UF UNIVERSITY of FLORIDA
Herbert Wertheim
College of Engineering

CIS4XXX

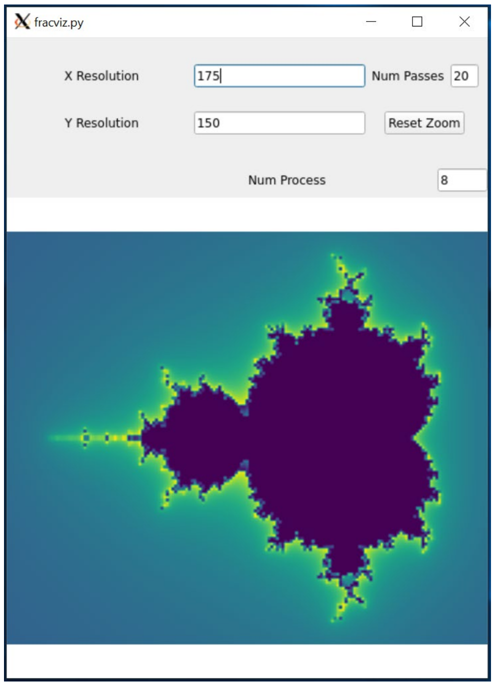
Exercise: Fractal GUI

Overview

In this exercise, students will practice working with the **Qt** GUI system, threading, and multiprocessing by building an application to display a visualization of a fractal set. Note that you will need to have an X-Server (e.g., VcXsrv) running to develop and test this exercise.

Requirements

Each student will develop an application using Qt. The application will provide an interactive interface used to display and navigate the visualization of the Mandelbrot Set. This will require multiprocessing in order to complete tasks allocated by the fractal class in order to calculate the visualization. The driver program and all classes will be in the Python routines will be in the `fracviz.py` file.



X Resolution: 175
Num Passes: 20
Y Resolution: 150
Reset Zoom
Num Process: 8

Visualization Application

The application should perform as follows.

General Structure

1) Run `wsl` module's **`set_display_to_host()`** method
2) Instantiate a Qt application and classes as outlined here
3) Load the `fracviz.ui` GUI file and display the GUI
4) Instantiate the fractal using parameters in UI file
5) Display the fractal as outlined in this document
6) Only run the application if invoked directly
7) Have ***no global variables***

Application Behavior

1) Accept only numeric input for text box entry
2) Reset visual and parameters to default when "Reset" is clicked
3) Zoom into fractal when mouse is dragged on the canvas widget
4) Set status to **`"Calculating set..."`** while generating image
5) Set status to **`""`** when image generation is complete

NOTE: The visual should change only when edited is completed. It should not change on every textbox keystroke entry; instead, it should change when a textbox loses focus or when "Enter" is pressed.

Key Libraries

It is worth reviewing the key libraries and classes that are used within the toolchains used for this exercise:

Qt (PySide2)

QObject:	Objects interacting with Qt must derive from `QObject` (e.g., driver classes, event handlers).
QWidget:	Forms and windows are made of a tree of `QWidget` objects (windows, forms, buttons, layouts, etc).

Matplotlib

FigureCanvas:	A widget that is associated with a `Figure` and is used to display it (e.g., a Qt Widget).
Figure:	Single visual figure representation, made up of one or more plotted `Axes` sets (graphs).
Axes:	Axes set in a visual area of a `Figure` onto which a graph or `AxesImage` can be plotted.
AxesImage:	Object holding an image drawn to an `Axes` set.

FractalWindow Class

The `FractalWindow` class should derive from the `QWidget` class. It is responsible for setting up and connecting elements of the GUI. It must have the following methods and properties / attributes.

Required Methods

`__init__``(self, filename, app)`

This is the constructor for the `FractalWindow` class. The `filename` should be used to load the UI from the file. This method is also responsible for preparing all elements of the GUI. This includes…

- Loading the UI file and generating a tree of widgets from it, with `self` as the parent of the root
- Instantiating a `FigureCanvas` object, creating a `Figure`, and generating an `Axes` set
- Adding the `FigureCanvas` object as a child of the `layout` instance (see properties)
- Linking widget signals to slots (methods / functions)
- Setting up event filters as necessary
- Any other actions required to facilitate proper application behavior

The `FigureCanvas` object <u>should not display axis information</u>; in addition, the `Axes` set object should be positioned and sized to fill the entire `Figure`. The `FigureCanvas` should otherwise use the object parameters. Finally, the `FigureCanvas` object should detect the mouse click-and-drag operations to facilitate the zoom functionality.

