Mark A. Miller

Modeling Planetary Climates with Python

Last updated: January 10, 2019

Preface

With this book, I hope to increase the reader's understanding of the physical and biogeochemical processes that determine planetary conditions through hands-on modeling and comparison of models with observations.

Python is a powerful tool in this context, because for most of us non-computer scientists it is easier to use than the alternatives out there, especially its predecessors. Earth and planetary science is a data-driven endeavor, and many useful libraries have been developed to help work with and present data. I have included modeling with Python in this book because I believe that numerical modeling forces a researcher to think deeply about a specific case, developing intuition that is hard to gain in any other way.

I am in effect scratching my own itch with this extended tutorial. I have found that most existing resources on programming, numerical methods, modeling, and even quantitative Earth science in general are either too esoteric or too qualitative and general to be useful. I hope I strike a balance between scientific detail, accessibility, and practical explanations.

While this book is intended to apply to the modeling of planets of all kinds, it is decidedly Earth-centric, and is an unapologetic effort to help the scientifically minded better understand our own world through observation and modeling. My hope is that by helping to train the mathematically literate, it can do its little part to steer humanity away from our tendency to view our world through lenses of hubris, anthropocentrism, and even excessive geocentrism!

That said, a bit of anthropocentrism can make us grateful. We may not actually live on the best of all possible worlds, but present-day Earth comes pretty close. Any time spent thinking about what things are like on most (if not all) other planets can make a person just plain thankful for how nice it is on Earth. We've got water in all three phases, beautiful forests, plains, beaches, mountains, and deserts, oxygen to breathe, and enough other life to feed us and replenish the oxygen we happen to consume. In the end, human beings evolved on Earth. Compared to other planets, it's not too hot, not too cold, not too stormy, not too acidic, and a few minutes in the sunshine won't destroy all of our DNA. And that's just to mention a few examples of how blessed we are. It's certainly worth our time to quantitatively understand why Earth's climate is the way it is, and what changes its conditions.

While our scientific understanding of Earth's climate system is based on fundamental physics and chemistry, it is powerfully augmented by data from 1) other planets and 2) the geologic past. I like to think that we can attain greater clarity of thought when we consider our own planet as just one manifestation of the realm of possibility.

Importantly, the same physics that set conditions on other planets or that operated in Earth's past also apply to the modern Earth. Therefore paleoenvironmental records as well as observations from planet-like bodies both serve as base cases for refining conceptual and numerical models of Earths climate physics.

From the airless bodies in our solar system such as the Moon and Mercury, to the runaway greenhouse of Venus, and distant possible magma ocean worlds like 55 Cnc e, there is a rich set of important test cases for framing our understanding of the Earth environment. Likewise, certain episodes from Earths environmental history also serve as good test cases, from the proto-Earth, the Boring Billion years, and the Snowball Earth periods to the much more recent Mystery Period of the last deglaciation, as well as the Anthropocene in which we now live.

One important message that planetary and exoplanet research brings us is that planets can take a wide variety of forms depending on their elemental abundances, size, and relationship to their host star. Meanwhile paleoclimate and paleoceanographic research shows us that life has been a geologic force on planet Earth for the majority of its history, modifying climate parameters through biogeochemical processes. In fact, it is only through the geochemical effects of ancient simple life that Earth's physical environment ever became habitable for humans.

This is no elegant mathematical text, and Python veterans are likely to chuckle at its amateurishness. Chuckle away. This book is hands-on and messy at times, just as real research is. I've often felt that university courses often send you away with a lot of interesting data, scientific stories to tell, and beautifully derived equations, but without actually being able to do anything real. Meanwhile the details of real research can be so messy details that it is easy to forget the big picture. I hope that this book bridges that gap.

Python, nonlinear partial differential equations, and their numerical solutions are certainly not light reading for the masses, but they are the necessary tools for coming to your own first-hand conclusions about important topics like planetary climate dynamics. And those conclusions are certainly worth sharing with the world in ways that they can understand.

Sierra Nevada, California, January 2019 *M.M.*

”Ultimately, in order to understand nature it may be necessary to have a deeper understanding of mathematical relationships.”

”I never pay attention to anything by ”experts”. I calculate everything myself.”

To those who do not know mathematics it is difficult to get across a real feeling as to the beauty, the deepest beauty, of nature ... If you want to learn about nature, to appreciate nature, it is necessary to understand the language that she speaks in.

- Richard Feynman

Contents

Chapter 1
Naked planet model

1.1 The Moon, a rock in space

The temperature of any planet or planet-like body is determined by the rate at which it receives energy and the rate at which it loses energy. We will build a climate model from the ground up, module by module, based on this simple fact.

Due to its relative simplicity and proximity to the Earth, there may be no better place in the Universe for laying the foundations of an Earth-centric planetary climate model than the Moon. The Moon is on average the same distance from the Sun as Earth, making it a decent analogue to consider for building a simple energy balance model for a point on the surface of an airless, naked planet. Recent robotic observations of lunar surface temperatures by NASA's Diviner radiometer enable comparison of lunar and terrestrial temperatures. 19th century physics (particularly the Stefan-Boltzmann law) allow simple modeling of various points on the surface of the Moon.

In this chapter, we'll use the Python interpreter and then short Python scripts to calculate radiative equilibrium temperatures and generate plots of simulated diurnal temperature variation under various assumptions. Then in Chapter 3, we will develop a dynamic soil heat exchange model in order to provide a reasonable simulation of the Moon's diurnal temperature curve.

1.2 Lunar temperature observations

1.2.1 The first observations from Earth's airless sibling, the Moon

Our first observations of the temperature of the Moon date back to the 1920s, when Edison Pettit and Seth Nicholson were hard at work developing clever methods to make stellar, planetary, and lunar observations using the 100-inch telescope at the Carnegie Instution's Mount Wilson Observatory, in the mountains above Los Angeles. Pettit and Nicholson (1922) describe an apparatus which made use of a bismuth/bismuth-tin alloy vacuum thermocouple paired with a galvanometer. When the thermocouple is placed at the Newtonian focus of the telescope, and the incoming beam is passed through a window of rock salt (Pettit and Nicholson, 1928), the intensity of the infrared filtered and measured in this manner can be used to estimate the surface temperature of a small region of the Moon's surface by employing blackbody temperature curves.

Pettit and Nicholson (1930) calculated lunar surface temperatures at the subsolar point of between 358 (during the quarter moon) and 407 K (during the full moon). This range corresponds incredibly well with later, more precise measurements made by satellites. To investigate the thermal inertia of the lunar surface, they made the same temperature estimates with the 100 inch telescope during the 1927 and 1939 lunar eclipses (Jaeger, 1953). What they demonstrated is that the lunar surface does not behave as a perfect blackbody, that it has a thermal inertia that results in gradual cooling after solar irradiance drops to zero either after lunar sunset or during an eclipse. Their 1927 results are shown in Figure 1.1:

This important observation gave us a hint that the specific thermal properties of a planetary surface cause significant departures from instantaneous radiative equilibrium. In other words, an absorbent surface environment with thermal inertia can cause the planet to retain heat, increasing its temperature at certain times above the blackbody equilibrium temperature curve.

Our observations of the Moon are far more detailed nowadays. If we are to understand planetary climate dynamics starting with the simplest possible real case and incrementally adding in complexity, the coldest, darkest place in the solar system is a fantastic place to begin. It turns out, surprisingly, that this place happens to be on Earth's nearest neighbor, at its highest latitudes.

1.2.2 Modern observations of lunar temperatures

About 75 kilometers (45 miles) from the Moon's north pole, beneath the 2500 m (8200 ft) tall south rim of the Hermite crater [1], there is a spot that probably hasn't

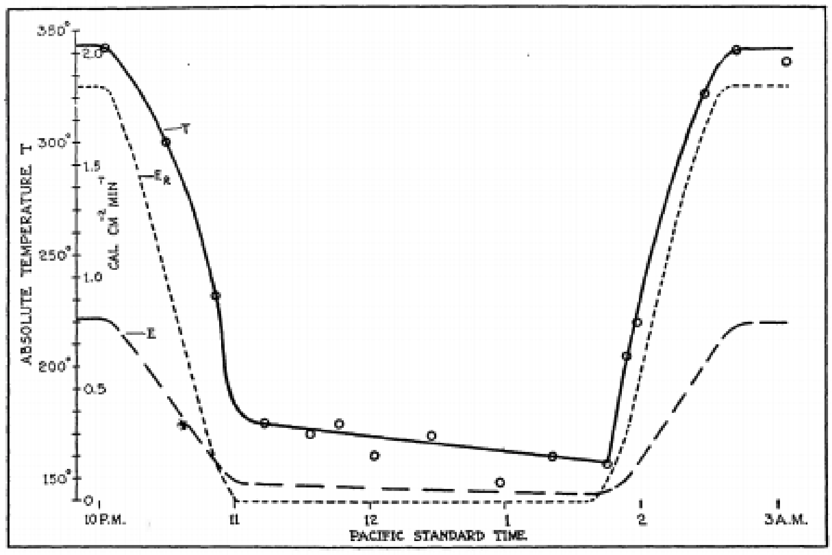

FIG. 6.—March of absolute temperature, T, of energy received from the sun by the moon, E_R, and of energy radiated, E, during the total lunar eclipse of June 14, 1927.

Fig. 1.1 Gradual cooling of lunar surface during eclipse, from Pettit and Nicholson (1930).

seen the light of day for millions of years. Unlike Earth, where the 23.5 degree tilt of the planet's rotational axis causes its polar regions to see half a year of sunshine and half a year of nighttime darkness, the Moon has an inclination of only 1.5 ° relative to the ecliptic [6], meaning that the topography of certain craters is sufficient to cast permanent shadows over parts of the Moon's airless surface.

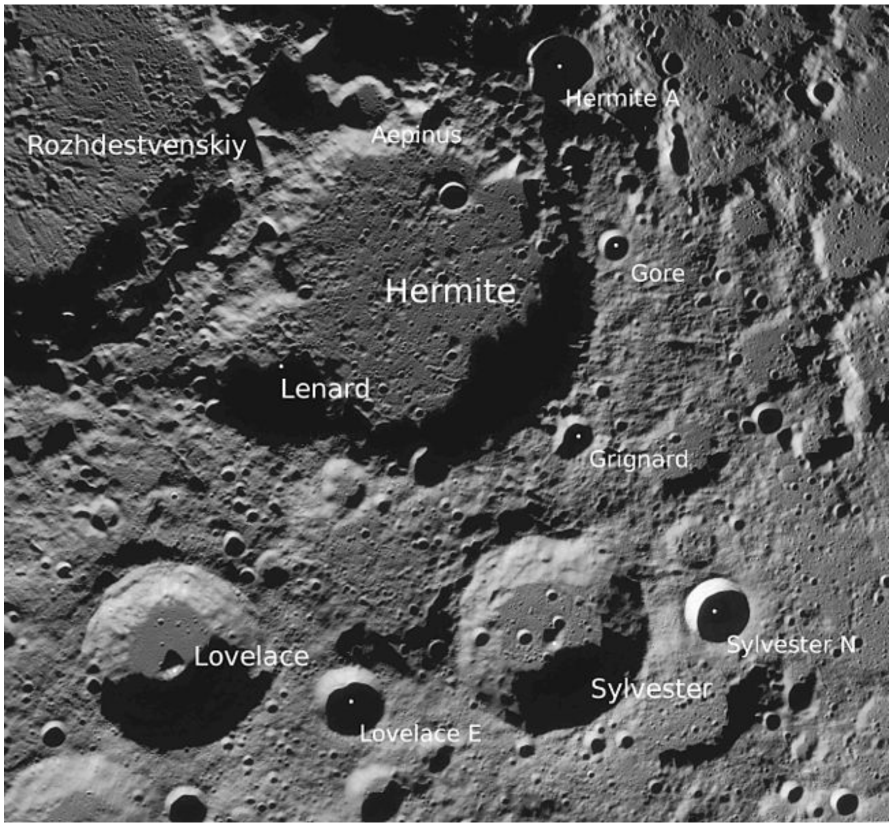

Fig. 1.2 Visual imagery of the Moon's Hermite crater, at 85 degrees north latitude (Image source: NASA / LRO Team)

In addition to the complete lack of solar energy at certain points beneath the rims of polar craters, the Moon is a rock in space with no atmosphere to trap what little heat there is or bring it in from warmer latitudes. Therefore we have long expected the Hermite crater to be very, very cold. But we didn't know precisely how cold until just the last few years.

In 2009 NASA launched a robot with a variety of instruments to map and observe the surface of the Moon. The Lunar Reconnaissance Orbiter (LRO) and its Diviner lunar radiometer have been in an eccentric polar orbit around the Moon ever since. Scientists were excited to find that Hermite crater harbors the coldest known place in the entire solar system, with surface temperatures measured as low as 26 degrees above absolute zero (26 Kelvins, $-247\,^{\circ}$C, $-413\,^{\circ}$F), which is even colder than distant Pluto [3].

The Moon is an essentially airless body, with less gas pressure on its surface than the atm ultra-high vacuum (UHV) systems used for scientific research on Earth.

Since there is no atmosphere to absorb or scatter incoming sunlight, at its equator the Moon's surface has been observed to reach temperatures as high as 399 K (+124 °C, +255 °F) [5]. Likewise, without an atmosphere to trap nighttime thermal radiation, temperatures can dip below 100 K (−173 °C, −180 °F) at the same equatorial locations. However, over the course of a lunar night, equatorial temperatures do not equilibrate with the darkness of space. If this were true, the nighttime low would be 2.7 K. Instead there is gradual nighttime cooling caused by the slow release of daytime solar heating absorbed by the subsurface, which has a climatically significant thermal inertia.

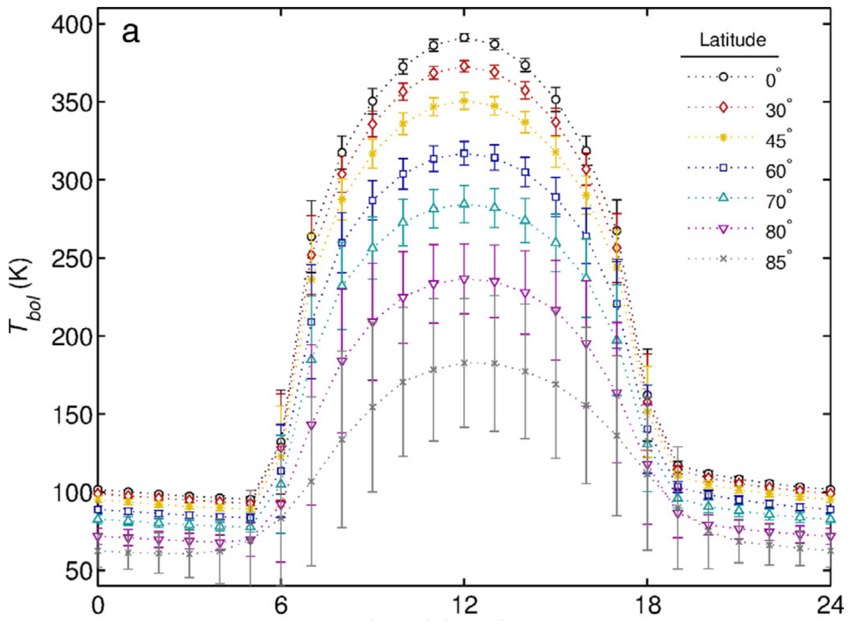

Fig. 1.3 Zonal average temperature observations for each lunar hour on the Moon at various latitudes as measured by the Diviner radiometer, aboard NASA's Lunar Reconnaissance Orbiter. Reproduced from [6].

The nights are very long on the Moon, with a synodic day of 29.53 days. In other words, if you were on the surface of the Moon, it would take 29 1/2 Earth days for the Sun to return to the same position in the sky. Nighttime is nearly two Earth weeks long. Diviner radiometry observations have shown that the Moon's surface cools rapidly in the days before sunset and then cools more slowly throughout the night until sunrise starts the diurnal temperature cycle all over again. Except in those shady places where the sun never shines, which simply stay cold all the time.

Orbiting the sun at the same average distance as Earth, the Moon is something like what our planet would be without an atmosphere, with extreme temperature swings that would be intolerable to life, at least not without extensive protection. Compare the Moon's extreme temperature range (26-399 K) to Earth's coldest in situ measured temperature of 184 K ($-89.2\,^\circ$C; $-128.6\,^\circ$F) at Vostok in Antarctica, and its hottest in situ temperature of 330 K ($+56.7\,^\circ$C; $+134.1\,^\circ$F) at Death Valley in California (WMO, 2018). To be fair, the lunar temperatures cited here weren't measured with bulb thermometers. Instead they are so-called bolometric temperatures calculated from satellite radiometer data. The calculation is based on the known relationship between the temperature of objects and the wavelengths of light that such bodies emit (Williams et al., 2017).

