8.1 A zero-D Energy-balance model of the climate system

Box, or zero-D models need not involve the reservoir of a substance (e.g. trace metal, carbon, or nutrient concentrations) *per se* – the reservoir and fluxes of energy (heat) will do just fine. Which leads us to the climate system.

In this Section, you are going to create, and then use in a series of applications, a zero-D equilibrium global 'climate model' – the simplest representation of the energy-balance of the Earth's climate that it is possible to make. The model assumes that the climate system is in balance, with no net gain or loss of energy, and hence that the energy absorbed from incoming (short-wave) solar radiation equals the (long-wave) radiative loss from the Earth's surface (or top-of-the-atmosphere). The equations are outlined in the Box and you'll need to rearrange them in terms of T (mean global surface temperature).

The exercises that follow are structured and you need to pay attention to which **m-files** you are creating from scratch, which ones, having been created and coded up, you do not then further edit ...

8.1.1 In this first Subsection ('*The basic EBM*'), you'll create a script (# `scr_1`[1]) containing the Energy Balance Model (EBM), and test it.

(See Figure 8.1.)

8.1.2 Next, you'll turn your EBM script (`scr_1`) into a function (`fun_1`)[2] – passing in the solar constant and albedo as parameters, and returning the surface temperature. (And test it.)
(See Figure 8.2.)

8.1.3 In the Subsection '*Parameter sensitivity experiments using the EBM – #1*', you will create a new script (`scr_2`) with a single loop in it. Within the loop, you will make a call to the EBM function (# `fun_1`) that you created.[3]
(See Figure 8.3.)

8.1.3 Then, in an extension to the previous Subsection work, you will create another new script (`scr_3`), this time with a double (nested) loop in it. As before – within the loop, you will make a call to the EBM function. Note that there is going to something of a diversion in this Subsection that will illustrate nested loops for you.
(See Figure 8.6.)

8.1.4 In the penultimate Subsection ('Calculating the evolution of the solar constant'), you'll create a new function (`fun_2`), which will take time (counted from the formation of the Sun) in *Ga*, and return the value of the solar constant at that time ($S(t)$ (Wm^{-2})).
(See Figure 8.9.)

Energy balance modelling (1)

The surface energy budget at the Earths surface, to a zero-th order approximation, can be thought of as a simple balance between incoming, sort-wave radiation that is *absorbed*, and out-going, infra-red radiation.

On average (over the Earths surface and annually), the energy flux per unit area received from the sun, can be written:

$$F_{in} = \frac{\alpha \cdot S_0}{4}$$

(the $\frac{1}{4}$ appears because the cross-sectional area of the Earth is $\frac{1}{4}$ of its total surface area – i.e. you take energy intercepted by the Earth, which has an effective area of $\pi \cdot r^2$, and spread it out over the entire surface – an area of $4 \cdot \pi \cdot r^2$).

Albedo (α) varies hugely across surface types (and angle of incoming radiation). A commonly used mean global approximation is to set: $\alpha = 0.3$.

Net outgoing infrared radiation proceeds according to black body emissions:

$$F_{out} = \epsilon \cdot \sigma \cdot T^4$$

where ϵ is the emissivity, σ is the Stefan-Boltzmann constant (in units of Wm^{-2}), and T the temperature in Kelvin (K) ($273.15K == 0.0°C$).

For a perfect black body radiator, we would set ϵ=1.0. However, it turns out that the Earth is not a smooth and perfectly matt black sphere radiating directly from the surface to space ... there is an atmosphere and water surface over ~70% of its surface etc etc. A common modification is then to reduce the effective emissivity of the surface to less than 1.0. A value of 0.62 is given in *Henderson-Sellers* [2014], making the expression for the out-going flux:

$$F_{out} = 0.62 \cdot \sigma \cdot T^4$$

[1] This is not a suggested name of the **m-file**, but an ID to help you not get confused as to which script or function is being referred to in the text ...

[2] **Once the EBM function has been created, you do not at any point edit it any further!**

[3] DO NOT put code the loops into the EBM function – leave the function alone ...

And then ...

8.1.5 ... finally (Subsection '*Evolution* of Earth's surface temperature'), you'll create one last script (`scr_4`), with a loop in time in it, and from within this loop, you'll call first the solar constant function (`fun_2`), taking time as an input and returning the value of $S(t)$, which you will then pass into the EBM (# `fun_1`), returning T.
(See Figure 8.10.)

8.1.1 The basic EBM

To kick off – create a new script (**m-file**) ('`scr_1`' in the summary notation) and code up the analytical solution to the basic global mean energy budget at the surface of the Earth (see Box) in a program structure illustrated schematically in Figure 8.1.[4] The equations for in-coming and out-going radiation (energy) were given previously. You simply need to re-arrange these and write them as code. This will form the basis of subsequent, more complex (and later, time-stepping) models. You will need to find (from the Internet?) the values of the constants you need ... and will need to be careful with units of these.

For now – prescribe the value of S_0 – for which the modern value is 1368 Wm^{-2} as well as the value of surface albedo ($\alpha = 0.3$ by default) – somewhere near the start of the program. Then run it.

If you found a reasonable value for the solar constant, and did not screw-up the units on the Stefan-Boltzmann constant, then you should have an equilibrium (global, annual mean) surface temperature of around 14°C ... If not – debug. Assuming that the code ran without errors but gave a nutty answer:

1. Check that the units are correct.
2. Check that the equation has been re-arranged correctly – a common root of errors is incorrect placement of parentheses ... or not placing parentheses around multiple variables you are divining something all by.
3. If still 'no' – maybe take the 2 component equations (for F_{in} and F_{out}), plug S_0 into the equation for F_{in} and then play with different values of T to find a value for F_{out} that is approximately equal – is the value for T sane? If not, double-check the units and values in both component equations.
4. If still 'no' – WHAT HAVE YOU DONE?

Once it is working, have a quick play about, changing the value of S_0 and albedo (α) (saving the **m-file** each time and re-running) to get a vague feel for how sensitive the surface temperature is to these two parameters.

