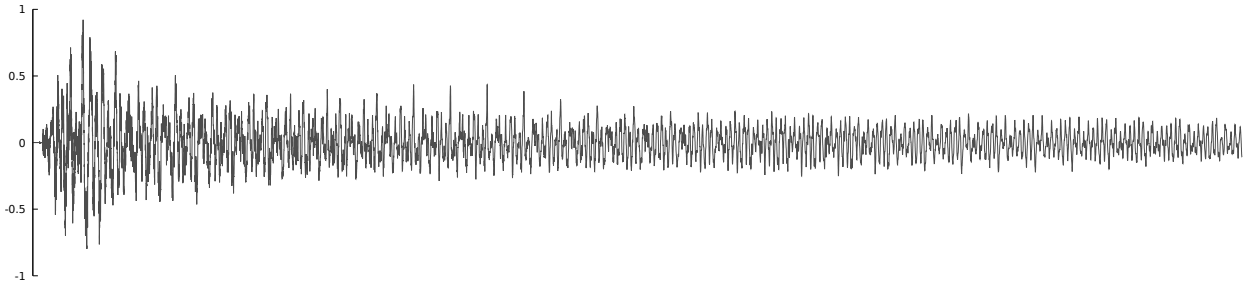
CPSC 231 ASSIGNMENT #6

Guitar Heroine

The goal of your final assignment in CPSC 231 is to explore computational algorithms for digital synthesis of sounds, instruments, and music. In the process, you'll practice using and creating data types in Python, and even spend some time writing a little recursive function.

Audio is encoded digitally on computers as a discrete *signal*, or a sequence of discrete *samples*, that represents the sound pressure level over time. In other words, it's no more than a big, one-dimensional array of numbers. Periodic repetitions and patterns in the numbers give the sound its characteristic tone and timbre. For example, the sound of a strum on an acoustic guitar looks like this when plotted as a graph:



If we can come up with an algorithm that generates these numbers so that the result has a meaningful tone and timbre, we can synthesize the sound of musical instruments! In 1983, Kevin Karplus and Alex Strong described a very simple algorithm that can generate the sound of plucking a string on a guitar.[1] Despite its simplicity, it produces an incredibly rich and realistic sound. You'll end your work in CPSC 231 this semester by implementing the Karplus-Strong algorithm to turn your computer into a live acoustic guitar synthesizer that you can play music on.

[1] Kevin Karplus and Alex Strong. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7 (2):43–55, 1983

Due Dates

Individual component	Friday, November 30, 11:59 PM
Paired component	Friday, December 7, 11:59 PM

Individual Component

As with all your previous assignments, we'll use the individual component to "warm up" a little bit before going about the live synthesis of the sound of an acoustic guitar. Here, we'll first use a recursive algorithm to generate a one-dimensional fractal noise signal. This kind of noise can create patterns that are more similar to natural phenomena than noise consisting of uniformly-distributed random numbers (*i.e.* white noise). We will stream this noise to the audio output of your computer to synthesize the sound of ocean waves crashing on the shore.

Before you begin, it may be worthwhile to review the sections in the course text that pertain to digital audio and music (pp. 171–175 and 231–235). You can use the exact same programming environment you set up for previous assignments to complete this assignment. Create and submit a separate Python program for each problem.

Problem 1: Brownian Bridge

A *Brownian bridge* is a stochastic process (think of this as a random function over time) that models a random walk connecting two points. In our case, the bridge is a function of time, and if we fix its value at an initial and a final time point, $b(t_0)$ and $b(t_1)$, we can compute the in-between values with the following recursive process:

1. Compute the midpoint of the interval, $y_m = \frac{1}{2}\left(b(t_0) + b(t_1)\right)$.
2. Add to the midpoint value a random value δ, drawn from a normal distribution[2] with zero mean and given variance, and assign the sum to the bridge function: $b(t_m) \leftarrow y_m + \delta$.
3. Recur on the subintervals (t_0, t_m) and (t_m, t_1), dividing the variance for the next level by a given scale factor.

[2] You can use `random.gauss()` or `random.normalvariate()` to generate such a random value, providing the mean and standard deviation (square root of variance) as arguments.

