AAP Lab 3

In this lab you will build upon what you learnt last week by extending the gain control plugin to a stereo panner with a slider to control pan position. You will also be introduced to Github for version control of your plugins.

The lab assumes you are working on the machines in M329. If you are using your own computer, please read through the Software Guide on GCU Learn to ensure your computer matches the setup in M329.

Learning objectives

By the end of this lab you will have learnt how to:

- Build a stereo panning plugin that implements a linear panning algorithm
- Setup the JUCE GUI editor to add interface controls
- Link the interface controls to the *processor* class
- Undertake basic version control during plugin development

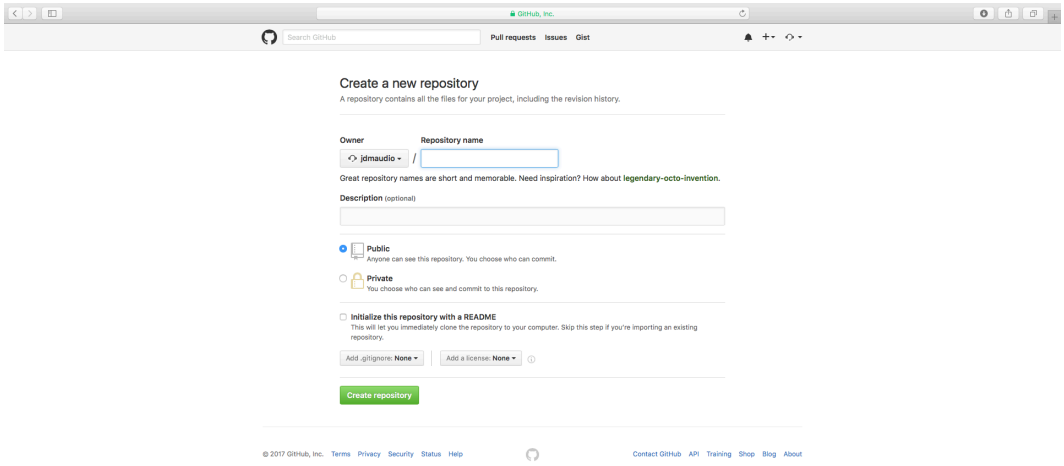
Github

When developing software it is useful to manage changes to source code over time. Github is a distributed version control system that is very popular in the software development industry right now. You will be using Github in this module to manage versioning of your plugin source code (see lecture 3 slides for more detail).

Create and account on Github.com and sign in with your credentials. You can take the tour to learn about Github workflow if you want.

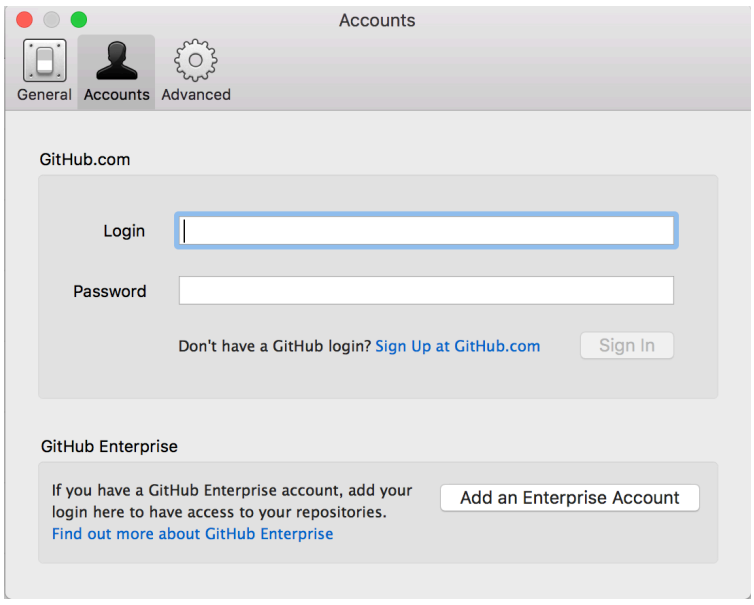
Next, click 'Start a project' to be taken to the repository creation page. A repository is used to organise a single project. Repositories can contain folders and files, images, videos, spreadsheets, and data sets – anything your project needs.

Name the project StereoPanner. The project can be set to 'Public' (private repositories are for paid github.com accounts only). Ignore the options for adding a Readme file and .gitignore and licensing for now. Click on 'Create Respository' to create the online Git repository.



Github Desktop

Open Github Desktop from the Applications directory on the M329 Macs. Go to Github Desktop->Preferences->Accounts and login using your github.com credentials.



