homework 10, version 1

Submission by: ***Neham Soni*** (neham@mit.edu)

Homework 10: *Climate modeling II*

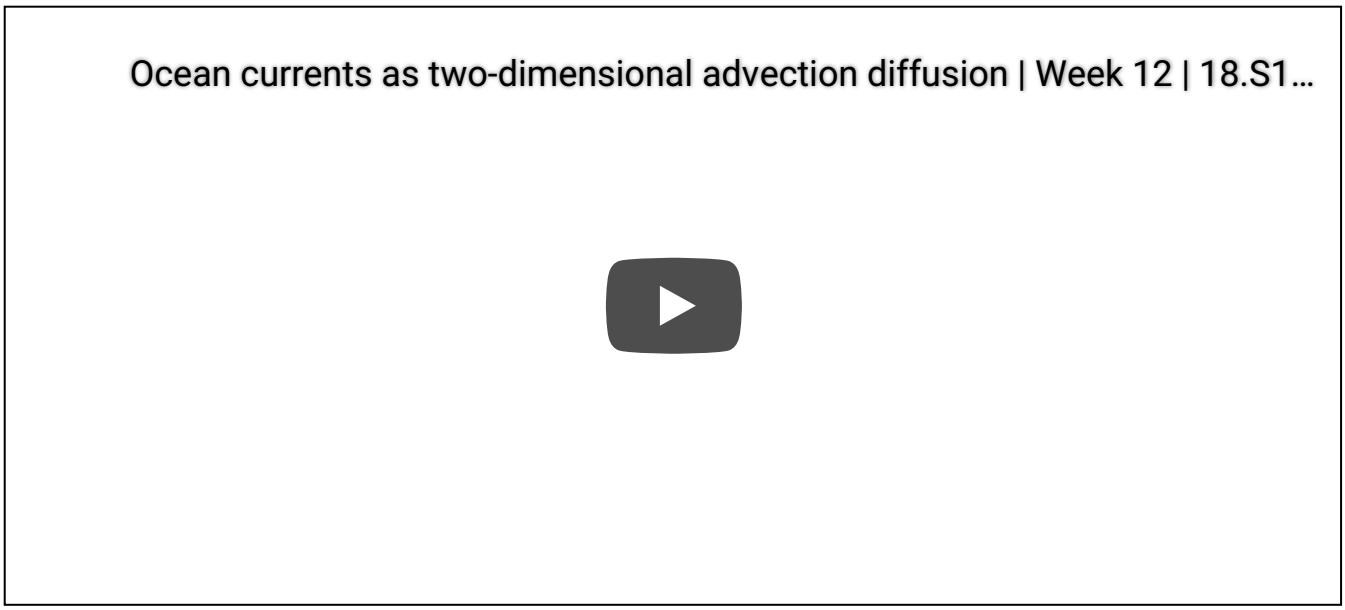
`18.S191`, fall 2020

```
student = (name = "Neham Soni", kerberos_id = "neham")
```

```
# edit the code below to set your name and kerberos ID (i.e. email without @mit.edu)

student = (name = "Neham Soni", kerberos_id = "neham")

# you might need to wait until all other cells in this notebook have completed running.
# scroll around the page to see what's up
```

Let's create a package environment:

```
begin
    import Pkg
    Pkg.activate(mktempdir())
    Pkg.add([
        "Plots",
        "PlutoUI",
        "Images",
        "FileIO",
        "ImageMagick",
        "ImageIO",
        "OffsetArrays",
        "PaddedViews",
        "ThreadsX",
        "BenchmarkTools",
    ])
    using Statistics
    using Plots
    using PlutoUI
    using Images
    using OffsetArrays
    using PaddedViews
    using ThreadsX
    using BenchmarkTools
end
```



In Lecture 23 (video above), we looked at a 2D ocean model that included two physical processes: **advection** (*flow of heat*) and **diffusion** (*spreading of heat*). This homework includes the model from the lecture, and you will be able to experiment with it yourself!

The model is written in a way that it can be **extended with more physical processes**. In this homework we will add two more effects, introduced in the Energy Balance Model from our last homework: *absorbed* and *emitted* radiation.

Exercise 1 - *Advection-diffusion*

Included below is the two-dimensional advection-diffusion model from Lecture 23. To keep this homework concise, we have only included the code. To see the original notebook with more detailed comments, use the link below:

> Click **here** to download and run the Lecture 23 notebook in a new tab.

Advection & diffusion

Notice that both functions have a main method with the following signature:

`(::Array{Float64,2}, ::ClimateModel)` *maps to* `::Array{Float64,2}`.

As we will see later, `ClimateModel` contains the grid, the velocity vector field and the simulation parameters.

```
advect (generic function with 3 methods)
```

```
begin
    # main method:
    advect(T::Array{Float64,2}, O::ClimateModel) =
        advect(T, O.u, O.v, O.grid.Δy, O.grid.Δx)

    # helper methods:
```

```
    advect(T::Array{Float64,2}, u, v, Δy, Δx) = pad_zeros([
        advect(T, u, v, Δy, Δx, j, i)
        for j=2:size(T, 1)-1, i=2:size(T, 2)-1
    ])

    advect(T::Array{Float64,2}, u, v, Δy, Δx, j, i) = .-(
        u[j, i].*sum(xgrad_kernel[0, -1:1].*T[j, i-1:i+1])/(2Δx) .+
        v[j, i].*sum(ygrad_kernel[-1:1, 0].*T[j-1:j+1, i])/(2Δy)
    )
end
```

```
xgrad_kernel = OffsetArray(reshape([-1., 0, 1.], 1, 3),  0:0, -1:1);
```

```
ygrad_kernel = OffsetArray(reshape([-1., 0, 1.], 3, 1),  -1:1, 0:0);
```

```
diffuse (generic function with 3 methods)
```

```
begin
    # main method:
    diffuse(T::Array{Float64,2}, O::ClimateModel) =
        diffuse(T, O.params.κ, O.grid.Δy, O.grid.Δx)

    # helper methods:
    diffuse(T, κ, Δy, Δx) = pad_zeros([
        diffuse(T, κ, Δy, Δx, j, i) for j=2:size(T, 1)-1, i=2:size(T, 2)-1
    ])
    diffuse(T, κ, Δy, Δx, j, i) = κ.*(
        sum(xdiff_kernel[0, -1:1].*T[j, i-1:i+1])/(Δx^2) +
        sum(ydiff_kernel[-1:1, 0].*T[j-1:j+1, i])/(Δy^2)
    )
end
```

```
xdiff_kernel = OffsetArray(reshape([1., -2., 1.], 1, 3),  0:0, -1:1);
```

```
ydiff_kernel = OffsetArray(reshape([1., -2., 1.], 3, 1),  -1:1, 0:0);
```

```
Main.workspace3.pad_zeros
```

Data structures

Let's look at our first type, `Grid`. Notice that it only has one 'constructor function', which takes `N` (number of longitudinal grid points) and `L` (longitudinal size in meters) as arguments. The struct contains more fields, these are precomputed and stored for performance.

```
begin
    struct Grid
        N::Int64
        L::Float64

        Δx::Float64
        Δy::Float64

        x::Array{Float64, 2}
        y::Array{Float64, 2}

        Nx::Int64
        Ny::Int64

        # constructor function:
        function Grid(N, L)
            Δx = L/N # [m]
            Δy = L/N # [m]
```

```
            x = 0. -Δx/2.:Δx:L +Δx/2.
            x = reshape(x, (1, size(x,1)))
            y = -L -Δy/2.:Δy:L +Δy/2.
            y = reshape(y, (size(y,1), 1))

            Nx, Ny = size(x, 2), size(y, 1)

            return new(N, L, Δx, Δy, x, y, Nx, Ny)
        end
    end

    Base.zeros(G::Grid) = zeros(G.Ny, G.Nx)
end
```

```
Grid(5, 300000.0, 60000.0, 60000.0, 1×7 Array{Float64,2}:
                                    -30000.0  30000.0  90000.0  150000.0  210000.0
```

```
Grid(5,300.0e3)
```

Next, let's look at three types.

Two structs: `OceanModel` and `OceanModelParameters`, and an abstract type: `ClimateModel`.