[4] Note that the code is relatively simple and does not involve (yet) loops or conditionals or anything like that. Although ... I am sure it will involve lots of nice juicy comments and sensible variable names(?)

Simply set up the values of the various constants and parameters you need at the start of the code, then solve for T at the end of the code. The structure (omitting `% comments`) of your code may look like:

```
% section for constants
(variables you do not expect
ever to change)
...
% section for parameters
(variables you might adjust)
...
% solve for T
T = ...
```

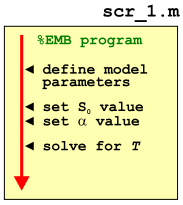

Figure 8.1: Form of the basic EBM model.

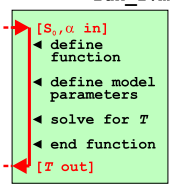

Figure 8.2: Form of the basic EBM model as a *function*.

8.1.2 *The EBM as a function*

We'll now make your model mode flexible so that it can be applied to the subsequent Examples. So – turn it into a *function*[5] that takes in 2 parameters – the solar constant (S_0) and the mean global albedo (α). The function should return the global mean surface temperature, T.[6] (See Figure 8.2)

Try playing with the function in the same way as before, but now passing the different values of S_0 and α (rather than having to edit the **m-file**, save, and re-run each time). To use the function (assuming you called it e.g. `fun_1`), and assuming the 2 passed parameters are in the order: S_0, α and are given their default values, you'd write (at the command line):

```
» fun_1(1368.0,0.3)
```

(and get a value close to 14°C returned).

8.1.3 *Parameter sensitivity experiments using the EBM – #1*

Now to utilize your new function ('`fun_1`' in the summary notation). Create a new blank script ('`scr_2`') and define 2 parameters near the start – one for the value of S_0 and one for α, then further down the code, call your function (`fun_1`), passing it these 2 parameters. So far so boring, as this is in effect what you had been doing in 'playing' with the function previously.

Common in numerical modelling is quantifying how sensitive a system is to the choice of parameter values – called a *sensitivity experiment*. You may already have gotten a feel for roughly how sensitive T was to changing S_0 on its own, or changing α on its own, but what about when both parameters vary together?

Lets start with a simple 1-D case, and consider just a change in the value of S_0. To automate generate different values of S_0 and call the function, you are going to need a loop[7]. There are two ways of constructing the loop[8]:

loop option #1 You could loop directly through the range of values of S_0 that you are interested in, e.g.

```
for S0 = 1000:100:1500
    % CODE GOES HERE
end
```

in which S_0 will go from 1000 to 1500 Wm^{-2} in steps of 100 Wm^{-2} [9].

Perhaps a little inconveniently, this does not pass through the modern value (1368 Wm^{-2}), although when you plot as a continuous line (e.g. in `plot`) or otherwise interpolate the results, maybe

[5] Refer to earlier in the text and also **help** on the required structure/syntax of a *function*. Recall the basic structure of a function **m-file**, has as its VERY FIRST LINE:

```
function [OUT] = ...
FUNCTION_NAME(IN)
```

where OUT represents one (or more) variables that are passed out (the 'result' of the function), `FUNCTION_NAME` is the name of your function, and IN is the name (or names, comma-separated) of one (or more) variables (parameter values) that are passed into the function. (The very last line of the function should have an `end`.)

For example, to pass in two variables, `IN_1` and `IN_2`, you'd have:

```
function [OUT] = ...
FUNCTION_NAME(IN_1,IN_2)
```

[6] Note that the parameters passed into, and returned by, the function, can be called anything you want. As long as they are useful (and clearly defined/explained in a comment somewhere).

[7] You are going to put the loop in the function (# `fun_1`), NOT the script (# `scr_2`).

An entire plane of Hell is reserved for anyone coding the loop in the function.

[8] In both cases a `for ...` loop.

[9] You can pick a different range and increment ... this is just a quasi-random example to illustrate ...

this does not matter. You could have addressed this by constructing a slightly less convenient form of the loop, e.g.:

```
for S0 = 1068:100:1568
    % CODE GOES HERE
end
```

which now passes exactly through the modern value of S_0.

loop option #2 Alternatively, you could have an integer count for the loop, and then derive a changing value of S_0 from this. For example:

```
S0_modern = 1368.0;
for m=-5:5
    S0 = S0_modern + 100*m
    % CODE GOES HERE
end
```

Look carefully through this code and follow what is going – as `m` counts from `-5` to `5` (in steps of `1`), `100` times the value of `m` is added to the modern value of S_0[10], meaning that S_0 ends up going from `S0_modern` - 500, to `S0_modern` + 500 Wm^{-2} (in steps of 100 Wm^{-2}).

[10] The variable definition `S0_modern = 1368.0` at the top of the code fragment.

Or, alternatively:

```
S0_modern = 1368.0;
for m=1:11
    S0 = S0_modern + 100*(n - 6)
    % CODE GOES HERE
end
```

which does exactly the same (do a mental check on this) but now counts `m` starting from a value of `1`.

So what does it matter, and/or is one 'better' than the other? Actually, both are equivalent and you could make either work out just fine. The advantage with the second version is that you implicitly have an integer counter. For the first version, you'd have to add lines, e.g.:

```
count = 0;
for S0 = 1068:100:1568
    count = count + 1;
    % CODE GOES HERE
end
```

And why might we want some sort of an integer counter in the first place? Well, you might want to save the data(!), i.e. the calculated (by your function) value of T vs. the inputted value of S_0.

There are also two ways of saving the data (assigning calculated values to sequential locations in an array):

save option #1 Create the necessary array(s) beforehand, e.g. using the `zeros` *function*. For instance, to create a vector with 11 rows (and 1 column), suitable for saving the value of T calculated by each call to the EBM function, you could write:

```
data_T = zeros(11,1);
```

which would create a (single) column vector with 11 rows. You'd need an equivalent vector (e.g. `data_S0` in this example) for storing the corresponding value of S_0 used in the temperature calculation. These vectors are created before the loop starts.
Then within the loop (and after the calculation of T), you'd assign your values of S_0 and T by using whichever index you created[11]:

```
data_S0(m)  = S_0;
data_T(m) = T;
```

or:

```
data_S0(count)  = S_0;
data_T(count) = T;
```

where `m` and `count` are integers, starting at a value of one, and incrementing by a value of one on each successive execution of the loop. `m` (or `count`) represents an index that allows you to store the result of each successive calculation (as well as the corresponding input value) in a vector.

save option #2 Or ... **MATLAB** will allow you to 'grow' a vector, one element at a time (but not for matrices).[12] The the code within the loop actually looks identical – you just omit the 2 lines at the start of the program creating vectors of appropriate size (and zero in value).

