Programming for Medical Physics

Neil Kirby, Ph.D.

Assistant Professor

The University of Texas Health Science Center San Antonio

Adapted for Python by Noah Bice

Lecture 3: Functions and Classes

3.1 Functions

The previous lecture focused on scripting, which is preferable to directly using the command line for complex calculations. Deciding to write a **function** is not about calculation complexity, it's a question of how often a calculation must be performed. For example, something that I commonly need in my programs is a source of Gaussian-distributed noise. The code to generate v-shaped Gaussian noise is not that complex and only requires 3 lines (excluding import statements):

```
import numpy as np
from scipy.special import erfinv

x = np.random.random(v)
x = (x-0.5)*erfinv(x*np.sqrt(2/np.pi))
x *= sigma
```

Although this code is not that long, its components are not something that I can recall from memory. Thus, if I were to insert this in a script, I would search for a program where I have used it and copy-paste it into my new script. Instead, I've turned it into a function, which allows me to generate Gaussian noise with the following line of code.

```
def gaussrand(v, sigma=1.):
    x = np.random.random(v)
    x = (x-0.5)*erfinv(x*np.sqrt(2/np.pi))
    x *= sigma
    return x

rand = gaussrand((3,2))
print(rand)
```

```
In [6]: runfile('C:/Users/noaht
[[-0.03696642  0.21333348]
 [ 0.23219659  0.00861551]
 [-0.04235102 -0.03835385]]
```

Any Python script can be interpreted as a standalone file or as a module. You can create functions and use them within the same script, or you can save them as .py files and import them in different scripts or from the command line. Note that if you are creating a larger Python project which contains

modules in subdirectories, you'll have to tell the Python interpreter to look for submodules with an **__init__.py** file. For example, suppose I have a function called my_function() saved in the file submodule.py at C:\Users\bicen\Documents\example\module\submodule.py. If I simply try to import my_function() from a Python prompt opened from the directory module\, the Python interpreter throws an error.

```
C:\Users\bicen\Documents\example\module>ls
submodule.py

>>> from module.submodule import my_function
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named module.submodule
```

Now if I create an empty file called __init__.py, the Python interpreter knows to search the module\ directory for submodules. I can now import my_function().

```
C:\Users\bicen\Documents\example\module>ls
__init__.py  submodule.py

>>> from module.submodule import my_function
```

The syntax for creating a function is a def statement followed by your function's name along with parentheses containing the arguments you want your function to take, followed by a colon and indented block of code. There are two types of arguments to Python functions: **positional arguments** and **keyword arguments**. Positional arguments must be entered every time a function is called, while keyword arguments have default values, so an input is not always required by the user. Positional arguments can be used by listing them in the order that they are required, and keyword arguments can be used with the syntax keyword=value. Following the indented code there should be a return statement followed by the function's output.

Note that if you have any code in a script which you are importing from that is not a class or function, it will be executed on import. For example, suppose that my file module\submodule.py contains the following code:

```
def my_function():
    print('This is a function from a submodule.')
    return

print('This code will be executed on import.')
```

Note that even though I don't expect any outputs from my_function(), I still write the return statement. The return statement is not necessary to satisfy python syntax, but it helps me visually sort blocks of code. Now, when I try to import my_function() in the command line, the second print() function is executed.

```
>>> from module.submodule import my_function
This code will be executed on import.
```

We can avoid this conflict by telling the Python to compute the print() function only if submodule.py is the primary file being executed. We can do this with the conditional if __name__ == "__main__".

```
def my_function():
    print('This is a function from a submodule.')
    return

if __name__ == "__main__":
    print('This code will not be executed on import.')
```

At the onset of interpretation of a script, Python creates a few special variables, one of them being __name__. A module's __name__ variable is set to "__main__" only if it is the script set to be executed. You'll often find Python packages that use this conditional for the purpose of testing functions.