OceanModelParameters

```
Base.@kwdef struct OceanModelParameters

    κ::Float64=1.e4
end
```

```
abstract type ClimateModel end
```

```
begin
    struct OceanModel <: ClimateModel
        grid::Grid
        params::OceanModelParameters

        u::Array{Float64, 2}
        v::Array{Float64, 2}
    end

    OceanModel(G::Grid, params::OceanModelParameters) =
        OceanModel(G, params, zeros(G), zeros(G))

    OceanModel(G::Grid) =
        OceanModel(G, OceanModelParameters(), zeros(G), zeros(G))
end;
```

true

```
OceanModel <: ClimateModel # it's a subtype!
```

Timestepping

The `OceanModel` struct contains a complete *description* of the model being simulated. The next struct, `ClimateModelSimulation`, contains a *model*, together with the simulation results. It is mutable: timestepping the model means modifying a `ClimateModelSimulation` object.

ClimateModelSimulation

```
begin
    mutable struct ClimateModelSimulation{ModelType<:ClimateModel}
        model::ModelType

        T::Array{Float64, 2}
        Δt::Float64

        iteration::Int64
    end

    ClimateModelSimulation(C::ModelType, T, Δt) where ModelType =
        ClimateModelSimulation{ModelType}(C, T, Δt, 0)
end
```

```
function timestep!(sim::ClimateModelSimulation{OceanModel})
    update_ghostcells!(sim.T)

    tendencies = advect(sim.T, sim.model) .+ diffuse(sim.T, sim.model)
    sim.T .+= sim.Δt*tendencies

    sim.iteration += 1
end;
```

update_ghostcells! (generic function with 1 method)

```
function update_ghostcells!(A::Array{Float64,2}; option="no-flux")
    # Atmp = @view A[:,:]
    if option=="no-flux"
        A[1, :] .= A[2, :]; A[end, :] .= A[end-1, :]
        A[:, 1] .= A[:, 2]; A[:, end] .= A[:, end-1]
    end
end
```

Exercise 1.1 - *Running the model*

In the next few cells, we set up a simulation. We have included an interactive visualisation of the simulation.

👉 Familiarize yourself with the simulation through interaction. Get a sense for each parameter by changing their values.

```
default_grid = Grid(10, 6000.0e3);
```

Uncomment (`Ctrl+/` or `Cmd+/`) one of the lines below to choose between the different velocity fields:

```
#ocean_velocities = zeros(default_grid), zeros(default_grid);
#ocean_velocities = PointVortex(default_grid, Ω=0.5);
ocean_velocities = DoubleGyre(default_grid);
```

Choose the initial temperature state:

```
# ocean_T_init = InitBox(default_grid; value=50);
ocean_T_init = InitBox(default_grid, value=50, xspan=true);
#ocean_T_init = linearT(default_grid);
```

We define our ocean simulation. Run this cell again to reset the simulation to the initial state.

```
ocean_sim = let
    P = OceanModelParameters(κ=κ_ex)

    u, v = ocean_velocities
    model = OceanModel(default_grid, P, u*2. ^U_ex, v*2. ^U_ex)

    Δt = 12*60*60*1
    ClimateModelSimulation(model, copy(ocean_T_init), Δt)
end;
```

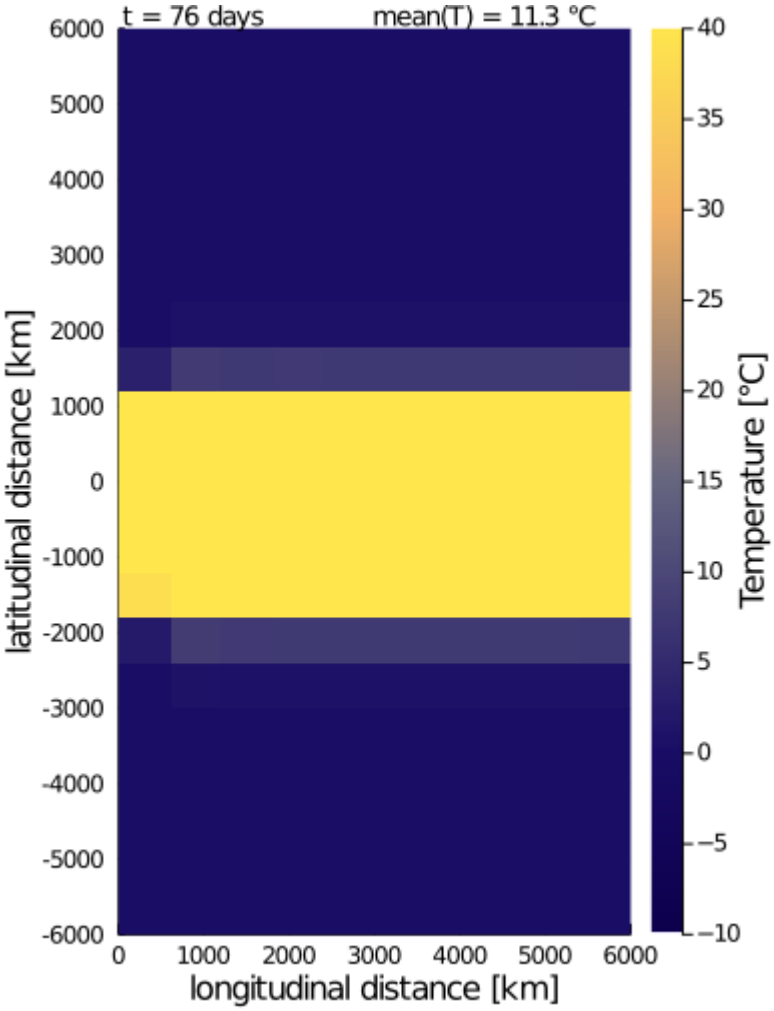
Simulation controls

Click to show the velocity field ☐ *or to show temperature* ***anomalies*** *instead of absolute values* ☐

Vary the current speed U: 1.0 [× reference]

Vary the diffusivity κ: 10000.0 [m²/s]

Start speed: 0.1 *secs / tick*




```
let
    go_ex
    if ocean_sim.iteration == 0
        timestep!(ocean_sim)
    else
        for i in 1:50
            timestep!(ocean_sim)
        end
    end
    plot_state(ocean_sim, clims=(-10, 40), show_quiver=show_quiver,
show_anomaly=show_anomaly, IC=ocean_T_init)
end
```

Velocity field for a single circular vortex

```
PointVortex (generic function with 1 method)
```

```
function PointVortex(G; Ω=1., a=0.2, x0=0.5, y0=0.)
    x = reshape(0. -G.Δx/(G.L):G.Δx/G.L:1. +G.Δx/(G.L), (1, G.Nx+1))
    y = reshape(-1. -G.Δy/(G.L):G.Δy/G.L:1. +G.Δy/(G.L), (G.Ny+1, 1))

    function ψ̂(x,y)
        r = sqrt.((y .-y0).^2 .+ (x .-x0).^2)

        stream = -Ω/4*r.^2
        stream[r .> a] = -Ω*a^2/4*(1. .+ 2*log.(r[r .> a]/a))

        return stream
    end

    u, v = diagnose_velocities(ψ̂(x, y), G)
    impose_no_flux!(u, v)

```

```
    return u,v
end
```

```
diagnose_velocities (generic function with 1 method)
```

```
impose_no_flux! (generic function with 1 method)
```

Quasi-realistic ocean velocity field $\vec{u} = (u, v)$

Our velocity field is given by an analytical solution to the classic wind-driven gyre problem, which is given by solving the fourth-order partial differential equation:

$$-\epsilon_M \hat{\nabla}^4 \hat{\Psi} + \frac{\partial \hat{\Psi}}{\partial \hat{x}} = \nabla \times \hat{\tau}\mathbf{z},$$

where the hats denote that all of the variables have been non-dimensionalized and all of their constant coefficients have been bundles into the single parameter $\epsilon_M \equiv \frac{\nu}{\beta L^3}$.

