Problem Set 10B: Room Acoustics

Please send back via NYU Classes

- A zip archive named as

`PS10B_<your name as FirstLast>.zip`

containing the C/C++ code files that implements all aspects of all problems.

Total points: 100

Points Allocation:

Problems 1 and 2 are 50 points each. The following will be considered:

- The program functions as intended to solve the problem
- The program has good programing style and structure
- The program has good error checking
- The program has informative comments

References

OpenGL and GLUT
http://www.lighthouse3d.com/tutorials/glut-tutorial/

Source image and parametric reverberation models
https://ccrma.stanford.edu/~jos/pasp/Early_Reflections.html

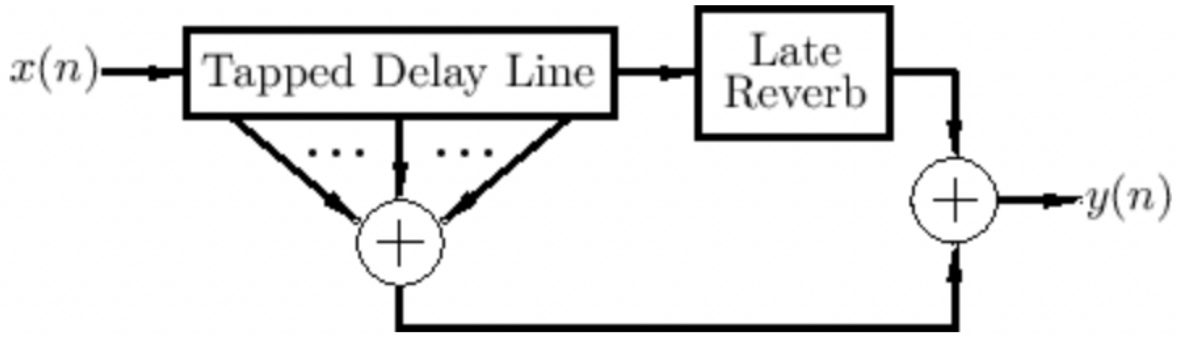
Parametric Reverberation
https://ccrma.stanford.edu/~jos/pasp/Schroeder_Allpass_Sections.html
https://ccrma.stanford.edu/~jos/pasp/Feedback_Comb_Filters.html#fig:fbcf
https://ccrma.stanford.edu/~jos/pasp/Schroeder_Reverberators.html

Overview

In problem 1, you will write a set of functions in the file **display_opengl.cpp** to display the room, source, listener and source images as a 3D model viewed on a 2D graphical display. It will provides controls to modify the display and reverberation parameters. All aspects ot the visual model are adjusted in real time in repose to keyboard control input. The file **display_opengl.cpp** will replace the file **display_ncurses.**cpp from the previous problem set. The programs are written in such a way that no code has to be changed to go from nCurses to OpenGL display (this is accomplished by the #defines for DISPLAY and PRINT). All other code in the previous problem set will be re-used to create the new program executable.

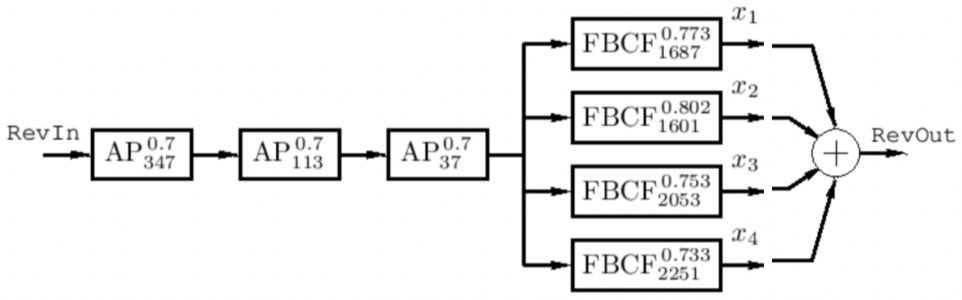
In problem 2, you will write a set of functions in **parametric_reverb.cpp** and **para_reverb,cpp** that implement a parametric model of the "reverb tail" of a room's reverberation. This portion of the reverberation typically is the result of many, many multi-bounce reflections to the point that there is no need to create each reflection but instead just generate a random and chaotic sound that has the appropriate decay envelope. This is added to the source images (from PS10A) to create a complete room reverberation model.

