CLIMATE MODELS AND CLIMATE MUDDLES

Willis Eschenbach

NETZERO WATCH

Climate Models and Climate Muddles

Willis Eschenbach

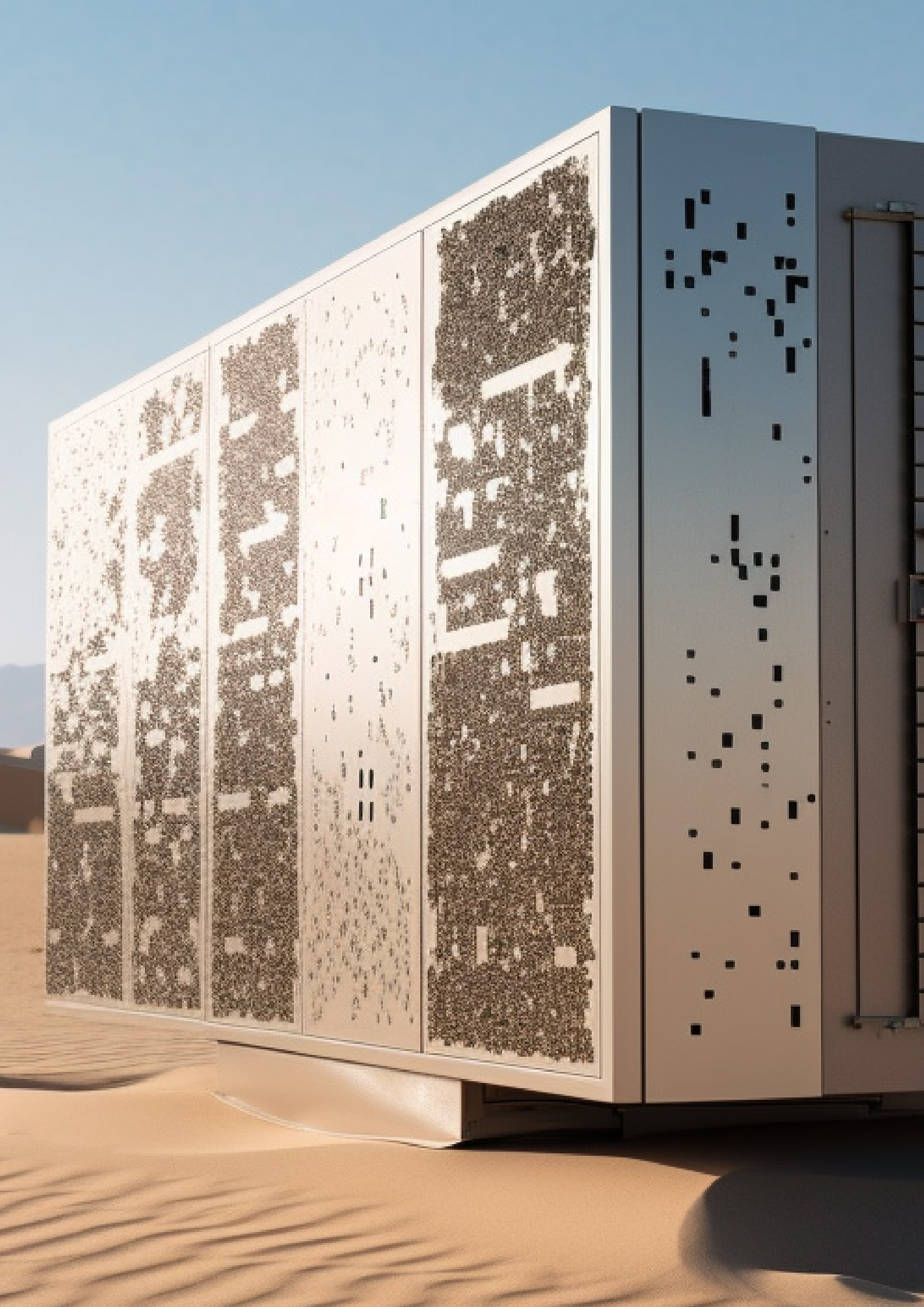
Contents

About the author

Willis Eschenbach has been programming computers for more than 60 years, as well as having written about climate change for more than a decade.

Acknowledgement

This paper is based on an article originally published at wattsupwiththat.com. Net Zero Watch are grateful to Anthony Watts for permission to republish.



