

Background Reading #3

Table of Contents

1. Package Summary.....	2
2. NumPy Package	2
3. Radioactivedecay Package	3
4. Integrate.solve_ivp Function from Scipy Package	5
4.1 Setting up the solve_ivp Integrator	5
4.2 Decay with Non-Constant Production Example	6
5. Matplotlib Basics	9
6. References	10

1. PACKAGE SUMMARY

This document (and the coding exercise) requires the use of several Python packages. These are listed in the table below along with a link to more information and examples for each one.

To install a package, follow the instructions provided in the **Package Import Instructions** file on D2L, found under the Coding Exercises > Download and Setup Instructions folder.

Package or Function Name	Optional Documentation or Useful Guides
Radioactivedecay	https://radioactivedecay.github.io/overview.html
Scipy.integrate	https://docs.scipy.org/doc/scipy/reference/integrate.html
Scipy.integrate.solve_ivp	Documentation (click me)
NumPy	This is a huge fundamental package, so there is no point in providing a link to anything specific. Google is more useful here for individual functions as we come across them.
Matplotlib	https://matplotlib.org/

2. NUMPY PACKAGE

NumPy is the fundamental package for scientific computing in Python. It is a collection of modules, called a library, that gives the programmer powerful array objects and linear algebra tools, among other things. Although it is primarily used for arrays (list of elements), it can also be used for single input values.

We will start getting comfortable with NumPy now by using it for basic operations (even ones without arrays).

```
import numpy as np

# using np.log10 to take log base 10 of array and single input
A = [1, 2, 3]
print(np.log10(A))

B = 4
print(np.log10(B))
```

Console Output:

```
[0.          0.30103    0.47712125]
0.6020599913279624
```

3. RADIOACTIVEDECAY PACKAGE

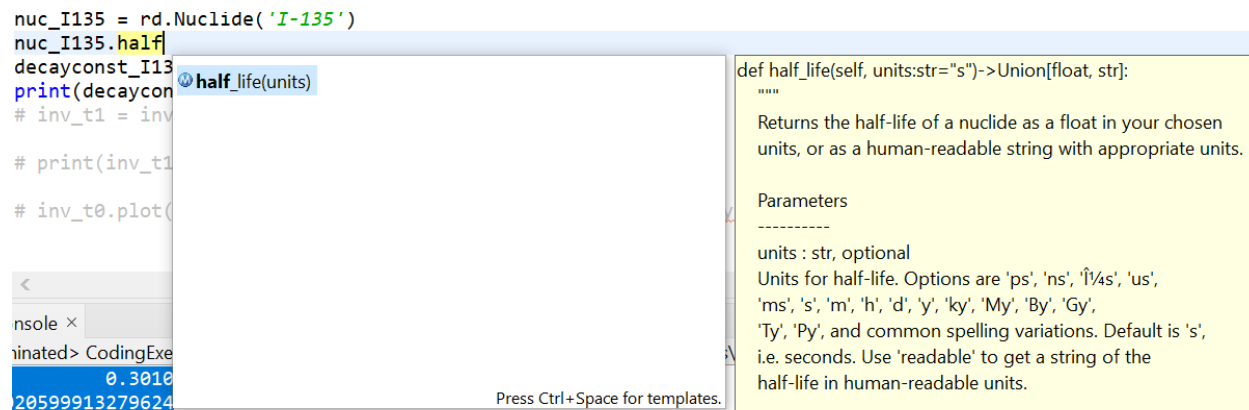
Radioactivedecay is a Python package for radioactive decay calculations. It contains functions to define inventories of radionuclides, perform decay calculations, and output decay data for radionuclides and decay chains. More information and some good examples of its capability can be found here: <https://radioactivedecay.github.io/overview.html>

This package is similar to the **periodictable** package that was introduced in Coding Exercise #2, but is more geared towards radioactive isotopes. We will begin by obtaining some useful information for the isotope iodine-135 (I-135). To define an isotope, we use the `radioactivedecay.Nuclide(isotopeidentifier)` function, where `isotopeidentifier` is a string in the form of element symbol-atomic mass number, e.g., 'I-135':

```
import radioactivedecay as rd

nuc_I135 = rd.Nuclide('I-135')
decayconst_I135 = 0.693 / (nuc_I135.half_life('s'))
print(decayconst_I135)
```

To obtain the half-life of I-135, we use the `radioactivedecay.half_life(units)` function, where a string input for the desired units of the half-life is the optional input. We can see that starting to type the name of the function shows the following information:



The screenshot shows a Jupyter Notebook interface. On the left, a code cell contains the following code:

```
nuc_I135 = rd.Nuclide('I-135')
nuc_I135.half
decayconst_I135 = 0.693 / (nuc_I135.half_life('s'))
print(decayconst_I135)
# inv_t1 = inv
# print(inv_t1)
# inv_t0.plot()
```

Below the code cell, the console output shows:

```
0.3010
20599913279624
```

On the right, a pop-up window displays the documentation for the `half_life` function:

```
def half_life(self, units:str="s")->Union[float, str]:
    """
    Returns the half-life of a nuclide as a float in your chosen
    units, or as a human-readable string with appropriate units.

    Parameters
    -----
    units : str, optional
        Units for half-life. Options are 'ps', 'ns', '1/4s', 'us',
        'ms', 's', 'm', 'h', 'd', 'y', 'ky', 'My', 'By', 'Gy',
        'Ty', 'Py', and common spelling variations. Default is 's',
        i.e. seconds. Use 'readable' to get a string of the
        half-life in human-readable units.
```

The pop-up in the screenshot above indicates that the default unit if we put nothing in parentheses is seconds (although it is good to specify this for via 's'). Other input options are also given.

Some of the features of this package are:

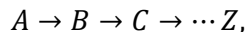
- **progeny()** method will find all of the daughter products in a decay chain - this is a quick way to check you have correctly found the right isotopes when working through assignments or practice questions, e.g., we can check the decay products of I-135.

```
nuc_I135 = rd.Nuclide('I-135')
NextDaughters = nuc_I135.progeny()
print(NextDaughters)
```

Console Output:

```
['Xe-135', 'Xe-135m']
```

- **Inventory()** and **decay()** methods, used to input the initial amount of a species undergoing decay and the period of time, respectively. This can be used to determine the amount of all species in that chain, but it is limited to decay chains (no production term for the parent species at the start of a decay chain), i.e., this is only for variable production via pure decay, i.e.,



where we start with some initial (non-replenishing) amount of A, and there are no other forms of production. E.g., we can create an inventory of 10.0 g of I-135 and set this amount to decay for 26.4 hours. The amount of I-135 reduces by a factor of 4, i.e., to 0.625 g as the half-life of iodine-135 is 6.6 h. The amounts of all other species down the chain are determined. Inputs and outputs include masses, moles and numbers of atoms:

```
# initial amount of parent (t=0)
inv_mass_t0 = rd.Inventory({'I-135': 10.0}, 'g')
# set time elapsed to calc new inventory after decay takes place for this time
inv_mass_t1 = inv_mass_t0.decay(26.3, 'h')
# obtain masses or moles of decay products
masses_progeny = inv_mass_t1.masses('g')
moles_progeny = inv_mass_t1.moles('mol')

# prints all daughters (listed alphabetically)
print (
    'amounts of all daughters in units specified when masses_progeny made: \n',
    masses_progeny)
print()
# prints only selected daughters, using square bracket 'X-A' notation
print (
    'moles Ba-135: ', moles_progeny['Ba-135'],
    '# atoms Ba-135:', inv_mass_t1.numbers()['Ba-135'], '\n',
    'activity I-135 in Mega Curies:', inv_mass_t1.activities('MCi')['I-135']
)
```

Console Output:

```
amounts of all daughters in units specified when masses_progeny made:
{'Ba-135': 5.896207072201131e-10, 'Cs-135': 3.246231847152375, 'I-135': 2.5,
'Xe-135': 4.236867060872483, 'Xe-135m': 0.016713762593710487}
```

```
moles Ba-135: 4.370614123152825e-12 # atoms Ba-135: 2632045345727.029
activity I-135 in Mega Curies: 8.838985965898791
```

- **plot()** method, which generates a quick graph of all daughter products (or a decay “branching”) plot. We will introduce plotting in more detail at the end of this reading, but this is a quick and easy way to make plots when using this package from purely decay reactions.

```
# can look at order of daughter products graphically
nuc_I135.plot()
# or rd.Nuclide('I-135').plot() would work if nuc_I135 not defined previously
inv_mass_t0.plot(100, 'h')
# plt.show()
```

4. INTEGRATE.SOLVE_IVP FUNCTION FROM SCIPY PACKAGE

While the **radioactivedecay** package could easily be used to carry out some activity calculations, it is limited to decay chains (meaning no production term for the parent species at the start of a decay chain). As a result, we need to know how to deal with nuclear reactions with additional production pathways via a more traditional ordinary differential equation (ODE) solver. This method can be used for any type of production, as this is simply a way to numerically solve first order differential equations.