We can now import my_function without executing the second print() function. Remember that to call a function, even if it requires no positional arguments, you need to use parentheses.

```
>>> from module.submodule import my_function
>>> my_function
<function my_function at 0x0000000003B170B8>
>>> my_function()
This is a function from a submodule.
```

When writing a function, the names of arguments and any variables defined inside the function are local. Any variables that you define outside of the function are still accessible within the function. However, if you choose to use a variable of the same name both globally and locally, the local assignment is favored inside the function's code. For example.

```
def print_X():
    X = 'Local variables rule!'
    print(X)
    return

X = 'Local variables drool!'
print_X()
print(X)
```

```
In [7]: runfile('C:/User
Local variables rule!
Local variables drool!
```

Similar to writing loops, there exists a compact syntax for creating short, "anonymous" functions, called **lambda functions**. The syntax for using a lambda function is lambda arguments: calculations. This creates a function which can be used immediately or named and used elsewhere. For example, we can write a function that squares an input and apply it to the number 2.

```
>>> (lambda x: x*x)(2)
4
>>> square = lambda x: x*x
>>> square(2)
4

>>> power = lambda x, power=2: x**power
```

```
>>> power(2, power=3)
8
```

There are entire programming languages based on this structure of computation called lambda-calculus. Python is not one of them, but it does still allow for the use of these convenient lambda functions.

There will sometimes be cases where you'll have to pass a function a list arguments or keyword arguments. To define a function that takes a variable length of arguments or keyword arguments, you can use the ***args** or ****kwargs**, respectively. When calling the function, the variables args and kwargs become a list and a dictionary, which can be iterated through in the function's code. See the following example.

```
def print_args(*args):
    for arg in args:
        print(arg)

print_args('a', 'b', 'c')
```

```
In [14]: runfile('C:/Us
a
b
c
```

Note that we can pass a list of positional arguments into a function by using a leading asterisk.

```
In [15]: my_list = [1,2,3]

In [16]: print_args(*my_list)
1
2
3
```

Similarly, you can pass a dictionary of keyword arguments into a function with two asterisks.

```
def average_khan_final(**kwargs):
    total = 0
    for key, value in kwargs.items():
        total += value
    total /= len(kwargs)
    return total
```

```
In [20]: khan_final = {'Killian': 95, 'Noah': 90, 'Jake':55}

In [21]: average_khan_final(**khan_final)
Out[21]: 80.0
```

3.2 Classes

When using Python, we are not limited to the built-in data types. We can construct our own objects for storing and manipulating data. NumPy arrays, MatPlotLib figures, and PyTorch models are all **classes** with unique **attributes** and **methods** that make them useful. To make an object of your own, the syntax is simply a class statement followed by the class name, the class from which you want your object to inherit properties, a colon, and indented code.

```
>>> class Rectangle(object):
...     pass
>>> my_rectangle = Rectangle()
```

Here, I have told the interpreter that my class Rectangle should inherit from the built-in type object. Inheriting from object is not entirely necessary in Python 3, but for the sake of maintaining compatibility with old interpreters, I always include it. Notice that the rectangle I have created is very boring. I can spice up my rectangle by giving it some attributes with the __init__() method. __init__() is a special function that tells the Python interpreter to perform some procedures upon the instantiation of the object. We usually pass this function the argument *self*, along with some other data which we wish to be used in our object.

```
>>> class Rectangle(object):
...     def __init__(self, height, width):
...         self.height = height
...         self.width = width
```

When I created the class Rectangle this time, I declared that when an object of class Rectangle is created, a height and width should be provided, which will be stored as the rectangle's attributes. Note that whenever I write self.height = height, I am asserting that the object's attribute height is set to whatever was entered in the "height" positional argument of __init__(). The names of the attribute and the argument do not need to be the same.

Now that I have declared that height and width are essential to the creation of a Rectangle, if I try and make a Rectangle without any positional arguments, the interpreter throws an error.

```
>>> my_rectangle = Rectangle()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() takes exactly 3 arguments (1 given)
>>> my_rectangle = Rectangle(3, 4)
```

Now that I have created a variable my_rectangle, I can access its attributes with a dot.

```
>>> my_rectangle.height
3
>>> my_rectangle.width
4
```

This is a handy way for storing groups of related data, but the true benefit of custom classes comes from our ability to create custom methods. Remember that methods are just functions which are specific to a certain object, like flatten() and reshape() for NumPy arrays.

```
>>> import numpy as np
>>> A = np.array([[1,2],[3,4]])
>>> A.flatten()
array([1, 2, 3, 4])
>>> A.reshape((1,4))
array([[1, 2, 3, 4]])
```

