3, 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 4. Do note that everything in algorithm 3 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 4: The main loop of the velocity of the atmosphere calculations

```
while TRUE do
    for lat ∈ [−nlat, nlat] do
        for lon ∈ [0, nlon] do
            u[lat, lon] ← −(p[lat+1,lon] − p[lat−1,lon]) / δy · 1 / (f[lat]ρ) ;
            v[lat, lon] ← (p[lat,lon+1] − p[lat,lon−1]) / (δx[lat]) · 1 / (f[lat]ρ) ;
        end
    end
end
```

The gradient calculation is done in algorithm 5. 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 8 [17], 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 [15].

$$2\pi r \tag{8}$$

Algorithm 5: Calculating the gradient δx

```
C ← 2πR ;
δy ← C / nlat ;
for lat ∈ [−nlat, nlat] do
    δx[lat] ← δy cos(lat · π/180) ;
    if lat < 0 then
        f[lat] ← −10^−4 ;
    else
        f[lat] ← 10^−4 ;
    end
end
```

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" [12]) 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 6

Algorithm 6: The main loop of the velocity of the atmosphere calculations

while *TRUE* **do**
  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
end

Do note that the pressure calculation is done between the temperature calculation in algorithm 3 and the u, v calculations in algorithm 6. 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).

2.3 Introducing an Ocean

Now we want to introduce an ocean, because most of the Earth is covered by oceans it plays quite an important role in atmospheric physics. To do this we need a new concept called albedo. Albedo is basically the reflectiveness of a material (in our case the planet's surface) [1]. The average albedo of the Earth is about 0.3. Now to add an ocean to the grid, we define a few areas where the albedo differs. Where you do this does not really matter for the current complexity. Defining the oceans is as easy as hardcoding (what we computer scientists refer to when setting parts of an array to be a specific value, where if you want to change the value you need to change it everywhere instead of doing it in a variable) the albedo value for the specific regions as we do in algorithm 7. Water also takes longer to warm up, so let us change the specific heat capacity (C_p in algorithm 3) from a constant to an array. The new C_p can also be found in algorithm 7, where we have made the specific heat capacity of water one order of magnitude (i.e. 10 times) larger.

Algorithm 7: Defining the oceans

$a \leftarrow 0.5$;
$a[5-55, 9-20] \leftarrow 0.2$;
$a[23-50, 45-70] \leftarrow 0.2$;
$a[2-30, 85-110] \leftarrow 0.2$;
$C_p \leftarrow 10^7$;
$C_p[5-55, 9-20] \leftarrow 10^8$;
$C_p[23-50, 45-70] \leftarrow 10^8$;
$C_p[2-30, 85-110] \leftarrow 10^8$;

Now that we have that defined, we need to adjust the main loop of the program (algorithm 3). For clarity, all the defined constants have been left out. 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 8. 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 8: The main loop of the temperature calculations

while *TRUE* **do**
  for $lat \in [-nlat, nlat]$ **do**
    for $lon \in [0, nlot]$ **do**
      $T_p[lat, lon] \leftarrow T_p[lat, lon] + \frac{\delta t((1-a[lat,lon])S + 4\epsilon\sigma(T_a[lat,lon])^4 - 4\sigma(T_p[lat,lon])^4)}{C_p[lat,lon]}$;
      $T_a[lat, lon] \leftarrow T_a[lat, lon] + \frac{\delta t(\sigma(T_p[lat,lon])^4 - 2\epsilon\sigma(T_a[lat,lon])^4)}{C_a}$;
      $t \leftarrow t + \delta t$;
    end
  end
end

3 Adding Mass to CLAuDE

3.1 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 9a and Equation 9b [21]. Note that we are ignoring vertical moevement, as this does not have a significant effect on the whole flow. All the symbols in Equation 9a and Equation 9b mean:

- u: The east to west velocity (ms^{-1}).
- t: The time (s).
- f: The coriolis parameter as in Equation 7a.
- 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 9a as Equation 9c. Here ∇u is the gradient of u in both x and y directions. Then if we write out ∇u we get Equation 9e. Similarly, if we want to get δv instead of δu we rewrite Equation 9b to get Equation 9d and Equation 9f.

