lab1

August 30, 2019

1 EE 120 Lab 1: Introduction to Python for Signals and Systems

v1 - Spring 2019: Dominic Carrano, Sukrit Arora, and Babak Ayazifar
v2 - Fall 2019: Dominic Carrano

2 iPython "Labs"?

In the Spring 2019 semester, we added a new component to EE 120, iPython notebook labs. Essentially, they're just homework assignments though.

To better prepare you for future endeavors in these areas, we want to give you exposure to real world applications of the material via Python (specifically, the Python scientific computing stack of numpy, scipy, and matplotlib all together in iPython notebooks), a popular tool in both industry and academia.

The labs showcase algorithms and topics that are typically not directly covered in lecture, but that are fully comprehensible using the knowledge you've gained from class. Typically, we provide a "Background" section (and/or information within the actual questions) at the start of the lab explaining the topic in sufficient detail for you to complete the lab. In addition, there will typically be a "References" section at the end containing relevant links for exploring the topic further, as well as links we used in developing the labs, although these aren't required reading by any means.

Generally, the format of any given lab will be: - Background - Questions - References

Occasionally, questions will be marked as optional, in which case the question is not graded. Doing them will *not* earn you extra credit; they just serve as extra practice and examples.

We have invested much time and effort into making these labs interesting and application-driven, and hope that you find them both informative and fun. We welcome and encourage any feedback you have on them - it's an invaluable source of data for us. We'll be releasing Google Forms after each submission deadline where you can provide us with anonymous feedback.

3 Submitting The Notebooks

In terms of what we expect from you to do in completing the labs: every place you're required to answer a question, whether it be in the form of writing code or interpreting plots/results that you generate, it will be marked with "TODO".

When done, go to the notebook menu (under the jupyter logo) and click `File -> Download as -> PDF via LaTeX (.pdf)`. This is what you'll submit to Gradescope. If this gives you an error message, you can instead try `File -> Print Preview` and download that for submission.

4 Q1: Intro to the Python Scientific Computing Stack [OPTIONAL]

For this problem, you're not required to write any code. Simply run all the cells and acquaint yourself with the functions and features that they showcase.

This question is optional, as many of you are likely familiar with the Python programming language from CS 61A and the basics of its scientific computing capabilities from EE 16AB. Nevertheless, if you'd like a refresher, feel free to run through the code cells in this question and play around with the features of the language and libraries to get (re)acquainted.

This question was modified and adapted from Berkeley Python Bootcamp 2013, Python for Signal Processing, EE 123, and EECS 126 iPython Notebook Tutorials.

4.0.1 Running iPython Notebook Cells

The ipython notebook is divided into cells. Each cell can contain texts, codes or html scripts. Running a non-code cell simply advances to the next cell. To run a code cell using Shift-Enter or pressing the play button in the toolbar above:

[1]:
```
1+2
```

[1]:
```
3
```

4.0.2 Interrupting the kernel

For debugging, often we would like to interupt the current running process. This can be done by pressing the stop button.

When a processing is running, the circle on the right upper corner is filled. When idle, the circle is empty.

[2]:
```
import time

while(1):
    print("error")
    time.sleep(1)
```

```
error
error
error
error
```

```
---------------------------------------------------------------------------

KeyboardInterrupt                         Traceback (most recent call last)

<ipython-input-2-d074f6fb2aed> in <module>
  3 while(1):
  4     print("error")
----> 5     time.sleep(1)


KeyboardInterrupt:
```

4.0.3 Restarting the kernels

Interrupting sometimes does not work. You can reset the state by restarting the kernel. This is done by clicking Kernel/Restart or the Refresh button in the toolbar above.

4.0.4 Saving the notebook

To save your notebook, either select `"File->Save and Checkpoint"` or hit `Command-s` for Mac and `Ctrl-s` for Windows

4.0.5 Undoing

To undo changes in each cell, hit `Command-z` for Mac and `Ctrl-z` for Windows. To undo `Delete Cell`, select `Edit->Undo Delete Cell`.

4.0.6 Tab Completion

One useful feature of iPython is tab completion

[3]:
```
one_plus_one = 1+1

# type `one_` then hit TAB will auto-complete the variable
print(one_plus_one)
```

```
2
```

4.0.7 Help Menu for Functions

Another useful feature is the help command. Type any function followed by `?` returns a help window. Hit the `x` button to close it.

[4]:
```
abs?
```

4.0.8 Other iPython Notebook navigation tips

- To add a new cell, either select `"Insert->Insert New Cell Below"` or click the white plus button
- You can change the cell mode from code to text in the pulldown menu. I use `Markdown` for text
- You can change the texts in the `Markdown` cells by double-clicking them.
- `Help->Keyboard Shortcuts` has a list of keyboard shortcuts

4.0.9 Libraries

These are the libraries that we will be using in this class:

Numpy

NumPy is the fundamental package for scientific computing with Python.

Scipy

The SciPy library is a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics and much more.

matplotlib

matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

About Each numpy and matplotlib are both very systematic, in the sense that you're usually not calling one-off functions from them, but rather using them throughout your code, almost like a sort of sub-programming language inside Python. By the end of this semester, you'll be fairly well-seasoned in both. Scipy, on the other hand, is more of a collection of various useful functions that you'll look up when you need to do something really specific that numpy doesn't have, like digital filter design (see EE 123 for more) or some more complicated algorithms like k-means clustering (as you saw in EE 16B).

4.0.10 Importing

To import a specific library `x`, simply type `import x`.