We can create our own methods for the Rectangle class by writing a function within the indented block of code. When writing methods, we often want to use attributes stored during the object's initialization. For this purpose, we simply include *self*, as the first positional argument when we create the function. We can also use methods to amend attributes tied to our object. In the following example, I add the method compute_area() to the Rectangle class,

```
>>> class Rectangle(object):
...     def __init__(self, height, width):
...         self.height = height
...         self.width = width
...     def compute_area(self):
...         self.area = self.height*self.width
...
>>> my_rectangle = Rectangle(3,4)
>>> my_rectangle.compute_area()
>>> my_rectangle.area
12
```

Another major benefit of using classes is **inheritance**. When you construct an object, you can tell it to take attributes and methods from already existing classes. If you don't write an __init__() function for the new class, the parent class' initialization will be used. When doing this, you can simply build off of the parent class by adding new methods. For example, let's create a new class Square that inherits from Rectangle. Since we know that a square's diagonal is root 2 times its side length, we might add a method that computes the diagonal length in this way. Note that the initialization of Square is the same as Rectangle, and we still have access to all of Rectangle's wonderful methods (compute_area()).

```
>>> class Square(Rectangle):
...     def compute_diagonal_length(self):
...         self.diagonal = np.sqrt(2)*self.height #(or width)
...         print(self.diagonal)
...
>>> my_square = Square(4,4)
>>> my_square.compute_diagonal_length()
5.65685424949
>>> my_square.compute_area()
>>> my_square.area
16
```

If we want Square to be initialized in a different way than Rectangle is initialized, we can write an __init__() function for Square. This will overwrite the __init__() method used in Rectangle. Note that if we try to use the function compute_area() from Rectangle it will still expect the attributes height and width to exist.

```
>>> class Square(Rectangle):
...     def __init__(self, side):
...         self.height = side
...         self.width = side
...     def compute_diagonal_length(self):
```

```
...         self.diagonal = np.sqrt(2)*self.height #(or width)
...         print(self.diagonal)

>>> my_square = Square(4)
>>> my_square.compute_area()
>>> my_square.area
16
```

Alternative to rewriting the individual attributes from Rectangle's __init__() function, we can use the built-in function super(). super() allows us to access methods available from the parent class, including __init__(). If we want to initialize the Square exactly like a Rectangle with the caveat that width=height, we can use super() as follows.

```
>>> class Square(Rectangle):
...     def __init__(self, side):
...         super().__init__(side, side)
>>> my_square = Square(4)
```

You'll find that custom classes are very popular amongst Python users. They provide a useful integration of data storage and manipulation. Deep learning packages like PyTorch rely heavily on the user being able to create custom models which inherit methods for fitting ("training") from general model classes. In the next section, we'll use SciPy's interp1d feature, which is implemented as a class.

3.3 Radiation Units

This class is meant to be a mix of programming and physics. For this reason, I will introduce some physics concepts in class to help guide the class examples and the homework sets. When introducing a physics topic, there is no better place to begin than with units, which is the topic of this section.

The most important unit for therapeutic medical physics is the Gray (Gy), which is Joules/kilogram. This unit represents the amount of radiation energy absorbed per unit mass and is generally used as a proxy for biological damage. It is also the unit by which the radiation treatments are prescribed. Outside of radiation oncology, a doctor might write prescriptions for a patient to take a certain number of mg of medicine, a number of times per day. In radiation oncology, the only difference is that the doses are prescribed with Gy instead of mg.

There are, however, a couple of issues with this unit. First, the total dose delivered to a patient does not fully determine biological response. If the same total dose is delivered with fewer treatment sessions (fractions), it yields a higher biological response. For this reason, the concept of biological equivalent dose (BED) was developed. Regardless of fractionation, giving the same BED will yield the same biological response (at least in theory). The following equation shows how to calculate this quantity.

$$BED = n \cdot d\left(1 + \frac{d}{\alpha/\beta}\right) \qquad \text{Eq. 3.1}$$

The quantity n denotes the number of fractions (treatment sessions), d is the dose per fraction, and α/β is a tissue-specific quantity that models how sensitive that tissue is to fractionation changes. For the

same n and d, tissues with smaller α/β values have larger resulting values for BED. In general, normal tissues have α/β values around 3 Gy and tumor cells around 10 Gy.

The second issue with the Gy unit is about how we measure it. Although we perform many measurements in the clinic to make sure our machines are delivering the proper dose, we do not actually measure dose. This is due to the fact that radiation is incredibly efficient at killing cells and the actual amount of energy deposited in a patient is relatively small and difficult to measure.