$$\frac{Du}{Dt} - fv = -\frac{1}{\rho}\frac{\delta p}{\delta x} \tag{9a}$$

$$\frac{Dv}{Dt} - fu = -\frac{1}{\rho}\frac{\delta p}{\delta y} \tag{9b}$$

$$\frac{\delta u}{\delta t} + \bar{u} \cdot \nabla u - fv = -\frac{1}{\rho}\frac{\delta p}{\delta x} \tag{9c}$$

$$\frac{\delta v}{\delta t} + \bar{u} \cdot \nabla v - fu = -\frac{1}{\rho}\frac{\delta p}{\delta y} \tag{9d}$$

$$\frac{\delta u}{\delta t} + u\frac{\delta u}{\delta x} + v\frac{\delta u}{\delta y} - fv = -\frac{1}{\rho}\frac{\delta p}{\delta x} \tag{9e}$$

$$\frac{\delta v}{\delta t} + u\frac{\delta v}{\delta x} + v\frac{\delta v}{\delta y} - fu = -\frac{1}{\rho}\frac{\delta p}{\delta y} \tag{9f}$$

Now that we have the momentum equations sorted out, we need to define a method to do the gradient calculations for us. Therefore we define two functions algorithm 9 and algorithm 10 that calculate the x and y gradients respectively.

Algorithm 9: Calculating the gradient in the x direction

Input : Matrix (double array) a, first index i, second index j
Output: Gradient in the x direction
$grad \leftarrow \frac{a[i,(j+1) \bmod nlon] - a[i,(j-1) \bmod nlon]}{\delta x[i]}$;
return $grad$;

Algorithm 10: Calculating the gradient in the y direction

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

With the gradient functions defined, we can move on to the main code for the momentum equations. The main loop is shown in algorithm 11. Do note that this loop replaces the one in algorithm 6 as these calculate the same thing, but the new algorithm does it better.

Algorithm 11: Calculating the flow of the atmosphere (wind)

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

3.2 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). 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 10, describes how the temperature spreads out over time [22]. 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 A.2.
- $\bar{u}$: The time derivative of u, or in symbols $\frac{\delta u}{\delta t}$.

$$\bar{u} = \alpha \nabla^2 u \tag{10}$$

Now to get this into code we need the following algorithms algorithm 12 and algorithm 13. algorithm 12 implements the laplacian operator, whereas algorithm 13 implements the diffusion calculations. Δ_x and Δ_y in algorithm 12 represents the calls to algorithm 9 and algorithm 10 respectively. ∇^2 in algorithm 13 represents the call to algorithm 12.

Algorithm 12: 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
for $lat \in [1, nlat - 1]$ **do**
  for $lon \in [0, nlon]$ **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
return *ouput* ;

Algorithm 13: The main loop for calculating the effects of diffusion

$\alpha_a \leftarrow 2 \cdot 10^{-5}$;
$\alpha_p \leftarrow 1.5 \cdot 10^{-6}$;
while *TRUE* **do**
  $T_a \leftarrow T_a + \delta t \alpha_a \nabla^2(T_a)$;
  $T_p \leftarrow T_p + \delta t \alpha_p \nabla^2(T_p)$;
end

3.3 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. 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. The advection equation is shown in Equation 11. 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 A.2).

$$\frac{\delta\psi}{\delta t} + \nabla \cdot (\psi u) = 0 \tag{11}$$

As we expect to use the divergence operator more often throughout our model, let us define a seperate function for it in algorithm 14. Δ_x and Δ_y in algorithm 14 represents the calls to algorithm 9 and algorithm 10 respectively. We do the multiplication with the velocity vector here already, as we expect that we might use it in combination with the divergence operator more frequently.

