

Chapter 1

Simulation

Experiments are often limited by a high level of danger, and because they are too expensive or simply impossible to arrange, such as in developing new medical treatments, vehicles for space flight, and also studying geologic events. In these cases we may benefit from the use of computer simulation to refine our understanding and narrow our investigation prior to an “official” observation. Since the promise of this technique is to accelerate information gathering for a relatively low total cost, interest has been gaining momentum *everywhere*.

Examples in this chapter include discrete, continuous, and interactive systems with particular importance given to the long-term (asymptotic) trend as the scale of a problem grows toward infinity. Our approach favors clarity and context over trivia or theory with the hope that this better enables early success, but of course in learning there are never any real guarantees. Efficiency matters but is not paramount as we prefer to implement more widely accessible solutions while perhaps merely suggesting a state-of-the-art method.

For all code listings we use Python 2 with Tk and PIL, plus gnuplot, but other options are available (e.g., Matlab[®], *Mathematica*, Python 3, Java, C/C++, Fortran, Scheme, Pascal) and our belief is these choices are sufficiently intuitive to serve as instructive pseudocode, that also happens to run!, for any environment you use. Regardless, we feel the central issues up-front are quality problems, a thoughtful sequencing of topics, and rapid feedback.

1.1 Random Walk

Imagine yourself outside on a pleasant day looking for a nice place to sit and read. By chance you are standing exactly halfway between your two favorite spots but are unable to decide which one to take. Out of curiosity you engage in a rather obscure process to settle the matter: you flip a coin and take one step in the indicated direction, then flip again followed by another step, and again and again, moving back-and-forth as the coin dictates, possibly coming very close to one spot or the

Table 1.1: Three examples of a size $n = 5$ random walk.

-----X-----	-----X-----	-----X-----
----X -----	----- X----	----- X----
-----X-----	----- X----	----- X----
----X -----	----- X----	----- X----
---X- -----	-----X-----	-----X-----
--X- -----	----X -----	----- X----
-X- -----	---X- -----	----- X----
-X- -----	---X- -----	----- X----
-X- -----	-----X-----	----- X----
--X- -----	----X -----	----- X----
---X- -----	---X- -----	----- X----
-X- -----	--X- -----	----- X----
-X- -----	-X- -----	----- X----
X- -----	--X- -----	----- X----
----- -----	-X- -----	----- X----
Steps: 14	X- -----	----- X----
	-X- -----	----- X----
	X- -----	Steps: 16
	----- -----	
	Steps: 18	

other but ending only when a final step brings you all the way there. We wish to simulate this random drifting process with computer code. Each lab is numbered so that the first digit indicates chapter (1-7), the second a particular problem (1-3) in the chapter, and the third digit your specific assignment (1-5) related to the problem.

Lab111: Trace of a Single Run

We begin with output as shown in Table 1.1 where variables specify the size of our walk and also track current position. If n is the distance from the halfway point to the edge just next to either destination then $m = 2n + 1$ is the total distance available for drifting.

Code Listing 1.1: Initializing variables.

```
#
n=5
m=2*n+1
j=  n+1
#
```

The values of n and m remain constant here but our current position j will start at the halfway point $j = n + 1$ and then change as we drift until either $j = 0$ or $j = m + 1$ and we have arrived at a reading spot. Whenever $j = 1$ or $j = m$ we are at one of the two edges, needing but a single step more in that direction to complete our walk. Commands to initialize and update these variables “over time” are shown in Code Listings 1.1 and 1.2, respectively.

Code Listing 1.2: A complete walk’s loop.

```
while 1<=j<=m:
    #
    if random()<0.5: # coin flip
        j+=1
    else:
        j-=1
    #
```

Of course we want to *see* the walk, too. Once we know everything is working properly this “trace” will be less important but we should first verify that our code is behaving as expected. We could just print the value of j at each step but it will be better if we draw a “picture” instead. Helper variable k loops over the entire drifting area in Code Listing 1.3 to display a single row from any of [Table 1.1](#)’s examples.

Code Listing 1.3: A loop to display the entire drifting area.

```
k=1
while k<=m:
    if k==j:
        print 'X',      # current position
    elif k==n+1:
        print '|',      # halfway point
    else:
        print '- ',
    k+=1
print
```

Your first assignment is to put all this code together into a 1-D random walk simulation, including counting-up the total number of steps.

Questions to consider:

- Do our examples represent a typical size $n = 5$ random walk?
- What happens to the number of steps, on average, as n increases?

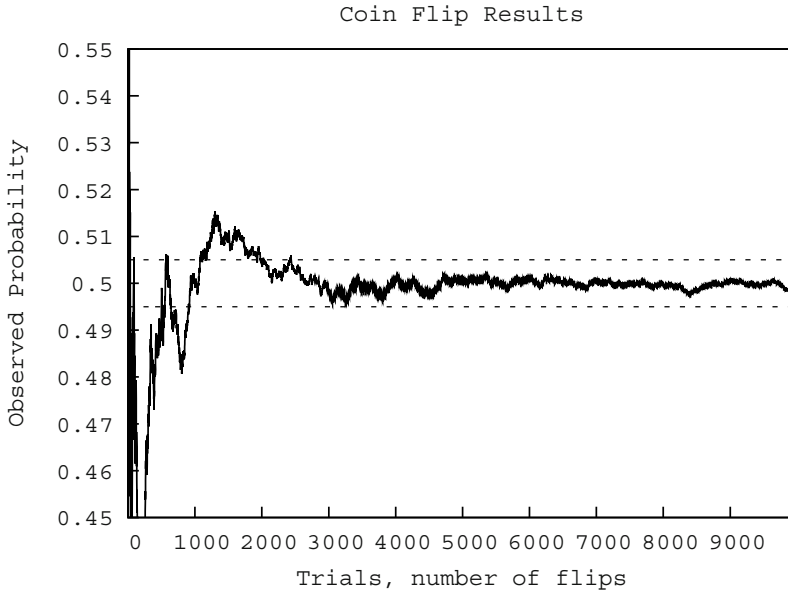


Fig. 1.1: Law of large numbers applied to coin flips.

Lab112: Law of Large Numbers

Before proceeding we ask an essential question: After how many coin flips are we reasonably sure the heads-tails ratio is “close” to even? Your assignment is to write a small program that only flips coins, over and over and over again, to calculate the percentage of heads, or tails, obtained. Code Listing 1.4 is a sample gnuplot script for a plot similar to the one shown in [Figure 1.1](#), where we assume the program’s output (total observed probability after each trial) has been stored in a plain text file.

Code Listing 1.4: Sample gnuplot script.

```
set terminal png
set output "lab112.png"
set title "Coin Flip Results"
set ylabel "Observed Probability"
set yrange[0.45:0.55]
set ytics 0.01
plot "lab112.txt" with lines notitle,0.505 w l notitle
```

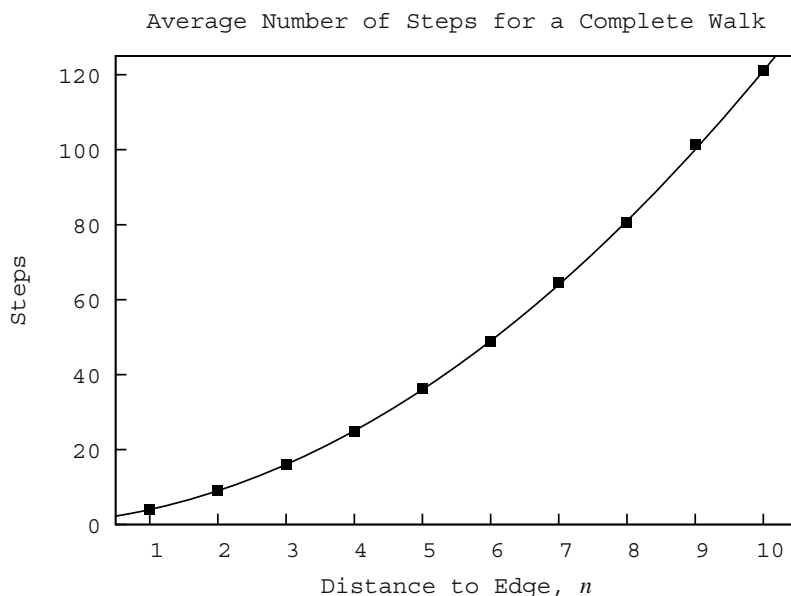


Fig. 1.2: Quadratic growth in the average number of steps to complete a walk.

Lab113: Scaling the Problem Size

We show the average number of steps for walks of size $n \leq 10$ in [Figure 1.2](#) overlaid with the curve $f(n) = (n+1)^2$ since it takes $n+1$ total steps in a particular direction to actually end the walk. A trial now is an entire random walk, not just a single coin flip, and we use Algorithm 1.1.1 and 10,000 trials for accurate results.

Algorithm 1.1.1 Average number of steps for various n .

```

1: while  $n \leq n_{max}$  do
2:    $steps = 0$ 
3:   while  $t = 1 \rightarrow trials$  do
4:     while random_walk do
5:       ...
6:        $steps = steps + 1$ 
7:     end while
8:   end while
9:   print  $n, steps/trials$ 
10: end while

```



Fig. 1.3: A large plume of ash during an eruption of the Mt. Cleveland volcano, Alaska, as seen from the International Space Station on 23 May 2006. Image courtesy of the Image Science and Analysis Laboratory, NASA Johnson Space Center.

Transport and Diffusion

Observation of the physical world often reveals behavior that may be explained at least partially by drifting. For instance, the plume of ash shown in [Figure 1.3](#) moves in two fundamental ways. First, wind blows the ash in some direction, a topic we will consider in more detail with the next problem. Second, the ash spreads out and eventually the plume will break-up in a process called *diffusion* that can be modeled by the random motion of individual ash particles. This means that our random walk, while an absurd method for picking a spot to read, does relate accurately to the diffusive aspect of a plume's movement over time.