To access the library function `f`, type `x.f`.

If you want to change the library name to `y`, type `import x as y`.

[1]:
```
# CONVENTION: "import numpy as np" when importing numpy, it lets us use "np" as␣
↪shorthand!
```

```
# You could just as easily do "import numpy as derp", but then you'd have all␣
 ↪this weird code
# like "derp.convolve(x, h, mode='full')" - the standard convention is to use␣
 ↪"np" for numpy
import numpy as np
np.ones((3,1))
```

[1]:
```
array([[1.],
       [1.],
       [1.]])
```

4.1 Data Types

4.1.1 Floats and Integers

Unlike MATLAB, there is a difference between the `int` and `float` types in Python 2. Mainly, integer division returns the floor in Python 2. However, in Python 3 there is no floor, but it is always good to check this when debugging!

[12]:
```
1 / 4
```

[12]:
```
0.25
```

[13]:
```
1 / 4.0
```

[13]:
```
0.25
```

4.1.2 Strings

Unlike MATLAB/C, double quotes ("I am double quoted!") and single quotes ('I, however, am single quoted') are the same thing. Both represent strings. `'+'` concatenates strings

[14]:
```
"EE " + '120' # we can mix and match single and double quotes and Python won't␣
 ↪care
```

[14]:
```
'EE 120'
```

4.1.3 Lists

A list is a mutable array of data. That is we can change it after we create it. They can be created using square brackets []

Important functions: - `'+'` appends lists. - `len(x)` to get length

[15]:
```
x = ["EE"] + [1, 2, 0]
print(x)
```

```
['EE', 1, 2, 0]
```

[16]:
```
print(len(x))
```

```
4
```

4.1.4 Tuples

A tuple is an immutable list. They can be created using round brackets (). They are usually used as inputs and outputs to functions.

[17]:
```
t = ("E", "E") + (1, 2, 0)
print(t)
```

```
('E', 'E', 1, 2, 0)
```

[18]:
```
# cannot do assignment to a tuple after creation - it's immutable
t[4] = 3 # will error

# note: errors in ipython notebook appear "inline", i.e. inside the notebook␣
↪itself
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-18-4793ada63bf1> in <module>
      1 # cannot do assignment to a tuple after creation - it's immutable
----> 2 t[4] = 3 # will error
      3
      4 # note: errors in ipython notebook appear "inline", i.e. inside the␣
↪notebook itself


TypeError: 'tuple' object does not support item assignment
```

4.1.5 Functions

Functions take in a set of arguments, and return (possibly multiple) values. Here's an example of a function that checks if a number is divisible by 3, using Python's modulus operator (`a % b` returns the remainder of dividing `a` by `b`; if the remainder is 0, then `a` is divisible by `b`):

[19]:
```
def is_divisible_by_3(x):
    return x % 3 == 0

print("6 is divisible by 3? {0}".format(is_divisible_by_3(6)))
print("8 is divisible by 3? {0}".format(is_divisible_by_3(8)))
```

```
6 is divisible by 3? True
8 is divisible by 3? False
```

You can also return multiple values from a function at once; just separate them with commas in the return statement. Similarly, to assign them to values after the function call, just separate the variables you want to assign the return values to by commas. Here's a dummy example that takes in two numbers, and returns their sum and their difference:

[20]:
```
def sum_and_diff(x, y):
    return x+y, x-y

a, b = sum_and_diff(3, 5)
print(a) # 3 + 5 = 8
print(b) # 3 - 5 = -2
```

```
8
-2
```

4.1.6 Numpy Array

The numpy array, aka an "ndarray", is like a list with multidimensional support and more functions. This will be the primary data structure in our class.

Arithmetic operations on NumPy arrays correspond to elementwise operations.

Important NumPy Array functions:

- `.shape` returns the dimensions of the array.
- `.ndim` returns the number of dimensions.
- `.size` returns the number of entries in the array.
- `len()` returns the first dimension.

To use functions in NumPy, we have to import NumPy to our workspace. This is done by the command `import numpy`. By convention, we rename `numpy` as `np` for convenience.

4.1.7 Creating a Numpy Array

[21]:
```
x = np.array([[1, 2], [3 , 4]])
print(x)
```

```
[[1 2]
 [3 4]]
```

4.1.8 Getting the shape of a Numpy Array

[22]:
```
x.shape # returns the dimensions of the numpy array
```

[22]:
```
(2, 2)
```

[23]:
```
np.shape(x) # equivalent to x.shape
```

[23]:
```
(2, 2)
```

4.1.9 Elementwise operations

One major advantage of using numpy arrays is that arithmetic operations on numpy arrays correspond to elementwise operations. This makes numpy amenable to vectorized implementations of algorithms, a common technique used for speeding up computer simulations than can be parallelized.

[24]:
```
print(x)
print()
print(x + 2) # numpy is smart and assumes you want this to be done to all␣
 ↪elements!
```

```
[[1 2]
 [3 4]]

[[3 4]
 [5 6]]
```

4.1.10 Matrix multiplication

You can use `np.matrix` with the multiplication operator or `np.dot` to do matrix multiplication.

[25]:
```
print(np.matrix(x) * np.matrix(x))
print() # newline for formatting

# Or
print(np.dot(x,x))
```

```
[[ 7 10]
 [15 22]]

[[ 7 10]
 [15 22]]
```

4.1.11 Slicing numpy arrays

Numpy uses pass-by-reference semantics so it creates views into the existing array, without implicit copying. This is particularly helpful with very large arrays because copying can be slow.