The shape of the function is controlled by two parameters: the *volatility* and the *Hurst exponent*. Volatility is the initial value of the variance, and controls how far the random walk strays from the straight line connecting the fixed endpoints. The Hurst exponent, H, controls the smoothness of the result. The variance of the distribution from which the random value δ is drawn is divided by 2^{2H} at each recursive level. When $H = 0.5$, the variance is halved after each recursion, and the result is a true Brownian bridge that models Brownian motion between the endpoints.

Note that Program 2.3.5 in the course text uses a recursive function to plot a Brownian bridge on the screen. Rather than drawing a graph, this problem requires you to fill an array with values of the function. Studying that example would certainly be a good idea, and you may complete this problem by starting from the textbook example and modifying it if you like.

Write a recursive Python function to fill an array with numerical values that form a Brownian bridge between two specified endpoints. The function should take a reference to an array, a first and last index, a variance, and a scale factor as parameters. These would allow the function to fill the array according to the recurrence described. For example, your function may have a definition that looks like this:

```
def fill_brownian(a, i0, i1, variance, scale):
    """
    Recursive function that fills an array with values forming a
    Brownian bridge between indices i0 and i1 (exclusive).
    Assumes that a[i0] and a[i1] are values of the fixed endpoints.
    :param a:  Array to fill with Brownian bridge.
    :param i0:  Index of first endpoint.
    :param i1:  Index of second endpoint
    :param variance:  Variance of the normal distribution from
                      which to sample a displacement.
    :param scale:  Scale factor to reduce variance for next level.
    """
```

When you've completed the function, write a program to test it out. Call your recursive function to fill an array with 129 elements, the first and last of which are anchored at a value of 0.0. Use a volatility of 0.05 and allow the Hurst exponent to be specified as a command line argument. Then visualize the contents of your array by drawing a graphical line plot of the values to the screen. You may either write your own code, use some of the code we wrote in class or that's found in the course text, or use the `stdstats` module from the booksite library to draw the plot.

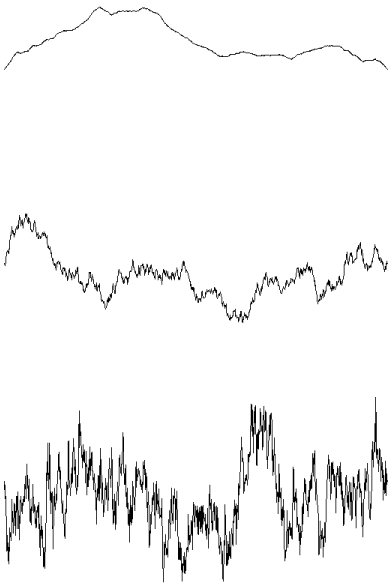


Figure 1: Line plots of 129 values of a Brownian bridge with Hurst exponents of 1.0, 0.5, and 0.2, respectively from top to bottom.

Inputs: The Hurst exponent for your Brownian bridge function, specified as a command line argument. For example, you might run your program with the following invocation:

```
$ python3 my-p1.py 0.5
```

Outputs: A line plot of your randomly-generated Brownian bridge between the two zero endpoints. It should resemble those shown on the right when run with the corresponding Hurst exponent.

Problem 2: Ocean Waves

If you stream the Brownian bridge function values to your computer's audio output,[3] the result sounds like a pleasant, rumbling kind of noise. In fact, if you generate the values with a Hurst exponent of 0.5, the result is what audio engineers call "brown noise" or "red noise". This is distinct from "pink noise" and the higher-pitched and more commonly found "white noise", which is simply a stream of uniformly distributed random numbers.

Both kinds of noise are useful, and we can actually synthesize a half-decent sound of ocean waves crashing on the shore by oscillating between brown noise and white noise at a very low frequency. If $n_b(t)$ represents the sequence of brown noise values over time, and $n_w(t)$ represents white noise, we can achieve the ocean wave effect by

[3] You can use the booksite library function `stdaudio.playSamples(a)` to stream a sequence of samples in the array `a` to your computer's default audio output device. The values in `a` should be `floats` between -1.0 and +1.0.

blending them with a time-varying blend factor, $s(t)$, as follows:

$$y(t) = (1 - s(t))\, n_b(t) + s(t)\, n_w(t),$$
$$\text{with} \quad s(t) = \sin^6(\pi f t).$$

In this equation, t represents the time (in seconds), and f is the frequency at which the ocean waves are crashing on the shore.