So pick one (i.e. a way of saving a pair of values each time around the loop) and code it up. (Or try both!) Then, at the end of your program, plot (`plot` or `scatter`) how T varies as a function of S_0.

The structure of your code should look like Figure 8.3. and your resulting figure (depending on the range you assume for S_0), something like Figure 8.4.

[11] i.e. which of the two OPTIONS you chose earlier.

[12] The vector automatically grows in length as you add values to it. If you don't believe me, try the following:

```
» A=1;
» A(2) = 2;
» A(3) = 3;
```

You could instead define at the start f the code (before the loop) a vector of zeros of the correct length, the 'correct length' being the number of time around the loop. See function `zeros`. Or even `NaNs` ...

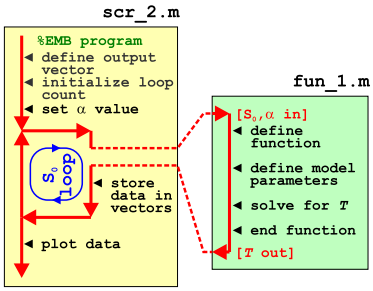

Figure 8.3: Schematic structure of the model configured to carry out a single parameter sensitivity study.

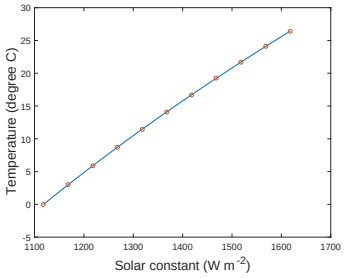

Figure 8.4: Sensitivity of global mean surface temperature vs. solar constant (mean surface albedo held constant at an albedo value of 0.3).

8.1.4 *Parameter sensitivity experiments using the EBM – #2*

In this Subsection, we'll extend the sensitivity experiment to 2D, assuming that you are interested in how T also varies as a function of α. So, you'll need to vary both S_0 and α, and in all combinations of the two. In fact, in a grid pattern, with S_0 increasing in steps on one axis (as before), and α on the other.

Hopefully, you might have guessed that you'll need a *nested loop*(?) – one loop going through all possible values of α, *for each and every possible value of* S_0??

Perhaps, as an aside, we'll go through a simpler example/system first.

A chess board consists of squares in a 8×8 grid. The squares alternate black and white. To define 8 squares (points) along the x-axis on the bottom row, you'd write something of the form:

```
for m=1:8
    % SOME CODE GOES HERE
end
```

Now, if you wanted to define 8 squares along each column (the y-axis), at each and every x-axis value, you'd need to loop through all the rows, So you need a loop in e.g. `n`, inside the loop for `m`:

```
for m=1:8
    for n=1:8
        % SOME CODE GOES HERE
    end
end
```

Follow this through to satisfy yourself that for each and every value of `m` from `1` to `8`, `n` loops from `1` to `8`, and hence visits every point in turn of a 8×8 (`n,m`) grid.

Actually, now we have got this far, it is good practice to consider how we'd define the black and white squares. We'll assume that black is represented by `'1'` (*true*) and white by `'0'` (*false*) and create a board (array) of all white squares to start with, i.e.

```
board = zeros(8);
```

(Refer to **help** or earlier for the syntax for help on the function `zeros`.[13])

If we start with a black square (`'1'`) at the bottom left, we could define an *algorithm* for creating the grid as: odd column number squares are black, as long as the row number is odd, otherwise they are white.[14] So to implement this in code – as we loop through both column (`m`) and row (`n`) on the board, we test for the column number being odd and row number odd, OR, the column number being even and row number being even. If *true*, the square is defined as black. The only tricky bit is to determine whether the row or column number is even or odd. We do this by testing whether there is any remainder after dividing by `2`, using the function `mod`.

The complete code looks like:

```
board = zeros(8);
for m=1:8
    for n=1:8
        if ((mod(m,2)>0 && mod(n,2)>0) || (mod(m,2)==0 && mod(n,2)==0))
            board(n,m) = 1;
        end
    end
end
```

[13] You could alternatively write this:

```
board = zeros(8,8);
```

`mod`

Not ... the opposite of `rocker` (which doesn't exist in **MATLAB** anyway) but short for *modulo*. Wikipedia helpfully tells us:

"*In computing, the modulo operation finds the remainder after division of one number by another (sometimes called modulus).*"

Or in MATLAB-speak:

```
b = mod(a,m)
```

"*returns the remainder after division of* `a` *by* `m`*, where* `a` *is the dividend and* `m` *is the divisor*".

It turns out that as long as a is positive, you can use to test for whether an integer `a` is *even* or *odd* by:

```
b = mod(a,2)
```

When the returned value `b` is `0`, `a` is *even*, and when `b` is `1`, `a` is *odd*.

[14] Look up a picture of a chess board to convince yourself that this works.

Spend a little time decoding the `if` statement for practice ... If you want to see that it works – code it in a new **m-file**, run it, and then plot up `board` by e.g. using `imagesc` (cf. Figure 8.5). Beautiful.

OK – that was easily the greatest diversion in pedagogical history, but nested loops should now come almost as second nature to you :o) So how about coding up the nested loop for the question we were meant to be addressing – carrying out a 2D sensitivity test of the parameters S_0 and α. See if you can create this.

Start with a new (script) **m-file** ('`scr_3`'). For constructing the loop – you have already seen the 1D example of parameter sensitivity code, and also an example of creating a nested loop for a 2D grid. Your chess board columns (`m`) become S_0, and rows (`n`) become α. You don't need to do anything so awful as that `if ...` statement – instead just call your function (`fun_1`) for solving the global surface temperature (passing it the values of S_0 and α generated in the loop). A schematic of the program structure is shown in Figure 8.6.