1. Introduction

Like many climate models, NASA's GISS Model E was not designed before construction started. Instead, it has grown over decades by accretion, with new parts added, kluges applied to fix problems, ad-hoc changes made to solve new issues, and the like. Or, to quote one of the main programmers, Gavin Schmidt, in a paper describing Model E:

> The development of a [climate model] is a continual process of minor additions and corrections combined with the occasional wholesale replacement of particular pieces.[1]

An additional complication is that, as with many such programs, it's written in the computer language FORTRAN. This was an excellent choice in 1983 when the model was born, but is a horrible language for 2023.

How much has the code grown over the last 40 years? If you exclude the auxiliary files, the FORTRAN code itself has reached 441,668 lines of code. As a result, it can only run on a supercomputer. I last looked at the code two decades ago, but in 2022, I thought I'd look again, to see how it has changed.

2. Stability and instability

Modern climate models have a hard time replicating the amazing stability of the climate system. They are 'iterative' models, meaning that the output of one timestep is used as the input for the next. As a result, any error in the output of timestep *J* is carried over as an error in the input into timestep *K*, and so on ad infinitum. This makes it very easy for the model to spiral down into a 'snowball earth' or to overheat, making the virtual planet go up in flames. Figure 1, for example, shows a couple of thousand runs of a climate model.

Notice, in the upper panel, how many runs fall out the bottom during the control phase. That has never happened with the real Earth.

To prevent the climate model 'losing the plot' and wandering away from reality in this way, a programmer needs to work out what is wrong with the physics of the model and then fix it. However, when I first looked at the Model E code 20 years ago, I found one area where the NASA team were doing something very different.

Figure 1: 2017 runs from the climateprediction. net climate model.

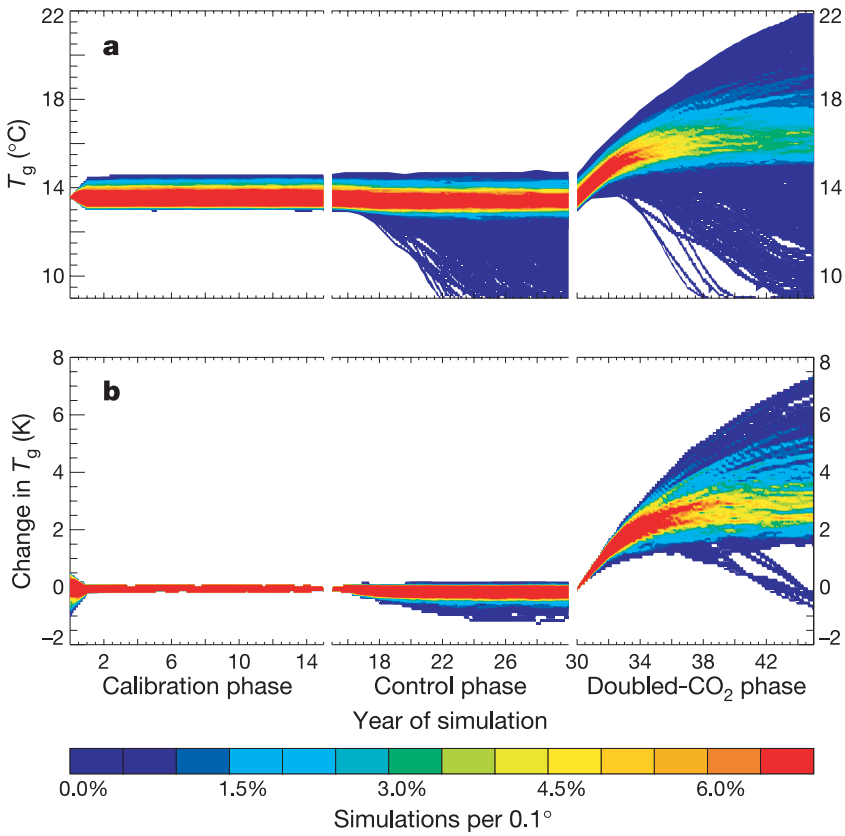


Figure 1 Frequency distributions of T_g (colours indicate density of trajectories per 0.1 K interval) through the three phases of the simulation. **a**, Frequency distribution of the 2,017 distinct independent simulations. **b**, Frequency distribution of the 414 model versions. In **b**, T_g is shown relative to the value at the end of the calibration phase and where initial-condition ensemble members exist, their mean has been taken for each time point.

