Contents

Global Warming Models 1
Time Stepping Naked Planet Model 4
  Week 1: Quiz 8
  Code Tricks: Heat Capacity, Time Steps, and Equilibration Time 9
Iterative Runaway Ice-Albedo Feedback Model 11
  Week 2 Quiz 19
  Code Trick: Hysteresis Into and Out Of the Snowball 20
Ice Sheet Dynamics 22
  Week 3 Quiz 29
  Code Tricks: Time Steps, Snowfall, and Elevation 30
2D Gridded Shallow Water Model 33
  Week 4 Quiz 45
  Code Trick: Geostrophic Flow and a Drifting Rossby Wave 46
  Code Trick: Geostrophic Flow and a Drifting Rossby Wave 46
  Code Trick: Gyre Circulation with Westward Intensification 49
Near Future Climate Model (Model of Climate Change Today!) 52
  Week 5 Quiz 59
  Code Trick: Aerosol Masking and Our Future 60

Global Warming Models

Monday, February 6, 2023
12:54 AM

This class provides a series of Python programming exercises intended to explore the use of numerical modeling in the Earth system and climate sciences. The scientific background for these models is presented in a companion class, Global Warming I: The Science and Modeling of Climate Change. This class assumes that you are new to Python programming (and this is indeed a great way to learn Python!), but that you will be able to pick up an elementary knowledge of Python syntax from another class or from on-line tutorials.

- 

Taught by:David Archer, Professor

Geophysical Sciences

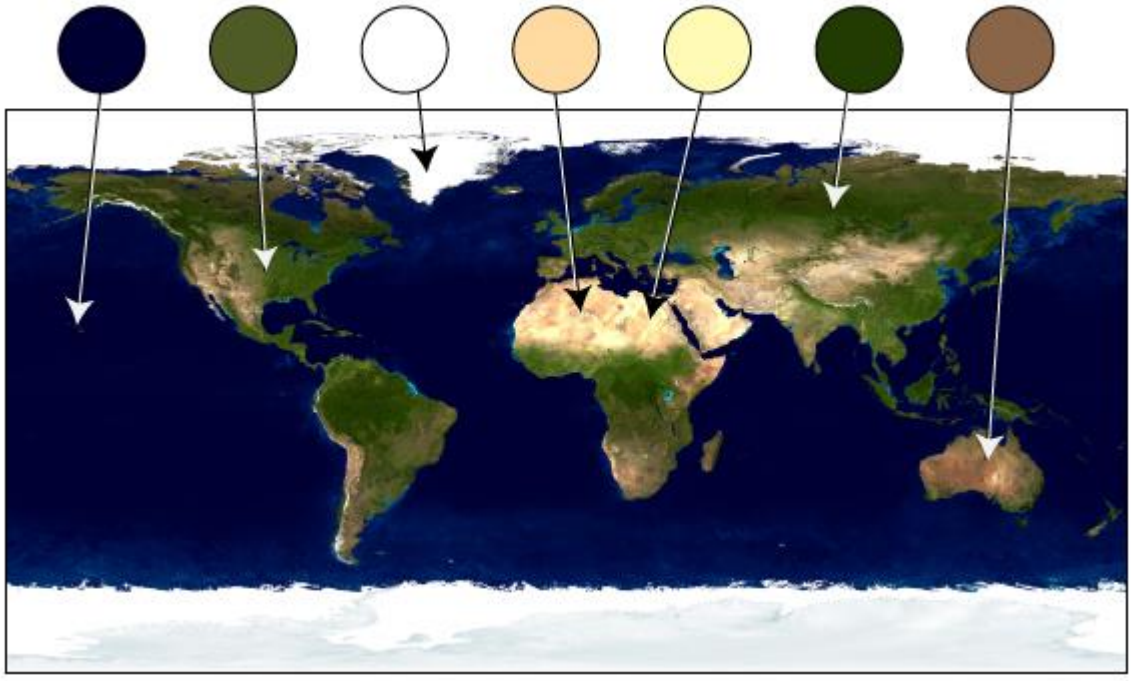
University of Chicago

From <https://www.coursera.org/learn/global-warming-model/home/info>

This class is intended to complement a Coursera class called Global Warming I: The Science and Modeling of Climate Change, which presents much of the background to the material here. In this class you'll be using spreadsheets (maybe) and Python (definitely) to do some simple numerical calculations on topics in Earth System Science. The model you'll be working on this week is based on material from Unit 3 of that class, called First Climate Model.

From <https://www.coursera.org/learn/global-warming-model/home/week/1>

The surface of the Earth is a patchwork of many colors. Some of the colors are dark, such as the blue of the ocean, brown soil, and green forests. Other colors are pale, such as yellow desert sands and white ice.



A sampling of Earth's colors.
Credit: UCAR SciEd with NASA image

When sunlight hits pale colored surfaces, much of it is reflected, bouncing back out to space. When sunlight hits dark colored surfaces, very little of it is reflected. Most of it is absorbed.

The amount of energy reflected by a surface is called **albedo**. Dark colors have an albedo close to zero, meaning little or no energy is reflected. Pale colors have an albedo close to 100%, meaning nearly all the energy is reflected.

Forests, for example, have an albedo of about 15%, which means that 15% of the sunlight that hit a forest is reflected out to space. Fresh snow, on the other hand, can have an albedo of 90%, which means that 90% of the sunlight that hits a snow-capped peak is reflected out to space.

The amount reflected back out to space is called the **planetary albedo**. It's calculated by averaging the albedo of all Earth surfaces – including the land, ocean, and ice. Above the Earth surface, clouds reflect large amount of sunlight out to space too. Earth's planetary albedo is about 31% meaning that about a third of the solar energy that gets to Earth is reflected out to space.

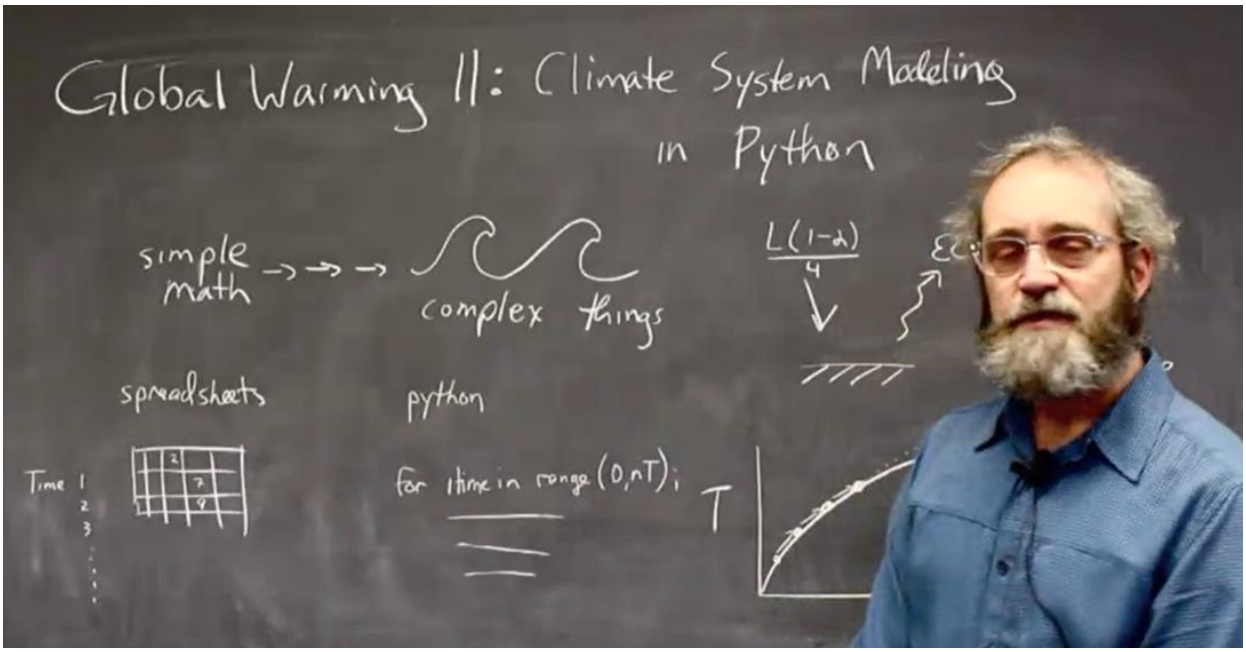
Why do we care what happens to sunlight that gets to Earth? Understanding how much energy from the Sun is reflected back out to space and how much is absorbed becoming heat is important for understanding climate.

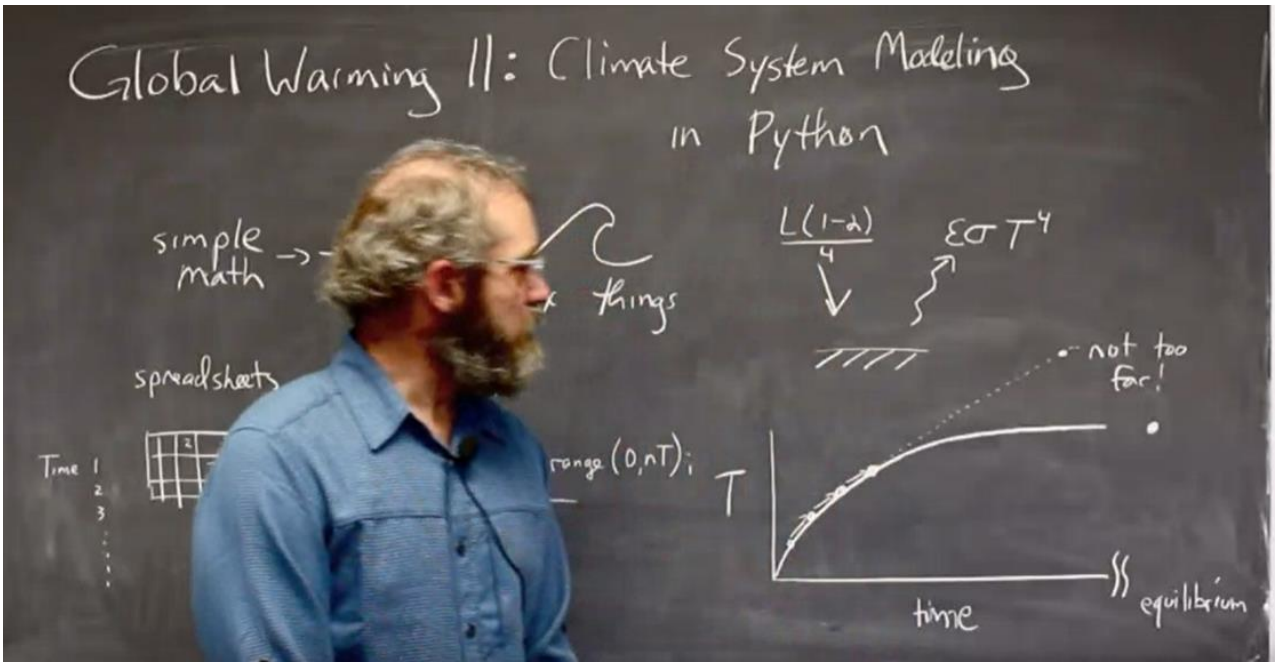
If Earth's climate is colder and there is more snow and ice on the planet, albedo increases, more sunlight is reflected out to space, and the climate gets even cooler. But, when warming causes snow and ice to melt, darker colored surfaces are exposed, albedo decreases, less solar energy is reflected out to space, and the planet warms even more. This is known as the **ice-albedo feedback**.

From <https://scied.ucar.edu/learning-zone/how-climate-works/albedo-and-climate>

Time Stepping Naked Planet Model

Monday, February 6, 2023
12:56 AM





Model Formulation

The background for this material can be found in Sections 2 and 3 of Part I of this class. Short version: Joules are units of energy; Watts are energy flow (J/s). The temperature of a planet is determined by balancing energy fluxes into and out of a planet. Incoming solar heat is determined by L * (1-albedo) / 4, and outgoing infrared is calculated as epsilon * sigma * T^4.

The goal is to numerically simulate how the planetary temperature of a naked planet would change through time as it approaches equilibrium (the state at which it stops changing, which we calculated before). The planet starts with some initial temperature. The “heat capacity” (units of Joules / m2 K) of the planet is set by a layer of water which absorbs heat and changes its temperature. If the layer is very thick, it takes a lot more heat (Joules) to change the temperature. The differential equation you are going to solve is

```
dHeatContent/dt = L*(1-alpha)/4 - epsilon * sigma * T^4
```

where the heat content is related to the temperature by the heat capacity

```
T[K] = HeatContent [J/m2] / HeatCapacity [J/m2 K]
```

The numerical method is to take time steps, extrapolating the heat content from one step to the next using the incoming and outgoing heat fluxes, same as you would balance a bank account by adding all the income and subtracting all the expenditures over some time interval like a month. The heat content "jumps" from the value at the beginning of the time step, to the value at the end, by following the equation

```
HeatContent(t+1) = HeatContent(t) + dHeatContent/dT * TimeStep
```

This scheme only works if the time step is short enough that nothing too huge happens through the course of the time step.

Set the model up with the following constants:

```
timeStep = 100          # years
waterDepth = 4000       # meters
L = 1350                # Watts/m2
albedo = 0.3
```

```
epsilon = 1
sigma = 5.67E-8          # W/m2 K4
```

From <https://www.coursera.org/learn/global-warming-model/supplement/m29aQ/model-formulation>

How to Solve Using a Spreadsheet

1. Construct a time column, with the first time point being 0 and the rest of them going up by an increment, the time step, say 5 years. Make the time step a labeled number in a cell so you can easily change it later. The second time step value = the first value + the time step, and the third relative to the second and so on. If the amount of time per time step is in cell A1 for example, the formula to calculate step 2 from step 1 should point to A1 in an absolute way, using A1, so when you copy the formula downwards it keeps pointing to A1 (as opposed to the previous time step pointer, without dollar signs, which shifts when you copy the cell downward. This saves retyping in formulas all the way down.

2. Construct a column for the planetary temperature at each time step. Set the initial temperature (at time 0) to be some "initial condition" value (0 works).

3. The heat capacity of the surface planet depends on the water thickness. Using the fact that 1 gram of water warms by 1 K with a calorie of heat, and the density of water, to calculate how many Joules of energy it takes to raise the temperature of 1 m^2 of the surface, given some water depth in meters. Make the water depth a separate cell so you dial it up and down later.

4. Create a column for the heat content of the surface layer at each time step. For the initial time, calculate the heat content from the temperature.

5. Create a column for the incoming and outgoing heat fluxes at each time step. Base the incoming heat flux on the parameters solar constant (L), and albedo. The outgoing heat flux is a function of the temperature according to the Stefan-Boltzmann equation.

6. Create yet another column where you combine the incoming and outgoing heat fluxes, and convert them from W/m^2 to J/m^2 timestep.

7. Create a formula for the heat content at the second time point (the row below the first one), equaling the last heat content plus what came in and out in the previous timestep.

8. Calculate the temperature at the second time step, from the heat content.

9. Copy the formulas down to fill in the rest of the values at the second time step.

10. Copy the second time step values down by 10 or 20 rows to create multiple time steps.

11. Plot temperature versus time.

12. Adjust the time step so that the temperature evolves smoothly (no numerical explosions) to an equilibrium value.

From <https://www.coursera.org/learn/global-warming-model/supplement/hmoCI/how-to-solve-using-a-spreadsheet>

How to Encode into Python or Fortran

Practice with Lab Sandbox

Python

1. You'll need to have numpy and matplotlib modules installed in your python programming environment, and in the first lines of your script import both of them.

```
1
2
import numpy
import matplotlib.pyplot
```

2. Define variables for the time step, the water depth, the heat capacity of the surface (units of J/m2 K), the solar constant, albedo, epsilon, and sigma. Since the energy influx doesn't change through the simulation, calculate what that is (units W/m2).