Among many other useful things, the **scipy** package has several functions that will numerically solve initial value problems for a system of ODEs. These can be accessed through **scipy.integrate**, which contains several functions to help with general integration and ODEs. More information can be found here:

<https://docs.scipy.org/doc/scipy/reference/integrate.html>

In this guide and in the accompanying Coding Exercise #3, we will be using the **solve_ivp** function. This function has the form:

```
scipy.integrate.solve_ivp(fun, t_span, y0, method='RK45', t_eval=None,
dense_output=False, events=None, vectorized=False, args=None, **options) (1)
```

A more detailed overview of each of the input arguments of **solve_ivp** can be found here: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html.

By default, **solve_ivp** uses the Runge–Kutta–Fehlberg (RK45) method. Like all Runge-Kutta methods, this is a variation of Euler's method, where the solution is determined iteratively by discretizing a time step. Since this is not a numerical methods course, no more detail on the RK45 method will be provided here.

4.1 Setting up the solve_ivp Integrator

If we write a general ODE in the form of:

$$\frac{dy}{dt} = f(y, t), \quad (2)$$

The argument **fun** in **solve_ivp** (first input in Equation 1) is the right-hand side, or some function that depends on **y** and **t**. Typically, we make a separate function for this, which we then call when applying the **solve_ivp** function to find our solution.

Let's use the example of the decay of a parent to create a daughter product that is also radioactive:



$$\frac{dN_A}{dt} = -\lambda_A N_A, \quad \frac{dN_B}{dt} = \lambda_A N_A - \lambda_B N_B \quad (4)$$

We set up a function for the right-hand sides of this system of ODEs in `solve_ivp` by first importing `solve_ivp` from `scipy.integrate` in addition to importing several other useful packages and Avogadro's constant. We then start building the **decay_funcs** function, using the usual notation,

```
def functionname(argument1, argument2,...):
    output1 = f(argument1 and/or argument2)
    output2 = f(argument1 and/or argument2)
    return output1, output2, ...

from scipy.integrate import solve_ivp
from scipy.constants import Avogadro as NA
import numpy as np
import radioactivedecay as rd

def decay_funcs(t, y, k1, k2):
    # decay only ODE, production = 0
    # units for y dictated by initial input
    # E.g., if y_0 is atoms, y = [atoms]
    dy0 = -k1 * y[0]
    dy1 = k1 * y[0] - k2 * y[1]
    dy = [dy0, dy1]

    return dy
```

Generic y and t notation is used here, which corresponds to the following terms from Equation 4:

$$dy_0 = \frac{dN_A}{dt}, \quad dy_1 = \frac{dN_B}{dt}, \quad y[0] = N_A, \quad y[1] = N_B \quad (5)$$

Because Python starts indexing at 0 (instead of 1), $y[0]$ is the variable N_A in the first ODE. The left-hand side of the first ODE has been named “ dy_0 ” here to match this notation. The return, “ dy ”, must be in the form of an array or list that consists of all of the right-hand side expressions in the system of ODEs, i.e., $dy = [dy_0, dy_1]$.

Only the arguments, t (what we are differentiating with respect to) and y (what we are looking to solve for), are required input. All additional parameters following these are optional arguments. Here, these are k_1 and k_2 .

4.2 Decay with Non-Constant Production Example

Let's say that our system of ODEs is for the decay of I-135, which forms Xe-135. Let us also assume that there is no Xe-135 and 10 g of I-135 initially ($t = 0$).

We can make a list of these isotopes, followed by using the `radioactivedecay` package to extract some useful properties. We will write a function to do this, which will allow us to iterate through any number of input isotopes quickly and easily.

This function takes the string input for each isotope that is passed through it, e.g., “I-135”, and calculates the decay constant in h^{-1} . It also sets an initial value of 10 g, which is converted to atoms, for only the first isotope in the input list. All other elements in the isotope list result in an initial value of 0 atoms.

```
iso_list = ['I-135', 'Xe-135']
```

```
def initial_val_decay_const(iso):
    # based on input isotope (iso), calcs decay constant
    # sets initial amount in grams, converted to atoms for initial value
    nuc = rd.Nuclide(iso)
    # log10 is base ten, log is natural log (ln)
    decay_const= np.log(2) / nuc.half_life('h')

    if iso == iso_list[0]:
        input_g = 10
        nuc_MW = nuc.atomic_mass
        N_0_atoms = NA * input_g / nuc_MW

    else:
        N_0_atoms = 0

    return N_0_atoms, decay_const
```