Similar temperature measurements have also been made on Earth. Mildrexler et al. (2011) searched through NASA's MODIS satellite radiometer dataset, part of the Earth Observing System (EOS), and found the hottest yet known land surface temperature yet: 344 K ($+70.7\,^\circ$C; $+159.3\,^\circ$F) in the Lut Desert of Iran during the summer of 2005. Scambos et al. (2018) report finding the coldest known surface temperature on Earth by the same method, 175 K ($-98.6\,^\circ$C; $-145\,^\circ$F) on the high Dome Fuji-Dome Argus region of the East Antarctic Plateau, at 82° south latitude and just over 3,800 m (12,500 ft) elevation.

On average, the Moon receives exactly the same amount of solar radiation as Earth, with a tiny amount of variation as it moves into its orbital extremes slightly closer to the Sun and slightly farther away. However, presumably because of its near-surface composition (an atmosphere, biosphere, and hydrosphere), Earth doesn't get quite as hot or nearly as cold as the Moon. These similarities and differences make the Moon a powerful, relatively simple test case for developing our physical understanding of the climate of Earth or any other planet-like body through modeling.

1.3 Simple 0-D surface energy balance

Consider a point on the surface of the Moon at the subsolar point, i.e. where the Sun is directly overhead. How hot will it get, and what controls its nighttime cooling? Unlike on Earth, where the high temperature is typically sometime in the late afternoon due to the thermal inertia of the surface and atmosphere, on an idealized airless Moon surface, the high temperature should happen at exactly lunar noon. We can define a lunar hour as 1/24 of a lunar day (lunation) whose length corresponds to Moon's synodic period of 29.53 Earth days. This means that lunar noon is six lunar hours after sunrise, the equivalent of about a week on Earth. Due to the Moon's slight axial tilt, the subsolar point isn't exactly on the equator at all times, but it isn't far.

Assuming no heat exchange with the subsurface, and that thermal equilibrium is achieved instantaneously, the radiant flux in watts per square meter received by our rock surface q_{in} and the power radiated back out to space q_{out} are equal:

$$q_{in} = q_{out} \tag{1.1}$$

Guemard (2018) and others have provided a value of the solar ”constant” for Earth (the flux of solar energy at the top of the atmosphere): $S_E = 1361.1 \pm 0.5\,Wm^{-2}$. This value is the same on average for the Moon, therefore if we assume that there is no scattering of incoming light, and that sunlight is the only energy source to the surface:

$$q_{in} = S_E \tag{1.2}$$

19th century physics (the Stefan-Boltzmann law) tells us that a body with emissivity ε will radiate energy proportional to the fourth power of its temperature T. For now we will assume that the Moon’s surface is a perfect radiator, that is, $\varepsilon = 1.00$. We can write the following expression for the power radiating outward (in W/m2) from the Moon’s surface at the subsolar point and at an equilibrium surface temperature T_s, where σ is the Stefan-Boltzmann constant $\sigma = 5.67 \times 10^{-8} Wm^{-2}K^{-4}$:

$$q_{out} = \varepsilon\sigma T_s^4 \tag{1.3}$$

Setting the input and output terms equal to each other (assuming thermal equilibrium):

$$S_E = \varepsilon\sigma T_s^4 \tag{1.4}$$

Solving for the surface temperature:

$$T_s = \left(\frac{S_E}{\varepsilon\sigma}\right)^{1/4} \tag{1.5}$$

We should verify that the units on both side of our equation correspond with each other:

$$[K] = \left(\frac{[W]m^{-2}}{[Wm^{-2}K^{-4}]}\right)^{1/4}$$

$$[K] = [\left(K^4\right)^{1/4}]$$

$$[K] = [K]$$

Inserting our data and assuming emissivity of 1.00,

$$T_s = \left(\frac{1361.1\,Wm^{-2}}{(1.00)(5.67 \times 10^{-8}\,Wm^{-2}K^{-4})} \right)^{1/4}$$

It's quite easy to perform this calculation in Python. First we fire up the Python interpreter at the command line:

```
$ python
```

A new prompt will appear, where we can type Python commands one line at a time. Amazingly, we can simply type the expression that we just wrote above for the surface equilibrium temperature:

```
>>> T_s = (1361.1/1.00/5.67e-8)**(1/4)
```

When you hit enter, Python calculates the value of T_s right away. Now we just need to display its value using the `print()` function, with `T_s` as its only argument:

```
>>> print(T_s)
```

Hitting return will give us

```
393.61962546775345
```

which expressed with three significant digits is

$$T_s = 394\,K$$

This is remarkably close to the maximum temperature of 399 K that the Diviner radiometer has measured on the Moon's surface!

What happens if we include real ballpark values for the measured reflectivity (albedo) and emissivity of the Moon's surface? Defining the normal albedo A as the fraction of insolation that is not absorbed (Bond albedo), Vasavada et al. [5] used Diviner data to calculate mean albedo values of 0.07 and 0.16 for mare and highland areas of the moon, respectively. We can use a ballpark mean of $A = 0.115$ here. Vasavada et al. also reported T_7 (IR/thermal channel 7) spectral emissivity of $\varepsilon_7 = 0.98$. With albedo A removing a fraction of insolation,

$$(1-A)S_E = \varepsilon\sigma T_s^4 \tag{1.6}$$

Solving for T_s,

$$T_s = \left(\frac{(1-A)S_E}{\varepsilon\sigma} \right)^{1/4} \tag{1.7}$$

To perform this calculation, this time let's write a Python script in a text editor such as Sublime or vim so that we can reuse and modify our code. We can save it in a

`<filename>.py` file such as `nakedplanet_simple.py` that we can run at the command line using `python nakedplanet_simple.py`. The `scipy` module includes a catalog of commonly used scientific constants, so we can include it in our code using the line `from scipy import constants`. For now we will hard-code our variable values right in this script, but later on we will store them in a separate data file for better program organization (modularity).

```
# nakedplanet_simple.py

from scipy import constants

sigma = constants.sigma
alpha = 0.115   # Mean bond albedo value (ranges from 0.07 to
    0.16) from Vasavada et al. (2012)
epsilon = 0.98  # T7 emissivity (Vasavada et al., 2012)
S_E = 1361.1    # Solar constant at Earth TOA, +/- 0.5 [W m^-2] (
    Gueymard, 2018)


T_s = ( ( ( 1 - alpha ) * S_E ) / epsilon / sigma )**(1/4)
print(T_s)
```

```
  $ python nakedplanet_simple.py
  383.7063753575507
```

$$T_s = 384\,K$$

What other sources of heat might there be to the lunar surface? These fluxes become most significant at night, so for this purpose there is no better place to think about than the eternal shade below the rim of the Hermite crater, which is at a perpetual low temperature of 26 K.

Shaded from the Sun, q_{in} for such an environment should have only two terms (which we have not yet considered for simplicity): q_g, the surface heat flux from the Moon's interior (primarily from radioactive decay), and q_c, the radiative flux from the cosmic microwave background. Siegler and Smrekar [4] used radiogenic thermal conduction models and measurements taken by the Apollo, GRAIL, and Selene missions to estimate global lunar geothermal surface heat flux values of q_g between $9-13\,mW\,m^{-2}$. Fixsen [2] calculated a cosmic background flux of $q_c = 3.13\times10^{6}\,Wm^{-2}$.

With a mean value of $q_g = 11\times10^{-3}\,Wm^{-2}$, $q_c \approx 0$ since $q_c \ll q_g$. Therefore $q_{in} \approx 11\times10^{-3}\,Wm^{-2}$ and the energy balance for regions of permanent shadow on the Moon becomes:

$$q_{in} \approx q_g = \varepsilon\sigma T_s^4 \tag{1.8}$$

Solved for T_s:

$$T_s = \left(\frac{q_g}{\varepsilon\sigma}\right)^{1/4} \tag{1.9}$$

We can perform this calculation by adding a few lines of code to our `nakedplanet_simple.py` script.

```
q_g = 11e-3     # Lunar geothermal surface heat flux [Wm-2], mean
     of 9-13e-3 Wm-2 (Siegler and Smrekar, 2014)
```

Then we modify our temperature calculations accordingly:

```
# Subsolar max temp calculation
T_s = ( ( ( 1 - alpha ) * S_E ) / epsilon / sigma )**(1/4)
print("Subsolar␣max␣temp:", T_s)

# Permanent shadow temp calculation (q_g only heat source)
T_s = ( q_g / epsilon / sigma )**(1/4)
print("Permanent␣shadow␣temp:", T_s)
```

```
$ python nakedplanet_simple.py
Subsolar max temp: 383.7063753575507
Permanent shadow temp: 21.09302465387775
```

Our ”permanent shadow temp” of 21 K is quite close to the observed coldest value of 26 K, and keeping in mind various possible values for emissivity (some materials are better radiators than others), we might interpret this calculated value as the theoretical coldest surface temperature possible anywhere on the Moon.

At this point we might as well format our `print` output to round to the correct number of significant digits. When formatting strings for the `print` function, the string format modifier `f` allows you to specify the number of digits to the right of the decimal point, for example `:.2f` would always print two digits to the right of the decimal point. However we would like the decimal point to move depending on the number of significant digits, so we can use the modifiers `g` and `G` (where ”g” stands for ”general” format). The two are the same except that `G` will use scientific notation for larger numbers, which is desirable in our case. Strictly speaking, since we are using estimates for radiogenic heat flux that have only two significant digits, we should be expressing our calculations using `:.2G`, or two significant digits. However since our temperature values are three digit numbers, this is fairly cumbersome to read. Therefore for readability we will use `:.3G` or three significant digits.

String objects in Python have a format method that can be used to modify the string according to certain parameters. To express our calculations with three digits, and indicate the units for our numbers, we can rewrite our print statements as:

```
print("Subsolar␣max␣temp:", "{:.3G}".format(T_s), "K")

print("Permanent␣shadow␣temp:", "{:.3G}".format(T_s), "K")
```

```
Subsolar max temp: 384 K
Permanent shadow temp: 22 K
```

With this we have a reasonable and simple model for the range of temperatures found on the Moon, including the coldest known place in the solar system, right in Earth's backyard.

1.4 Periodic solar forcing

A first step to simulating the diurnal temperature cycle on the surface of the Moon is to simply assume instantaneous thermal equilibrium and recalculate temperatures at each new time step.

For this we want to derive an expression for input flux to the surface as a function of time $q_{in}(t)$. On a horizontal surface at a point on the solar equator,

$$q_{in}(t) = S_E(t) + q_g + q_c \tag{1.10}$$

where q_g and q_c are the geothermal and cosmic heat fluxes, respectively. Since $S_E \gg q_g, q_c$, to a first approximation between sunrise and sunset

$$q_{in}(t) \approx S_E(t) \tag{1.11}$$

The geothermal and cosmic heating terms become more important at night, however, so for our time-dependent model we should leave them.

To simulate a time series of temperatures through a period of 24 lunar hours, we need to model how solar irradiance varies as the Moon rotates around its axis. Solar irradiance on a rotating body is a periodic function of the time.

Let Θ be the local zenith angle of the Sun (angle from the vertical) and $S_E(t)$ be the actual solar irradiance as a function of time t, measured in sidereal seconds.

$$cos\,\Theta(t) = \frac{S_E(t)}{S_E} \tag{1.12}$$

$$S_E(t) = (1-A)\,S_E\,cos\,\Theta(t) \tag{1.13}$$

We consider here a point on the moon at the solar equator. When the solar zenith angle $\Theta = 0$, the sun is directly overhead. If we start the clock in our model ($t_0 = 0$) at one of these noontime solar maxima, the next time that the sun will be overhead is P_{moon} seconds later, once the Moon has undergone a rotation of 2π. The period of the moon's rotation P_{moon} (in seconds) is the length of the lunar day in sidereal days times the number of seconds in a day: $P_{moon} = 29.53059 \times 24 \times 3600\,s = 2.55 \times 10^6\,s$.

Likewise from our perspective, the zenith angle of the Sun proceeds to $\Theta = \frac{\pi}{2}$ at sunset ($t = t_0 + 0.25\, P_{moon}$), $\Theta = \pi$ at local midnight ($t = t_0 + 0.5\, P_{moon}$), $\frac{3\pi}{2}$ at sunrise, and back to $\Theta = 2\pi$ (coterminal with $\Theta = 0$) at $t = t_0 + P_{moon}$.

If we define the hour angle $h(t)$ such that $cos\, h(t) = cos\, \Theta$, we can express the hour angle in terms of the rotational period P_{moon}:

$$h(t) = 2\pi(\frac{t}{P_{moon}}) \tag{1.14}$$

The hour angle increases linearly with time to infinity, but has the important property that at the end of each lunation, when time t is an integer multiple of P_{moon}, the value of $h(t)$ is an integer multiple of 2π.

Since $cos\, h(t) = cos\, \Theta$,

$$S_E(t) = S_E\, cos\, h(t) \tag{1.15}$$

between sunrise and sunset.

At nighttime, however, this function would yield a negative insolation value, so we need to use a function that drops to exactly zero at all angles between $\frac{\pi}{2}$ and $\frac{3\pi}{2}$. The clipping function

$$\psi(x) = \frac{1}{2}\,(cos\, x + \mid cos\, x \mid) \tag{1.16}$$

has this behavior. Therefore we can define the solar irradiance as a function of time at the solar equator as:

$$S_E(t) = (1 - A)\, S_E\, cos\, \psi(h(t)) \tag{1.17}$$

Just to make sure the sun is shining realistically on our model point on the Moon, let's use Python to plot this solar irradiance function over a couple of lunar days. We'll write our code in a Python file that we will call `solar.py`.

Very soon we will modularize our Python code into different files, objects, and functions, but still for now we will store all of our information in a single file. Our file will be organized as follows:

```
# filename: solar.py

# Import modules

# Set planetary parameters

# Set model parameters
# Create and initialize arrays

# Run time loop

# Plot calculated array values using matplotlib
```

The code below will produce two plots using matplotlib. One will show the calculated solar irradiance as a function of time in sidereal seconds, and the second will show the solar irradiance as a function of time in lunar hours. Comments are provided for explanation.

```
# solar.py

import matplotlib.pyplot as plt    # for plotting
import numpy as np    # numpy arrays for calculations and
    matplotlib plots

pi = np.pi    # so we don't have to type np.pi all the time

# Moon climate parameters

S_E = 1361.1 # Solar constant at Earth TOA, +/- 0.5 [W m-2] (
    Gueymard, 2018)
albedo = 0.115 # Mean bond albedo value (ranges from 0.07 to
    0.16) from Vasavada et al. (2012)
P_moon = 29.53059*24.*3600. #Mean length of Moon SYNODIC day [s]

Sabs = S_E * (1.0 - albedo)

# Model parameters
modelruntime = 4*P_moon
dt = 1.0 * 3600. # timestep in seconds (1.0*3600*24 = 1 earth day
    )
Nt = int(round(modelruntime/dt))    # Calculate number of
    timesteps (rounded and converted to integer)

# Create numpy arrays for calculations and plotting
t = np.zeros(Nt) # start the clock at zero
h = np.zeros(Nt) # hour angle
psi = np.zeros(Nt) # clipping function
solar = np.zeros(Nt) # array of insolation values to be
    calculated

# Create numpy array for lunar hour

lunarhour = np.zeros(Nt)

# Time loop, goes until modelruntime
for n in range(0, Nt):
        t[n] = n*dt    # Calculate the model time based on the
            current timestep number
        h[n] = 2*pi*(t[n]/P_moon) # Calculate hour angle
        psi[n] = 0.5 * ( np.cos(h[n]) + np.abs(np.cos(h[n]))) #
            Calculate clipping function
        solar[n] = Sabs * psi[n]   # Calculate solar flux

        lunarhour[n] = t[n]/P_moon*24   # Calculate lunar hour (
            for plotting only)

# Plot solar and t arrays with matplotlib
plt.scatter(t, solar)
plt.plot(t, solar)
plt.xlabel('Time␣(s)')
plt.ylabel('Solar␣irradiance␣(W/m2)')
plt.show()

plt.scatter(lunarhour, solar)
plt.plot(lunarhour, solar)
plt.xlabel('Time␣(lunar␣hours)')
plt.ylabel('Solar␣irradiance␣(W/m2)')
plt.show()
```

The output is a beautiful simulation of solar irradiance at a single point on the solar equator of the Moon over the course of four lunar days (about four Earth months). Note that the daily maxima occur at hours 24, 48, 72, and 96, and that every 12 hours starting at lunar 6 P.M. there is zero insolation for the entire nighttime. Only the second plot from solar.py (in lunar hours) is shown here.