Write a program that generates 20 seconds of crashing ocean waves using the method described above, then plays the sound through the computer's audio output. If you use the `stdaudio` module, the output sample rate is fixed to 44 100 Hz (samples per second), which means you will need to generate a total of $20 \times 44\,100 = 882\,000$ data values. Use your Brownian bridge function from Problem 1 to generate values for $n_b(t)$.[4] You can adjust any parameters within these equations that you'd like in order to synthesize a more realistic or more pleasant ocean sound, but perhaps we can suggest that you start with the following:

- Use a Hurst exponent of $H = 0.5$. Increase it slightly if you want lower-frequency noise, and decrease it for higher-frequency noise.
- A frequency of $f = 0.25$ will give you a wave crash every four seconds. Adjust as you like.
- The exponent of 6 on the blend factor, $s(t)$, controls the balance of the two noise types during each cycle. Increase the exponent for quicker wave crashes, but use even exponents to keep the blend factor non-negative.
- An amplitude of about 0.25 works well for the white noise component. In other words, generate random values between -0.25 and +0.25 for $n_w(t)$. Adjust as you like as well.

You may also want to introduce one more slowly-oscillating function to modulate the amplitude (volume) of your output sound to achieve the ultimate ocean wave simulator, but that's purely optional.

Inputs: None.

Outputs: The sound of synthetic ocean waves crashing to sooth and relax you for 20 seconds when you run your program.

[4] You will want to re-anchor your bridge to a value of 0.0 several times a second so that the values don't drift too far from the neutral zero-point. In other words, rather than generating all 882 000 samples with a single function call, you should generate a few thousand samples at a time, setting the two endpoints to 0.0 each time.

Paired Component

For once, the paired component of the assignment might actually be easier than the individual component! We know it's the last week of the term, and you're both busy and exhausted with all your other end-of-semester work, so this is our gift to you for making it all the way to the finish line. The end result is quite fantastic, especially for the amount of time required to complete it. And if you've really

got time and energy to spare at this point, we've got many bonus opportunities worth taking up.

You may still work with a partner of your own choosing to complete the remainder of this assignment, though remember that it may not be the same partner you've had for any of the first five assignments in this class. If you are solving these problems as a pair, one submission is sufficient for both students.

Problem 3: Guitar String

Start by implementing the Karplus-Strong algorithm to synthesize the sound generated from plucking a single guitar string. The algorithm maintains a *wavetable*, or an array of sample values, that is played back in a loop. Thus, if p is the size (number of samples) of the wavetable, then the nth sample played in the stream is the same as sample $n - p$:

$$y_n = y_{n-p}$$

If the wavetable is unmodified as it is played in a loop, the resulting sound is purely periodic (repeating at a frequency of 44 100 / p) and usually sounds very artificial. The key innovation of the Karplus-Strong algorithm is to apply a very simple modification to a wavetable after each sample is played, which simulates the propagation of vibration waves along a plucked string anchored at both ends, resulting in a realistic guitar sound. Rather than simply repeating the wavetable value p samples previous, it simply two successive values in the wavetable:

$$y_n = \frac{1}{2}\left(y_{n-p} + y_{n-p-1}\right)$$

We can make use of a *circular buffer* to implement this algorithm in software. This data structure allows us to examine and remove elements from its left, and to append elements to its right. One iteration of the Karplus-Strong algorithm is realized by examining the first two samples in the buffer, calculating their average (and optionally multiplying by a decay factor), appending the result to right, then removing the first value on the left to keep the length of the buffer.

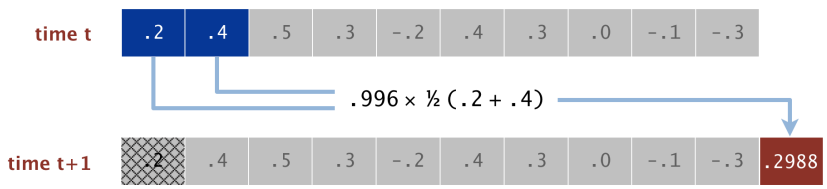


Figure 2: One iteration of the Karplus-Strong wavetable update, with a decay factor of 0.996, implemented on a circular buffer. (Image borrowed from the Princeton COS 126 assignment specification of the same name.)