The `app` parameters should hold a reference to the `FractalApp` instance which called the constructor.

Required Properties (or Public Attributes)

The following may be implemented at public attributes or as read-write properties.

`axes`
`Axes` set object for the `FigureCanvas` object.

`canvas`
`FigureCanvas` object.

`figure`
`Figure` object for the `FigureCanvas` object.

`iterations`
`QLineEdit` widget from UI file with name **`"iterations"`**.

`layout`
`QVBoxLayout` widget from UI file with name **`"layout"`**.

`processes`
`QLineEdit` widget from UI file with name **`"processes"`**.

`resolution_x`
`QLineEdit` widget from UI file with name **`"resolution_x"`**.

`resolution_y`
`QLineEdit` widget from UI file with name **`"resolution_y"`**.

`reset_button`
`QPushButton` widget from UI file with name **`"reset_button"`**.

`status`
`QLabel` widget from UI file with name **`"status"`**.

FractalApp Class

The `FractalApp` class should derive from the `QObject` class. It is responsible for storing and updating the fractal set as well as its image representation (`AxesImage`).

It is recommended that students use this class to hold the slots for standard widget signals. It must have the following methods and properties.

Required Methods

`__init__``(self, filename)`
This is the constructor for the `FractalApp` class. The filename should be used to instantiate a `FractalWindow` widget as defined earlier in this document. In addition to instantiating the `FractalWindow` widget, this method should instantiate an instance of the `Mandelbrot` set class using the default values from the GUI, updating the plotting image, and displaying it in accordance with the rest of this specification.

`update_plot``(self)`
This method should kickoff the update of the `AxesImage` instance. It is responsible for ensuring that the display is updated **without locking up the application** or causing it to exhibit "lag". This should be done by:

1) Create and start a non-GUI thread for image processing so that the GUI does not "freeze"
2) Non-GUI thread should generate the fractal tasks using the process count (see `Mandelbrot` specification)
3) Non-GUI thread should create the appropriate number of processes to execute tasks.
4) The processes should start but take care that no task runs more than once!
5) Non-GUI thread should wait for all tasks to be completed.
6) Non-GUI thread should get updated image data and update the `AxesImage` object.
7) Non-GUI thread should signal the GUI thread to update the figure's plot, then terminate.
8) GUI thread, upon receipt of signal, should update the visualization.

Required Properties (Read-Only)

`fractal`
`Mandelbrot` set object currently in use by the application.

`image`
`AxesImage` object holding the visualization and linked to the `FigureCanvas` object's display.

`root_widget`
`FractalWindow` widget that is the root of the GUI display.

Fractal Module

The `fractal` module includes two classes – a base `Fractal` class and a derived `Mandelbrot` class.

Fractal Class

The abstract `Fractal` class defines a standard interface for multiprocessing and display.

Methods

`__init__``(self, image_width, image_height, iterations)`

The constructor sets up the variables, but does not complete image processing, for fractal image generation, given image dimensions and a number of iterations on the set.

`generate_tasks``(self, shared_mem_manager, num_tasks)`

This method generates the tasks necessary to process the fractal image. It uses the `shared_mem_manager` to allocated shared memory blocks that can be accessed across processes (for multiprocessing). It returns a list of tasks (lambdas) and a list of shared memory blocks allocated by the object as **`(tasks, data)`**.

Properties (Read-Write)

`x_range (tuple)`

Get or set the range of x-axis values to display from the set graph (min, max).

`y_range (tuple)`

Get or set the range of y-axis values to display from the set graph (min, max).

`dimensions (tuple)`

Get or set the resolution (in pixels) of the image to be generated (x, y).

`iterations (int)`

Get or set the number of iterations to be completed on the fractal set to generate the image (scalar).

Mandelbrot Class

The `Mandelbrot` is derived from the `Fractal` class and its methods are specific to the Mandelbrot set.

Methods

`data_to_image_matrix``(self, data)`

Upon completion of processing tasks, this method can be called to generate the image matrix from the `data`. This `data` should be the same list returned by the **`generate_tasks`** method, e.g.:

```
tasks, data = fractal.generate_tasks(smm, num_tasks)
for task in tasks: task() # Don't do this; it's single-threaded and just an example.
image_matrix = fractal.data_to_image_matrix(data)
```

Submissions

NOTE: Your output must match the example output *exactly*. If it does not, ***you will not receive full credit for your submission***! Please submit only and exactly these files:

Files:	fracviz.py
Method:	Submit on Canvas