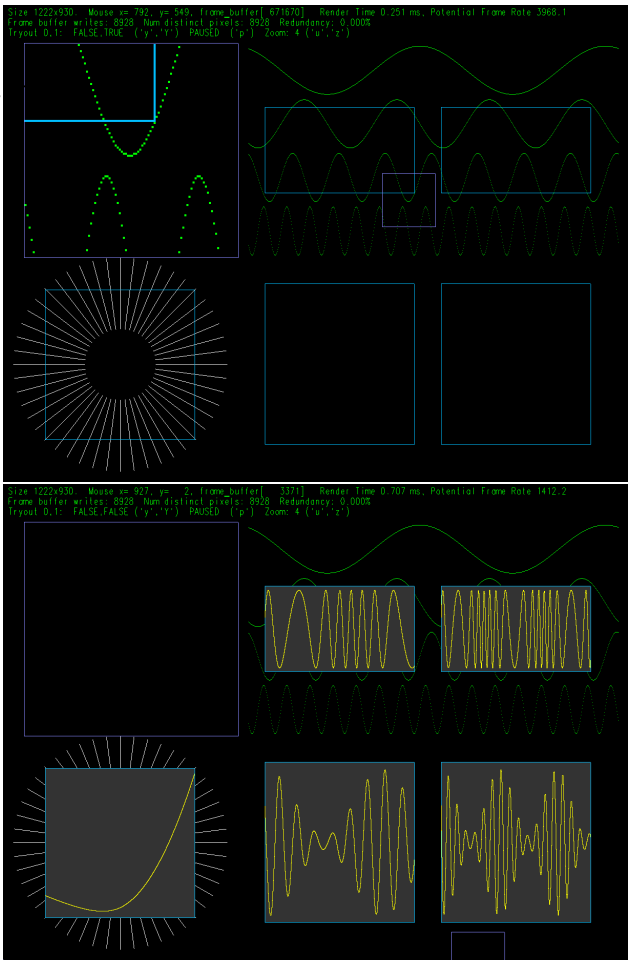
LSU EE 4702-1 **Homework 1** **Due: 9 September 2026**

Formatted 18:10, 1 September 2026

All of the code for this assignment is in the course repo. HTMLized versions of the assignment file are at `https://www.ece.lsu.edu/koppel/gpup/2026/hw01.cc.html`.



Problem 0: Follow the instructions on the `https://www.ece.lsu.edu/koppel/gpup/proc.` page for account setup and programming homework work flow. Compile and run the homework code unmodified. It should initially show some sine waves and some radially arranged white rectangles with an inner red circle and an outer green ring (which is just another name for a thick green circle). There should also be blue boxes. The box in the upper left of the screenshot shows a magnified version of the area around the mouse pointer. See the screenshot to the upper right. When the assignment is finished the blue boxes will show function plots, see the screenshot to the lower right. The top row of blue boxes show FM-modulated signals. The lower-left plot shows an *activation function* recently described in the AI literature, and plots to the right of that the lower right show AM-modulated signals.

Display of Performance-Related Data

The top green text line shows performance-related and other information. `Size` refers to the size of the window. `Mouse` refers to the coordinates of the mouse pointer. Coordinate $(0, 0)$ is at the **lower** left of the window. Text `frame_buffer[N]` shows the index of the frame buffer corresponding to the point under the mouse pointer. (In the assignment file `frame_buffer` is abbreviated to `fb`, for convenience.)

`Render Time` and `Potential Frame Rate` show the CPU time needed to write the frame buffer. They can be ignored for this assignment.

General User Interface

Press `Ctrl=` to increase the size of the green text and `Ctrl-` to decrease the size. Press `F12` to generate a screenshot. The screenshot will be written to file `hw01.png` or `hw01-debug.png`.

The value of two Boolean debug-support variables, `tryout0` and `tryout1`, are shown in the green text, pressing `y` toggles `tryout0` (between `true` and `false`) and pressing `Y` toggles `tryout1`. The variables are available in routine `render_hw01` and `plot_func`, use them to try things out.

Problem-Specific User Interface

The box in the upper left shows a zoomed version of the area around the mouse pointer. Pressing `z` will increase the zoom amount and pressing `u` will decrease it.

Some of the functions to be plotted change with time and so they appear animated. Press `p` to toggle this animation on and off.