However, volcanic eruptions may have a worldwide impact over many years, scales far too large for 3-D particle-by-particle calculations even for state of the art algorithms running night and day on the fastest computers in the world. Simulations of such cases require a team of experts (we need you!) to build different models, better algorithms, and bigger computers.



Fig. 1.4: Blue Mountain supercomputer, Los Alamos National Laboratory, New Mexico, one of the most powerful computing systems in the world at the end of the twentieth century. Image courtesy of Los Alamos National Security, LLC.

Parallel Processing

One popular technique for large-scale simulations is the use of multiple systems connected together to process sub-units of the overall code in parallel. For instance, individual trials in the previous lab are independent of each other and therefore may be run simultaneously on separate machines in order to speed up calculation.

The parallel computing system shown in [Figure 1.4](#) was one of the world's most powerful supercomputers. It has since been decommissioned and not even a decade later the same level of capability became commercially available in graphics cards, also massively parallel but no longer the exclusive domain of a premier national lab the use of whose image requires:

Unless otherwise indicated, this information has been authored by an employee or employees of the Los Alamos National Security, LLC (LANS), operator of the Los Alamos National Laboratory under Contract No. DE-AC52-06NA25396 with the U.S. Department of Energy. The U.S. Government has rights to use, reproduce, and distribute this information. The public may copy and use this information without charge, provided that this Notice and any statement of authorship are reproduced on all copies. Neither the Government nor LANS makes any warranty, express or implied, or assumes any liability or responsibility for the use of this information.

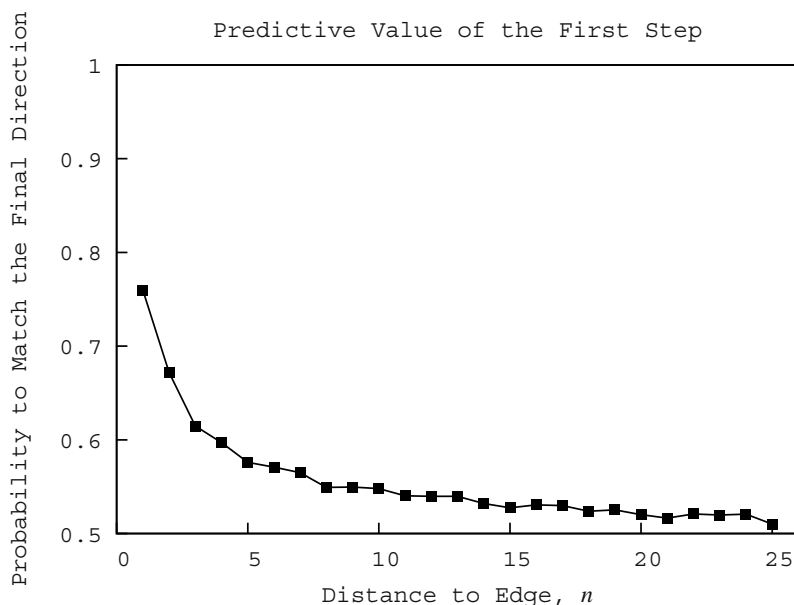


Fig. 1.5: As the size of the walk increases the importance of the first step decreases.

Lab114: Predictive Value of the First Step

We want to know how important the first step is in determining the final direction our drifting leads. Code Listing 1.5 shows how the first step can be “remembered” for later comparison after the entire walk has finished. Your job is to count the number of times this very first step matches the final direction. Results for $n \leq 25$ shown in [Figure 1.5](#) reflect again the use of 10,000 trials for each size.

Code Listing 1.5: Remembering the first step for later comparison.

```
j=n+1
if random()<0.5:
    j+=1
else:
    j-=1
first_step=(j-(n+1)) # either 1 or -1
#
while 1<=j<=m:
    #
    ...
```

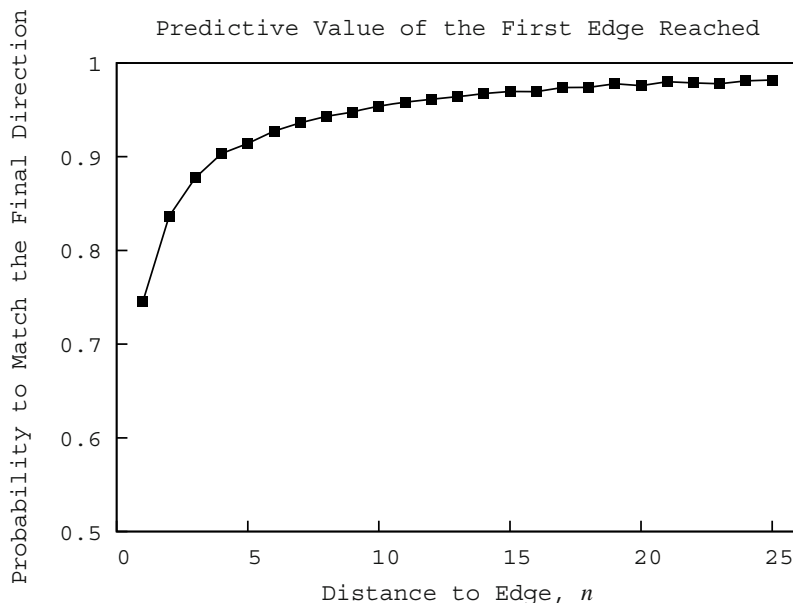



Fig. 1.6: The importance of the first edge reached increases with the size of the walk.

Lab115: Predictive Value of the First Edge Reached

Asking a similar question about the first edge allows us to use the symmetry of our problem to speed up runtime. It is not necessary to start in the middle and wait for the drifting to eventually reach an edge; instead, just start on an edge at step one! Results for $n \leq 25$ shown in [Figure 1.6](#) are consistent regardless of which edge we start on and again reflect the use of 10,000 trials for each size.

Code Listing 1.6: Using symmetry to speed up a first edge calculation.

```
j=m
#
while 1<=j<=m: # walk starts at edge
    #
    ...
    #
#
if j==m+1:      # was there a match ?
    #
    ...
```

1.2 Air Resistance

Imagine some friends in a sunny field kicking the soccer ball around. If the ball leaves your foot at 60 miles per hour and forming a 30 degree angle with the ground then how far does it go? What does its trajectory look like? Such questions are often asked in math and science classrooms* but our investigation will approach this problem from the standpoint of computer simulation.

Lab121: Deconstructed Parabola

We begin by neglecting air resistance but it will be clear that, using our approach, including air resistance in the model is straightforward. In all cases we restrict ourselves to 2-D and some behavior, such as lift and spin, will not be considered.

The soccer ball's velocity and position are initialized in Code Listing 1.7 where we assume $v_0 = 26.82$ meters per second and $\theta = \pi/6$ radians have already been converted from mph and degrees, and `cos` and `sin` are imported from `math`.

Our main loop shown in Code Listing 1.8 updates the position (x,y) based on the velocity (v_x, v_y) with v_x constant (for now) and v_y itself updated by the “pull” of gravity, $g = -9.81 \text{ m/s}^2$ near the surface of the Earth.

Code Listing 1.7: Initializing variables.

```
#
vx=v0*cos(theta)
vy=v0*sin(theta)
#
x=0.0
y=0.0
```

Code Listing 1.8: A complete trajectory's loop.

```
#
while y>=0.0:
    #
    x+=(vx*dt)
    y+=(vy*dt)
    #
    vy+=(g*dt)
    #
```

* See, for instance: R. Larson et al., *Algebra 2*. McDougal Littell, 2004.

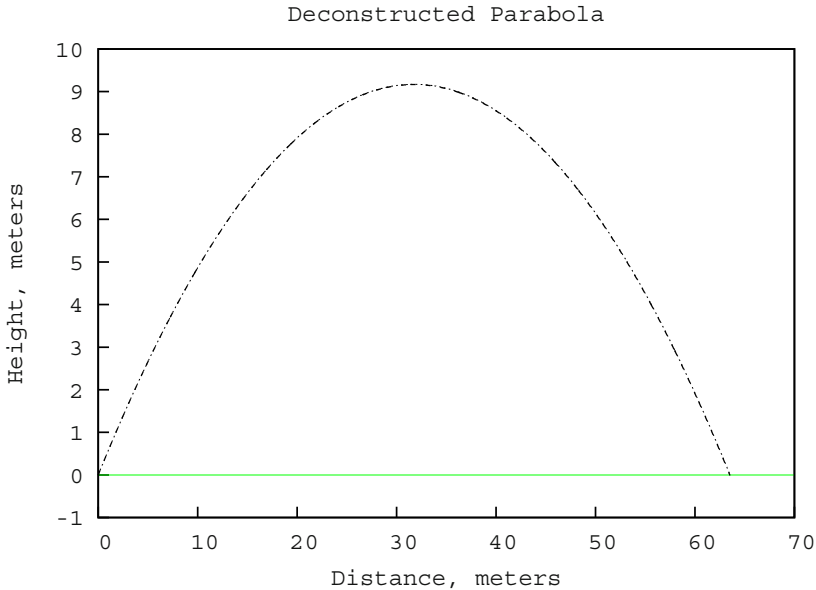


Fig. 1.7: Trajectory plot without air resistance.

Obviously v_y changes since what goes up must come down. We can think of v_x as a non-diffusive “transport” similar to the previously unexamined aspect of the ash plume’s motion. We use a timestep $dt = 0.001$ seconds and total time t can also be tracked. If our loop outputs (t, x, y) at each step then Code Listing 1.9 plots the overall (x, y) trajectory as shown in [Figure 1.7](#).

Code Listing 1.9: Trajectory gnuplot script.

```
set terminal png
set output "lab121.png"
set title "Deconstructed Parabola"
set xtics nomirror
plot "lab121.txt" using 2:3 w l notitle, 0 w l notitle
```

Two observations about a soccer ball’s motion make this simulation possible:

- Horizontal and vertical movement can be treated separately.
- Small enough timesteps allow for straightforward modeling.