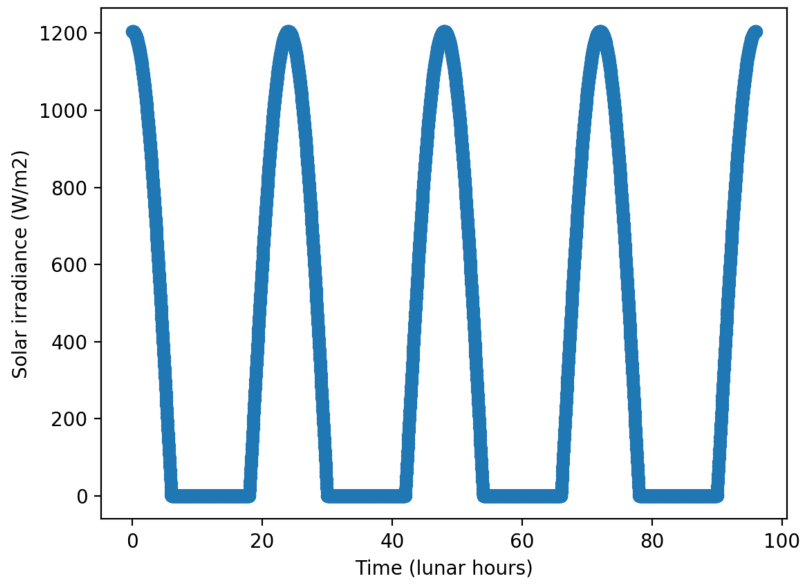

Fig. 1.4 Calculated solar irradiance at a point on the solar equator of the Moon over four lunations

If we'd like to see what just one lunar day of sunshine looks like, we can start the lunarhour array at lunar midnight:

```
lunarhour[n] = t[n]/P_moon*24 - 12
```

and tell matplotlib to show only the lunar hours from 0 to 24, with ticks at 0, 6, 12, 18, and 24:

```
plt.xlim(0, 24)
plt.xtick(0, 6, 12, 18, 24)
```

We can modify modelruntime in solar.py to show only one lunar rotational period:

```
modelruntime = 1*P_moon
```

and for comparison with the Diviner diurnal temperature observations, modify the time loop to start at lunar midnight, halfway through the first lunation (because of how we set modelruntime, it will end 24 lunar hours later). Since for loops only deal with integers, we need to typecast the starting point of our for loop as an int:

```
for n in range(0 + int(P_moon/2), Nt):
```

We don't really need the first matplotlib plot (expressed in sidereal seconds), so we can simply delete it from the script.

It is convenient to save this new modified code as solar_oneday.py. Running it at the command line:

```
python solar_oneday.py
```

provides us with the beautiful result seen in Figure 1.4.

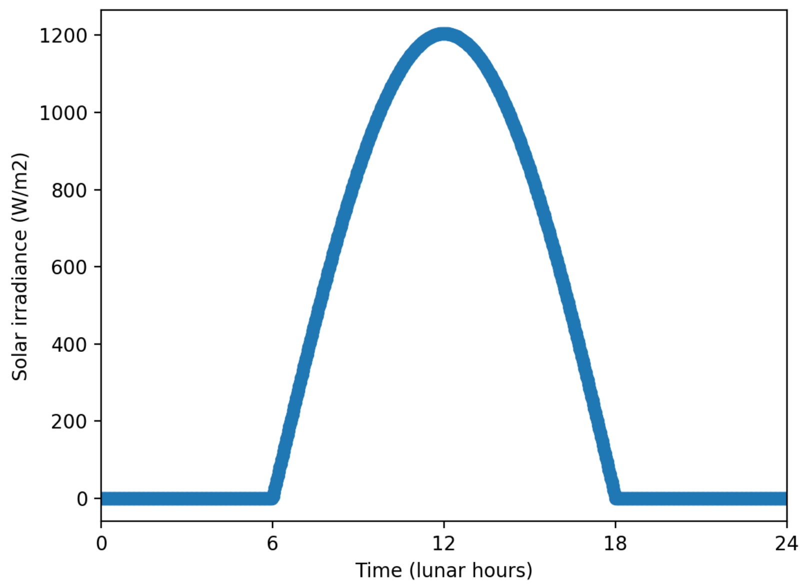

Fig. 1.5 Calculated solar irradiance at a point on the solar equator of the Moon over a single lunar day

The curve we have generated here isn't so different from the insolation curve on a clear day at any given location in the tropics on Earth: The Sun rises around six a.m., sets around six p.m., and insolation reaches a local maximum of $(1-A)S_E \approx 1200\,Wm^{-2}$ right around twelve noon. The difference here is of course that a lunar day lasts nearly an Earth month, and each lunar hour is a bit more than an Earth day long. Hopefully the analogy is clear to the reader.

Once we calculate the diurnal equilibrium temperature curve for our point at the solar equator on the Moon (neglecting soil heat fluxes), it should follow the diurnal insolation curve fairly closely.

1.5 Linking solar forcing to the 0-D surface energy balance

It won't take much to generate a time-dependent surface temperature curve from our existing solar.py module. Actually, just couple lines of Python code will do. You may have noticed that the file solar.py is already pushing past 50 lines. It works just fine the way it is, and that's not something that we want to take for granted. In the next chapter, we will look at how Git and GitHub can be used for version control, that is, for saving our working versions with comments as we go along. We'll also clean up our model's code by organizing it into modules that can be added, used, and re-used like tools in a toolbox.

But for now let's keep all our code in one file and save our working version of `solar.py` the old-fashioned, pre-Git (pre-2005) way by simply copying its contents into another file, let's call it `solar_working_notemp.py`:

```
cp solar.py solar_working_notemp.py
```

For now we'll just add our temperature code to solar.py, a file we are going to pick apart and separate into modules in the next chapter.

We can add a numpy array for surface temperature to the end of the initialization section:

```
# Create numpy arrays for calculations and plotting
t = np.zeros(Nt) # start the clock at zero
h = np.zeros(Nt) # hour angle
psi = np.zeros(Nt) # clipping function
solar = np.zeros(Nt) # array of insolation values to be
    calculated
T_s = np.zeros(Nt) # time array of surface temperatures
```

Then we can add an equilibrium temperature calculation to the time loop section. Recalling that if we neglect geothermal, soil, and cosmic heating so that insolation is the only energy input to the system and thermal radiation is the only output, the equilibrium surface temperature is:

$$T_s = \left(\frac{(1-A)S_E}{\varepsilon\sigma}\right)^{1/4} \tag{1.18}$$

We can express the time-dependence of this function with a simple change of notation:

$$T_s(t) = \left(\frac{(1-A)S_E(t)}{\varepsilon\sigma}\right)^{1/4} \tag{1.19}$$

In Python form, this can be written in solar.py as:

```
    # Calculate surface temperature given solar flux
    T_s[n] = ((1-albedo)*solar[n]/epsilon/sigma)**(1/4)
```

Be careful, however, because we haven't yet defined the variables epsilon and sigma in solar.py, so we should make sure to do this by adding the following lines in the appropriate sections above the time loop:

```
    from scipy import constants
    sigma = constants.sigma

    epsilon = 0.98      # T7 emissivity (Vasavada et al., 2012)
```

And since if we make a calculation but we don't present it graphically, it might as well never have happened at all, we should make sure to add in some matplotlib code to the appropriate section to generate a nice temperature plot.

```
plt.scatter(lunarhour, T_s)
plt.plot(lunarhour, T_s)
plt.xlabel('Time␣(lunar␣hours)')
plt.ylabel('Surface␣temperature␣(K)')
plt.ylim(-20, 400)
plt.xticks(np.arange(0, 25, step=6))
plt.show()
```

Before we run the code, what do we expect to see in our temperature plot? The total energy input to the system should be $q_{in}(t) = S_E(t) + q_g + q_c + q_{soil}(t)$. However we have neglected all heating terms except insolation so that $q_{in}(t) \approx S_E(t)$. This means that at night, since we've designed such a beautiful clipping function, there is no insolation and therefore no heating of the surface from any source. So far there is no thermal inertia in our system so we should expect the temperature to be exactly absolute zero between sunset and sunrise.

When we run the code to see our result:

```
python solar.py
```

since we have no print statements there is no output to the terminal, but a matplotlib plot does appear. The first is our plot of insolation through the day as we saw before. If we close that window, our desired temperature plot will appear.

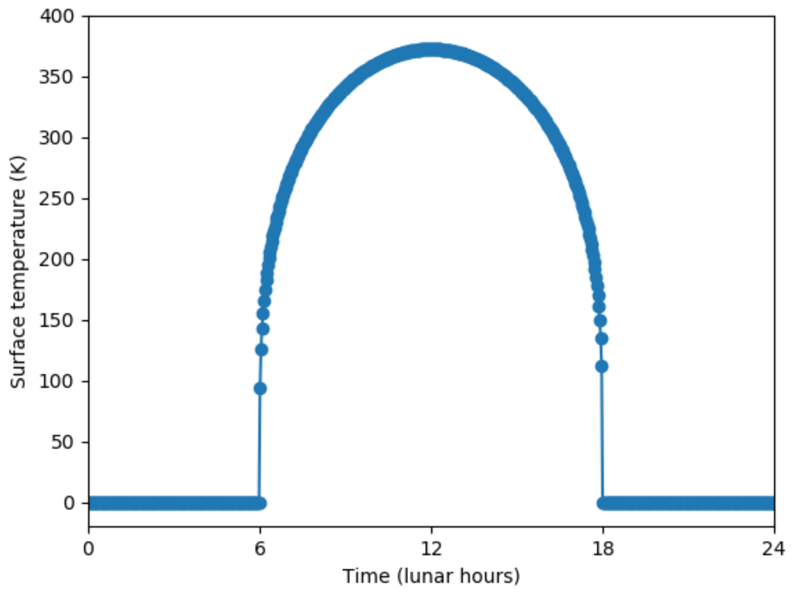

Fig. 1.6 Calculated surface temperature at a point on the solar equator of the Moon over a single lunar day

1.5.1 Improving the visual representation of data in matplotlib

We'd like to see what happens when we add in the various other heat flux terms to the energy balance. We can make our plots a little bit better by generating a text box that automatically displays the high and low temperatures (max and min). Since there's already a fair amount of visual information on the left and right margins of the plot, let's put this annotation in the upper right corner.

We'll need some background information on matplotlib to proceed with any confidence. First of all, matplotlib has two interfaces: a state-based one and an object-oriented one. For the couple of plots we've generated so far, we've taken the simplest possible route available to us: matplotlib.pyplot, which is the state-based, procedural interface for simple plotting. With pyplot, everything has to be executed in a one-off sequence, and nothing can be re-used. The advantage of pyplot's state-based interface is that it is easier to use for those unfamiliar with matplotlib.

However for any more sophisticated, publication-quality customization of plots, we have to make use of the object-oriented interface to matplotlib. This approach is a bit more nuanced but more flexible. Its building blocks are the ”Figure” object and the ”Axes” object.

These terms are important for understanding how to modify matplotlib plots. A ”Figure” in matplotlib-speak is the final rendered image that you see for example in Figure 1.5. It is composed of one or more ”Axes,” which are individual plots. As one official matplotlib tutorial states, ”The Figure is like a canvas, and the Axes is a part of that canvas on which we will make a particular visualization, [and] Figures can have multiple Axes on them” (https://matplotlib.org/tutorials/introductory/lifecycle.html). Another official explanation is: ”The Figure is the final image that may contain 1 or more Axes. The Axes represent an individual plot (don’t confuse this with the word ”axis”, which refers to the x/y axis of a plot).” (https://matplotlib.org/tutorials/introductory/lifecycle.html).

The reason this is important to know is that most modifications to a plot require modifying an ”Axes” object, almost always called ”ax” by convention. There isn’t much to be done with ”Figure” objects, called ”fig” by convention, but in order to access an ax object we call the plt.subplots() method, which returns a fig, ax tuple. So in order to make anything but the most basic customizations, we need to use the following essentially boilerplate code to access an Axes object:

```
fig, ax = plt.subplots()
```

You’ll live a lot longer if you just accept that we’ll hardly ever use the fig object, but to make pretty plots we’ll often want to play with the ax object, and figs go along with axes because of the plt.subplots() method. And be careful! plt.subplots() for this purpose has a plural ”s” on the end. plt.subplot() without an ”s” is an entirely different function.

The methods for matplotlib Axes objects are written slightly differently from the functions for pyplot/plt, so we’ll have to rewrite our plotting code accordingly.

It is a good idea to try to write this code on your own by using the matplotlib API documentation for the Axes object. The code below is the object-oriented Axes equivalent of the pyplot code we used earlier, and generates a surface temperature plot identical to that seen in Figure 1.5. We can now replace the pyplot-based code with this Axes-based code in solar.py.

```
fig, ax = plt.subplots()
ax.scatter(lunarhour, T_s)
ax.plot(lunarhour, T_s)
ax.set_xlim(0,24)
ax.set_ylim(-20,400)
ax.set_xticks(np.arange(0, 25, step=6))
ax.set_xlabel('Time␣(lunar␣hours)')
ax.set_ylabel('Surface␣temperature␣(K)')
plt.show()
```

In a sense we’re right back where we started, but this time by more sophisticated means. Now we can finally annotate our plot with the max and min temperatures as we set out to do earlier. For this we use the ax.annotate() method. Two arguments are enough to use the ax.annotate() method in this case: a string to display and the location to display the string.

```
ax.annotate('Max␣temp:', xy=(17,350))
ax.annotate('Min␣temp:', xy=(17,325))
```

This won't display any values, but will put some text in a good spot on the plot. The string is the first argument by default and is enclosed in either single or double quotes. The location can be given as a keyword argument xy=(xlocation, ylocation), where x and y are measured in terms of the data you are plotting. In the code snippet below, I place the annotations at lunar hour 17, at temperature values of 350 and 325, respectively. Other keyword argument options are available for placement of the annotation, such as by pixel coordinates, but since the numbers in this plot are nice and round, it made sense to use point coordinates.

Next we want to display a variable value. To do this we can use Python's format() method. A good user-friendly reference with examples for use of this method can be found at: https://www.geeksforgeeks.org/python-format-function/

To use format(), we insert a placeholder with curly braces inside the string literal (the text inside the quotation marks). Then we invoke the format method on the string object using the dot operator:

```
ax.annotate('Max␣temp:␣{}␣K'.format(400), xy=(17,350))
ax.annotate('Min␣temp:', xy=(17,325))
```

If we put this in our code and run it, we'll have an annotation that says "Max temp: 400" and "Min temp:" without a value for the min temp. Let's test whether we can insert a variable where we wrote "400" in the previous code snippet:

```
ax.annotate('Max␣temp:␣{}␣K'.format(Sabs), xy=(17,350))
ax.annotate('Min␣temp:', xy=(17,325))
```

This will display the value of Sabs (the solar constant reduced by the albedo) as the maximum temperature, which doesn't make much sense but at least it shows that we can use a variable as an argument to format().