Patients are often approximated as water, which for many cases is actually quite accurate. The number of Gy deposited in water can be measured by finding the increase in its temperature and then multiplying by the specific heat of water. A typical daily radiation dose is 2 Gy (J/kg). The specific heat for water is 4,187 J/(kg*K). Thus, the temperature increase of this irradiated tissue is only 0.48 mK. This method for measuring dose deposition is actually possible in laboratory environments, but the precision required to do this is too demanding for a typical radiation oncology clinic.

For this reason, we actually measure exposure, which is the amount of charge collected per unit mass of air and use calibration and correction factors to convert exposure to dose. This system of exposure measurement and conversion allows us to determine delivered dose with accuracy on the scale of 1%. The SI unit for exposure is C/kg and another common unit is the Roentgen (R), which is 258 μC/kg. Figure 1 shows a diagram of an ionization chamber, which is our tool for measuring exposure.

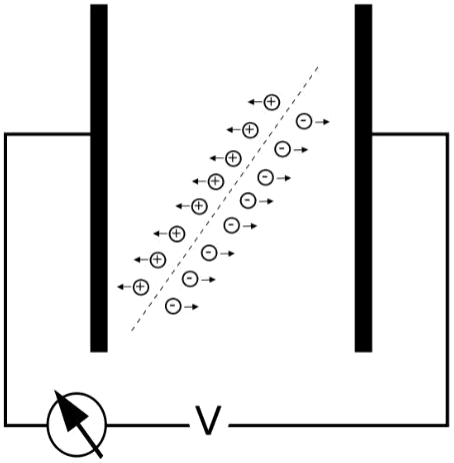

FIG. 1. An ionization chamber.

An ionization chamber is two electrodes, with an applied voltage, that enclose a volume of air. Radiation causes electrons to be ionized from the air, which are then collected by one of the electrodes and measured with an electrometer.

When we get a new chamber that we would like to use to measure radiation (via AAPM's TG-51 protocol), the first thing we do is send it to a calibration laboratory. They have a cobalt-60 source, for which they have measured the radiation dose rate at a specific position relative to it (using previously mentioned methods). They put our chamber at this position and measure the current from it. The ratio of these numbers yields a chamber-specific, exposure-to-dose calibration factor called $N_{D,w}^{co\text{-}60}$. However, most treatments are not delivered with cobalt-60 sources, but with x-rays from higher energy linear accelerators (linacs). The difference in x-ray energy changes the ratio of dose to exposure. Thus,

another quantity called k_Q was introduced to account for this. The measured dose (D) can then be calculated with the following equation.

$$D = k_Q \cdot N_{D,w}^{60Co} \cdot M \qquad \text{Eq. 3.2}$$

The quantity M denotes the measured charge. The determination of the k_Q factor is fairly straight forward. As mentioned, k_Q accounts for differences in radiation energy. For the photon radiation considered, the higher the energy, the deeper it penetrates into a material. To determine k_Q you make something like a transmission measurement in reference conditions at a depth of 10 cm. Then, for your chamber, you can find data tables that give you the k_Q factor as a function of this transmission. As an example, for a PTW 31003 cylindrical ionization chamber, k_Q is 1, 0.996, 0.992, 0.984, 0.967, and 0.946 for the transmission values of 58, 63, 66, 71, 81, and 93%, respectively. Note, cobalt-60 has the transmission value of 58%, which is why k_Q is 1 at that value. If you have a transmission value between the tabulated values, you need to interpolate, which brings us to doing this in Python. SciPy has efficient routines for interpolating in any number of dimensions. Here, we only require 1-D interpolation, which can be performed with the scipy.interpolate.interp1d class.

scipy.interpolate.interp1d

class **scipy.interpolate.interp1d**(*x, y, kind='linear', axis=-1, copy=True, bounds_error=None, fill_value=nan, assume_sorted=False*)

Interpolate a 1-D function. [source]

x and *y* are arrays of values used to approximate some function f: `y = f(x)`. This class returns a function whose call method uses interpolation to find the value of new points.

Note that calling interp1d with NaNs present in input values results in undefined behaviour.