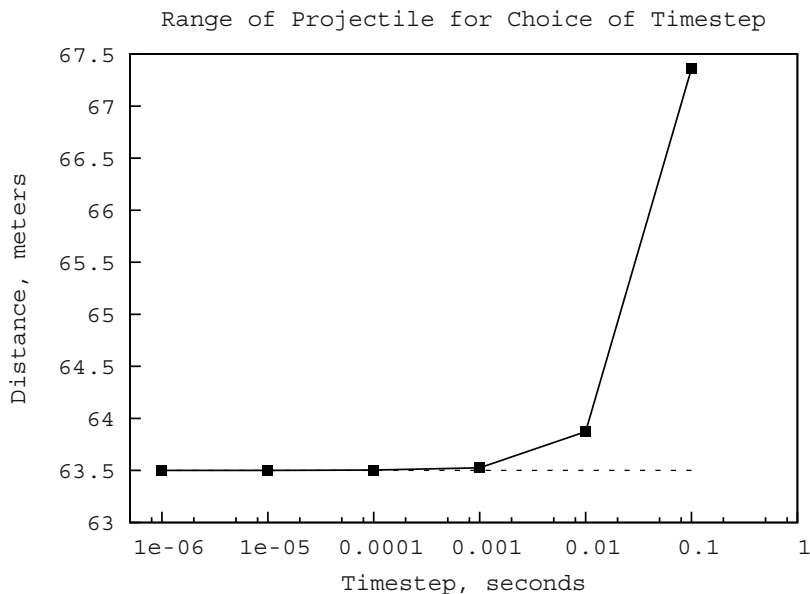


Fig. 1.8: Logarithmic scale for x with `set logscale x` in gnuplot.

Lab122: Timestep Study

The range of a projectile is horizontal distance traveled before impact; a test of timesteps for the soccer ball range is shown in [Figure 1.8](#) with $dt = 10^{-N}$ and $N \leq 6$. Our results indicate $dt = 0.001$ is a good compromise (if $dt = 0.0001$ then our main loop would require 10 times the number of steps). Some anticipated objections:

- Use the exact formula $y = gt^2/2 + v_0 \sin(\theta)t + y_0$ instead of simulation.
 - First, with air resistance the exact formula will be more complicated to derive.
 - Second, our [Figure 1.7](#) had the simulation and “exact” values in two different dash-patterns and yet they overlayed so well as to seem like only one, not two.
- Our choice of $dt = 0.001$ seconds gives an incorrect range.
 - It is hard to say precisely what is the correct range. Did the ball leave your foot at 60 mph or was it really 61 mph? Was the angle 30 degrees or 29 degrees?

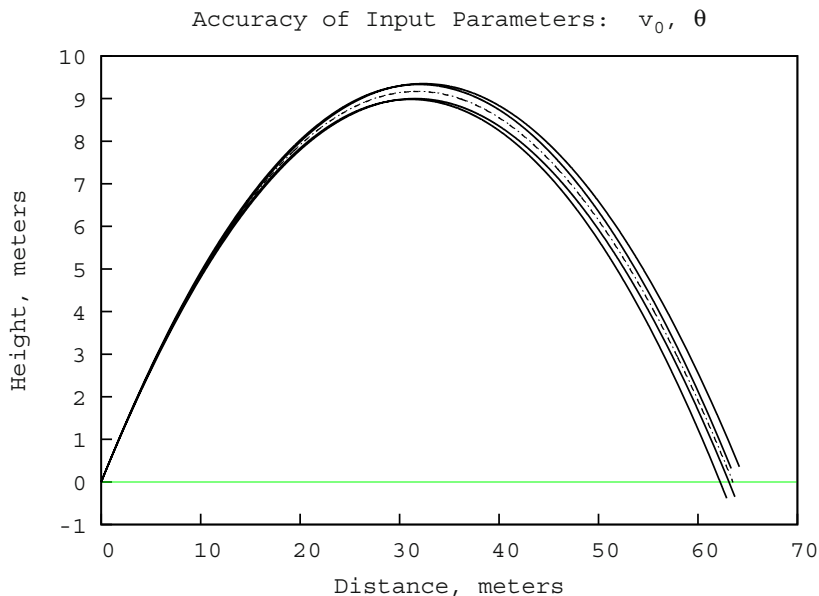


Fig. 1.9: Variation in trajectory for a 1% change in each input parameter.

To elaborate on this, [Figure 1.9](#) shows the variation in trajectory corresponding to a 1% change in each input parameter, using the exact formula. Results are closer when only the angle changes but still the two dashed curves clearly coincide best. Even comparing the range to its exact value $x = v_0^2 \sin(2\theta)/|g|$, the horizontal line from [Figure 1.8](#), our simulation with $dt = 0.001$ seconds is accurate to within one-tenth of one percent.

Lab123: Physical Model

Since the simulation is now working we wish to augment our model $a_x = 0, a_y = g$ with terms that reflect air resistance. Code Listing 1.10 shows how this might look.

Code Listing 1.10: Acceleration that varies with velocity.

```
#
ax= ( -c1*vx)
ay=(g-c1*vy)
#
```

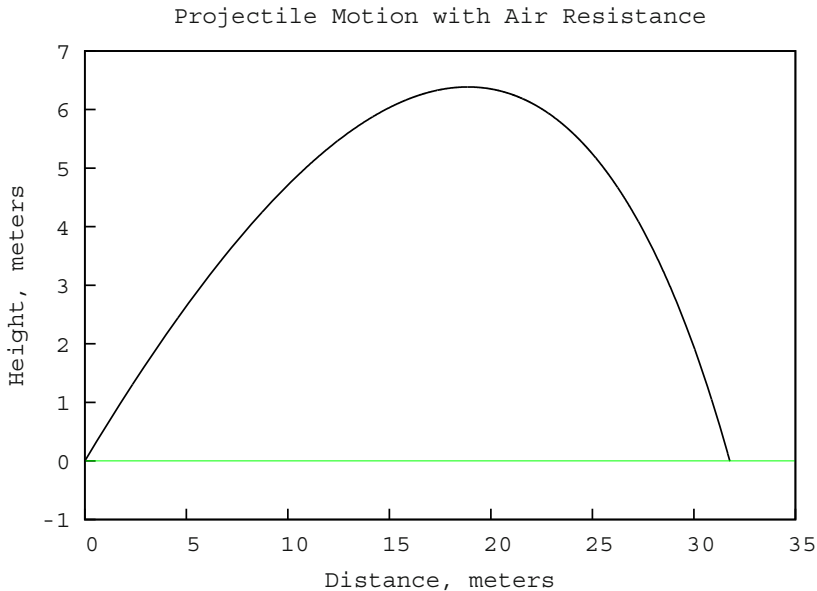


Fig. 1.10: Trajectory plot with air resistance.

A better model for air resistance is $a = -c_1 v - c_2 v^2$ but the quadratic term is more complicated in 2-D and our focus here is foremost on the “big idea” and how it appears in your code. Namely, the idea is that air resistance opposes the direction of motion (hence the minus sign) and also scales with increased speed.

Our plots reflect $c_1 = 0.5$, an ad hoc value, where in general this would depend on the material property of the projectile, its shape, the composition of air, current altitude, etc. The mass is also an implicit part of the c_1 coefficient.

Note the dramatic changes in both height and range compared to our parabolic trajectory; if this seems too unreasonable you should change c_1 . (After all, it is your code.) Disclaimer: Any similarities to an actual product or scenario-of-interest is unintentional and must occur purely by chance.

Code Listing 1.11: Fix the x -axis in place for comparing multiple plots.

```
set terminal png
set output "lab124.png"
set xrange[:35]
plot "lab124.txt" u 2:3 w l notitle,0 w l notitle
```

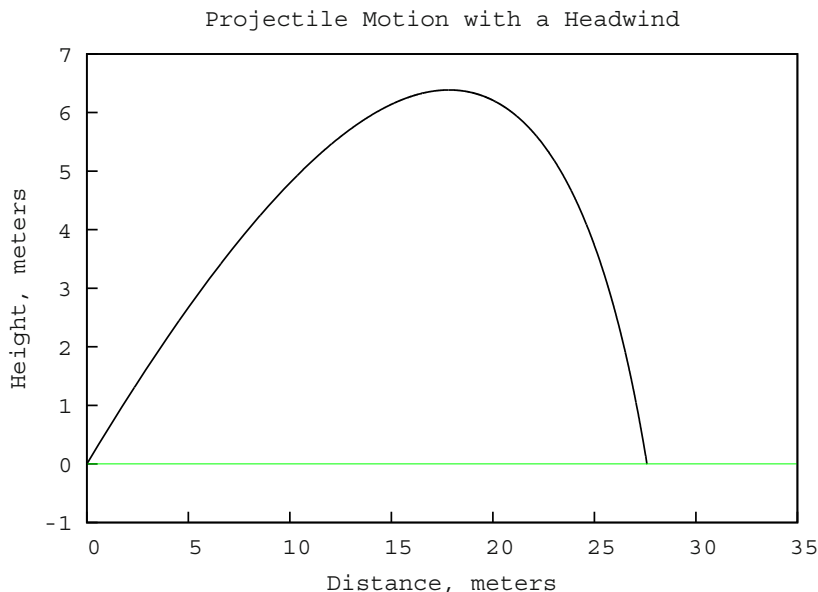


Fig. 1.11: Trajectory plot with air resistance and a headwind of 10 mph.

Lab124: Wind

Code Listing 1.11 shows a script that fixes the x -axis in place for comparison of multiple plots. If we assume there is a horizontal wind only and call v_w its velocity then we can model this wind by calculating the relative velocity ($v_x - v_w$) as shown in Code Listing 1.12. We use relative velocity instead of the absolute v_x to determine the horizontal acceleration due to air resistance.