We can now call **initial_val_decay_const** to get the decay constants and initial values that we will use as input along with the **decay_funcs** function we created above to solve our system of ODEs. We first make some empty lists (**k** and **initialvalues_y_0**), then loop through our list of isotopes, calling the **initial_val_decay_const** function and appending (adding) the returns to the empty lists.

```
initialvalues_y_0 = []
k = []

for iso in iso_list:
    k.append(initial_val_decay_const(iso)[1])
    initialvalues_y_0.append(initial_val_decay_const(iso)[0])
```

Lastly, we set up the range of times over which we wish to solve our ODEs along with which specific values of t we would like to evaluate N_A and N_B at. Let's set the range over which to integrate as 0 to 80 hours and choose a large set of times at which to calculate the solution. This will give us many (t, y) points, which will be convenient for plotting our output (Section 5).

```
overalltime_range = [0, 80]

t_solutions = []
for i in range(0, 80, 2):
    t_solutions.append(i)
```

We recall that **range(0, 80, 2)** means all values between 0 and 79 (80 is not included) at a time step of 2, meaning every other number. We are now ready to calculate the solution to the system of ODEs, which we will assign to the variable **sol**:

```
sol = solve_ivp(
    decay_funcs, overalltime_range, initialvalues_y_0, t_eval=t_solutions,
    args=(k), dense_output = True
)
```

Note: all input to solve_ivp must be in the form of lists, i.e., comma separated list entries inside square brackets. If we wanted for the ODEs to be solved at only a single value of t , e.g., $t = 40$ hours, we would define `t_solutions = [40]`, a single element array.

If we try to print the variable `sol`, this generates some basic and not very useful information from the solver. To extract values of `y[0]` and `y[1]` or N_A and N_B , we need `sol.y[0]` and `sol.y[1]`. These are the solutions at each of the time points we asked for in calling the solver function.

```
print('solution to first ODE, I-135:', sol.y[0])
print('')
print('t-values ODEs solved at:', sol.t)
```

Console Output:

```
solution to first ODE, I-135: [4.46381892e+22 3.61467137e+22 2.92694920e+22
2.37017605e+22
1.91940717e+22 1.55412737e+22 1.25850711e+22 1.01926153e+22
8.25274954e+21 6.68210268e+21 5.41217387e+21 4.38337250e+21
3.54841945e+21 2.87311641e+21 2.32762378e+21 1.88551547e+21
1.52623781e+21 1.23562036e+21 1.00105633e+21 8.11206599e+20
6.56705188e+20 5.31554984e+20 4.30551659e+20 3.48964410e+20
2.82621633e+20 2.28739015e+20 1.85211688e+20 1.50094385e+20
1.21627102e+20 9.84486756e+19 7.96908076e+19 6.45585256e+19
5.23288767e+19 4.23739812e+19 3.42951374e+19 2.77717960e+19
2.25081426e+19 1.82372325e+19 1.47666132e+19 1.19580936e+19]

t-values ODEs solved at: [ 0  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78]
```

Since we selected a lot of points to find the solution at (handy for plotting in next section), it would be convenient to also plug in a value of t and get a solution. This can be done by setting the **dense_output** input argument in **solve_ivp** to “True”, as done above (by default, this is set to False). We can then use the following notation to extract continuous output from our integrator:

```
t_input = 40
specific_sol = sol.sol(t_input)
print(specific_sol)
```

Console Output:

```
[6.56705188e+20 5.30875257e+21]
```

These are the values for N_A and N_B (the atoms of I-135 and Xe-135), respectively at $t = 40$ hours.

Note: A word of caution. Earlier, we specified times at which we wanted `solve_ivp` to evaluate our ODEs. Using the `dense_output` parameter tells the function to interpolate between the time steps we have specified instead of calculating the solution using the solver (e.g., the RK45 default).

5. MATPLOTLIB BASICS

Matplotlib is a library for Python that allows you to plot lists of x and y points, as well as many other features. It is designed to be intuitive and easy to use, and it mimics the plotting interface of MATLAB, a widely used toolkit and language for applied mathematics and computation. Therefore, if you have experience with MATLAB, using Matplotlib will be very familiar.