It would be nice to format our value. In our current context, we don't really need any decimal places for our temperature results, so inside the curly brace placeholder we can let Python know that we would like to display a float with zero decimal places. We should try to understand this notation, since it will always come in handy for expressing our modeling results. If we wanted to express multiple variables with the same string, we could number our curly brace placeholders: 'Max temperatures: 1:.0f K, 2:.0f K'.format(temp1, temp2). In this little example, "1" indicates that it is the first variable to be displayed, and "0" is the number of decimal places to be shown for the float value. Likewise "2" in the second placeholder indicates that it is the second variable to be displayed. These index numbers aren't required, since format() will just display the variables in the order they are written by default if no indices are specified. So for display of a single float variable within a string with zero decimal places, we can leave out the index number to the left of the colon, and just write ’:.0f’.format(variable):

```
ax.annotate(’Max␣temp:␣{:.0f}␣K’.format(Sabs), xy=(17,350))
ax.annotate(’Min␣temp:’, xy=(17,325))
```

Alright, great, we’ve formatted Sabs, but that’s just a dummy variable, and we want to express the max and min temps. For this we can use numpy’s amax() and amin() methods, which will find the maximum and minimum values for given arrays. We can apply amax() to our surface temperature array `T_s`:

```
ax.annotate(’Max␣temp:␣{:.0f}␣K’.format(np.amax(T_s)), xy
    =(17,350))
```

and do the same with amin() for the minimum temperature:

```
ax.annotate(’Min␣temp:␣{:.0f}␣K’.format(np.amin(T_s)), xy
    =(17,325))
```

Lastly, as we start generating a plethora of plots, we’ll be glad when our image files have at least somewhat explanatory titles, so we should give our plot a title:

```
ax.set_title(’Solar␣heating␣only’)
```

The final result of this incremental development is a decent plot of surface temperature at the solar equator on the Moon under solar forcing for one lunation, from lunar midnight to lunar midnight:

```
ax.annotate(’Max␣temp:␣{:.0f}␣K’.format(np.amax(T_s)), xy
    =(17,350))
ax.annotate(’Min␣temp:␣{:.0f}␣K’.format(np.amin(T_s)), xy
    =(17,325))
ax.set_title(’Model␣lunar␣equatorial␣sfc␣temperature,␣solar␣
    heating␣only’)
```

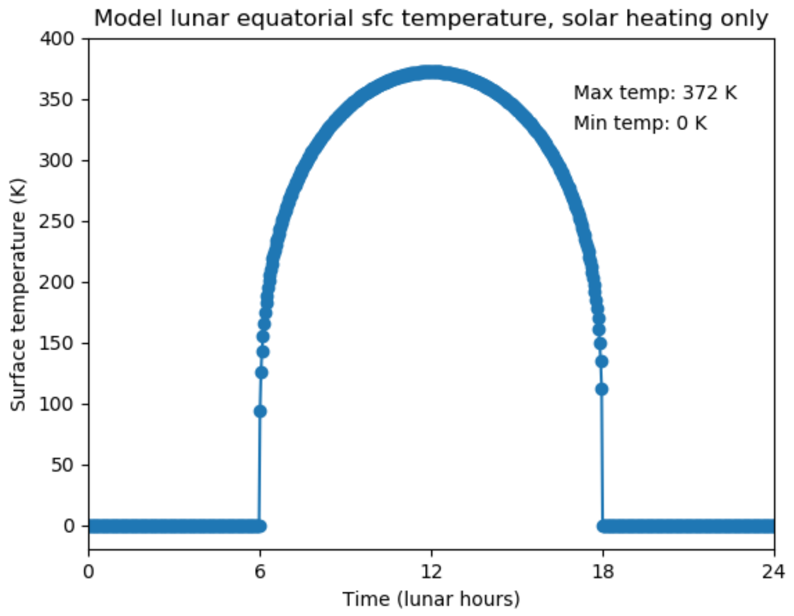

Fig. 1.7 Calculated surface temperature at a point on the solar equator of the Moon over a single lunar day, solar heating only, annotated with maximum and minimum temperatures

That annotation took a little bit of upfront work, but it will pay off now that are a bit better at effectively displaying our model results in a single visual representation. Later we should modularize our code so that we can reuse it, but for the next numerical experiments we will simply copy and paste the code we wish to modify and comment out the older versions.

1.6 Numerical experiment: Including cosmic background radiation in the surface energy balance

We can see from Figure 1.6 that our energy balance model is obviously incomplete. The nighttime temperatures are too cold (absolute zero). First of all we know that the coldest known place in the solar system (Hermite crater) doesn't even get that cold, and second we are trying to bring our model into harmony with the Diviner bolometric temperature observations, which indicate equatorial minimum temperatures around 100 K, with slow nighttime cooling. Let's see what happens when we add terms to the input side of our energy balance equation.

We know that the cosmic background radiation flux contribution to the surface energy budget is small, but does it affect the nighttime minimum temperature?

Starting with our good old equilibrium energy conservation equation, with each term expressed as a function of time:

$$q_{in}(t) = q_{out}(t) \tag{1.20}$$

in this iteration of our experiment we can add the cosmic background radiation flux by letting $q_{in}(t) = (1-A)S_E(t) + q_c$, where q_c is assumed to be constant. We can recall here Fixsen's [2] calculated cosmic background flux of $q_c = 3.13 \times 10^6 Wm^{-2}$.

Rewriting the equilibrium condition:

$$(1-A)S_E(t) + q_c = \varepsilon\sigma T_s^4 \tag{1.21}$$

and solving for the surface temperature:

$$T_s = \left(\frac{(1-A)S_E + q_c}{\varepsilon\sigma} \right)^{1/4} \tag{1.22}$$

we can see that adding terms to the input side of the energy balance can be accomplished in the temperature equation by simply adding our desired terms to the numerator. This can serve as a shortcut to full repeated derivations as we proceed.

Let's define `q_c` in solar.py:

```
# Moon climate parameters

q_c = 3.13e-6 # Cosmic background radiation flux [W m^-2] (Fixsen
    , 2009)
```

then comment out our old temperature equation in solar.py so that we can modify a copy to include q_C:

```
#T_s[n] = ((1-albedo)*solar[n]/epsilon/sigma)**(1/4) # Calculate
    surface temperature given solar flux
T_s[n] = (((1-albedo)*solar[n] + q_c)/epsilon/sigma)**(1/4) #
    Calculate surface temperature given solar flux and cosmic
    flux
```

We should also copy/comment out our plotting code and update the title in the new version (soon we will deal with this sort of code reuse more efficiently through modularization):

```
"""
fig, ax = plt.subplots()
ax.scatter(lunarhour, T_s)
ax.plot(lunarhour, T_s)
ax.set_xlim(0,24)
ax.set_ylim(-20,400)
ax.set_xticks(np.arange(0, 25, step=6))
ax.set_xlabel('Time (lunar hours)')
ax.set_ylabel('Surface temperature (K)')
ax.annotate('Max temp: {:.0f} K'.format(np.amax(T_s)), xy
    =(17,350))
ax.annotate('Min temp: {:.0f} K'.format(np.amin(T_s)), xy
    =(17,325))
ax.set_title('Model lunar equatorial sfc temperature, solar
    heating only')
plt.show()
"""

fig, ax = plt.subplots()
ax.scatter(lunarhour, T_s)
ax.plot(lunarhour, T_s)
ax.set_xlim(0,24)
ax.set_ylim(-20,400)
ax.set_xticks(np.arange(0, 25, step=6))
ax.set_xlabel('Time (lunar hours)')
ax.set_ylabel('Surface temperature (K)')
ax.annotate('Max temp: {:.0f} K'.format(np.amax(T_s)), xy
    =(17,350))
ax.annotate('Min temp: {:.0f} K'.format(np.amin(T_s)), xy
    =(17,325))
ax.set_title('Model lunar equatorial sfc temperature, solar +
    cosmic heating')
plt.show()
```

Now we can generate new model output:

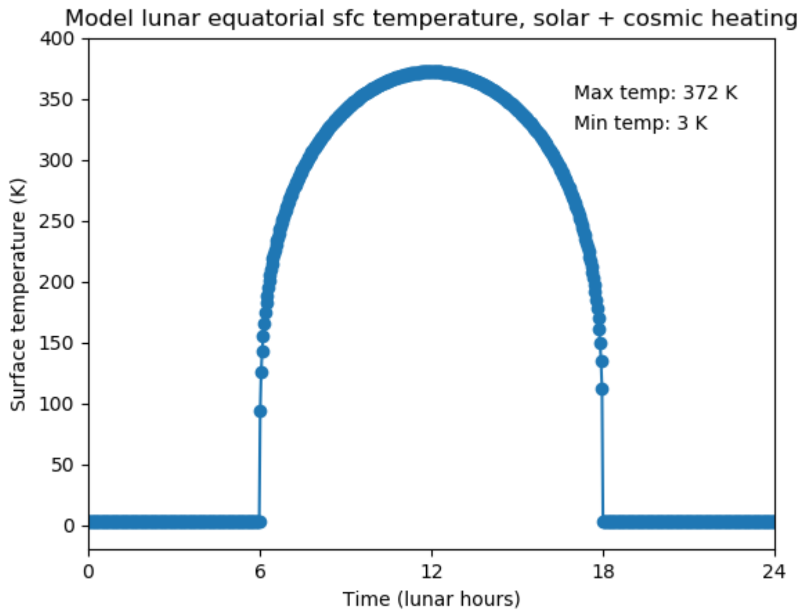

Fig. 1.8 Calculated surface temperature at a point on the solar equator of the Moon over a single lunar day, solar + cosmic heating, annotated with maximum and minimum temperatures

We recall that space is supposed to have an average temperature of 2.7 K. Is that what our model has reproduced? We can change the ax.annotate() line on our plot to show a single decimal point for the minimum temperature to see if this is the case:

```
ax.annotate('Min temp: {:.1f} K'.format(np.amin(T_s)), xy
    =(17,325))
```

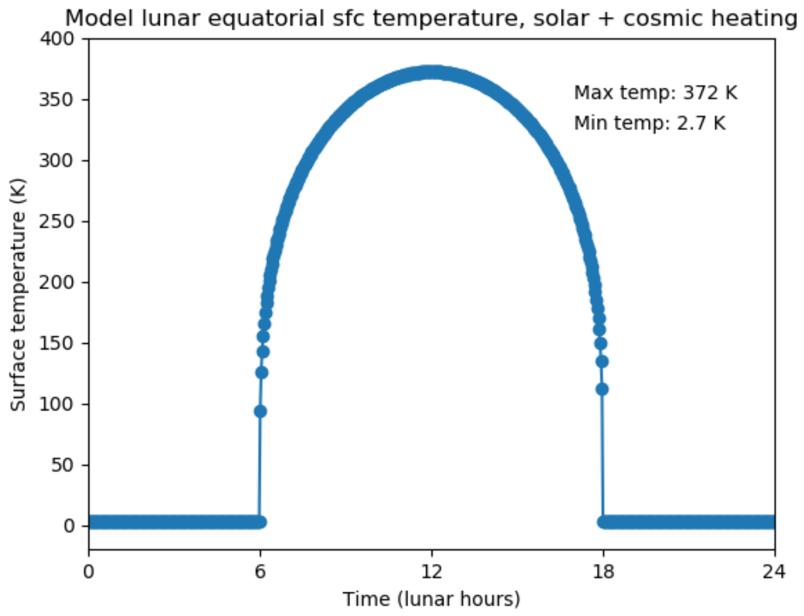

Fig. 1.9 Calculated surface temperature at a point on the solar equator of the Moon over a single lunar day, solar + cosmic heating, annotated with maximum and minimum temperatures, minimum expressed with one decimal point

Well that's certainly interesting, we've managed to simulate something closer to reality, but it's still too cold at night.

1.7 Including geothermal heating

We've already seen that all we have to do to add a term to the energy input in our temperature equation is add it to the numerator, and we can recall Siegler and Smrekar's [4] estimate of global lunar geothermal surface heat flux values q_g between $9 - 13\, mW\, m^{-2}$. Using a ballpark medium of $q_g = 11 \times 10^{-3}\, W\, m^{-2}$, we can declare this as a variable in solar.py:

```
# Moon parameters
q_g = 11e-3 # Mean lunar geothermal surface flux [W m^-2] (
    Siegler and Smrekar (2014)
```

Now we can modify the temperature calculation:

```
#T_s[n] = ((1-albedo)*solar[n]/epsilon/sigma)**(1/4) # Calculate
    surface temperature given solar flux
#T_s[n] = (((1-albedo)*solar[n] + q_c)/epsilon/sigma)**(1/4) #
    Calculate surface temperature given solar flux and cosmic
    flux
T_s[n] = (((1-albedo)*solar[n] + q_c + q_g)/epsilon/sigma)**(1/4)
     # Calculate surface temperature given solar, cosmic, and
    geothermal flux
```

and modify our plot code to make sure we have a new title and express the minimum temperature with only one decimal point:

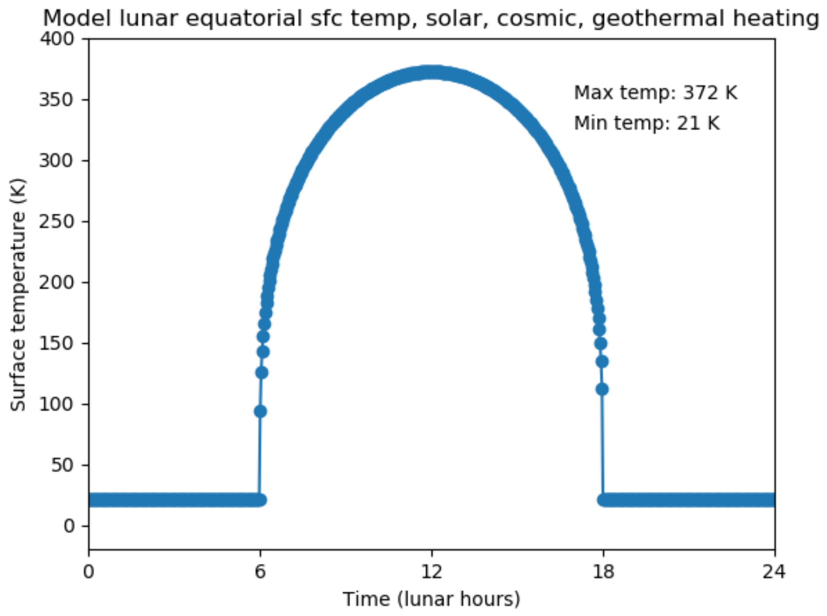

Fig. 1.10 Calculated surface temperature at a point on the solar equator of the Moon over a single lunar day, solar, cosmic, and geothermal heating, annotated with maximum and minimum temperatures

Now our nighttime temperature represents something close to the temperature we have observed in the permanent shade regions of Hermite crater, which can be understood to be controlled primarily by cosmic and geothermal heating. Notice that the max temp does not change under these varying scenarios, since insolation dwarfs other heating contributions. Still, to get our equatorial nighttime minimum to match observations, we have to account for yet another factor, the thermal inertia of the soil.

1.8 Approximating the influence of soil thermal inertia

A true dynamic calculation of the soil heat flux term requires a numerical solution of the heat conduction equation, which we will dive into in Chapter 3. However we can approximate the soil heating term q_{soil} by solving the energy balance equation given a ballpark nighttime temperature $T_s = 100\,K$.

$$q_{in} = q_{out} \tag{1.23}$$

$$q_c + q_g + q_{soil} = \varepsilon\sigma T_s^4 \tag{1.24}$$

$$q_{soil} = \varepsilon\sigma T_s^4 - q_c - q_g \tag{1.25}$$

Given that we have already laid down the values of each of these variables in solar.py, we can quickly perform this calculation by adding a line to the script:

```
# Moon climate parameters
# <other variables>

# estimate mean equatorial soil heat flux from ballpark nighttime
     temp of 100 K
q_soil = epsilon*sigma*(100**4) - q_c - q_g
print('q_soil: {:.2f}  W m-2'.format(q_soil))
```

```
q_soil: 5.55 W m-2
```

or in pretty math notation:

$$q_{soil} \approx 5.55\,W m^{-2} \tag{1.26}$$

at the solar equator.

It's a bit of a punt to just stick this number into our energy balance model, but it would be nice to see what our temperature curve looks like with a ballpark mean soil flux thrown in. We'll also update the title in our plotting code.

```
        T_s[n] = (((1-albedo)*solar[n] + q_c + q_g + q_soil)/
            epsilon/sigma)**(1/4) # Calculate surface temperature
             given solar, cosmic, and geothermal flux

        ax.set_title('Model lunar equatorial sfc temp with: S,
            q_c, q_g, q_soil')
```

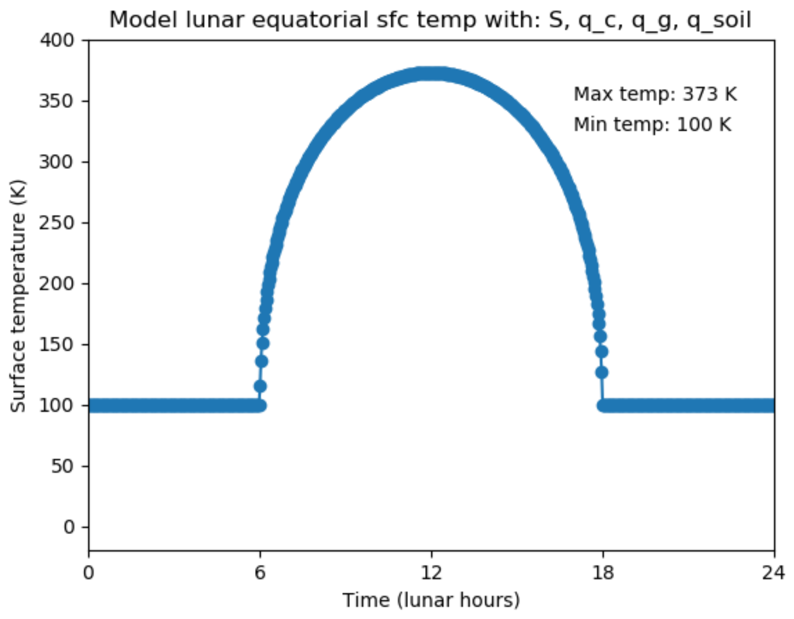

Fig. 1.11 Calculated surface temperature at a point on the solar equator of the Moon over a single lunar day, solar, cosmic, geothermal, and estimated soil heating, annotated with maximum and minimum temperatures

Considering the fact that we are using real data and real physics (except for the soil heat flux estimate), this is a pretty satisfactory approximation of the Diviner observed diurnal temperature curve at the solar equator. The only way we will be able to improve upon this model is to 1) clean up our code through modularization and version control (introduced in Chapter 2) and 2) model the conduction of solar energy into and out of the soil, providing our model with dynamic thermal inertia (which we will do in Chapter 3).