Since v_y is not affected by v_w the calculation of a_y is exactly as it was before and comparing [Figures 1.10](#) and [1.11](#) we see that height is unchanged but range is diminished. Plots reflecting even stronger winds are shown in [Figure 1.12](#).

Code Listing 1.12: Relative velocity due to wind.

```
#
ax= ( -c1*(vx-vw) )
ay=(g-c1*(vy    ))
#
```

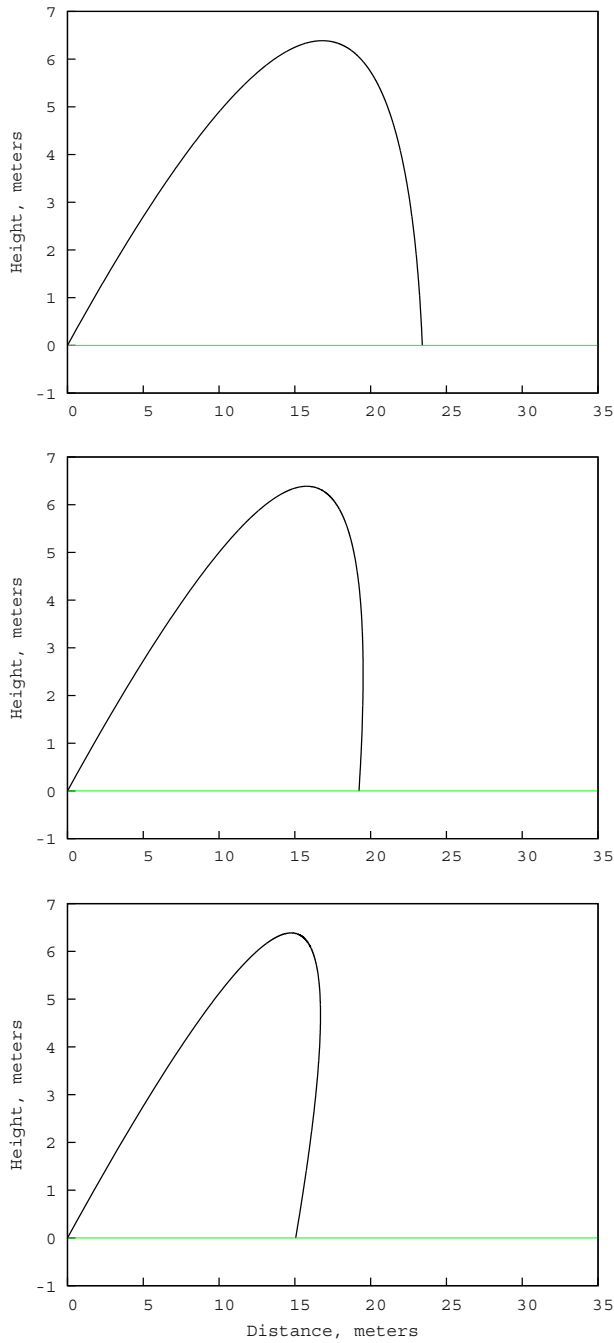


Fig. 1.12: Headwind trajectory plots. Top to bottom: 20 mph, 30 mph, and 40 mph.

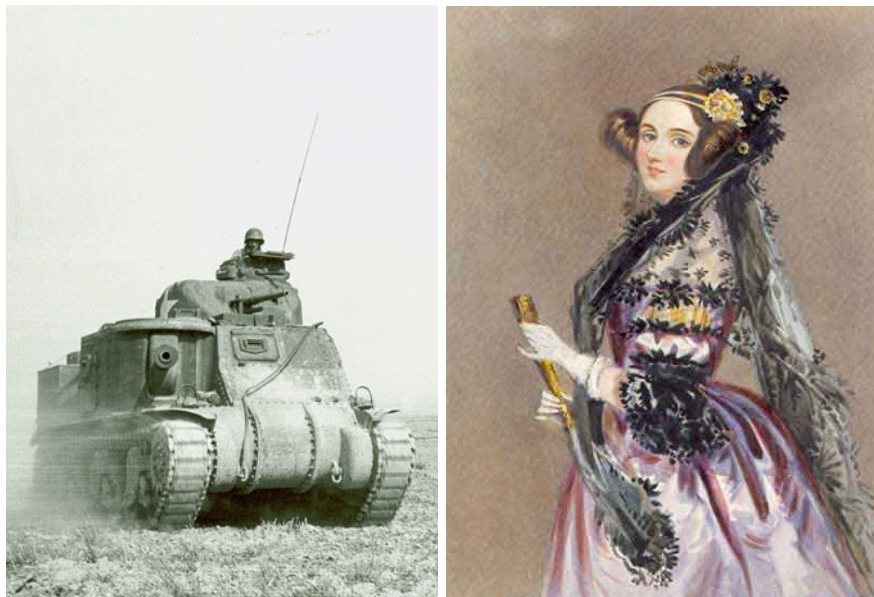


Fig. 1.13: Early users of modern computing systems. Left: A tank in Tunisia, 1943, the World War II campaign that motivated ENIAC. Right: Ada Lovelace, 1838. Images courtesy of the U.S. Army and NASA. Tank photo credit U.S. Army Military History Institute, WWII Signal Corps Photograph Collection.

Z1, ABC, and ENIAC

The idea of a computer is not new. In the nineteenth century mechanical devices were built to perform otherwise difficult calculations. The first programmer was Ada Lovelace, shown in [Figure 1.13](#) alongside a World War II era tank, and U.S. Defense Department programming language Ada was named after her. During the North African Campaign our need to update artillery firing tables motivated ENIAC, famous as the “first” computer although Z1 was already programmable in Germany and ABC was already electronic at Iowa State University.

While more elaborate equations are used in practice, each entry of a firing table based on our 2-D model would require finding the range x given the wind v_w and initial angle θ , assuming that v_0 , g , and all other parameters are fixed. These ranges could be calculated in parallel and since each entry is independent of any other this is an “embarrassingly parallel” problem, so-called because the parallel code is relatively straightforward to write.

Besides having already generated such a firing table and simply looking up the answer, how could you find θ to hit a specific target range x_T given a known v_w ?

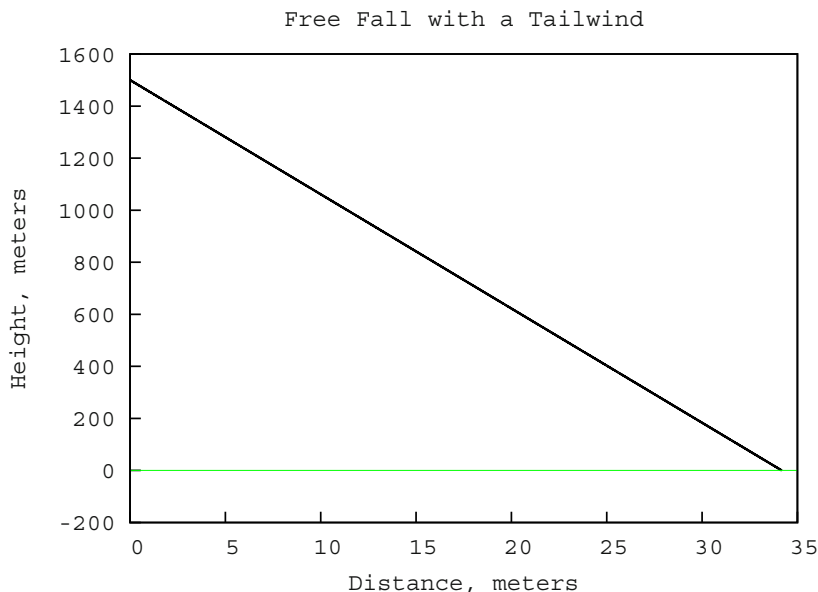


Fig. 1.14: Trajectory plot for free fall from a height of just under one mile with a tailwind of not quite 1 mph. Note carefully the vast difference between horizontal and vertical scales.

Lab125: Free Fall

To simulate free fall we set $v_0 = 0.0$ and perhaps $y_0 = 1500.0$ meters, with a slight tailwind of $v_w = 0.447$ meters per second. Results are shown in [Figure 1.14](#) where we note the vast difference between horizontal and vertical scales.

When v_y stops changing the projectile has reached “terminal velocity” indicating that $a_y = 0$ because the two terms g and $c_1 v_y$, shown again in [Code Listing 1.13](#) for reference, balance each other out. Detailed plots in [Figure 1.15](#) show the evolution of vertical velocity and acceleration over time.

Question: When would $a_x = 0$ and thus v_x stop changing, too?