Our full reverberation model has this block diagram, where the "taps" off of the tapped delay line are the delays computed for each source image. These are summed to make an "early reflection" component of the reverberation, which you implemented in PS10A. In this problem set we add a model of "late reverberation" which will take as input the last delayed sample from the tapped delay line and whose output is added to the early reflection reverb model. This is done for every input signal sample.



The "late reverb" model we will use is the "Schroeder Reverberator," which is composed of two sets of blocks, "AllPass Filter" (AP) and "Feedback Comb Filter" (FBCF) as shown in the following figure:

The cascade of three allpass filters are realized as shown here:



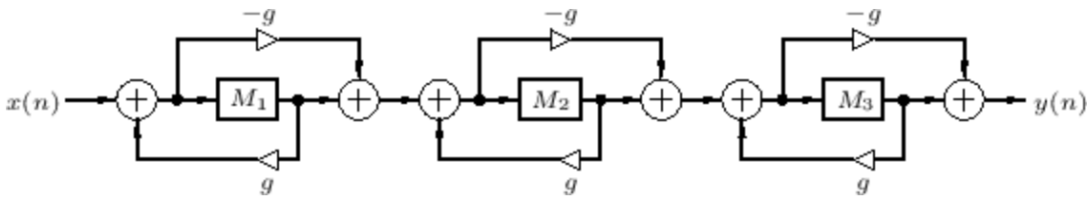


Figure 3.4: A cascade of three Schroeder allpass sections. A typical value for g is 0.7.

The delay-line lengths M_i are typically mutually prime and spanning successive orders of magnitude, *e.g.*, $1051, 337, 113$.

Each of the feedback comb filters are realized as shown here:

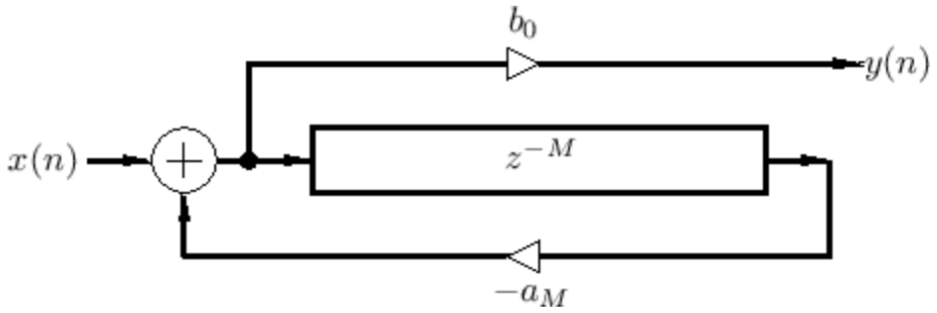


Figure 2.24: The feedback comb filter.

Note that the allpass filters and the feedback comb filters both employ a delay line. In fact, they have nearly identical structure, except that the output of the allpass filter is the input to the delay line multiplied by a coefficient (-g) and added to the delay line output, while output of the feedback comb filter is just the input to the delay line multiplied by a coefficient (b_0). We will exploit this common structure to create a single C++ Class, `paraReverb`, that realizes either allpass or feedback comb filters.

As a further note on efficiency, the `paraReverb` class will use a circular buffer to implement the delay line. In this way there is no need to "move" samples in a buffer, but only adjust a pointer to the "first" and "last" samples in the delay line.

What is supplied

In this problem set, you will use all of the code from the previous problem set (PS10A) except the file **display_ncurses.cpp**. For this problem, the instructor has supplied the following:

- **display_opengl.cpp** - The framework for functions call OpenGL functions and create and animate figure. You add the code
- **para_reverb.h** - Complete C++ header file. No need to add anything

- **para_reverb.cpp** -The framework for defining methods in class paraReverb
- **parametric_reverb.cpp** - Declares instances of paraReverb and and implements the full parametric reverberator. No need to add anything.
- **buildB.sh** - Bash script to build the complete program
- **signals** - Folder with example signals