For saving the data (within the loop), you cannot not simply index the locations you want in a 2D array (matrix) that did not previously exist and expect it to 'grow' as before, becasue a matrix must have all complete rows and columns. Instead, near the start of the code (before the loop), create a matrix of the size of the parameter grid. For example, if you were going to loop through 10 different values of S_0 and 10 of α, you could write:

```
data_output = zeros(10);
```

(creating a 10×10 array of zeros). Or if for example, you had 20 different values of S_0, and 10 of α:

```
data_output = zeros(10,20);
```

(20 columns times 10 rows).

Within an (`n`,`m`) loop you then assign your calculated value of T to the appropriate location:

```
data_output(n,m) = T;
```

Don't forget that you'll also need to know the values of S_0 and α that correspond to the column and row numbers. Perhaps save these as 2 individual vector (as per before) or ignore them for now.

One slight complication if you use a pair of counters and increment their value each time around their respective loops (rather than having a integer count for the loop itself (i.e. `n` and `m`)) – the innermost counter must be reset in value each time the outer loops starts:

```
count_outer = 0;
for ...
    count_outer = count_outer + 1;
```

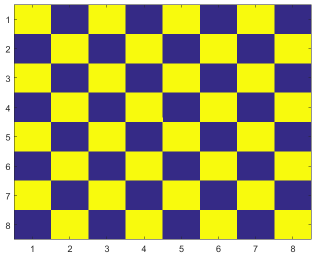

Figure 8.5: Chess board grid pattern.

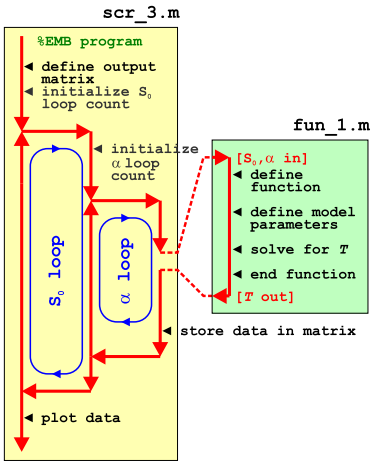

Figure 8.6: Schematic structure of the model configured to carry out a double (in terms of solar constant AND now albedo) parameter sensitivity study.

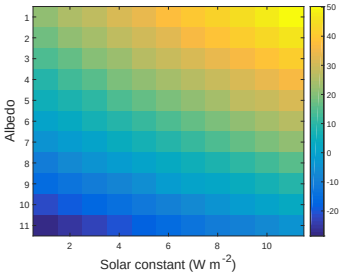

Figure 8.7: Global mean surface temperature (°C) as a function of solar constant and surface albedo grid point number.

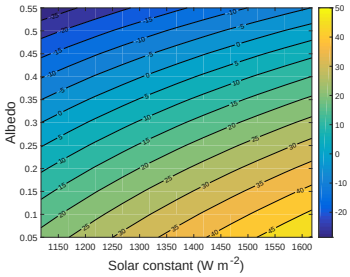

Figure 8.8: Global mean surface temperature (°C) as a function of the value of solar constant and surface albedo.

```
    count_inner = 0;
    for ...
        count_inner = count_inner + 1;
        % CODE GOES HERE
    end
end
```

(Try it instead by initializing both prior to the outer loop, and see what happens ...)

When you *think* you have this working and generating a matrix of T values[15], plot the resulting surface of T vs. the two parameters. Rather than using e.g. `imagesc` (Figure 8.8)[16], try `contour`[17] or `contourf` (e.g. Figure 8.7).

[15] HINT: create a 2D array of the appropriate size first, before the *loop* starts, using `zeros`, and then populate it with the values of T as the *loop* loops.

[16] Note that the temperature grid points are plotted as a function of column and row number and that the plots ends up 'up-side-down' compared to the `coutourf` version.

[17] You'll need to employ `meshgrid` based on the same 2 vectors of values that the loop creates for S_0 and α.

8.1.5 *Creating a function for the evolution of solar constant through geological time*

In this and the final Subsection, you are going to leave the 2D-ness aside and consider how Earths surface temperature has changed through geological time.

So far you only have a function equating solar constant (S) to temperature (T). What you need is some way of equating time (t) to the value of the solar constant at that time S_t (which you can then turn into temperature). We'll remedy this toot sweet.

Start by creating a new (blank) **m-file** and define it as a *function* that takes in time (in units of *Ga*) and spits out S_0 (Wm^{-2}) (this will be '`fun_2`' in the on-going notation).

The background to the equation that will go into your function is given in the **Solar constant Box**. In this, you'll first need to substitute the modern value of the solar constant into the equation to leave it in terms of S_t (the solar constant value at time t) rather than L_t. Your function, aside from the all-important 1st line (and `end` at the end) and appropriate `% comments`, need have little more in than a definition for any constant you might want to use, such as the modern value of S_0 and perhaps time now (4.57 Ga) ... and a single line for the equation giving the value of S_t. Be careful that in the equation, t is measured as the age of the Sun (since its formation), meaning that time 'now' (modern), is equivalent in the equation to $t = -4.57$ (Ga).

When you think you have done this – check it – plug in values of time into your function, i.e.

```
» fun_S(4.57)
```

for passing the time now into a *function* called '`fun_2`' in the on-going notation (which in this example should return a value of 1368 (Wm^{-2})).

Solar constant

The long-term evolution of solar luminosity L_t as a function of time t can be approximated [*Gough* [1981]; *Feulner* [2012]) by:

$$\frac{L_t}{L_0} = \frac{1}{1+\frac{2}{5}\cdot(1-\frac{t}{t_0})}$$

where t_0 is the age of the sun – 4.57 Gyr (4.57×10^9 yr) and L_0 is the present-day solar luminosity (3.85×10^{26} W).

The value of L_0 is equivalent to a flux (Wm^{-2}) of 1368 Wm^{-2} incident at the top of the atmosphere at Earth, which is given the symbol S_0. In the equation, L_0 can be substituted for S_0 to give the value of S at ant time, i.e. S_t (Wm^{-2}).