The solution makes use of an advanced *asymptotic method* (valid in the limit that $\epsilon \ll 1$) known as *boundary layer analysis* (see MIT course 18.305 to learn more).

```
DoubleGyre (generic function with 1 method)
```

Some simple initial temperature fields

```
constantT (generic function with 1 method)
```

```
linearT (generic function with 1 method)
```

```
InitBox (generic function with 1 method)
```

```
plot_kernel (generic function with 1 method)
```

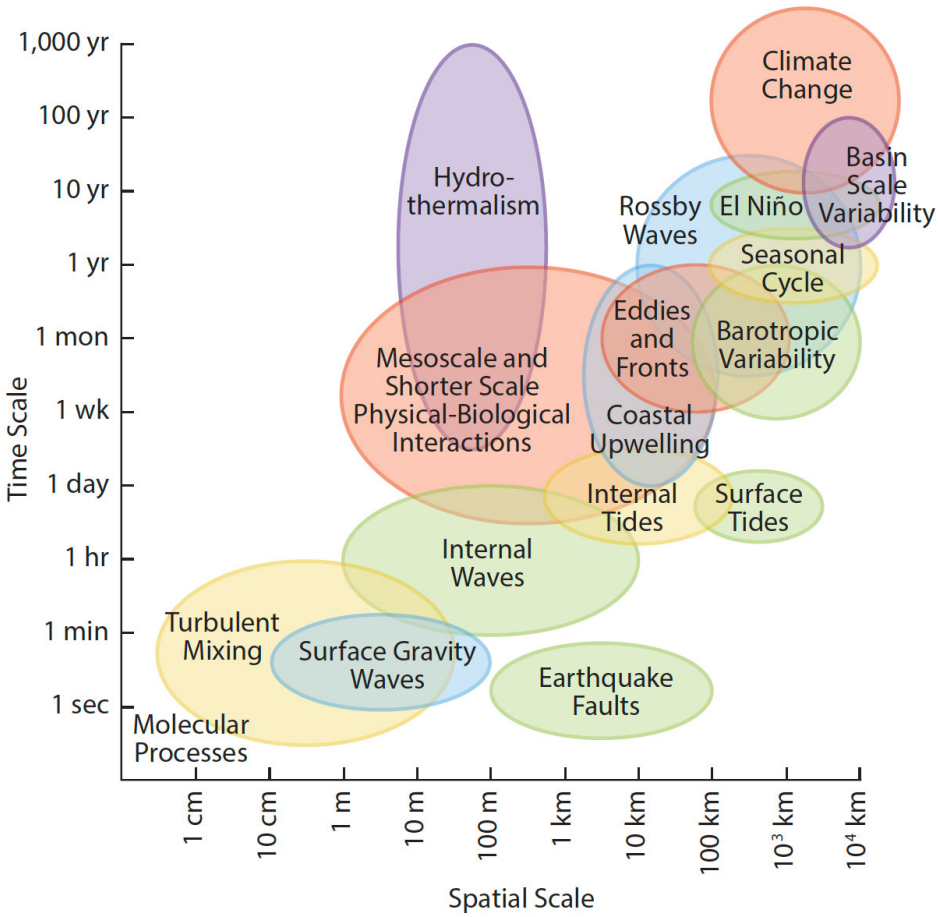
👉 Some parameters have a physical meaning (`κ`, `u` and `v`), while other parameters control our numerical process. Choose two of these *numerical* parameters, and describe their effect on the simulation's runtime and the simulation's accuracy.

`numerical_parameters_observation =`

Δt and N (length) of the grid are primarily one of the most significat numerical parameters which marvellously affects the reliability and duration of simulation. Albeit Δt can compensate inversely with u, v & κ values to provide same results but the major concern is to deal with boundary conditions which if left unrespected can relay unwanted results in the simulation. Even, initial conditions too play huge aspect of determining the conditions. As N, length of the grid increases, it reduce the discreetness of the simulation to be more continuous in nature, obviously tolling on CPU's processing power.

Exercise 2 - *Complexity*

In this class we have restricted ourself to small problems ($N_t < 100$ timesteps and $N_{x;\,y} < 30$ spatial grid-cells) so that they can be run interactively on an average computer. In state-of-the-art climate modelling however, the goal is to push the *numerical resolution* N to be as large as possible (the *grid spacing* Δt or Δx as small as possible), to resolve physical processes that improve the realism of the simulation (see below).



Here, we provide a simple estimate of the *computational complexity* of climate models, which reveals a substantial challenge to the improvement of climate models.

Our climate model algorithm can be summarized by the recursive formula:

$$T_{i,\,j}^{n+1} = T_{i,j}^{n} + \Delta t * (\text{tendencies})$$

for each time step $n \in \{1, \ldots, M\}$, and for each grid point $i \in \{1, \ldots, N_x\}, j \in \{1, \ldots, N_y\}$.

Our goal is to simulate an ocean of fixed size (e.g. 6000 km), for a fixed amount of time (e.g. 100 years). By choosing smaller Δt, Δx and Δy, we get a more accurate simulation, but for the same *size*, it requires more steps. i.e.

$$M = \mathcal{O}(\Delta t^{-1}) \qquad N_x = \mathcal{O}(\Delta x^{-1}) \qquad N_y = \mathcal{O}(\Delta y^{-1}).$$

Now, the *total runtime* of our simulation is proportional to the number of steps we need to take, which is $M \cdot N_x \cdot N_y$. For a fixed aspect ratio $N_y = 2N_x$, we get

$$\text{runtime} = \mathcal{O}(M)\mathcal{O}(2N_x^2) = \mathcal{O}(M)\mathcal{O}(N_x^2).$$

Exercise 2.1

For constant M, we want to verify that $\text{runtime} = \mathcal{O}(N_x^2)$ holds for our numerical model.

👉 Write a function `model_runtime` that takes `N` as an argument, and sets up a model with grid of resolution `N`, and returns the runtime of a single `timestep!`.

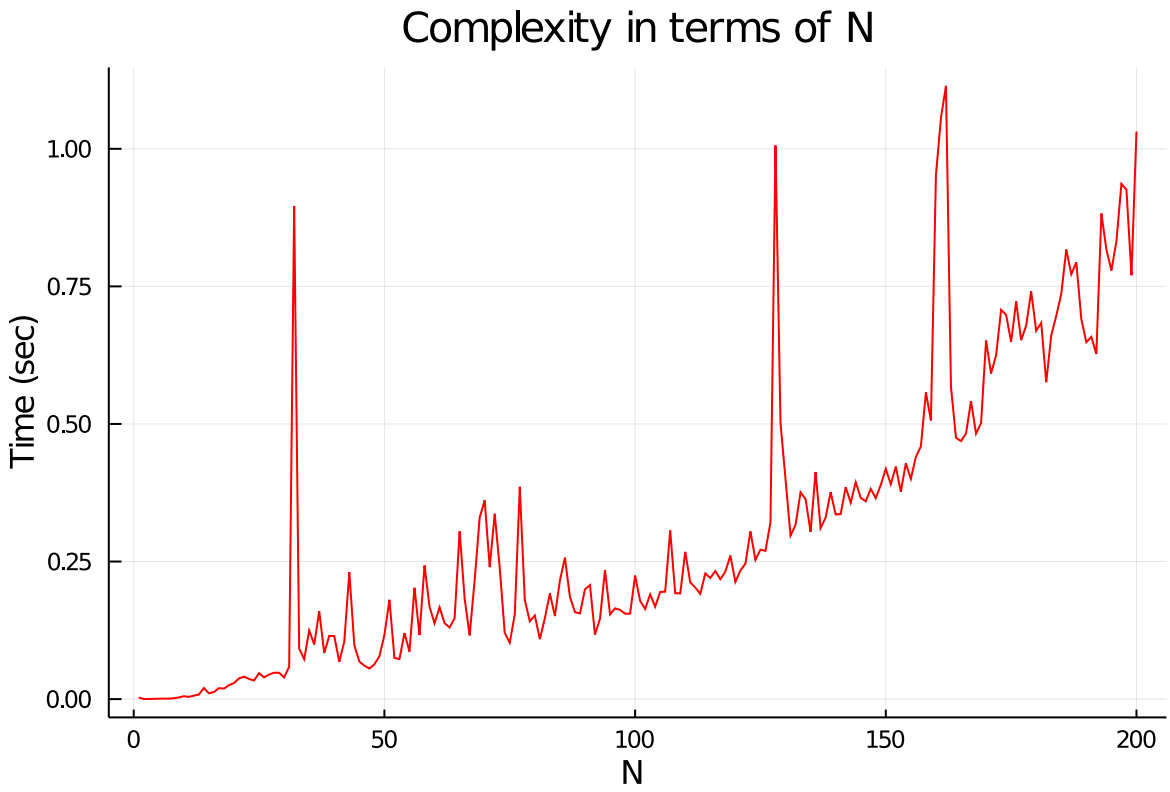
```
runtime (generic function with 1 method)
```

```
function runtime(N)
    G=Grid(N, 6000.0e3);
    u,v = zeros(G), zeros(G);
    P = OceanModelParameters(κ=1000);
    model = OceanModel(G, P, u*2. ^1, v*2. ^1);
    Δt = 12*60*60;


    return @elapsed timestep!(ClimateModelSimulation(model,linearT(G) , Δt))
end
```