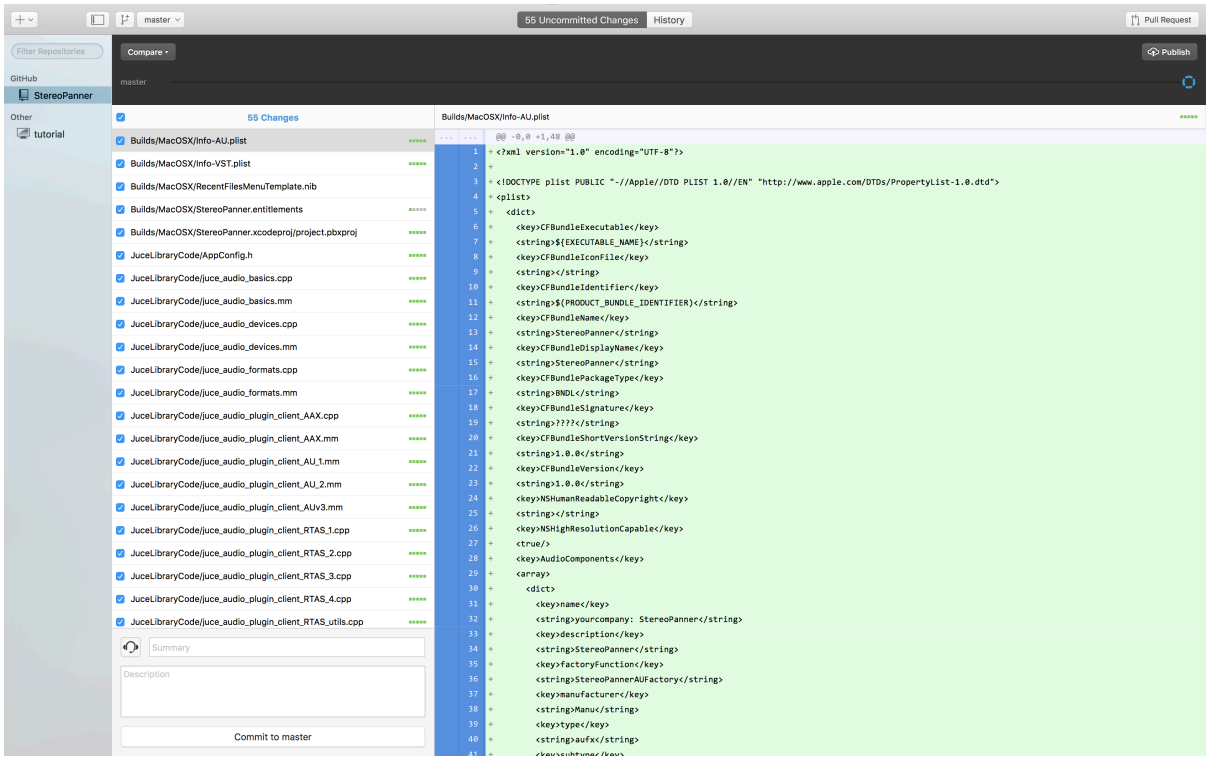
Next, select File->Clone Repository and select the StereoPanner repository you created on github.com. You will be asked for path to store the respository so just select the Desktop for now.

Juce Project configuration

Next, setup a new JUCE Audio Plugin project using the Projucer (see lab last week if you are unsure of this). Name your project StereoPanner and ensure your project is created on the Desktop i.e. the same location and name as your Github repository.

Click 'Create' then ensure only 'Build VST' is selected for the plugin format and the deployment target is set to OS X 10.11 for debug and release modes to ensure compatibility with the operating system used in M329.

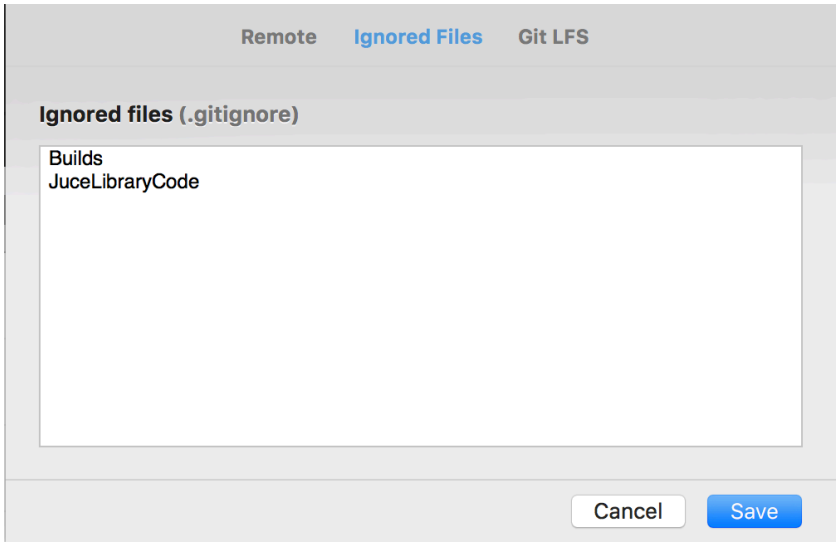
If you go back to Github Desktop now you should see that it has picked up on a large number of file changes e.g.



These changes are basically all of the new files that have been generated by Projucer in the StereoPanner project folder the Github repository is associated with. However, not all these files need to be tracked in our repository. Really the only files

that need to be tracked are the 4 x plugin source code files that contain the *processor* and *editor* classes. Therefore, it is useful to setup a .gitignore file to tell Github that all other files should not be tracked. This is straightforward in Github Desktop.