3. At the end of the simulation, in the code you submit for review, you will use

```
1
2
matplotlib.pyplot.plot( time_list, temperature_list)
matplotlib.pyplot.show()
```

to create a plot of the temperature versus time. Create an array for time_list, initially [0] (a 'list' or array with one value in it, the number 0), and similarly for temperature_list. Set the initial temperature (maybe to 0 K, or whatever you like), and calculate the initial heat content (units of J/m2).

4. Create a loop for time steps. Each step, calculate the outgoing heat flux, and the new heat content after taking up and giving off heat. Append the value of the time, and the temperature, from each step, at the end of the time and temperature lists.

Python tricks you may find useful:

a. arrayname.append(value) to add another item to the end of a list

b. pow(T[-1],4) would raise the last temperature in the T list to the power of 4

My script for this model is 23 lines long.

From <https://www.coursera.org/learn/global-warming-model/supplement/Xq8qc/how-to-encode-into-python-or-fortran>

```
import numpy
import matplotlib.pyplot as plt

timeStep = 100 # years
waterDepth = 4000 # meters
L = 1350 # Watts/m2
albedo = 0.3
epsilon = 1
sigma = 5.67E-8 # W/m2 K4

heatCapacity = waterDepth * 4.2E6 # J/K m2
timeYears = [0]
TK = [0.]
heatContent = heatCapacity * TK[0]
heatIn = L * (1 - albedo)/4
heatOut = 0
for itime in range(0, 100):
    timeYears.append ( timeStep + timeYears[-1] )
    heatOut = epsilon * sigma * pow(TK[-1],4)
    print(timeYears[-1], heatOut)
    heatContent = heatContent + (heatIn - heatOut) * timeStep * 3.14e7
    TK.append (heatContent / heatCapacity)
#    print(itime, timeYears[-1])
#print(TK[-1])
plt.plot(timeYears, TK)
plt.show()
```

Week 1: Quiz

Monday, February 6, 2023
5:32 PM

- Lekhraj Sharma

- Global Warming II: Create Your Own Models in Python
- Week 1
- Code Tricks: Heat Capacity, Time Steps, and Equilibration Time

Previous Next

- **Resources**
- **Time-Stepping Naked Planet Model**
- **Video:** VideoHow the Model Works
  . Duration: 3 minutes3 min
- **Reading:** ReadingModel Formulation
  . Duration: 10 minutes10 min

- **Reading:** ReadingHow to Solve Using a Spreadsheet
  . Duration: 10 minutes10 min
- **Reading:** ReadingHow to Encode into Python or Fortran
  . Duration: 10 minutes10 min
- **Programming Assignment:** Code Check
  . Duration: 3 hours3h
- **Peer-graded Assignment:** Code Review
  Grading in progress

  Review Your Peers: Code Review
- **Quiz:** Code Tricks: Heat Capacity, Time Steps, and Equilibration Time
  4 questions

Code Tricks: Heat Capacity, Time Steps, and Equilibration Time

Quiz30 minutes • 30 min

Submit your assignment

Due February 19, 11:59 PM ISTFeb 19, 11:59 PM IST
Attempts 3 every 8 hours
Start assignment

Receive grade

To Pass 80% or higher

Your grade

-Not available
Like
Dislike
Report an issue

Back

Code Tricks: Heat Capacity, Time Steps, and Equilibration Time

Graded Quiz. • 30 min. • 4 total points available.4 total points
DueFeb 19, 11:59 PM IST

1.

Question 1
Adapt the code to simulate a dry planet, like Mars. The effective heat capacity of a solid surface is much smaller than that of an ocean, because heat diffuses very slowly in solids compared to fluid mixing in liquids. On time scales of a few years, heat penetrates a meter or a few meters into a soil column. Approximate a solid surface by changing the depth of the water in your model to 1 meter.

The trick here is that you will have to take smaller time steps to keep the numerical method from blowing up. A good way to see what's going on is to decrease the ocean depth in stages. Start with 2000 meters, then 1000, then 100, then 10, then 1. Each time you'll have to make the time step shorter, and if it's too long you'll see the numerics to crazy.

What is the longest time step you can take for model stability, using a water depth of 1 m?

Answer: 0.04 Years

1 point

2.

Question 2

Change the initial temperature to 400 Kelvins. What is the value of the outgoing emission flux at time 0, within a tolerance of 1 Watt/m2?

Answer: 1451.52

From <http://localhost:8888/notebooks/Downloads/Modeling/modeling.ipynb>

1 point

3.

Question 3

Set the ocean depth to 4000 meters and the time step to 10 years. Keep he initial temperature 400 K from the last problem. What is the outgoing heat flux after 100 years?

Answer: 444.1265170428673

From <http://localhost:8888/notebooks/Downloads/Modeling/modeling.ipynb>

1 point

4.

Question 4

What is the outgoing heat flux, in Watts/m2, after 2000 years? (Your model should be in equilibrium at this point).

Answer: 236.25014150815934

From <http://localhost:8888/notebooks/Downloads/Modeling/modeling.ipynb>

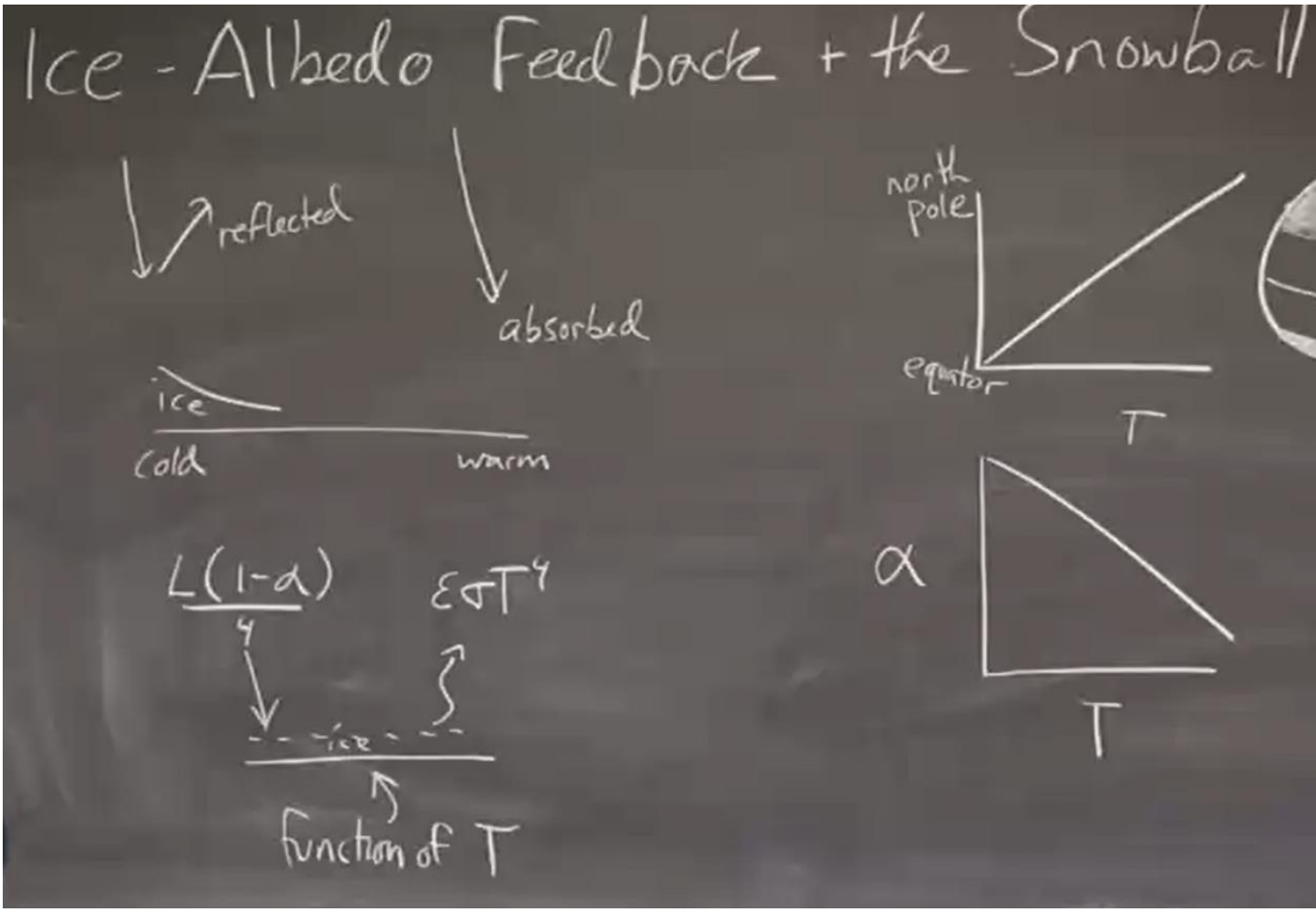
From <https://www.coursera.org/learn/global-warming-model/exam/t8bz4/code-tricks-heat-capacity-time-steps-and-equilibration-time/attempt>

Iterative Runaway Ice-Albedo Feedback Model

Monday, February 6, 2023
8:21 PM

The ideas behind this model were explained in Unit 7, Feedbacks, in Part I of this class. First we get to generate simple linear "parameterization" functions of planetary albedo and the latitude to which ice forms (colder = lower latitude ice). Second, for any given value of the solar constant, L, we'll use iteration to find consistent values of albedo and T, to show the effect of the ice albedo feedback on Earth's temperature, running away to fall into the dreaded "snowball Earth".

From <https://www.coursera.org/learn/global-warming-model/home/week/2>


Ice - Albedo Feedback + the Snowball
reflected
absorbed
ice
cold
warm
north pole
equator
T
α
T
L(1-α)
εσT⁴
ice
function of T
Snowball.py

```
import numpy
import matplotlib.pyplot as plt

niters = 100
albedoM = -1e-2
albedoB = 2.8
iceLatM = 1.5
iceLatB = -323
sigma = 5.67E-8

x = []
y = []

plotType = "iterDown" # "L", "iterUp", "iterDown"
LRange = [1200, 1500]

L = LRange[0]
albedo = 0.65

while L < LRange[1]+1:
    for iter in range(0,niters):
        T = L * (1 - albedo) / (4 * sigma)
        T = pow(T, 0.25)
        albedo = T * albedoM + albedoB
        albedo = min( albedo, 0.65 )
        albedo = max( albedo, 0.15 )
        latice = T * iceLatM + iceLatB
        latice = min( latice, 90 )
        latice = max( latice, 0 )
        if plotType is "iter" or plotType is "iterUp":
            x.append(iter)
            y.append(T)
    if plotType is "iter" or plotType is "iterUp":
        x.append(numpy.nan)
        y.append(numpy.nan)
    if plotType is "L":
        x.append(L)
        y.append(T)
    L = L + 10

while L > LRange[0]-1:
    for iter in range(0,niters):
        T = L * (1 - albedo) / (4 * sigma)
        T = pow(T, 0.25)
        albedo = T * albedoM + albedoB
        albedo = min( albedo, 0.65 )
        albedo = max( albedo, 0.15 )
        latice = T * iceLatM + iceLatB
        latice = min( latice, 90 )
        latice = max( latice, 0 )
--:---  snowball.py    Top L14    (Python)----------------------
```

The second model you're going to be working on is an iterative model
of the Ice-Albedo Feedback and how it affects the temperature of the Earth.
So instead of stepping through the time in this model,
we're going to be making successive guesses of what the right answers are and
the guesses will converge and get closer.
You get the same answer every time you guess as you get towards the end, so
it's a fundamentally different kind of calculation.

Play video starting at ::30 and follow transcript0:30

So the idea is that if planet is cold, it will have ice and
snow, which is very reflective, and so that will reflect incoming energy.
Whereas if the planet is warmer there's none of that stuff, and so
the sunlight is absorbed more effectively.
So if this is the energy balance of the planet,
here is the incoming solar [COUGH], and

here is the outgoing infrared which is a function of the temperature of the planet.
Play video starting at :1: and follow transcript1:00
The Albedo here kind of comes off the top of the incoming solar,
gets reflected away, rather than absorbed and
this is a function of ice, which is a function of temperature.
Play video starting at :1:12 and follow transcript1:12
So, the calculation gives you a linear function
of temperature to describe the latitude,
that ice will form on a planet, so the colder it is,
the more the ice can be found closer and closer to the equator.
And another equation is given with the Albedo as a function of temperature,
and again, you're going to fit a straight line to the data that you're given.
And so the idea is to start with a guess for the Albedo,
and then using that Albedo, calculate the temperature.
Play video starting at :1:55 and follow transcript1:55
And then from this temperature get another guess for the Albedo, and
then go back for another temperature, and then back and forth, back and forth,
iterating between Albedo and temperature.
This is what the snowball code looks like in Python,
it has one loop going up and the other loop going down
where we're looping over different values of the solar constant.
So, for going
down we start with a high solar constant,
and so, the temperature is warm and the Albedo is probably ice free and
that's true for many of these different values of L.
But then, at some point you cross a value of L where it starts to freeze a little
bit and so then it takes a few iterations, this is the number of iterations here for
the temperature and Albedo to stop changing.
And then at some point there's a critical solar constant value at which
all of a sudden the feedbacks run all the way to the equator, and the temperatures
Play video starting at :3:10 and follow transcript3:10
go all the way down, the Earth gets covered with ice like a snowball.
Play video starting at :3:17 and follow transcript3:17
So if i change the plot type here and
run that again, we can see.
We are now looking at the equilibrium temperature after the iterations as
a function of the solar constant in this dimension.
And the reason why this isn't a simple line is because there
is hysteresis in the system, if you start up here from an ice-free state and
you cool the sun down in this dimension, the temperature,
of course, gets colder but you don't start to get any
ice until you reach an insulation that's sort of here,
solar sunlight rate insulation.
And then at some point you drop down into the snowball make it colder,
it's just ice frozen all the way anyways, so just the sun effect here.
But then as you warm it back up, the ice sheet
is able to perpetuate its own existence by reflecting sunlight back to space.
And so to get out of the snowball, it takes considerably more work,
than it took to get- than it took sort of coolness to get into it,
this is what we call hysteresis, or path dependence.

From <https://www.coursera.org/learn/global-warming-model/lecture/rvNth/how-the-model-works>

Parameterized Relationship Between T, Ice Latitude, and Albedo

The first part of this calculation is a pair of regressions to calculate the slope and intercept for two linear functions: one is the latitude to which there will be snow and ice on the surface (lower latitudes, closer to the equator, when it's colder), and the second is the overall albedo from ice that you will get at that temperature. You are looking for functions Ice latitude = $m_1 * T + b_1$ and albedo = $m_2 * T + b_2$. The regression is probably easier to do in a spreadsheet, either using a built-in least-squares function, or by setting up a trial function with values of m and b in cells that you tweak by hand, and compare the resulting line in a plot with the plotted values of the data.

Use the following table to derive functions to describe the ice latitude and the albedo from the ice as a function of the mean planetary top-of-atmosphere (radiating) temperature.Assume that the functions are linear (which they pretty much are in this table).

Mean Planetary Temperature	Ice Latitude	Planetary Albedo
265	75	0.15
255	60	0.25
245	45	0.35
235	30	0.45
225	15	0.55
215	0	0.65

The functions should be in slope / intercept form, as in ice_latitude = m1 * T + b1 and albedo = m2 * T + b2.