Integer Coordinate and Vector Types

This assignment uses both integer and floating point types for coordinates and vectors. The integer types are `iCoor` for 2D coordinates and `iVect` for 2D vectors. Most other course code uses type `pCoor` for floating-point 3D coordinates (actually 4D, we'll learn in a few weeks) and `pVect` for floating-point 3D vectors.

The `iCoor` and `iVect` classes each have two members, `x` and `y`:

```
class iCoor {
public:
  iCoor( int x, int y ):x(x),y(y){}
  int x, y;
};
class iVect {
public:
  iVect( int x, int y ):x(x),y(y){}
  int x, y;
};
```

The addition, subtraction, and division operators are overloaded with these types. Here are some examples of how to use them:

```
void demo() {
  iCoor c0(3,4);  // Set to x=3, y=4.
  iCoor c1;
  c1.x = 5; c1.y = 7;  // Set to x=5, y=7;

  iCoor c3 = c1;

  // Component-wise subtraction.
  // Equivalent to v1.x=c0.x-c1.x;  v1.y=c0.y-c1.y;
  iVect v1 = c0 - c1;

  // Component-wise addition.
  // Equivalent to c4.x=c1.x+v1.x;  c4.y=c1.y+v1.y;
  iCoor c4 = c1 + v1;

  // Not allowed: Can't add two coordinates.
  iCoor c5 = c4 + c1;
}
```

Code Generation and Debug Support

The compiler generates an optimized version of the code, `hw01`, and a debug-able version of the code, `hw01-debug`. The `hw01-debug` version is compiled with optimization turned off, which makes it easier to debug. When needed, you are strongly encouraged to run `hw01-debug` under the GNU debugger, `gdb`. See the material under "Running and Debugging the Assignment" on the course procedures page. **You must learn how to debug.** If not, you will be at a severe disadvantage.

To help you debug your code and experiment in one way or another, the user interface lets you change variables. These are `hw01_data.tryout0` and `hw01_data.tryout1`. You can use these variables in your code (for example, `if ( hw01_data.tryout1 ) { x += 5; }`) to help debug, to help familiarize yourself with how the code works, or to experiment with new ideas. Keys `y` and`Y` to toggle their values, which are shown in the green text at the label `Tryout 0,1:`.

Resources

A good reference for C++ is `https://en.cppreference.com/w/`. See Homework 1 assigned in the past few semesters for similar problems.

Collaboration Rules

Each student is expected to complete his or her own assignment. It is okay to work with other students and to ask questions in order to get ideas on how to solve the problems or how to overcome some obstacle (be it a question of C++ syntax, interpreting error messages, how a part of the problem might be solved, etc.) It is also acceptable to seek out programming and graphics resources. It is okay to make use of AI LLM tools such as Claude and Copilot to generate sample code. (Do not assume LLM output is correct. Treat LLM output the same way one might treat legal advice given by a lawyer character in a movie: it may sound impressive, but it can range from sage advice to utter nonsense.)

After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

Student Expectations

To solve this assignment students are expected to avail themselves of references provided in class and on the Web site, such as for C++ programming and to seek out any additional help and resources that might be needed. (Of course this doesn't mean asking someone else to solve it for you.) Students are expected to experiment to learn how graphics work, and how to code C++ sequences. Experimentation might be done on past homework assignments. Students are also expected to learn what error messages mean by consulting documentation and by asking others (including Dr. Koppelman), and also to develop debugging skills. It is the students' responsibility to resolve frustrations and roadblocks quickly. (Just ask for help!)

This assignment cannot be solved by blindly pasting together code fragments found in class notes or past assignments. Solving the assignment is a multi-step learning processes that takes effort, but one that also provides the satisfaction of progress and of developing skills and understanding.

Problem 1: Routine `func_plot`, when completed, will plot a given function over a given range into a given rectangle in the frame buffer. The declaration is:

```
template< typename F > void func_plot
( pFrame_Buffer& fb,
  iCoor plot_lower_left, int plot_width, int plot_height,
  float func_x0, float func_x1, F func )
```

Argument `fb` is the frame buffer object used in many frame-buffer assignments. For examples on how it can be used see the code in `line_draw` and `render_hw01`.