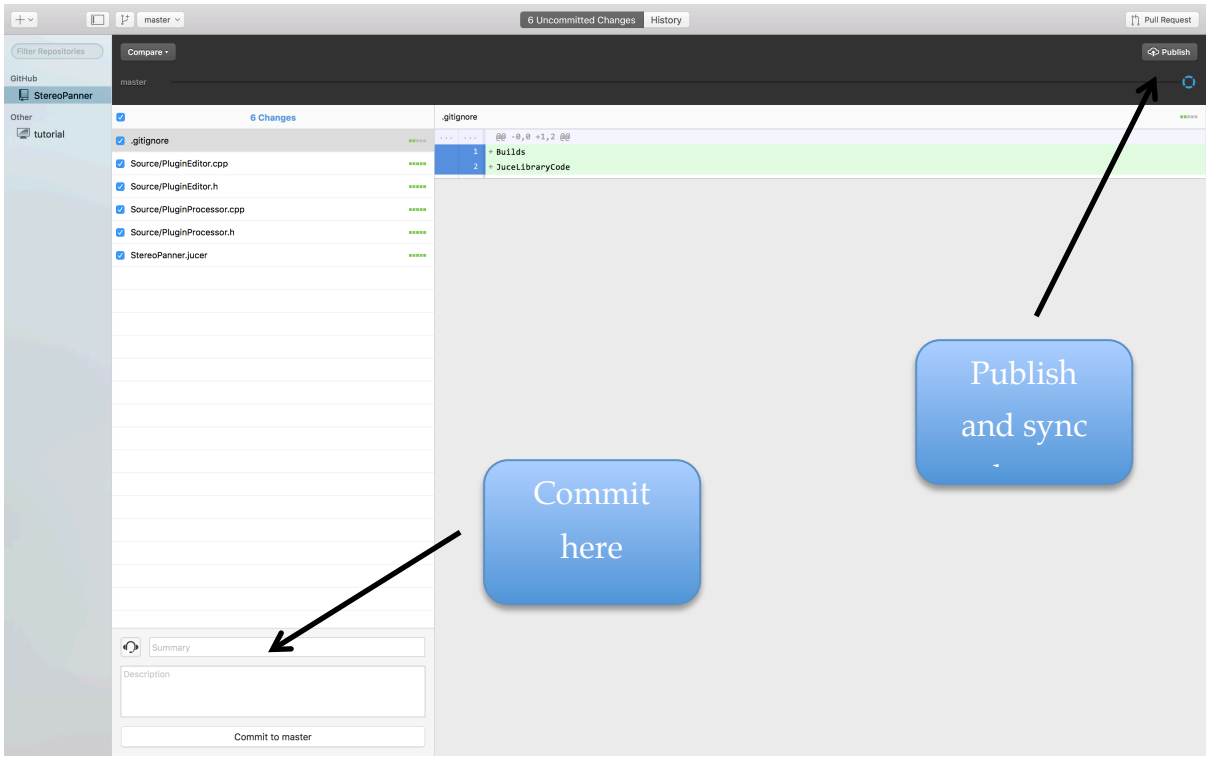
Select Repository->Repository Settings->Ignored Files (this creates a .gitignore file) Type in 'Builds' and then 'JuceLibraryCode' so ensure that only the 4 plugin source code files and the .jucer project file are ever kept in the repository. Click 'Save'.



You should see the following in your Github Desktop interface.

Next, you should commit these 6 files to the repository for the first time. A Commit allows you to keep track of your progress as you work on a branch or master. The first commit is traditionally named 'Initial Commit'. Type this in to the summary text box then click on 'Commit to Master'. Next, click on Publish to sync this with your online repository.

For any future changes you make to your code it is a good idea to create a commit and then push them to Github.com using the sync button.



Finally, go back to Projucer and open your project in Xcode by clicking on 'Save and Open in IDE…".

At the end of each of the sections following add a commit to your repository and ensure you sync up with github.com

Setting up a Plugin Graphical User Interface using the Projucer

There are two main options when building a GUI in JUCE. One way involves adding all the code you need yourself for each GUI component in the *Editor* class. The other way is to use the GUI Editor built in to the Projucer. The latter is the preferred approach in this module as it allows you to visually place your controls within the Plugin interface.

The GUI Editor is not enabled for working with JUCE audio plugins by default. The default *Editor* class is not compatible with the Projucer GUI editing functions. However, it is fairly straightforward to change this.

Select the *"Files" Tab* (this will show the project files with code previews in a basic editor). Right-click the StereoPanner Project icon and select "Add New GUI Component".

Browse to your project's source directory, *select "PluginEditor.cpp", Click "Save"* to overwrite the file (it will actually replace both the .h and .cpp files for the *PluginEditor*). Now, when you *click PluginEditor.cpp in the source file browser* of the Projucer, it will display five tabs: Class, Subcomponents, Graphics, Resources and Code. If you don't see these tabs it is likely you chose the wrong file to overwrite.

For now, select the *"Class" Tab* to make the edits that are needed to tie the new GUI Component to your *Processor* class (the core code for your VST plugin)...