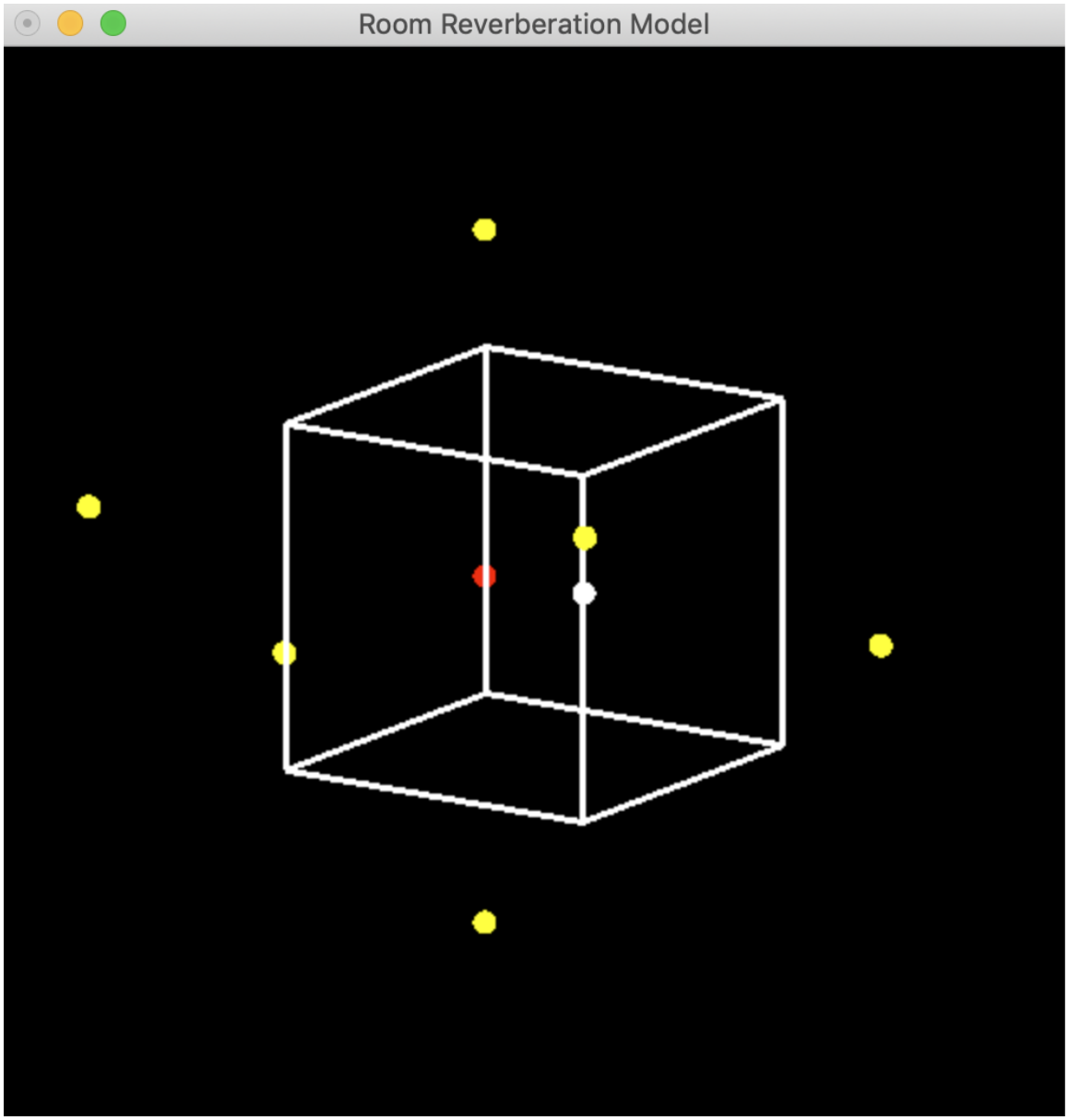
In addition, there is an executable that is the instructor's solution to the problem (for Mac platforms). Note that these programs are "C++" in that they have the ".cpp" extension and hence must use "g++" compiler (see build_nc.sh). Only the files that include **para_reverb.h** must be "aware" of the `paraReverb` class and so require the g++ compiler.