Hint

[redacted]

👉 Call your `runtime` function on a range of values for `N`, and use a plot to demonstrate that the predicted runtime complexity holds.




```
let
    T=1:200
    plot(T,runtime.(T);xlabel="N ",ylabel="Time (sec)",title="Complexity in terms of N",legend=false,color=:red)
end
```

Exercise 2.2 - *The CFL condition on* Δt

In Exercise 1, look for the definition of `Δt`. It is currently set to `12*60*60` (12 hours).

👉 Double `Δt` and run the simulation again. You should see that it runs faster, great! Now, keep doubling `Δt` until you see something 'strange'. Describe what you see.

```
Δt_doubling_observations =
```

Multiplying `Δt` by 2^8 we see a strange grid of checks (white & yellow) with mean temp falling bellow 0 °C pretty soon.

```
Δt_doubling_observations = md"""
Multiplying `Δt` by 2^8 we see a strange grid of checks (white & yellow) with mean temp falling
bellow 0 °C pretty soon.


"""
```

What you experienced is a *numerical instability* of the discretization method in our simulation. This is not caused by floating point errors – it is a theoretical limitation of our method.

To ensure the stability of our finite-difference approximation for advection, heat should not be displaced more than one grid cell in a single timestep. Mathematically, we can ensure this by checking that the distance $L_{CFL} \equiv \max(|\vec{u}|)\Delta t$ is significantly less than the width $\Delta x = \Delta y$ of a single grid cell:

$$L_{CFL} \equiv \max(|\vec{u}|)\Delta t \ll \Delta x$$

or

$$\Delta t \ll \frac{\Delta x}{\max(|\vec{u}|)},$$

which is known as **the Courant-Freidrichs-Levy (CFL) condition**.

The exact meaning of $\ll$ here depends on the simulation at hand, but in our case, it means that there is at least an order of magnitude difference:

$$\Delta t < 0.1 \frac{\Delta x}{\max(|\vec{u}|)}.$$

Given below is a function `CFL_advection` that takes a `ClimateModel` and computes the CFL value: $\Delta x / \max(|\vec{u}|)$.

```
CFL_advection (generic function with 1 method)
```

```
function CFL_advection(model::ClimateModel)
    model.grid.Δx / maximum(sqrt.(model.u.^2 + model.v.^2))
end
```

```
(43200.0, 2.21181e6)
```

```
ocean_sim.Δt, 0.1 * CFL_advection(ocean_sim.model)
```

👉 Using the interactive simulation of Exercise 1, verify that the CFL condition is (somewhat) true. Increase the magnitude of $\vec{u}$ until you start to see numerical artefacts. Now, halve the value fo Δt, and again, increase the magnitude of $\vec{u}$. You should find that *halving Δt allows for twice the velocity magnitude*. The same link exists between Δt and Δx.

The CFL inequality states that if we want to decrease the grid spacing Δx (or increase the *resolution* N_x), we also have to decrease the timestep Δt by the same factor. In other words, the timestep can not be thought of as fixed – it depends on the spatial resolution: $\Delta t \equiv \Delta t_0 \Delta x$. This means that $M = \mathcal{O}(N_x)$.

Revisiting our complexity equation, we now have

$$\text{runtime} = \mathcal{O}(M)\mathcal{O}(N_x^2) = \mathcal{O}(N_x^3).$$

In other words, in a 2-D model, **2x the spatial resolution requires 8x the computational power**.

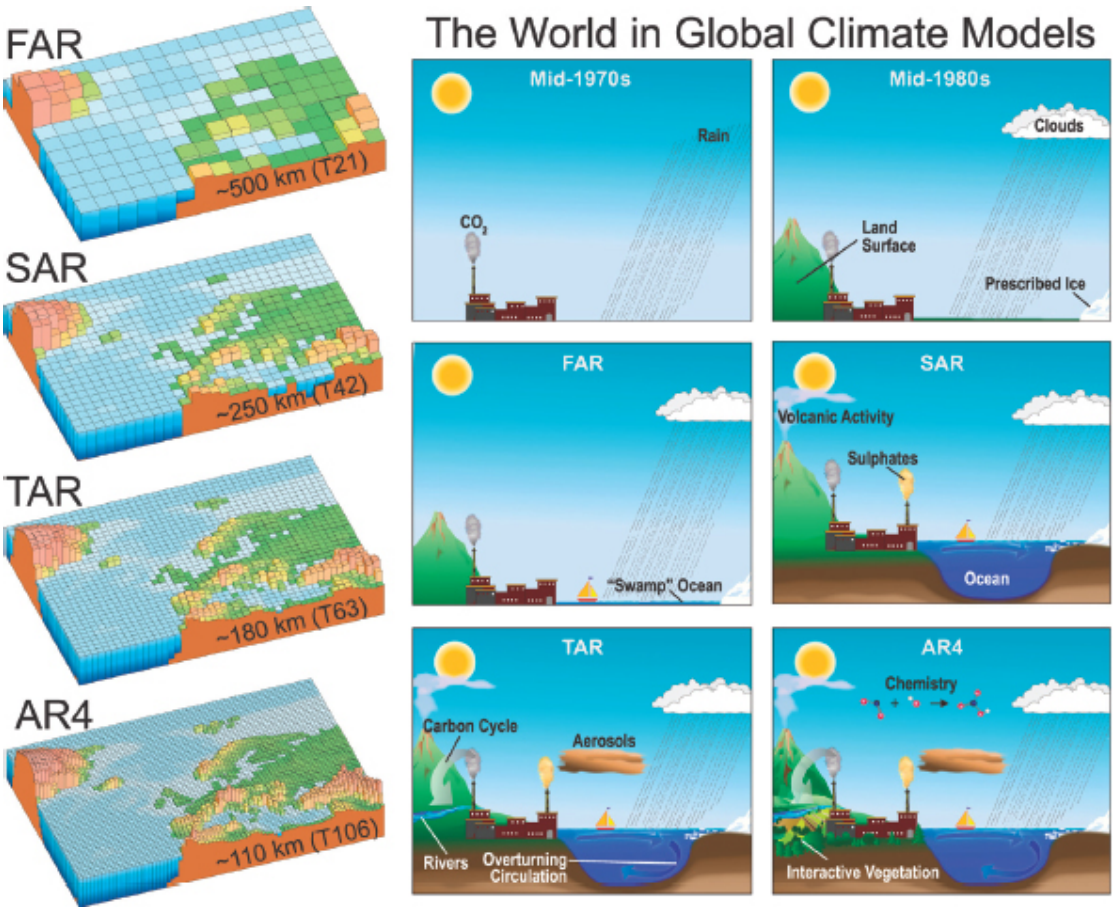
Exercise 2.3 - *Moore's Law*

In practice, state-of-the-art climate models are 3-D, not 2-D. It turns out that to preserve the aspect ratio of oceanic motions, the *vertical* grid resolution should also be increased $N_z \propto N_x$, such that in reality the computational complexity of climate models is:

$$\text{runtime} = \mathcal{O}(N_x^4).$$

This is the fundamental challenge of high-performance climate computing: to increase the resolution of the models by a factor of 2, the model's run-time increases by a factor of $2^4 = 16$.

The figure below shows how the grid spacing of state-of-the-art climate models has decreased from 500 km in 1990 (FAR) to 100 km in the 2010s (AR4). In other words, grid resolution increased by a factor of 5 in 20 years.