References

[1] Arya, A. S., Rajasekhar, R. P., Thangjam, G., Gopala Krishna, B. and Kiran Kumar, A. S. [2011]. Surface age and morphology of Hermite crater of lunar North Pole using high resolution datasets, *EPSC-DPS Joint Meeting 2011*, p. 1850.

[2] Fixsen, D. J. [2009]. The temperature of the cosmic microwave background, *The Astrophysical Journal* **707**(2): 916.

[3] Freeberg, A. and Jones, N. [2009]. Goddard Release No. 09-87. retrieved 11 Dec 2018.
URL: *https://www.nasa.gov/mission$_p$ages/LRO/news/agu − results − 2009.html*

[4] Siegler, M. A. and Smrekar, S. E. [2014]. Lunar heat flow: Regional prospective of the Apollo landing sites, *Journal of Geophysical Research: Planets* **119**(1): 47–63.

[5] Vasavada, A. R., Bandfield, J. L., Greenhagen, B. T., Hayne, P. O., Siegler, M. A., Williams, J.-P. and Paige, D. A. [2012]. Lunar equatorial surface temperatures and regolith properties from the Diviner Lunar Radiometer Experiment, *Journal of Geophysical Research: Planets* **117**(E12).

[6] Williams, J.-P., Paige, D. A., Greenhagen, B. T. and Sefton-Nash, E. [2017]. The global surface temperatures of the Moon as measured by the Diviner Lunar Radiometer Experiment, *Icarus* **283**: 300–325.

Chapter 2
Naked planet model with soil thermodynamics

2.1 Lunar regolith thermodynamic model

To simulate a transient transfer of energy into soil with enough thermal inertia to cause gradual nighttime cooling, we need to describe the thermodynamics of the lunar regolith. As a general rule, ”rocky, coherent surfaces and blocks with higher thermal inertia provide larger reservoirs of heat and remain warmer than the pulverized, fine-grained regolith” (Williams et al., 2017). Here we will initially limit ourselves to regolith to build our model.

Heat conduction into (and back out of) the lunar regolith is a nonlinear process that requires finite difference schemes to model through time. The thermodynamic definition of temperature is that it is a quantity proportional to the mean kinetic energy (or thermal energy) of the particles at some point. What we seek is a function $T(z,t)$ that describes the temperature T as it changes through time t and at various depths z in the soil column.

2.2 Derivation of 1-D heat conduction equation

Energy conservation for a small volume of lunar ground V can be described as

$$E_{soil} = \int_V \rho e \, dV \tag{2.1}$$

where E_{soil} is the total (extensive) energy of the volume of soil, ρ is the volumetric mass density, e is the energy per unit volume (intensive), and the integral is over the entire volume.

If there are no heating sources within the volume (radiogenic heating is only significant on a much larger scale, and assuming no phase changes within the regolith matrix), we can express the rate of change of the total energy of the volume (in J/s) as

$$\frac{dE_{soil}}{dt} = \int_V \rho \frac{\partial e}{\partial t} dV \tag{2.2}$$

The l.h.s. expresses the Lagrangian or total derivative for the system as a whole, and the r.h.s. includes the Eulerian or local derivative for each point within the system.

Since $\frac{\partial e}{\partial t} = c\frac{\partial T}{\partial t}$, where c is the specific heat,

$$\frac{dE_{soil}}{dt} = \int_V \rho c \frac{\partial T}{\partial t} dV \tag{2.3}$$

Letting $\dot{Q}$ be the rate at which energy enters the volume (rate of heat transfer), another expression of energy conservation would be

$$\frac{dE_{soil}}{dt} = \dot{Q} \tag{2.4}$$

Assuming no internal heat source, the rate of heat transfer in turn can be written in vector form as

$$\dot{Q} = -\int_A \mathbf{q}'' \cdot \mathbf{n}\, dA \tag{2.5}$$

where $\mathbf{q}''$ is the heat flux vector, $\mathbf{n}$ is the normal outward surface vector of the surface element dA, and the area integral is over the surface area of the system.

The divergence theorem allows us to transform the area integral into a volume integral:

$$\int_A \mathbf{q}'' \cdot \mathbf{n}\, dA = \int_V \nabla \cdot \mathbf{q}'' dV \tag{2.6}$$

Now we can write another expression for energy conservation in terms of the heat flux and temperature:

$$\int_V \rho c \frac{\partial T}{\partial t} dV = -\int_V \nabla \cdot \mathbf{q}'' dV \tag{2.7}$$

$$\int_V \rho c \frac{\partial T}{\partial t} dV + \int_V \nabla \cdot \mathbf{q}'' dV = 0 \tag{2.8}$$

$$\int_V \left(\rho c \frac{\partial T}{\partial t} + \nabla \cdot \mathbf{q}'' \right) dV = 0 \tag{2.9}$$

$$\rho c \frac{\partial T}{\partial t} + \nabla \cdot \mathbf{q}'' = 0 \tag{2.10}$$

Here we make use of Fourier's law of heat conduction (cf. Ingersoll and Zobel, 1913: "When different parts of a solid body are at different temperatures, heat flows from the hotter to the colder portions by a process of transference probably from molecule to molecule known as conduction.")

$$\mathbf{q}'' = -k \nabla T \tag{2.11}$$

where $\mathbf{q}''$ is the heat flux vector, k is mean thermal conductivity of the slab in $Wm^{-2}K^{-1}$, and ∇T is the gradient of the temperature field: $\nabla T = \frac{\partial T}{\partial x}\mathbf{i}, \frac{\partial T}{\partial y}\mathbf{j}, \frac{\partial T}{\partial z}\mathbf{k}$. Here z is zero at the surface and increases downward. The negative sign in Fourier's law indicates that the heating flows down the temperature gradient, from hotter to colder regions.

Neglecting horizontal heat flows, Fourier's law reduces to a vertical gradient:

$$\mathbf{q}'' = -k \frac{\partial T}{\partial z} \mathbf{i} \tag{2.12}$$

Now using this reduced form of Fourier's law, we can rewrite the energy conservation equation as

$$\rho c \frac{\partial T}{\partial t} - (\nabla \cdot k \frac{\partial T}{\partial z} \mathbf{i}) = 0 \tag{2.13}$$

$$\rho c \frac{\partial T}{\partial t} - (\nabla \cdot k \frac{\partial T}{\partial z} \mathbf{i}) = 0 \tag{2.14}$$

$$\rho c \frac{\partial T}{\partial t} - \frac{\partial}{\partial z} \left(k \frac{\partial T}{\partial z} \right) = 0 \tag{2.15}$$

which gives us the one-dimensional soil heat conduction equation (HCE)

$$\rho c \frac{\partial T}{\partial t} = \frac{\partial}{\partial z} \left(k \frac{\partial T}{\partial z} \right) \tag{2.16}$$

In canonical form, the HCE is written as

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial z^2} + f \tag{2.17}$$

where α is the thermal diffusivity in units of m^2/s, defined as $\alpha = \frac{k}{\rho c}$, and f represents an internal source of heat. Heating from decay of radioactive elements is negligible for small volumes on the Moon (but significant on the global scale, as we have seen from our surface temperature calculations that require a geothermal flux q_g) so we assume $f \approx 0$. The parameter α is used as part of the stability criterion for numerical solutions of this class of equations.

Although this is a canonical Initial Value Boundary Problem (IVBP) with a parabolic partial differential equation (PDE), the quantities ρ, c, and k vary throughout the subsurface as a function of depth and temperature. The resulting nonlinearity excludes an analytical solution. Since the HCE derived above has a first order derivative in time, its solution, whether analytical or numerical, requires an initial condition, which in this case we can take to be the temperature field at time zero. The second order derivative in space makes two boundary conditions necessary, essentially at the top and at the bottom of the soil column. As we derive our first numerical solution, however, we will hold the thermal diffusivity α constant for simplicity.

2.3 Numerical solution (discretization and finite difference scheme)

There are many details involved in solving the heat conduction equation for lunar regolith, so it can be helpful to idealize the case to work out the essential details and add in complicating factors step by step so that our model provides an incrementally better and better simulation of reality.

There are a variety of numerical schemes that are available for solving a second-order PDE. The study of numerical methods is an academic field unto itself that lies at the intersection of applied mathematics, science, and engineering. Practice has shown that certain schemes have greater stability for the class of PDEs, in one state variable plus time, than others. In other words a stable solution minimizes error and converges realistic solutions. Here we will use the Crank-Nicholson finite differencing algorithm, a form of forward Euler time integration (also known as explicit timestepping). It is one of the most popular methods in practice.

We first need to discretize our variables by mapping the continuous state function (in our case $T(z,t)$) onto a grid or mesh of discrete z,t values over our domain of interest.

$$z_i = i\Delta z \quad i = 0, \dots, N_z \tag{2.18}$$

$$t_n = n\Delta t \quad n = 0, \ldots, N_t \tag{2.19}$$

where N_z is the number of vertical layers, N_t is the number of timesteps, i is the layer index, n is the timestep index, and Δz, Δt represent the uniform discrete spacing between gridpoints in z,t space. As we will soon see, there are reasons to set grid spacing at each layer dynamically (for example, with geometrically increased grid spacing based on the penetration depth of the diurnal temperature wave in the soil column). For initial simplicity and understandability we will use a uniform grid with constant Δz. $T_i^{(n)}$ is then taken to approximate the value of the temperature function in grid space, $T(z_i,t_n)$.

We are now equipped to rewrite the HCE in uniform grid space:

$$\frac{\partial}{\partial t}T(z_i,t_n) = \alpha \frac{\partial^2}{\partial z^2}T(z_i,t_n) \tag{2.20}$$

Later we will find that α also varies in time and space in lunar regolith, but for simplicity we will hold it constant as we first build our model. Using a forward difference in time and a central difference in space, we can write our HCE as

$$\frac{T_i^{(n+1)} - T_i^{(n)}}{\Delta t} = \alpha \frac{T_{i+1}^{(n)} - 2T_i^{(n)} + T_{i-1}^{(n)}}{(\Delta z)^2} \tag{2.21}$$

which is merely an algebraic equation which can be solved for our desired unknown. $T_i^{(n)}$ is the temperature at the current (spatial) gridpoint i and the current timestep n. The value of T at the same gridpoint but at the next timestep is $T_i^{(n+1)}$. Solving the discrete HCE for $T_i^{(n+1)}$ gives us

$$T_i^{(n+1)} = T_i^{(n)} + \alpha \frac{T_{i+1}^{(n)} - 2T_i^{(n)} + T_{i-1}^{(n)}}{(\Delta z)^2}\Delta t \tag{2.22}$$

One way to interpret each new temperature is that it is calculated by adding its old value to a weighted average of the temperature values surrounding it. This fits with the physical reality of a thermal diffusion model.

In choosing our grid spacing (and timestep duration), for the solution to be stable we need to make use of a stability criterion F based on the mesh's Fourier number. We define F as

$$F \equiv \alpha \frac{\Delta t}{(\Delta z)^2} \tag{2.23}$$

where a stable solution has the following stability criterion:

$$F \leq 0.5 \tag{2.24}$$

If it weren't for the fact that α, the thermal diffusivity, depends on a number of variables (temperature and density, in particular) in real soil, we'd be ready to simply implement (2.22) in some Python code and fire up a simulation of just another day on the Moon. That is, if things were easy and thermal diffusivity were a universal constant, we could just set α to some fixed value. If you've got some spare time you could try that out as an exercise, but what you might find is that unless α changes with temperature, your soil layers won't conduct heat with each other quickly enough when things start getting cold on the surface at sunset. This can make your code break in strange ways.

2.4 Nonlinear soil thermodynamic properties

In real environments, thermal diffusivity $\alpha = \frac{k}{\rho c}$, with units of $[m^2/s]$, depends upon both temperature and depth. Obviously with increasing overburden, ρ should increase with depth, and conductivity can be expected to change as well with changing density.

One recent standard textbook on partial differential equations (Olver, 2014) calls the heat conduction equation (more generally referred to as the diffusion equation) with variable diffusivity a "much thornier nonlinear diffusion problem" than the constant diffusivity case.

Before we tackle the top and bottom boundary conditions in the next sections, we can derive expressions for calculating soil properties (including temperature) at the inner mesh points.

Assuming no internal heating, the heat conduction equation can be written as

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial z^2} \tag{2.25}$$

or more specifically,

$$\frac{\partial T}{\partial t} = \frac{k}{\rho c} \frac{\partial^2 T}{\partial z^2} \tag{2.26}$$

where once again k is the thermal conductivity with units of $Wm^{-2}K^{-1}$, ρ is the mass density with units of $kg\,m^{-3}$, and c is the specific heat capacity with units of $J\,kg^{-1}\,K^{-1}$.

If we make use of Fourier's law in one dimension: $q = -k\frac{\partial T}{\partial z}$,

$$\frac{\partial T}{\partial t} = \frac{1}{\rho c}\frac{\partial q}{\partial z} \tag{2.27}$$

Discretizing the time step,

$$\Delta T = \frac{\Delta t}{\rho c}\frac{\partial q}{\partial z} \tag{2.28}$$

Using a forward difference formula to approximate the heat flux across each layer q_i,

$$q_i \approx k_i \frac{T_{i+1} + T_i}{\Delta z} \tag{2.29}$$

so that

$$T_i^{(n+1)} = T_i^{(n)} + \frac{\Delta t}{\rho_i c_i}\frac{\partial}{\partial z} q_i \tag{2.30}$$

since

$$\frac{\partial}{\partial z} q_i \approx \frac{q_i - q_{i-1}}{\Delta z} \tag{2.31}$$

$$\frac{\partial}{\partial z} q_i \approx k_i \frac{T_{i+1} - T_i}{\Delta z} - k_{i-1}\frac{T_i - T_{i-1}}{\Delta z} \tag{2.32}$$

$$T_i^{(n+1)} = T_i^{(n)} + \frac{\Delta t}{\rho_i c_i}\left\{ k_i \frac{T_{i+1} - T_i}{\Delta z} - k_{i-1}\frac{T_i - T_{i-1}}{\Delta z} \right\} \tag{2.33}$$

The soil parameters from (2.33) at each discrete level i, ρ_i, k_i, and c_i, all need to be calculated. Here I directly use the empirical equations discussed by Hayne et al. 2017.

Density at each depth is given by:

$$\rho_i = \rho_d - (\rho_d - \rho_s)e^{-z/H} \tag{2.34}$$

where H is a scale parameter $H \approx 0.06\,m$ (Hayne et al. 2017). The resulting calculated soil density profile is shown in Figure 2.1.

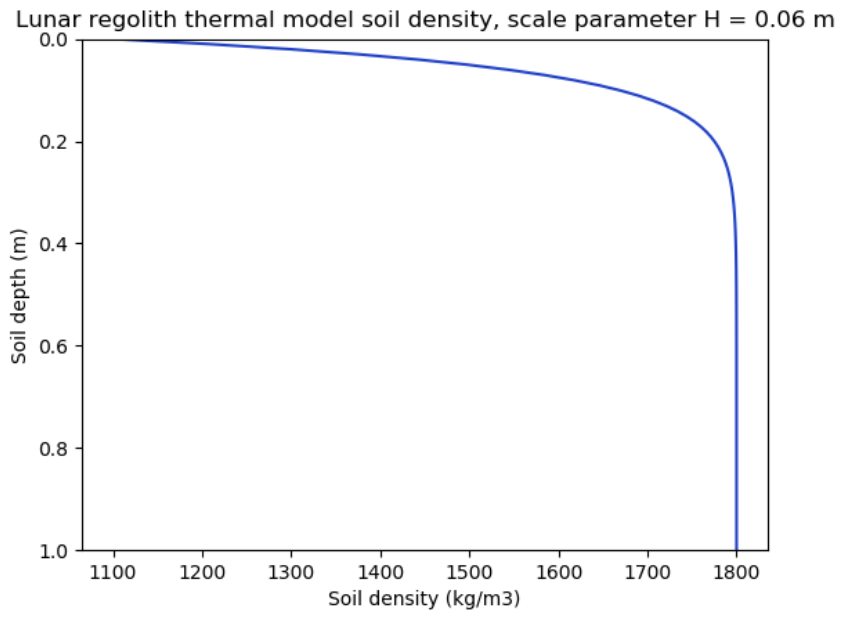

Fig. 2.1 Calculated soil density profile with scale parameter H set to 0.6 m

The contact conductivity is given by:

$$k_{C,i} = k_d - (k_d - k_s)\frac{\rho_d - \rho_i}{\rho_d - \rho_s} \tag{2.35}$$

which can in turn be used to calculate the overal thermal conductivity taking into account internal radiative fluxes:

$$k_i = k_{C,i}\left[1 + \chi\left(\frac{T_i}{350\,K}\right)^3\right] \tag{2.36}$$

where χ is a radiative conductivity parameter, taken to be $\chi = 2.7$ by Hayne et al. 2017 and Vasavada et al. 2012.