Example program output

Open a terminal window and go to your directory for this problem. Run the example executable:

```
./room_acoustics_demo signal/vega_mono.wav
```

The Susan Vega signal will play to your speaker. A new window will pop up. Initially this is all black (at least on my Mac), but just press one of the arrow keys and the following will display:



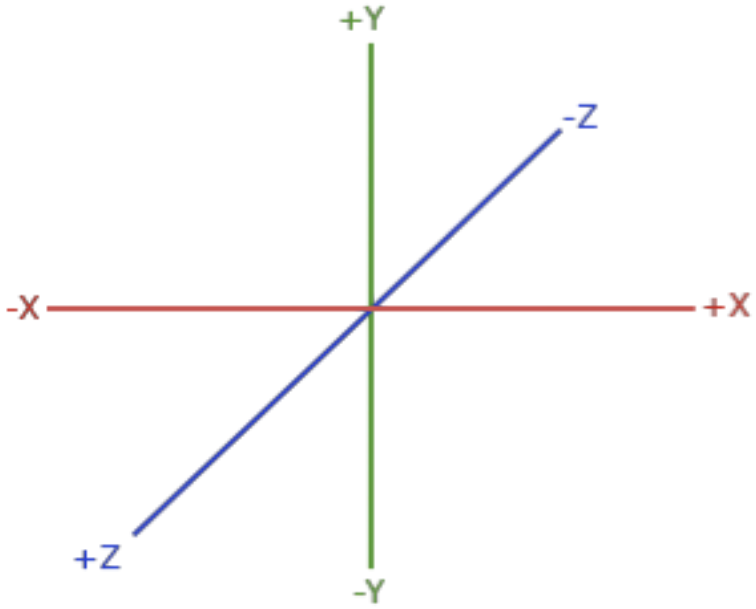
OpenGL

This figure is created using the OpenGL graphics library. What you see is a 2-D view of a 3-D model. It shows the room (a cube), the source (a red sphere), the listener (a white sphere) and the 6 images (yellow spheres). In your program you will be able to rotate your view of the 3D scene, move source and listener and see how the images move in response the movements of the source.

OpenGL uses a coordinate system as shown below, in which:

- The X-axis is left/right, with more negative values moving left and more positive values moving right.
- The Y-axis is up/down, with more negative values moving down and more

- positive values moving up.
- The Z-axis is front/back, with more negative values moving back and more positive values moving front (i.e. “out of the screen”).



This program has somewhat modified controls as compared to the one in the previous problem set PS10A, problem 3. This problem’s controls are shown here:

Key	**Result**
KEY_LEFT	Rotate angle of view around the x axis such that we see “bottom” of room.
KEY_RIGHT	Rotate angle of view around the x axis such that we see “top” of room
KEY_UP	Rotate angle of view around the y axis such room moves “counter clockwise.”
KEY_DOWN	Rotate angle of view around the y axis such room moves “clockwise.”
S or s	Enable movement of source
L or l	Enable movement of listener
X	Move (source or listener) in positive x direction by 1 unit (meter)
x	Move (source or listener) in negative x direction by 1 unit (meter)
Y	Move (source or listener) in positive y direction by 1 unit (meter)
y	Move (source or listener) in negative y direction by 1 unit (meter)
Z	Move (source or listener) in positive z direction by 1 unit (meter)
z	Move (source or listener) in negative z direction by 1 unit (meter)
+	Increase room size by 10%

-	Decrease room size by 10%
R	Increase room wall reflectivity by R_FAC.
R	Increase room wall reflectivity by R_FAC.
I	Set room->images = TRUE
i	Set room->images = FALSE
P	Set room->parametric = TRUE
p	Set room-> parametric = FALSE
D	Set room->delay = TRUE
d	Set room->delay = FALSE
Q, q or ESC	Exit program

Note:

- If S or s is pressed, then subsequent X,x,Y,y,Z,z keys affect the source location.
- If L or l is pressed, then subsequent X,x,Y,y,Z,z keys affect the listener location.
- If room size changes, re-calculate the source and listener positions and write to the Room structure.
- When increasing wall reflectivity, limit resulting reflectivity to no more than 1.0.
- When moving the source or the listener, limit the source or listener position so that source and listener are not closer than MIN_DIST to each other and neither is outside the perimeter of the room.

The same logic for constraining source or listener position as was used in PS10A can be re-used in this problem.
Try pressing 'l' or 'i' and 'P' or 'p' to enable (or disable) the source image or parametric reverberation tail signals. You should be able to hear the differences.

The Problems

The full program (e.g. the example executable) will be broken up into two sub-problems, which in turn are based on the code from PS10A. Each problem adds features and functionality to the previous problem. In addition, and perhaps most importantly, the sub-problem structure permits you to create and test your program in smaller chunks.

Problem 1

display_opengl.cpp
The file display_opengl.cpp contains #include code, declaration of global variables, function prototypes and the definition of functions

- display_openGL() This is complete code. It initialized OpenGL, sets up the display window, registers the callback functions and passes control to the "main loop" of openGL.