Moore's law is the observation that the number of transistors in a dense integrated circuit doubles about every two years. In the context of climate modelling, we can interpret this as meaning that the computational complexity allowed by our best high-performance computers $\mathcal{C}$ doubles every two years:

$$\mathcal{C}(t) = \mathcal{C}(2020) * 2^{(t-2020)/2}$$

Present-day simulations have a grid spacing of $\Delta x = 30$ km (or about $N_x = L/\Delta x \approx \frac{20000 \text{ km}}{30 \text{ km}} \approx 700$).

👉 By extrapolating Moore's law forward into the future, estimate how long it would take for Δx to reach to the 500 meter scale of clouds, one of the important climate processes ($N_x = L/\Delta x \approx 40000$).

```
cloud_resolution_possible_at = (2028, 2031)
```

```
cloud_resolution_possible_at = let

    floor(Int,2*log(2,500/30)+2020),floor(Int,2*log(2,40000/700)+2020)
end
```

Exercise 3 - *Radiation*

In Homework 9, we used a **zero-dimensional (0-D)** Energy Balance Model (EBM) to understand how Earth's average radiative imbalance results in temperature changes:

$$\begin{aligned} C\frac{\partial T}{\partial t} = & \frac{S(1-\alpha)}{4} && \text{(absorbed radiation)} \\ & -(A-BT) && \text{(outgoing radiation)} \end{aligned}$$

This week we will do the same, but now in **two-dimensions (2-D)**, where in addition to heat being added or removed from the system by radiation, heat can be *transported around the system* by oceanic **advection** and **diffusion** (see also Lectures 22 & 23 for 1-D and 2-D advection-diffusion). The governing equation for temperature $T(x,y,t)$ in our coupled climate model is:

$$\begin{aligned} \frac{\partial T}{\partial t} = & \; u(x,y)\frac{\partial T}{\partial x} + v(x,y)\frac{\partial T}{\partial y} && \text{(advection)} \\ & + \kappa\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) && \text{(diffusion)} \\ & + \frac{S(x,y)(1-\alpha(T))}{4C} && \text{(absorbed radiation)} \\ & - \frac{(A-BT)}{C}. && \text{(outgoing radiation)} \end{aligned}$$

```
md"""
## **Exercise 3** - _Radiation_


In Homework 9, we used a **zero-dimensional (0-D)** Energy Balance Model (EBM) to
understand how Earth's average radiative imbalance results in temperature changes:

$\begin{split}
C\frac{\partial T}{\partial t} =&\ \hphantom{+}
\frac{S(1 - \alpha)}{4}
&\qquad\text{(absorbed radiation)}
\\

&- (A - BT)
&\qquad\text{(outgoing radiation)}
\end{split}$

This week we will do the same, but now in **two-dimensions (2-D)**, where in
addition to heat being added or removed from the system by radiation, heat can be
*transported around the system* by oceanic **advection** and **diffusion** (see also
Lectures 22 & 23 for 1-D and 2-D advection-diffusion). The governing equation for
temperature $T(x,y,t)$ in our coupled climate model is:

$\begin{split}
\frac{\partial T}{\partial t} = &  \hphantom{+}
u(x,y) \frac{\partial T}{\partial x} + v(x,y) \frac{\partial T}{\partial y}
&\qquad\text{(advection)}
\\

& + \kappa \left( \frac{\partial^{2} T}{\partial x^{2}} + \frac{\partial^{2} T}
{\partial y^{2}} \right)
&\qquad\text{(diffusion)}
\\

& + \frac{S(x,y)(1 - \alpha(T))}{4C}
&\qquad\text{(absorbed radiation)}
```

```
\\

& - \frac{(A - BT)}{C}.
&\qquad\text{(outgoing radiation)}
\end{split}$
"""
```

Parameters

Below we define two new types: `RadiationOceanModel` and `RadiationOceanModelParameters`. Notice the similarities and differences with our original model. By making `RadiationOceanModel` also a subtype of `ClimateModel`, we will be able to re-use much of our original code.

```
begin
    struct RadiationOceanModel <: ClimateModel
        grid::Grid
        params::RadiationOceanModelParameters

        u::Array{Float64, 2}
        v::Array{Float64, 2}
    end

    RadiationOceanModel(G::Grid, P::RadiationOceanModelParameters, u, v) =
        RadiationOceanModel(G, P, u, v)
    RadiationOceanModel(G::Grid, P::RadiationOceanModelParameters) =
        RadiationOceanModel(G, P, zeros(G), zeros(G))
    RadiationOceanModel(G::Grid) =
        RadiationOceanModel(G, RadiationOceanModelParameters(), zeros(G), zeros(G))
end;
```

RadiationOceanModelParameters

```
Base.@kwdef struct RadiationOceanModelParameters
    κ::Float64=4.e4

    C::Float64=51.0 * 60*60*24*365.25 # converted from [W*year/m^2/K] to [J/m^2/K]

    A::Float64=210
    B::Float64=-1.3

    S_mean::Float64 = 1380
    α0::Float64=0.3
    αi::Float64=0.55
    ΔT::Float64=2.0
end
```

Notice that this struct has `Base.@kwdef` in front of its definition. This allows us to:

1. assign default values to the struct fields, directly inside the definition, and
2. automatically create an easier constructor function that takes *keyword arguments*:

```
RadiationOceanModelParameters(40000.0, 1.6094376e9, 210.0, -1.3, 1380.0, 0.3, 0.55,
```

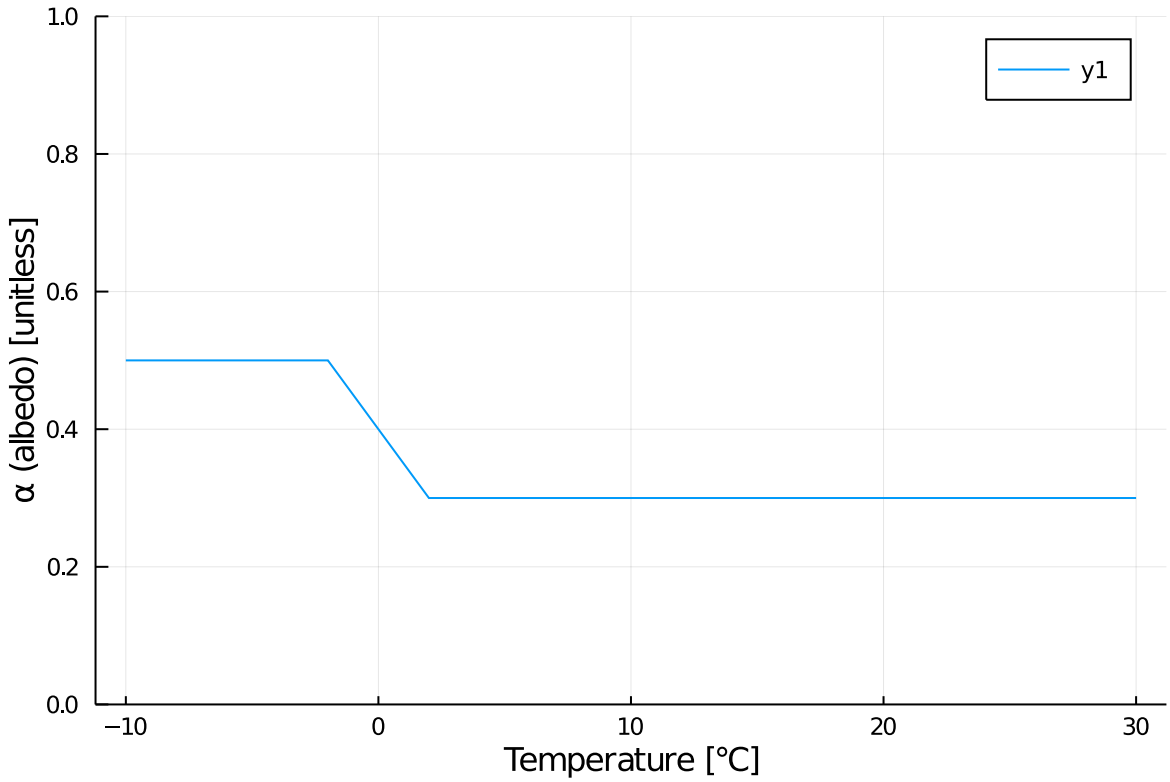
```
RadiationOceanModelParameters()
```

```
RadiationOceanModelParameters(40000.0, 1.6094376e9, 210.0, -1.3, 2000.0, 0.3, 0.55,
```

```
RadiationOceanModelParameters(S_mean=2000)
```