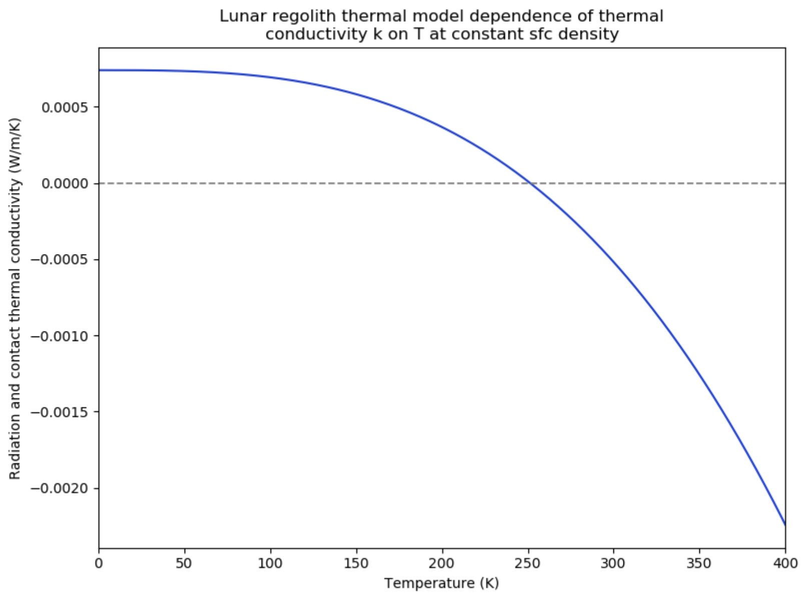

Fig. 2.2 Dependence of model thermal conductivity on temperature, at surface density

The fact that the resulting calculated thermal conductivity k is negative for temperatures greater than about 250 K suggests that this is a model-tuned parameter and not physically realistic (thermal conductivity cannot be negative). The equations modeling this parameter should be scrutinized further. However we can draw the generalization that thermal conductivity decreases with increasing temperature in the model.y

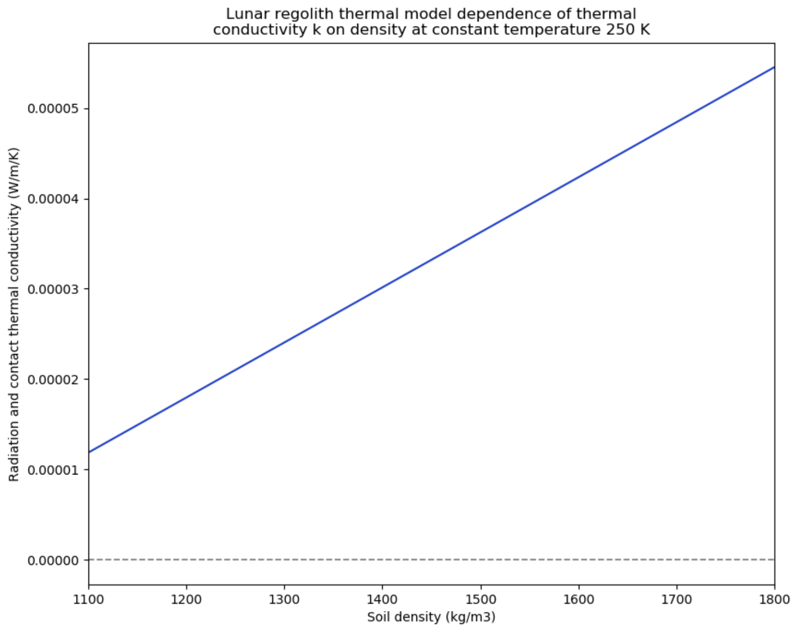

Fig. 2.3 Dependence of model thermal conductivity on density, at constant temperature T = 250 K

The specific heat of lunar regolith c_i at each gridpoint i is calculated as an empirical function of the gridpoint's temperature T_i:

$$c_i = c_0 + c_1 T_i + c_2 T_i^2 + c_3 T_i^3 + c_4 T_i^4 \tag{2.37}$$

where the coefficients $c_0 \ldots c_4$ are taken from Hayne et al. 2017:

c_0	$-3.6125\, J\, kg^{-1}\, K^{-1}$
c_1	$+2.7431\, J\, kg^{-1}\, K^{-1}$
c_2	$+2.3616 \times 10^{-3}\, J\, kg^{-1}\, K^{-1}$
c_3	$-1.2340 \times 10^{-5}\, J\, kg^{-1}\, K^{-1}$
c_4	$+8.9093 \times 10^{-9}\, J\, kg^{-1}\, K^{-1}$

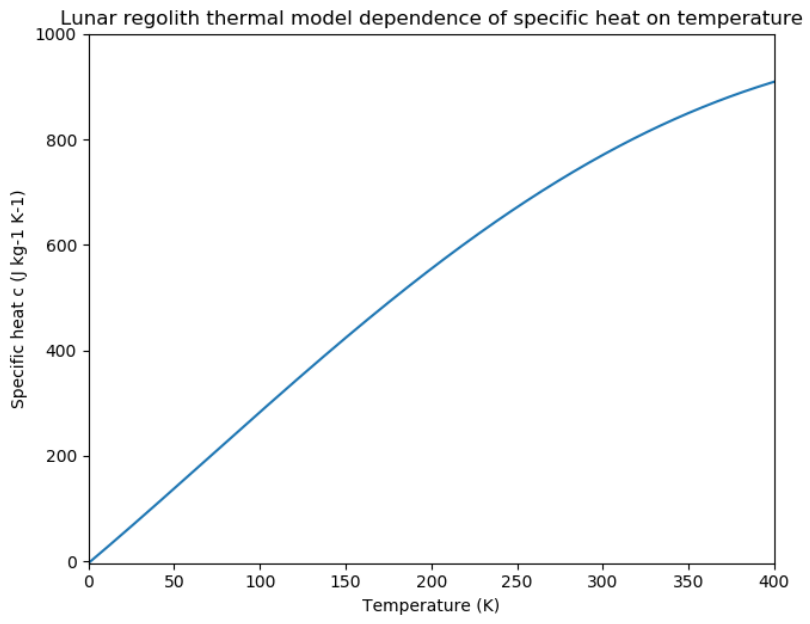

Fig. 2.4 Dependence of model specific heat on temperature

2.5 Overview of boundary conditions

There are generally three options available for defining boundary conditions for PDEs. Here I will describe them in terms of the heat conduction equation.

<u>Neumann boundary condition (simple diffusion)</u>

The system exchanges energy in strict proportionality to the temperature gradient. Only the gradient of the temperature field across the boundary must be specified. A special case of the Neumann boundary is the adiabatic (thermal isolation) case, where the gradient is zero and there is no energy exchange.

<u>Dirichlet boundary condition (equilibrium)</u>

The system exchanges energy instantaneously with its surroundings, so that both sides of the boundary are at thermal equilibrium and thus have the same temperature. The value of the solution must be specified.

<u>Robin boundary condition (nonlinear diffusion)</u>

The system exchanges some energy with its surroundings with a nonlinear dependence on both the temperature gradient and the temperature value. Information regarding both the value and the gradient at the boundary must be specified.

Lunar regolith can be considered to have a Robin boundary condition at the surface which ”leaks” soil column energy to the surroundings by means of thermal radiation according to its temperature. Solar energy also passes downward through the boundary by means of conduction as well as some radiation that penetrates into the spaces between grains.

Below a certain depth, the regolith is no longer sensitive to diurnal temperature swings, is at equilibrium with its surroundings and thus has the same temperature as its most immediate surroundings. This is a Dirichlet condition.

2.5.1 Soil thermal model - lower boundary (Dirichlet)

The simplest way to express thermal equilibrium is to simply let the temperature of the bottom layer be equal to the temperature in the environment just below the bottom layer such that $T(z=D,t)=T^*_{bot}$ at the bottom boundary. This is a Dirichlet boundary condition, where T^* indicates a temperature just outside of the system.

2.5.2 Soil thermal model - upper boundary (Robin)

The energy balance at the surface is the familiar

$$q_{in} = q_{out} \tag{2.38}$$

where (just to be clear) q_{in} is the heat flux into the system (our point on the surface) and q_{out} is the heat flux out. The only inputs are the solar flux q_{solar}, the cosmic background heat flux q_c, and the ”geothermal” heat flux from the interior of the planet q_g. As we’ve already discussed, the maximum values of q_{solar} ($> 10^3\,W\,m^{-2}$) are orders of magnitude greater than the $10^{-3}\,W\,m^{-2}$ cosmic background flux q_c. The cosmic background radiation is sufficient to keep nighttime temperatures on an airless body above absolute zero, but only to the $2.7\,K$ ambient temperature of space. As we saw from our calculations of the energy balance in regions of permanent shadow on the Moon, the ”geothermal flux” or heat flux from the interior of the Moon q_g (essentially from radioactive decay) is sufficient to bring the Moon’s minimum surface temperature up to a modeled $21\,K$ and an observed $26\,K$. In this particular model, since we have chosen to express the lower boundary condition as a Dirichlet condition with thermal equilibrium, the geothermal flux is accounted for by the constant bottom layer temperature.

However to approximate the observed equatorial surface minimum temperature of $100K$ we need to invoke a "soil" or regolith heat flux q_{soil} which represents the slow return of absorbed solar energy from the subsurface. Assuming a nighttime temperature of $100\,K$, we estimated a value of $q_{soil} \approx 5.5\,W\,m^{-2}$.

Now that we've got the picture, we can improve our previous approximations that involved a static, constant q_{soil} and use it in our dynamic soil model as a conduit between the subsurface and the surface. Something that is important to recognize is that during the daytime, q_{soil} could actually be negative, representing a loss of energy from the surface to the colder subsurface. Likewise when the surface gets cold, as it does very rapidly as the sun goes down, q_{soil} has a positive value in the sense that it contributes positive energy to the surface from the warmer layers below.

In terms of our input and output energy balance model $q_{in} = q_{out}$, it's very easy to express the input of energy to the surface "system":

$$q_{in} = q_{soil} + q_{solar} + q_{cosmic} \tag{2.39}$$

which we can simplify if we like to

$$q_{in} = q_{soil} + q_{surface} \tag{2.40}$$

where the $q_{surface}$ term simply represents all energy reaching the surface from above $(q_{surface} = q_{solar} + q_{cosmic})$.

Fourier's law in one dimension helps us to rewrite q_{soil} in terms of the vertical temperature gradient:

$$q_{soil} = -k \frac{\partial T}{\partial z}\bigg|_{z=0} \tag{2.41}$$

where the negative sign indicates that heat will only flow in the direction of positive z if the temperature decreases with increasing z (downward).

And so to be explicit, the input term to the surface energy balance is

$$q_{in} = -k \frac{\partial T}{\partial z}\bigg|_{z=0} + q_{surface} \tag{2.42}$$

Meanwhile the output term to the surface energy balance is radiative (according to the Stefan-Boltzmann relation):

$$q_{out} = \varepsilon \sigma T_s^4 \tag{2.43}$$

Putting this all together, we have the Robin radiative boundary condition

$$-k \frac{\partial T}{\partial z}\bigg|_{z=0} + q_{surface} = \varepsilon \sigma T_s^4 \tag{2.44}$$

which if we rewrite as

$$\varepsilon \sigma T_s^4 + k \frac{\partial T}{\partial z}\bigg|_{z=0} - q_{surface} = 0 \tag{2.45}$$

we can recognize as a fourth order polynomial equation.

2.5.3 Numerical solution of radiative boundary condition

Our goal is to model the temperature of the surface of the Moon in response to periodic solar forcing and the dynamic effects of soil thermal inertia. We have already derived expressions for the temperature $T_i^{(n+1)}$ in layer i at a new timestep $n+1$ at the bottom layer and at the internal points within the soil mesh. Now comes the trickiest part of our numerical model, solving for the surface temperature T_s.

Direct methods for finding the roots of fourth degree polynomials are difficult to use (Iyengar and Jain, 2009). The standard reference for numerical computation in the sciences (Press et al., 2007) states: ”There are no good, general methods for solving systems of more than one nonlinear equation [and] there never will be.”

Sounds like a meaningful challenge to me!

Our first step is to discretize the continuous Robin radiative boundary condition we derived in (2.44).

If we make use of a three-point numerical scheme to approximate the derivative in the diffusion term,

$$\frac{\partial T}{\partial z} \approx \frac{-3T_0 + 4T_1 - T_2}{2\Delta z} \tag{2.46}$$

where T_0 represents the temperature in layer 0 (the surface temperature), we can rewrite (2.44) as:

$$\varepsilon \sigma T_0^4 + k_0 \frac{-3T_0 + 4T_1 - T_2}{2\Delta z} - q_{surface} = 0 \tag{2.47}$$

Note that the value of the thermal conductivity k_i is specific to each particular layer. Therefore here we are referring to k_0, the current top layer (surface) thermal conductivity. Here we can make use of (2.36):

$$k_i = k_{C,i}\left[1+\chi\left(\frac{T_i}{350\,K}\right)^3\right] \tag{2.48}$$

which by substitution into (2.47) yields:

$$\varepsilon\sigma T_0^4 + k_{C,0}\left[1+\chi\left(\frac{T_0}{350\,K}\right)^3\right]\frac{-3T_0+4T_1-T_2}{2\Delta z} - q_{surface} = 0 \tag{2.49}$$

Expanding the second term:

$$\left(k_{C,0} + k_{C,0}\chi\left(\frac{T_0}{350\,K}\right)^3\right)\left(\frac{-3T_0+4T_1-T_2}{2\Delta z}\right) \tag{2.50}$$

$$\left(k_{C,0} + k_{C,0}\chi\left(\frac{T_0}{350\,K}\right)^3\right)\left(\frac{-3T_0}{2\Delta z} + \frac{4T_1-T_2}{2\Delta z}\right) \tag{2.51}$$

*Is this actually correct? In case you can't quite remember exactly what works and what doesn't when decomposing fractions, you can test the decomposition we use here with a little Python script. The question is whether
$\frac{-3T_0+4T_1-T_2}{2\Delta z}$ is equivalent to $\frac{-3T_0}{2\Delta z} + \frac{4T_1-T_2}{2\Delta z}$. In the Python interpreter you can check this for plausible but random T values:

```
>>> import random
>>> T0 = random.random() * 400
>>> T1 = random.random() * 400
>>> T2 = random.random() * 400
>>> T0
10.397014739021326
>>> T1
68.84833555806256
>>> T2
186.66411573424267
>>> dz = 0.01
>>> left = (-3*T0 + 4*T1 -T2)/2/dz
>>> right = -3*T0/2/dz + (4*T1 - T2)/2/dz
>>> left
2876.9091140471787
>>> right
2876.909114047179
```

Now by the distributive property (FOIL/First Outside Inside Last... turns out middle school math is actually good for something: numerical methods for planetary climate modeling!), (2.51) is expanded to become something wildly complicated:

$$\left(k_{C,0} + k_{C,0}\chi\left(\frac{T_0}{350\,K}\right)^3\right)\left(\frac{-3T_0}{2\Delta z} + \frac{4T_1 - T_2}{2\Delta z}\right) \tag{2.52}$$

$$first + outside + inside + last \tag{2.53}$$

$$k_{C,0}\frac{-3T_0}{2\Delta z} + k_{C,0}\frac{4T_1 - T_2}{2\Delta z} + k_{C,0}\chi\left(\frac{T_0}{350\,K}\right)^3\frac{-3T_0}{2\Delta z} + k_{C,0}\chi\left(\frac{T_0}{350\,K}\right)^3\frac{4T_1 - T_2}{2\Delta z} \tag{2.54}$$