3. Hard limits

Polynyas are pools of meltwater that form on top of the seasonal sea ice at the poles. These are important in calculating the reflectivity ('albedo', in the jargon) of the sea ice, which is an important factor in determining how much of the sun's incoming heat is reflected straight back upwards. The first time I looked at Model E, I discovered that polynyas had been a big problem – the model was suggesting they were present on the ice for far too much of each year. The model was essentially too hot. However, rather than work out why, they had simply added a hard time limit on the number of days during which melt pools could form.

Looking at the code as it is today, it appears that they've done some work in this area, because I can no longer find the routine that created that restriction on the number of days. Instead there is a new subroutine that sets hard limits on the values for the albedo for sea ice and melt ponds, as well as specifying constant values for wet and dry snow on the sea ice. Finally, it also specifies the limits and values for visible light (VIS), and for five bands of the near infrared (NIR1-5). The code is shown in Code Block A, for those who are interested (the 'c' or the '!' in a line indicates a comment).

This means that there is code to calculate the albedo of the sea ice, but sometimes that code comes up with unrealistic output. But rather than figuring out why, and then fixing the problem, the NASA team are just replacing the bad value with the corresponding maximum or minimum values. Science at its finest!

Code Block A

```
C**** parameters used for Schramm sea ice albedo scheme (Hansen)
!@var AOImin,AOImax     range for seaice albedo
!@var ASNwet,ASNdry     wet,dry snow albedo over sea ice
!@var AMPmin      mininimal melt pond albedo
       REAL*8 ::
C                          VIS   NIR1   NIR2   NIR3   NIR4   NIR5
      *     AOImin(6)=(/ .05d0, .05d0, .05d0, .050d0, .05d0, .03d0/),
      *     AOImax(6)=(/ .62d0, .42d0, .30d0, .120d0, .05d0, .03d0/),
      *     ASNwet(6)=(/ .85d0, .75d0, .50d0, .175d0, .03d0, .01d0/),
      *     ASNdry(6)=(/ .90d0, .85d0, .65d0, .450d0, .10d0, .10d0/),
      *     AMPmin(6)=(/ .10d0, .05d0, .05d0, .050d0, .05d0, .03d0/)
```

Code Block B shows a comment in describing another bit of melt-pond fun.

Code Block B

```
C**** safety valve to ensure that melt ponds eventually disappear (Ti<-10)
      if (Ti1 .lt.-10.) pond_melt(i,j)=0.  ! refreeze
```

In plain English, it is saying that if the temperature is less than −10°C, and the polynya hasn't refrozen, make it refreeze. Without this bit of code, some of the melt ponds might never refreeze, no matter how cold it got…you have to love that kind of physics – water that doesn't freeze! This is what the climate modellers mean when they say that their model is 'physics-based'. They use the term in the same way Hollywood producers do when they say a movie is 'based on a true story'.

Code Block C (which is close enough to plain English for anyone to understand) is another great comment from the Model E code.

Code Block C

```
!@sum tcheck checks for reasonable temperatures
!@auth  Ye Cheng/G. Hartke
!@ver   1.0
c ----------------------------------------------------------------------
c  This routine makes sure that the temperature remains within
c  reasonable bounds during the initialization process. (Sometimes the
c  the computed temperature iterated out in left field someplace,
c  *way* outside any reasonable range.) This routine keeps the temp
c  between the maximum and minimum of the boundary temperatures.
c ----------------------------------------------------------------------
```

In other words, when the temperature goes off the rails… don't investigate why and fix it. Just set it to a reasonable temperature and keep rolling.

And what is a reasonable temperature? Turns out they just set it to the temperature of the previous timestep and keep on keeping on…physics, you know. Code Block D is another.

Code Block D

```
c  ucheck makes sure that the winds remain within reasonable
c    bounds during the initialization process. (Sometimes the computed
c    wind speed iterated out in left field someplace, *way* outside
c    any reasonable range.) Tests and corrects both direction and
c    magnitude of the wind rotation with altitude. Tests the total
c    wind speed via comparison to similarity theory. Note that it
c    works from the top down so that it can assume that at level (i),
c    level (i+1) displays reasonable behavior.
```

So once again, the climate model goes off the rails: the wind is blowing at five hundred miles per hour. But don't look for the reason why; just prop it up, put it back on the rails, and…keep going.

4. Tuning

Tunable parameters are a completely different class of non-physics. Here's a description from the Gavin Schmidt paper cited above:

> The model is tuned (using the threshold relative humidity…for the initiation of ice and water clouds) to be in global radiative balance (i.e., net radiation at [the top of the atmosphere] within ±0.5 W m−2 of zero) and a reasonable planetary albedo (between 29% and 31%) for the control run simulations.