Algorithm 14: Calculate the result of the divergence operator on a vector

Input : A matrix (double array) a
Output: A matrix (double array) containing the result of the divergence operator taken over that element
$dim_1 \leftarrow$ Length of a in the first dimension ;
for $i \in [0, dim_1]$ **do**
  $dim_2 \leftarrow$ Length of a in the second dimension (i.e. the length of the array stored at index i) ;
  for $j \in [0, dim_2]$ **do**
    $output[i, j] \leftarrow \Delta_x(au, i, j) + \Delta_y(av, i, j)$;
  end
end
return *output* ;

With the divergence functon defined, we now need to adjust algorithm 13 to incorporate this effect. The resulting algorithm can be found in algorithm 15. Here ∇ represents the function call to algorithm 14.

Algorithm 15: The main loop for calculating the effects of advection

$\alpha_a \leftarrow 2 \cdot 10^{-5}$;
$\alpha_p \leftarrow 1.5 \cdot 10^{-6}$;
while *TRUE* **do**
  $T_{add} \leftarrow T_a + \delta t \alpha_a \nabla^2(T_a) + \nabla(T_a)$;
  $T_a \leftarrow T_a + T_{add}[5 : -5, :]$ //Only add T_{add} to T_a for indices in the interval $[-nlat + 5, nlat - 5]$. ;
  $T_p \leftarrow T_p + \delta t \alpha_p \nabla^2(T_p)$;
end

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 12 [23].

$$\frac{\delta\rho}{\delta t} + \nabla \cdot (\rho v) = 0 \tag{12}$$

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 11. 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 15 as well to include the calculations for ρ. The new version can be found in algorithm 16. Again the ∇ represents the call to algorithm 14.

Now that we have a varying density, we need to account for that in the temperature equations. So let us do that. We need it in the denominator as the density has a direct effect on the heat capacity of the atmosphere. The changes are reflected in algorithm 17.

Algorithm 16: The main loop for calculating the effects of advection

$\alpha_a \leftarrow 2 \cdot 10^{-5}$;
$\alpha_p \leftarrow 1.5 \cdot 10^{-6}$;
while *TRUE* **do**
  $T_{add} \leftarrow T_a + \delta t \alpha_a \nabla^2(T_a) + \nabla(T_a)$;
  $T_a \leftarrow T_a + T_{add}[5 : -5, :]$ //Only add T_{add} to T_a for indices in the interval $[-nlat + 5, nlat - 5]$. ;
  $\rho \leftarrow \rho + \delta t \nabla \rho$;
  $T_p \leftarrow T_p + \delta t \alpha_p \nabla^2(T_p)$;
end

Algorithm 17: The main loop of the temperature calculations

while *TRUE* **do**
  for $lat \in [-nlat, nlat]$ **do**
    for $lon \in [0, nlot]$ **do**
      $T_p[lat, lon] \leftarrow T_p[lat, lon] + \frac{\delta t((1-a[lat,lon])S + 4\epsilon\sigma(T_a[lat,lon])^4 - 4\sigma(T_p[lat,lon])^4)}{\rho[lat,lon]C_p[lat,lon]}$;
      $T_a[lat, lon] \leftarrow T_a[lat, lon] + \frac{\delta t(\sigma(T_p[lat,lon])^4 - 2\epsilon\sigma(T_a[lat,lon])^4)}{\rho[lat,lon]C_a}$;
      $t \leftarrow t + \delta t$;
    end
  end
end

3.4 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 7a, where Θ is the latitude. Coriolis (f) is currently defined in algorithm 5, so let's incorporate the changes which are shown in algorithm 18.

Algorithm 18: Calculating the gradient δx

$C \leftarrow 2\pi R$;
$\delta y \leftarrow \frac{C}{nlat}$;
$\Omega \leftarrow 7.2921 \cdot 10^{-5}$;
for $lat \in [-nlat, nlat]$ **do**
  $\delta x[lat] \leftarrow \delta y \cos(lat \cdot \frac{\pi}{180})$;
  $f[lat] \leftarrow 2\Omega \sin(lat \frac{\pi}{180})$;
end

4 Removing Some Assumptions and Mistakes from CLAuDE

The first half of this stream was spent looking through the code and fixing some mistakes. To spare you dear reader from making these same mistakes, they have already been incorporated into the previous sections, hooray! This does not only save you some work, but it also spares you from staring at a model that does not function due to wrongly defined constants or using the wrong values.