Simulating a guitar string using the Karplus-Strong algorithm involves maintaining a dynamic wavetable of length p, with

$$p = \left\lceil \frac{44\,100}{f} \right\rceil,$$

where f is the desired frequency (pitch) of the guitar string, and the quotient is rounded *up* to obtain an integer value for p. We must also provide the following two operations:

Plucking the string. Simulating a pluck of the guitar string essentially amounts to exciting it at a multitude of frequencies. This can be accomplished by filling the wavetable with white noise.

Updating the wavetable and playing a sample. Apply the Karplus-Strong algorithm to compute a new sample for the wavetable, then stream that sample to the audio output.

This combination of data (the wavetable), and specific operations on the data, is a sign that a Python `class` would be a great way to create an abstraction of a guitar string. Create a class that implements a `GuitarString` data type with the API shown in Table 1. Start with a decay factor of 0.996 for your wavetable update, as shown in Figure 2 above, but feel free to adjust this value to get the sound you like. When you've completed your implementation of the `GuitarString` data type, test it with the client program listed on the right.

Test client for Problem 3:

```
import stddraw
import stdaudio
from picture import Picture
from guitarstring import GuitarString

a_string = GuitarString(440.00)
c_string = GuitarString(523.25)

# show a nice background picture
p = Picture('cpsc231-guitar.png')
stddraw.picture(p)
stddraw.show(0.0)

escape = False
while not escape:
    # check for and process events
    stddraw._checkForEvents()
    while stddraw.hasNextKeyTyped():
        key = stddraw.nextKeyTyped()
        if key == chr(27):
            escape = True
        elif key == 'a':
            a_string.pluck()
        elif key == 'c':
            c_string.pluck()

    # simulate and play strings
    y = a_string.tick()
    y += c_string.tick()
    stdaudio.playSample(y)
```

`GuitarString(f)`	Create a guitar string that vibrates at the given fundamental frequency (pitch), f.
`g.pluck()`	Pluck the guitar string by replacing all values in the wavetable with white noise. Use random values between -0.5 and +0.5.
`g.tick()`	Advance the Karplus-Strong simulation by one step, returning the sample value that was computed and added to the wavetable.

Table 1: The `GuitarString` application programming interface that you need to implement for Problem 3.

`deque()`	Create an empty deque.
`deque(iterable)`	Create a deque containing the elements of an *iterable*, such as a `list` or `str` object.
`d.append(x)`	Add x to the right side of the deque.
`d.popleft()`	Remove and return an element from the left side of the deque.
`d.clear()`	Remove all elements from the deque.
`d[index]`	A reference to the element at the given index (left is index 0).
`len(d)`	Number of elements in the deque.

Table 2: The application programming interface of the `deque` data type from the Python `collections` module.

Python's `collections` module includes a double-ended queue data type, called a `deque`, that you can use to store the wavetable. It supports the operations we need, and with the degree of efficiency

we need, to maintain a circular buffer of samples for the guitar string. The relevant portions of the `deque` API are shown in Table 2 above.

Inputs: Key presses ('a' or 'c') to play your two guitar strings while your program is running.

Outputs: Synthesized sounds of an acoustic guitar playing a concert A or a concert C coming out live from your computer.

Problem 4: Guitar Heroine

Now that you have an abstracted and tested guitar string, it's time to turn your computer into a guitar! Write a client similar to that from Problem 3, but allows you to play three full octaves of notes. Map keys on your keyboard to guitar strings corresponding to music notes as shown in Figure 3 below. When a key is pressed, pluck the guitar string corresponding to its note in your program.