Solution: Since it is linear equation, we can solve it easily given above data.

m1 = dL/dT = 15/10 = 1.5 , m2 = dA/dT = -0.10/10 = -0.01

b1 = L - m1 * T = 60 - 1.5 * 255 = -322.5 , b2 = A - m2 * T = 0.25 - -0.01 * 255 = 2.8

From <https://www.coursera.org/learn/global-warming-model/supplement/fqAsP/parameterized-relationship-between-t-ice-latitude-and-albedo>

Spreadsheet Instructions

It is possible for a spreadsheet to do all the calculations that the Python version of this exercise will do, but it's starting to get cumbersome. You will want to do about 40 values of L, and each value of L will require 100 iterations or so before we can be sure that the process has converged. In a spreadsheet, typically, all of the iterations would be saved in separate cells. (It is also possible to use built-in iteration functions in some spreadsheets, which might avoid this problem). A sensible solution might be to restrict the spreadsheet to working on a single value of L, which will be useful for checking and debugging your Python script, and then leave the power-crunching of lots of different values of L to the script.

Set a value of L as a labelled number in a cell (with units!). You'll also need an initial guess for the albedo, and maybe values for epsilon and sigma would be useful. Also make cells with the slope and intercept values for your parameterized fits to the albedo and ice-latitude functions of temperature you derived in the first part.

Each iteration will take a separate line in the sheet. The left-most column should contain an iteration number. Start with 1 in the top cell and then write a formula adding 1 to that, in the cell below it. Then copy that formula down until you reach 100. (This can be done in one big copy operation).

The next column will be temperature guesses, and the third albedos.

Solve for the first temperature guess assuming the initial guess for the albedo. Then use that temperature to calculate a new albedo, the second guess, below the first. The albedo depends on the temperature in your parameterization, but don’t allow your parameterization to extrapolate to values outside the range in the data, that is, higher than a value of 0.7, or lower than a value of 0.15. You can use the MIN and MAX functions of the spreadsheet to limit the values to the reasonable range. Then, from that albedo second guess, calculate a second guess for the temperature. Construct these formulas so that they can be copied down to fill out the rest of the iterations.

Make a plot of temperature versus iteration number to see how many iterations it takes before it converges. A value of L to try is 1250 W/m2, with an initial guess for the albedo of 0.15.

From <https://www.coursera.org/learn/global-warming-model/supplement/G6W8j/spreadsheet-instructions>

Coding Instructions

Practice with Lab Sandbox

Python

As in the last problem, begin by importing library packages that we'll need, for messing with lists (arrays), and for plotting.

```
1
2
```

```
import numpy
import matplotlib.pyplot
```

Define and initialize variables for the number of iterations, the M and B values from the albedo and ice line regressions from Part I, and for epsilon and sigma. Also set up variables for the range of L values over which you are going to do the calculation

```
LRange = [ 1200, 1600 ]
```

Each pass over the range in L values requires two nested loops, the outer one over values of L, and the inner one for the iterations. Start with a cooling sweep from the highest L value

```
L = LRange[1]
albedo = 0.15
while L > LRange[0]-1:
    blah blah blah
    L = L - 10
```

where you'll want to replace the pseudo-code ("blah blah blah") with another, inner, loop, in which you iterate, finding new values for albedo, then T, then albedo again, until the T and albedo values you wind up with are consistent with each other (it "converges"). Before you begin any of the iterations, set the albedo to 0.15, and for each iteration (for each L), set the initial albedo for your iteration equal to the value you got for the converged albedo from the last iteration. Then go back and forth, calculating T from that initial albedo, then albedo from that T (limited to the range 0.15 to 0.65), back and forth each time. In contrast to the spreadsheet, there is not much penalty for taking lots of iterations; it doesn't get ungainly but just runs slower.

Copy this loop-within-a-loop into a second pass in your code, then modify it to start from the lowest value of L (LRange[0]), and work back up to LRange[1]. Use the same strategy for albedo: for the initial guess, use the final value from the last set of iterations (the last L value).

For plotting, build three options. It's easiest to just have one plot at a time, so define a variable plotType which contains a string telling what kind of plot you want to see.

As the code runs, it should fill up two lists, which we can call x and y. What you put into x and y as the run progresses depends on the value of the variable plotType.

Initialize the lists with

```
x_list = []              # an empty list
y_list = []
```

then each time you want to add something to the list, append the values to the end of the list, as, for example,

```
x_list.append( L )
```
The three options for plots types I'd recommend are:

Option 1: the temperature after the iterations are done for each value of L, plotted as a function of L, for the entire series of calculations as L first sweeps down and then back up again. This plot will show the hysteresis in the ice albedo feedback, how there are multiple steady states for some intermediate values of L (including our own!).

Option 2: the temperature each iteration, for each value of L, plotted as a function of the iteration number, for the first sweep over L values, when each L is lower than the last. The trick here is to insert values of numpy.nan (stands for Not A Number) into the x and y arrays at the end of each iteration loop, when the number of iterations resets from the maximum number (for the last L value) to 0 (for the next L value). When matplotlib makes a plot of these lists, we don't want it to draw a line connecting these. It makes the plot messy and doesn't mean anything. So the nan value tells the plot library not to draw that line.

Option 3: same as option 2 but for the second sweep through the range in L, when the sun is getting hotter as you go through.

There are other interesting things you might want to see, such as the albedo or the ice latitude in any of these plot types. To see the ice latitude as a function of L, change the y_list.append( ) statement to append the ice latitude rather than the temperature.

Then, when all the calculations are done,

```
matplotlib.pyplot.plot( x, y )
matplotlib.pyplot.show( )
```
will create a plot of your results.

My script required about 60 lines of python code. New python functions you may find useful include min() and max().

Fortran

Again the logic of coding in Fortran is the same as for Python, and again there is no plotting capability in Fortran, so substitute formatted write statements to create output tables that are easy to read.

From <https://www.coursera.org/learn/global-warming-model/supplement/QSiHE/coding-instructions>

Week 2 Quiz

Wednesday, February 8, 2023
8:33 PM

- **Iterative Relaxation to Consistent T and Albedo Given L**
- **Video:** VideoHow the Model Works
  . Duration: 4 minutes4 min
- **Reading:** ReadingParameterized Relationship Between T, Ice Latitude, and Albedo
  . Duration: 10 minutes10 min
- **Reading:** ReadingSpreadsheet Instructions
  . Duration: 10 minutes10 min
- **Reading:** ReadingCoding Instructions
  . Duration: 10 minutes10 min
- **Programming Assignment:** Code Check
  . Duration: 3 hours3h
- **Peer-graded Assignment:** Code Review
  Grading in progress
  Review Your Peers: Code Review
- **Quiz:** Code Trick: Hysteresis Into and Out Of the Snowball
  2 questions

Code Trick: Hysteresis Into and Out Of the Snowball

Quiz30 minutes • 30 min

Submit your assignment

Due February 26, 11:59 PM ISTFeb 26, 11:59 PM IST
Attempts 3 every 8 hours
Resume assignment

Receive grade

To Pass 45% or higher

Your grade

-Not available

Like
Dislike
Report an issue

Back

Code Trick: Hysteresis Into and Out Of the Snowball

Graded Quiz. • 30 min. • 2 total points available.2 total points
DueFeb 26, 11:59 PM IST

1.

Question 1
On the downward sweep through L (getting colder each time), What is the maximum value of the solar constant, L, in units W/m2, for which the planet freezes all the way to the equator? (Make sure you have enough iterations to see that your temperature and albedo have reached equilibrium). The tolerance in the grader is a range of 20 W/m2.

Answer: 1250

1 point

2.

Question 2
Now on the upward sweep through the range in L, at some critical L value (in Watts/m2), the temperature jumps up to a higher value. What value of L do you get for this sudden transition, as the snowball melts? The tolerance in the grader is a range of 50 W/m2.

Answer: 1390

1 point

Coursera Honor Code [Learn more]

I, **Lekhraj Sharma**, understand that submitting work that isn't my own may result in permanent failure of this course or deactivation of my Coursera account.

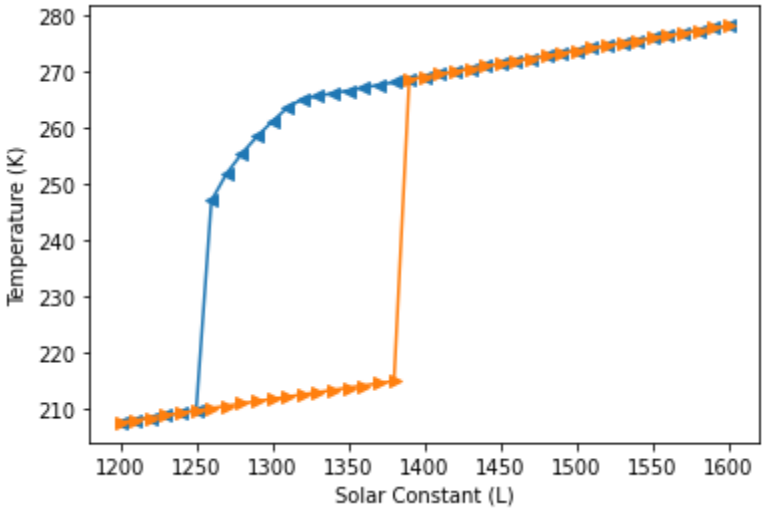
Submit **Save draft**

Like
Dislike
Report an issue

From <https://www.coursera.org/learn/global-warming-model/exam/v4qnf/code-trick-hysteresis-into-and-out-of-the-snowball/attempt>

0.65 [(1600, 278.2748546226214), (1590, 277.83902735790974), (1580, 277.40113942929844), (1570, 276.9611679584168), (1560, 276.5190896651926), (1550, 276.0748808581722), (1540, 275.62851742454353), (1530, 275.17997481985054), (1520, 274.7292280573872), (1510, 274.27625169725985), (1500, 273.82101983510404), (1490,

273.3635060904435), (1480, 272.90368359467686), (1470, 272.44152497867816), (1460, 271.9770023599951), (1450, 271.5100873296303), (1440, 271.04075093838804), (1430, 270.5689636827694), (1420, 270.09469549039784), (1410, 269.6179157049561), (1400, 269.13859307061375), (1390, 268.6566957159255), (1380, 268.1721911371772), (1370, 267.6850461811579), (1360, 267.1952270273318), (1350, 266.70269916938685), (1340, 266.2074273961316), (1330, 265.70937577171327), (1320, 265.20850761512713), (1310, 263.63789345764667), (1300, 261.1720688389635), (1290, 258.4782091060152), (1280, 255.452427682946), (1270, 251.87298645138333), (1260, 247.02724348151042), (1250, 209.57223828048924), (1240, 209.1518304700069), (1230, 208.72887212680502), (1220, 208.3033268511552), (1210, 207.87515742029214), (1200, 207.4443257628261)]
0.15 [(1200, 207.4443257628261), (1210, 207.87515742029214), (1220, 208.3033268511552), (1230, 208.72887212680502), (1240, 209.1518304700069), (1250, 209.57223828048924), (1260, 209.9901311595615), (1270, 210.405543933807), (1280, 210.81851067789196), (1290, 211.2290647365307), (1300, 211.63723874564425), (1310, 212.04306465274823), (1320, 212.44657373660326), (1330, 212.84779662616046), (1340, 213.24676331883225), (1350, 213.64350319811703), (1360, 214.0380450506057), (1370, 214.4304170823956), (1380, 214.820646934937), (1390, 268.6566957159255), (1400, 269.13859307061375), (1410, 269.6179157049561), (1420, 270.09469549039784), (1430, 270.5689636827694), (1440, 271.04075093838804), (1450, 271.5100873296303), (1460, 271.9770023599951), (1470, 272.44152497867816), (1480, 272.90368359467686), (1490, 273.3635060904435), (1500, 273.82101983510404), (1510, 274.27625169725985), (1520, 274.7292280573872), (1530, 275.17997481985054), (1540, 275.62851742454353), (1550, 276.0748808581722), (1560, 276.5190896651926), (1570, 276.9611679584168), (1580, 277.40113942929844), (1590, 277.83902735790974), (1600, 278.2748546226214)]



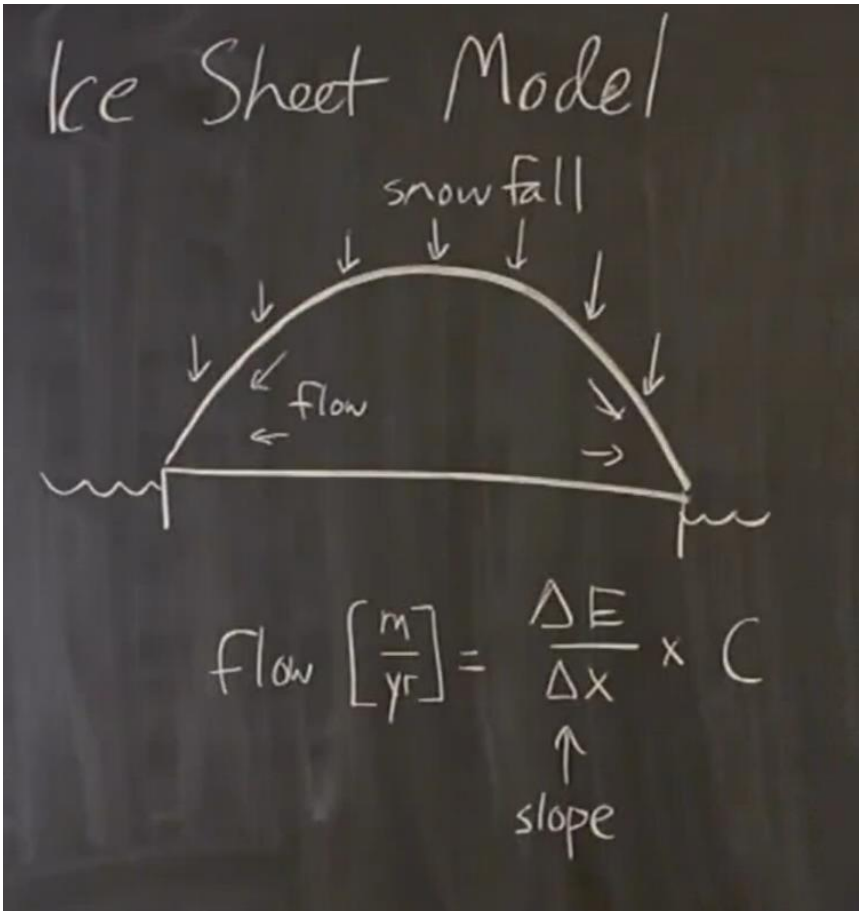
From <http://localhost:8888/notebooks/Downloads/Modeling/modeling.ipynb>

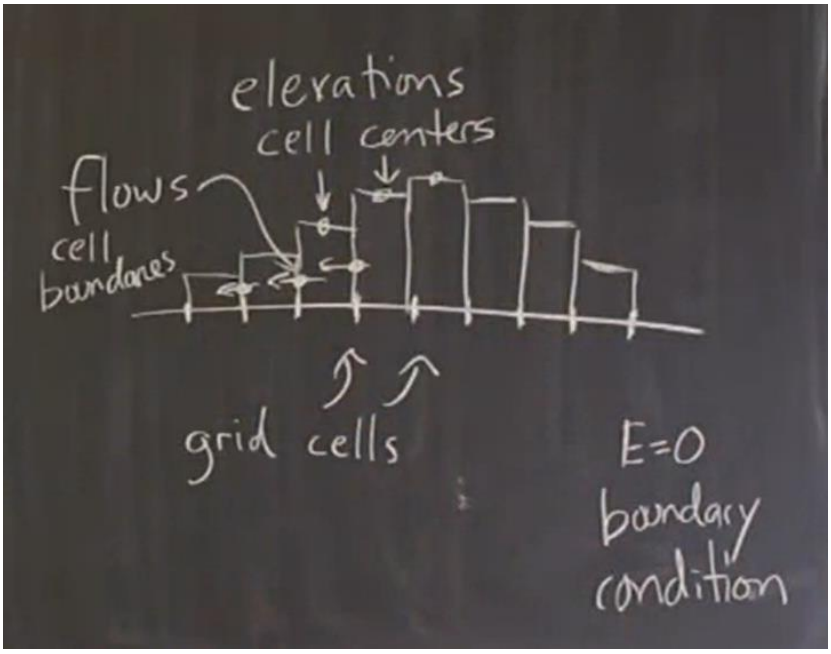
Ice Sheet Dynamics

Wednesday, February 8, 2023
6:03 PM

Ice flows like extra-thick molasses, downhill. The shape of the ice sheet (altitude versus distance across) is determined by the relationship between ice surface slope and the flow rate of the ice.