4.1 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 calculations in algorithm 17 we would calculate the velocities and afterwards we would calculate the advection. Instead let us change it to what is shown in algorithm 19.

Algorithm 19: Main loop that can simulate flow and advection conditionally

```
while TRUE do
    algorithm 17 ;
    if velocity then
        algorithm 11 ;
        if advection then
            algorithm 16 ;
        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 19, before algorithm 17 we add algorithm 20. 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. Currently it will break the model, so I recommend leaving it on `FALSE` until it is fixed in subsection 4.3.

Algorithm 20: 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
```

4.2 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 21. The uniform distribution basically says that each allowed value in the interval has an equal chance of being picked [18].

Algorithm 21: Varying the albedo of the planet

```
V_a ← 0.02 ;
for lat ∈ [−nlat, nlat] do
    for lon ∈ [0, nlon] do
        R ← Pick a random number (from the uniform distribution) in the interval [−V_a, V_a] ;
        a[lat, lon] ← a[lat, lon] + V_a · R;
    end
end
```

4.3 Fixing the Advection

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 [12]. 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 22. 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 22: The main loop for calculating the effects of advection

$\alpha_a \leftarrow 2 \cdot 10^{-5}$;
$\alpha_p \leftarrow 1.5 \cdot 10^{-6}$;
$boundary \leftarrow 7$;
while *TRUE* **do**
  $T_{add} \leftarrow T_a + \delta t \alpha_a \nabla^2(T_a) + \nabla(T_a)$;
  $T_a \leftarrow T_a - 0.5 T_{add}[boundary : -boundary, :$
  ] //Only subtract T_{add} to T_a for indices in the interval $[-nlat + boundary, nlat - boundary]$. ;
  $\rho[boundary : -boundary, :] \leftarrow$
  $\rho - 0.5(\delta t \nabla \rho)$ //Only change the density for indices in the interval $[-nlat + boundary, nlat - boundary]$;
  $T_p \leftarrow T_p + \delta t \alpha_p \nabla^2(T_p)$;
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 11 in algorithm 19.

5 Up up and away! Adding More Layers to the Atmosphere

Up until now we have neglected any vertical movement. This hampers the model, as the rising of warm air that then flows to the poles, cools down and flows back to the tropics is not possible as the warm air cannot rise. So let us change that, let's add some vertical motion and some more layers to the atmosphere.

Remember Equation 4? We need this equation for every layer in the atmosphere. This also means that we have to adjust the main loop of the code, which is described in algorithm 17. 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. Let us define some useful variables in algorithm 23. We opt for adding a dimension as that costs less memory than defining new arrays [3]. 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 $lat, lon, layer$.

Algorithm 23: Definition of variables that are used throughout the code

$nlevels \leftarrow 4$;
$heights \leftarrow$ Array with $nlevels$ layers, each with a uniform thickness of $\frac{100 \cdot 10^3}{nlevels} m$;

We also need to change all the gradient functions (algorithm 9 and algorithm 10) to incorporate the atmospheric layers. Additionally we need a new gradient function that calculates the gradient in the z

[3] 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

direction (vertical). Let us first change the existing gradient functions to take the atmospheric layer in effect. The changes can be found in algorithm 24 and algorithm 25. Let us improve the gradient in the y direction as well. Since we are using the central difference method (calculating the gradient by taking the difference of the next grid cell and the previous grid cell) there is no gradient at the poles. What we can do instead of returning 0 for those cases is forward differencing (calculating the gradient by taking the difference of the cell and the next/previous cell, multiplied by 2 to keep it fair). A special change in both functions is checking whether k is equal to `NULL`. We do this as sometimes we want to use this function for matrices that does not have the layer 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 layer part in the gradient.

Algorithm 24: 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 $grad$;

Algorithm 25: 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 $grad$;

With those changes done, let us define the gradient in the z direction. The function can be found in algorithm 26. Here $a.dimensions$ 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.