[26]:
```
x = np.array([1,2,3,4,5,6])
print(x)
```

```
[1 2 3 4 5 6]
```

We slice an array from a to b-1 with `[a:b]`.

[27]:
```
y = x[0:4]
print(y)
```

```
[1 2 3 4]
```

Because slicing does not copy the array, changing `y` changes `x`.

[28]:
```
y[0] = 7
print(x)
print(y)
```

```
[7 2 3 4 5 6]
[7 2 3 4]
```

To actually copy x, we should use `.copy()`.

[29]:
```
x = np.array([1,2,3,4,5,6])
y = x.copy()
y[0] = 7
print(x)
print(y)
```

```
[1 2 3 4 5 6]
[7 2 3 4 5 6]
```

4.1.12 Useful Numpy function: arange

We use `arange` to create integer sequences in numpy arrays. It's exactly like the normal range function in Python, except that it automatically returns the result as a numpy array, rather than the plain vanilla Python list.

`arange(0,N)` creates an array listing every integer from 0 to N-1.

`arange(0,N,m)` creates an array listing every `m` th integer from 0 to N-1 .

[30]:
```
print(np.arange(-5,5)) # every integer from -5 ... 4
```

```
[-5 -4 -3 -2 -1  0  1  2  3  4]
```

[31]:
```
print(np.arange(0,5,2)) # every other integer from 0 ... 4
```

```
[0 2 4]
```

4.2 Plotting

In this class, we will use `matplotlib.pyplot` to plot signals and images.

By convention, we import `matplotlib.pyplot` as `plt`.

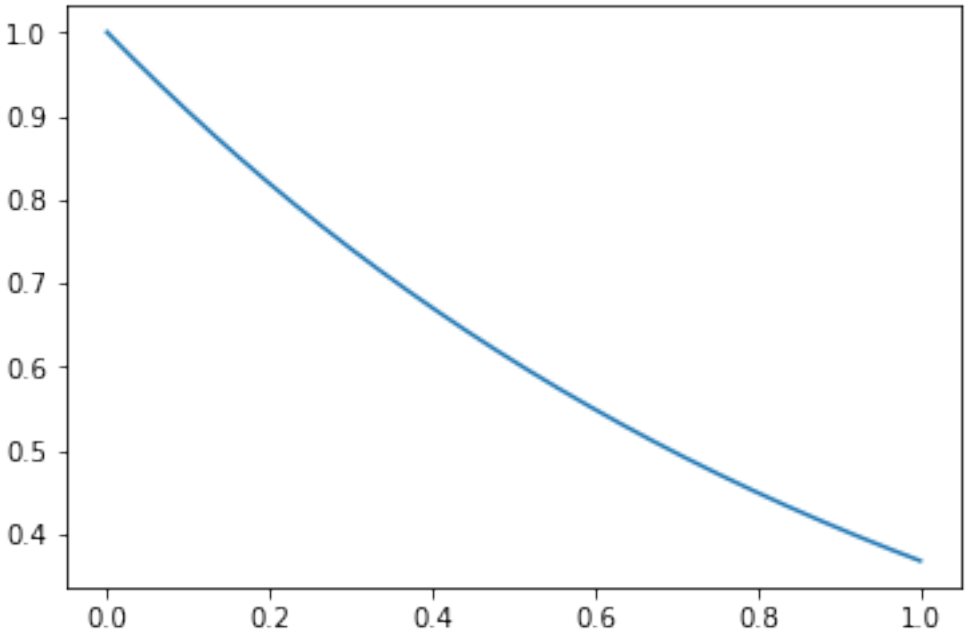
To display the plots inside the browser, we use the command `%matplotlib inline` - do not forget this line whenever you start a new notebook. We'll always include it for you, but in case your plots aren't showing up in the notebook when you're playing around on your own, it's probably because you forgot this. If you don't include it, Python will default to displaying it in another window on your computer, which normally is fine, but here we need the plots in the notebook so they show up in your submission PDF.

[2]:
```
import matplotlib.pyplot as plt # by convention, we import matplotlib.pyplot as␣
↪plt

# plot in browser instead of opening new windows
%matplotlib inline
```

[3]:
```
# Generate signals
x = np.arange(0, 1, 0.001)
y1 = np.exp(-x)                                # decaying exponential
y2 = np.sin(2 * np.pi * 10.0 * x)/4.0 + 0.5  # 10 Hz sine wave
```