From <https://www.coursera.org/learn/global-warming-model/home/week/3>




elevations
cell centers
flows
cell boundaries
grid cells
E=0
boundary
condition

```
nX = 10
domainWidth = 1e6  # meters
dx = domainWidth / nX
#dz = 500
timeStep = 100  # years
nYears = 20000
nSteps = int( nYears / timeStep)
flowParam = 1e4 # m horizontal / yr
snowFall = 0.5   # m / y
plotLimit = 4000

elevations = numpy.zeros(nX+2)
flows = numpy.zeros(nX+1)

fig, ax = plt.subplots()
ax.plot(elevations)
ax.set_ylim([0,plotLimit])
plt.show(block=False)

# nX = 2
#
# e0=0 |  e1  |  e2  |  e3=0
#     f0     f1     f2
#

for iTime in range(0, nSteps):

    for ix in range(0, nX+1):
        flows[ix] = ( elevations[ix] - elevations[ix+1] ) / dx * flowParam * \
            ( elevations[ix] + elevations[ix+1] ) / 2 / dx

    for ix in range(1,nX+1):
        elevations[ix] = elevations[ix] + \
            ( snowFall + flows[ix-1] - flows[ix] ) * timeStep

    print( "year", iTime*timeStep )
    ax.clear()
    ax.plot( elevations )
    ax.set_ylim([0,plotLimit])
    plt.show(block=False)
    fig.canvas.draw()

ax.clear()
ax.plot( elevations )
ax.set_ylim([0,plotLimit])
plt.show()
fig.canvas.draw()
```

The Ice Sheet Model is formulated in one dimension,
a horizontal dimension and so we have snow that's
accumulating all the way across this dimension.
And as the ice gets thicker and thicker you get a slope at the ice
surface which drives a pressure gradient which drives flow in both directions.
The flow is always going sort of down hill.
The flow is given by the surface slope,
the change in elevation over the change in x of the slope
of the surface there, times some constant, which will be given to you.
So we're going to solve this problem numerically by breaking it up into pieces,
and each of those pieces, we're going to call a grid cell.
And so there will be some number of grid cells across the domain, and
each grid cell has an elevation which is defined in the center of the cell and so
Play video starting at :1:11 and follow transcript1:11
the slope in the elevation would be the difference in x divided by the difference
in x so the slope would be sort of between two adjacent points like that.
And then there are flows defined in the domain as well

going between one grid cell and another.
So the flows are actually defined on the cell faces,
whereas the elevations are in the cell centers.
Another crucial part of this problem is the boundary condition.
We have to keep it zero at the edges where the ice is flowing into the water.
We're not going to let it build up higher than that.
And that sort of controls the whole domain of the ice sheet.
And the way that boundary condition manifests itself in
a grid like this is to have some extra grid points, some sort of ghost
Play video starting at :2: and follow transcript2:00
grid cells on both sides,
where you specify that the elevation has to be equal to zero at all times.
So you have to treat those cells special.
>> So this is what the Ice Sheet Model looks like in my spreadsheet version.
I have, as usual, some parameters up at the top govern things, so
I can change stuff on the fly and have it reflected everywhere, including units.
So time step in years, 100 years.
Play video starting at :2:30 and follow transcript2:30
I have the time steps on rows going down here.
So, this is the time in years, and this is going up to 23
thousand years and then we have to have elevations and we have to have flows.
So I have elevations, I have ten of those, and then I have two grid cells at
the edges, and then the flows go between those elevations.
So, can see right
Play video starting at :3:2 and follow transcript3:02
here there's difference in elevation
Play video starting at :3:7 and follow transcript3:07
with x and that means that you get a flow there.
Play video starting at :3:13 and follow transcript3:13
In Python, this is what the code looks like.
Play video starting at :3:18 and follow transcript3:18
It's a short little code, very easy, and this is what it looks like when it runs.
So now we're making the Python make a new plot every single time step and
so you can see how the elevation is changing through time.
Play video starting at :3:38 and follow transcript3:38
We change the number of grid points.
Say we make 20 grid points instead of ten.
Play video starting at :3:50 and follow transcript3:50
We get a smoother curve but we have to be careful about the time step.
Play video starting at :3:55 and follow transcript3:55
Because as you make the grid points closer together, you need to make the time step
Play video starting at :4:1 and follow transcript4:01
shorter, or else something like that happens where it just blows up.
Decrease the time step.
Play video starting at :4:10 and follow transcript4:10
In general, a factor of two change in the size,
factor of two change in the time step should work.
More or less, we'll see.
Play video starting at :4:24 and follow transcript4:24
Taking longer for the action to happen here,
because we're taking shorter time steps.

From <https://www.coursera.org/learn/global-warming-model/lecture/1PhJh/how-the-model-works>

Model Formulation

Ice flows, like any other fluid, only very slowly, due to its high viscosity. The force driving the flow is differences in pressure in the interior of the ice, which arise from differences in the elevation of the ice surface. If you make a pile of ice, it's like a pile of molasses, in that it will flow outward and flatten itself out.

The model is formulated in one dimension, on a horizontal grid. Start with 10 grid cells. Let them span a horizontal distance of 1000 km, or 10^6 meters. Each grid cell will have an elevation of ice. Flow between adjacent cells depends on the difference between their elevations. Snow falls equally on all grid cells. Also, vitally important to this type of problem, is a boundary condition. We'll assume that the ice sheet is confined to a landmass like Greenland or Antarctica, so that the thickness of the ice at the boundaries has to be zero. Ice flows into the ocean and disappears, both in reality and in our model.

In the real world, the ice surface might slope upward in some direction throughout what we are calling a grid point. The elevation we are keeping track of might be the average elevation within that region. For the best approximation to reality from our simple grid, let's specify that the elevation of a cell applies in the center of the cell. In contrast, the flow between the cells can most sensibly be defined on the cell boundaries. This combination is called a "staggered" grid, where all of the variables are not defined in the same places. Including flows from the grid cells on the edges of the ice sheet, there will be 11 flows among the 10 grid cells of the simulation. My suggestion is to draw a cartoon with your grid, showing the index numbers of the two variables. It can get confusing.

The confusion is compounded by the boundary condition (the ice vanishes when it hits the ocean). The easiest way is to define two extra grid cells for the elevations, and keep them always set to a value of 0. This means that you will have nX+1 = 11 values in the list of fluxes (between each of the 10 boxes), and nX+2=12 values in the list for elevations, to contain the "ghost" cells that will always have 0 elevation.

The model will step forward in time, using a time step of 100 years. It will begin to rise uniformly, but the elevations at the edges will be eroded by flow to the ocean. Eventually it will reach a steady state, where snowfall in each grid cell is balanced by flow from the grid cell, which is all determined by the slope of the ice surface.

Parameter Settings

Set the following parameter values

```
1
2
3
4
5
6
```

```
nX = 10                 # number of grid points
```

```
domainWidth = 1e6       # meters
timeStep = 100          # years
nYears = 50000          # years
flowParam = 1e4         # m horizontal / yr
snowFall = 0.5          # m / y
```
Initialize the elevations to zero.

Calculate the flows as

```
flow[ix] = ( elevation[ix] - elevation[ix+1] ) / dX * flowParam  * \
    ( elevation[ix]+elevation[ix+1] ) / 2 / dX
```
where the first incidence of the dX variable is to calculate the gradient in elevation, and the second one corrects for the aspect ratio of the grid cell, how much horizontal flow translates into vertical elevation. Be sure that you have the indexing right for your implementation of the staggered grid, and be sure to get the sign right, so that flow from the first cell in the list to the second would be defined as positive.

Then update the elevations by adding (snowFall + flow[ix-1] - flow[ix]) * timeStep.

Repeat the cycle, calculating flows, then taking time steps in the elevation. Watch the ice sheet grow in, through time, to its equilibrium shape, which will be determined by the math in the equations described above

From <https://www.coursera.org/learn/global-warming-model/supplement/W5kDj/model-formulation>

Spreadsheet Tips

I always find it useful to take some space at the top of a spreadsheet for model parameters, like the time step etc. I use three cells for each parameter, one for the name, one for the units (years, in this case), and a third for the numeric value.

Beginning below the parameter area, make a column of time values, with a step equal to the time step. You'll want to see many thousands of years.

Set up an array of 12 columns alongside the time column, for the elevations. Fill the first and 12th column with zeros. Fill the first row with zeros.

Set up another array of 11 columns to hold the flow rates. Create the formula for the first flow rate (leftmost, topmost) by referencing the contents of two elevation values, and parameters like the flow constant. Use relative addressing to point to elevations, and absolute addressing to point to parameters, so that you can copy the formula you create across and down across the entire block of flow values.

Go back to the elevation array, and write a formula to update the elevations in the first grid cell (leftmost, topmost). The ice in that cell in year 100 will be equal to that in year 0, plus snowfall, plus flow coming in

from the left (which will be positive if it's coming into this cell), minus flow going out to the right. Because the flow is actually going to the left in this first cell, the flow number you'll be using should be negative.

Use relative and absolute addressing as you did with the flow formula so that you can copy the elevation formula to the rest of the elevation block.

Make a plot of the elevation as a function of distance at some time slice, to see the shape of the ice sheet as it evolves toward its steady state.

From <https://www.coursera.org/learn/global-warming-model/supplement/XF0nh/spreadsheet-tips>

Coding

Practice with Lab Sandbox

Python

The advantage of a scripting tool like python for this calculation is that it's no trouble at all to do a longer calculation -- we don't have to save all the intermediate values of all the variables.

Initialize the elevations and flows

```
elevations = numpy.zeros(nX+2)
flows = numpy.zeros(nX+1)
```

Create a time loop which first calculates flows (by looping over the horizontal grid points, using the current values of the elevations), then takes a time step for the elevations (looping over the horizontal grid again).

The plotting infrastructure in Python, matplotlib, can be used to create a cool animation of the evolving ice sheet. Before starting the time loop, set things up using

```
fig, ax = matplotlib.pyplot.plot.subplots()
ax.plot(elevations)
ax.set_ylim([0,plotLimit])
matplotlib.pyplot.plot.show(block=False)
```

where the block=False flag tells Python to throw the plot on the screen then keep moving. Within the time loop, issue the commands

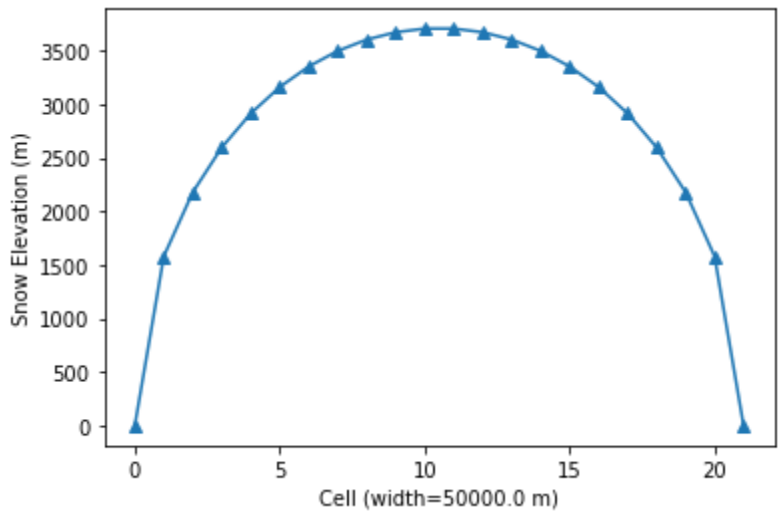
```
    ax.clear()
    ax.plot( elevations )
    matplotlib.plot.show( block=False )
    matplotlib.pyplot.pause(0.001)
    fig.canvas.draw()
```

These commands will update the plot to show the latest elevations, and then keeps moving.

Fortran

Again the logic is similar to that for Python, and again you will need text tables to watch the output (which you will have to disable for the automatic code checking). This problem is getting computationally strenuous enough that you could go to much higher resolution (number of grid points) in Fortran than you could with Python.

From <https://www.coursera.org/learn/global-warming-model/supplement/VPLJj/coding>



From <http://localhost:8888/notebooks/Downloads/Modeling/modeling.ipynb>

Week 3 Quiz

Wednesday, February 8, 2023
9:32 PM

- **A Simple 1-D Ice Sheet Flow Model**
- **Video:** VideoHow the Model Works
  . Duration: 4 minutes4 min
- **Reading:** ReadingModel Formulation
  . Duration: 10 minutes10 min
- **Reading:** ReadingSpreadsheet Tips
  . Duration: 10 minutes10 min
- **Reading:** ReadingCoding
  . Duration: 10 minutes10 min
- **Programming Assignment:** Code Check
  . Duration: 3 hours3h
- **Peer-graded Assignment:** Code Review
  Grading in progress
  Review Your Peers: Code Review
- **Quiz:** Code Tricks: Time Steps, Snowfall, and Elevation
  5 questions

Code Tricks: Time Steps, Snowfall, and Elevation

Quiz30 minutes • 30 min

Submit your assignment

Due March 5, 11:59 PM ISTMar 5, 11:59 PM IST
Attempts 3 every 8 hours
Start assignment

Receive grade

To Pass 80% or higher

Your grade

-Not available

Like
Dislike
Report an issue

Back

Code Tricks: Time Steps, Snowfall, and Elevation

Graded Quiz. • 30 min. • 5 total points available.5 total points

DueMar 5, 11:59 PM IST

1.

Question 1

Set up the code to run for 25,000 years of simulated time. Does the code reach equilibrium by the end of this time?

1 point

Yes (Answer)

no

2.

Question 2

What elevation does the ice sheet reach, in meters?

Answer: 3867.5 m

1 point

3.

Question 3

What is the longest value of the time step that you can use, which still looks like the solution you got above?

Answer: 125

1 point

4.

Question 4

When the model uses a time step which is just a bit too long, what happens to the run?

1 point

It crashes immediately

It starts up unstable (with oscillating values) then settles down for the equilibrium part of the simulation.

It runs OK while it's spinning up but gets twitchy and blows up after it reaches equilibrium

Answer: Flow Numbers overflow with large step!

5.

Question 5
If you double the snowfall rate, the equilibrium elevation of the ice sheet

Answer: With 2*snowfall: New maximum elevation: 5477 is 5477/3867 = 1.4163

With 4*Snowfall, new max elevation: 7746/3867 = 2.0031

1 point

doubles

more than doubles

less than doubles (Answer)

stays the same

Coursera Honor Code Learn more

I, **Lekhraj Sharma**, understand that submitting work that isn't my own may result in permanent failure of this course or deactivation of my Coursera account.