Change the class name field to match the *editor* class originally generated by Projucer i.e.
StereoPannerAudioProcessorEditor

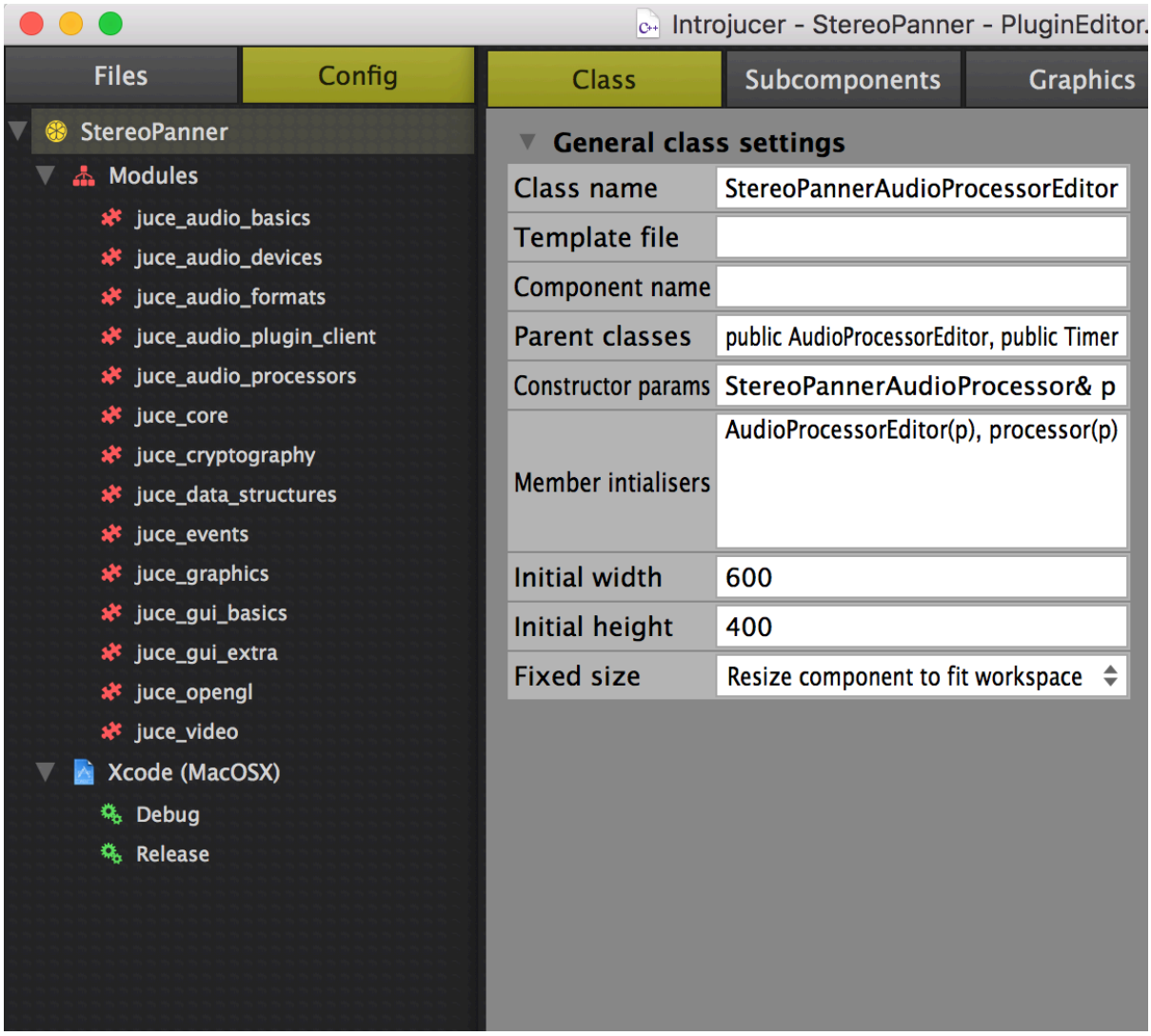
Change the parent class field to:
public AudioProcessorEditor, public Timer

Add constructor parameters:
StereoPannerAudioProcessor& p

Member initialisers:
AudioProcessorEditor(p), processor(p)

After making these changes, the settings in the class tab should match those in the screenshot below.

Note: you can change the width and height dimensions of your plugin interface from the Class tab (default is 600 x 400 pixels)



Save the Editor file (⌘s). Make sure that you do this anytime you make edits in the Projucer. Click on the Config tab in the Projucer. Then click on "Save Project and Open in IDE…" to open the project up in Xcode.

The next steps will require you to open up and add/edit the JUCE automatically generated code for your project in the four main source code files for the plugin. Open up PluginEditor.h from the Project files navigator in Xcode. Locate the following code:

```
//[Headers]     -- You can add your own extra header files here --
#include "JuceHeader.h"
//[/Headers]
```

Add an additional include statement as follows:

```
//[Headers]     -- You can add your own extra header files here --
#include "JuceHeader.h"
#include "PluginProcessor.h"
//[/Headers]
```

This statement alerts the compiler where to look for the *Processor* class definition allowing you to reference the *processor* from your *editor* class.

Important: when adding code to the JUCE files, it is recommended you add the code between the commented sections... for example all header include changes should be between "//[Headers]" and "//[\Headers]". Otherwise, you risk having changes overwritten later when you update any settings in the Projucer.