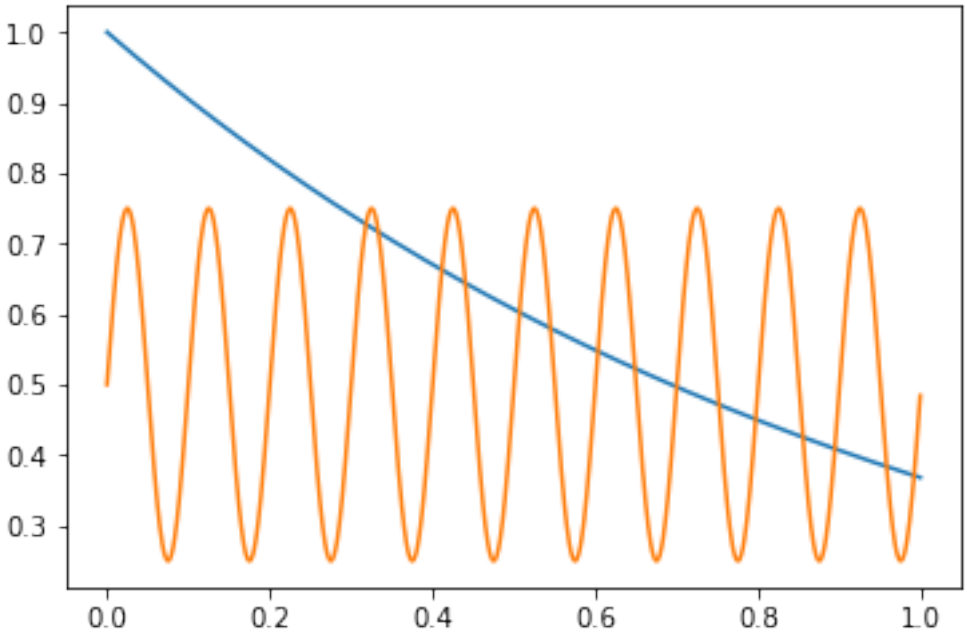
4.2.1 Plotting One Signal

[4]:
```
plt.figure()
plt.plot(x, y1)
plt.show()
```



4.2.2 Plotting Multiple Signals in One Figure

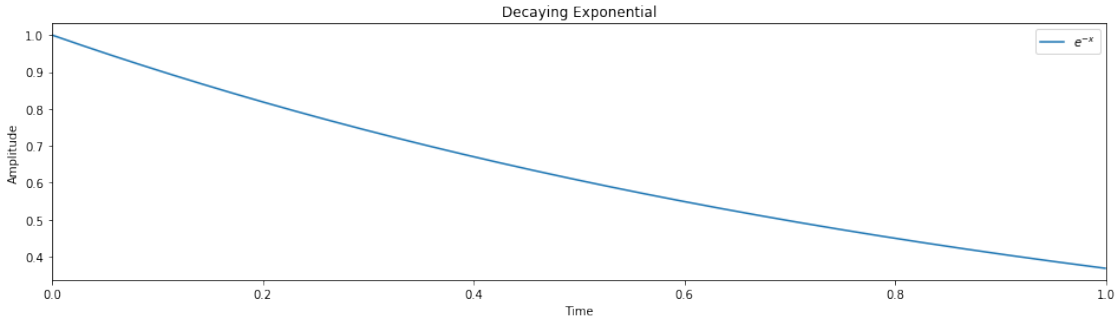
[5]:
```
plt.figure()
plt.plot(x, y1)
plt.plot(x, y2)
plt.show()
```

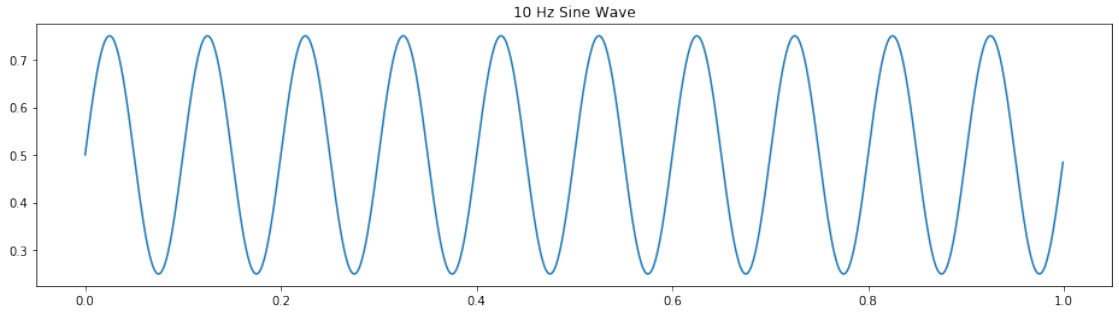


4.2.3 Plotting multiple signals in different figures

[6]:
```
# figsize is the dimensions of the figure:
# - the first argument is the width
# - the second is height

# it's useful when you want to adjust the figure's dimensions, e.g. you
# need a huge x-axis for data taken over a long time period
plt.figure(figsize=(16, 4))
plt.plot(x, y1)

# fancy formatting stuff - try playing with it!
plt.title("Decaying Exponential")
plt.xlabel("Time")
plt.ylabel("Amplitude")
plt.legend(["$e^{-x}$"])    # LaTeX fancy formatting
plt.xlim([0, 1])            # zoom in on x-axis


# asking plt for a new figure before plotting will put the next call to plt.plot
# on that new figure
plt.figure(figsize=(16, 4))
plt.plot(x, y2)
plt.title("10 Hz Sine Wave")
```

```
# ALWAYS make sure to call plt.show() *ONCE* after all your plotting code so␣
 ↪your plots are displayed!
# You only need to call it once per code cell, even if you have multiple␣
 ↪figures.
plt.show()
```