Exercise 3.1 - *Absorbed radiation*

The ocean absorbs solar radiation, increasing the temperature. Just like in our EBM model, we model the ocean as a surface with a *temperature-dependent albedo*: $\alpha(T)$. A high albedo means that more sunlight is reflected, and less is absorbed. We model the albedo function as:




```
α (generic function with 2 methods)
```

```
function α(T::Float64; α0, αi, ΔT)
    if T < -ΔT
        return αi
    elseif -ΔT <= T < ΔT
        return αi + (α0-αi)*(T+ΔT)/(2ΔT)
    elseif ΔT <= T
        return α0
    end
end
```

An area of ocean below 0°C is covered in ice, which is more reflective, and therefore absorbs less solar radiation. In our EBM model, this *positive feedback* leads to a bifurcation: under the same external conditions, the climate system has multiple equilibria.

In this week's two-dimensional model, the factor α is also two-dimensional: instead of a global albedo, every grid cell has its own temperature, which determines its own albedo, $\alpha(T(x, y, t))$. We can now have an ocean with warm and cold regions, which absorb different amounts of radiation. The same positive feedback can have a *local* effect.

Here is a second method for `α` that takes a 2D array `T` with the current ocean temperatures and a `RadiationOceanModel`, and returns the 2D array of albedos. We use the **dot operator** to apply `α` pointwise to `T`, also called *broadcasting*.

```
α (generic function with 2 methods)
```

```
function α(T::Array{Float64,2}, model::RadiationOceanModel)
    α.(T; α0=model.params.α0, αi=model.params.αi, ΔT=model.params.ΔT)
```

```
end
```

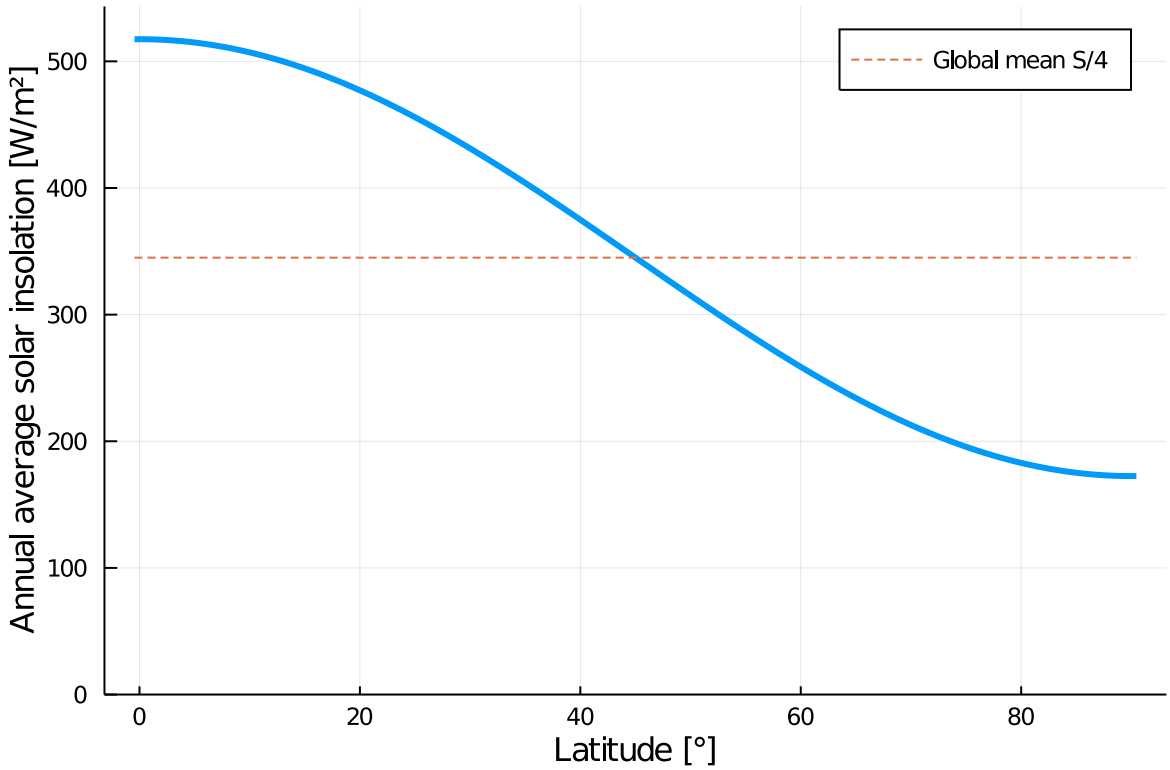
Solar insolation

Our rectangular grid represents the North Atlantic Ocean, stretching from the equator (latitude = 0°) to the North Pole (latitude = 90°) in the latitudinal direction (`y`), and from the east coast of North America to the west coast of Europe in the longitudinal direction (`x`). In reality, climate models have to explicitly deal with the curvature of the Earth when constructing their model grids. Here, we will just treat the North Atlantic Ocean as a rectangle with roughly the correct dimensions.

Just like the albedo, every grid cell will have a local amount of solar insolation. In our model, we use the **annual average** at the latitude of a grid cell $S(y)/4$. This is given by: (***source***)

```
S_at (generic function with 1 method)
```

```
function S_at(y::Float64; grid::Grid, S_mean::Float64)
    S_mean .* (1+0.5*cos(2 * y_to_lat(y; grid=grid)))
end
```



Note that latitude is the horizontal axis in this graph, not the vertical.

```
y_to_lat (generic function with 1 method)
```

👉 Write a method `absorbed_solar_radiation` that takes a 2D array `T` with the current ocean temperatures and a `RadiationOceanModel`, and returns the tendencies corresponding to absorbed radiation. This is the analogue of `advect` and `diffuse`.

```
absorbed_solar_radiation (generic function with 1 method)
```

```
function absorbed_solar_radiation(T::Array{Float64,2}, model::RadiationOceanModel)
    y=model.grid.y[:]
    S=repeat(S_at.(y;grid=model.grid,S_mean=model.params.S_mean),1,model.grid.Nx)
    final = S .* (1 .- α(T,model))
```

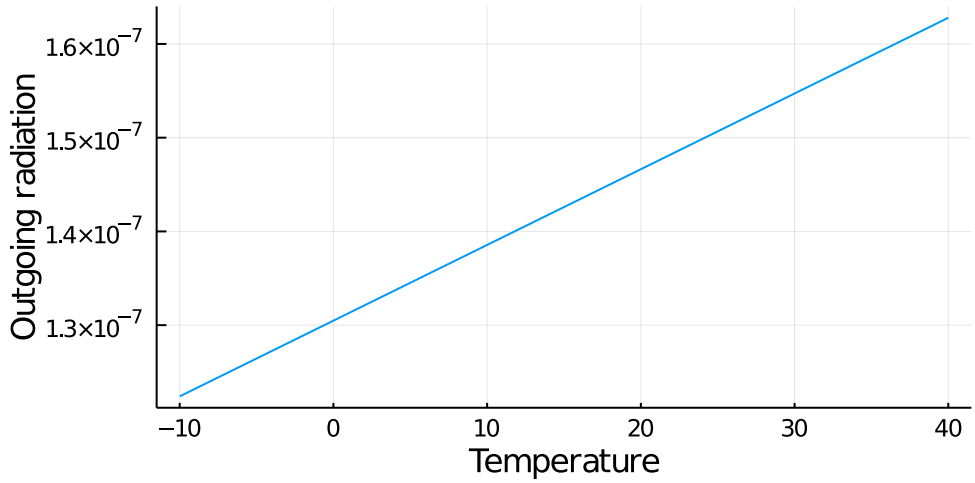
```
    p=4*model.params.C
    return final ./ p
end
```