Parameters:

x : *(N,) array_like*
A 1-D array of real values.

y : *(...,N,...) array_like*
A N-D array of real values. The length of *y* along the interpolation axis must be equal to the length of *x*.

kind : *str or int, optional*
Specifies the kind of interpolation as a string ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'previous', 'next', where 'zero', 'slinear', 'quadratic' and 'cubic' refer to a spline interpolation of zeroth, first, second or third order; 'previous' and 'next' simply return the previous or next value of the point) or as an integer specifying the order of the spline interpolator to use. Default is 'linear'.

axis : *int, optional*
Specifies the axis of *y* along which to interpolate. Interpolation defaults to the last axis of *y*.

copy : *bool, optional*
If True, the class makes internal copies of x and y. If False, references to *x* and *y* are used. The default is to copy.

bounds_error : *bool, optional*
If True, a ValueError is raised any time interpolation is attempted on a value outside of the range of x (where extrapolation is necessary). If False, out of bounds values are assigned fill_value. By default, an error is raised unless `fill_value="extrapolate"`.

fill_value : *array-like or (array-like, array_like) or "extrapolate", optional*

- if a ndarray (or float), this value will be used to fill in for requested points outside of the data range. If not provided, then the default is NaN. The array-like must broadcast properly to the dimensions of the non-interpolation axes.
- If a two-element tuple, then the first element is used as a fill value for `x_new < x[0]` and the second element is used for `x_new > x[-1]`. Anything that is not a 2-element tuple (e.g., list or ndarray, regardless of shape) is taken to be a single array-like argument meant to be used for both bounds as `below, above = fill_value, fill_value`.
  New in version 0.17.0.
- If "extrapolate", then points outside the data range will be extrapolated.
  New in version 0.17.0.

assume_sorted : *bool, optional*
If False, values of *x* can be in any order and they are sorted first. If True, *x* has to be an array of monotonically increasing values.

interp1d requires two positional arguments: your x and y data. From these data it creates an interpolant f(x) which passes through your data points. The form of f(x) is specified by the keyword argument "kind". When the class interp1 was written, they programmer used a special method, like __init__(), named __call__(). The __call__() method assigns a default function and allows you to use a class syntactically like a function. See the following example.

```
>>> from scipy.interpolate import interp1d
>>> import numpy as np

>>> x = np.random.random((10))
>>> y = np.random.random((10))
>>> interpolator = interp1d(x,y)
>>> interpolator(0.5)
array(0.22875802781448823)
```

See that although interpolator was written like an object, we can call with it like a function because of the __call__() method. The default interpolation employed in this function is linear interpolation. This can be changed by passing the function a string for your preferred method (linear, nearest neighbor, cubic …). For the k_Q data, the method used will not make that big of a difference, but it will for other cases. See the following demonstration (ignoring the fact that we have not gone over plotting yet) and Fig. 2.

```
>>> theta = np.linspace(0,2*np.pi, 101)
>>> downsampled = theta[::10]
>>> y = np.cos(downsampled)
>>> y_linear_interp = interp1d(downsampled, y, kind='linear')(theta)
>>> y_nearest_interp = interp1d(downsampled, y, kind='nearest')(theta)
>>> y_cubic_interp = interp1d(downsampled, y, kind='cubic')(theta)
>>> import matplotlib.pyplot as plt
>>> plt.plot(theta, np.cos(theta), 'k', label='cos(x)')
>>> plt.plot(theta, y_linear_interp, 'b--', label='linear')
>>> plt.plot(theta, y_nearest_interp, 'c--', label='nearest')
>>> plt.plot(theta, y_cubic_interp, 'g--', label='cubic')
>>> plt.legend()
>>> plt.show()
```

