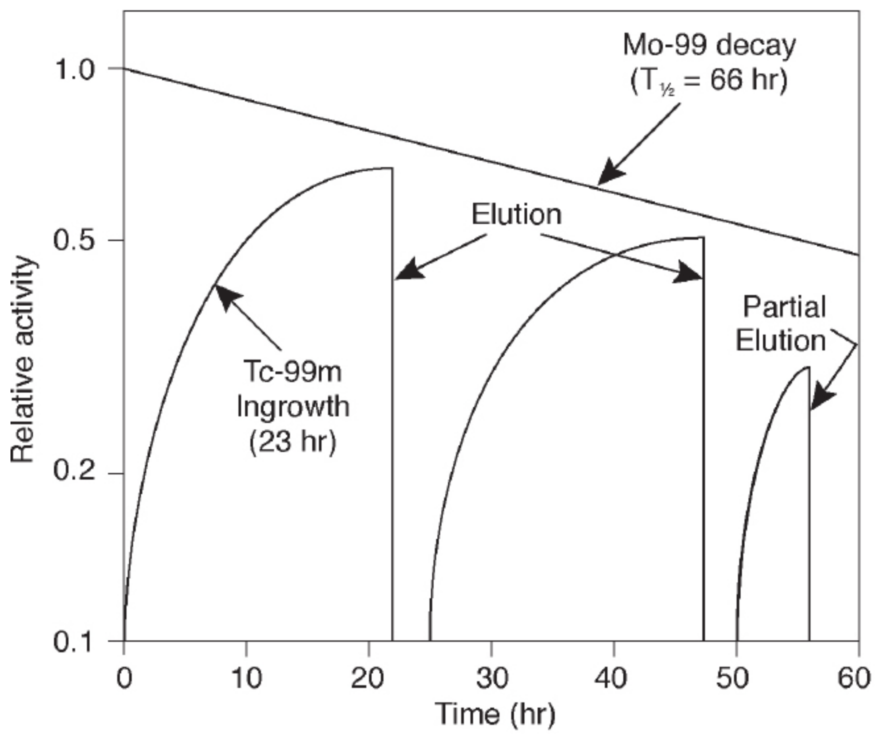
Project 4: Radionuclides

ENGR 151

November 13, 2023

Due December 1, 2023, 11:59 P.M.



1 Introduction

One important medical technique for diagnosis and treatment is the use of radiopharmaceuticals, i.e. radioactive nuclei attached to pharmaceutical drugs. For example, by attaching a fluorine-18 nucleus to an analogue of a glucose molecule, the radioactive nuclei will be concentrated in areas of high metabolic uptake (such as hard-working parts of the brain, or cancerous tumors). The emitted radiation can be detected and used to make an image or even a "movie" of what is happening in the body.

There are many different radioactive nuclei used in medicine for these purposes. These radioactive particles should be short lived so that they don't irradiate the body for much longer than the treatment/diagnosis and therefore they need to be produced at the hospital shortly before the procedure. One method is if a long-lived radioactive nucleus (the "parent" nuclide) decays to an appropriate short-lived nucleus (the “child” nuclide) that may be used for the procedure, the long-lived isotope can be stored on-site, with the short lived radionuclide building up over time until there is enough to be extracted as a sample. We are going to write a code that models the decay of radioactive “parent” nuclei into their “children” using the `cstdlib` **random number generator**.

2 Radioactive Decay

Radioactive decay is a **random process**. Any single radioactive nucleus, which is characterized by its atomic number, has a fixed probability per unit time to decay, λ, which only depends on the type of nucleus. That means that in a **short** time interval t_0, **the probability that the nucleus will decay is** $P_{t_0} \approx \lambda \times t_0$.

The rate equation for the decay of a large number of **identical** nuclei $N(t)$ is

$$\frac{dN}{dt} = -\lambda N \ ,$$

i.e. describing the decay of N fluorine-18 nuclei for example. By inspection or otherwise, we can show that the solution is $N(t) = N_0 e^{-\lambda t}$, i.e. the number of nuclei decays exponentially starting from N_0, the initial number of nuclei.

For the “child” nucleus, it is more complicated because as it decays it is also being increased in number by the decay of the “parent”.