Exercise 3.2 - *Outgoing radiation*

Just like in our EBM from before, when our ocean heats up by absorbing solar radiation, it also *emits radiation back to space*. This **outgoing thermal radiation** is what allows the ocean to eventually come to an equilibrium. The difference here is that each individual grid cell of our model radiatives according to it's local temperature $T_{i,j}$.

```
outgoing_thermal_radiation (generic function with 1 method)
```

```
function outgoing_thermal_radiation(T; C, A, B)
    (A .- B .* (T)) ./ C
end
```



👉 Write a method `outgoing_termal_radiation` that takes a 2D array `T` with the current ocean temperatures and a `RadiationOceanModel`, and returns the tendencies corresponding to outgoing radiation. This is the analogue of `advect` and `diffuse`.

```
outgoing_thermal_radiation (generic function with 2 methods)
```

```
function outgoing_thermal_radiation(T::Array{Float64,2}, model::RadiationOceanModel)

    return outgoing_thermal_radiation(T; C=model.params.C, A=model.params.A,
B=model.params.B)
end
```

Exercise 3.3 - *Running the model*

Let's define a new `timestep!` method for our new model. This is exactly the same as our advection-diffusion model, with the addition of the two radiation tendencies.

```
timestep! (generic function with 2 methods)
```

```
function timestep!(sim::ClimateModelSimulation{RadiationOceanModel})
    update_ghostcells!(sim.T)
    tendencies =
        advect(sim.T, sim.model) .+
        diffuse(sim.T, sim.model) .+
        absorbed_solar_radiation(sim.T, sim.model) .-
        outgoing_thermal_radiation(sim.T, sim.model)

```

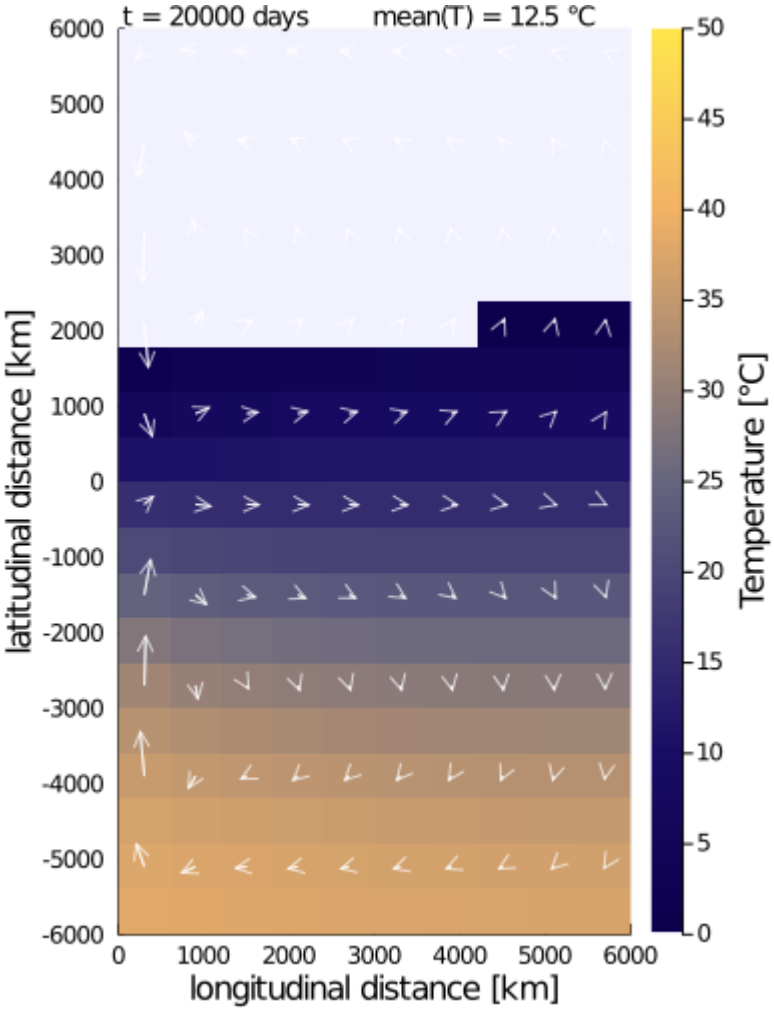
```
    sim.T .+= sim.Δt*tendencies

    sim.iteration += 1
end
```

We can now simulate our radiation ocean model, reusing much of the code from our advection-diffusion simulation.

```
radiation_sim =
▸ ClimateModelSimulation{RadiationOceanModel}(RadiationOceanModel(Grid(10,  6.0e6,  600000
```

```
radiation_sim = let
    grid = Grid(10, 6.e6)
    # you can specify non-default parameters like so:
    #params = RadiationOceanModelParameters(S_mean=1500, A=210, κ=2e4)
    params = RadiationOceanModelParameters()

    u, v = DoubleGyre(grid)

    T_init_value = 10
    T_init = constantT(grid; value=T_init_value)

    model = RadiationOceanModel(grid, params, u, v)
    Δt = 400*60*60

    ClimateModelSimulation(model, copy(T_init), Δt)
end
```

Start speed: 0.1 *secs / tick*




```
let
    go_radiation
    for i in 1:100
        timestep!(radiation_sim)
    end
    plot_state(radiation_sim; clims=(-0,50))
end
```

👉 Play around with the simulation to find the effect of each parameter. In particular, discover the effect of the solar insolation, `S_mean`, the initial temperatures, `T_init`, the liquid and ice albedos: `α0` and `αi`, and the amount of emitted heat at 0°C, `A`.

Exercise 3.4 - *Stable states*

So far, we are able to set up a model and run it interactively. You see that the model quickly goes from the initial temperatures to a *stable state*: a state with balanced energy (radiation out, radiation in). Changing the initial state slightly will probably result in the same stable state.

But let's see what happens when we initialize with extremely high or low initial temperatures...

👉 For $S = 1380$ (present-day value) and default parameters, does `T_init_value=-50` give a different result than `T_init_value=+50`? What about `T_init_value=+55`? By trying various values for `T_init_value`, **how many stable states do you find?**

`S_1380_stable_states` =
I found 3 stable states at -44°C, 6°C, 24°C to initial Temp. -50°C, 0°C, 50°C respectively.

👉 Answer the same question for $S = 1000$ and $S = 2000$.

`S_1000_stable_states =`
I found only one stable state at -75°C

`S_2000_stable_states =`
I found again only single stable stable Equilibrium Temp at 108°C

Exercise 3.5

That's right, we have found a *bifurcation*! Under the right conditions, there are multiple stable states. But under different conditions, there is only one stable state.

Hypothesis: The multiple stable states at $S = 1380$ are caused by the feedback effect of the ice albedo. A cold ocean reflects more heat and stays cold, a warm ocean absorbs more heat and stays warm.

👉 Find a way to confirm this hypothesis by changing the model parameters in the interactive model. Describe your findings.

`albedo_hypothesis_description =`
Indeed, it turns out if we change α function to always return 0 we will always get a single stable point even for different values of S_mean.

Hence albedo feedback effect bifuricates $T_{\text{equilibrium}}$

👉 Why is the number of stable states different for a lower or higher value of S?

`num_stable_states_answer =`

$$\frac{\partial T}{\partial t} = \quad T_{x,y}(t) + S_{mean}$$

where $T(t)$ is 5 degree polynomial function of t.

Hence, It more likely depends upon S_{mean} for $T_{x,y}(t)$ to have how many roots with negitive slope (stable points). Thereby, S_{mean} predominantly dictates the number of Equilibrium Temprature points.

Exercise 4 - *Bifurcation diagram*

So far, we are able to set up a model and run it interactively, and we discovered that by changing the initial value of `S_mean`, we find a different number of stable states.

In this final exercise, we will generate a visualization to help us understand the relationship between `S_mean` and the number of stable states. Instead of running a single model interactively, we write a function that takes the model parameters as input, runs the model until equilibrium, and returns the

final mean temperature. We will run this high-level function for various initial values, generating a single graph: the *bifurcation diagram*.