Translating that into plain English, in the model, the sun's incoming heat doesn't end up in equilibrium with the heat escaping to space, so the virtual Earth in the model either overheats or turns into a snowball. The solution applied is to adjust a parameter buried deep in the model until equilibrium is reached. You simply turn the tuning knob and presto! It all works fine!

In fact, the tuning knob worked so well that they put in two more…plus another hard limit (see Code Block E):

Code Block E

```
!@dbparam U00a tuning knob for U00 above 850 mb without moist convection
!@dbparam U00b tuning knob for U00 below 850 mb and in convective regions
!@dbparam MAXCTOP max cloud top pressure
```

All models are subjected to what I call 'evolutionary tuning'; the process whereby a change is made, and then the model is tested against the only thing we have to test it against – the historical record. If the model is better able to replicate the historical record, then the change is kept. But if the change makes it work worse at 'hindcasting', it's thrown out.

Unfortunately, as the stockbrokers' ads in the US are required by law to say, 'Past performance is no guarantee of future success'. The fact that a climate model can hindcast (reproduce the past) means absolutely nothing about whether it can successfully predict the future. This is particularly true when the model is propped up and kept from falling over by hard limits and tunable parameters, and then evolutionarily tuned to reproduce the past…

5. Etcetera

What else is going on? Well, as in many such ad-hoc projects, the Model E team have ended up with a single variable name representing two different things in different parts of the pro-

Code Block F

```
SUBR  identifies after which subroutine WATER was called
SUBR  identifies where CHECK was called from
SUBR  identifies where CHECK3 was called from
SUBR  identifies where CHECK4 was called from

ss = photodissociation coefficient, indicies
SS = SIN(lat)*SIN(dec)

ns = either 1 or 2 from reactn sub
ns = either ns or 2 from guide sub i2 newfam ifam dummy variables

nn = either nn or ks from reactn sub
nn = either nn or nnn from guide sub
nn = name of species that reacts, as defined in the MOLEC file.

ndr = either ndr or npr from guide sub
ndr = either nds or ndnr from reactn sub

Mo = lower mass bound for first size bin (kg)
Mo = total mass of condensed OA at equilibrium (ug m-3)

ks = local variable to be passed back to jplrts nnr or nn array.
ks = name of species that photolyses, as defined in the MOLEC file.

i,j = dummy loop variables
I,J = GCM grid box horizontal position
```

gram, which may or may not be a problem, but is a dangerous programming practice that can lead to unseen bugs. (Note that FORTRAN is not 'case sensitive', so 'ss' is the same variable as 'SS'; there is nothing to stop the programmer using both, in different parts of the program, but this is highly inadvisable.) Code Block F shows some of these duplicate variable names.

Finally, there's the problem of the model failing to conserve energy and mass. Code Block G shows one way it's handled…

Curiously, the subroutine 'addEnergyAsDiffuseHeat' is defined twice in different parts of the program…but I digress. When energy is not conserved, what the code does is simply take the difference and spread it equally all over the globe. Now, a

Code Block G

```
C**** This fix adjusts thermal energy to conserve total energy TE=KE+PE
       finalTotalEnergy = getTotalEnergy()
       call addEnergyAsDiffuseHeat(finalTotalEnergy - initialTotalEnergy)
```

Figure 2: Murphy gauge

subroutine like this is necessary because computers are only accurate to a certain number of decimals. Rounding errors are inevitable. The method used is not unreasonable. However, twenty years ago I asked Gavin Schmidt if he had some kind of 'Murphy gauge' on this subroutine to stop the program if the energy imbalance was larger than some threshold. (A Murphy gauge gives an alarm if some user-set value is exceeded; see Figure 2). Without such a gauge, the model could be either gaining or losing a large amount of energy without anyone noticing. Gavin said no, he didn't have any alarm to stop the program if the energy imbalance was too large. So I asked him how large the imbalance usually was. He said he didn't know.

So revisiting the code 20 years later, once again I looked for such a 'Murphy gauge'…but I couldn't find one. I've searched the subroutine 'addEnergyAsDiffuseHeat' and the surrounds, as well as looking for all kinds of keywords, such as 'energy', 'kinetic', 'potential', 'thermal', as well as for the FORTRAN instruction 'STOP' which stops the run, and 'STOP_MODEL' which is their subroutine to halt the model run under certain conditions and then print out a diagnostic error message.