Code Listing 1.13: Terminal velocity occurs when $a_y = 0$.

```
#
ay=(g-c1*vy)
#
```

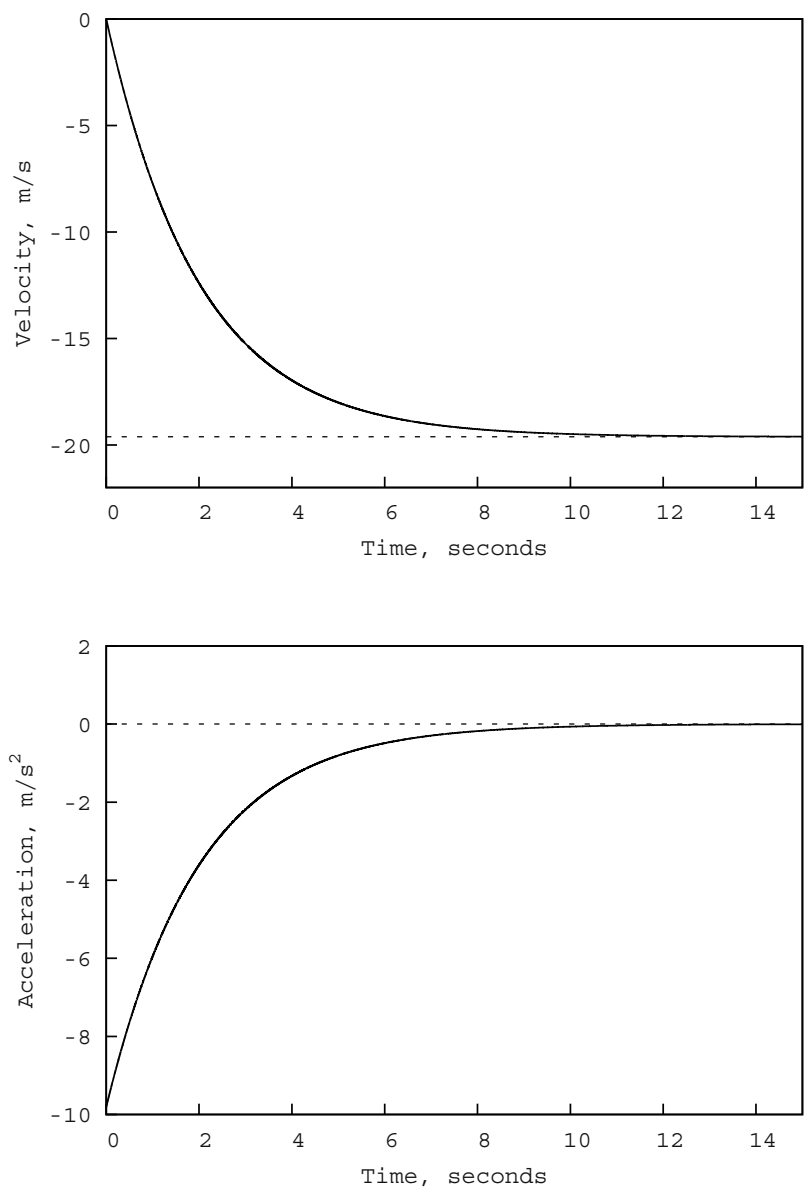


Fig. 1.15: Vertical velocity and acceleration during free fall.

1.3 Lunar Module

Our final problem for this chapter involves landing on the Moon. Unlike the two previous problems our code will now have to allow the user to interact with a running program. Note that [Figure 1.16](#) contains an anachronism: the Earthrise background photo is from the Apollo 8 mission on 24 December 1968, but the *Eagle* lunar module photo is from Apollo 11 on 20 July 1969, the day before extravehicular activity.

Lab131: Uncontrolled Descent

We begin first with no user interaction because the animation alone requires your complete, focused attention. The underlying model is 1-D freefall with no air resistance as the lunar atmosphere is practically a vacuum. Code Listing 1.14 shows a complete program, written in Python 2 with Tk and PIL, and your assignment is to understand what it does.

Line 31 establishes that a_y is determined entirely by $g_M = -1.62 \text{ m/s}^2$ and we note that compared to Earth's $g_E = -9.81 \text{ m/s}^2$ the gravitational acceleration is only 16.5% as strong on the Moon. So, when we conclude that our speed on impact is almost 40 mph keep in mind that this figure would be closer to 100 mph on Earth; well, only if air resistance is neglected... thank goodness for air! And parachutes!

Lines 14-16 are familiar, note however they are no longer contained in a loop but instead a function `tick` to control animation. Commands to make this animation work are underlined on Lines 11 and 24, and the call on Line 57 to the canvas object's `after` method starts the entire process with a request that “after 1 millisecond” the tick function should be called,

but it does not actually call the tick function itself.

The function is defined starting on Line 11 and takes no arguments. The conditional call on Line 24 keeps the process going (tick, tick, tick, tick, tick, ...) following the first call. In addition to facilitating animation we can also think of this repeated calling as taking the place of our main loop, with Line 23's `if`-statement acting like the “keep going” condition of a `while`-loop.

Of course only a single frame can be displayed on the screen at any one time and the graphics are repeatedly updated by Line 21 which moves our image of the *Eagle* as it falls. The illusion of crashing is maintained because Line 20 and Line 37 consider $y = 0$ to be only 70% of the way downscreen, an ad hoc figure.

Curiously, the Tk system requires that the names `pmg1` and `pmg2` on Line 47 and Line 51 be different; otherwise, if the variable of a photo image is discarded then the corresponding Tk image will disappear (!) from the window.

[Figure 1.17](#) shows our goal: landing on the moon instead of crashing.

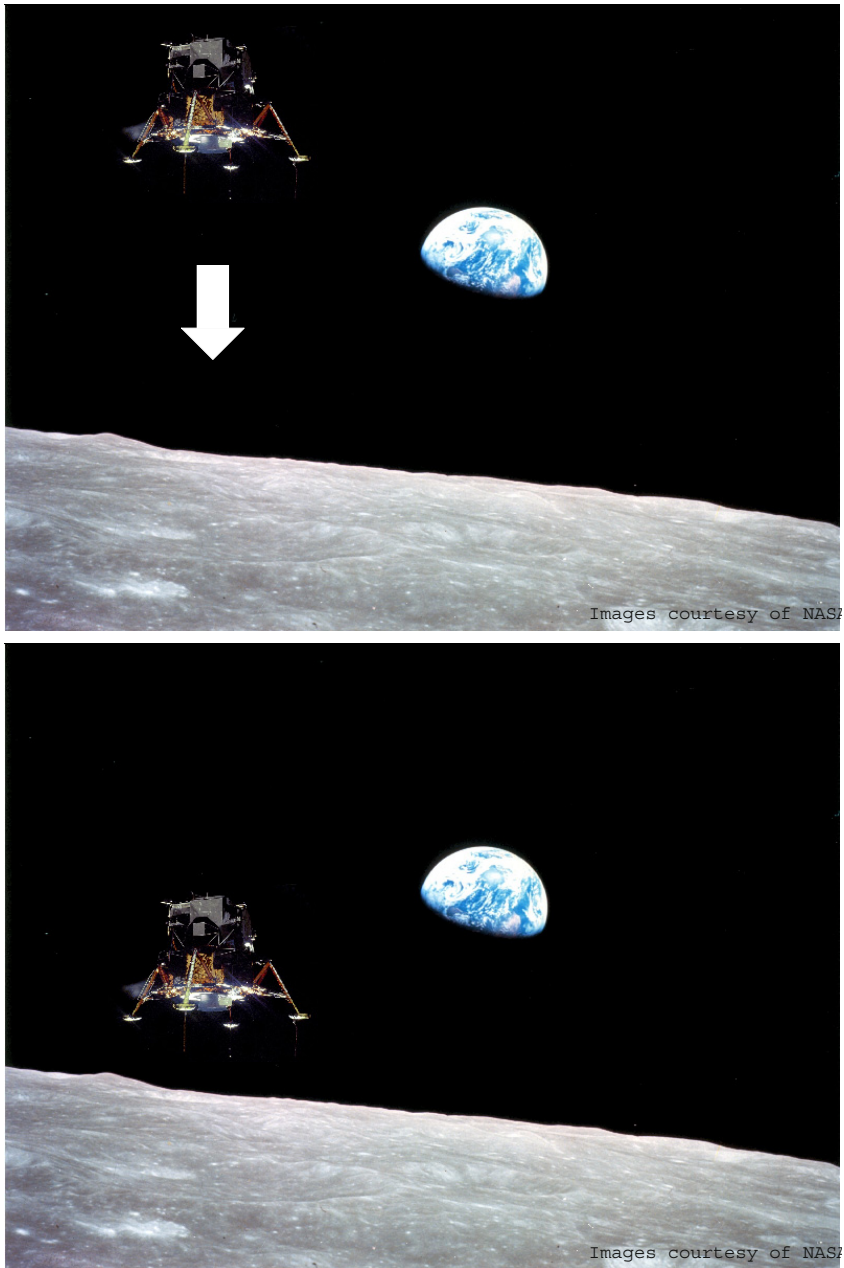


Fig. 1.16: Descent of the lunar module (not shown to scale). Top: beginning free fall at an altitude of 100 meters, a contrived scenario. Bottom: crashing at over 40 mph. Images courtesy of NASA, lunar module photo credit Michael Collins.

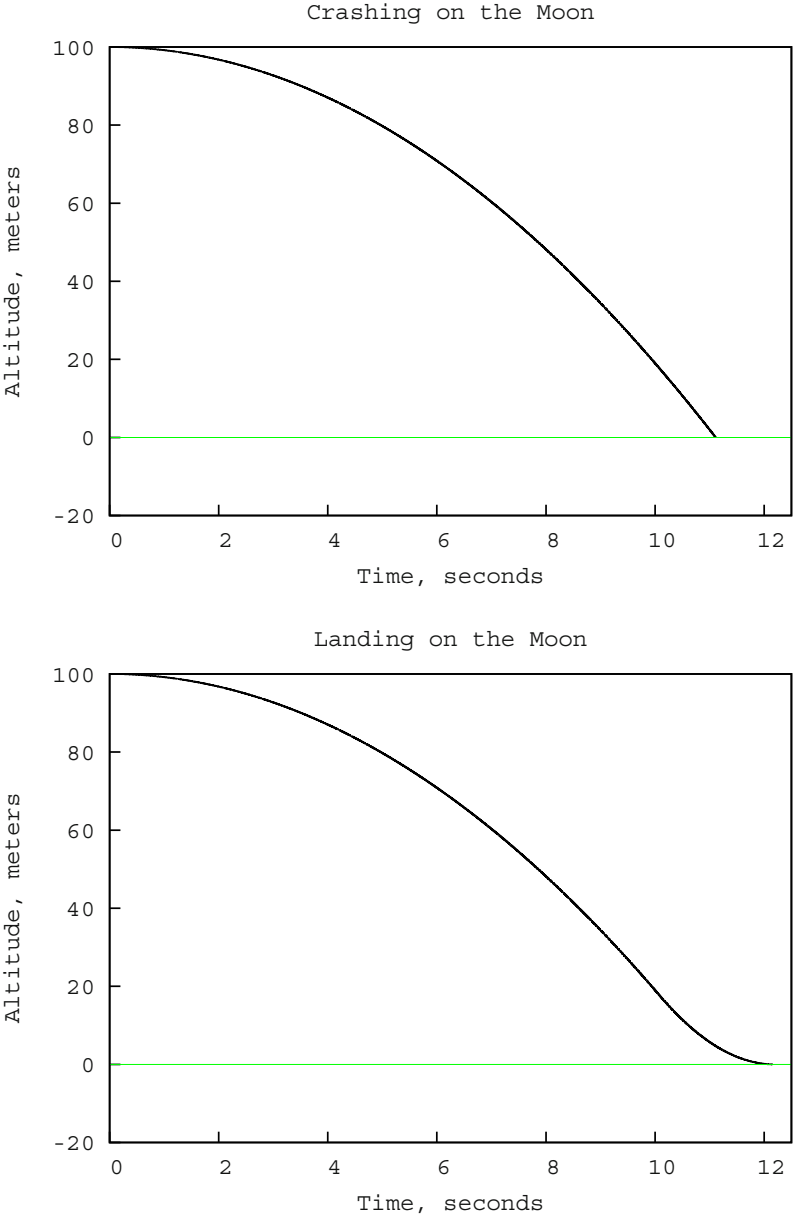


Fig. 1.17: Altitude over time during lunar descent. Top: crashing at over 40 mph. Bottom: controlled landing with vertical thrusters burning just before touchdown so that impact velocity is well under 2 mph.

