of the half-life? If so, you can claim to have verified the code for this problem.

Decay series

For heavy nuclei, a daughter nucleus resulting from radioactive decay, may itself be unstable and undergo decay. The process may continue through several "generations" until the decay results in a stable nucleus. This is called a decay series or decay chain.

There are four radioactive decay series found in nature: the decay of Uranium-235, Uranium-238, Thorium-232 and Neptunium-237. ← NATURAL DECAY SERIE

Modify your code to simulate a radioactive decay series which decays through two or more generations. Each element in the series should have a different half-life. You can either choose your own values for a hypothetical series, or select values from a section of one of the natural decay series; for example, the half-lives for the last four decays in the Uranium-235 series are:

Lead-211 $\xrightarrow{T_{1/2}=36\,\text{mins}}$ Bismuth-211 $\xrightarrow{T_{1/2}=2.1\,\text{mins}}$ Thallium-207 $\xrightarrow{T_{1/2}=4.8\,\text{mins}}$ Lead-207 (stable)

2.4 Relevant course sections

Studying the following course sections will help you complete this checkpoint:

- Classes, Objects and Methods
- Object Oriented Design

Additional material that you may find useful:

- Errors and Exceptions

2.5 Marking Scheme

- Radioactive Decay Checkpoint Marking Scheme

3 Mandelbrot Set

> *"Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightening travel in a straight line."*
>
> Benoit Mandelbrot

3.1 Aim

A fractal is a geometric shape that contains self-similar images within itself - you can zoom in on a section and it will have just as much detail as the whole fractal. Many objects in the natural world exhibit fractal behaviour. For example, the human circulatory system is a fractal. If you look at the blood vessels in your hand, they resemble the overall shape that the complete system takes on. The Mandelbrot set is also an example of a fractal - it is recursively defined and infinitely detailed.

The Mandelbrot Set is a set of complex numbers C resulting from repeated iterations of the following function:

$$z_{n+1} = z_n^2 + C$$

with the initial condition $z_0 = 0$.

A given complex number C belongs to the Mandelbrot set if $|z_n|$, the magnitude of z_n, remains bounded, i.e. does not diverge. If $|z_n|$ diverges then C does not belong to the Mandelbrot set.

In fact, it can be shown that if $|z_n| > 2$ for some value of n, it will subsequently radidly tend to infinity, i.e. diverge, meaning that C is not in the Mandelbrot set.

You can also assume that if $|z_n|$ has not diverged after 255 iterations, it will not diverge at larger values of n, meaning that C is in the Mandelbrot set.

3.2 Checkpoint task

Write a PYTHON program that will give a visual representation of the Mandelbrot set.

To obtain a visual plot of the Mandelbrot set, the complex plane can be represented as a 2D grid and the value of N (the number of iterations needed to reach the threshold $|z_n| > 2$) calculated for complex numbers C corresponding to points on the grid. As explained above, you should set an upper iteration limit of $N = 255$.

The value of N can then be converted to a colour and plotted on the grid.

You should explore values of C in the range x={-2.025 $\rightarrow$ 0.6} and y={-1.125 $\rightarrow$ 1.125}. Note that as you increase the number of points on the grid you will not only increase the resolution of the display but will also significantly increase the computation time.

3.3 Optional extra

Write a program to display a Julia set. Julia Sets are produced from the same formula as the Mandelbrot set but used in a different way. When making a picture of a Julia set, C remains fixed during the whole generation process, while the value of Z_0 varies. The value of C determines the shape of the Julia set: in other words, each point of the complex plane is associated with a particular Julia set. ← NOTE

Some of the more famous Julia sets are:

- C = -1,0
- C = 0,-1 – the Dendrite
- C = 0.5,0
- C = -0.1 0.8 – the Rabbit
- C = 0.36,0.1 – the Dragon

3.4 Relevant course sections

Studying the following course sections will help you complete this checkpoint:

- NumPy arrays
- Plotting using Matplotlib

3.5 Marking Scheme

- Mandelbrot Set Checkpoint Marking Scheme

4 Traffic

4.1 Aim

This checkpoint requires you to code a simple cellular automaton which attempts to model traffic flow. This is an example where the effective theory is not known, so we invent a theory that has the basic properties that we believe are important. By running simulations we can look for emergent phenomena, in this case the onset of congestion (traffic jams).

4.2 The Model

The simulation box is a straight line of N cells (the road) which can each only have two values: 1 if a car is present on that section of road, 0 otherwise. The update rules for each iteration are very simple:

- If the space in front of a car is empty then it moves forward one cell;
- Otherwise it stays where it is.

If we use $c(j)$ to indicate the state of the jth cell, and use a subscript n to represent the iteration, we can write down rules that determine $c_{n+1}(j)$ from values at the previous iteration $c_n(j-1)$, $c_n(j)$ and $c_n(j+1)$.