3 Project Details

3.1 Starter Code

In the `starter-code` directory on the Google Drive, you will find four files:

- `test_classes.cpp` – A main ‘driver’ program housing four tests for your classes
- `process_command_line.hpp` – To be implemented, see Section 3.3
- `radionuc.hpp` – To be implemented, see Section 3.4
- `fluorine.hpp` – To be implemented, see Section 3.5

Note that `test_classes.cpp` will not compile right away – it needs your implementations! However, we want to encourage you to work on the project section by section, so we are providing the following instructions for running portions of the project even if you have not implemented all of the portions yet. After you complete a portion, remember to add any necessary `#include` statements at the top of `test_classes.cpp`.

- Once you have finished `process_command_line.hpp`, you can comment out the function `test_case(...)` and its call in `main(...)`, and then `test_classes.cpp` should compile. Ask yourself how you might test your implementation of the `process_command_line(...)` function at this point?

- Once you have finished `radionuc.hpp`, you can comment all of `test_cases(...)` back in *except for case 4*, and `test_classes.cpp` should compile. You should be able to run cases 1, 2, and 3 at this point.
- Once you have finished `fluorine.hpp`, `test_classes.cpp` should compile as given, and you should be able to run all four test cases.

3.2 Testing

The file `test_classes.cpp` in the starter code has four test cases implemented in it which should help you test your project. For a detailed explanation of each test case, and the desired output for a given seed value, see section 5.1 in the Appendix.

3.3 Task 1: Command Line Parsing

For the first part of this project, you will be writing a function to process the command line arguments of the program. The program expects exactly three command line arguments: the executable name, a seed value, and a test case number. The function declaration has been provided for you in the file `process_command_line.hpp`. For an explanation of how to handle command line arguments in C++, see section 5.2 in the Appendix. For this project, the program expects the following command line arguments in this order:

```
./<executable_name> <seed> <case_number>
```

In `process_command_line.hpp` you must:

1. Process the command line arguments and set `case_number` to the correct value.
2. Seed the random number generator. This can be accomplished by simply running "`srand(<seed>);`".

An important part of writing any program is error checking your inputs. You would not want your program to do something unexpected if somebody fed it inputs it was not expecting. Therefore, your program should guard against the following likely error cases.

1. Your program is only expecting three command line arguments, and it should check that this is the case. If it is not, then print "Incorrect number of arguments." and call `exit(1)`.
2. There are only 4 test cases, and so your program should check that the command line argument for the case number is between 1 and 4. If it is not, then print "Case number must be either 1, 2, 3, or 4." and call `exit(1)`.

Note: When you use `exit(1)` in a program, you are essentially signaling to the operating system that the program should terminate with an error status (in this case, 1 indicates an error). The program will terminate immediately after running exit(1).

3.4 Task 2: Radionuclide Class

For the second part of this project, you will be writing a class to model the behavior of a homogenous population of radionuclide molecules. Sparse starter code has been provided for you in the file `radionuc.hpp`. You may implement private member variables as you see fit. Below is a list of the public member functions your class should implement.

- `radioNuclide()`
  A default constructor for your class.

- `radioNuclide(const double, const int)`
  A constructor which takes in a value for λ and the population size (the number of nuclei).

- `int Count()`
  A 'getter' which returns the current population size.

- `int Decay(const double)`
  Takes in a value representing a short length of time, $t_{interval}$. The function must then:

  1. Compute the probability for a given single nucleus to decay in $t_{interval}$ seconds.
  2. Use that probability to compute the number of atoms in the population which decay in $t_{interval}$ seconds. This should be accomplished by computing a random number between 0 and 1, and then comparing that number to the probability you computed in step 1. If the generated number is less than the probability, then that particular nucleus decayed, otherwise, that particular nucleus did not decay. Repeat this procedure for all nuclei in the population.
  3. Reduce the population size by the number of atoms that decayed, and return the number of atoms that decayed.

  Hint: You have seen this function depends on a random number generator, and so its behavior will be somewhat random. In order to allow the autograder to grade your program, you will have to use a **seeded random number generator**. A seeded random number generator is a random number generator which first takes in a value (a seed), and, for any given seed, always produces the *exact same* sequence of random numbers. In Task 1 you seeded your random number generator. Now, to generate a random number between 0 and the built-in constant `RAND_MAX`, call `rand()`.

- `void Add(const int)`
  A 'setter' which takes as an input an integer number of radionuclide atoms to add to the population, and adds them to the population.

- `void Zero()`
  A 'setter' which sets the population size to zero.

- `void operator+=(const int)`
  An overload of the `+=` operator which behaves exactly like `Add()`.

You must also implement an overload of the `<<` operator which prints out the population size of the input `radioNuclide` instance, although this should **not** be a member function of your class.