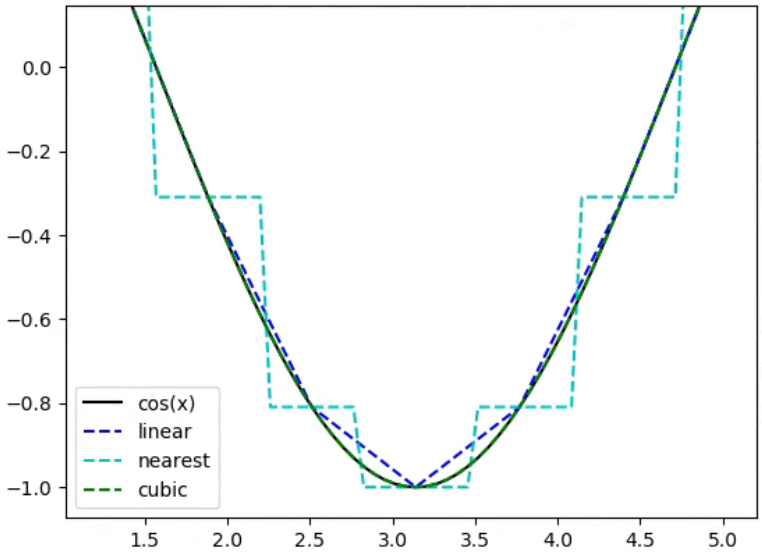

FIG. 2. The effect of different interpolation methods.

Notice that in the example, I used np.linspace() in 101 steps, so that the down sampled array would contain 2*pi as it's last element. If I were to use 100 steps, we'd be trying to predict what's happening from 5.71 to 6.28 based on the single point at 5.71.

```
>>> theta = np.linspace(0, 2*np.pi, 100)
>>> downsampled = theta[::10]
>>> y = np.cos(downsampled)
>>> print(downsampled[-1], theta[-1])
5.711986642890533 6.283185307179586
>>> interp1d(downsampled, y)(theta)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: A value in x_new is above the interpolation range.
```

We can combat this error in a few ways. We can turn off the error message by setting the keyword argument bounds_error=False. This will write NaNs outside of the interpolant's domain.

```
>>> linear_interp = interp1d(downsampled, y, bounds_error=False)(theta)
>>> linear_interp[-1]
nan
```

You can also set the tell the interpolator to extrapolate, or fill entries with a specified value.

```
>>> linear_interp = interp1d(downsampled, y, bounds_error=False, fill_value='extrapolate')(theta)
>>> linear_interp[-1]
1.2771841129465593

>>> linear_interp = interp1d(downsampled, y, bounds_error=False, fill_value=0)(theta)
>>> linear_interp[-1]
0.0
```

Note, the first two arguments of interp1d must be linear arrays, but the interpolation values do not need to be scalars or arrays. The function will return you y interpolated values that match the dimensions of your requested x values.

```
>>> interpolator = interp1d(downsampled, y, bounds_error=False)
>>> A =  np.array([theta, theta[::-1]])
>>> linear_interp = interpolator(A)
>>> linear_interp.shape
(2, 100)
```

This class and many others are well documented on the SciPy website. Whenever you are looking for some scientific function that you expect to exist, a Google search will rarely fail you. Libraries like SciPy are especially detailed in their documentation and will provide thorough descriptions of a function or class' arguments. There will often be example usage as well. For less popular packages, documentation varies, but you can always examine a package's code yourself.

Beyond accounting for energy dependence of your ionization chamber, you also need to correct for other imperfections in your charge collection system. One of these is the temperature and pressure correction. Many ionization chambers are open, meaning that they can exchange air with their environment. The calibration factors we receive for our chambers are meant for standard temperature and pressure (22 C and 101.33 kPa). If, for example, the air pressure is higher than that of the standard, our chamber will contain more air molecules and will therefore also generate more charge at the same dose. For this reason, charge measurements from open-air chambers must be multiplied by a correction factor $P_{T,P}$. The following is an example function to calculate this.

```
>>> def ptp(T,P):
...     ptp = (T + 273.15)/295.15*101.33/P
...     return ptp
...
>>> ptp(25, 100)
1.0235995087243774
```

3.4 Radiation Film Measurements

So far, we've gone over how radiation dose can be determined from ionization chamber exposure measurements. This really is the gold standard in our field for determining dose, but there are also many instances where we cannot use an ionization chamber. For example, we might need to make dose measurements with a spatial resolution smaller than our ionization chambers. In these cases, we need to switch to other dosimeters, which require different methods to connect their signal to dose. The homework set involves film dosimetry so I will provide a brief introduction for these measurements.

In the most basic terms, film changes its opacity when irradiated. We can expose film to known doses, measure the resulting opacity, and create an interpolation table to find an unknown dose from an opacity measurement. We never send our film to a calibration laboratory. Instead, we derive our own calibration for the film. However, we rely on our calibrated ion chambers to be able to expose our film to known doses.