Arguments `plot_lower_left`, `plot_width` and `plot_height` specify where in the frame buffer the plot is to be placed. Argument `plot_lower_left` specifies the frame buffer pixel coordinates of the lower-left of the plot to be drawn. (See the discussion of the `iCoor` object covered earlier in this assignment.) Arguments `plot_width` and `plot_height` are the width and height of the plot to be drawn. These variables are already used by code in `func_plot` to draw a box around the plot area.

Argument `func` is the function to be plotted, it can be any function that takes a `float` type as an argument and returns a `float` type. Arguments `func_x0` and `func_x1` specify the range of x values to be plotted.

The code below calls `func_plot` to plot `sinf` (sine to floating-point precision) from -2π to 7π. The plot will be drawn in a rectangle of width 400 pixels and height 200 pixels. (See `render_hw01` for other examples of how to call `func_plot`.) Note that for this assignment there is no need to call `func_plot`, that's already done.

```
iCoor plot_ll( 20, 20 );
float p_wd = 400,  p_ht = 200;
float px0 = -2 * numbers::pi,  px1 = numbers::pi * 7;
func_plot( fb,   plot_ll, p_wd, p_ht,   px0, px1,   sinf );
```

In the unmodified code `func_plot` draws a deep-sky-blue box around the plot rectangle:

```
const uint32_t color_deep_sky_blue(0x00bfff);
iCoor p0 = plot_lower_left; // Just an abbreviation.
iVect vx( plot_width, 0 ), vy( 0, plot_height );
iCoor p1 = p0 + vx + vy;
line_draw( fb, p0, p0 + vy, color_deep_sky_blue );
line_draw( fb, p0, p0 + vx, color_deep_sky_blue );
line_draw( fb, p0 + vy, p1, color_deep_sky_blue );
line_draw( fb, p0 + vx, p1, color_deep_sky_blue );
```

It also computes the first and the last values of `func` to be plotted, but does not plot them.

```
float func_y0 [[maybe_unused]] = func(func_x0);
float func_y1 [[maybe_unused]] = func(func_x1);
```

(*a*) Modify `func_plot` so that it sets the plot area to `color_background`, where `color_background` is a variable in `func_plot`. Do this after the lines are drawn. *Note: this is easy.*

(*b*) Modify `func_plot` so that the function is plotted as described below.

Let *plot area* refer to the rectangle specified by `plot_lower_left`, `plot_width`, and `plot_height`. In the discussion below x_0 and x_1 refer to `func_x0` and `func_x1`, w and h refer to `plot_width` and `plot_height`, i_0 and j_0 refer to `plot_lower_left.x` and `plot_lower_left.y`; and $i_1 = i_0 + w$ and $j_1 = j_0 + h$. Let $y = f(x)$ be the value returned by `func` for input x.

To solve this two mappings are needed, one from pixel space coordinates to function space, and one from function space coordinates to pixel space. Pixel space coordinates are what are used to write the frame buffer. Values i_0, i_1, j_0, j_1 are pixel space coordinates. Values x_0, x_1, $f(x_0)$, and $f(x_1)$ are function space coordinates. The function space x coordinate is used for the argument to $f(x)$ and the function space y coordinate is used for the return value of $y = f(x)$. The mappings should be linear, and so they can be achieve by maybe multiplying and adding something to, say, a pixel space x coordinate to get a function space x coordinate.

The code needs to iterate from i_0 and i_1, and for each $i \in [i_0, i_1]$ the code needs to use the mapping to convert i, which is in pixel space, to a function-space x value. Using that compute $y = f(x)$ and then convert y to pixel space, call the value j, and write the frame buffer at pixel coordinate i, j (and at other locations to avoid gaps).

- [ ] Plot the function from x_0 to x_1, with the value for x_0 placed along the left edge (at pixel x coordinate i_0) and the value for x_1 placed along the right edge.

- [ ] Scale the values returned by `func` so that they fit within the plot area and so that the largest $y = f(x)$ is placed $.025h$ pixels from the top of the plot area and the smallest $y = f(x)$ is $.025h$ pixels from the bottom of the plot area. [ ] Make sure the top and bottom of the plots don't touch the top and bottom of the plot box.

- [ ] The plot should be continuous (without gaps) so for each i (pixel x coordinate) it may be necessary to plot multiple points. In the screenshot on the lower left the plots are not continuous (they are dotted). Make sure the plots are continuous as in the screenshot on the lower right.

- [ ] The code should be reasonable efficient. Don't call the line drawing routine to fill the gaps between points because the overhead of setting up a line draw can be avoided here.