In Model E there are 846 calls to 'STOP_MODEL' for all kinds of things – lakes without water, problems with files, 'mass diagnostic error', 'pressure diagnostic error', solar zenith angle not in the range [0.0 to 1.0], infinite loops, and ocean variables out of bounds. One STOP_MODEL call actually prints out 'Please double-check something or another', while one of my personal favourites calls a halt when there is 'negative cloud cover' or 'negative snow depth'. I hate it when those happen…

And this is all a very good thing. These are Murphy gauges, designed to stop the model when it goes off the rails. They are an important and necessary part of any such model. But I couldn't find any Murphy gauge for the subroutine that takes excess or insufficient energy and sprinkles it evenly around the planet. Now, to be fair, there are 441,668 lines of code, and it's very poorly commented…so it might be there, but I couldn't track it down.

6. Conclusions

I wrote my first computer program over a half-century ago, and have written uncountable programs since. On my computer right now, I have over 2000 programs I wrote in the computer language R, with a total of over 230,000 lines of code. I've forgotten more computer languages than I speak, but I am (or at one time was) fluent in C/C++, Hypertalk, Mathematica (three languages), VectorScript, Basic, Algol, VBA, Pascal, FORTRAN, COBOL, Lisp, LOGO, Datacom, and R. I've done all of the computer analysis for the ~1,000 posts that I've written for the WattsUpWithThat website. I've written programs to do everything from testing blackjack systems, to providing the CAD/CAM files for cutting the parts for three 80' steel fishing boats, to a bidding system for complete house construction, to creating the patterns for cutting and assembling a 15-meter catenary tent, to…well, the program that I wrote today to search for keywords in the code for the GISS Model E climate model. So regarding programming, I know whereof I speak.

Next, regarding models. On my planet, I distinguish two kinds of model: single-pass and iterative models. Single-pass models take a variety of inputs, perform some operations on them, and produce some outputs. Iterative models, on the other hand, take a variety of inputs, perform some operations on them, and produce some outputs, but, unlike single-pass models, then use those outputs as inputs to the next iteration; the process is then repeated over and over to give a final answer.

There are a couple of very large challenges with iterative models. First, as I discussed above, they're generally sensitive and touchy as can be. This is because any error in the output becomes an error in the input, making them unstable. And, as also mentioned above, there are two ways to fix that – correct the code, or include guardrails to keep it from going off the rails. The right way is to correct it…which leads us to the second challenge.

The second challenge is that iterative models are very opaque. Weather models and climate models are iterative. Climate models typically run on a half-hour timestep. This means that if a climate model predicting, say, 50 years into the future, the computer will go through 48 steps per day times 365 days per year, times 50 years, or 876,000 iterations. And if it comes out with an answer that makes no sense, or defies physics, how can we find out where it went off the rails?

Please be clear that I'm not picking on the GISS model. These same issues, to a greater or lesser degree, exist within all large complex iterative models. I'm simply pointing out that these are *not* 'physics-based' – they are propped up and fenced in to keep them from crashing.