Note that in the formula, t is counted (in Gyr) relative to the formation of the Sun (i.e. present-day would be: $t = 4.57$).

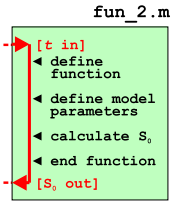

Figure 8.9: Schematic structure of code for calculating the solar constant (output) as a function of time (input).

8.1.6 *Using multiple functions and calculating global surface temperature as a function of geological time*

Finally ... you are going to bring it all together and calculate and plot the surface temperature of the Earth, at 100 Myr intervals, from 4.0 Gyr (4 billion years) in the past, to 4.0 Gry in the future – spanning approximately the age of the Earth and much of its potential long-term future.

Start by creating one final new (blank **m-file**) script ('`scr_4`').. You are going to need a loop in time, perhaps looping from 4.0 to -4.0 Ga relative to now (but you can chose what limits you like ... except remembering the Sun is only 4.57 Ga old ...). Within the loop, you will:

1. Pass to your solar constant function the current time, and obtain the corresponding value of the S_t – remember that you must add 4.57 to the time you pass into your function as the equation for S_t is in terms of time since the formation of the Sun, not relative to now.
2. Call your EBM function to calculate the corresponding surface temperature, passing it the value of S_t you have just calculated.
3. Store in an array, or pairs of vectors, time and the corresponding value of T.

Likely bug possibilities include the units of time (Gyr), and that time in the equation for S_0 is counted forwards from the formation of the Sun. Also be careful with nested parentheses (()). A schematic of the program structure is shown in Figure 8.10.

Assuming that you have managed something like Figure 8.11[18] – what strikes you, in light of (hopefully) what you know about the past history of climate and evolution of life on this planet, about your model projection (for the past)? What is 'missing'?

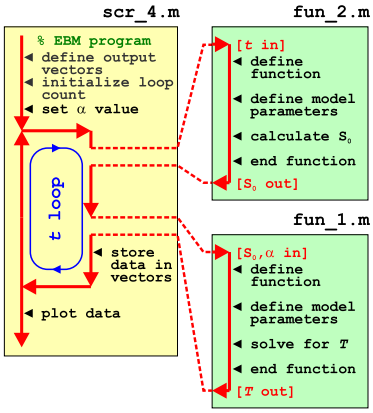

Figure 8.10: Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, and solar constant and EBM functions.

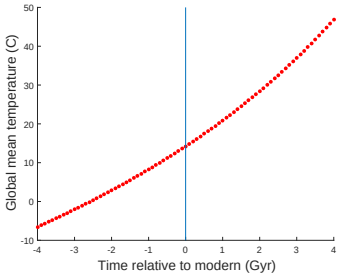

Figure 8.11: Simple EBM projection of the evolution of Earth surface temperature with time. Time at the present-day is highlighted by a vertical line (drawn using the **MATLAB** `line` function).

[18] Note that a line has been added to highlight $t = 0$ (i.e. the present-day) – see `line`.

`line` ... quite simply, draws a line. The basic syntax of the command is:

```
line(X,Y)
```

which plots a line between a paid of (x,y) coordinates. In the MATLAB usage, for a single straight line segment: the vector `X` contains both the x coordinate values, and `Y` both the y coordinate values.

In the specific Example in the text, the vertical line is drawn by:

```
line([0 0],[-10 50]);
```

NOT forgetting to put `hold on` first ...

8.2 'Daisy World'

There is an absolutely classic paper from the early 1980s – *Watson and Lovelock* [1983] – that illustrates how simple (biological) feedback on climate can lead to a close regulation of global climate over an appreciable span of the Earths past (and future). The premise for this model is a planet covered in bare soil (essentially, as per in the earlier EBM), but on which 2 different species of daisies (could be any pair of plants with contrasting properties) can grow – one white (high albedo) and one black (low albedo)[19]. Because the two species modify their local (temperature) environment and their net growth depends on how close the local temperature is to their optimum growth temperature, a powerful climate feedback operates and as the solar constant increases, the abundance of daisies switches from black to white – driving an increasing cooling tendency of the planet surface in the face of increasing solar-driven warming. This regulation emerges as a property of the dynamics of the population ecology and interaction with climate and does not require an explicit regulation of climate to be specified. Just dumb daisies doing their day-to-day stuff.

We'll code up this model ... but as before, in discrete stages (aka, the following Subsections).

[19] As pointed out in *Watson and Lovelock* [1983], the actual 'colors' are immaterial – just tat the albedos differ.

8.2.1 This will be the simplest addition to your previous model[20]. You'll create a new 'fixed daisy' function (`fun_3`) which will take no(!) inputs, and return a value for mean global albedo. You'll also copy-rename yourself a new script ('`scr_5`' – based on `scr_4`) and in it, take the albedo value generated by the call to the daisy function, and pass it into you EBM function (`fun_1`). (See Figure 8.12.)

[20] i.e. the one comprising a loop through time, and within this loop, calls to your function to convert time to solar constant, and take the solar constant (and albedoo and solve for mean global surface temperature. This was '# `scr_4`' in the previous Section notation.

8.2.2 Now, in the next stage it gets a little more complicated, because in a further new function ('`fun_4`' – copy-renames-and-edited from `fun_3`) you'll modify the equations such that the relative abundance of each daisy type is now responsive to the value of global temperature. The situation thus becomes – the relatively fractions of dark and light colored daisies is a function of global surface temperature, yet ... global surface temperature, through the mean (fractional area weighted) albedo of the daisies, is a function of the relatively fractions of dark and light colored daisies – a circularity (feedback loop). We'll resolve this circularity (i.e. come to a steady state solution) by creating an inner loop that comprises only the daisy function and EBM function and keeps looping until ... well, we'll start by simply prescribing a fixed number of iterations of the loop.

(See Figure **??** for a schematic of the code setup.)