3.5 Task 3: Derived Fluorine18 Class

For the third part of this project you will be writing a derived class `Fluorine18` in `fluorine.hpp`. This class should inherit from `radionuclide` and model the decay of the Fluorine-18 nuclei. Similar to the class `radionuclide`, you are allowed to implement private member variables as you see fit (Note: private member variables and functions are not inherited by a derived class). The value of λ of Fluorine18 is log(2.0) / (109.771 / 60.0). Your class derives the necessary public member functions from `radionuclide`, and the constructors that your class should implement are below:

- `Fluorine18()`
  A default constructor for your class.
- `Fluorine18(const int)`
  A constructor which takes in a value for the population size (the number of nuclei).

4 Grading

The project should be submitted to Project 4: Radioactive Decay on autograder.io. Only submit the three header files you write, `process_command_line.hpp`, `radionuc.hpp`, `fluorine.hpp` to the autograder, do not submit your `test_classes.cpp` file. For this project you will only be allowed **a maximum of 5 submissions per day to the autograder**, so verify your code works before submitting! If in doubt seek help from office hours/piazza.

This project will be graded in two parts. First, the autograder will evaluate your submission and provide a maximum score of 100 points. Second, one of our graders will evaluate your submission for style and commenting, and will subtract 0-10 points from the score that autograder evaluated. In Table 1 you can find a breakdown of the point available from the autograder, and in Table 2 you can find a breakdown of the penalties that will be applied for poor coding style.

Table 1: Point breakdown for autograder.

Task	Possible Points
Project file is correctly named and compiles successfully	-
Generating proper outputs from given case inputs	80
Correctly tests for command line input errors	20

Table 2: Point breakdown for style. **Note these points will be deducted for poor style, not added for acceptable style.**

Stylistic Item	Deducted Points
Each file has name/section number in header comment	2
Comment are used appropriately (major steps explained)	2
Indenting and whitespace are appropriate	2
Variables have meaningful names	2
Program is modular and organized	2

5 Appendix

5.1 Test Cases

There are four test cases in `test_classes.cpp`:

- Test case 1 uses your `radioNuclide` class to create a population of 10,000,000 Fluorine-18 nuclei and prints the size of that population as it decays over 20 simulated hours. If you have implemented your `radioNuclide` class correctly, the last four lines of output for test case 1, with seed value 0, should be:

  ```
  19.97 5060
  19.98 5041
  19.99 5018
  20 4996
  ```

- Test case 2 uses your `radioNuclide` class simulate a technetium generator. First, it creates a population of 10,000,000 Molybdenum-99 nuclei, and a population of 0 Technetium-99m nuclei. As Molybdenum-99 decays it, it produces Technetium-99m, and this test prints the size of the Technetium-99m population over 100 simulated hours. If you have implemented your `radioNuclide` class correctly, the last four lines of output for test case 2, with seed value 0, should be:

  ```
  97 32056
  98 31608
  99 31287
  100 30979
  ```

- Test case 3 is very similar to case 2, except that the Technetium-99m is extracted from the generator every 12 hours. If you have implemented your `radioNuclide` class correctly, the last four lines of output for test case 3, with seed value 0, should be:

  ```
  132 16331
  144 14265
  156 12381
  168 11002
  ```

- Test case 4 is exactly the same as Case 1, except that your `fluorine18` class is used to simulate the population of Fluorine-18 nuclei, instead of your `radioNuclide` class. If you have implemented your classes correctly the last four lines of output for test case 4, with with seed value zero, should be exactly that same as the last four lines from test case 1 above.

Notice also that since the populations of radionuclides decay exponentially, you can produce ballpark estimates for what the correct outputs should be using any graphing software fairly easily (MATLAB, Microsoft Excel, Desmos, etc). **In many cases, these ballpark estimates may actually be more helpful for debugging purposes than the rigid correct answers.**

5.2 C++ Command Line Argument Parsing

In `test_classes.cpp`, you may notice the `main` function takes in two parameters you have not seen before: `argc` and `argv`. These are standard parameters used to allow C++ programs to interface with the command line.

- `int argc` is equal to the number of command line arguments received.
- `char* argv[]` is an array of strings representing the command line arguments received.

For example, if we compile `test_classes.cpp` into an executable named `project4.exe` and run the program With the following command:

```
./project4.exe 2 1
```

Then the values of `argc` and `argv` in the `main` function would be:

```
argc = 3
argv[0] = "project4.exe"
argv[1] = "2"
argv[2] = "1"
```

And these values can be accessed just like one would normally access the value of any other integer or array. Note that in the command line, arguments are delimited by a space character, not by a comma.