To properly create film calibration data, it is important to understand the physics of radiation-induced opacity changes. For this reason, I will present a simple physics model for this. There are different types of film, with different physical mechanisms responsible for the changing in opacity. For example, with radiographic film, radiation along with the film development process leaves silver grains on the film which absorbs light. For radiochromic film, radiation breaks polymers in the film, which affects its transmission of light. The exact mechanism for the opacity change is, however, not critical for our purposes here.

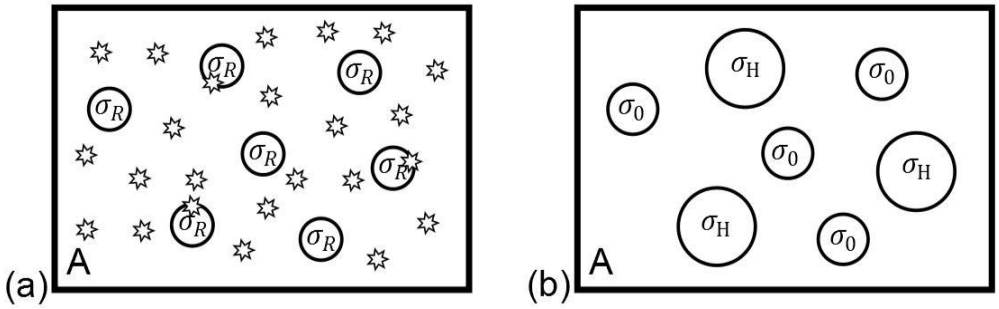

FIG. 3. (a) Radiation incident upon a film of size A and absorbers with a cross section of σ_R to radiation interaction. (b) This same film after irradiation. Absorbers that have been hit have their optical absorption cross section increase from σ_0 to σ_H.

Let my film have a total area of A, have a number of absorbers N_A, and let there be a number of N high-energy radiation particles incident upon it (see Fig. 3a). Additionally, let each of the absorbers have an interaction cross-section for the incident radiation. The following equation then represents the probability that an absorber does not interact with the incident radiation, P.

$$P = (1 - {}^{\sigma_R}/_{A})^N = 1 - N \cdot \left(\frac{\sigma_R}{A}\right) + \frac{N(N-1)}{2!} \cdot \left(\frac{\sigma_R}{A}\right)^2 - \frac{N(N-1)(N-2)}{3!} \cdot \left(\frac{\sigma_R}{A}\right)^3 + \cdots \qquad \text{Eq. 3.3}$$

For N>>1 and $(\sigma R/A)N$<< 1, this expression becomes close to the expansion for an exponential

$$P = \exp\left(-\frac{N \cdot \sigma_R}{A}\right) \qquad \text{Eq. 3.4}$$

function and can be approximated by N/A is the density of radiation incident on the film, which is also proportional to dose, D. The interaction cross section is also a constant so Eq. 3.4 can be rewritten as the following.

$$P = \exp(-\gamma \cdot D) \qquad \text{Eq. 3.5}$$

where γ is simply a constant. The opacity for our film is read with optical light though, so we must introduce different cross sections for these absorbers to this wavelength. Let σ_0 and σ_H represent the cross sections before and after a radiation interaction. The probability that an optical photon is transmitted through the film, T, can then be expressed as the following equation (after using the above mentioned exponential approximation).

$$T = \exp\left[-\frac{N_A \cdot \sigma_H}{A}(1 - P)\right] \cdot \exp\left[-\frac{N_A \cdot \sigma_0}{A} \mathrm{P}\right] \qquad \text{Eq. 3.6}$$

Let me rewrite this formula for the log of the transmission, after approximating the P exponential by its first order Taylor series expansion (1 − γ· D).

$$\log(T) \approx -\frac{N_A \cdot \sigma_0}{A} - \frac{N_A \cdot (\sigma_H - \sigma_0) \cdot \gamma \cdot D}{A} \quad \text{Eq. 3.7}$$

N_A/A, σ_H, σ_0, and are all constants for a fixed type of film and radiation and optical density (OD) is related to the natural logarithm of the transmission by a constant; therefore, Eq. 3.7 can be rewritten as the following.

$$OD \approx C_1 + C_2 \cdot D \quad \text{Eq. 3.8}$$

C_1 and C_2 are constants. The Taylor series expansion approximation means that Eq. 3.8 is not perfect representation, but in general you should expect the optical density to change linearly with deposited dose. This is especially true for small dose ranges.