Code Listing 1.14: A complete program, written in Python 2 with Tk and PIL.

```

1 #####
2 #
3 # Chapter 1: Simulation
4 # Problem 3: Lunar Module
5 # Lab 1.3.1: Uncontrolled Descent
6 #
7 #####
8 from Tkinter import Tk,Canvas
9 from PIL import Image,ImageTk
10 #####
11 def tick():
12     global t,y,vy
13     #
14     t += dt
15     y += (vy*dt)
16     vy += (ay*dt)
17     #
18     print t,y,vy
19     #
20     yp = 0.7*h/y0*(y0-y)
21     cnvs.coords(tkid,w/4.0,yp)
22     #
23     if y>0.0:
24         cnvs.after(1,tick)
25 #
26 #####
27 w,h= 800,600
28 #
29 y0 = 100.0 # meters --> we must stipulate that images are not shown to scale
30 vy = 0.0 # m/s --> so, assuming some previous arrangements at play here
31 ay = -1.62 # m/s^2, acceleration due to gravity near the surface of the Moon
32 #
33 t = 0.0
34 dt = 0.001
35 #
36 y = y0
37 yp = 0.7*h/y0*(y0-y) # linearly interpolate --> crash before the very bottom
38 #
39 print t,y,vy
40 #####
41 #
42 root=Tk()
43 cnvs=Canvas(root,width=w,height=h,bg='black')
44 cnvs.pack()
45 #
46 img1=Image.open('earthrise.jpg').resize((w,h))
47 pmg1=ImageTk.PhotoImage(img1)
48 cnvs.create_image(w/2.0,h/2.0,image=pmg1)
49 #
50 img2=Image.open('eagle.jpg').resize((200,170))
51 pmg2=ImageTk.PhotoImage(img2)
52 tkid=cnvs.create_image(w/4.0,yp,image=pmg2)
53 #
54 f=('Times',14,'bold')
55 cnvs.create_text(w-110,h-15,text='Images courtesy of NASA.',font=f)
56 #
57 cnvs.after(1,tick)
58 root.mainloop()
59 #
60 # end of file
61 #
62 #####

```

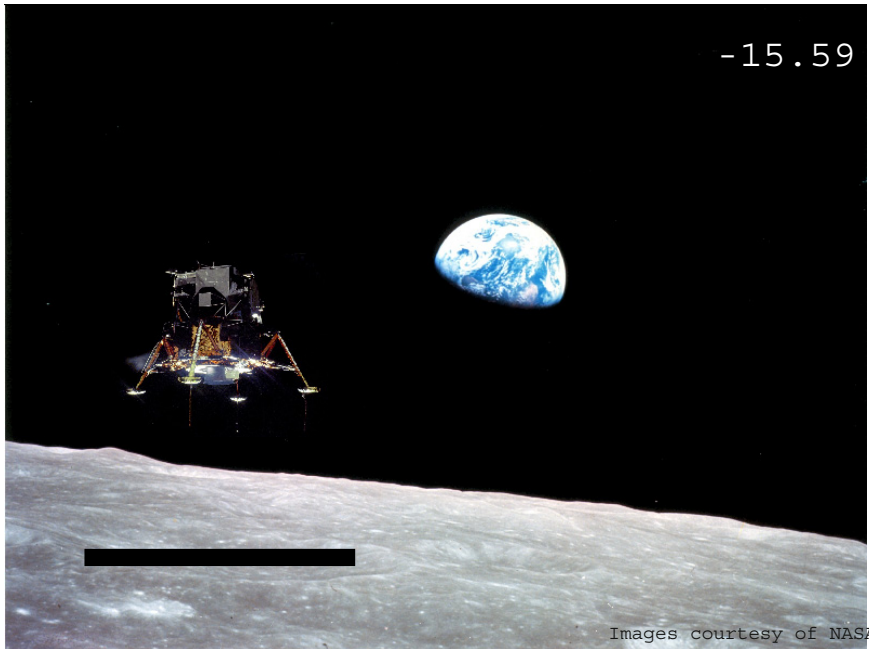


Fig. 1.18: Controlled lunar descent where the black rectangle now artificially marks our landing site and the current velocity is indicated in meters per second. Images courtesy of NASA, lunar module photo credit Michael Collins.

Lab132: Controlled Descent

Now that our animation is working we can add user interaction. Code Listing 1.15 shows commands both to display current velocity v_y and also to fire vertical thrusters by pressing the spacebar. Figure 1.18 suggests further helping the user control for a soft landing by artificially marking the landing site.

Code Listing 1.15: Additional commands that allow for user interaction.

```
def tick():
    cnvs.itemconfigure(tkid2, text='%0.2f'%vy)
#
def spacebar(evnt):
    global vy
    vy+=1.0 # another ad hoc figure
#
tkid2=cnvs.create_text(w-75,50, text='%0.2f'%vy, fill='white')
#
root.bind('<space>', spacebar)
```

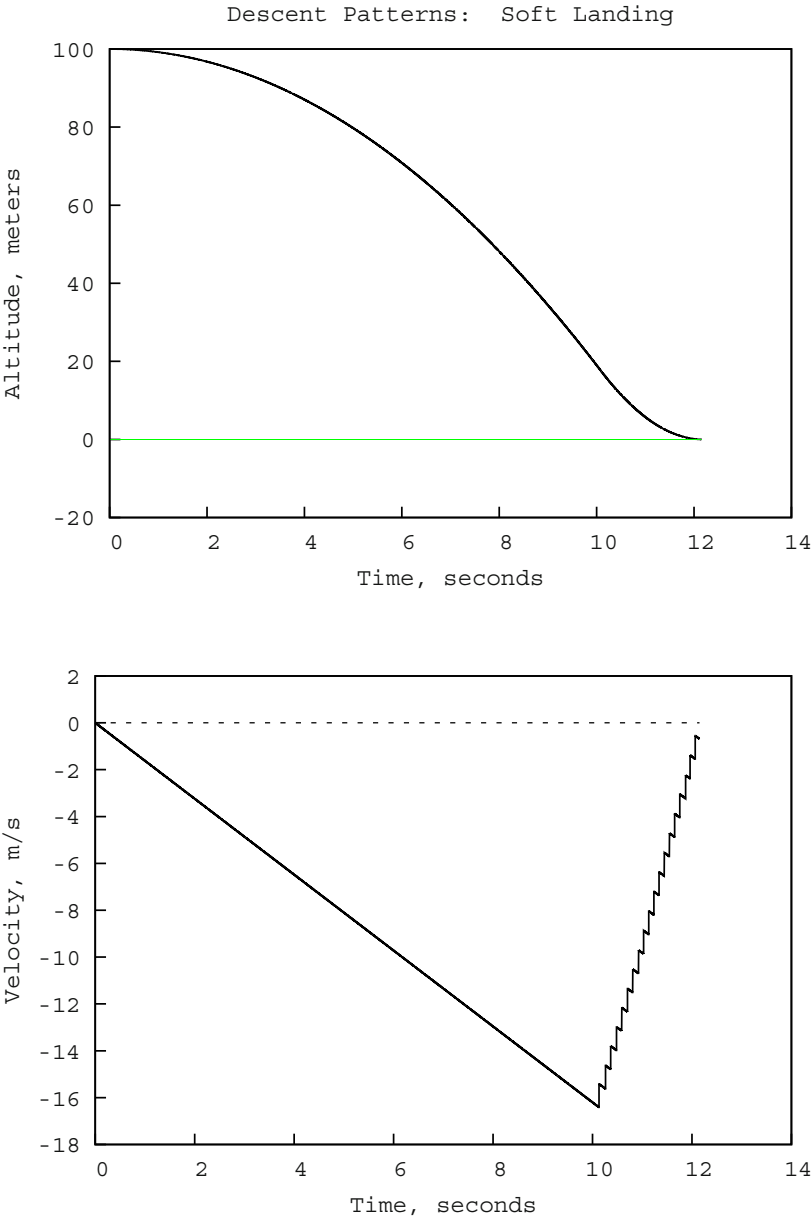



Fig. 1.19: Altitude and velocity over time during lunar descent. The case shown here is a “soft landing” where vertical thrusters are burned just after the 10 second mark.