Algorithm 26: Calculating the gradient in the z direction

Input : Matrix (double array) a, first index i, second index j, third index k
Output: Gradient in the z direction
if $a.dimensions == 1$ **then**
  if $k == 0$ **then**
    $grad \leftarrow \frac{a[k+1]-a[k]}{\delta z[k]}$;
  else if $k == nlevels - 1$ **then**
    $grad \leftarrow \frac{a[k]-a[k-1]}{\delta z[k]}$;
  else
    $grad \leftarrow \frac{a[k+1]-a[k-1]}{2\delta z[k]}$;
else
  if $k == 0$ **then**
    $grad \leftarrow \frac{a[i,j,k+1]-a[i,j,k]}{\delta z[k]}$;
  else if $k == nlevels - 1$ **then**
    $grad \leftarrow \frac{a[i,j,k]-a[i,j,k-1]}{\delta z[k]}$;
  else
    $grad \leftarrow \frac{a[i,j,k+1]-a[i,j,k-1]}{2\delta z[k]}$;
return $grad$;

As you can see, we have used δz however, we have not defined it yet. Let us do that in algorithm 27.

Algorithm 27: Defining δz for later use throughout the code

for $k \in [0, nlevels - 1]$ **do**
  $\delta z[k] \leftarrow heights[k + 1] - heights[k]$;
end
$\delta z[nlevels - 1] \leftarrow \delta z[nlevels - 2]$;

Let's incorporate the changes for the Laplacian operator (algorithm 12) as well. The new code can be found in algorithm 28.

Of course we also need to incorporate the new layers in the divergence operator (algorithm 14). The new changes can be found in algorithm 29. Here we use w, the vertical wind velocity. We define w in the same way as u and v, it is all zeroes (in the beginning) and has the same dimensions as u and v.

With all those changes in the functions done, let us incorporate the changes into the model itself. We now need to account for the temperature change throughout the layers. Let us look at the atmospheric temperature equation again (Equation 4). We need to account for one more thing, the absorbtion of energy from another layer. The new equation is shown in Equation 13a. 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 13b. Let us also clean up the equation for the change in the surface temperature (Equation 3c) in Equation 13c.

$$\Delta T_{a_k} = \frac{\delta t(\sigma\epsilon_{k-1}T_{a_{k-1}}^4 + \sigma\epsilon_{k+1}T_{a_{k+1}}^4 - 2\epsilon_k\sigma T_{a_k}^4)}{C_a} \tag{13a}$$

$$\Delta T_{a_k} = \frac{\delta t\sigma(\epsilon_{k-1}T_{a_{k-1}}^4 + \epsilon_{k+1}T_{a_{k+1}}^4 - 2\epsilon_k T_{a_k}^4)}{C_a} \tag{13b}$$

$$\Delta T_p = \frac{\delta t(S + \sigma(4\epsilon_p T_a^4 - 4T_p^4))}{4C_p} \tag{13c}$$

Algorithm 28: 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 $lat \in [1, nlat-1]$ **do**
    for $lon \in [0, nlon]$ **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 $lat \in [1, nlat-1]$ **do**
    for $lon \in [0, nlon]$ **do**
      for $k \in [0, nlevels-1]$ **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* ;

Algorithm 29: Calculate the result of the divergence operator on a vector

Input : A matrix (double array) a
Output: A matrix (double array) containing the result of the divergence operator taken over that element
$dim_1 \leftarrow$ Length of a in the first dimension ;
for $i \in [0, dim_1]$ **do**
  $dim_2 \leftarrow$ Length of a in the second dimension (i.e. the length of the array stored at index i) ;
  for $j \in [0, dim_2]$ **do**
    $dim_3 \leftarrow$ Length of a in the third dimension ;
    for $k \in [0, dim_3]$ **do**
      $output[i, j] \leftarrow \Delta_x(au, i, j, k) + \Delta_y(av, i, j, k) + \Delta_z(aw, i, j, k)$;
    end
  end
end
return *output* ;

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 30.