- display() This function draws the display in response to user interaction (i.e. key presses).
- normalKeys() This is the processing for all ASCII keys. After processing it refreshes the display.
- specialKeys() This is the processing for special keys, such as the keyboard "arrows.". After processing it refreshes the display.

In this problem you will complete the code in `normalKeys()` and `specialKeys()`.

`normalKeys()`

In `normalKeys()` implement the following. You could use a `switch()` statements or `if()` statements to conditionally execute the processing code. As was done in the `display_ncurses()` function from the previous problem set, copy the Pt structure values from room->listener to a temporary Pt l. However, since the source can also move in this function, copy the Pt structure values from room->source to a temporary Pt s.

- If S or s, set goblal variable sl = MOVE_SOURCE
- If L or l, set goblal variable sl = MOVE_LISTENER
- For X, x, Y, x, Z, z, move the source or listener according to the table above. Test `sl` to determine which of source or listener should be moved. Be sure to stay within the room boundary and not to get the source and listener too close (closer than MIN_DIST). Increment (decrement) the temporary Pt, test if the resulting position is allowed, and if it is, then copy the x, y, or z of the temporary Pt to that of the source or listener Pt.

The remainder of the user control is given to you:

- If + (or -) increase (or decrease) the room dimensions by 10%. This can be done by making the global variable `scale` larger or smaller. Examine function display() to see that the call to function
  ```
  glScalef( scale, scale, scale );
  ```
  uses this global variable to scale dimensions x, y, z equally.
- If R or r increase (or decrease) the wall reflectivity by R_FAC.
- If I or i enable (or disable) the source image signal
- If P or p enable (or disable) the parametric reverberation signal
- If D or d enable (or disable) the diagnostic source image printout
- If Q, q or ESC, quit the program by calling this code:
  ```
  glutDestroyWindow(wh);
  exit(0);
  ```
- Before returning from this function, refresh the display using this code:
  ```
  // Request display update
  glutPostRedisplay();
  ```

`specialKeys()`

In `specialKeys()` implement the following. You could use a `switch()` statements or `if()` statements to conditionally execute the processing code.

- If GLUT_KEY_RIGHT then increment the global variable `ang_x` (i.e. `ang_x++`).
- If GLUT_KEY_LEFT then decrement the global variable `ang_x`.
- If GLUT_KEY_UP then increment the global variable `ang_y`
- If GLUT_KEY_DOWN then increment the global variable `ang_y`.
- Before returning from this function, refresh the display using this code:

```
//  Request display update
glutPostRedisplay();
```

The completed program for problem 1 should permit you to rotate the 3D room model in the viewing widow and control source and listener position. The source images should move in response to the source position and the reverberation (as direct plus images) you should be able to hear the reverberation changing.

Problem 2

In this problem you will complete the code in **para_reverb.cpp** and integrate the parametric reverberator into the callback in **room_acoustics.cpp**.

The file **para_reverb.h** is the paraReverb class header file. This is complete and does not need to be touched. It defines the class, the class variables and the class methods.

The file **parametric_reverb.cpp** is complete and does not need to be touched. It uses #define to define the coefficients in each of the blocks that make up the reverberator, declares the class objects that will implement the reverberator block and has the code, in function `prametric_reverb()`, that implements the full parametric reverberator.

The file **para_reverb.cpp** has complete code for

- The `paraReverb` class constructor. Note that the constructor stores initialization values into the class coefficient variables, allocates storage for the delay line, and finally initializes the value of the bottom, top, `first` and `last` pointers.
- The `paraReverb` class destructor. It must free allocated storage.

Your task is to complete the `filter()` class method (function) in **para_reverb.cpp**. This implements one section of the All Pass or Comb Filter reverberator. The complete parametric reverberator is implemented in parametric_reverb.cpp.

- The filter method implements one block, or section, of the parametric reverberator. Its input argument is the input to the block and it returns the output of the block.
- You will test the value of the class variable `mode` to determine if the filter should implement allpass (see Figure 3.4, above) or feedback comb filtering (see Figure 2.24, above), with the appropriate code for each. This could be via an if() or switch() statement.
- After the filtering, your code must "shift" the delay line. Since the delay line is implemented using a circular buffer, the shift amounts to decrementing the first and last pointers, but with wrap-around if the decremented value is below the bottom of the delay line storage.

When you run your complete program, it should behave exactly like the example executable discussed above!