Overview

When a guitar string is plucked, the string vibrates and creates sound. The length of the string determines its *fundamental frequency* of vibration. We model a guitar string by sampling its *displacement* (a real number between $-\frac{1}{2}$ and $+\frac{1}{2}$) at N equally-spaced points (in time), where N equals the *sampling rate* (44100 Hz) divided by the fundamental frequency (rounded to the nearest integer). We store these displacement values in a data structure that we will refer to as a *ring buffer*.

The excitation of the string can contain energy at any frequency. We simulate the excitation by filling the buffer with white noise: set each of the N sample displacements to a random real number between $-\frac{1}{2}$ and $+\frac{1}{2}$.

After the string is plucked, the string vibrates. The pluck causes a displacement which spreads wave-like over time. The Karplus–Strong algorithm simulates this vibration by maintaining a ring buffer of the N samples: the algorithm repeatedly deletes the first sample from the buffer and adds to the end of the buffer the average of the first two samples, scaled by an energy decay factor of 0.996.

Why it works

The two primary components that make the Karplus–Strong algorithm work are the ring buffer feedback mechanism and the averaging operation.

- The ring buffer models the medium (a string tied down at both ends) in which the energy travels back and forth. The length of the ring buffer determines the fundamental frequency of the resulting sound. Acoustically, the feedback mechanism reinforces only the fundamental frequency and its harmonics (frequencies at integer multiples of the fundamental). The energy decay factor (0.996 in this case) models the slight dissipation in energy as the wave makes a roundtrip through the string.
- The averaging operation serves as a gentle *low-pass filter* (which removes higher frequencies while allowing lower frequencies to pass, hence the name). Because it is in the path of the feedback, this has the effect of gradually attenuating the higher harmonics while keeping the lower ones, which corresponds closely with how actually plucked strings sound.

Setup

Download all starter files from this link:

Midterm Project starter files[1]

[1] The MD5 checksum for this file is `6cff3485754d9ec8c8976bfcbd265157`. You can verify your download by searching online for "online MD5 checker" or using `md5sum`.

You will need to set up a Python *virtual environment*, which allows us to install Python packages without messing up the existing system Python installation. To do this, run the command `python -m venv guitarenv`. This will produce a folder called `guitarenv`. Now your project has its own virtual environment. Generally, before you start using it, you'll first activate the environment by executing a script that comes with the installation:

- In Linux/macOS, run `source guitarenv/bin/activate`.
- In Windows, run `guitarenv\Scripts\activate`.

Once you can see the name of your virtual environment (`guitarenv` in this case) in your terminal, then you know that your virtual environment is active.

Next, install the packages used by this project by running `pip install -r requirements.txt`. This file lists the packages, and more importantly, their respective version numbers, which ensures consistency.

Verify that the packages have been installed correctly by running the command `pip freeze`. It should look something like this:

```
brian@thonkpad ~/classes/csci30/projects/guitar
(guitarenv) % pip freeze
numpy==2.1.0
pygame==2.6.0
```

> **Important!** If you are having trouble setting up the virtual environment, please don't proceed further. Don't hesitate to ask for help!

Provided as part of the starter files is the `stdaudio` library, which serves as a convenient wrapper for low-level audio synthesis functions from PyGame. To test whether the `stdaudio` library works in your machine, you can run `python stdaudio.py` in the terminal (and additionally, you may want to turn down the volume first since it may be loud). You should hear a short tune being played.

For macOS users, you may need to install Homebrew and the Xcode Command Line Tools, and you might also need to install the SDL dependencies required for PyGame.[2]

Once you are done using the virtual environment, you can deactivate it by running the `deactivate` command. After executing the `deactivate` command, your terminal returns to normal. This change means that you've exited your virtual environment. If you interact with Python or `pip` now, you'll interact with your globally configured Python environment.

If you want to go back into a virtual environment that you've created before, you again need to run the `activate` script of that virtual environment.