Now to put the expanded second term back into (2.49):

$$\varepsilon\sigma T_0^4 + k_{C,0}\frac{-3T_0}{2\Delta z} + k_{C,0}\frac{4T_1 - T_2}{2\Delta z} + k_{C,0}\chi\left(\frac{T_0}{350\,K}\right)^3\frac{-3T_0}{2\Delta z} + k_{C,0}\chi\left(\frac{T_0}{350\,K}\right)^3\frac{4T_1 - T_2}{2\Delta z} - q_{surface} = 0 \tag{2.55}$$

and regrouping the terms into standard form (descending powers of T_0):

$$\varepsilon\sigma T_0^4 + k_{C,0}\chi\left(\frac{T_0}{350\,K}\right)^3\frac{-3T_0}{2\Delta z} + k_{C,0}\chi\left(\frac{T_0}{350\,K}\right)^3\frac{4T_1 - T_2}{2\Delta z} + k_{C,0}\frac{-3T_0}{2\Delta z} + k_{C,0}\frac{4T_1 - T_2}{2\Delta z} - q_{surface} = 0 \tag{2.56}$$

$$\varepsilon\sigma T_0^4 + \frac{-3k_{C,0}\chi}{2\Delta z(350\,K)^3}T_0^4 + \frac{k_{C,0}\chi}{(350\,K)^3}\left(\frac{4T_1 - T_2}{2\Delta z}\right)T_0^3 + \frac{-3k_{C,0}}{2\Delta z}T_0 + k_{C,0}\frac{4T_1 - T_2}{2\Delta z} - q_{surface} = 0 \tag{2.57}$$

$$\left(\varepsilon\sigma + \frac{-3k_{C,0}\chi}{2\Delta z(350\,K)^3}\right)T_0^4 + \frac{k_{C,0}\chi}{(350\,K)^3}\left(\frac{4T_1 - T_2}{2\Delta z}\right)T_0^3 + \frac{-3k_{C,0}}{2\Delta z}T_0 + k_{C,0}\frac{4T_1 - T_2}{2\Delta z} - q_{surface} = 0 \tag{2.58}$$

Written this way, we can recognize that (2.58) an algebraic (polynomial) equation of form

$$f(x) = P_n(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + ... + a_{n-1}x + a_n = 0 \tag{2.59}$$

with coefficients

$$a_0 = \varepsilon\sigma + \frac{-3k_{C,0}\chi}{2\Delta z(350\,K)^3} \tag{2.60}$$

$$a_1 = \frac{k_{C,0}\chi}{(350\,K)^3}\left(\frac{4T_1 - T_2}{2\Delta z}\right) \tag{2.61}$$

$$a_2 = 0 \tag{2.62}$$

$$a_3 = \frac{-3k_{C,0}}{2\Delta z} \tag{2.63}$$

$$a_4 = k_{C,0}\frac{4T_1 - T_2}{2\Delta z} - q_{surface} \tag{2.64}$$

where a_4 has no direct dependence on the T_0.

Now that we have the Robin radiative boundary condition expressed as a discretized fourth-degree polynomial equation, when we implement this as Python code we can choose a root-finding algorithm to solve for the surface temperature T_0 at each timestep.

2.6 Initial condition

It's quite unrealistic, but we can set the initial temperature (at time $t_0 = 0$) throughout the soil column at $250\,K$. This will help us visualize the simulated diffusion of thermal energy through the soil column.

2.7 Python implementation of simple thermal model of lunar regolith

Quite a few components go into making this model work. If we write our code as a single procedural script, the rough organization might be:

```
# soilheat.py
# Import dependencies
# Set constants
# Set Moon parameters
# Set model parameters (modelruntime, dt, dz, Nt, Nz, etc.)
# Create numpy arrays
# Set initial conditions
# Time-stepping loop:
    # Calculate solar forcing
    # Set lower Dirichlet boundary condition
    # Solve upper Robin boundary condition
    # Update the new temperature array
    # Assign the old temp array to the new values for use in the
        next iteration
# Plot data
```

The core engine of our time-integrating model is a for control statement that loops through each timestep n up until the total number of timesteps Nt. Letting dt be the length of each timestep, the modelruntime is equal to Nt*dt. We could run our model for a split second, a few seconds, a few minutes, years, or even millions or billions of years, as some people do for astrophysical and paleoclimate models.

```
for n in range(0, Nt):
        (compute T at inner mesh points)
```

Here is a working implementation of this soil thermal model with temperature-dependent diffusivity and nonlinear grid spacing:

```
#soilheat.py

import numpy as np
import matplotlib.pyplot as plt
import scipy.constants as spc
import scipy.optimize
import sympy
import math

# Set constants
pi = np.pi
sigma = spc.sigma

# Moon parameters

S = 1361. # Annual mean solar constant (W/m2)
albedo = 0.12 # Mean Bond albedo for Moon
epsilon = 0.98 # T7 emissivity (Vasavada et al.)
Sabs = S * (1.0 - albedo)
P_moon = 29.53059*24.*3600. #Mean length of Moon SYNODIC day [s]
a = 1e-08   # Value for lunar regolith thermal diffusivity
    provided by Hayne et al 2017
#a = 1e-5 # try unrealistically high thermal diffusivity
k = 0.6e-03     # Equatorial thermal conductivity in Wm-2K-1 (
    summarized by Hayne et al 2017 Table A2)
```

```
q_g = 11e-3 # Mean lunar geothermal surface flux [W m^-2] (
    Siegler and Smrekar (2014)
q_c = 3.13e-6 # Cosmic background radiation flux [W m^-2] (Fixsen
    , 2009)

# Nonlinear Moon parameters (from Hayne et al 2017)

rho_d = 1800 # Deep layer density [kg m-3] Carrier et al 1991
rho_s = 1100 # Deep layer density [kg m-3] Hayne et al 2013
K_d = 3.4e-3 # Deep layer thermal conductivity [W m-1 K-1] Hayne
    et al 2017
K_s = 7.4e-4 # Surface layer thermal conductivity [W m-1 K-1]
    Hayne et al 2017
c0 = -3.6125 # Coefficients for specific heat capacity function [
    J kg-1 K-1] Hayne et al 2017
c1 = 2.7431
c2 = 2.3616e-3
c3 = -1.2340e-5
c4 = 8.9093e-9
chi = 2.7 # Radiative conductivity parameter from Hayne et al
    2017

# Model parameters

T_sfc = 250.0 # Initial temperature [K] at top boundary above the
     zeroth layer
T_bot = 250.0 # Initial temperature [K] at bottom boundary below
    the last layer

modelruntime = 2*24*P_moon/24
dt = 3600 # timestep in seconds (1.0*3600*24 = 1 earth day)
Nt = int(round(modelruntime/dt))   # Calculate number of
    timesteps (rounded and converted to integer)

D = 1.0     # Thickness (depth) of entire slab/system [m]
dz = 0.1    # Thickness of each layer (dist. btwn gridpoints) [m]
Nz = int(round(D/dz))   # Calculate number of layers (rounded and
     converted to integer)

F = a*dt/(dz**2)    # Mesh Fourier number - term in heat
    conduction equation

Ts = np.float() # surface temperature variable used in top
    boundary condition

# Create np arrays for solar calculations and plotting
t = np.zeros(Nt) # start the clock at zero
lt = np.zeros(Nt) # local time in local hours
h = np.zeros(Nt) # hour angle
psi = np.zeros(Nt) # clipping function
solar = np.zeros(Nt) # time array of insolation values to be
    calculated
surfaceflux = np.zeros(Nt) # time array of solar + cosmic
Ts_array = np.zeros(Nt) # time array of surface temperatures
```

```
# Create np arrays for soil thermodynamic properties
rho = np.zeros(Nz+1) # array of densities
K_c = np.zeros(Nz+1) # array of thermal conductivities
K = np.zeros(Nz+1) # array
c = np.zeros(Nz+1) # array of specific heat capacities

# Create np arrays for current soil temperature (T_n) and new
    soil temperature at timestep n+1 (T)

T_n = np.zeros(Nz+1)
T = np.zeros(Nz+1)

# Create np array for soil depth (for plotting)

z = np.zeros(Nz+1)

# Set initial condition T(z,0) and fill in values for depth array
     (for plotting)

for i in range(0, Nz+1):
    if i == 0:
        T_n[i] = T_sfc # Ballpark equatorial max temp [K]
    elif i == Nz:
        T_n[i] = T_bot # Ballpark day/night mean temp for bottom
            layer [K]
    else:
        T_n[i] = T_bot # Unrealistic: set all remaining layers to
             T_bottom
    z[i] = i*dz

# Time-stepping loop

for n in range(0, Nt):
    # Calculate surface flux from above
    t[n] = n*dt # Calculate model time elapsed
    lt[n] = t[n]/P_moon*24 - 12 # Calculate local time in local
        hours
    h[n] = 2*pi*(t[n]/P_moon) # Calculate hour angle
    psi[n] = 0.5 * ( np.cos(h[n]) + np.abs(np.cos(h[n]))) #
        Calculate clipping function
    solar[n] = Sabs * psi[n]  # Calculate solar flux
    surfaceflux[n] = solar[n] + q_c # Calculate total flux of
        energy downward onto surface - Probably useless

    # Calculate soil thermal properties at this timestep

    # Scalar version

    for i in range(0, Nz+1):
        rho[i] = rho_d - (rho_d - rho_s)*math.exp(-i/D)
        K_c[i] = K_d - (K_d-K_s)*(rho_d - rho[i])/(rho_d - rho_s)
        K[i] = K_c[i] * (1 + chi* ( (T_n[i]/350)**3 )   )
        c[i] = c0 + c1*T_n[i] + c2*(T_n[i]**2) + c3*(T_n[i]**3) +
             c4*(T_n[i]**4)

    # Set top boundary condition (Robin)

    Ts = T_n[0]

    T[0] = Ts

    #Coefficients of top boundary condition in polynomial form
    Kc0 = K_c[0]
    a0 = epsilon*sigma + 3/2/dz*Kc0*chi/(350**3)
    a1 = Kc0*chi/2/dz/(350**3)*(4*T_n[1] - T_n[2])
    a2 = 0
    a3 = -3*Kc0/2/dz
    a4 = -3*Kc0/2/dz*(4*T_n[1]-T_n[2]) - surfaceflux[n]

    def f(Ts):
            return a0*Ts**4 + a1*Ts**3 + a2*Ts**2 + a3*Ts + a4

    try:
        Ts = abs(scipy.optimize.newton(f, Ts))
        method = "Newton"

    except:
        print("root␣solver␣not␣happy")

    # Set bottom boundary condition
    T[Nz] = T_bot


    # Set top layer temp
    T[0] = Ts

    # Compute u at inner mesh points (vectorized, nonlinear
        diffusivity)

    T[1:Nz-1] = T_n[1:Nz-1]+ dt/rho[1:Nz-1]/c[1:Nz-1] * (K[1:Nz
        -1]/dz*(T[2:Nz] - T[1:Nz-1]) - K[0:Nz-2]/dz*(T[1:Nz-1] -
        T[0:Nz-2]))


    # Switch variables before next timestep
    #T_n, T = T, T_n
    T_n = T

    # Populate plotting array

    Ts_array[n] = Ts
    print(n*dt/P_moon*24, surfaceflux[n], Ts, T_n[0], T_n[1], T_n
        [2], T_n[3], method)


fig, ax = plt.subplots()
```

```
ax.scatter(lt, Ts_array)
ax.plot(lt, Ts_array)
ax.set_xlim(0,24)
ax.set_ylim(-20,400)
ax.set_xticks(np.arange(0, 25, step=6))
ax.set_xlabel("Time␣(lunar␣hours)")
ax.set_ylabel("Surface␣temperature␣(K)")
ax.annotate("Max␣temp:␣{:.0f}␣K".format(np.amax(Ts_array)), xy
    =(17,350))
ax.annotate("Min␣temp:␣{:.0f}␣K".format(np.amin(Ts_array)), xy
    =(17,325))
ax.set_title("Model␣lunar␣equatorial␣sfc␣temp,␣dynamic␣soil␣
    thermal␣model␣included")
plt.show()
```

As you can see, the file is fairly long because it is written as a one-off procedural script. We could shorten it significantly by modularizing it. Because of the static grid spacing scheme we have chosen, we have to hard-code a constant vertical grid spacing dz. For a soil column with depth D, a ten-layer model with dz = 0.1 seems like a reasonable starting point. At first glance our choice of timestep length dt (3600 s) might seem unreasonably long, but it turns out that this model's results aren't terribly sensitive to the timestep. There is, however, a strong sensitivity to dz, which we can illustrate here. Let's run the code:

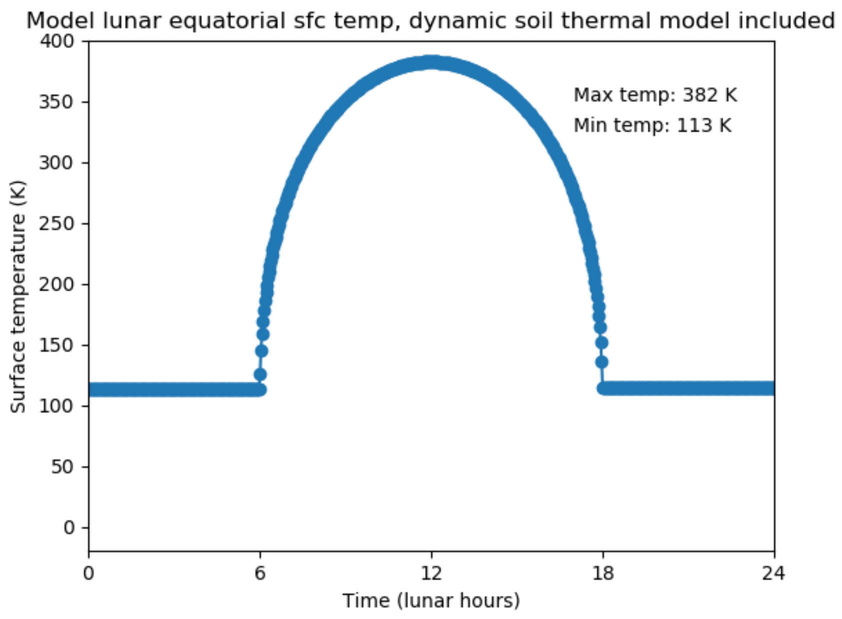

Fig. 2.5 First successful run of dynamic soil model

Well that's not at all terrible! The nighttime temperature is too warm, and it's hard to see any gradual nighttime cooling, but we do have something recognizably close to the Diviner temperature observations for the lunar equator.

We have realistic physics in our model. However we have the important task of tuning our model parameters to give a realistic solution. Let's test the model's dependence on the grid spacing dz. The first order of business is to add an annotation to our matplotlib code that indicates the value of dz used in the model run:

```
ax.annotate("dz:␣{}".format(dz), xy=(1, 350))
```

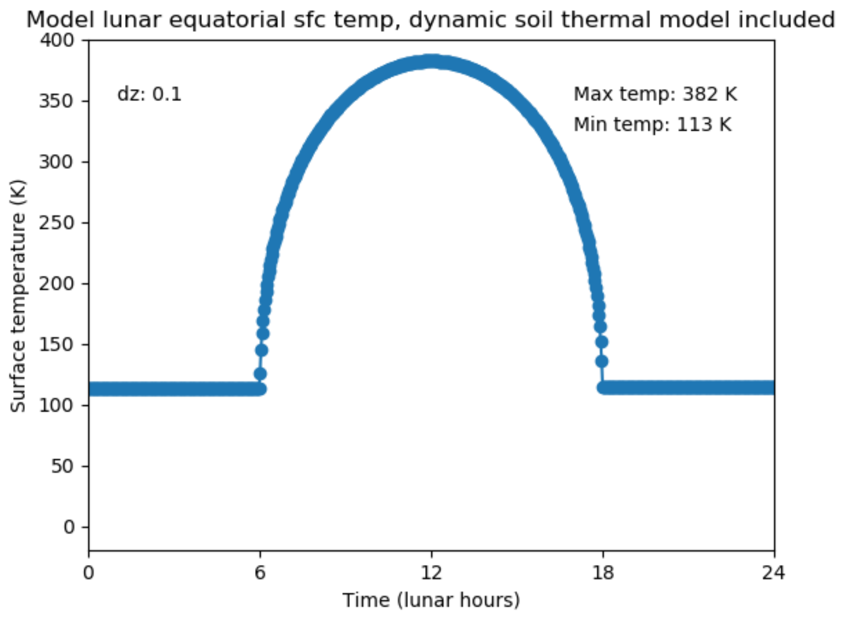

Fig. 2.6 Soil model results with dz set to 0.1 m

That's more like it. Now let's see what happens if we decrease the grid spacing to dz = 0.01:

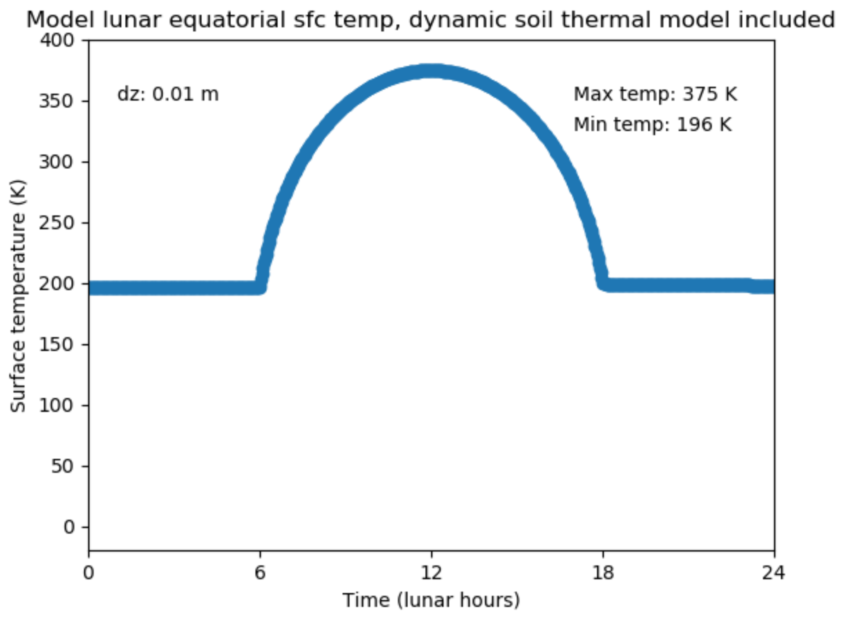

Fig. 2.7 Soil model results with dz set to 0.01 m

Well that might be a bit unexpected. The model actually overestimates the nighttime surface temperature with smaller grid spacing. What happens with dz = 0.2?