Exercise 4.1 - *Equilibrium temperature*

👉 Write a function `eq_T` that takes two argments, `S` and `T_init_value`, that sets up a radiation ocean model with `S` as `S_mean`, and with `T_init_value` as the constant initial temperature. Run the model until you have reached equilibrium (approximately), and return the average temperature.

```
eq_T (generic function with 1 method)
```

```
function eq_T(S, T_init_value)
    G=Grid(10,6.e6)
    params = RadiationOceanModelParameters(S_mean=S)
    u, v = DoubleGyre(G)
    T_init = constantT(G; value=T_init_value)
    model = RadiationOceanModel(G, params, u, v)
    Δt = 400*60*60
    simu = ClimateModelSimulation(model, copy(T_init), Δt)

    while true
        initial=simu.T
        for i in 1:10000
            timestep!(simu)
        end
        final=simu.T
        if norm(final-initial) ≈ 0.0
            return mean(final)
        end
    end
end
```

Hint

[redacted]

👉 Use the function `eq_T` to verify your answer to Exercise 3.4.

```
▸ Float64[-42.1159,  6.99741,  24.2314,  -42.1238]
```

```
eq_T.(1380,[-100,10,100,-1000])
```

Hint

Exercise 4.2

👉 Make a bifurcation diagram. For a couple of pairs `(S,T_init)`, calculate the equilibrium temperature. Create a scatter plot with S on the horizontal axis, and $T_{\text{equilibrium}}$ on the vertical.

> This calculation can take quite some time. Some tips:
>
> 1. Start out with a small amount of points.
> 2. Run the `eq_T` calculations in one cell, and store the result as a vector. Generate the scatter plot in a different cell.
> 3. You can replace a `map` with `ThreadsX.map` to run it in parallel across multiple CPU cores. More on this **below**.

```
md"""
#### Exercise 4.2

👉 Make a bifurcation diagram. For a couple of pairs `(S,T_init)`, calculate the equilibrium temperature. Create a scatter plot with ``S`` on the horizontal axis, and ``T_{\text{equilibrium}}`` on the vertical.

> This calculation can take quite some time. Some tips:
> 1. Start out with a small amount of points.
> 1. Run the `eq_T` calculations in one cell, and store the result as a vector. Generate the scatter plot in a different cell.
> 1. You can replace a `map` with `ThreadsX.map` to run it in parallel across multiple CPU cores. More on this [below](#threadsx).
"""
```

```
▸ Int64[-50, 0, 50]
```

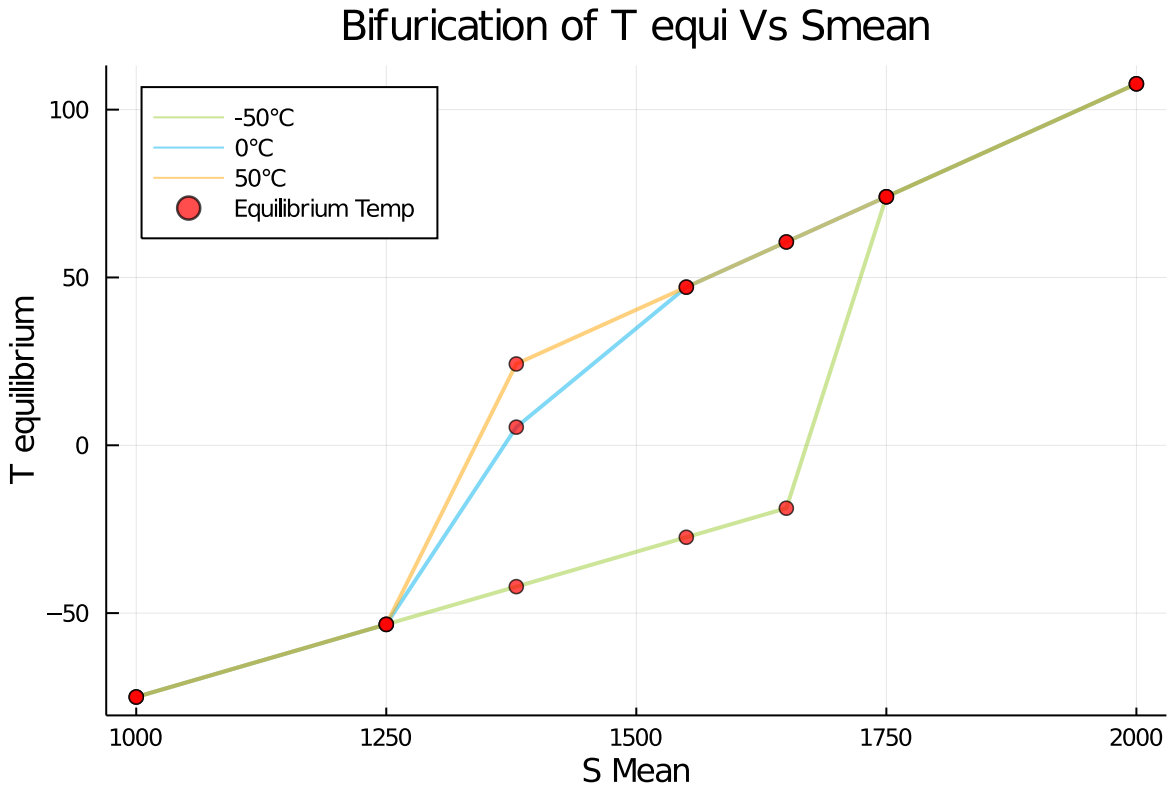
```
begin
    S_val= [1000, 1250, 1380, 1550, 1650, 1750, 2000];
    T_val= [-50, 0, 50];
end
```

```
bifurcation_ST =
▸ Tuple{Int64,Int64}[(1000, -50), (1000, 0), (1000, 50), (1250, -50), (1250, 0), (1
```

```
# you can change me!
bifurcation_ST = [(S,T) for S in S_val for T in T_val]
```

```
sults =
loat64[-74.9998, -74.9993, -74.998, -53.3654, -53.3644, -53.3514, -42.1155, 5.39897,
```

```
Results = ThreadsX.map(bifurcation_ST) do P
    eq_T(P[1],P[2])
end
```



Parallelize a `map`

You use `map` to apply a function to each element of a vector. Why not use a `for` loop? One nice property of `map` code is that you only describe the operation for each element, not the order to run the operations in. This means that it can be *easily parallelized* by the computer. For example, on a computer with 4 cores, the computation `map(sqrt, 1:100)` can be parallelized by handling `1:25` on the first core, `26:50` on the second, etc., at the same time.

In Julia, many functional primitives (`map`, `filter`, `sum`, `maximum`, `all`, and more) have an automatically multithreaded version, in the `ThreadsX.jl` package. As a demo, compare the two cells below. (You can see the runtime in the bottom right of a cell.) You might need to run the cells a second time, the first time includes Julia's compiler doing its thing.

```
▸ Int64[1, 4, 9, 16, 25, 36, 49, 64]
```

```
map(1:8) do i
    sleep(.1) # to simulate an expensive computation
    i ^ 2
end
```

```
▸ Int64[1, 4, 9, 16, 25, 36, 49, 64]
```

```
ThreadsX.map(1:8) do i
    sleep(.1) # to simulate an expensive computation
    i ^ 2
end
```

Exercise XX: *Lecture transcript*

(MIT students only)

Please see the link for the transcript document on **Canvas**. We want each of you to correct about 500 lines, but don't spend more than 20 minutes on it. See the the beginning of the document for more instructions.

👉 Please mention the name of the video(s) and the line ranges you edited:

```
lines_i_edited =
```

Abstraction, lines 1-219; Array Basics, lines 1-137; Course Intro, lines 1-144 (*for example*)

```
lines_i_edited = md"""
Abstraction, lines 1-219; Array Basics, lines 1-137; Course Intro, lines 1-144 (_for example_)
"""
```

Appendix

```
plot_state (generic function with 1 method)
```

```
ice_gradient =
```

Function library

Just some helper functions used in the notebook.

```
hint (generic function with 1 method)
```

```
almost (generic function with 1 method)
```

```
still_missing (generic function with 2 methods)
```

```
keep_working (generic function with 2 methods)
```

```
yays =
▸ Markdown.MD[Fantastic!, Splendid!, Great!, Yay ❤, Great! 🎉, Well done!, Keep it up
```

```
correct (generic function with 2 methods)
```

```
not_defined (generic function with 1 method)
```

```
todo (generic function with 1 method)
```