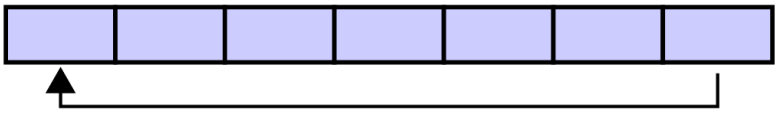
[2] https://www.pygame.org/wiki/MacCompile

Part 1: Ring buffer

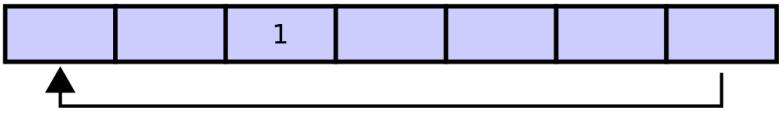
Your first task is to create a data type to model the ring buffer. Write a class named `RingBuffer` that implements the following interface:

```
class RingBuffer:
    def __init__(self, capacity: int):
        # create an empty ring buffer, with given max capacity
    def size(self) -> int:
        # return number of items currently in the buffer
    def is_empty(self) -> bool:
        # is the buffer empty (size equals zero)?
    def is_full(self) -> bool:
        # is the buffer full (size equals capacity)?
    def enqueue(self, x: float):
        # add item x to the end
    def dequeue(self) -> float:
        # return and remove item from the front
    def peek(self) -> float:
        # return (but do not delete) item from the front
```

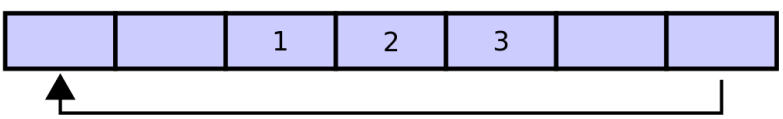
Since the ring buffer has a known maximum capacity, implement it using a fixed array of that length. For efficiency, use a *circular queue*. To see how it works, here is a 7-element buffer:



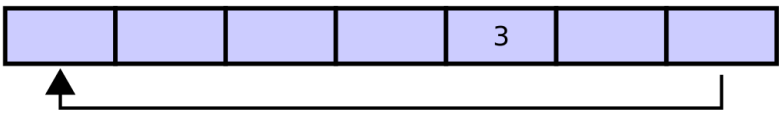
Assume that a 1 is written into the middle of the buffer (exact starting location does not matter in a ring buffer):



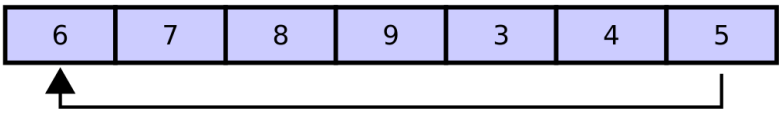
Then assume that two more elements are added (2 and 3) which get appended after the 1. Here, it is important that the 2 and 3 are placed in the exact order and places shown:



If two elements are then removed from the buffer, the oldest two values inside the buffer are removed. The two elements removed, in this case, are 1 and 2, leaving the buffer with just a 3:



If we then enqueue 4, 5, 6, 7, 8, 9, the ring buffer is now as shown below:



Note that the 6 was enqueued at the leftmost entry of the array (i.e., the buffer wraps around, like a ring). At this point, the ring buffer is full, and if another `enqueue()` is performed, then an exception will occur.

Maintain one variable `_front` that stores the index of the least recently inserted item. Maintain a second variable `_rear` that stores the index one beyond the most recently inserted item. To insert an item, put it at index `_rear` and increment `_rear`. To remove an item, take it from index `_front` and increment `_front`. When either index equals `capacity`, make it wrap-around by changing the index to 0.

Implement `RingBuffer` to raise an exception if the client attempts to `dequeue()` or `peek()` from an empty buffer or `enqueue()` into a full buffer.