8.2.3 Finally (almost) – we'll allow the daisies affect their *local * (temperature) environment. Now it gets more interesting (honest!). Although the code structure is exactly the same as in the last step[21], you will require a further copy-rename-and-edit of the previous daisy function ('`fun_4`' → '`fun_5`') and one further copy-rename-and-edit of the previous script ('`scr_6`' → '`scr_7`') that calls the daisy function.

8.2.4 In a minor extension to the previous work, we can modify the loop involving the daisy function and EBM function such that it will proceed until an adequately accurate solution (for global temperature) has been converged upon (rather than looping a fixed number of times).

8.2.1 'fixed daisy' daisy-world

To start: read *Watson and Lovelock* [1983]. You should be able to take away from this some of the essential information that you need to specify and keep track of. For now, we'll just concern ourselves with defining the albedo of bare ground (soil) and the albedo of each daisy together with how much area is covered by each species of daisy.

Create a new function (`fun_3`) – configure it so that it returns a single parameter – albedo. For now it has no inputs.[22] How it relates to your previous program and code for how the Earth's surface temperature evolves over geological time, is illustrated in Figure 8.12.

Now, in the daisy function (`fun_3`) near the top, define yourself some parameters for the daisy model:

```
% define model parameters - daisy albedo
par_a_s = 0.3; % albedo - bare soil
par_a_w = 0.5; % albedo - white daisies
par_a_b = 0.1; % albedo - black daisies
% define model parameters - daisy land fraction
par_f_w = 0.01; % (land) fraction - white daisies
par_f_b = 0.01; % (land) fraction - black daisies
```

(or using whatever parameter names you prefer). Here, the albedo values associated with each daisy type are fixed and will be used regardless of what the model does. The values have been chosen, assuming equal proportions of black and white daisies, to given an average of 0.3 – the albedo of bare soil and also the assumed value in the previous EBM. You'll modify and play with this value all too soon enough. The surface area fraction values are just initial values to start the model off with.[23]

Next, and actually the only line of any note in the function – you need to calculate an average albedo[24] – calculated based on the area

[21] A loop through geological time, as per in the previous Section. Within this main loop, you'll have a sub-loop with just the daisy function followed by the EBM function.

[22] A funny sort of function, although pretty well much like `pi`.

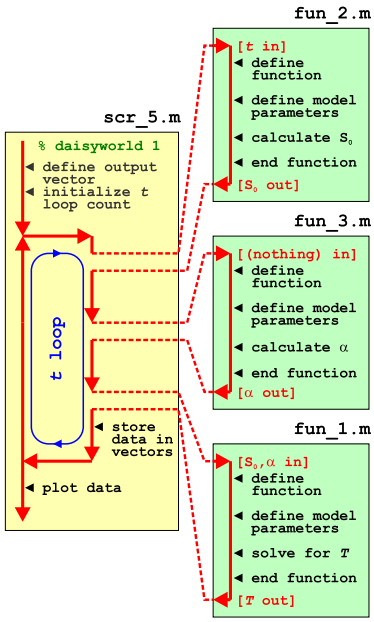

Figure 8.12: Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, the solar constant and EBM functions, and now the 'daisy' albedo function.

[23] As you'll come to see subsequently, these cannot be zero. Or rather, a daisy species can start with a fractional area of zero, but you'll never ever get any of that species growing, regardless of the environmental conditions (because there are none to start with!).

[24] Note that it is very easy to accidently prescribe a total area covered by daisies of >100%. You should ideally put a check (`if ... end`) in the code before it tries to calculate anything for whether the total area initially covered by daisies exceeds what is possible. If this is the case, your code might spit out a warning message (a simple `disp` command would do). You might also terminate your program (see `exit`).

weighted average of: bare soil, white daisies, black daisies. The calculation is simple and you already have the areas of the two species of daisy as fractions. You weight the contribution to global albedo by the albedo of each daisy by its fractional area. You then just need to calculate the fraction of the Earths surface that is bare soil – the area fraction not covered by daisies. In maths-speak, the mean albedo is given be:

$$\alpha = F_w \cdot \alpha_w + F_b \cdot \alpha_b + (1.0 - F_w - F_b) \cdot \alpha_s$$

where α_w, α_b, and α_s, are the albedos of white and black daisies, and bare soil, respectively, and F_w and F_b are the fractional areas of occupied by white and black daisies, respectively (with bare soil comprising the remainder). You simply need to translate this into **MATLAB** code using the parameters you defined earlier (for α_w, α_b, and α_s, and F_w and F_b). Write this line of code, which the one and only calculation the function carries out, just before the `end` of the function.

Thats actually it. All the parameter values are specified and fixed (see above), so nothing particularly exciting is going to happen ... Regardless – run the the complete model with the value of albedo now depending on the fraction of white and black daisies – it should look identical to before in terms of the evolution of surface temperature with time (it must, because the default parameters above ensure that the mean albedo is always 0.3 and the daisies don't even know anything about growing (or dying) yet). Model (surface temperature) output, including how the populations of the 2 species of daisy also vary with time, is shown in Figure 8.13).

You might play briefly with the prescribed daisy fractions and albedo values and e.g. check that when you specify a configuration with 100% of land area covered by black daisies, the climate is much warmer throughout the simulation, and when white daisies are assigned an initial value of 1.0, the climate is always much cooler compared to in the default simulation.

8.2.2 *'dumb daisy' daisy-world*

OK – step #2 in the evolution of Daisy World, and for the next modification and one which will actually make something 'happen' (i.e. the simulation will be different to that of the default EBM based simulation of mean global temperature response to increasing S_0). In fact, the daisies are going to grow and die (but unlike Southern California, not burn), with their population changing over time until an equilibrium is reached (for a particular specified value of S_0). *Watson and Lovelock* [1983] give a simple population model formulation for

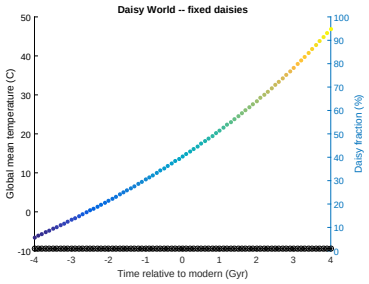

Figure 8.13: Evolution of global surface temperature and the two populations of daisies with time ... but with no change allowed in the daisy populations (d'uh!). The fractional coverage of white daisies is shown by large empty circles, and for black, by small filled black circles. Data points for mean surface temperature are color-coded by temperature (color scale not shown).