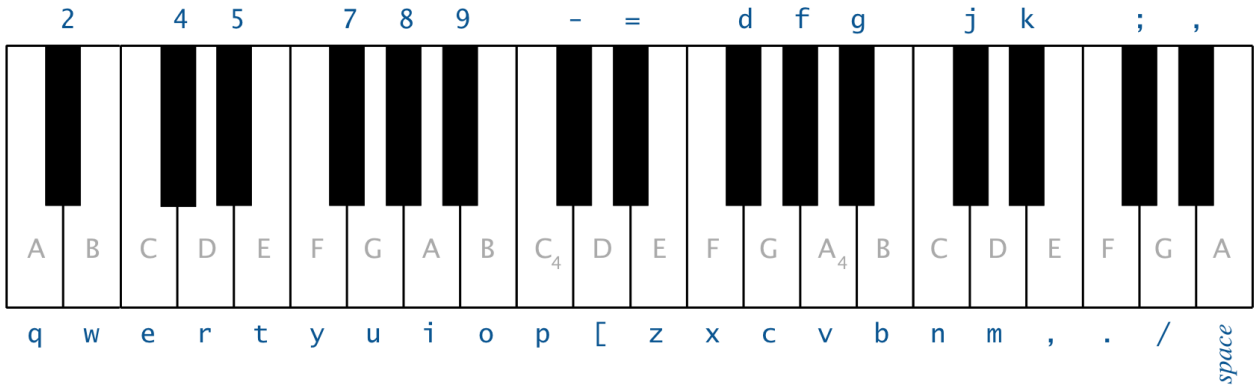


Figure 3: Mapping of keys on the computer keyboard to notes on the piano keyboard. (Image borrowed from the Princeton COS 126 assignment specification of the same name.)

The lowest note on this keyboard (A_2) corresponds to a fundamental frequency of 110 Hz. Each octave corresponds to a doubling of the fundamental frequency (*e.g.* A_3 = 220 Hz), and contains 12 semitones. We will use an equal-tempered chromatic scale, which means the frequencies of the semitones are equally spaced apart in a geometric series. Thus, the frequency of each semitone is $2^{1/12}$ times that of the previous. For example, the fundamental frequency of the first B♭ (which corresponds to the black piano key labelled '2') is

$$f_{B\flat} = 2^{\frac{1}{12}} f_A = 116.54 \text{ Hz}.$$

After completing this part, you should have a real-time synthesized interactive "guitar" that allows you to play notes (including chords) and songs by typing keys on your keyboard. When playing multiple strings, the resulting sound wave is the *superposition* of the sound waves generated by each individual string. Thus, the output sample at each time step is simply the sum of the individual samples from each guitar string.

Play a few songs to test out your brand new acoustic guitar. When you're satisfied with it, submit all the files needed to run your program, and you're done for the semester!

Inputs: Key presses playing some serious guitar rips while your program is running.

Outputs: Beautifully synthesized acoustic guitar music coming out live from your computer.

Creative Bonus

We're offering a creative bonus in each of three different categories for you to undertake, just in case this final assignment is too easy or too short for your liking. Here they are:

Best song. Create a client program of your `GuitarString` class that plays back a song you've saved to a file.[5] You may choose any song you'd like, though an ideal song would be about 1–4 minutes in duration. We will judge this bonus as if we were judging a guitarist's audition. You may want to extend your program to play notes with different amplitudes, to lengthen or shorten the notes, *etc.*, to really make your song sound good. We will also accept a song that you play live to us, using your program from Problem 4, as a submission to this bonus, if you prefer that over a saved file.

Best orchestra or electric guitar. You can also use the Karplus-Strong algorithm, with a few modifications, to synthesize sounds for instruments with many different timbres. Drums and harp-like timbres have been described by Karplus and Strong in their original article,[6] which you can read for some ideas. Create a program that synthesizes the sounds of a few other instruments of your choice, and have your program play back a short musical piece (like an Apple GarageBand loop) that showcases your "virtual orchestra". Alternatively, turn your acoustic guitar into an electric guitar,[7] and we will award the bonus to the best-sounding electric guitar.

Best Guitar Heroine game. You thought we were going to have you clone the Guitar Hero game for this assignment, didn't you? Since you've already worked so hard to program a lot of games in CPSC 231 this semester, we thought we'd take it easy on you this time. If you still wanted to make a playable Guitar Hero game out of your synthesized guitar, you can, and we'll happily give you bonus credit for it (as long as it's fun to play).