A test harness (which is a proper subset of what will be actually used for grading) named `ringbuffer_tester.py` is provided to test your ring buffer implementation, though you are also highly encouraged to write your own tests. To run the test harness, simply run the command `python ringbuffer_tester.py`.

```
brian@thonkpad ~/classes/csci30/projects/guitar
(guitarenv) % python ringbuffer_tester.py
..........
----------------------------------------------------------------------
Ran 10 tests in 0.001s

OK
```

> **Important!** You are required to use a fixed-size `list` to implement your ring buffer, so you are not allowed to use a `collections.deque` object for this purpose.

Part 2: Guitar string

Next, create a data type to model a vibrating guitar string, which uses `RingBuffer` to replicate the sound of a plucked string. Write a class named `GuitarString` that implements the following interface:

```
class GuitarString:
    def __init__(self, frequency: float):
        # create a guitar string of the given frequency,
        #   using a sampling rate of 44100 Hz
    def make_from_array(cls, init: list[int]):
        # create a guitar string whose size and initial values are
        #   given by the array
    def pluck(self):
        # set the buffer to white noise
    def tick(self):
        # advance the simulation one time step
    def sample(self) -> float:
        # return the current sample
    def time(self) -> int:
        # return the number of ticks so far
```

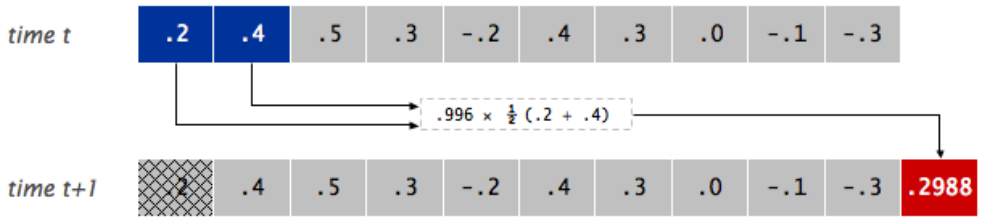
The constructor creates a `RingBuffer` of the desired capacity N (sampling rate 44100 Hz divided by frequency, rounded up to the nearest integer), and initializes it to represent a guitar string at rest by enqueueing N zeros.

The `make_from_array(init)` method creates a `RingBuffer` of capacity equal to the size of the array `init`, and initializes the contents of the buffer to the values in the array `init`. For this project, its main purpose is for debugging and grading; you won't actually use this yourself. **The starter files already provide an implementation of this function, so no need to implement your own.**

The `pluck()` method sets the buffer to white noise by replacing the N items in the ring buffer with N random values from $-\frac{1}{2}$ to $+\frac{1}{2}$. You may use the built-in `random.uniform()` function to generate random real numbers for this purpose.

The `tick()` method applies the Karplus–Strong update by:

1. deleting the sample at the front of the ring buffer, and
2. adding to the end of the ring buffer the average of the first two samples, multiplied by the energy decay factor.



the Karplus-Strong update

The `sample()` method simply returns the value of the item at the front of the ring buffer.

The `time()` method simply returns the number of times `tick()` was called.

A test harness (which is a proper subset of what will be actually used for grading) called `guitarstring_tester.py` is also provided to test your guitar string implementation, though, again, you are also highly encouraged to write your own tests.

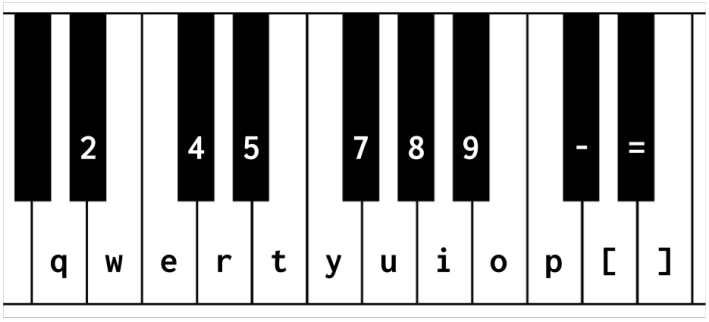
Part 3: Guitar player

Also provided with the starter files is `guitarlite.py`, a sample client that plays the guitar in real-time, using the keyboard to input notes. When the user types the lowercase letter a or c, the program plucks the corresponding string. Since the combined result of several sound waves is the superposition of the individual sound waves, we play the sum of all string samples.

