3.2: Numerical Techniques

Engineering majors are a majority of the students in the kind of physics course for which this book is designed, so most likely you fall into that category. Although you surely recognize that physics is an important part of your training, if you've had any exposure to how engineers really work, you're probably skeptical about the flavor of problem-solving taught in most science courses. You realize that not very many practical engineering calculations fall into the narrow range of problems for which an exact solution can be calculated with a piece of paper and a sharp pencil. Real-life problems are usually complicated, and typically they need to be solved by number-crunching on a computer, although we can often gain insight by working simple approximations that have algebraic solutions. Not only is numerical problem-solving more useful in real life, it's also educational; as a beginning physics student, I really only felt like I understood projectile motion after I had worked it both ways, using algebra and then a computer program. (This was back in the days when 64 kilobytes of memory was considered a lot.)

In this section, we'll start by seeing how to apply numerical techniques to some simple problems for which we know the answer in "closed form," i.e., a single algebraic expression without any calculus or infinite sums. After that, we'll solve a problem that would have made you world-famous if you could have done it in the seventeenth century using paper and a quill pen! Before you continue, you should read Appendix 1 on page 908 that introduces you to the Python programming language.

First let's solve the trivial problem of finding how much time it takes an object moving at speed v to travel a straight-line distance dist. This closed-form answer is, of course, dist/v, but the point is to introduce the techniques we can use to solve other problems of this type. The basic idea is to divide the distance up into n equal parts, and add up the times required to traverse all the parts. The following Python function does the job. Note that you shouldn't type in the line numbers on the left, and you don't need to type in the comments, either. I've omitted the prompts >>> and ... in order to save space.

```
import math
def time1(dist,v,n):
  x=0 # Initialize the position.
  dx = dist/n # Divide dist into n equal parts.
  t=0 # Initialize the time.
  for i in range(n):
    x = x+dx # Change x.
    dt=dx/v # time=distance/speed
    t=t+dt # Keep track of elapsed time.
  return t
```

How long does it take to move 1 meter at a constant speed of 1 m/s? If we do this,

```
>>> print(time1(1.0,1.0,10)) # dist, v, n
0.99999999999999989
```

Python produces the expected answer by dividing the distance into ten equal 0.1-meter segments, and adding up the ten 0.1-second times required to traverse each one. Since the object moves at constant speed, it doesn't even matter whether we set n to 10, 1, or a million:

```
>>> print(time1(1.0,1.0,1)) # dist, v, n
1.0
```

Now let's do an example where the answer isn't obvious to people who don't know calculus: how long does it take an object to fall through a height h, starting from rest? We know from example 8 on page 83 that the exact answer, found using calculus, is *[Math Processing Error]*. Let's see if we can reproduce that answer numerically. The main difference between this program and the previous one is that now the velocity isn't constant, so we need to update it as we go along. Conservation of energy gives *[Math Processing Error]* for the velocity *[Math Processing Error]* at height *[Math Processing Error]*, so *[Math Processing Error]*. (We

choose the negative root because the object is moving down, and our coordinate system has the positive *[Math Processing Error]* axis pointing up.)

```
import math
def time2(h,n):
  g=9.8 # gravitational field
  y=h # Initialize the height.
  v=0 # Initialize the velocity.
  dy = -h/n # Divide h into n equal parts.
  t=0 # Initialize the time.
  for i in range(n):
    y = y+dy # Change y. (Note dy<0.)
    v = -math.sqrt(2*g*(h-y)) # from cons. of energy
    dt=dy/v # dy and v are <0, so dt is >0
    t=t+dt # Keep track of elapsed time.
  return t
```

For h=1.0 m, the closed-form result is *[Math Processing Error]*. With the drop split up into only 10 equal height intervals, the numerical technique provides a pretty lousy approximation:

```
>>> print(time2(1.0,10)) # h, n
0.35864270709233342
```

But by increasing n to ten thousand, we get an answer that's as close as we need, given the limited accuracy of the raw data:

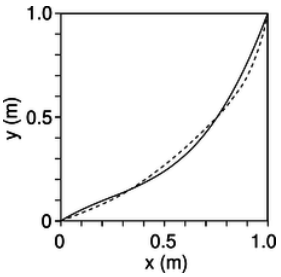
```
>>> print(time2(1.0,10000)) # h, n
0.44846664060793945
```