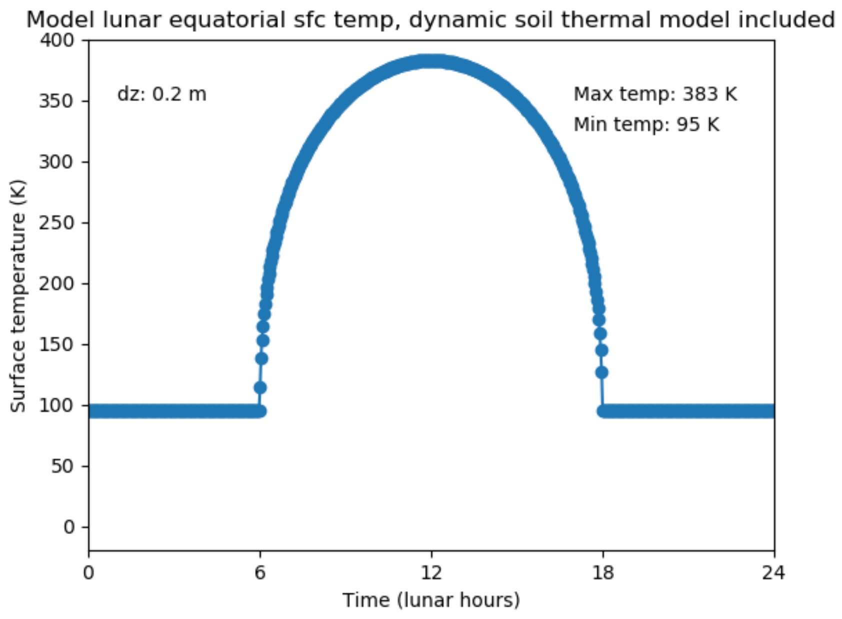

Fig. 2.8 Soil model results with dz set to 0.2 m

This seems to be the most reasonable result, which we had to obtain by blindly hand-tuning. This implies that there must be a better way. We would also like to understand why there is insufficient nighttime cooling. The best approach will be to set our grid spacing dynamically according to the mesh Fourier number stability criterion. This will, however, require us to update our numerical solutions, since in their present formulation they assume constant grid spacing Δz.

Appendix A

Running a tight ship: Modularization and version control

A.1 Setting the right course

As we build our simple climate model, it's never to soon to modularize our Python code and set up version control to track our changes. One very good reason to modularize right away is that it would be great to be able to re-use our lunar temperature model for other airless bodies in the solar system such as Mercury. Some other reasons are that when we don't modularize, our code becomes increasingly difficult to read, and ridiculously difficult to modify.

We've already written something of a module for calculating insolation in a simplified case along the solar equator, the file `solar.py`. In our rather procedural code written so far, we've hard-coded planetary parameters into the scripts themselves. It's time to separate this information into different files that we can run together as a Python package.

I'm giving the climate model we are building here in this book an arbitrary but symbolic name: *whitebark*, a species of conifer that lives at timberline in the mountains of western North America, from California to British Columbia and Alberta.

```
whitebark/

          requirements.txt
          planets.py
          solar.py
          surface_temperature_model.py
```

The file `requirements.txt` is simply a record of the computing environment variables we have used, specifically what versions of Python modules are used, etc. This is necessary for experiment reproducibility. Another researcher can use your requirements.txt file to reproduce your specific Python dependency versions so that running your code should in principle yield the same results.

The file `planets.py` contains an archive of planetary data for use in any planetary climate model. The version we use here is based off of the work of Raymond Pierrehumbert (now at the University of Oxford) and modified by Paul Hayne (now at the University of Colorado, Boulder). The variables set by Pierrehumbert and Hayne guide the naming of the variables used in the *whitebark* planetary climate model.

The file `solar.py` provides modules for calculating insolation as a function of time. We have already done the bulk of the coding for this module.

The file `surface_temperature_model.py` uses the other scripts in the package to calculate surface temperatures at a given point on a planet (for now, the Moon, at a point on the solar equator).

We should create a directory called `whitebark` in the location of our choice. I have created whitebark on my Mac in the directory `~/EarthControlPanel/models/whitebark`, where `~` indicates my user home directory. You can of course choose whatever path you want.

A.2 Setting up a git repository for version control of our Python project

Any good laboratory scientist keeps a lab notebook for recording daily notes regarding what he or she did in the laboratory. As climate modelers we are acting as simulation scientists, working in a virtual laboratory, and scientific notes are equally indispensable for us. Human memory is imperfect and fallible, and detailed notes serve as a virtual memory for us to recall what we did and changed at any given point in the past. Notes help us think and share testable results with others.

When we write code, we are constantly making changes to a script and then testing the effects of those changes. Sometimes those changes break our code, especially if we are inexperienced, but if we are lucky, they lead to breakthroughs. The process of writing computer code in a language such as Python is essentially the process of incremental change and incremental debugging.

Git is a tool written by the developer of the Linux operating system, Linus Torvalds, a native of Finland. Git allows programmers to save snapshots of their incremental work so that if something goes wrong, they can revert their code back to where things worked correctly or to some previous state. When combined with Github, Git allows programmers to create a backup of their work on a remote server, and if desired share their work with potential collaborators who can read, clone, or fork their code repositories. It is not a coincidence that ”sharing your Github” is a common part of applying for programming jobs.

For our purposes here, we will create a local Git repository on our own machine, and push these snapshots to a remote Github repository for redundancy. This will allow us to revert our code if we take one step too far down the wrong path, and will provide us with a remote backup of our code if our local machine happens to crash somehow.

In addition to Git and Github, I also regularly back up the contents of my hard drive using Apple's Time Machine just in case I have a hardware failure. I urgently suggest that you do the same or similar so that you do not lose work in the event of hardware failure, which can happen to anyone, anytime. It has certainly happened to me before. You never know, lightning might strike your house (which has happened to me, frying my power cord), your hard drive might crash (which has also happened to me, costing me days of urgent work) or you might get mugged on the way home from work, as once happened to a certain Earth scientist I know. She was nearly finished with her PhD at the University of Southern California when a man stopped her on the street and demanded that she give him her laptop. Being a strong-willed Russian, she forcefully declined, since her entire PhD thesis was on the laptop. A tug-of-war match ensued, and she fortunately prevailed. I don't care to risk death in a tug-of-war match in the ghetto, so I back up my data to an external USB hard drive and to a cloud server.

Instructions on installing Git and other tools are included in Appendix A, so in this discussion I will assume that you have Git successfully installed. You can make sure that this is the case at the command line by typing:

```
which git
```

Assuming you do have Git installed, the next step is to take a look at your current Git settings:

```
git config --list
```

If anything is out of line, you can change Git configurations according to the following example:

```
git config --global user.name "Mark␣Miller"
git config --global user.email "markmillr@github.io"
git config --global core.editor "sublime"
```

The last line will come in handy when it comes time to document your committed changes. Rather than having to struggle with whatever text editor pops up by default (it could be vim, which is frustrating for the uninitiated), you can set it to your preferred text editor, such as Sublime, which is much easier to use.

Let's assume we are in our home directory. We can create a directory for our whitebark model with the Linux/Unix command:

```
mkdir whitebark
```

After we navigate to the newly created whitebark directory:

```
cd whitebark
```

we can very easily create a Git repository:

```
git init
```

What just happened? We can try to figure this out using the ls command:

```
ls
```

This doesn't show any changes, so we use ls -l -A, which gives a detailed list of all of the contents of the directory, including the hidden ones:

```
ls -l -A
drwxr-xr-x  10 markmiller  staff  320 Jan  2 18:18 .git
```

The "d" at the beginning of the output indicates that .git is a directory, not a file. What's inside the .git directory?

```
cd .git
ls -l -A

-rw-r--r--   1 markmiller  staff   23 Jan  2 18:18 HEAD
drwxr-xr-x   2 markmiller  staff   64 Jan  2 18:18 branches
-rw-r--r--   1 markmiller  staff  137 Jan  2 18:18 config
-rw-r--r--   1 markmiller  staff   73 Jan  2 18:18 description
drwxr-xr-x  13 markmiller  staff  416 Jan  2 18:18 hooks
drwxr-xr-x   3 markmiller  staff   96 Jan  2 18:18 info
drwxr-xr-x   4 markmiller  staff  128 Jan  2 18:18 objects
drwxr-xr-x   4 markmiller  staff  128 Jan  2 18:18 refs
```

Suffice it to say here that Git has created an instance of its version control system right within the whitebark directory. We are free to move the .git directory around as our project evolves, but if we move files in and out of our .git location, Git will no longer be able to keep track of these files.

Using the program structure we outlined earlier:

```
whitebark/

          requirements.txt
          planets.py
          solar.py
          surface_temperature_model.py
```

we can use the touch command to create all of these files without any content:

```
touch requirements.txt
touch planets.py
touch solar.py
touch surface_temperature_model.py
```

In case you're not convinced that these new files actually exist, you can check using ls:

```
ls -l -A
drwxr-xr-x  10 markmiller  staff  320 Jan  2 18:18 .git
-rw-r--r--   1 markmiller  staff    0 Jan  2 18:28 planets.py
-rw-r--r--   1 markmiller  staff    0 Jan  2 18:28 requirements.
    txt
-rw-r--r--   1 markmiller  staff    0 Jan  2 18:28 solar.py
-rw-r--r--   1 markmiller  staff    0 Jan  2 18:28
    surface_temperature_model.py
```

Staging files in the Git repository

Now while our files are empty, there's no better time to add our files (referred to as ”staging”) to the local repository.

```
git add requirements.txt
git add planets.py
git add solar.py
git add surface_temperature_model.py
```

Depending on how you've configured Git, you may need to add ”sudo” to the beginning of each command, which executes the command as the root user.

We can check the status of our Git repository:

```
git status

On branch master

No commits yet

Changes to be committed:
  (use "git␣rm␣--cached␣<file>..." to unstage)

        new file:   planets.py
        new file:   requirements.txt
        new file:   solar.py
        new file:   surface_temperature_model.py
```

Git knows to watch these files for changes, but no snapshots are recorded until we commit those changes. This couldn't be more simple. As soon as we have a small functional change to any of our files, we can use the command

```
git commit -am --status "<useful␣comment␣goes␣here>"
```

The -a and -m flags on the git commit command cause tracked files to be automatically staged, and allow you to enter a commit message from the command line. The commit message is like an entry in your scientific log book: you should make it useful to you or anybody else, now matter how far down the road it is being read. It should make sense, in other words. The –status flag will display the git repository status. Here is an example first commit:

```
git commit -am --status "This␣is␣a␣first␣commit.␣I␣have␣added␣
    empty␣files."
```

```
This is a first commit. Empty files have been added.
4 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 planets.py
 create mode 100644 requirements.txt
 create mode 100644 solar.py
 create mode 100644 surface_temperature_model.py
```

If you type git status after this but before you have made any more changes, this is what you should see:

```
git status

On branch master
nothing to commit, working tree clean
```

Let's make a change to one of our files. The requirements.txt file is easy to fill using pip freeze. This takes a snapshot of all the existing versions of modules currently used by your machine with its version of Python, and lists them as text output.

```
pip freeze > requirements.txt
```

Take a look at your requirements.txt file to make sure changes were made:

```
vim requirements.txt

alabaster==0.7.10
anaconda-client==1.6.9
anaconda-navigator==1.7.0
anaconda-project==0.8.2
---
```

This file is over two hundred lines long on my machine, so I'm not reproducing the whole file text here, but the important thing is that the file now has useful content.

```
git commit -am "Generated␣requirements.txt␣using␣pip␣freeze"

Generated requirements.txt using pip freeze
 1 file changed, 235 insertions(+)
```

How often should you commit? Let me quote from an O'Reilly guide on scientific computing, Effective Computation in Physics:

"In the same way that it is wise to often save a document that you are working on, so too is it wise to save numerous revisions of your code. More frequent commits increase the granularity of your undo button. Commit often. Good commits are atomic, the smallest change that remains meaningful. They should not represent more work than you are willing to lose... Write commit messages as if they were the scientific notation that they are. Like section headings on the pages of your lab notebook, these messages should each be one or two sentences explaining the change represented by a commit. The frequency of commits is the resolution of your undo button. Committing frequently makes merging contributions and reversing changes less painful.”

We can view the history of Git commits with the git log command:

```
git log

commit 79d7be20825f20f7bf1965e0eb0064908dc02a43 (HEAD -> master)
Author: markmillr <zzz@protonmail.ch>
Date:   Wed Jan 2 18:59:43 2019 -0800

    Generated requirements.txt using pip freeze

commit 9ccf8d176c753bd1166c30a8a532a3d8407bb650
Author: markmillr <zzz@protonmail.ch>
Date:   Wed Jan 2 18:44:11 2019 -0800

    This is a first commit. Empty files have been added.
```

We can see that our commits and associated comments are listed with the most recent at the top, and that each commit has a unique number associated with it.

Let’s say that I am not happy with my requirements.txt file, and would like to revert my project back to its state at a certain previous commit.

I can use the git reset command along with the number of the commit that I would like to return to. This will revert all of the files in the project as well as undo any subsequent commits:

```
git reset --hard 9ccf8d176c753bd1166c30a8a532a3d8407bb650

HEAD is now at 9ccf8d1 This is a first commit. Empty files have
    been added.
```

If I check the contents of requirements.txt:

```
vim requirements.txt

~
~
~
~
```

It’s empty! You can prove it to yourself with a little experimentation, but with the reset command I have successfully reverted to an older version of the whitebark project.

Since the reset was just a learning exercise, I’ll redo my previous step and generate the requirements.txt file once again, then commit this change with the same comment:

```
pip freeze > requirements.txt
git commit -am "Generated␣requirements.txt␣using␣pip␣freeze"
```

```
[master 5f9ae91] Generated requirements.txt using pip freeze
 1 file changed, 235 insertions(+)
```

A.3 Pushing the local git repository to a remote GitHub repository

Now it's time to commit to GitHub. You'll need to register for a GitHub account and then link it to Git on your local machine.

We don't want to have to log into Github on our local machine each time we connect from the command line, so we'll want to set up SSH keys.

First check your system to see if there are any already existing public SSH keys:

```
ls -al ~/.ssh
```

If any of the following file names are present in the output, you already have SSH keys:

```
id_dsa.pub
id_ecdsa.pub
id_ed25519.pub
id_rsa.pub
```

If not, you'll have to generate new SSH keys. For information on how to do this you can check https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/

If you do already have SSH keys, you can do further set up using information on the same page under Adding your SSH key to the ssh-agent.

Once you've got your SSH keys configured and added to your GitHub profile, you can create a repository on your GitHub profile. I've named mine whitebark and it is publicly available at https://github.com/markmillr/whitebark

I'd now like to sync my local repository with the whitebark repository.

```
git remote add origin https://github.com/markmillr/whitebark
```

This links the local and remote repositories of the same name.

Now I can push local changes to the server with a single command:

```
git push origin master
```