SubmitSave draft

Like
Dislike
Report an issue

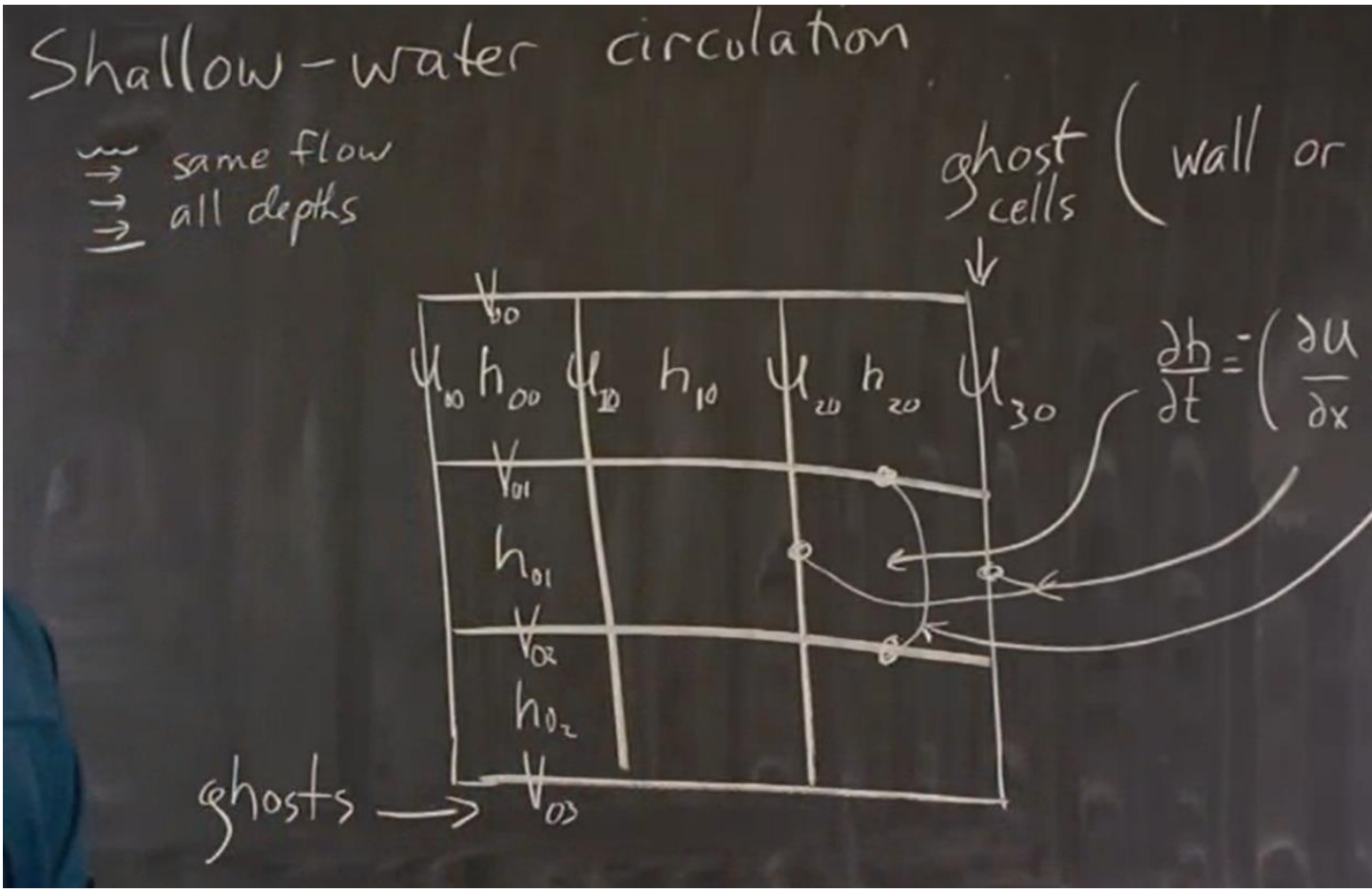
From <https://www.coursera.org/learn/global-warming-model/exam/WRGJR/code-tricks-time-steps-snowfall-and-elevation/attempt>

2D Gridded Shallow Water Model

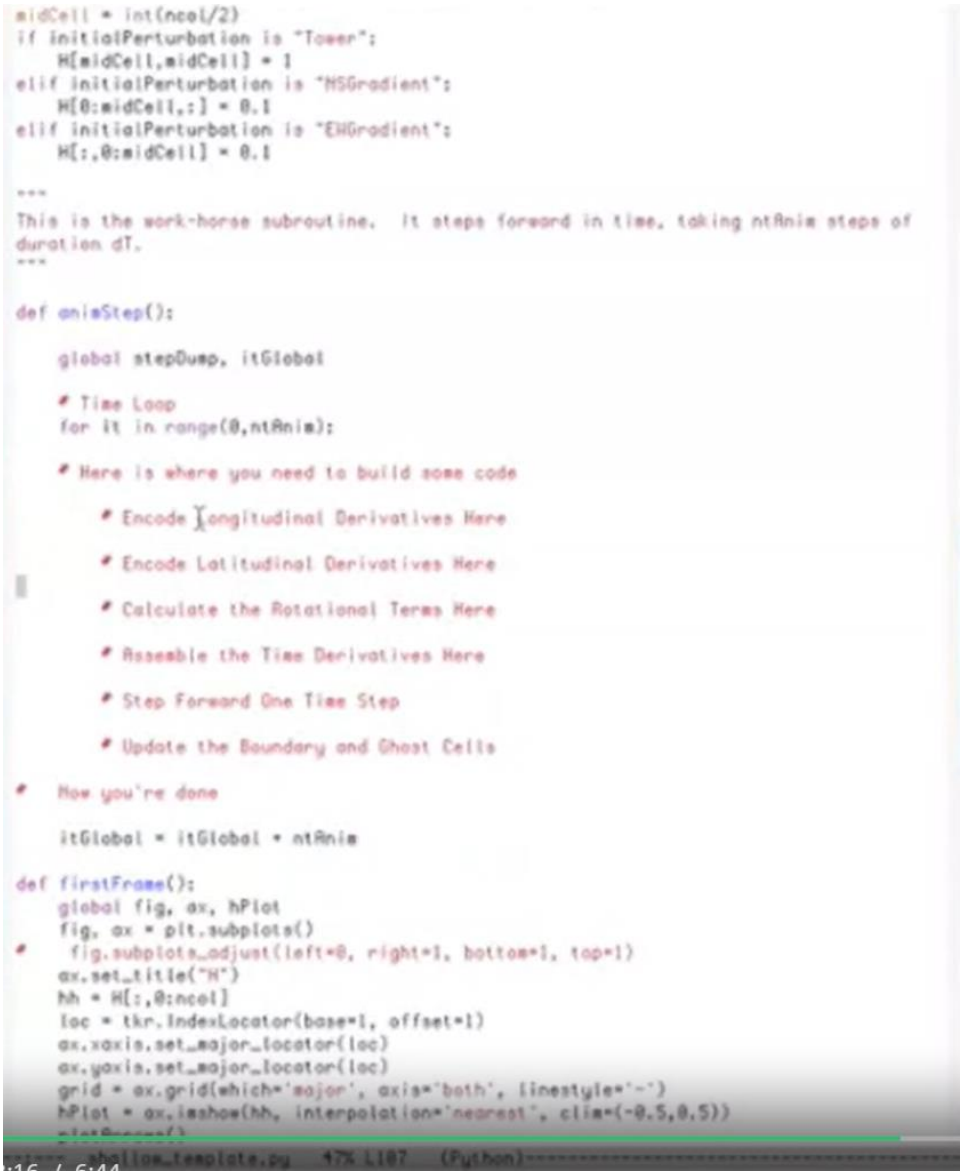
Wednesday, February 8, 2023
11:36 PM

Planetary rotation and fluid flow were explained in Part I of this class, Unit 6, on Weather and Climate.

From <https://www.coursera.org/learn/global-warming-model/home/week/4>



```
"""

The first section of the code contains setup and initialization
information.  Leave it alone for now, and you can play with these later
after you get the code filled in and running without bugs.

---

# Set up python environment.  numpy and matplotlib will have to be installed
# with the python installation.

import numpy
import matplotlib.pyplot as plt
import matplotlib.ticker as tkr
import math

# Grid and Variable Initialization -- stuff you might play around with

ncol = 3          # grid size (number of cells)
nrow = ncol

nSlices = 2         # maximum number of frames to show in the plot
ntAnim = 1          # number of time steps for each frame

horizontalWrap = True # determines whether the flow wraps around, connecting
                      # the left and right-hand sides of the grid, or whether
                      # there's a wall there.
interpolateRotation = True
rotationScheme = "PlusMinus"   # "WithLatitude", "PlusMinus", "Uniform"
windScheme = ""  # "Curled", "Uniform"
initialPerturbation = "Tower"    # "Tower", "NSGradient", "EWGradient"
textOutput = False
plotOutput = True
arrowScale = 30

dT = 600 # seconds
G = 9.8 # m/s2
HBackground = 1000 # meters
dX = 10.E3 # meters
dxDegrees = dX / 110.e3
flowConst = G / dX  # m/s2
dragConst = 1.E-6  # about 10 days decay time
meanLatitude = 30 # degrees

# Here's stuff you probably won't need to change

latitude = []
rotConst = []
windU = []
for lat in range(0,nrow):
```



```
midCell = int(ncol/2)
if initialPerturbation is "Tower":
    H[midCell,midCell] = 1
elif initialPerturbation is "NSGradient":
    H[0:midCell,:] = 0.1
elif initialPerturbation is "EWGradient":
    H[:,0:midCell] = 0.1

"""
This is the work-horse subroutine.  It steps forward in time, taking ntAnim steps of
duration dT.
"""

def animStep():

    global stepDump, itGlobal

    # Time Loop
    for it in range(0,ntAnim):

    # Here is where you need to build some code

        # Encode Longitudinal Derivatives Here

        # Encode Latitudinal Derivatives Here

        # Calculate the Rotational Terms Here

        # Assemble the Time Derivatives Here

        # Step Forward One Time Step

        # Update the Boundary and Ghost Cells

#   Now you're done

    itGlobal = itGlobal + ntAnim

def firstFrame():
    global fig, ax, hPlot
    fig, ax = plt.subplots()
#    fig.subplots_adjust(left=0, right=1, bottom=1, top=1)
    ax.set_title("H")
    hh = H[:,0:ncol]
    loc = tkr.IndexLocator(base=1, offset=1)
    ax.xaxis.set_major_locator(loc)
    ax.yaxis.set_major_locator(loc)
    grid = ax.grid(which='major', axis='both', linestyle='-')
    hPlot = ax.imshow(hh, interpolation='nearest', clim=(-0.5,0.5))
    plotBase()
```


The shallow-water equations are not necessarily called that because
water has to be shallow, but because the flows are the same at all water depths.
So we're doing the calculation of water flow that is
independent of water depth where you don't have different flows.
You have to maintain different grid cells in the vertical,
you just need a lattice of grid cells in two dimensions in the horizontal.
So this is what that grid looks like, so this is latitude and longitude.
And each grid cell has an elevation of water level height which is defined,
just like in the ice sheet model, it's defined in the cell centers.
And then there are flows between the cells,
which are like in the ice sheet model designed on the cell edges.
So the grid could look like h00, h01, h02,
and then v velocities, which are the north south velocities defined on the south
faces there are now four of them as opposed to the three elevations.
Just like in the ice sheet model, you have to worry about the boundaries.

Now the north and
south boundaries, we are going to say that there is a wall there so.
These flows are both going to all equal zero across that wall and that wall.
In the horizontal dimension there will be two options you're going to build
into your code.
One is to put walls there which means that you'll be setting these velocities,
these u values to be zero.
And the other is to have it wrap around in which case,
you have stuff can flow out one side and it would just flow in the other side.
So the way that you calculate the evolution of the elevation
of the surface is by tracking the flows coming and
going out the boundaries.
So dU/dX, that's saying you have,
more flowing in this side than that side.
That would give you a value
du dx that would actually be decreasing as you go this way.
And so the negative of that multiplied by the aspect ratio.
Tells you how much of this water is going to be piling up and
making this box get deeper.
Play video starting at :2:31 and follow transcript2:31
So to accomplish this model,
you are going to be given a template which has a lot of the overhead dealt with for
you already like that makes the animation and what not.
So this is what that template looks like.
There's some stuff here at the top that you can play with after you get
everything going.
Play video starting at :2:52 and follow transcript2:52
Some stuff here that you probably won't need to change.
It's just sort of set up and then here is the main time
stepping subroutine, and there are instructions for
how to write Python loops to calculate the derivatives.
The DUDX and the VDDY and things like that.
In this loop, so you just need to flesh out this loop here.
As the template comes initially,
it's simplest to de-bug a thing like this when it's got a really small grid.
So, this is what the thing will look like, just in the three by three grid.
And the colors are the elevations and
the arrows are the flows defined at the cell faces, like I said.
And this particular simulation starts out with a big tower of water in the middle.
And you get wild flow initially but then it settles down to a geostrophic
balance where the rotation is balancing the pressure gradient sort of sideways.
Three by three isn't very exciting so if we expand that to a bigger grid so
there's the initial transient waves going all over the place.
But then it settles down, and you get this sort of rotation
going around the perturbation, this high pressure in the center.
And this wave will tend to drift to the west.
Because the rotation of the planet in higher latitudes is faster
than the rotation rate in lower latitudes and that's set in this model.
And so that tends to make raspy waves like this drift toward the west.

So you can see this cell is getting redder and redder than that one as this peak is
sort of going to slowly move across this grid, toward the West.
I should say that this is a very, very small ocean.
And it's got a very big degree of rotational change between
the lower part latitude and the high latitude here.
So it's like a small ocean on a small planet, but with the limited power of
Python, we have to do what we can to make things run quickly.
Play video starting at :5:21 and follow transcript5:21
Here's another trick that the shallow water circulation code can do.
In this case, you have winds that are blowing in
this direction toward the west in the lower
Play video starting at :5:35 and follow transcript5:35
half of this domain and in the upper half the winds are blowing toward the east.
And with the rotation what that does is it causes water to pile up
in the center of this gyre.
And once this water piles up,
that pressure gradient actually starts to drive the flow around the gyre.
So it's going the way the wind would tell it to go, but
it's going there because the pressure gradient in the water is telling it to,
not because the wind is telling it to.
These arrows arise as this pressure difference in the water arises.
And then this is like a giant Rossby wave, it's drifting to the west.
And what that does in this case where you have walls at the east and west boundaries
is it tends to lead to an intense flow on the Westward boundary.
This is called Westward intensification and it's responsible for
the Gulf stream and the Kuroshio and other Western boundary currents
around the world that are very intense for this dynamical reason.

From <https://www.coursera.org/learn/global-warming-model/lecture/8pW0H/how-the-model-works>

Model Description

Overview

shallow_template

PY File

This is a Python code to solve the time evolution of a slab of shallow water. This calculation is too complex for a spreadsheet. The numerical guts could be done in Fortran, and it would be capable of much higher resolution (more grid points) than the Python version could handle in a reasonable amount of computer time. Plotting an animation in Fortran is not built-in or convenient in a code that might take a long time to run (like a climate model), so the usual strategy would be to save data files periodically from the running Fortran, then use some other software to read those data files and make a plot or movie (this is called "post processing").

This problem is complex enough that you are given a "template" in Python which sets up the grid and the plotting routines. The automatic grader would accept a Fortran code which mimics the I/O of the Python code, but due to the lack of animation in Fortran a comparable template for Fortran is not provided. If you want to take on the challenge of writing a Fortran code that will pass the automatic grader, I say go for it and email me a copy when you get it working!

The flow in the model is driven by differences in the elevation of the water surface, which arise from the flow itself. The flow is also altered by rotation (as on a rotating planet), potentially by wind, and by friction. The water is assumed to be homogeneous in the vertical, no differences in temperature, density, or velocity are tracked. Because of the vertical homogeneity, these are called the "shallow water equations". A similar formulation can be used to model flows in the atmosphere, with the same assumption that the fluid "piles up" in some places, generating differences in pressure at the base of the fluid which drive fluid flow.