Write a program `guitar.py` that is similar to `guitarlite.py`, but supports a total of 20 notes on the chromatic scale from 220 Hz to about 660 Hz. In general, make the ith character of the string below play the ith note.

```
keyboard = "q2we4r5ty7u8i9op-[=]"
```

This keyboard arrangement imitates a piano keyboard: the "white keys" are on the `qwerty` row and the "black keys" on the `12345` row of the keyboard.



The ith character of the string corresponds to a frequency of $440 \times 1.059463^{i-12}$ Hz,[3]

[3] In case you're wondering, 1.059463 is just the twelfth root of 2. This number represents the frequency ratio of a semitone in twelve-tone equal temperament.

so that the character `q` is close to 220 Hz, `i` is close to 440 Hz, and `]` is approximately 660 Hz.

Please don't even think of including 20 individual `GuitarString` variables or a 20-way if statement! Instead, create a list of 20 `GuitarString` objects and use `keyboard.index(key)` (or something similar) to figure out which key was typed. Make sure your program does not crash if a key is played that is not one of your 20 notes.

Other requirements

Project quiz

Before you start implementing the project, you will first need to answer a quiz about the project, which assesses whether you have fully understood everything in the project specs.

The project quiz will be made available on Canvas starting **September 16**, and will be due on **September 28 at 8 PM**. You will only have one attempt to answer the quiz, but you may submit anytime before the due date. Take note that **the project quiz is to be done individually**, so no collaboration will be allowed in this case.

Submission

Submit the following files:

- The Python source files `ringbuffer.py`, `guitarstring.py`, `guitar.py`
- A fully-accomplished Certificate of Authorship (use whichever version is appropriate), with the filename `guitar-[ID1]-coa.pdf` or `guitar-[ID1]-[ID2]-coa.pdf` or `guitar-[ID1]-[ID2]-[ID3]-coa.pdf` (without the brackets)
- **Do not** include the Python virtual environment folder and the `__pycache__` folder

Compress these files into a zip archive (`.zip`) or a gzipped tarball (`.tar.gz`) with the filename `guitar-[ID1]` or `guitar-[ID1]-[ID2]` or `guitar-[ID1]-[ID2]-[ID3]` (again, without the brackets) with the appropriate file extension, and submit it in the Canvas submission module for this project.

The midterm project is due on **September 30 (Monday) at 8 PM.** See the section below for the late submission policy.

Project defense

Upon submission, you will need to sign up for a project defense. Project defense days are scheduled from **September 30** to **October 4**. Defenses will be held at **F-220** and **F-222**. In case you don't know where those rooms are, drop by the DISCS office at F-208 and ask the staff.

(*Note:* Should the need arises, we are willing to open more slots beyond the specified dates, but we would prefer not doing so.)

Sign-ups will open on **September 23 (Monday) at 12 PM.** Go to the following link to sign up for a slot once it opens:

(link to be announced)

Slots will close three hours before their scheduled start time.

Have one of you in the group sign up for a slot using the link above. **Please make sure that everyone in your group will be present on your chosen time slot.** Also, please arrive about 5 to 10 minutes before your scheduled time slot. Groups that arrive late will be allowed to use only the remaining minutes of their slot.

The project defense will last **at most twenty (20) minutes**; this could end earlier if there are no major issues or problems with your project. No need to bring your devices for the defense. As such, **please ensure that you have already submitted on Canvas** (especially those who will be having the defense on Sep 30).