Next add the following code to the User Variables section of the template code in PluginEditor.h

```
//[UserVariables]   -- You can add your own custom variables in this section.
StereoPannerAudioProcessor& processor;
//[/UserVariables]
```

This is reference variable of the same type as your *processor* class. Through this variable you will be able to access the *processor* class members. An instance of the processor class is assigned in the constructor of the *Editor*.

Next add the following timeCallback() definition to the UserMethods section of PluginEditor.h

```
//[UserMethods]     -- You can add your own custom methods in this section.
void timerCallback();
//[/UserMethods]
```

This function is called repeatedly and is where you inform the *Editor* controls of any updates to plugin parameters in the *processor*.

Open up PluginEditor.cpp and add the following startTimer code in the Constructor section of this file:

```
//[Constructor] You can add your own custom stuff here..
startTimer(200);//starts timer with interval of 200mS
//[/Constructor]
```

This code starts the timer and causes the timerCallback() function to be called at regular intervals (200ms). This is the rate your GUI checks for internal plug-in changes.

Finally, add the implementation of the timeCallback() function to the MiscUserCode section:

```
//[MiscUserCode] You can add your own definitions of your custom methods or any
other code here...
void StereoPannerAudioProcessorEditor::timerCallback()
{
    //exchange any data you want between UI elements and the Plugin "ourProcessor"
}
//[/MiscUserCode]
```

This is the implementation of the timerCallback defined above. More on the importance of this next week. For this plugin project you can leave it empty as long as it is defined the code will still compile.

Product->Build For->Profiling (shortcut: ⇧⌘I) to compile your plugin. If you have followed the steps above precisely being careful to copy the code provided, your plugin will compile with no errors.

Finally, go back to Github Desktop and add a commit. You could name it 'Setup Editor' then sync your project.

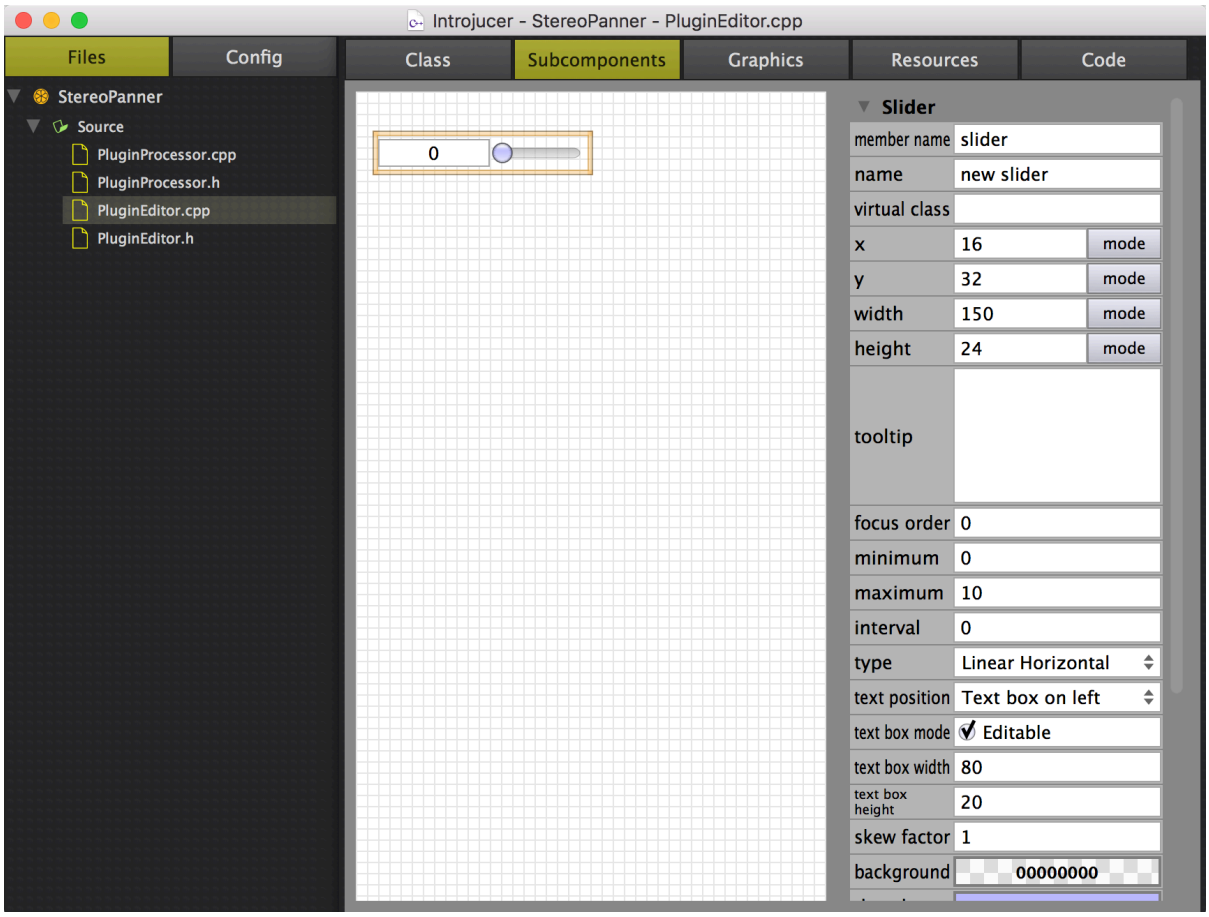
Note: The default Projucer *plugin is configured for stereo input and output. However, it is possible to change any multichannel input and/or output – more on this next week.*

Setting up a Plugin GUI in the Projucer

A link between the *Editor* and *Processor* classes has now been made and you can start to add user interface controls using the GUI editor in the Projucer.