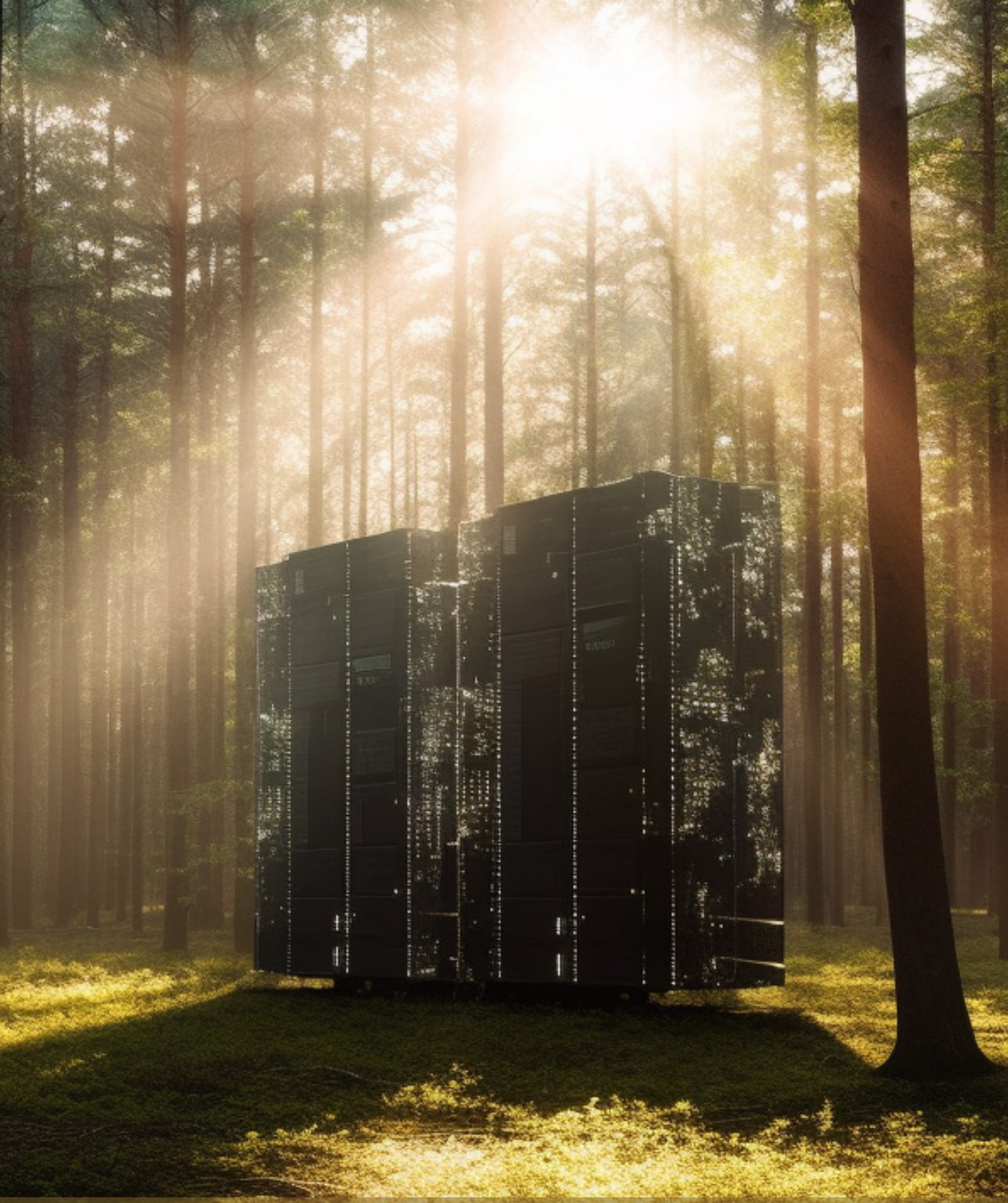
In conclusion, a half-century of programming and decades of studying the climate have taught me a few things:

- All a computer model can do is make visible and glorify the under- and, more importantly, the misunder-standings of the programmers. If you write a model under the belief that CO_2 controls the temperature…guess what you'll get?
- As the scholar of semantics Alfred Korzybski famously said, 'the map is not the territory'. He used the phrase to poetically express the idea that people often confuse models of reality with reality itself. Climate modellers have this problem in spades, far too often discussing their results as if they were real-world facts.
- The climate is far and away the most complex system we've ever tried to model. It contains at least six subsystems – atmosphere, biosphere, hydrosphere, lithosphere, cryosphere, and electrosphere. All of these have internal reactions, forces, resonances, and cycles, and they all interact with all of the others. The system is subject to variable forces from both within and without. My First Rule of Climate says 'In the climate, everything is connected to everything else…which in turn is connected to everything else…except when it isn't.'
- We've only just started to try to model the climate.
- Iterative models are not to be trusted. Ever. Yes, modern airplanes are designed using iterative models…but the designers still use wind tunnels to test the results. Unfortunately, we have nothing that corresponds to a 'wind tunnel' for the climate.
- The first rule of buggy computer code is, when you squash one bug, you probably create two others.
- Complexity is not Reliability. Often a simple model will give better answers than a complex model.

Bottom line? The current crop of computer climate models (which should really be referred to as 'climate muddles') is far from being fit to be used to decide public policy. To verify this, you only need to look at the endless string of bad, failed, crashed-and-burned predictions that they have produced. Pay them no attention. They are not 'physics-based' except in the Hollywood sense, and they are far from ready for prime time. Their main use is to add false legitimacy to the unrealistic fears of the programmers.

Notes

1 https://journals.ametsoc.org/view/journals/clim/19/2/jcli3612.1.xml.



For further information about Net Zero Watch, please visit our website at www.netzerowatch.com.

NETZERO WATCH