Project 4: Radionuclides

ENGR 151

Released: November 25, 2024

Due: December 09, 2024, 11:59 P.M.

1 Introduction

Exciting news! You have just been hired as an undergraduate research assistant for a laboratory in the NERS Department which works on applications of radioactive materials in medical science. You will be working on modeling radioactive particles called "radionuclides". Your first major project as a member of this lab is to write a simulation of a Technetium Generator, which is a device that extracts the short-lived isotope Technetium-99 from the more stable isotope Molybdenum-99. To help you accomplish this, we have broken the project into three tasks, listed below.

1. Writing a class to model the behavior of a homogeneous population of radionuclides.
2. Writing two classes modeling populations of the particular radionuclides Molybdenum-99 and Technetium-99, as child-classes of the previous generic radionuclide class.
3. Simulating the Technetium Generator:
   - Writing a function to read and store (or "parse", in CS lingo) useful simulation parameters supplied at the command line.
   - Writing the simulation itself.

Each of these tasks is detailed further in the following sections. Do not worry if you have no experience with radionuclides, all the background you need is provided in this specification. As always, we strongly encourage you to read this project specification thoroughly, and to start the project early!

2 Background Information: Radioactive Decay

The main behavior of radionuclides which we are interested in modeling is the random process known as radioactive decay. Any single radioactive nucleus, has a fixed probability to decay per unit time, λ, which depends only upon the type of nucleus. That means that in a short time interval t_0, the **probability that the nucleus will decay is** $P_{t_0} \approx \lambda \times t_0$**.**

As is the case with many real world systems, a differential equation (an equation involving derivatives) can be used to model the behavior of a population of radionuclides. The differential equation governing a large population of **identical** radionuclides $N(t)$ is:

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

By inspection or otherwise, we can show that the solution is:

$$N(t) = N_0 e^{-\lambda t}$$

where N_0 is the initial number of nuclei. Practically speaking, this means that over time, a given population of radionuclides will decay exponentially.

3 Generic Radionuclide Class

You will write a class to model the behavior of a homogeneous 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 which should set reasonable values for all private members of your class.
- `radioNuclide(const double lambda, const int N)`
  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 length of time, $t_{interval}$. The function must then:
  1. Compute the probability for a given single nucleus to decay in $t_{interval}$ seconds. See Section 2.
  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, well, 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 then, for any given seed, always produces the *exact same* sequence of "random" numbers. **You must seed the random number generator before using it!** This can be accomplished by running the line:

  ```
  srand(0);
  ```

  before you use the random number generator. Because this should be done **only once**, this line will not appear in your class, but rather at the beginning of your main program (the program you write in part 3, and any program you may write to test your classes). Then, to generate a random integer between 0 and the built-in constant `RAND_MAX`, call `rand()`. Notice you need a random double between 0 and 1, not a random integer between 0 and `RAND_MAX`, so think about how best to modify the random number you receive to achieve this.
- `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()`.

Note: Because the remainder of the project depends upon this class, it is a very good idea for you to test your implementation before moving on. To that end, we have provided a testing program in the file `testRadionuc.cpp`, as well as the expected output in `testRadionucOutput.txt`. To test your implementation, compile and run `testRadionuc.cpp`, pipe the output into a file, and use the Linux diff command to check whether your output is identical to the expected output. Be aware that the test program will take several minutes to run.

4 Molybdenum-99 and Technetium-99 Derived Classes

Now you will write two child-classes of the generic radionuclide class from Section 3 to model populations of Molybdenum-99 (Mo-99) and Technetium-99 (Tc-99). You should write these in two files names `molybdenum.hpp` and `technetium.hpp`, respectively. Similar to the class `radioNuclide`, you are allowed to implement private member variables as you see fit. The probabilities of decay for Mo-99 and Tc-99 are given here:

$$\lambda_{Mo-99} = \frac{\ln 2}{65.94} \qquad \lambda_{Tc-99} = \frac{\ln 2}{6.0067}$$

Remember, as child classes, these class will inherit all the public member functions from `radioNuclide`, so all you must do is define a new default and custom constructor for each class.

- `Mo99()`
  Default constructor for your Mo-99 class.
- `Mo99(const int)`
  Custom constructor for your Mo-99 which takes in a value for the population size (the number of nuclei).
- `Tc99()`
  Default constructor for your Tc-99 class.
- `Tc99(const int)`
  Custom constructor for your Tc-99 class which takes in a value for the population size (the number of nuclei).

Hint: This task does not require you to write a great deal of code, but rather to effectively leverage inheritance to accomplish it.