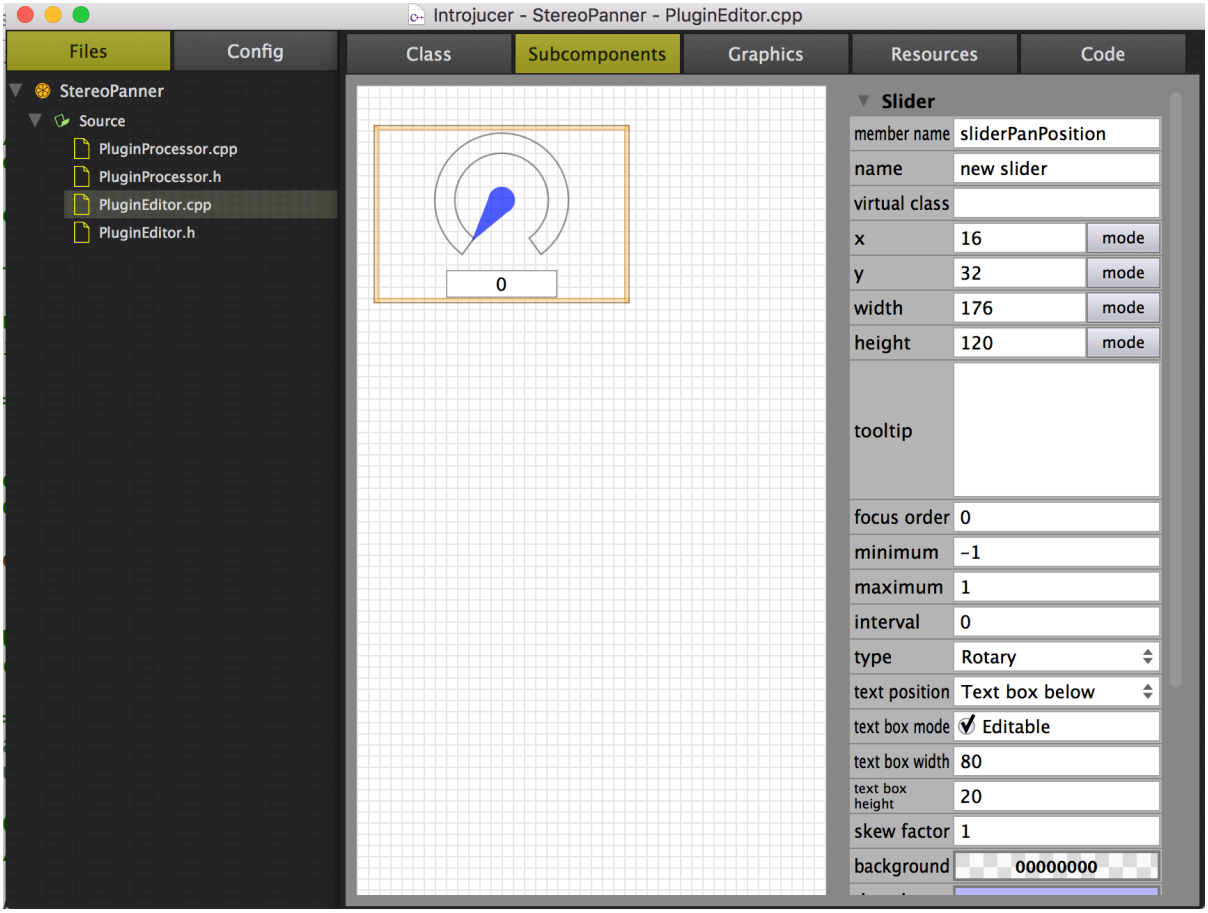
Open the Files tab in the Projucer and click on PluginEditor.cpp. Then click on the "Subcomponents" tab along the top. The grid area that opens up is where you place plugin controls e.g. button, slider etc.

The plugin we are building in this lab will just have a single slider control for the pan position. Right click in the grid area and select "New Slider" to add a slider to the interface. Click and drag it to a suitable position on the plugin interface e.g.



