

2.2: Improving the Program

Let's return to the program for calculating the electric potential, which we discussed in the [previous part of the tutorial](#), and improve it further. These improvements will show off some more advanced features of Python and Scipy which are good to know about.

We'll also make one substantive change in the physics: instead of treating the particles as point-like objects, we'll assume that they have a finite radius R , with all the charge concentrated at the surface. Hence, the potential produced by a particle of total charge q_0 and position x_0 will have the form

$$\phi(X) = \begin{cases} \frac{q_0}{|X - x_0|}, & \text{if } |X - x_0| \geq R \\ \frac{q_0}{R} & \text{if } |X - x_0| < R. \end{cases} \quad (2.2.1)$$

Open a few Python file, and call it `potentials2.py`. Write the following into it:

```
from scipy import *
import matplotlib.pyplot as plt

## Return the potential at measurement points X, due to particles
## at positions xc and charges qc. xc, qc, and X must be 1D arrays,
## with xc and qc of equal length. The return value is an array
## of the same length as X, containing the potentials at each X point.
def potential(xc, qc, X, radius=5e-2):
    assert xc.ndim == qc.ndim == X.ndim == 1
    assert len(xc) == len(qc)
    assert radius > 0.

    phi = zeros(len(X))
    for j in range(len(xc)):
        dphi = qc[j] / abs(X - xc[j])
        dphi[abs(X - xc[j]) < radius] = qc[j] / radius
        phi += dphi
    return phi

## Plot the potential produced by N particles of charge 1, distributed
## randomly between x=-1 and x=1.
def potential_demo(N=20):
    X = linspace(-2.0, 2.0, 200)
    qc = ones(N)

    from scipy.stats import uniform
    xc = uniform(loc=-1.0, scale=2.0).rvs(size=N)

    phi = potential(xc, qc, X)

    fig_label = 'Potential from ' + str(N) + ' particles'
    plt.plot(X, phi, 'ro', label=fig_label)
    plt.ylim(0, 1.25 * max(phi))
```

```
plt.legend()
plt.xlabel('r')
plt.ylabel('phi')
plt.show()

potential_demo(100)
```

We will now go through the changes that we've made in the program.

2.2.1 Optional Function Parameters

As before, we define a function named `potential`, whose job is to compute the electric potential produced by a collection of particles in 1D. However, you might notice a change in the function header:

```
def potential(xc, qc, X, radius=5e-2):
```

We have added an **optional parameter**, specified as `radius=5e-2`. An optional parameter is a parameter which has a default value. In this case, the optional parameter is named `radius`, and its default value is `5e-2` (which means 5×10^{-2} ; you can also write it as `0.05`, which is equivalent). If you call the function omitting the last input, the value will be assumed to be 0.05. If you supply an explicit value for the last input, that overrides the default.

If a function call omits a *non*-optional parameter (which as `xc`), that is a fatal error: Python will stop the program with an error message.

2.2.2 Assert Statements

In the function body, we have added the following three lines:

```
assert xc.ndim == qc.ndim == X.ndim == 1
assert len(xc) == len(qc)
assert radius > 0.
```

The `assert` statement is a special Python statement which checks for the truth value of the following expression; if that expression is false, the program will stop and an informative error message will be displayed.

Here, we use the `assert` statements to check that

- `xc`, `qc`, and `X` are all 1D arrays (note: the `==` Python operator checks for numerical equality)
- `xc` has the same length as `qc`
- `radius` has a positive value (note: `0.` is Python short-hand for the number `0.0`).

Similar to writing comments, adding `assert` statements to your program is good programming practice. They are used to verify that the assumptions made by the rest of the code (e.g., that the `xc` and `qc` arrays have equal length) are indeed met. This ensures that if we make a programming mistake (e.g., supplying arrays of incompatible size as inputs), the problem will surface as soon as possible, rather than letting the program continue to run and causing a more subtle error later on.

2.2.3 Advanced Slicing

Inside the for loop, we have changed the way the potential is computed:

```
for j in range(len(xc)):
    dphi = qc[j] / abs(X - xc[j])
    dphi[abs(X - xc[j]) < radius] = qc[j] / radius
    phi += dphi
```

As discussed above, we are now considering particles of finite size rather than point particles, so the potential is constant at distances below the particle radius. This is accomplished using an advanced [array slicing](#) technique.

For each particle j , the potential is computed in three steps:

- Calculate the potential using the regular formula $q_j/|X - x_j|$, and save those values into an array, one for each value of X .
- Find the indices of that array which correspond to values with $|X - x_j| < R$, and overwrite those elements with the constant value q_j/R . To find the relevant indices, we make use of the following slicing feature: if a comparison expression is supplied as an index, that refers to those indices for which the comparison is true. In this case, the comparison expression is `abs(X-xc[j]) < radius`, which refers to the indices of X which are below the minimum radius. These indices are the ones in the `dphi` array that we want to overwrite.
- Add the result to the total potential.

Demo Function

Finally, we have a "demo" or ("demonstration") function to make the appropriate plots:

```
## Plot the potential produced by N particles of charge 1, distributed
## randomly between x=-1 and x=1.
def potential_demo(N=20):
    X = linspace(-2.0, 2.0, 200)
    qc = ones(N)

    from scipy.stats import uniform
    xc = uniform(loc=-1.0, scale=2.0).rvs(size=N)

    phi = potential(xc, qc, X)

    fig_label = 'Potential from ' + str(N) + ' particles'
    plt.plot(X, phi, 'ro', label=fig_label)
    plt.ylim(0, 1.25 * max(phi))
    plt.legend()
    plt.xlabel('r')
    plt.ylabel('phi')
    plt.show()

potential_demo(100)
```

Whereas our previous program put the plotting stuff at "top level", here we encapsulate the plotting code in a `potential_demo()` function. This function is called by the top-level statement `potential_demo(100)`, which occurs at the very end of the program.