5 Simulating a Technetium Generator

The radionuclide Technetium-99 (Tc-99) is useful for a variety of diagnostic medical procedures. Unfortunately, it has a half-life of about 6 hours, and thus is very short lived. This makes it difficult to store or transport Technetium-99 over appreciable distances on its own. A Technetium Generator is a device invented in 1958 to solve the problems brought on by Tc-99's short life span. It uses a much longer lived isotope, Molybdenum-99 (Mo-99), which naturally decays into Tc-99 as a long-term of the valuable isotope. You will now write a program which simulates Technitium Generator with an initial population of Mo-99 nuclei over a given time frame, including the processes of:

- Mo-99 decaying into Tc-99
- Tc-99 decaying
- Harvesting the Tc-99 once a sufficient population has been built up

You should write this simulation in the file `tcGenerator.cpp`, where you will find a bit of starter code.

5.1 Process Command Line Arguments

You simulation program should take in two arguments at the command line, first, the size of the initial Mo-99 population, and second, the length of the simulation in hours. For example, if you compiled `tcGenerator.cpp` into an executable called `tcgenerator.exe`, and you wanted to simulate the generator with an initial population of 1,000,000 Mo-99 nuclei over 100 hours, you would run the following line:

```
./tcgenerator.exe 1000000 100
```

To accomplish this, see we clearly need a way to interact with arguments given at the command line. Luckily, C++ has this functionality built in. If you look at the definition of the `main` function in `tcGenerator.cpp`, you will notice it 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.

To illustrate the utility of these parameters, let us consider an example. If we compile a program called `helloWorld.cpp` into an executable named `helloWorld.exe`, and run that program with the following command:

```
./hellowWorld.exe my names ralph 3
```

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

```
argc = 5
argv[0] = "helloWorld.exe"
argv[1] = "my"
argv[2] = "names"
argv[3] = "ralph"
argv[4] = "3"
```

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

Implement the function `processCommandLine` in `tcGenerator.cpp`. Because you wouldn't want your program doing something unexpected if it gets bad arguments at the command line, you must guard against the following conditions on the supplied command line arguments:

- There should be only three command line arguments supplied. If to many or too few are given, print "Incorrect number of command line arguments." and then call `exit(1)`.
- The initial population of Mo-99 specified must be between 10^3 and 10^7 nuclei, inclusive. If it is not, print "Invalid initial Mo-99 population." and call `exit(1)`.
- The length of the simulation must be between 10 and 500 hours, inclusive. If it is not, print "Invalid simulation length." 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).

5.2 Simulation Specifics

Now its time to implement the simulation itself! **Do not forget** to seed your random number generator with 0, as seen in Task 1! You should do this in the function `technetiumGenetor`. Recall that when a nucleus of Mo-99 decays, it creates a nucleus of Tc-99, and consider how to put this into the simulation using the functions you have written for your radionuclides. You simulation must:

- Start with an initial population of Mo-99 nuclei as specified by the command line arguments received, and an initial population of zero Tc-99 nuclei.
- Use 1 hour as the time-step.
- Run for the number of hours specified at the command line (inclusive).
- Print out the results as it runs:

  - Once, in the beginning, print column headers:

    ```
    t   Mo99    Tc99
    ```

    Note that the spaces you see should be tab characters.
  - At the beginning of each pass of the simulation, print out the current time in hours, the size of the population of Mo-99 nuclei, and the size of the population of Tc-99 nuclei, again separated by tab characters.

- Harvest the Tc-99 nuclei. Whenever the population of Tc-99 nuclei reaches 10,000, remove all the Tc-99 nuclei. Keep track of how many you harvest over the course of the simulation, and return this number at the end of the function.

Finally, after the simulation is complete, print the following:

```
Initial Mo99 population: <number>
Simulation length: <number> hours
Total Tc99 harvested: <number>
```

Where `<number>` is replaced by the appropriate quantity for that simulation. Note that there should be a new line after your last output (after the line state the total Tc-99 harvested). To help you debug your implementation, we have provided the full correct output to a simulation of the Technetium generator for an initial Mo-99 population of 1,000,000 over a time span of 75 hours in `output_tcgen_1e6_75.txt`. To be clear, if you have implemented your Technetium generator correctly, when you run:

```
./tcgen.exe 1000000 75
```

you should get exactly the same output as is in the given file.

6 Grading

The project should be submitted to Project 4: Radioactive Decay on autograder.io. You should submit the following files:

- `radionuc.hpp`
- `molybdenum.hpp`
- `technetium.hpp`
- `tcGenerator.cpp`

Do not submit `testRadionuc.cpp`. 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, we 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
Task 1	30
Task 2	10
Task 3: Input Validation	20
Task 3: Simulations	40
Total	100

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