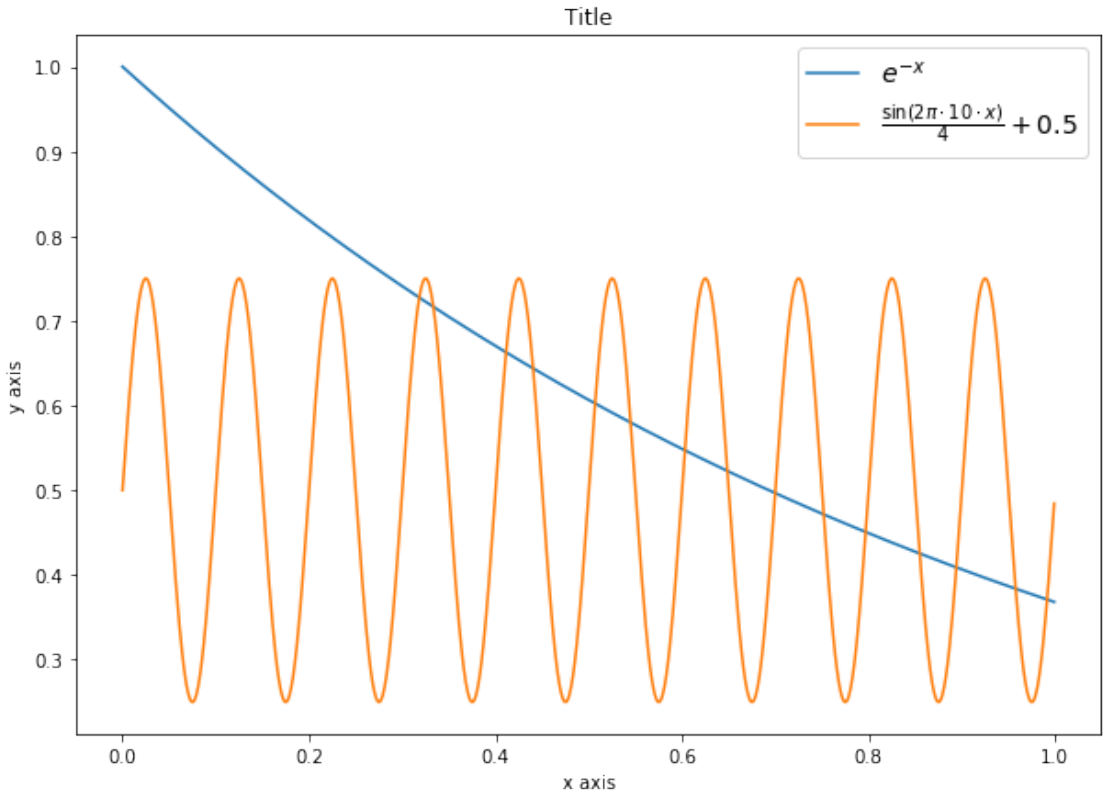
Make no mistake - the data points used for plotting on a computer truly always are discrete, but matplotlib's `plt.plot()` function interpolates them, giving us the continuous waveforms you see above.

4.2.4 You can also add a title and legend with `plt.title()`, `plt.legend()` to make your plots look professional!

Using dollar signs you can add math symbols (like Latex)!

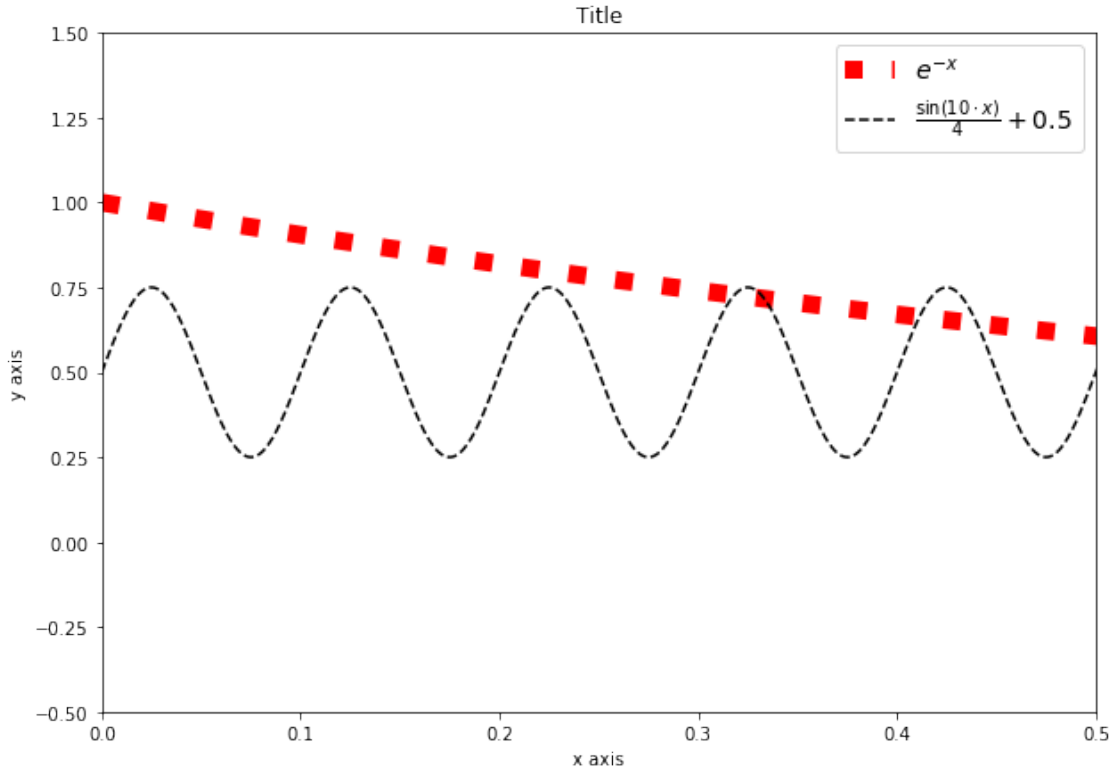
[7]:
```
# The figsize parameter can help you shape your figure
plt.figure(figsize=(10,7))
plt.plot(x, y1)
plt.plot(x, y2)
plt.xlabel("x axis")
plt.ylabel("y axis")
plt.title("Title")
```

```
# You can also change the legend font size by passing in the fontsize= paramater
plt.legend((r'$e^{-x}$', r'$\frac{\sin(2\pi \cdot 10\cdot x)}{4}+0.5$'),␣
 ↪fontsize=14)
plt.show()
```