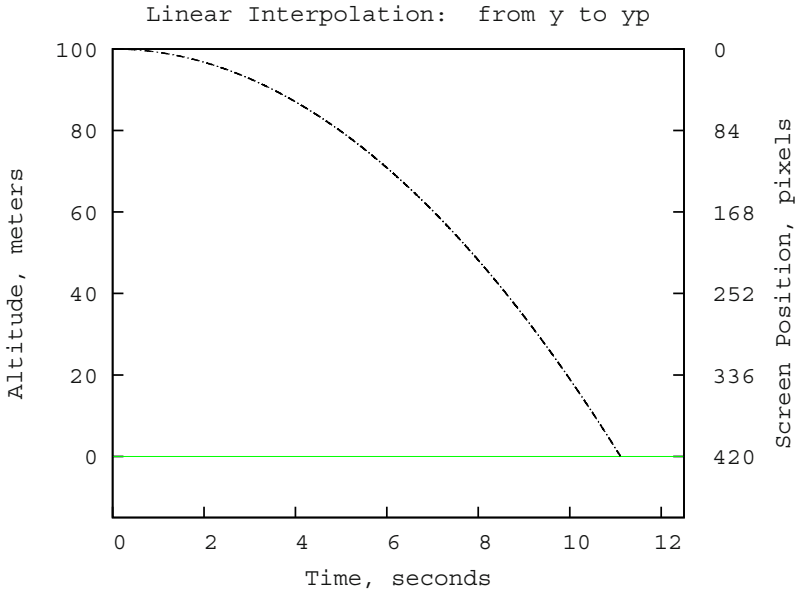


Fig. 1.20: Linear interpolation from altitude to screen position.

Lab133: Descent Patterns

The “sawtooth curve” in [Figure 1.19](#)’s velocity plot reflects the fact that we instantaneously add 1.0 to v_y whenever the spacebar is pressed, because the code is easier to write this way, but in reality the act of burning thrusters would itself take time making the v_y curve smoother.

Translating from altitude to screen position, as shown again in Code Listing 1.16 for reference, assumes $\frac{yp-0}{0.7h-0} = \frac{y-y_0}{0.0-y_0}$ which calculates yp from a known y given that 0 (screen position) corresponds to y_0 , and $0.7h$ with 0.0 (altitude). Vertical coordinates are often inverted as [Figure 1.20](#) shows because the upper-left corner tends to anchor a graphics window.

Your assignment is to generate the plots shown in [Figures 1.21](#), [1.22](#), and [1.23](#).

Code Listing 1.16: Linear interpolation of vertical position.

```
#
yp=0.7*h/y0*(y0-y)
#
```

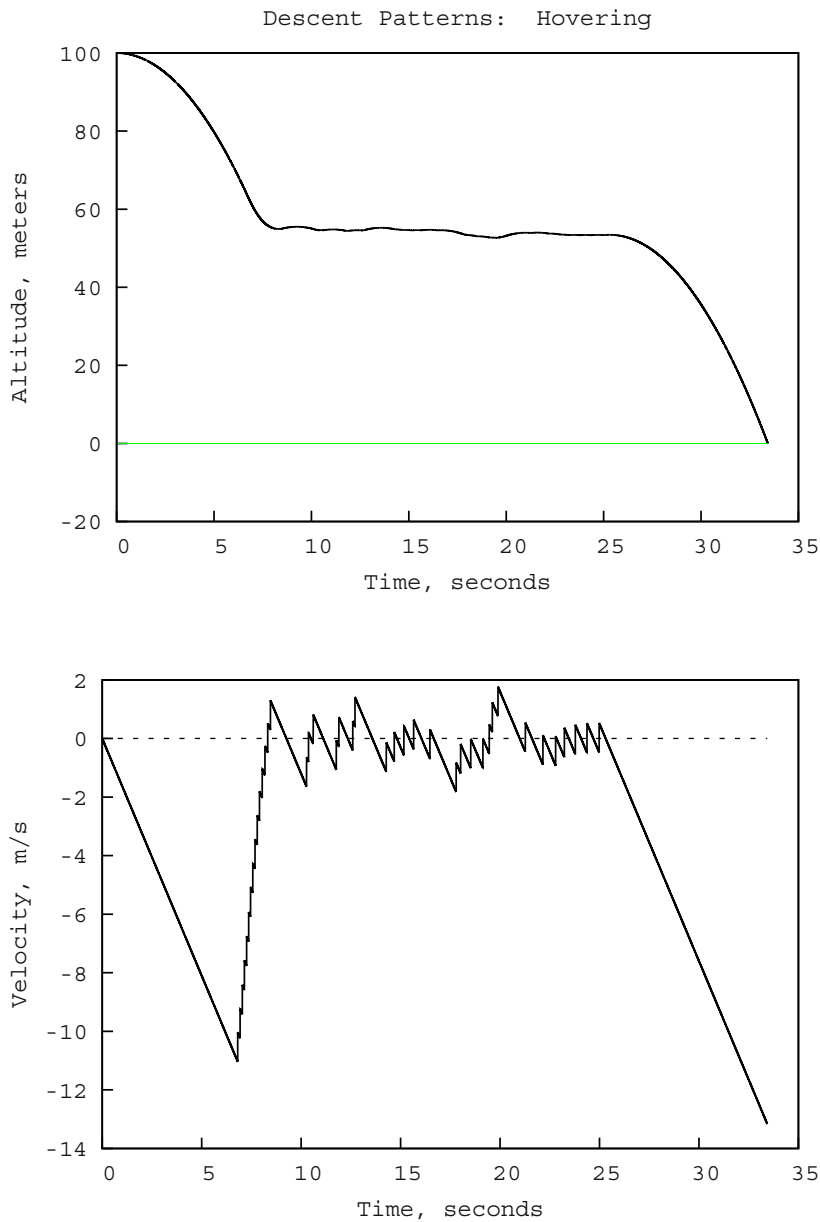


Fig. 1.21: Altitude and velocity over time during lunar descent. In this case the lunar module “hovers” mid-descent before crash landing at just under 30 mph.

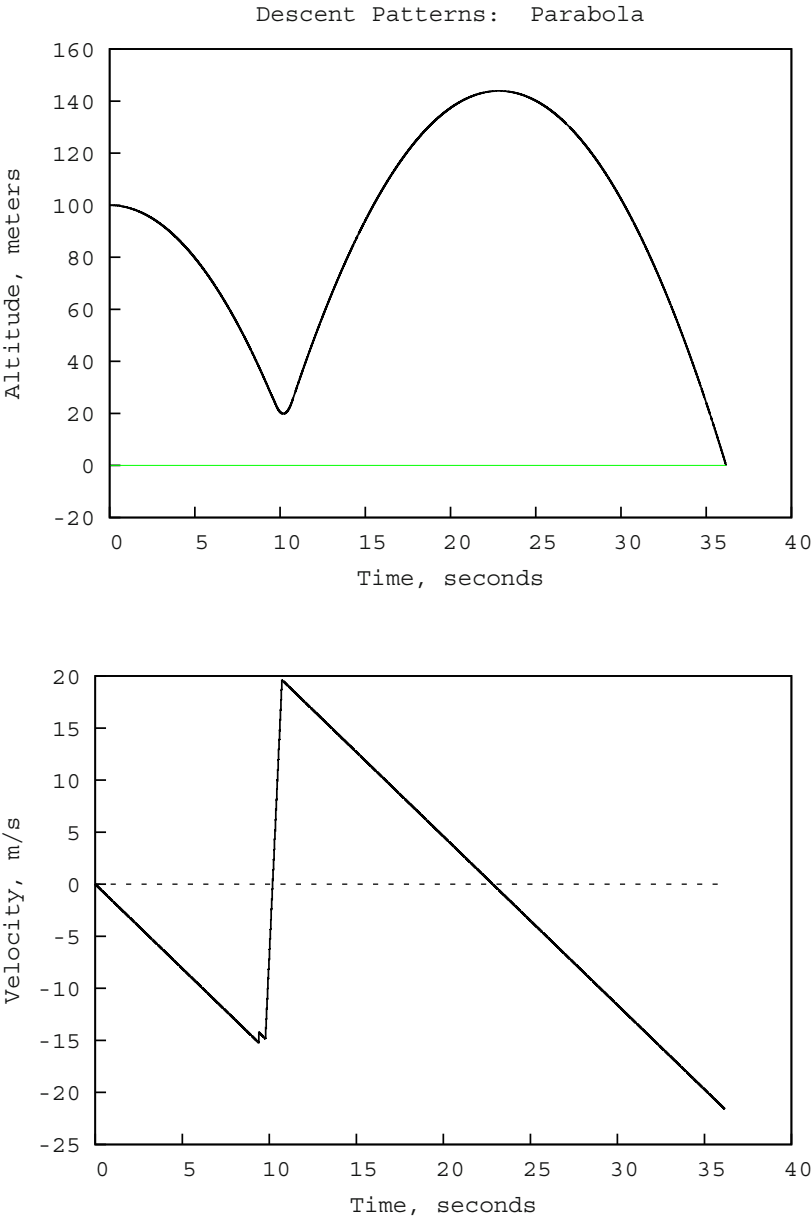


Fig. 1.22: Altitude and velocity over time during lunar descent. In this case the lunar module performs a secondary “climb” before crash landing at just under 50 mph.