A subtle point is that we changed y in line 9, and *then* on line 10 we calculated v, which depends on y. Since y is only changing by a ten-thousandth of a meter with each step, you might think this wouldn't make much of a difference, and you'd be almost right, except for one small problem: if we swapped lines 9 and 10, then the very first time through the loop, we'd have v=0, which would produce a division-by-zero error when we calculated dt! Actually what would make the most sense would be to calculate the velocity at height y and the velocity at height y+dy (recalling that dy is negative), average them together, and use that value of y to calculate the best estimate of the velocity between those two points. Since the acceleration is constant in the present example, this modification results in a program that gives an exact result even for n=1:

```
import math
def time3(h,n):
  g=9.8
  y=h
  v=0
  dy = -h/n
  t=0
  for i in range(n):
    y_old = y
    y = y+dy
    v_old = math.sqrt(2*g*(h-y_old))
    v = math.sqrt(2*g*(h-y))
    v_avg = -(v_old+v)/2.
```

```
    dt=dy/v_avg
    t=t+dt
  return t
```

```
>>> print(time3(1.0,1)) # h, n
0.45175395145262565
```



a / Approximations to the brachistochrone curve using a third-order polynomial (solid line), and a seventh-order polynomial (dashed). The latter only improves the time by four milliseconds.

Now we're ready to attack a problem that challenged the best minds of Europe back in the days when there were no computers. In 1696, the mathematician Johann Bernoulli posed the following famous question. Starting from rest, an object slides frictionlessly over a curve joining the point *[Math Processing Error]* to the point *[Math Processing Error]*. Of all the possible shapes that such a curve could have, which one gets the object to its destination in the least possible time, and how much time does it take? The optimal curve is called the *brachistochrone*, from the Greek “short time.” The solution to the brachistochrone problem evaded Bernoulli himself, as well as Leibniz, who had been one of the inventors of calculus. The English physicist Isaac Newton, however, stayed up late one night after a day's work running the royal mint, and, according to legend, produced an algebraic solution at four in the morning. He then published it anonymously, but Bernoulli is said to have remarked that when he read it, he knew instantly from the style that it was Newton --- he could “tell the lion from the mark of his claw.”

Rather than attempting an exact algebraic solution, as Newton did, we'll produce a numerical result for the shape of the curve and the minimum time, in the special case of *[Math Processing Error]*=1.0 m and *[Math Processing Error]*=1.0 m. Intuitively, we want to start with a fairly steep drop, since any speed we can build up at the start will help us throughout the rest of the motion. On the other hand, it's possible to go too far with this idea: if we drop straight down for the whole vertical distance, and then do a right-angle turn to cover the horizontal distance, the resulting time of 0.68 s is quite a bit longer than the optimal result, the reason being that the path is unnecessarily long. There are infinitely many possible curves for which we could calculate the time, but let's look at third-order polynomials,

[Math Processing Error]

where we require *[Math Processing Error]* in order to make the curve pass through the point *[Math Processing Error]*. The Python program, below, is not much different from what we've done before. The function only asks for *[Math Processing Error]* and *[Math Processing Error]*, and calculates *[Math Processing Error]* internally at line 4. Since the motion is two-dimensional, we have to calculate the distance between one point and the next using the Pythagorean theorem, at line 16.

```
import math
def timeb(a,b,c1,c2,n):
  g=9.8
  c3 = (b-c1*a-c2*a**2)/(a**3)
  x=a
  y=b
  dx = -a/n
  t=0
```

```
  for i in range(n):
    y_old = y
    x = x+dx
    y = c1*x+c2*x**2+c3*x**3
    dy = y-y_old
    v_old = math.sqrt(2*g*(b-y_old))
    v = math.sqrt(2*g*(b-y))
    v_avg = (v_old+v)/2.
    ds = math.sqrt(dx**2+dy**2) # Pythagorean thm.
    dt=ds/v_avg
    t=t+dt
  return t
```

As a first guess, we could try a straight diagonal line, *[Math Processing Error]*, which corresponds to setting *[Math Processing Error]*, and all the other coefficients to zero. The result is a fairly long time:

```
>>> a=1.
>>> b=1.
>>> n=10000
>>> c1=1.
>>> c2=0.
>>> print(timeb(a,b,c1,c2,n))
0.63887656499994161
```

What we really need is a curve that's very steep on the right, and flatter on the left, so it would actually make more sense to try *[Math Processing Error]*:

```
>>> c1=0.
>>> c2=0.
>>> print(timeb(a,b,c1,c2,n))
0.59458339947087069
```

This is a significant improvement, and turns out to be only a hundredth of a second off of the shortest possible time! It's possible, although not very educational or entertaining, to find better approximations to the brachistochrone curve by fiddling around with the coefficients of the polynomial by hand. The real point of this discussion was to give an example of a nontrivial problem that can be attacked successfully with numerical techniques. I found the first approximation shown in figure a,

[Math Processing Error]

by using the program listed in appendix 2 on page 911 to search automatically for the optimal curve. The seventh-order approximation shown in the figure came from a straightforward extension of the same program.

Contributors and Attributions