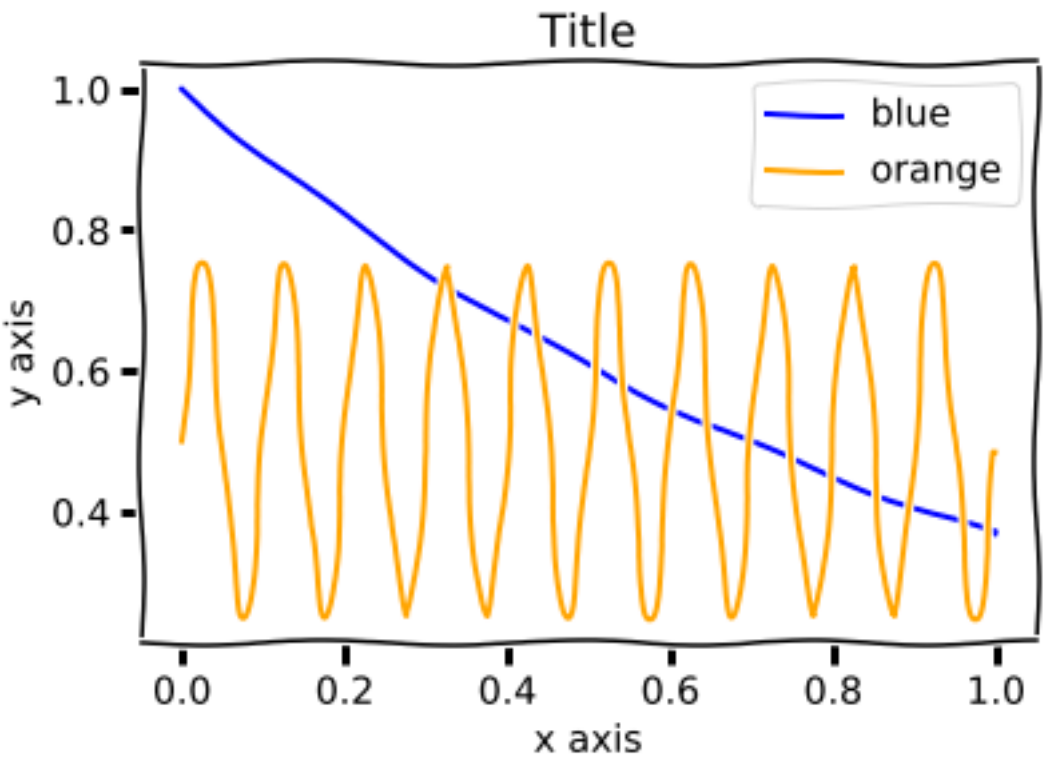
You can also specify more options in `plot()`, such as color and linewidth. You can also change the axis using `plt.axis`

[8]:
```
plt.figure(figsize=(10,7))
plt.plot(x, y1, ":r", linewidth=10)
plt.plot(x, y2, "--k")
plt.xlabel("x axis")
plt.ylabel("y axis")

plt.title("Title")

plt.legend((r'$e^{-x}$', r'$\frac{\sin(10\cdot x)}{4}+0.5$'), fontsize=14)

# plt.axis takes in a list of the form [x_lower, x_upper, y_lower, y_upper]
plt.axis([0, 0.5, -0.5, 1.5])
plt.show()
```



[9]:
```
# xkcd: the Comic sans of plot styles
with plt.xkcd():
    plt.figure()
    plt.plot(x, y1, 'b')
    plt.plot(x, y2, color='orange')
    plt.xlabel("x axis")
    plt.ylabel("y axis")
    plt.title("Title")
    plt.legend(("blue", "orange"))
    plt.show()
```

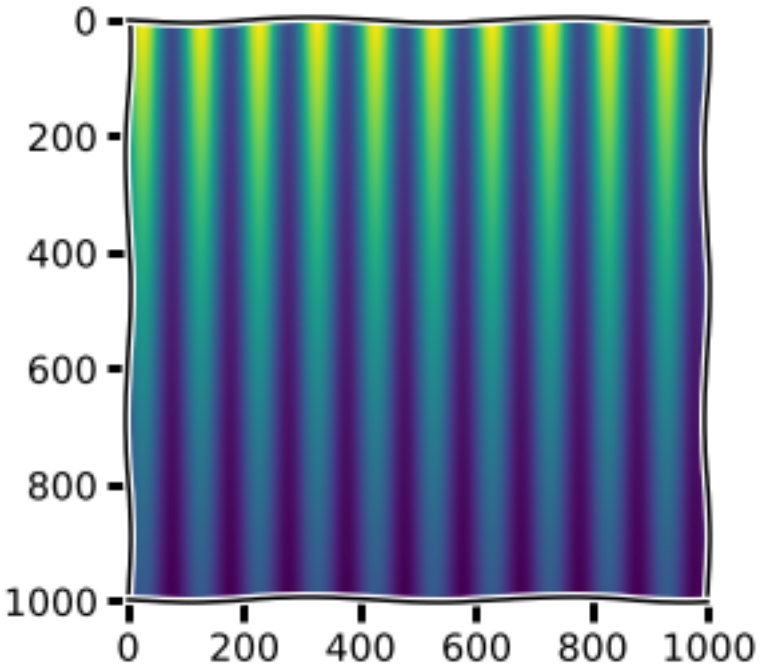
```
/home/oscar/anaconda3/lib/python3.6/site-
packages/matplotlib/font_manager.py:1316: UserWarning: findfont: Font family
['xkcd', 'Humor Sans', 'Comic Sans MS'] not found. Falling back to DejaVu Sans
  (prop.get_family(), self.defaultFamily[fontext]))
```