Daisy population dynamics (1)

For an area fraction occupied by white and black daisies of F_w and F_b, respectively, the change in occupied fractional area with time (t) can be written:

$$dF_w/dt = F_w \cdot (x \cdot \beta_w - \gamma)$$
$$dF_b/dt = F_b \cdot (x \cdot \beta_b - \gamma)$$

where x is the free (i.e. not occupied by daisies of any color) area of (fertile) ground, equal to:

$$x = 1.0 - F_w - F_b$$

(assuming here, unlike the more general case in *Watson and Lovelock* [1983], that all the land area is potentially fertile), β is a temperature-dependent growth function (one for each species of daisy), and γ the mortality rate (as a proportion of the area covered by that species of daisy per unit time). The value of γ given in *Watson and Lovelock* [1983] is 0.3, but this could be a parameter that you could play about with and investigate its effects.

To simplify things to start with, growth is a function only of the global mean temperature (in °C):

$$\beta_w = 1.0 - 0.003265 \cdot (22.5 - T)^2$$
$$\beta_b = 1.0 - 0.003265 \cdot (22.5 - T)^2$$

(where the value of 22.5 °C is a reference temperature and represents where optimal (maximum) growth occurs).

the change in area fraction covered by both sorts of daisy with time (also see Box) that we will implement here.

The unit of population in Daisy World is fractional area covered. So each time-step, the fractional area or each species will grow or shrink, depending on whether mortality is higher than growth. Both growth and mortality are formulated as being dependent on the fractional area (at the previous time-step), i.e. growth in covered area depends on how much is already covered. Similarly, mortality also depends on how many daisies are currently there. The growth rate is further modified by the available fractional area, such as that the area left shrinks, the growth rate shrinks. (Effectively, this is perhaps trying to account perhaps for shrinking resources available for further growth. It also has the effect of adding numerical stability to the model and helps presents over-shoots where the total fractional area covered by daisies far exceeds 1.0 ...).

How them to implement this in code?

- In general – start by identifying any constants – i.e. fixed and invariant, fundamental values, such as π or the Stefan-boltzmann constant. These values could be hard-coded into the equation as numbers, but better is to replace them with variables that you'd define at the top of the m-file as this makes for neater and easier-to read **MATLAB** code.
- Next identify any parameters – values that are not fundamental properties of the universe, but may be considered invariant for sequential uses of the equation. The characteristic albedos of the two species of daisies is a good example – these values are 'fixed', although, one day you might change them. If the code file is a script – define MATLAB variables and assign values to them, near the start of the code file. Otherwise, if a function, you may need to pass these parameters into the function and so they need to appear in the function definition on the 1st line of the code.
- Identify any output variables, i.e. result(s) of the calculation. In a function, these are invariably pass back out and hence need to appear in the function definition on the 1st line of the code. Output variable may also be input variables – i.e. a calculation may take the current value of a variable (as an input), update it, and then pass it back out. In which case, the variable will need ot appear as both input and output. Perhaps pick distinction variable names to avoid confusion, e.g. `var_in` and `var_out`.
- You may have local variables (i.e. used only within the script and out outside of it). If scalars, these need not be defined and initialized, unless used as e.g. a counting or running-sum variable. If in doubt, maybe also define and initialize e.g. to zero local variables.

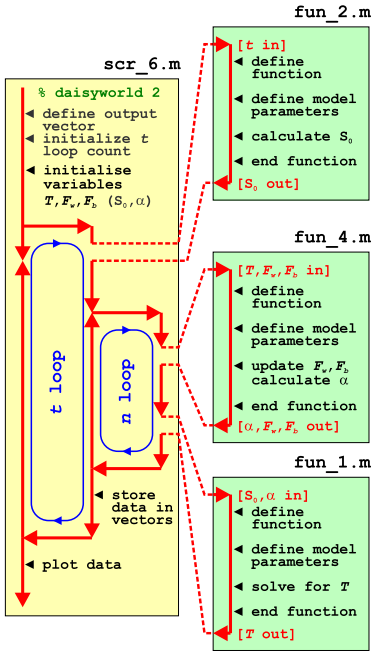

Figure 8.14: Schematic of the evolution of surface temperature over geological time program, and relationship between main program script, the solar constant, EBM, and 'daisy' albedo functions. Note the creation of an inner loop, with EBM, and 'daisy' albedo functions called from within this, while the solar constant remains called form the start of the outer loop as before.

- Otherwise, it is mostly just a case of writing the maths, in MATLAB – changing symbols where necessary and replacing the letters (invariably) used in the equations with your variable names.

Figure 8.14 gives a schematic of the overall code structure for this model. **DON'T PANIC.** There are actually only 2 (or 3-ish), relatively incremental changes, compared to previously. Start off by noting what is the same – both the function for the solar constant (`fun_2`) and the EBM model (`fun_1`) are exactly the same as before. The loop in (geologic time) and hence some of the script (`scr_6`) is also the same. What is different and yet to-do?

1. Lets start with the daisy function. You could deal with the inputs and outputs first. As as well as T, now the previous values of the fractional areas of the two daisies are required (F_w, F_b) (which is different from before where the values were assumed and the respective parameters set at the start of the function[25]). This is because each time the daisy function is called, the fractional areas are updated (hence why they are inputs). And outputs. Beasue the daisy function is updating the fraction al areas, these two parameters also need to be outputs too. So the very first thing to do is to modify the function definition, so that the inputs are:

   T, F_w, F_b

   and the outputs are:

   α, F_w, F_b

   (see help of various sorts on *functions*, but it not at all a fundamental change as to compared to before).

   Then, the only other development in the function, is to implement the equations for daisy growth/death and update the values of F_w, F_b (and at the end, calculate the value of α as before). And ... set the parameter values for β and γ of the two daisy species (near the start of the function).