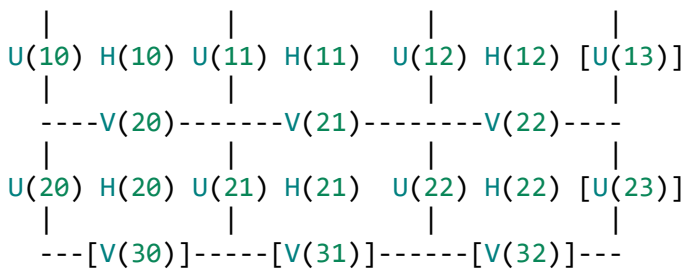
In its simplest configuration, this model can start with an initial "hill" of high water level in the center of the computational grid. Water starts to flow outward from the hill, but it rotates to the side, and so tends to find a pattern where the flow is going around and around the hill, rather than simply flowing straight down as it would if there were no rotation.

Note that steps have been taken in the template file to accelerate the evolution of the model, so that the relatively slow Python language can make plots evolve at a visually satisfying fast rate. These adjustments are (1) low gravity, which slows down the waves and allows a longer time step, (2) a small ocean with grid cells that are not too large, so that the amount of water in them can change quickly, and (3) a much steeper change in rotation rate with latitude, as if the ocean was on a very small planet.

The Grid

For the simple 3x3 case, the placement of the velocities and elevations is as shown below. Longitudinal velocities (u) are defined on the left-hand cell faces, and latitudinal velocities (V) on the cell tops. The elevation of the fluid in each box (H) is defined in the cell centers.

```
  ----V(00)-------V(01)--------V(02)----
  |           |            |           |
U(00) H(00) U(01) H(01)  U(02) H(02) [U(03)]
  |           |            |           |
  ----V(10)-------V(11)--------V(12)----
```



Variables that are enclosed with square brackets in this diagram are "ghost" variables. They aren't computed as part of the real grid, but are used to make it simpler to calculate differences, say between the North/South velocities (V) at the top and the bottom of each cell.

The Differential Equations

dU/dT = C_rotation * V - C_flow * dH/dX - C_drag * U + C_wind

dV/dT = - C_rotation * U - C_flow * dH/dY - C_drag * V

dH/dT = - (dU/dX + dV/dY) * Hbackground / dX

The terms like dU/dT denote derivatives, in this case the rate of change of the velocity (U) with time (t). C_ terms are constants, for rotation, induction of flow, drag, or wind input. Notice how the rotation term transfers energy between the U and V velocities. Terms like dV/dY or dH/dX are spatial derivatives, how much the velocity (V) changes with latitude (y), for example. Ultimately these "time tendency" (e.g. dU/dT) terms will be used to update the values according to, for example

U(time+1) = U(time) + dU/dT * delta_t

where delta_t is a time step.

The first equation can be interpreted as a list of things (on the right hand side) which tend to make the velocity (u) change with time (dU/dT). Rotation changes the flow direction, moving velocity between the two directions U and V according to a rotational constant, C_rotate, where a higher number (away from zero) would rotate faster, and the sign of the constant determines the direction of rotation. The term dh/dX tells whether the sea surface is sloping; if it is, it drives the flow to accellerate according to a flow constant C_flow.

Drag slows the flow down, the larger the flow (U), the faster the slowdown (dU/dT). And finally we'll set up to allow wind to blow, in an East/West (U) direction, driving circulation.

Mapping the Equations Numerically onto the Grid

The differential equations, above, apply to a continuous fluid, where you can imagine a slope of the sea surface as a tangent to a wavy surface. Numerically, we cast these equations onto much coarser systems of boxes, and derive our pressure driver, for example, from differences of heights between adjacent boxes. We need to do this paying attention to how the variables (U, V, and H) are arrayed in space on the grid (above).

For example, in the equation for dU/dT, above, the flow is driven by a sloping sea surface (dH/dX). Looking at the grid diagram, the slope in the sea surface (H), appropriate to the second U in from the left, U(01), would span the position of U(01), to be H(01) - H(00), divided by the grid spacing delta_x.

```
  |           |           |           |
U(00) H(00) U(01) H(01)  U(02) H(02) [U(03)]
  |           |           |           |
```

In the code, we want to have grids of the three important variables U, V, and H, and also we'll construct arrays of intermediate variables like dH/dX and dV/dY. They will be indexed in the grid, as in

dU/dT[irow,icol] = flowConst * dH/dX[irow,icol] + ...

Pay attention to the grid diagram to make sure the indexes work out in constructing these arrays.

There will probably be errors in your first attempts; half of the art of coding is finding bugs, figuring out why your code isn't doing what you expect it to. To make things easiest to debug, the template provided is set up for a very simple 3x3 case, just one time step, to give you results you can compare with what we'll give you.

What You Need to Do

A shell of a code has been provided for you, which sets up the variables and the driving parameters of the model, and contains routines for making the animation. What you need to do is to fill in code to calculate the derivative terms in the governing equations above, as they are discretized onto the staggered finite difference grid.

Longitudinal Derivatives

Loop over nrow and ncol (stepping from 0 to nrow-1 or ncol-1)

Calculate dH/dX (variable dHdX), making sure to put the value in each index [irow, icol] so that it applies to the horizontal velocity at that index location (U[irow, icol]).

If the variable horizontalWrap is set to True, the flow out the right-hand side of the domain (U[:,ncol]) will equal the flow in from the left side (U[:,0]). (The colon means, in this case, "all rows")

Assume that there is are "ghost cells" at the right-hand end of the U and H arrays. U[:,ncol] = U[:,0], and H[:,ncol] = H[:,0].

For example, if ncol = 3, the H values that are in the domain will have indices 0, 1, and 2. For convenience, we'll set up an H[:,3] set of cells, which equal the H[:,0] cells, so we can take a difference for dHdX[:,3] from H[:,2] and our ghost H[:,3], which are on either side of U[:,2].

Or, if horizontalWrap is set to False, the U velocities at the left and right sides of the domain will be set to zero.

Also calculate dU/dx (dUdX), again making sure that the result that has index [irow, icol] applies to the elevation H[irow, icol]. Assume that the ghost cells U[:,ncol] are already set, if it's wrapped, or that the boundary velocities U[:,0] and U[:,ncol-1] = 0, if it's a wall. (There can be flow along the wall (V) but not through it (U)).

For a test 3x3 grid, after this routine has run one time, the values of H and dHdX will be (notice the ghost cells on the right, the fourth element in each line)

```
 H    [[ 0.  0.  0.  0.]
       [ 0.  1.  0.  0.]
       [ 0.  0.  0.  0.]]

 dHdX [[ 0.  0.  0.  0.]
       [ 0.  1.e-3 -1.e-3  0.]
       [ 0.  0.  0.  0.]]
```

The velocities are all zero, initially, so their gradient will also be zero. On the second time through the routine, U and dUdX should look like

```
U at beginning of second step:
[[ 0. 0. 0. 0.]
 [ 0. -0.00098  0.00098 0.] [ 0. 0. 0. 0.]]

dUdX after second step:
[[ 0. 0. 0.] [ -9.80000000e-08   1.96000000e-07  -9.80000000e-08] [ 0. 0. 0.]]
```

Latitudinal Derivatives

Within a loop over all grid cells, calculate dHdY, remembering that dHdy[irow, icol] should be comprised of H values that straddle a particular V[irow, icol].

The northern and southern boundaries of the domains are always walls. This means that the flows at the north boundary (V[0,:]), and the south (a ghost cell would be V[ncol,:]) are both set to zero. Assume that this will be true going into this loop, and that the ghost cell exists.

The gradient in the surface elevation, dHdY, should be set to zero at the top boundary. dHdY[0,:] would be used to calculate V[0,:], which is going to be zero anyway.

After the first time through this time loop, dHdY should look like

```
dHdY [[ 0. 0. 0.]
      [ 0. 1.e-3 0.]
      [ 0. -1.e-3 0.]]
```

and entering the second time through the loop, the velocities (V, notice the ghost cells in the fourth row) should look like

```
V [[ 0. 0. 0.]   [ 0. -0.00098 0.]   [ 0. 0.00098 0.]   [ 0. 0. 0.]]
```

and this will drive a gradient dVdY of

```
dVdY [[ 0. -9.80000000e-08 0.]      [ 0. 1.96000000e-07 0.]      [ 0. -
9.80000000e-08 0.]]
```

Rotation

The effect of Earth's rotation is to transfer velocity between the U and V directions. There are two ways to calculate this effect: an easy but somewhat biased way which will mostly work but lead to some weird flow patterns, and a better way. You don't have to do either one to pass the Code Check, but there are optional code checkers for both schemes if you want to try your hand. The two formulations diverge in longer simulations, where the simpler method will generate diagonal "stripes" of water level (waves) across the grid, as an artifact of its less-accurate method.

Easier First Method: Loop over all rows and columns, and calculate

rotU[irow,icol] = rotConst[irow] * U[irow,icol]

This array will be used to update V[irow,icol]. rotConst is a function of latitude so that rotation can be stronger in high latitudes, like rotation on a real sphere.

Similarly, calculate rotV from rotConst and V.

Better Method (Maybe add this as a second step, after getting the simpler way to work first)

1. The problem with the first method is that the U and V values are not located at the same places in the grid. The better method (save this task for later, after you get the first way working, is my advise) is, first, to interpolate the U and V values onto the cell centers. For U, this means averaging adjacent values, and putting them into the index so that the irow, icol values correspond to those of the H value in each cell. Put them into a temporary variable array which you can call whatever you wish. Declare and initialize the array using numpy.zeros, as was done above for U etc. Do the same thing for V, interpolating them into its own array with grid points located in the cell centers. It would be a good idea to print them out once, for a test, and make sure it's doing what you want it to.

2. Next calculate the rotational transformation of U and V as gridded on the H points, by multiplying each array by rotConst[]. You are done with the temporary arrays you generated before, so you could replace their values with the product, or you could create new arrays, whatever you like.

3. Finally, the rotated velocities, placed at the cell centers, need to be back-interpolated to the grid locations where the velocities are. For example, the rotational addition to U velocity at the boundary between two cells should come from averaging the results you just got for the two cell centers on either side of the cell face you're interested in.

For the component that adds to V, generate a variable array called rotU (rotConst * U), each index of which corresponds with the placement of the V values on the grid. rotU[0,:] will apply to V[0,:], which are set at zero because they are at the North wall of the simulation. The velocities at the South wall, also zero, are stored in the ghost cells V[ncol,:], which won't get updated or need rotU[] values anyway.

For updating U, we'll use rotV (rotConst * V). Here there are two cases to plan for, whether the flow wraps around connecting the left and right sides of the domain, or not.

If there's a wall at the boundaries, U[:,0] and U[:,ncol] (ghost cells) will be held at zero, so rotV[:,0] has to be set to zero (or left that way, as it was initialized).

If the flow wraps around, the rotational adjustment to U[:,0] (the eastern and western boundaries) should come from averaging the values in the centers of the eastern and western-most boxes.

For the simple configuration set up in the template python file, the first time to check your values of the rotational terms is after the second time step. The first time step, the velocities were zero going into the calculation, so the rotational terms calculated in that time step would be also zero. After the second time step, using the simplest, non-interpolated scheme above, the values I get for rotU and rotV are

```
rotU
[[ 0. 0. 0.]
 [ 0.  3.43000000e-08  -3.43000000e-08]
```

```
 [ 0. 0. 0.]]

rotV
[[ 0. 0. 0.]
 [ 0.   3.43000000e-08  0.]
 [ 0.  -2.51533333e-08  0.]]
```

Note how the rotation rate depends on the latitude but not the longitude: this is with the rotationScheme variable set to "PlusMinus", which imposes a linear change in rotation rate with latitude (they call this the "beta plane approximation").

When I do the interpolation, the numbers I get are

```
rotU
[[  0. 0. 0.]
 [  8.57500000e-09  0.  -8.57500000e-09]
 [  8.57500000e-09  0.  -8.57500000e-09]]

rotV
[[ 0.   1.08616667e-08   1.08616667e-08]
 [ 0. 0. 0.]
 [ 0.  -6.28833333e-09  -6.28833333e-09]]
```

Time Derivatives

Encode the equations for dU/dT, dV/dT, and dH/dT, given above, by looping over the grid and calculating values to put in arrays dUdT, dVdT, and dHdT. Be sure that the indices of the arrays correspond to those of the arrays U, V, and H that they are going to update. It's very easy to make a mistake of this type, and the flow results you get will be strange and non-physical.

Step forward in time by looping over the grid, updating each variable U, V, and H by adding the time derivative multiplied by the time step.

Maintain the Ghost Cells

At the end of each time step, do any maintenance that needs doing for the ghost cells, which are cells at the edges that mirror other cells in the grid to make it easier to calculate spatial derivatives at the edges.

The velocities at the north wall should be zeroed.

If the horizontal flow wraps around the grid (meaning: you should write to code to see if the variable horizontalWrap is set to a value of True, and if it is), set the ghost cells for U and H, for indices [:, ncol], to equal their values at indices[:,0]. Your code should calculate new values for U and H in the leftmost grid cell (column 0), while you need to update the ghost cell. This is in column ncols, because the numbering of the columns starts at 0. Columns numbered 0 to (ncols-1) are in the computational grid, and column number ncols is an "extra" ghost cell.

If the flow doesn't wrap, set U = zero at the eastern and western boundaries (indices [:,0] and [:,ncol]).

Real-time Graphical Output

The code template has been set up to make a plot of the coupled elevation / flow system in real time as the model run progresses. The colors in the boxes represent the elevation, with warmer colors (reds and blues) indicating high elevations, and colder (blues) are "holes" or dips in the sea surface. The flows are defined on the cell faces, and are indicated by arrows that originate on the faces and point into the box in which the flow is going. There is a parameter in the code, arrowScale, which turns up or down the length of the arrows, to get them in a range you can see.

Tinkering with the Code

You can learn a lot and have a lot of fun playing with the parameters in the code: the number or size of the grid cells, the rotation rate, the density of the fluid, the driving wind strength. If you mess with anything like, pay attention to the time step. In non-rotating flow, a larger grid size will buy you a longer time step, but in a rotating flow as here, the time step is limited by resolving the rotation.

From <https://www.coursera.org/learn/global-warming-model/supplement/aK0dY/model-description>

Week 4 Quiz

Friday, February 10, 2023
6:37 PM

- **A 2-D Gridded Shallow Water Model**
- **Video:** VideoHow the Model Works
  . Duration: 6 minutes6 min
- **Reading:** ReadingModel Description
  . Duration: 10 minutes10 min
- **Programming Assignment:** Code Check
  . Duration: 3 hours3h
- **Practice Programming Assignment:** Optional Code Check, Simple Rotation Scheme
  . Duration: 3 hours3h
- **Practice Programming Assignment:** Optional Code Check, Interpolated Rotation
  . Duration: 3 hours3h
- **Peer-graded Assignment:** Code Review
  Grading in progress
  Review Your Peers: Code Review
- **Quiz:** Code Trick: Geostrophic Flow and a Drifting Rossby Wave
  2 questions
- **Quiz:** Code Trick: Gyre Circulation with Westward Intensification
  3 questions

Code Trick: Geostrophic Flow and a Drifting Rossby Wave

Quiz30 minutes • 30 min

Submit your assignment

Due March 12, 11:59 PM ISTMar 12, 11:59 PM IST
Attempts 3 every 8 hours
Start assignment

Receive grade

To Pass 80% or higher

Your grade

-Not available
Like
Dislike
Report an issue

Back

Code Trick: Geostrophic Flow and a Drifting Rossby Wave

Graded Quiz. • 30 min. • 2 total points available.2 total points
DueMar 12, 11:59 PM IST

1.

Question 1
Set up the code to simulate a Rossby wave in the ocean, a pile of water flowing around itself. Increase the resolution to at most 10x10 grid cells, and set the following parameter values

```
ncol = 10
nslices = 200
ntAnim = 200
horizontalWrap = True
interpolateRotation = True
rotationScheme = "PlusMinus"
plotOutput = True
arrowScale = 30
```

Run the code, and save a copy of it for Code Review. After the flows settle down a bit, which direction do the flows go around the hill of water?

1 point

Clockwise (Answer)

counter-clockwise

2.