We also need to initialise the ϵ value for each layer. We do that in algorithm 31.

Now we need to add vertical winds, or in other words add the w component of the velocity vectors. We do that by editing algorithm 11. We change it to algorithm 32. Here we use gravity (g) instead of the coriolis force (f) and calculate the pressure gradient in the z direction.

Lastly, we need to add the correct indices to the advection algorithm algorithm 22. Let us add it, with algorithm 33 as a result. Here the ':' means all indices of the 3 dimensional matrix.

Algorithm 30: The main loop of the temperature calculations

while *TRUE* **do**
  for $lat \in [-nlat, nlat]$ **do**
    for $lon \in [0, nlot]$ **do**
      for $layer \in [0, nlevels]$ **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
          $T_a[lat, lon, layer] \leftarrow T_a[lat, lon, layer] +$
          $\frac{\delta t\sigma(\epsilon[layer-1](T_a[lat,lon,layer-1])^4+\epsilon[layer+1]T_a[lat,lon,layer+1]-2\epsilon[layer](T_a[lat,lon,layer])^4)}{\rho[lat,lon,layer]C_a\delta z[layer]}$
          ;
        $t \leftarrow t + \delta t$;
      end
    end
  end
end

Algorithm 31: Intialisation of the insulation of each layer (also known as ϵ)

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

6 Making a Dummy THICC Atmospheric Model*

* The naming of this section is decided by the stream name, I did not come up with this [10]. During this stream, a lot of plotting improvements have been made, which is not the scope of this manual and hence has been left out. The plan was to add vertical momentum and advection, though things did not go according to plan...

6.1 Discovering That Things Are Broken

While trying to add vertical momentum, it appears that some parts of the model are broken in their current state. The horizontal advection is one of the things that is broken. If you recall, we needed to use the Laplacian operator in the advection equations (as shown in Equation 10, diffusion is considered a part of advection since diffusion transports energy and matter which is what advection does as well). The Laplacian operator (as shown in algorithm 28) did not work. This is because there was a misplaced bracket causing weird numerical errors. This has been fixed in the code (but was never present in the manual, yay for me) and can be safely enabled, though for this stream we disabled the Laplacian operator as it has a small effect on the total advection (and because it was at this time broken).

Another thing that we found out was broken is the vertical momentum. We tried to add it, ran into problems and just set it to 0 to fix the other problems that occured. One of those problems was a wrong initialisation of the density. We basically told the model that the density is the same on every layer of the atmosphere, which is obviously not true. Hence we need to adjust that. The new initialisation is described in algorithm 34. Note that the $\rho[:, : i]$ notation means that for every index in the first and second dimension,

Algorithm 32: Calculating the flow of the atmosphere (wind)

$S_{xu} \leftarrow \texttt{gradient\_x}(u, lan, lon)$;
$S_{yu} \leftarrow \texttt{gradient\_y}(u, lan, lon)$;
$S_{xv} \leftarrow \texttt{gradient\_x}(v, lan, lon)$;
$S_{yv} \leftarrow \texttt{gradient\_y}(v, lan, lon)$;
$S_{px} \leftarrow \texttt{gradient\_x}(p, lan, lon)$;
$S_{py} \leftarrow \texttt{gradient\_y}(p, lan, lon)$;
$S_{pz} \leftarrow \texttt{gradient\_z}(p, lan, lon)$;
while *TRUE* **do**
  for $lat \in [1, nlat - 1]$ **do**
    for $lon \in [0, nlon]$ **do**
      for $layer \in [0, nlevels]$ **do**
        $u[lan, lon, layer] \leftarrow u[lat, lon, layer] + \delta t \frac{-u[lat,lon,layer]S_{xu} - v[lat,lon,layer]S_{yu} + f[lat]v[lat,lon,layer] - S_{px}}{\rho}$;
        $v[lan, lon, layer] \leftarrow v[lat, lon, layer] + \delta t \frac{-u[lat,lon,layer]S_{xv} - v[lat,lon,layer]S_{yv} - f[lat]u[lat,lon,layer] - S_{py}}{\rho}$;
        $w[lan, lon, layer] \leftarrow w[lat, lon, layer] + \delta t(\frac{S_{pz}}{\rho[lat,lon,layer]} + g)$;
      end
    end
  end
end