It is useful to do this because if, in the future, you want the program demonstrate something else (e.g. producing a different kind of plot), it won't be necessary to delete the `potential_demo` function (and risk having to rewrite it if you change your mind). Instead, you can write another demo function, and revise that single top-level statement to call the new demo function instead.

The `potential_demo` function provides another example of using optional parameters. It accepts a parameter `N=20`, specifying the number of particles to place. When the program runs, however, the function is invoked through the top-level statement `potential_demo(100)`, i.e. with an actual input of 100 which overrides the default value of 20. If the top-level statement had instead been `potential_demo()`, then the default value of 20 would be used.

Sampling Random Variables

The demo function generates `N` particles with random positions. This is done using this code:

```
from scipy.stats import uniform
xc = uniform(loc=-1.0, scale=2.0).rvs(size=N)
```

The first line imports a function named `uniform` from the `scipy.stats` module, which is a module that implements random number distributions. As this example shows, import statements don't have to be top-level statements. In some cases, we might choose to perform an import only when a particular function runs (usually, this is done if that function is the only one in the program relying on that module).

The `uniform` function returns an object which corresponds to a particular uniform distribution. One of the methods of this object, named `rvs`, generates an array of random numbers drawn from that distribution.

Plotting

After computing the total potential using a call to the `potential` function, we plot it:

```
fig_label = 'Potential from ' + str(N) + ' particles'
plt.plot(X, phi, 'ro', label=fig_label)
```

To begin with, concentrate on the second line. This is a slightly more sophisticated use of Matplotlib's `plot` function than what we had the last time.

The first two arguments, as before, are the x and y coordinates for the plot. The next argument, `'ro'`, specifies that we want to plot using red circles, rather than using lines with the default color.

The fourth argument, `label=fig_label`, specifies some text with which to label the plotted curve. It is often useful to associate each curve in a figure with a label (though, in this case, the figure contains only one curve).

This way of specifying a function input, which has the form `FOO=BAR`, is something we have not previously seen. It relies on a feature known as [keyword arguments](#). In this case, `label` is the keyword (the name of the parameter we're specifying), and `fig_label` is the value (which is a string object; we'll discuss this below). Keyword arguments allow the caller of a function to specify optional parameters in any order. For example,

```
plt.plot(X, phi, 'ro', label=fig_label, linewidth=2)
```

is equivalent to

```
plt.plot(X, phi, 'ro', linewidth=2, label=fig_label)
```

The full list of keywords for the `plot` function is given in its [documentation](#).

Constructing a Label String

Next, we turn to the line

```
fig_label = 'Potential from ' + str(N) + ' particles'
```

which creates a Python object named `fig_label`, which is used for labeling the curve. This kind of object is called a [character string](#) (or just **string** for short).

On the right-hand side of the above statement, we build the contents of the string from several pieces. This is done in order to get a different string for each value of `N`. The `+` operator "concatenates" strings, joining the strings to its left and right into a longer string. In this case, the string `fig_label` consists of the following shorter strings, concatenated together:

- A string containing the text `'Potential from '`.
- A string containing the numerical value of `N`, in text form. This is computed using the `str` function, which converts numbers into their corresponding string representations.
- A string containing the text `' particles'`.

The rest of the `potential_demo` function is relatively self-explanatory. The `ylim` function specifies the lower and upper limits of the plot's y -axis (there is a similar `xlim` function, which we didn't use). The `plt.legend()` statement causes the curve label to be shown in a legend included in the plot. Finally, the `xlabel` and `ylabel` functions add string labels to the x and y axes.

Running the Program

Now save your `potential2.py` program, and run it. (Reminder: you can do this by typing `python -i potential2.py` from the command line on GNU/Linux, or F5 in the Windows GUI, or `import potential2` from the Python command line). The resulting figure looks like this:

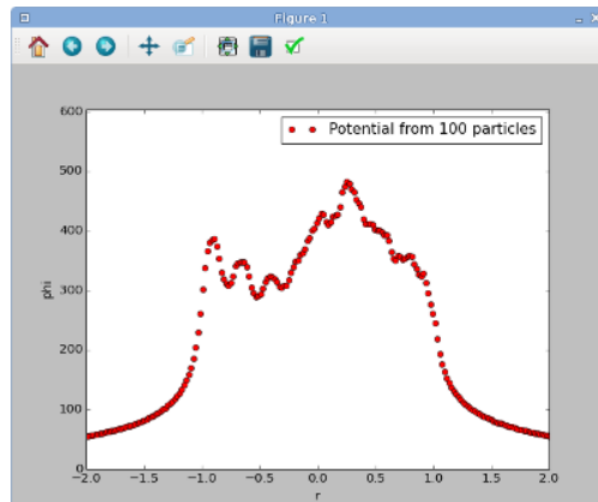


Figure 2.2.1

This page titled [2.2: Improving the Program](#) is shared under a [CC BY-SA 4.0](#) license and was authored, remixed, and/or curated by [Y. D. Chong](#) via [source content](#) that was edited to the style and standards of the LibreTexts platform.