4.2.5 Other Plotting Functions

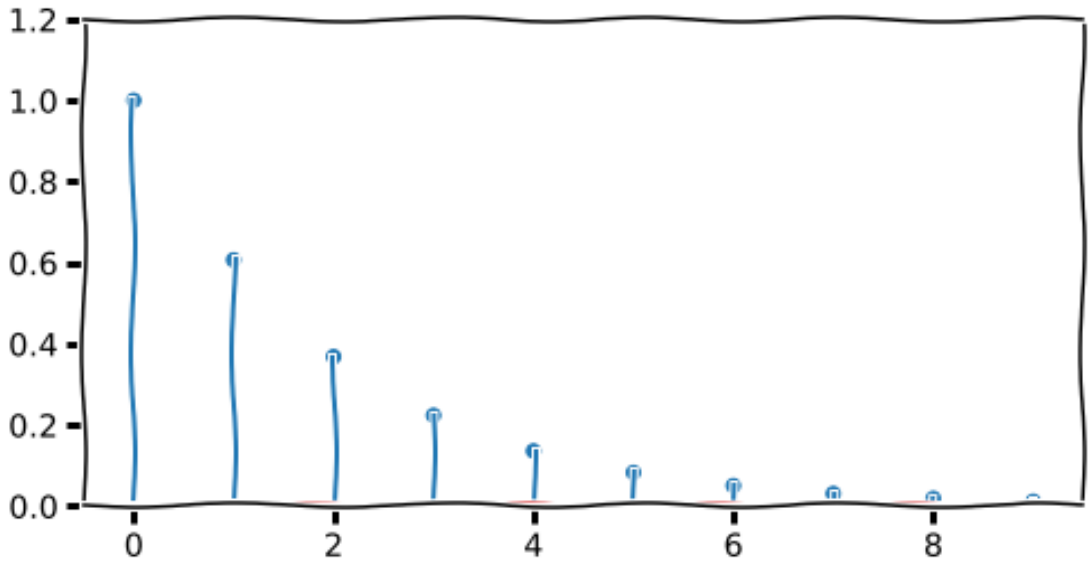
There are many other plotting functions. For example, we will use `plt.imshow()` for showing images.

[10]:
```
image = np.outer(y1, y2) # plotting the outer product of y1 and y2

plt.figure()
plt.imshow(image)
plt.show()
```



Similarly, we use `plt.stem()` for plotting discretized signals (technically, on a computer, everything is discretized, but it's often as a result of sampling something continuous, in which case it's often more informative to plot it as a continuous signal with `plt.plot()`).

[11]:
```
# decaying exponential
n = np.arange(0, 10)
signal = np.exp(-n / 2)

# stem plot on an 8-by-4 inch figure
plt.figure(figsize=(8, 4))
plt.stem(n, signal)

# zoom out a little on both axes to get cleaner looking plot
plt.xlim([-.5, 9.5])
plt.ylim([0, 1.2])
plt.show()
```



5 Q2: Practice with Signals

Now that you're acquainted with the basics of Python and the iPython notebook environment, it's time to practice generating and plotting signals, since it's such a large part of what you'll be doing in the labs.

[12]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

5.1 Signals on computers vs. pencil and paper

In EE 120, we will consider two types of signals: - Discrete Time (DT) signals, which are functions from the integers ($\{..., -2, -1, 0, 1, 2, ...\}$) to the real (or complex) numbers. - Continuous Time (CT) signals, which are functions from the real numbers to the real (or complex) numbers.

From a mathematical point of view, considering signals as functions is incredibly useful. It enables us to do all sorts of operations on them that you're already familar with, such as multiplying them by each other or taking linear combinations of them.

5.1.1 Interpreting time indices

However, when working on a computer, there is a key limitation we must cope with: **finite memory**. We can't store a signal's value for every integer (in the DT case) or every real number (in the CT case). Instead, we represent signals as discrete arrays of some fixed length N. As a result, there

are understood conventions in signals and systems for how to interpret the time indices associated with a signal's array representation: - In DT, the first value is considered to be $n = 0$, and the last to be $n = N - 1$, for a total of N values. Since the first value you have is the start of the signal, it makes to call it time "zero". This is convenient, as it's the same as the array indexing convention in (most) programming languages: the first value has index 0, and the last has index in a length N array is $N - 1$. Outside of these N values, the signal is assumed to be implicitly zero. - In CT, signals are first *sampled* before they go onto a computer for processing, a method we'll discuss in more detail later in the semester. Again, the first sample is considered as time zero as in the DT case, but based on the *sampling rate*, we can assign a time in seconds to each sample. For example, digital audio is typically obtained by sampling acoustic pressure on a microphone (a CT signal) at 44.1 kHz, or 44100 times a second. This means that each sample is separated by 1/44100 seconds, so our samples correspond to $t = 0$ seconds (the 1st sample), $t = 1/44100$ seconds (the 2nd sample), $t = 2/44100$ seconds (the 3rd sample), and so on, until $t = (N - 1)/44100$ seconds when the last sample is taken for N total samples. Again, outside of these N samples, the signal is assumed to be zero.