Algorithm 33: The main loop for calculating the effects of advection

$\alpha_a \leftarrow 2 \cdot 10^{-5}$;
$\alpha_p \leftarrow 1.5 \cdot 10^{-6}$;
$boundary \leftarrow 7$;
while *TRUE* **do**
  $T_{add} \leftarrow T_a + \delta t \alpha_a \nabla^2(T_a) + \nabla(T_a)$;
  $T_a \leftarrow T_a - 0.5 T_{add}[boundary : -boundary, :, :$] //Only subtract T_{add} to T_a for indices in the interval $[-nlat + boundary, nlat - boundary]$. ;
  $\rho[boundary : -boundary, :, :] \leftarrow \rho - 0.5(\delta t \nabla \rho)$ //Only change the density for indices in the interval $[-nlat + boundary, nlat - boundary]$;
  $T_p \leftarrow T_p + \delta t \alpha_p \nabla^2(T_p)$;
end

only change the value for the index i in the third dimension.

Algorithm 34: 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 Using Python to Model the Earth's Atmosphere

This stream Simon was not feeling that well and it felt like his brain was not working, so be wary of errors! You have been warned. also the resolutin (size of an individual cell on the latitude longitude grid) has been decreased to 5 degrees per cell instead of 3 degrees.

7.1 Interpolating the Air Density

In order to interpolate (see subsection A.3) the air density, 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 [6]. 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 35. 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.

Algorithm 35: 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 ;
  $standardHeight.add(sh)$;
  $standardTemperature.add(st)$;
  $standardDensity.add(sd)$;
end
$densityProfile \leftarrow$ `interpolate`$(heights, standardHeight, standardDensity)$;
$temperatureProfile \leftarrow$ `interpolate`$(heights, standardHeight, standardTemperature)$;
for $alt \in [0, nlevels]$ **do**
  $\rho[:, :, alt] \leftarrow densityProfile[alt]$;
  $T_a[:, :, alt] \leftarrow temperatureProfile[alt]$;
end

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 20, though you still need to figure out what value you need for λ (see subsection A.3). This is left as an exercise for the reader.

7.2 Fixing Vertical Motion

Another attempt was made at fixing the vertical motion. The changes are incorporated in algorithm 33. Do keep in mind that the low air density in the upper layers messes a lot with the vertical motion. In other words, it kinda works but not really. Another idea to help fix it, is to introduce a variable called *top* which indicates the highest point that the atmosphere may have. This value is initialised as $8 \cdot 10^3$ in meters (so 8 km). We then change the definition of *heights* to: An array of uniform thickness of $\frac{top}{nlevels} m$. We also added the δz to algorithm 30 as that was something that was still missing.

The current theory why the vertical velocity is not right is that the vertical thermodynamics may be wrong. This will be investigated further and we will report on this in future sections.

8 Getting Radiation Right in our Climate Model! 3D Motion Here We Come

The time has come to finally fix 3D motion. For this to work, we need to use a radiation scheme, which Simon ~~shamelessly stole~~ got inspired by the Isca project [24]. So he followed the references and found a paper which he is going to use in our model [13]. Great, so let's get into it shall we.

8.1 Grey Radiation Scheme

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 14a and Equation 14b [13]. In those equations, the symbols are:

- U: Upward flux.
- D: Downward flux.
- B: The Stefan-Boltzmann equation (see Equation 1a).
- τ: Optical depth.

$$\frac{dU}{d\tau} = U - B \tag{14a}$$

$$\frac{dD}{d\tau} = B - D \tag{14b}$$

With Equation 14a and Equation 14b 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 15. 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.

$$Q_R = \frac{1}{c_p \rho} \frac{\delta(U - D)}{\delta z} \tag{15}$$

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 16. 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.

$$\tau = \tau_0 [f_l (\frac{p}{p_s}) + (1 - f_l)(\frac{p}{p_s})^4] \tag{16}$$

As one can see, Equation 16 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.