2. Secondly, it is going to take a number of iterations for the daisies to grow/die ... changing their fractional areas and hence albedo as their fractional areas change ... and hence ultimately, reaching a new equilibrium with global climate. Each time around the outer loop – because the value of S_0 will change each time, climate will change and the daisy population will no longer be in equilibrium (because their fractional areas are carried over from the previous loop iteration). Hence in the outer loop you will need an inner loop to determine the new equilibrium and global temperature for that particular value of S_0. For now the loop can be

[25] So if you are copy-pasting the previous Daisy function, you need to delete the lines:

```
par_f_w = 0.01;
par_f_b = 0.01;
```

quite simple – we'll assume 100 iterations (i.e. the loop counter `n`, will go from `1` to `100`).

3. Lastly, the initialization of the main program (`scr_6`) will be a little different from before. Because the daisy function now takes as input, F_w and F_b – you'll need to give these variables each an initial value (near the start of the program) so that first time the function is called, there is a value for the equations to work with. Similarly, temperature T now also becomes an input to the daisy function (and it is not set anywhere else beforehand in the very first iteration of the loops), so it also needs an initial values to be assigned.[26]

If you have set this daisy population dynamics enabled EBM (a DPDE-EBM!) up correctly, and drive it with your -4.0 to +4.0 Ga solar constant calculating script, you should get something like Figure 8.15.

OK, so actually, this is not different in terms of the global mean temperature response (to solar evolution), to before. But then again, you have set both species of daisy with the same temperature growth response. In other words, as the white daisies with a high albedo grow, so to the black ones with a low albedo. Equally. And their different albedos balance, meaning that α still never changes. One thing you could try to liven things up a little is to change on of the value of β (and/or γ) so that their population dynamics are not identical. Now, if the relative abundance of white and black daisies changes, so too with global mean albedo and hence global temperature.

[26] For completeness, you could also initialize S_0 and α, but it is not strictly needed, as they are calculated and defined before they are first used.

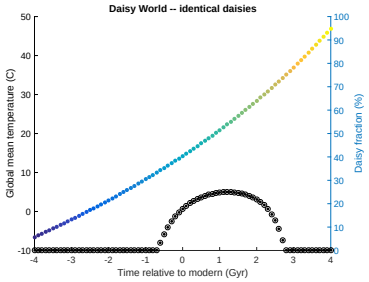

Figure 8.15: Evolution of global surface temperature and the two populations of daisies with time ... but now assuming that the growth of each depends on the global mean surface temperature.

8.2.3 *'clever daisy' daisy-world*

The last step is to give each species of daisy a different environmental preference for growth (why? because that is how the World works – different plants and ecosystems tend to inhabit different environmental regimes as a result of being (evolutionary) adapted to different environmental parameters). *Watson and Lovelock* [1983] assume that both species of daisy have the same temperature preference but modify their local environment differently – white daisies inducing a local cooling relative to the global mean temperature, and the presence of black daisies driving a local heating (see Box). The result is Figure 8.16.

Now the behaviour of the system and the evolution of global mean surface temperature with time, is very different. Towards the start of the experiment, and at very low values of S_0, the global mean temperature is too cold to support a daisy population (of either type). As the value of S_0 increases, initially global mean temperature follows the path it did before, in the absence of daisies (or with fixed, or equal populations). At a certain point, black daisies, because of their

Daisy population dynamics (2)

To make the different species of daisies interact differently with the environment, the temperature-dependent modifiers of growth are made functions of the local (to the daisy population or individual), rather than global, temperature:

$$\beta_w = 1.0 - 0.003265 \cdot (22.5 - T_w)^2$$
$$\beta_b = 1.0 - 0.003265 \cdot (22.5 - T_b)^2$$

There are all sorts of says of defining how the local temperature deviates form the global mean. In *Watson and Lovelock* [1983] this is simply reduced to a simple deviation that scales linearly with the difference between mean global and local (daisy) albedo:

$$T_w = T + q \cdot (A - A_w)$$
$$T_b = T + q \cdot (A - A_b)$$

(noting that A is albedo here, not alpha as was the case in the original (non daisy enabled) EBM). q is a simple scaling factor that describes how strongly the local temperature deviates from the mean (or conversely, how efficiently heat energy is mixed between differen daisy fractions) and is assigned a default value of 10.0.

advantage that they absorb more sunlight and drive a locally warmed climate, take off in population and rise to dominate 70% of the land surface. The global mean temperature transitions sharply to a much higher temperature state. As S_0 further increases in value, they increase slightly further in dominance (and global temperature climb a little further in response) until locally they reach their optimal temperature for growth. Past this (optimal temperature) point, white daisies start to grow and slowly replace the black ones. Global climate is almost perfectly stabilized during this interval. Beyond this, there is a short interval where black daisies die out and white daisies go on to reach their own (local) temperature optimum. Beyond this again, everything suddenly goes extinct in a rapid warming feedback of increasing temperatures, declining white daisy numbers, further solar radiation absorption and warming, etc etc. How everything is dead and I how you are feeling happy with yourself.

You could code this modification in – adjusting the (local) value of T that each species of daisy 'sees' (as per the Box and the reference). Or ... we could simply give them different temperature optima, which is what the value of 22.5°C accomplishes in the temperature-dependent growth modifier equation. For now, this is the way-simpler approach and involves only a minimal edit to your existing daisy function. So where in the equation for β_w and β_b you currently have values of 22.5 (°C) in each – try making these different. Reasonable would be to assume that the white daisies are more adapted to hot climates and hence have a higher temperature tolerance, with black daisies being better adapted to colder climates, using their higher albedo and presumably local heating to make up for a colder ambient environment. (You could be able to come up with something not entirely dissimilar to Figure 8.16.)

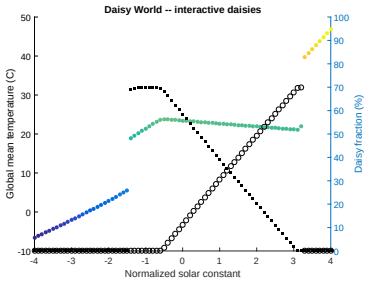

Figure 8.16: Evolution of global surface temperature and the two populations of daisies with time.