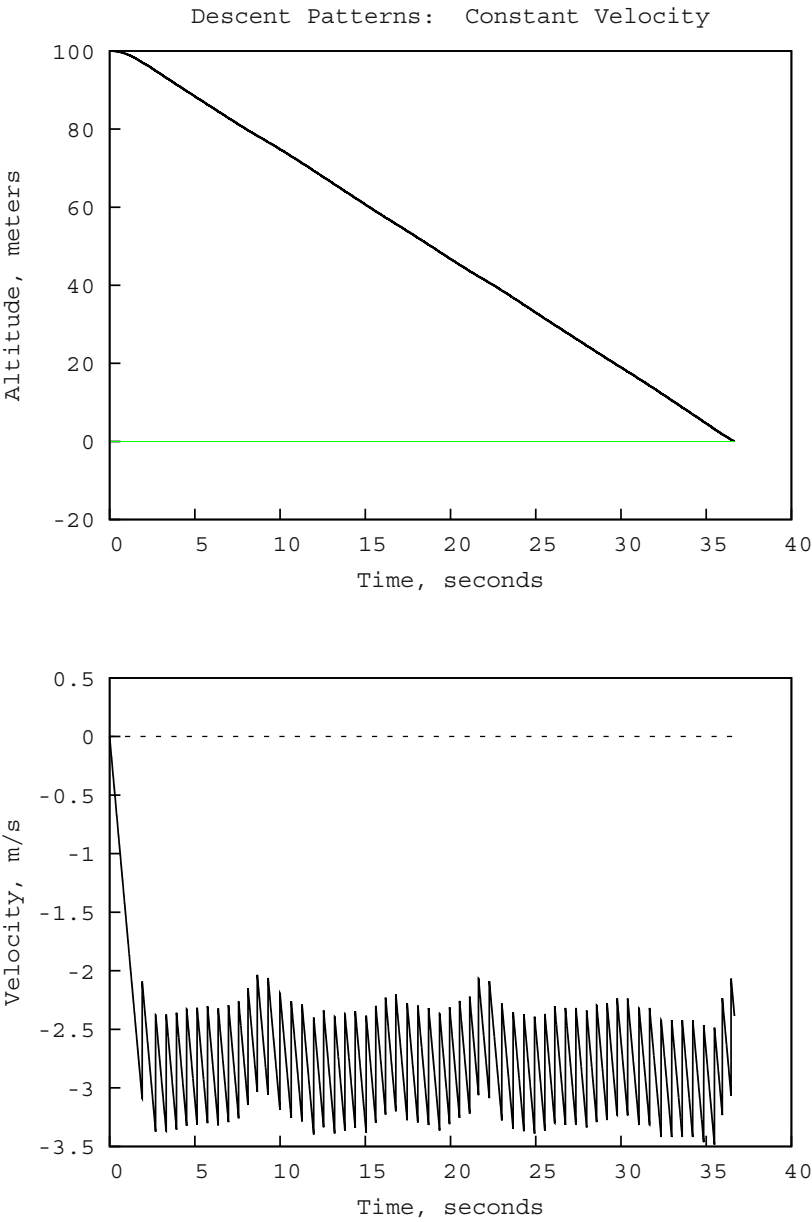


Fig. 1.23: Altitude and velocity over time during lunar descent. In this case the lunar module maintains a constant velocity “profile” and lands safely at just over 5 mph.

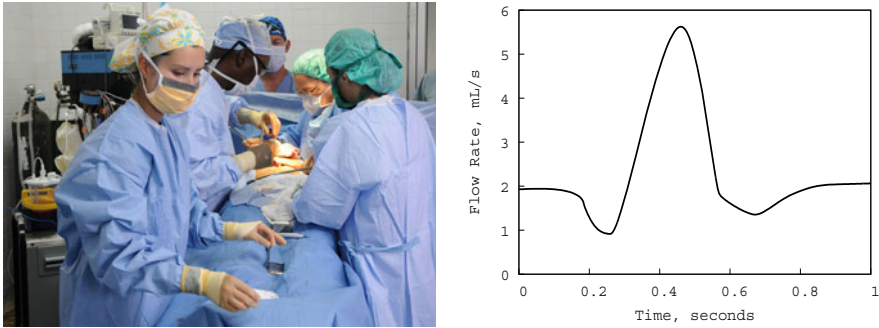


Fig. 1.24: Computing applied to diagnosis and treatment. Left: An operating room during surgery, the culmination of extensive preparation. Right: Typical volumetric bloodflow profile from a single heartbeat, one starting point toward the expansion of knowledge. Image courtesy of the U.S. Army, photo credit C. Todd Lopez.

Medical Applications

Simulation has become sufficiently robust to influence medicine by, for instance, modeling the flow of blood within our bodies. [Figure 1.24](#) suggests one piece of minimal information needed for diagnosis or treatment.

The process by which arteries transport blood away from our heart is similar to incompressible flow in a pipe where viscosity dictates the interaction with tissue lining the inner wall of our blood vessels. Unlike a lunar module descent velocity, bloodflow is periodic and the waveform pattern is repeated with each heartbeat.

These flows were studied by Poiseuille in the nineteenth century while the more general set of equations were, around the same time, formulated by both Navier and Stokes. An inviscid model had already been developed by Euler in the eighteenth century using Newton's famous second law of motion from the seventeenth century. But only in the latter twentieth century have computers become powerful enough to apply these models to practical cases.

So, science waited centuries for machines to catch up. Now what?

Running an experiment that may pose some danger to a human patient is governed chiefly by issues related to medical ethics, and we choose to err on the side of "first, do no harm." Again this makes simulation an attractive technique but also when a patient's life-and-death (!) depends on proper diagnosis and treatment then the accurate performance of our code takes on a fundamentally different level of importance.

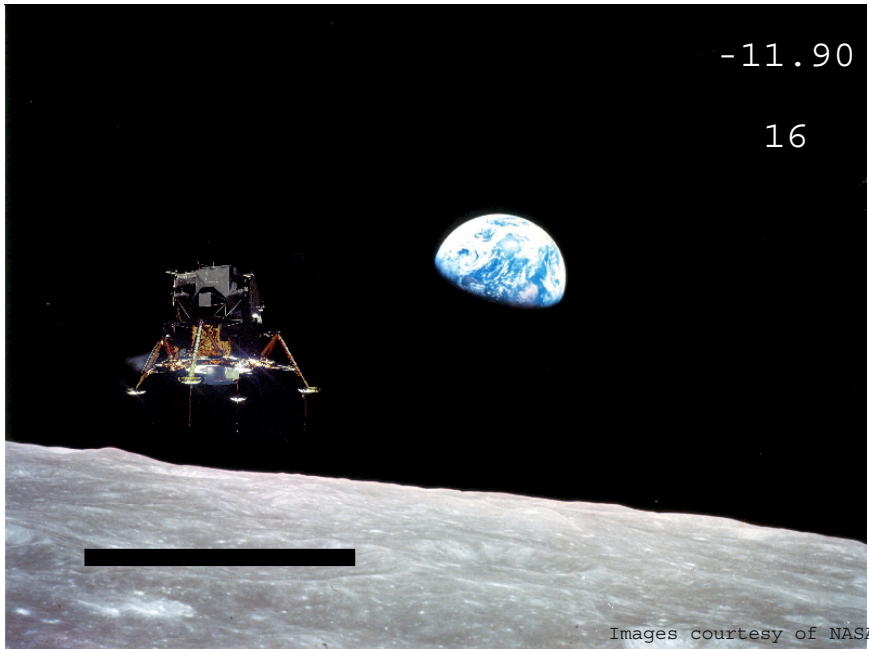


Fig. 1.25: Controlled lunar descent with limited fuel availability. Remaining fuel is indicated in number of times vertical thrusters may be burned, initially set to 20. Images courtesy of NASA, lunar module photo credit Michael Collins.

Lab134: Limited Fuel

While we can press the spacebar as often as we like in real life the lunar module contained a limited amount of fuel. Code Listing 1.17 shows one way to model limited fuel in our code and [Figure 1.25](#) shows the current fuel level communicated to the user; when the number reaches 0 then we can no longer fire the thrusters.

Code Listing 1.17: Limited fuel where we track the current level remaining.

```
def spacebar(evnt):
    global vy,u
    #
    if u>0:
        #
        vy+=1.0
        u -=1
```

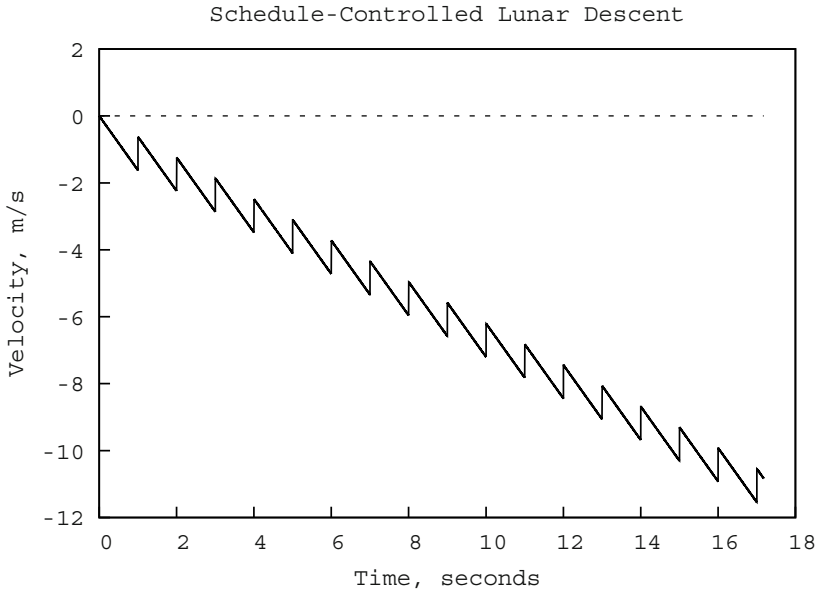


Fig. 1.26: As shown, vertical thrusters fire at regular intervals but we do not even use all our fuel. Upon modification of the firing schedule your author's best effort was a $v_y = -0.17$ m/s landing, under 0.5 mph.

Lab135: Optimal Strategy

Do you think the target rectangle, fuel indicator, and velocity text are easy to use? An alternative would be to define a strategy in the code and thus automate the entire process (i.e., no longer use the spacebar at all). Code Listing 1.18 specifies a firing schedule based on current elapsed time with results, crash, reported in [Figure 1.26](#).

Code Listing 1.18: Control of vertical thrusters with a firing schedule.

```
def tick():
    global t,y,vy,i
    #
    if i<len(fs) and t>=fs[i]:
        vy+=1.0
        i +=1
    #
    #
    i=0
    fs=[ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,10.0,\
        11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,20.0]
```




<http://www.springer.com/978-1-4614-1887-0>

Applied Computer Science

Torbert, S.

2012, X, 202 p., Hardcover

ISBN: 978-1-4614-1887-0