8.2 Getting the equations to code

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 16. The resulting code can be found in algorithm 36. This algorithm replaces the temperature calculations we have done in algorithm 30, 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 17 [12]. 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 1a), just as $D[nlevels]$ is the boundary condition. S_z represents the call to algorithm 26.

$$\tau_0 = 3.75 + \cos(lat \frac{\pi}{90}) \frac{4.5}{2} \tag{17}$$

Note that in this form, it did not work on stream yet. This may be due to a coding error or to a missing equation, constant or something similar. If it turns out to be a simple fix, then it will be fixed in this section. If a lot of other things change in order for the fix to work, then it will probably be a seperate section with a reference to that section here.

Algorithm 36: Adding in radiation

for $lat \in [-nlat, nlat]$ **do**
  for $lon \in [0, nlon]$ **do**
    $pressureProfile \leftarrow p[i, j, :]$;
    $\tau = \tau_0(lat) f_l \frac{pressureProfile}{pressureProfile[0]} + (1 - f_l)(\frac{pressureProfile}{pressureProfile[0]})^4)$;
    $U[0] \leftarrow \sigma T_p[lat, lon]^4$;
    for $level \in [1, nlevels]$ **do**
      $U[level] \leftarrow U[level - 1] + (\tau[level] - \tau[level - 1])(U[level - 1] - \sigma \cdot (mean(T_a[:, :, level]))^4)$;
    end
    $D[nlevels] \leftarrow 0$;
    for $level \in [nlevels - 1, 0]$ **do**
      $D[level] \leftarrow (\tau[level] - \tau[level - 1])(\sigma \cdot (mean(T_a[:, :, level]))^4) - D[level + 1]$;
    end
    for $level \in [0, nlevels]$ **do**
      $Q[level] \leftarrow -\frac{S_z(U-D,0,0,level)}{10^3 \cdot densityProfile[level]}$;
    end
    $T_a[lat, lon, :] \leftarrow T_a[lat, lon, :] + Q$;
  end
end

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 [29]. 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 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 [4]. 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 [5]. But what are the divergence and gradient? The gradient is defined in Equation 18a and the divergence is defined in Equation 18b. 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 [4]. Also, i, j and k are the basis vectors of $\mathbb{R}^{\text{IF}}$, 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 18c.

$$\text{grad } \phi = \nabla\phi = \frac{\delta\phi}{\delta x}i + \frac{\delta\phi}{\delta y}j + \frac{\delta\phi}{\delta z}k \tag{18a}$$

$$\text{div}\Phi = \nabla \cdot \Phi = \frac{\delta\Phi_1}{\delta x} + \frac{\delta\Phi_2}{\delta y} + \frac{\delta\Phi_3}{\delta z} \tag{18b}$$

$$\nabla^2\phi = \nabla \cdot \nabla\phi = \frac{\delta^2\phi}{\delta x^2} + \frac{\delta^2\phi}{\delta y^2} + \frac{\delta^2\phi}{\delta z^2} \tag{18c}$$

For a vector field Φ the Laplacian operator is defined as in Equation 19. Which essential boils down to taking the Laplacian operator of each function and multiply it by the basis vector.

$$\nabla^2\Phi = (\nabla^2\Phi_1)i + (\nabla^2\Phi_2)j + (\nabla^2\Phi_3)k \tag{19}$$

A.3 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 [7]. 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 20. 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.

$$z = (1 - \lambda)x + \lambda y \tag{20}$$

References

[1] 155.42.27.xxx (Anonymous Internet User). Albedo. https://en.wikipedia.org/wiki/Albedo, Jun 2020.

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

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

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

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

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

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

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

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

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

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

[12] Simon Clark.

[13] 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.

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

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

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

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

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

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

[20] 192.2.69.128 (Anonymous Internet User). Gas constant. https://en.wikipedia.org/wiki/Gas_constant#Specific_gas_constant, Jun 2020.

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

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.