In the following example, a plot is created for the solutions of the I-135 system of ODEs from Section 4, and properly annotated using Matplotlib:

```
import matplotlib.pyplot as plt

# this first for loop is not necessary to make the plots, but is more
# convenient than having to declare a bunch of variables for just the
# I-135 y vals, then the Xe-135 vals (and so on, for a larger system of ODEs)

for solution in sol.y:
    # plot function takes (x point list, y point list)
    plt.plot(sol.t, solution, linestyle='-', marker='.')

# info provided on font style, this is in "dictionary" style
# we have not learned about dictionaries yet, but the defining
# of font properties using keywords below is easy to follow
font = {'family': 'arial',
        'color': 'black',
        'weight': 'normal',
        'size': 12,
        }

# plt.xlabel(xlabel, fontdict, labelpad)
# first arg is the string name of the label, second arg is the font info,
# third arg is the desired padding space between the axis values and the
# label location

plt.xlabel('Time (h)', fontdict = font, labelpad=8)
plt.ylabel('Number of Atoms', fontdict=font, labelpad=8)

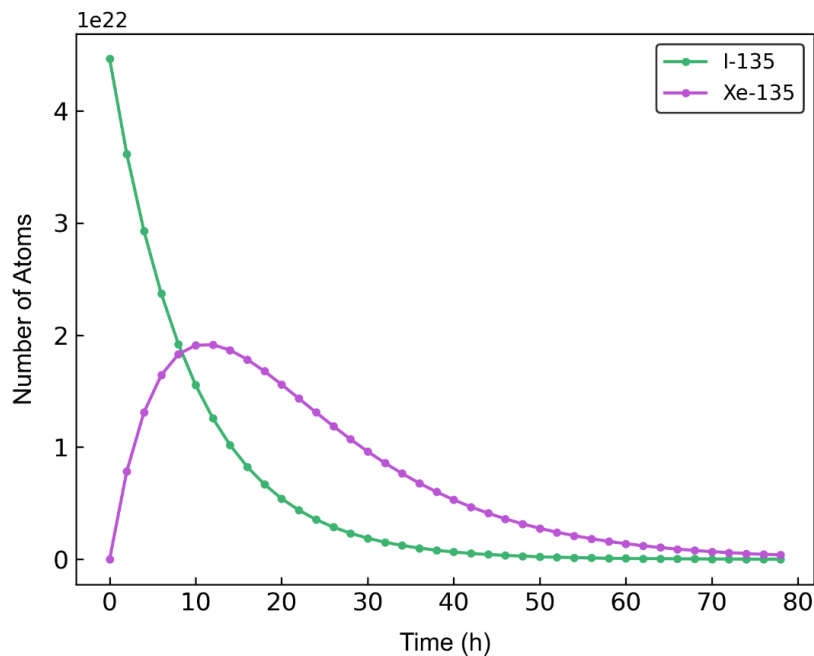
labels = iso_list
# labels are the string names of the different curves
# ncol is how many columns we want the legend labels to be in
# edgecolor is the colour of the legend box
# loc is the location of the legend
plt.legend(labels, ncol=1, edgecolor='black', loc='upper right')
# turns tickmarks inward...only monsters have their ticks facing outward...
plt.tick_params(axis="both",direction="in")
# shows plot, which is useful when tweaking so you don't have to keep opening
# the saved file. Comment out plt.show() when you are happy with the plot and
# would like to save the figure

plt.show()
# saves figure with the string name.extension of the filetype you would like
# as the first arg...dpi is the resolution (300 or 400 dpi is good)
# figure is saved to your Eclipse workspace folder by default
plt.savefig('I-135_activiy.png', dpi=300)
```

The customization options when making plots are endless. If you need to make many of the same plot, making a template will ensure your plots has similar formatting (a feat that is challenging to achieve in Excel). To select the colours of the lines on a plot, the “color” argument is used (American spelling). We can modify the above code as follows to include a list of colours for each curve. Similar customizations are possible for marker type, colour, size, etc.

```
plotcolours = ['mediumseagreen', 'mediumorchid']

for solution, colr in zip(sol.y, plotcolours):
    # plot function takes (x point list, y point list)
    plt.plot(sol.t, solution, color = colr, linestyle = '-', marker = '.')
```



Matplotlib takes base colour string inputs such as ‘red’ or ‘r’, ‘green’ or ‘g’, etc., but there are also many other options (e.g., Hex or RGB colours). This is a guide that has some great CSS colour options (and colour names) towards the bottom of the webpage (Happy colour choosing!):

https://matplotlib.org/stable/gallery/color/named_colors.html.

Note: multiple **plt.show()** commands throughout your code will require you to close all of the plots (hit the “X” on the plot pop-up windows) before the code will continue to run. This is not required for **plt.savefig()**, which will passively save the figure to your local git repo folder while the rest of the code continues to run; however, you will have to manually open the figure to inspect the plot.

6. REFERENCES

“Computational Nuclear Engineering and Radiological Science Using Python,” Ryan G. McClarren, Elsevier, 2017.