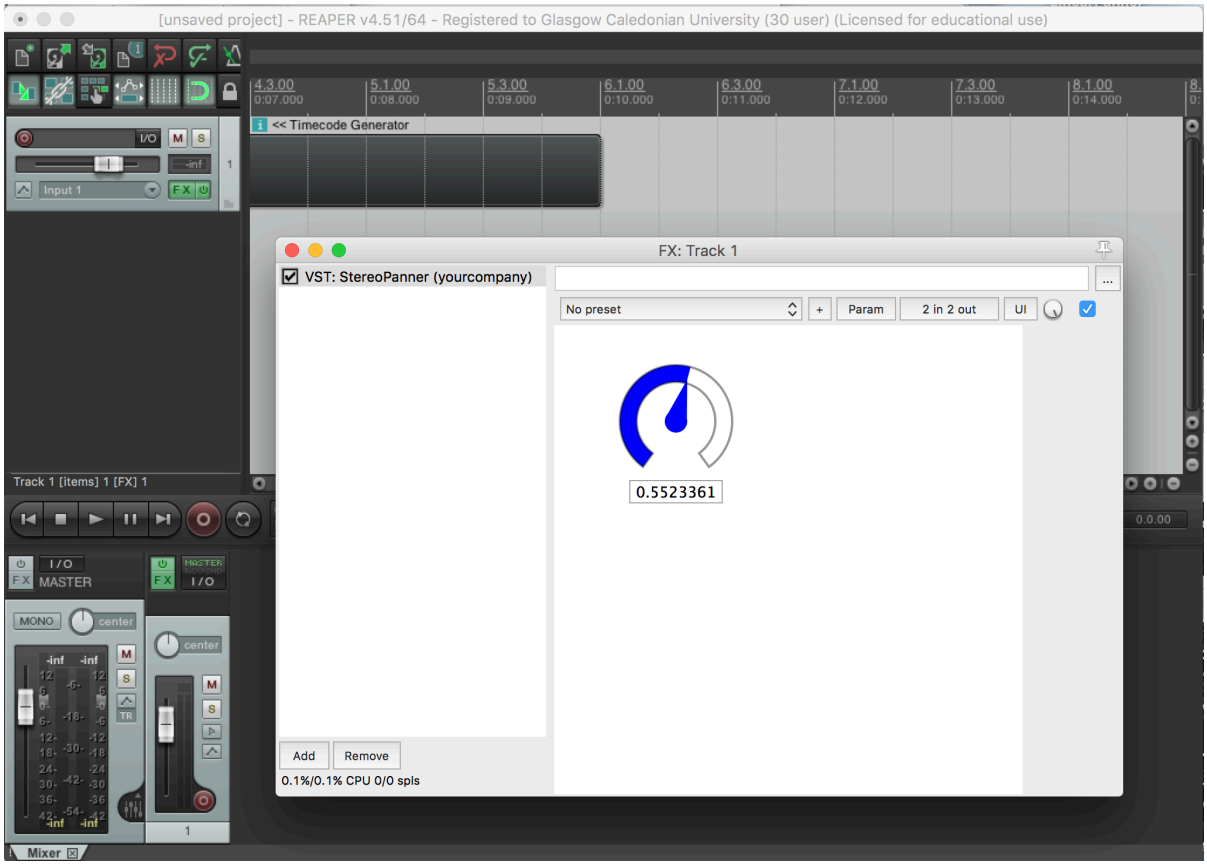
If you click to select your slider you will notice it has a number of properties in the right window which are changeable. Change the first property "member name" to sliderPanPosition. This is the name of the slider object added to the *Editor* class for this interface.

Next click on the dropdown box for 'type' and select 'Rotary' to change the slider to a rotary control. Then change the 'text position' property to 'text box below' to place the numeric readout of the slider position below the rotary control. Set the slider range using the minimum and maximum fields (-1 min, 1 max). Finally, resize the slider to make it bigger by clicking and dragging from the bottom right corner i.e.



Resave PluginEditor.cpp (⌘S) and return to Xcode. If you look through the *Editor* code in .h and .cpp, you will notice that Projucer has automatically added in code relating to the properties of the slider e.g. sliderPanPosition->setRange (-1, 1, 0);

Select Product->Build For->Profiling (shortcut: ⇧⌘I) to compile your plugin. If you open Reaper and add your plugin to an audio track, this is what it should look like:



At this stage the rotary controller on your stereo panner should function and display the correct range of values. However, the values from this control are not passed to the processor class or tied to any audio processing code.

Open PluginProcessor.h then add a float member variable named panPosition after the public member functions (highlighted in the code below). This member variable in the *processor* class is going to store the pan value from the slider control.

```
class StereoPannerAudioProcessor  : public AudioProcessor
{
public:
    //==============================================================================
    StereoPannerAudioProcessor();
    ~StereoPannerAudioProcessor();

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;

    void processBlock (AudioSampleBuffer&, MidiBuffer&) override;

    //==============================================================================
    AudioProcessorEditor* createEditor() override;
    bool hasEditor() const override;

    //==============================================================================
    const String getName() const override;

    bool acceptsMidi() const override;
    bool producesMidi() const override;
    bool silenceInProducesSilenceOut() const override;
    double getTailLengthSeconds() const override;

    //==============================================================================
    int getNumPrograms() override;
    int getCurrentProgram() override;
    void setCurrentProgram (int index) override;
    const String getProgramName (int index) override;
    void changeProgramName (int index, const String& newName) override;

    //==============================================================================
    void getStateInformation (MemoryBlock& destData) override;
    void setStateInformation (const void* data, int sizeInBytes) override;

    float panPosition;

private:
    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (StereoPannerAudioProcessor)
};
```

Next, open PluginEditor.cpp and add the following code to the 'if-statement' of the sliderValueChanged() function.

```
//[UserSliderCode_sliderPanPosition] -- add your slider handling code here..
processor.panPosition = sliderPanPosition->getValue();
//[/UserSliderCode_sliderPanPosition]
```

Now every time the slider control position is changed, this function is called and the value of sliderPanPosition (returned by getValue()) is assigned to the panPosition member variable of the processor class.