Grading

Your midterm project is graded using the following criteria:

- 15 points — project quiz (to be answered individually on Canvas)
- 30 points — `RingBuffer` correctness/efficiency (autograded)
- 20 points — `GuitarString` correctness/efficiency (autograded)
- 20 points — `guitar.py` functionality and code style
- 15 points — project defense

The last two criteria are graded on a **subtractive** scale. That means, the default score is 20 or 15 points respectively if there are no problems, but this will be subject to deduction for any issues/errors found.

Extra credit opportunity

Using your finished project, perform a short piece (about a minute long is fine) and then post a recording of your performance in the upcoming Canvas discussion thread. The same groupings for the project will be in effect, so perform as a group and all members will receive extra credit, or as solo for individual extra credit. Do this by **October 9, 2024 at 11:59 PM** to receive the extra credit.

You will not be required to physically appear in the video recording, and you may include people outside your group in your performance if you want.

Extra fun opportunity (not for extra credit)

> **Important!** Please make sure that your project works first! However, you may attempt these once your project has been graded.

Modify the Karplus–Strong algorithm to synthesize a different instrument. Consider changing the excitation of the string (from white-noise to something more structured) or changing the averaging formula (from the average of the first two samples to a more complicated rule) or anything else you might imagine. Again, you will not receive extra credit for it, but you may use it as the basis for your extra credit performance.

Alexander Strong (the "Strong" in Karplus–Strong) suggests a few simple variants you can try:

- The frequency formula in the project uses "perfect tuning" that doesn't sound equally good in every key. Instead, most musicians use *stretched tuning* that equalizes the distortions across all keys. To get stretched tuning, use the formula $440 \times \mathbf{1.05956}^{i-12}$ (note the different base). Try experimenting a bit with the base of the exponent to see what sounds best.
- Add additional keys to your keyboard string to play additional notes (higher or lower). Higher notes especially will benefit from stretched tuning. You will need to update the 12 in your frequency formula to change the frequency of the lowest note.
- Make the decay factor dependent on the string frequency. Lower notes should have a higher decay factor; higher notes should have a smaller decay. Try different formulas and see what sounds best.
- Flipping the sign of the new value before enqueueing it in `tick()` will change the sound from guitar-like to harp-like. You may want to play with the decay factors and note frequencies to improve the realism.
- Randomly flipping (with probability $\frac{1}{2}$) the sign of the new value before enqueueing it in `tick()` will produce a drum sound. You will need lower frequencies for the drums than for the guitar and harp, and will want to use a decay factor of 1.0 (no decay). The note frequencies for the drums should also be spaced further apart.
- Assign some keyboard keys to drums, others to guitar, and still others to harp (or any other instruments you invent) so you can play an ensemble.

Other matters

Groupings

You may work alone or you may work in groups of **at most three** members (this can be different from your pset partners). Your groupmates can come from different sections.

Despite this, a peer grading system will **not** be implemented. That means, with rare exceptions, all members of the group will receive the same grade (except for the project quiz) even if groupings ended up being somewhat inequitable. As per the syllabus, the instructors reserve the right to intervene and administer peer evaluations to a group/pair that is suspected to have issues regarding inequity of contributions. In such cases, the information from the peer evaluations will be used to determine the *individual* grade for the requirement in question.

Late submission policy

Late submissions are allowed but strongly discouraged. Submissions made after the due date will be penalized with an 85% cap on the grade. Any slots you have signed up for the project defense prior to submission will be disregarded.

Rescheduling a project defense

In situations where you or your groupmate (or all of you) became suddenly unavailable for your chosen time slot due to extenuating circumstances, you are advised to inform your instructor immediately so that necessary adjustments can be made. In most cases, you will be asked to reschedule your project defense.

You have at most 48 hours to sign up for another slot if asked to reschedule. Failure to reschedule your project defense may lead to a grade of zero for the project defense component of your grade.

Any questions?

Any questions, clarifications, and concerns about the midterm project specs should be directed to the `#midterm-project` channel on the Discord server, or via email or Canvas.

Changelog

- Sep 9: Initial release