Question 2
In the above simulation, the change in rotation rate as a function of latitude causes the hill to drift, very slowly, in which direction?

1 point

to the East

to the West (Answer)

to the North

to the South

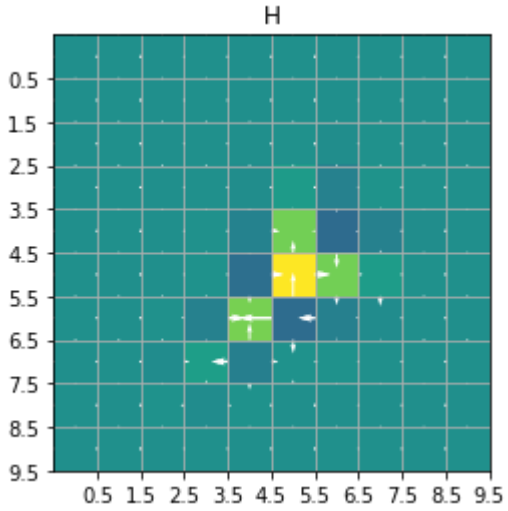
Coursera Honor Code [Learn more]

I, **Lekhraj Sharma**, understand that submitting work that isn't my own may result in permanent failure of this course or deactivation of my Coursera account.

SubmitSave draft

Like
Dislike
Report an issue

From <https://www.coursera.org/learn/global-warming-model/exam/jXkPb/code-trick-geostrophic-flow-and-a-drifting-rossby-wave/attempt>



- **Reading:** ReadingModel Description
  . Duration: 10 minutes10 min
- **Programming Assignment:** Code Check
  . Duration: 3 hours3h
- **Practice Programming Assignment:** Optional Code Check, Simple Rotation Scheme
  . Duration: 3 hours3h
- **Practice Programming Assignment:** Optional Code Check, Interpolated Rotation
  . Duration: 3 hours3h
- **Peer-graded Assignment:** Code Review
  Grading in progress
  Review Your Peers: Code Review
- **Quiz:** Code Trick: Geostrophic Flow and a Drifting Rossby Wave
  2 questions
- **Quiz:** Code Trick: Gyre Circulation with Westward Intensification
  3 questions

Code Trick: Gyre Circulation with Westward Intensification

Quiz30 minutes • 30 min

Submit your assignment

Due March 12, 11:59 PM ISTMar 12, 11:59 PM IST

Attempts 3 every 8 hours

Start assignment

Receive grade

To Pass 80% or higher

Your grade

-Not available

Like

Dislike

Report an issue

Back

Code Trick: Gyre Circulation with Westward Intensification

Graded Quiz. • 30 min. • 3 total points available.3 total points

DueMar 12, 11:59 PM IST

1.

Question 1

Beginning with your previous simulation, change the following parameters

```
ncol = 5
nSlices = 400
ntAnim = 1000
horizontalWrap = False
windScheme = "curled"
initialPerturbation = ""
arrowScale = 30
```

Run the simulation to the end and observe how the surface elevation interacts with the flow. Save your code for Code Review, after this.

Where does the water tend to pile up?

1 point

on the Eastern side

on the Western side

in the middle of the basin (Answer)

on the North and South edges

2.

Question 2

Where are the flows the strongest, in the steady-state circulation

1 point

along the Southern edge

along the Northern edge

along the eastern edge

along the western edge (Answer)

3.

Question 3

Which edge of the Atlantic is the Gulf Stream found on

1 point

the Northern edge

the Southern edge (the equator, maybe?)

the eastern edge

the western edge (Answer, West to Atlantic Ocean but east of US, Florida side)

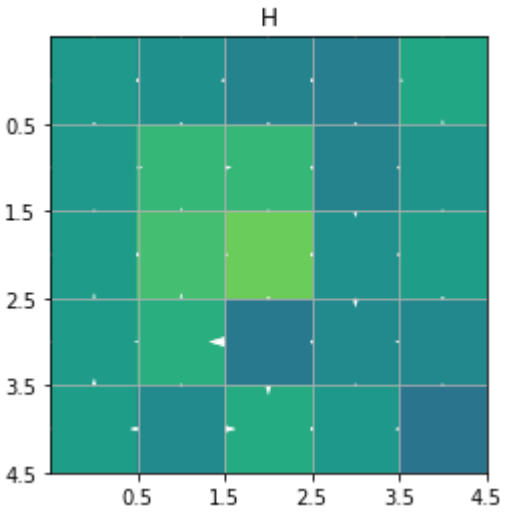
Coursera Honor Code [Learn more]

I, **Lekhraj Sharma**, understand that submitting work that isn't my own may result in permanent failure of this course or deactivation of my Coursera account.

SubmitSave draft

Like
Dislike
Report an issue

From <https://www.coursera.org/learn/global-warming-model/exam/GKa6j/code-trick-gyre-circulation-with-westward-intensification/attempt>

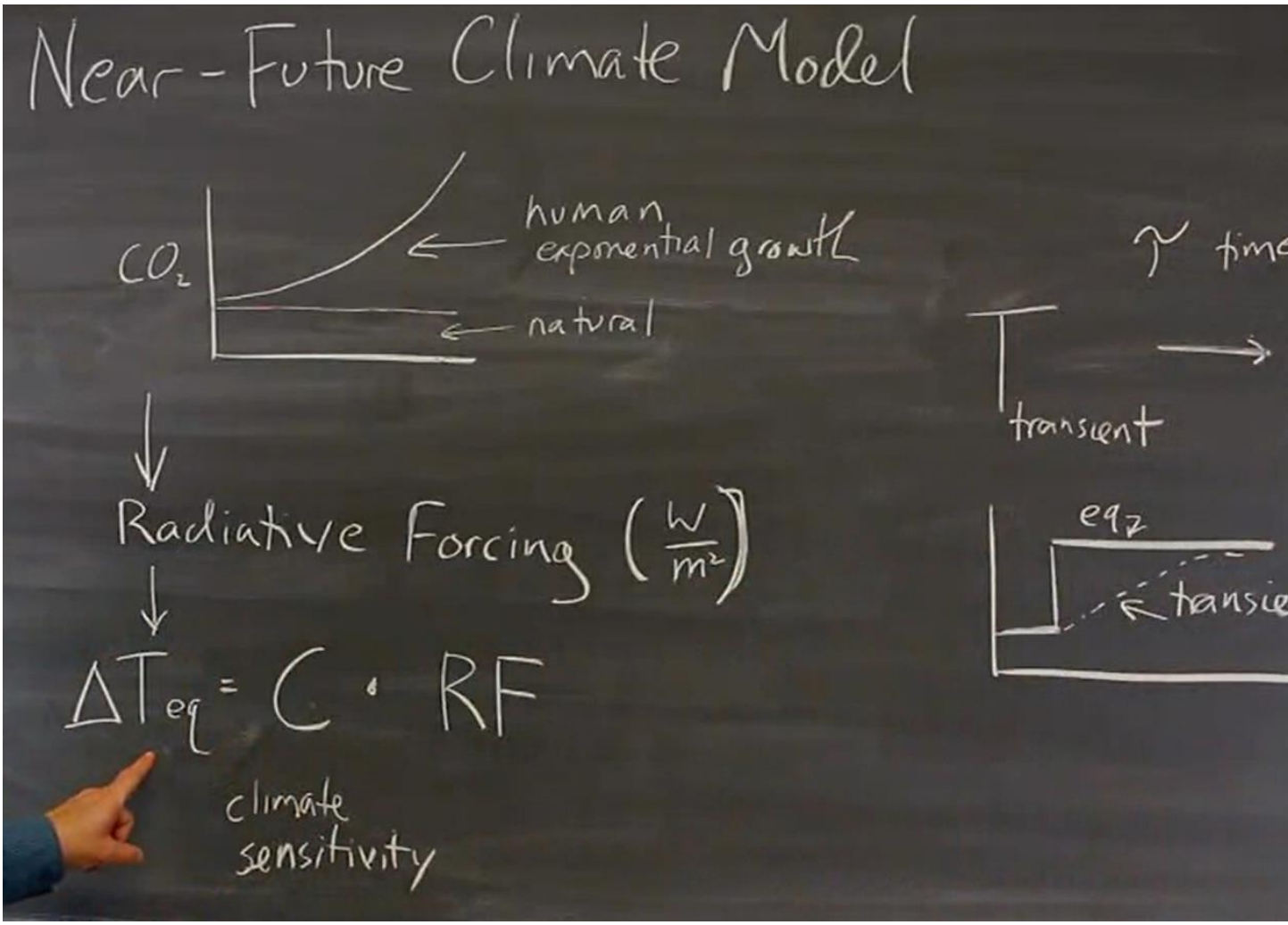


Near Future Climate Model (Model of Climate Change Today!)

Friday, February 10, 2023
7:49 PM

Background for this model was presented in Part I of this class, Unit 9, The Perturbed Carbon Cycle.

From <https://www.coursera.org/learn/global-warming-model/home/week/5>



```
import numpy
import matplotlib.pyplot as plt
import math

timeStep = 5 # years
eqCO2 = 280
initCO2 = 290
CO2_exp = 0.0225
CO2RampExp = 0.01
aerosol_Wm2_now = -0.75
watts_m2_2x = 4
climateSensitivity2x = 3
climateSensitivityWm2 = climateSensitivity2x / watts_m2_2x
TResponseTime = 20

years = [ 1900 ]
bauCO2 = [ initCO2 ]
incCO2 = [0]
rfCO2 = [0]
rfMask = [ 0 ]
rfCO2Ramp = [0]
rfMaskRamp = [0]
rfTot = [0]
Teq = [0]
TTrans = [0]
rampCO2 = [initCO2]
rfTotRamp = [0]
TeqRamp = [0]
TTransRamp = [0]

# business

while years[-1] < 2100:
    years.append( years[-1] + timeStep )
    bauCO2.append( eqCO2 + (bauCO2[-1]-eqCO2) * (1 + CO2_exp * timeStep) )
    incCO2.append( (bauCO2[-1] - bauCO2[-2]) / timeStep )
    rfCO2.append( watts_m2_2x * math.log( bauCO2[-1]/eqCO2 ) / math.log(2) )

i2015 = years.index(2015)
aerosolCoeff = aerosol_Wm2_now / \
    ( ( bauCO2[i2015] - bauCO2[ i2015-1 ] ) / timeStep )

for i in range(1, len(years)):
    rfMask.append( max( incCO2[i]*aerosolCoeff, aerosol_Wm2_now ) )
    rfTot.append( rfCO2[i] + rfMask[i] )
    Teq.append( rfTot[i] * climateSensitivityWm2 )
    TTrans.append( TTrans[-1] + (Teq[i] - TTrans[-1]) * timeStep / TResponseTime )
```

The near future climate model is intended to try to
capture some of the dynamics of what's going on the nowish,
this decade, this 100 years with rising CO2.
The radiative forcing that that imposes on the climate.
And then the time evolving planetary response to that
change in the energy balance.
[COUGH] So, the CO2 concentration in the atmosphere, we're sort of
decomposing into a natural constant amount that was there before we were.
And then an exponentially growing part that's due to the industrial activity.
So given the CO2 concentration, we can calculate the radiative
forcing from that, which is number in watts per square meter.
Which indicates how much the change in CO2 from the initial
value has changed the energy balance.
So, eventually the planet warms up and makes the energy balance go back to zero.
So, this radiative forcing is defined after you put the CO2 in the air,
but before the temperature has had a chance to change at all.
So the radiative forcing is directly,
linearly proportional to the equilibrium temperature change.
Now it could be that in reality the temperature change could be some more
complicated function of the radiative forcing.
But to at first approximation, one watt per square meter
radiative forcing gets you three-quarters of a degree of temperature change,
and that proportionality is the climate sensitivity.
There's uncertainty to that, of course.
So the equilibrium temperature then is the temperature
that the planet is relaxing to given enough time.
But there's a long, non-neglible time scale for how long it takes for
the planet to reach the equilibrium temperature.

So, if the equilibrium temperature were to just suddenly change like this,
the transient temperature kind of relax torted on some longer timescale.
So this transient temperature is the one that's actually is what's controlling our weather.
My python version of the world without us code looks like
it's got some variables initialized at the beginning and
a bunch of lists that are initialized and get filled up in a series of for loops.
And if we run it.
This is the CO2 concentrations as a function of time.
So, here is the business as usual, exponential ramp up.
And then here is the world without us,
where the CO2 is sort of slowly declining as it dissolves in the ocean.
Play video starting at :3:7 and follow transcript3:07
We changed the plot, Comment out that one.
Reveal that one.
We will see the radiative forcing that are driving everything.
Play video starting at :3:21 and follow transcript3:21
So this is the radiative forcing for masking.
It is proportional to the rate of increase of CO2,
just kind of call it industrial activity.
And then this suddenly drops to 0 at the world without us,
because the smoke all gets cleaned out of the air and that happens quickly.
So, in the business as usual scenario, the masking
is assumed to just stay at this value.
And so the temperature, really the total radiative forcing
starts to go up faster due to the rising CO2 concentration.
Here is the radiative forcing for the world without us simulation,
so it's following the business as usual up to the present day.
And then when the CO2 sort of stops rising but declines slowly that's here.
And then you also get this extra boost because the masking
effect goes away quickly.
So what we'll see
Play video starting at :4:38 and follow transcript4:38
If we plot the temperatures.
Play video starting at :4:42 and follow transcript4:42
The blue is business as usual and the green is actually the world without us.
And you see that pulling away the masking cooling effect from the aerosols
results in a slight increase in temperature in spite of the fact that
people are no longer on the planet to be putting CO2 in the air.

From <https://www.coursera.org/learn/global-warming-model/lecture/Ea5bP/how-the-model-works>

Description of the Model Formulation

Your job is to simulate the near-term future of Earth's climate as a function of the three main uncertainties: future CO_2 emission, the climate sensitivity, and the current cooling impact of industrial aerosols and short-lived greenhouse gases (the masking effect).

This is a time-stepping calculation, like the last one. The simulation has two stages, one with people releasing CO_2 and generating aerosols, which we'll call Business-as-Usual, and a second, diverging from the first, in which we suddenly stop both of those things, at which point the CO_2 largely persists, while the aerosols go away immediately.

The calculation can be done straightforwardly in a spreadsheet, and with a bit more logistical overhead in python. As with previous exercises, you will ultimately submit your python code for code check and review, while you won't really get credit for a spreadsheet version. However, you might find it helpful to develop a spreadsheet version; I generally do, when developing some new ideas in a code.

Numerical Guts of the Model

1. In either framework, you will need a list of years, from 1900 to 2100 in steps of 1 year.

2. Business-as-usual CO_2 is increasing exponentially, following a function of the form

pCO_2 (time 2) = 280 + (pCO_2 (time 1)-280) * (1 + A * D_time)

where D_time is the time step, time_1 and time_2 are time points (start from a known pCO2 at time_1 and calculate what it will be at time_2). A is a tunable growth rate parameter equal to a value of 0.0225 / year, gives an atmospheric rise rate of 2.5 ppm when the pCO_2 value is 400 ppm; a reasonable fit to the current pCO_2 value and rate of rise.

3. For each pCO_2 level, calculate the radiative forcing using the formula

RF from CO_2 = 4 [Watts/m2] * ln(pCO_2 / 280) / ln (2)

where the last factor, ln(pCO2/280) / ln(2), gives the number of doublings of the CO2 concentration over an initial value of 280.