Finally, go back to Github Desktop and add a commit. You could name it 'Added a Slider' then sync your project.

Panning algorithm code

Now the interface has been configured with a control for pan position, and there is a link between the control in the *Editor*, and a variable in the *Processor* classes, the next step is to add the panning algorithm code to the *processor* audio callback function.

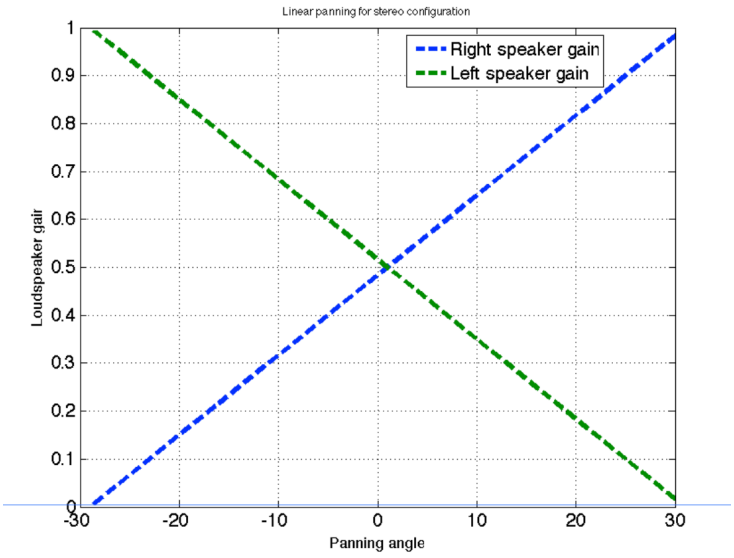
The panning algorithm you will implement is for linear gain panning between the left and right channels. This is probably the simplest method for amplitude panning. It just creates a linear crossfade between the left and right channels (shown in the figure below). The equations for linear panning are:

$g_L = 1 - p'$
$g_R = p'$

$p' = (p + 1) \ / \ 2$

g_L is the left speaker gain
g_R is the right speaker gain

p represents the pan position, a left pan position is -1, a right pan position is 1, centre is 0.



Open PluginProcessor.cpp and scroll down to the *processBlock()* function. Remember from last week that this is the audio callback function so is repeatedly called by the

DAW every time it has some audio to process. It is where you add your code to process the audio that is passed in through the AudioSampleBuffer parameter for this function. Copy and replace the existing processBlock function with this one:

```
void StereoPannerAudioProcessor::processBlock (AudioSampleBuffer& buffer, MidiBuffer&
midiMessages)
{
    // In case we have more outputs than inputs, this code clears any output
    // channels that didn't contain input data, (because these aren't
    // guaranteed to be empty - they may contain garbage).
    // I've added this to avoid people getting screaming feedback
    // when they first compile the plugin, but obviously you don't need to
    // this code if your algorithm already fills all the output channels.
    for (int i = getNumInputChannels(); i < getNumOutputChannels(); ++i)
        buffer.clear (i, 0, buffer.getNumSamples());

    // Retrieve the total number of samples in the buffer for this block
    int numSamples = buffer.getNumSamples();

    // channelDataL and channelDataR are pointers to arrays of length numSamples which
    // contain the audio for one channel.  You repeat this for each channel
    float *channelDataL = buffer.getWritePointer(0);
    float *channelDataR = buffer.getWritePointer(1);

    // calculate p'
    float pDash = (panPosition + 1.0) / 2.0;

    // Loop runs from 0 to number of samples in the block
    for (int i = 0; i < numSamples; ++i)
    {
        // Simple linear panning algorithm where:
        channelDataL[i] = channelDataL[i] * (1.0 - pDash);
        channelDataR[i] = channelDataR[i] * pDash;
    }

}
```

Select Product->Build For->Profiling (shortcut: ⇧⌘I) to compile your plugin. Re-open Reaper and reload the plugin and test out the panner with some audio on a track. A Reaper session file with a pink noise generator has been added to GCU Learn for you to use.

Finally, go back to Github Desktop and add a commit. You could name it 'Added panning algorithm' then sync your project.

Note: Linear panning problem: amplitude appears to be fainter as a source is positioned in the centre resulting in a "hole in the middle" of the stereo image.

Additional exercise

Can you extend this plugin to implement a constant power panning algorithm? Constant power amplitude panning maintains a constant level of sound as a sound source is panned from left to right. This is the most commonly used algorithm in mixing desks and DAW mixers. The equations for constant power panning are:

$g_L = \cos(p')$
$g_R = \sin(p')$

$p' = \pi \times (p + 1) / 4$

g_L is the left speaker gain
g_R is the right speaker gain

p represents the pan position, a left pan position is -1, a right pan position is 1, centre is 0

You will need to use the c++ math functions cos and sin in your code for this algorithm as above. You can hard code the value of pi as a float (3.14159). A solution to this exercise will be provided on GCU Learn.