Your TA will have liberty to award a bonus to the submission that he or she deems to have the most creative or artistic merit in each of these three categories. This could be judged in terms of musicality, realism of sound, quality of experience, entertainment value, or any combination of the aforementioned.

You may have heard or even experienced that Python programs are not known to be very quick. Up until now, computational efficiency has not been a concern for us, but apparently calculating sample values and updating the wavetable for 37 guitar strings, 44 100 times a second, can be a little too much for the Python interpreter to handle.

If you implemented your guitar string carefully, you should be able to run the full three octaves interactively on a modern desktop or laptop computer. However, if you're working on an older laptop computer, your guitar synthesizer may stutter when your program can't feed it samples fast enough. In such a case, we suggest that you develop and test your program using just one octave of strings, then switch to the full three octaves before submitting it. Alternatively, you can complete this assignment using the CPSC lab machines, but please don't forget to bring headphones!

[5] Programs 1.5.8 and 2.1.4 in the course text play a sequence of tones with given durations that are read from a file or input stream. A few sample songs are available on the book's web site. You can use the same format to encode your song, or use whatever format you find most convenient.

[6] Kevin Karplus and Alex Strong. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7 (2):43–55, 1983

[7] Charles R. Sullivan. Extending the Karplus-Strong algorithm to synthesize electric guitar timbres with distortion and feedback. *Computer Music Journal*, 14(3):26–37, 1990

Java Bonus

Here is one more opportunity for you to earn a bonus for being a multi-lingual computer scientist. We know this one is very obtainable because the students taking COS 126 at Princeton University do this same assignment (paired component) in Java. We will award you another bonus if you submit correct Java programs for Problems 3 and 4, in addition to your Python programs.[8]

You may set up your Java programs however you'd like, as long as it is convenient enough for your TA to run them on the CPSC lab machines. There exists a version of our course text that uses the Java programming language, titled *Computer Science: An Interdisciplinary Approach*.[9] It has corresponding support libraries that can be found at `https://introcs.cs.princeton.edu/java/home/`. If you're not sure where to start with Java, you may find it most convenient to follow the instructions from that booksite. You may also use functionality from their Java Booksite Library if that helps you to create Java-language equivalents of your Python programs.

[8] Note that you should consider your Python programs to be your de facto solutions for this assignment. You will not get credit for solving these problems unless your Python solutions are correct.

[9] Robert Sedgewick and Kevin Wayne. *Computer Science: An Interdisciplinary Approach*. Addison-Wesley, 2016

Submission

When you've completed the individual component, submit all the Python programs (`.py` files) that are necessary to run your scripts for Problems 1 and 2. After completing the paired component, submit your solutions to those problems as `.py` files as well. If you completed the creative bonus, ensure that you include any data files required for those program(s) to run correctly. If you completed the Java Bonus, submit the source of your Java programs (`.java` files) for Problems 3 and 4. Please ensure that your files are named descriptively, with the problem number included, so that your TA can easily see which program to run for each problem.

Use the University of Calgary Desire2Learn system[10] to submit your assignment work online. Log in using your UofC eID and password, then find our course, CPSC 231 L01 and L02, in the list. Then navigate to Assessments → Dropbox Folders, and find the folder for Assignment #6 here. Upload your programs for problems 1 and 2 in the individual folder, and your other programs in the paired folder (if your partner hasn't already done so). One submission will suffice for each pair of students.

If you are using one or more of your grace "late days" for this assignment, indicate how many you used in the note accompanying your submission. Remember that late days will be counted against *both* partners in the paired component. If you completed any of the bonus problems, please indicate that here as well.

[10] `http://d2l.ucalgary.ca`

Credits

This wouldn't be a proper course taught with a Princeton-authored textbook if we did not complete at least one Princeton assignment! The paired component of this assignment was developed by Andrew Appel, Jeff Bernstein, Maia Ginsburg, Ken Steiglitz, Ge Wang, and Kevin Wayne for the COS 126 course taught at Princeton University. Original materials for this assignment can be obtained from `https://introcs.cs.princeton.edu/java/assignments/guitar.html`.