4. We need to account for the radiative impact of short-lived industrial stuff, primarily the sum of cooling from sulfate aerosols and warming from short-lived greenhouse gases like methane. But the difficulty is that we don't know very well what that number is. Assume that the intensity of the masking (how strong it is in Watts / m^2) depends on the rate of industrial activity, which we can estimate from the rate of pCO_2 rise. In other words, use a function of the form

RF masking scaled = B * (pCO_2 (year 2015) – pCO_2 (year 2014))/(1 year)

where you can get the value of B by fitting to a radiative forcing estimate for the present day. In other words, to get a masking effect of –0.75 Watts / m2, when the pCO_2 rise is 2.5 ppm / year, B would have to be –0.3. Solve for B in the code based on the assumption that the present-day masking RF is -0.75 Watts/m2. Later you'll change that assumption, so the code should know how to calculate B on its own.

The model looks more reasonable in the future if we assume that this function only works for the past, but that further increases in CO2 emissions don't lead to still further increases in aerosols, as in

RF masking = max(RF masking scaled, RF masking in 2015)

It is easy for coal plants to clean up their sulfur, which leads to the aerosols (and lots of air quality problems), without cutting the carbon emissions.

5. Compute the total radiative forcing as the sum from CO_2 and from masking.

6. Compute the equilibrium temperature change that you would get from that RF, using a climate sensitivity Delta_T_2x of 3 degrees per doubling of CO2, but to get that number into units of Watts/m2 rather than CO2 doublings (so it can deal with the effect of aerosols), use the fact that doubling CO2 gives 4 Watts/m2 of forcing, so that

climate_sensitivity_Watts_m2 = climate_sensitivity_2x / 4 [Watts/m2 per doubling CO2]

7. Compute the evolution of temperature by relaxing toward equilibrium on a time scale of 100 years, in other words, taking

change in T per timestep = (T(equilibrium) – T) / t_response_time (20 years) * X years / timestep

That was business as usual. Now for The World Without Us.

8. Beginning in the year when pCO_2 hits 400, make a new column or list with pCO_2 values that decrease rather than increase, following a somewhat fudged form

change in CO_2 per timestep = (340 - CO_2) * (0.01 * timestep)

where the CO_2 concentration is now relaxing toward a higher concentration than the initial, due to the long-term change in ocean chemistry (acidification). The time scale for the CO_2 invasion into the ocean is 100 years (1 / 0.01), which is a composite of fast equilibration times with the surface ocean and the land biosphere, and slower with the deep ocean.

9. Compute the RF from CO_2 from this as you did for the last pCO_2 column.

10. Compute the equilibrium temperature from the RF from CO_2. There is no masking RF for this stage, because that all goes away quickly.

11. Compute the time-evolving temperature from this period from the equilibrium temperature as you did before.

From <https://www.coursera.org/learn/global-warming-model/supplement/oFgve/description-of-the-model-formulation>

Tips for Encoding

In Python, begin by importing some required libraries

```
import math
import matplotlib.pyplot
import numpy
```
where the new module (to us) called math will be needed for the exponential function.

Initialize variables for the time step, an equilibrium CO_2 valued 280, an initial CO_2 for our simulation valued at 290, exponential growth (0.0225) and drawdown (0.01) rates. A variable holds the aerosol radiative forcing today, which you should initialize as -0.5 (Watts/m2). We will use two forms of the climate sensitivity, one an equilibrium temperature change for doubling CO2 (which is a radiative forcing of 4 Watts/m2), and one that is the temperature change per Watts/m2 of forcing (which therefore equals the 2X climate sensitivity / 4 Watts/m2). Use a value of 3 into a variable for the doubled CO2 climate sensitivity, and calculate what that means for the Watts/m2 version of the climate sensitivity.

Create lists for years, and, for both cases (business-as-usual and world without us) create lists for atmospheric CO_2, the radiative forcing from that, the total radiative forcing, the equilibrium temperature, and the time-evolving temperature. For business-as-usual there will also need to be a radiative forcing from masking, and it is handy to create a list of the rate of change of the CO_2 concentration. The CO_2 values should be initialized to the initial CO_2 (290), and the first entry in the list called years should be 1900. The rest of the lists are initialized with 0's.

The first loop should create the years list, from 1900 to 2100 in steps of 1 year. It should also calculate lists of the business-as-usual CO_2 concentration, the radiative forcing from the CO2, as

$$1$$

```
rfCO2 = climateSensitivity2x * math.log( bauCO2 / eqCO2 ) / math.log( 2 )
```
Also calculate the list of the rate of CO_2 rise, in ppm / timestep, which will be used to scale the masking radiative forcing.

After the first loop is done, you need to calculate the value of B for whatever value of the present-day masking you choose. You can use the python array.index(2015) function to get the index of the present day within the years array, which will also be the index for the other lists from the first loop. Verify that the atmospheric CO_2 concentration is close to 400 for the present day, and extract the incremental CO_2 change from the list of those that you kept from the first loop. Calculate B from this, such that B times the present-day rate of CO_2 rise gives your radiative forcing value.

Create a second loop that completes the business-as-usual scenario by running through all the years. In it calculate the radiative forcing from masking (by multiplying the incremental CO2 change in any given timestep by the coefficient B you just found). Calculate the total radiative forcing as equal to that from CO2 and that from the aerosol masking. Then calculate the equilibrium temperature that would arise from that radiative forcing, if the forcing were held constant for a very long time. This is simply the radiative forcing times the Watts/m2 version of the climate sensitivity you calculated above. So that's the equilibrium temperature, but recall from the last coding exercise you did that the temperature doesn't equilibrate instantly, because of the heat capacity of the oceans mostly. So calculate a transient temperature which relaxes toward the equilibrium temperature each time step according to

$$1$$

```
change in T_transient = ( T_eq - T_transient ) * timestep_years / t_response_time
```

where a t_response_time equilibration time scale of 20 years is kind of a fudge combination of faster equilibration of the surface ocean, and slower equilibration with the deep ocean.

In the World Without Us stage, start out by copying the values from the Business-as-usual lists of variables that you might want to plot into the World Without Us lists, up to the point where the paths diverge (when atmospheric CO2 is close to 400 ppm, our present). These lists are the CO2 concentrations, the radiative forcing from CO2, and the transient and equilibrium temperatures.

In a final loop, calculate the slow rampdown of atmospheric CO2 as described in the model formulation, and the radiative forcing from that. Masking is short-term, so when human activity ceases it will go away immediately, so the total radiative forcing in this stage is just that from the CO2. The temperature change at the start of the World Without Us stage should come from the value from the Business-as-Usual temperature evolution; it doesn't magically reset at 0.

From <https://www.coursera.org/learn/global-warming-model/supplement/4sKUZ/tips-for-encoding>

Week 5 Quiz

Monday, February 13, 2023
12:52 AM

- **Time Step Through the Past and Future**
- **Video:** VideoHow the Model Works
  . Duration: 5 minutes5 min
- **Reading:** ReadingDescription of the Model Formulation
  . Duration: 10 minutes10 min
- **Reading:** ReadingTips for Solving in a Spreadsheet
  . Duration: 10 minutes10 min
- **Reading:** ReadingTips for Encoding
  . Duration: 10 minutes10 min

- **Programming Assignment:** Code Check
. Duration: 3 hours3h
- **Peer-graded Assignment:** Code Review
. Duration: 1 hour1h

  Review Your Peers: Code Review
- **Quiz:** Code Trick: Aerosol Masking and Our Future
4 questions
- **Survey on MOOC technologies**

Code Trick: Aerosol Masking and Our Future

Quiz30 minutes • 30 min

Submit your assignment

Due March 19, 11:59 PM ISTMar 19, 11:59 PM IST

Attempts 3 every 8 hours

Start assignment

Receive grade

To Pass 50% or higher

Your grade

-Not available

Like

Dislike

Report an issue

Back

Code Trick: Aerosol Masking and Our Future

Graded Quiz. • 30 min. • 4 total points available.4 total points

DueMar 19, 11:59 PM IST

1.

Question 1

Set up the code to make a plot of the transient temperature evolution for both cases, business-as-usual and the world without us. Have the code also print out the value of the transient temperature in the year 2015. The actual temperature in 2015 was about 0.8 degrees warmer than natural due to global warming. Assume that the radiative forcing from aerosols in 2015 was -0.75 Watts/m2. Tune the value of the climate sensitivity (delta T 2x, the equilibrium warming you'd get from doubling CO2, a radiative forcing of 4 W/m2) until you reproduce (within 0.05 degrees) the warming in 2015. (When you change delta_T_2x, the value of the climate sensitivity in Watts/m2 will also change). What value of the climate sensitivity is required?

Answer: 3

3 0.8087025787953348

From <http://localhost:8888/notebooks/Downloads/Modeling/modeling.ipynb>

1 point

2.

Question 2
In the decade immediately following 2015, which scenario predicts the warmer climate?

1 point

business-as-usual

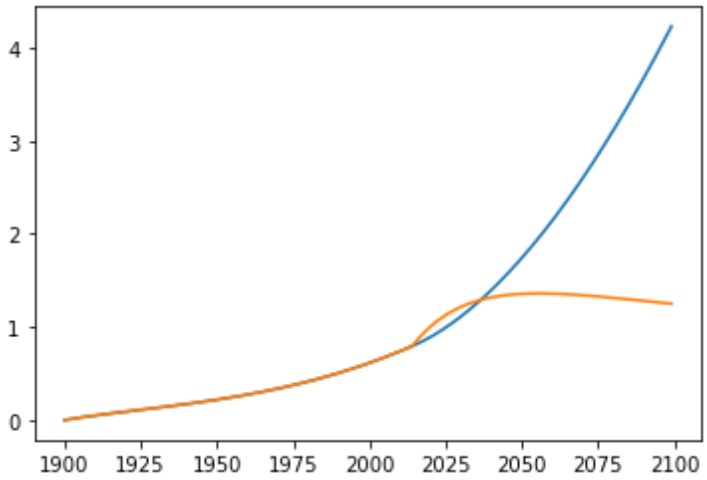
world without us (Answer)

World without us remains warmer until 2036 and then only starts to cool down due to masking effect (even though we are not contributing to increase in CO2!) So goodness of climate change takes almost 20 years to achieve even after we stop contributing to it! So better act now!

(year, temp_bau, temp_without_us)

[(2014, 0.7944396914164049, 0.7944396914164049), (**2015, 0.8087025787953348, 0.8349650614551064**), (2016, 0.8237842909016193, 0.8731134340212119), (2017, 0.8396672218728316, 0.9090066054424373), (2018, 0.856334830657493, 0.9427602621565586), (2019, 0.873771584081367, 0.9744842862640605), (2020, 0.8919629026132462, 1.0042830458043666), (2021, 0.9108951086966296, 1.032255670519449), (2022, 0.9305553775206024, 1.058496313830420 4), (2023, 0.9509316901098004, 1.0830944017164341), (2024, 0.9720127886195875, 1.1061348691507544), (2025, 0.9937881337285043, 1.1276983847161086), (2026, 1.0162478640256818, 1.1478615639903367), (2027, 1.039382757296269, 1.1666971722637944), (2028, 1.0631841936130026, 1.1842743171219017), (2029, 1.0876441201468767, 1.2006586313995498), (2030, 1.112755017614454, 1.2159124469887554), (2031, 1.1385098682837074, 1.2300949599568698), (2032, 1.1649021254644094, 1.2432623874097948), (2033, 1.191925684413008, 1.2554681165129304), (2034, 1.2195748545856357, 1.266762846061944), (2035, 1.2478443331764288, 1.2771947209758452), (**2036, 1.2767291798816687, 1.2868094600662305**), (**2037, 1.3062247928334259, 1.295650477418864**), (2038, 1.3363268856493862, 1.3037589977069552), (2039, 1.3670314655483746, 1.311174165739527), (2040, 1.3983348124837862, 1.3179331505330958), (2041, 1.4302334592496753, 1.324071244180477), (2042, 1.4627241725166589, 1.3296219557768372), (2043, 1.4958039347570686, 1.3346171006501084), (2044, 1.5294699270209329, 1.3390868851305264)]

From <http://localhost:8888/notebooks/Downloads/Modeling/modeling.ipynb>



3.

Question 3

Now assume that the net "masking" radiative forcing is actually -1.5 Watts/m2, which is still within the range of uncertainty. Adjust the climate sensitivity (both expression of this in your code, per doubling CO2 and per Watt/m2) so that the transient temperature in 2015 is within 0.05 degrees of 0.8 degrees. What value of the climate sensitivity, delta_T_2x for doubling CO2, is required to do this?

Answer: 5.5 (so sensitivity almost doubles if masking is doubled!)

5.5 0.7555667232421225

From <http://localhost:8888/notebooks/Downloads/Modeling/modeling.ipynb>

1 point

4.

Question 4

In your version of the code from the last problem (with aerosol cooling at -1.5 Watts/m2 today and a climate sensitivity that leads to warming in 2015 of 0.75-0.85 degrees C), change the output to print out a list of the difference between the transient temperatures between business-as-usual and the world without us scenarios, by year. You should observe that the world without us would actually be warmer than business-as-usual, because the short-term effect of masking is greater than the short term effect of decreasing CO2 emissions in this scenario. How long does this surprising behavior last, in years?

Answer: 37 years (2052-2015=37 , It takes until 2053 for temperatures to be cooler than business as usual!)

(year, temp_bau, temp_without_us)

[(2014, 0.8131945929197096, 0.8131945929197096), (2015, 0.8242546071732244, 0.9330295724927676), (2016, 0.8378255589066658, 1.046171345145851), (2017, 0.8538285719939189, 1.1529604641331304), (2018, 0.872189083150829, 1.2537204148989602), (2019, 0.8928366189067849, 1.3487584685846719), (2020, 0.9157045834333406, 1.4383664928624567), (2021, 0.940730056689765, 1.5228217222299356), (2022, 0.9678536023728851, 1.6023874897923265), (2023, 0.9970190851846971, 1.6773139224577798), (2024, 1.0281734969560166, 1.7478386013751748), (2025, 1.061266791188008, 1.8141871893522), (2026, 1.0962517255958133, 1.8765740269046567), (2027, 1.1330837122597655, 1.935202698505373), (2028, 1.171720675009871, 1.990266570522698), (2029, 1.2121229136884257, 2.0419493022640496), (2030, 1.2542529749538467, 2.090425331469213), (2031, 1.2980755293061064, 2.1358603355308565), (2032, 1.3435572540305758, 2.1784116696558504), (2033, 1.3906667217726851, 2.218228783120309), (2034, 1.4393742944706074, 2.2554536147136135), (2035, 1.4896520223872267, 2.2902209684119295), (2036, 1.5414735479959878, 2.322658870269688), (2037, 1.5948140144878697, 2.3528889074680936), (2038, 1.6496499786787397, 2.381026550412758), (2039, 1.7059593281077183, 2.407181458727959), (2040, 1.763721202127993, 2.4314577719526516), (2041, 1.8229159168017501, 2.4539543857030908), (2042, 1.8835248934205973, 2.474765214028704), (2043, 1.9455305904820528, 2.4939794386514955), (2044, 2.0089164389613856, 2.5116817457447675), (2045, 2.0736667807263447, 2.527952550874155), (2046, 2.1397668099501392, 2.5428682126928055), (2047, 2.207202517385425, 2.556501235952963), (2048, 2.2759606373690584, 2.568920464368093), (2049, 2.3460285974340183, 2.580191263832979), (2050, 2.4173944704111414, 2.590375696483851), (2051, 2.490046928909277, 2.599532686056508), (**2052, 2.5639752020680464, 2.607718174977493**), (**2053, 2.639169034482702, 2.6149852736016284**), (2054, 2.7156186472055843, 2.6213844019885566)]

From <http://localhost:8888/notebooks/Downloads/Modeling/modeling.ipynb>

1 point

Coursera Honor Code Learn more

I, **Lekhraj Sharma**, understand that submitting work that isn't my own may result in permanent failure of this course or deactivation of my Coursera account.

SubmitSave draft

Like
Dislike
Report an issue

From <https://www.coursera.org/learn/global-warming-model/exam/GQC5B/code-trick-aerosol-masking-and-our-future/attempt>