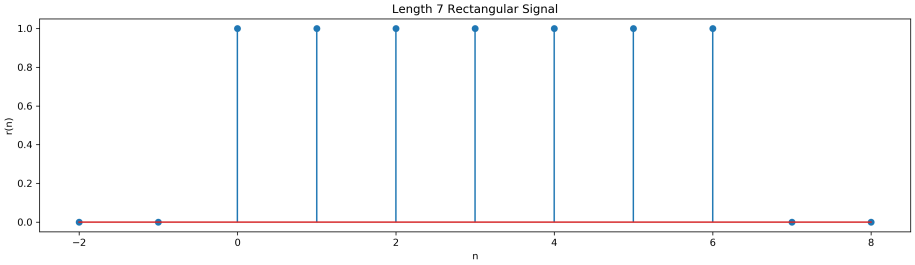
In real world signal processing tasks, you're typically given information about where your signal came from, including whether or not it's truly DT or a sampled version of a CT signal, and if so, what the sampling rate was. In this lab, we'll only consider DT signals, with associated time indices given to you. Sometimes, you'll be asked to include some of the implicit zeros as you gain comfort with these conventions.

5.2 Q2a: The Rectangular Signal

The length L rectangular signal (sometimes also called the "rect" for short, or, alternatively, the "boxcar" signal) is defined as

$$r(n) = \begin{cases} 1 & n = 0, 1, 2, ..., L - 1 \\ 0 & \text{otherwise} \end{cases}$$

Here's an example plot for $L = 7$, with time indices shown from -2 to 8 (so some implicit zeros are shown):



Some alternate definitions of the rect will normalize it (so that each nonzero point of the signal has height $1/L$), and some will center it around zero (although this can only be done when L is odd, so that there is a center point and an equal number of nonzero signal points on each side of $n = 0$).

5.2.1 Your Job

Fill in the cell below to *generate and plot* a **length 5** rectangular signal, with the time indices ranging from -2 to 8.

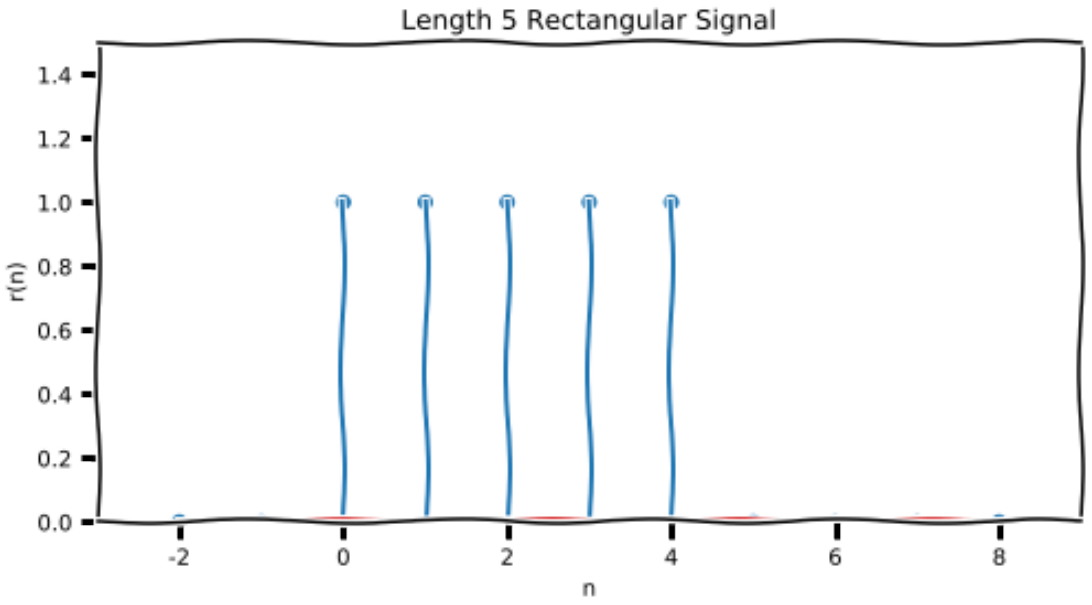
Some quick refreshers from the content of Q1:
- We represent both signals and time indices as numpy arrays. For example, a length 3 rectangular signal can be constructed using the code `np.array([1, 1, 1])`, assuming you wanted to consider it for $n = 0, 1, 2$. - The function `np.arange` is extremely useful for generating sequences of numbers, as we often require when creating time indices for our data, as numpy arrays. - When calling `plt.stem(x, y)`, the number of elements in `x` must equal the number of elements in `y`. To solve this, we can pad on some of the implicit zeros in our signal, as was done in the above plot, where the zeros for $n = -2, -1, 7, 8$ were included.

As in the plot above, label x and y axes with `plt.xlabel` and `plt.ylabel` (using n and $r(n)$ respectively as was done above is fine; we just want you to get in the habit of labelling your plots). Additionally, use `plt.title` to title your plot as "Length 5 Rectangular Signal".

Hint: We use `plt.stem`, not `plt.plot`, for generating "lollipop" plots for discrete-time signals.

[13]:
```
def box_signal(time_min, time_max, length):
    time_indices = np.arange(time_min, time_max + 1)
    signal = np.zeros(time_max + 1 - time_min)
    if time_min != 0:
        signal[-time_min: length - time_min] = 1
    else:
        signal[:length] = 1
    plt.figure(figsize=(8, 4))
    plt.stem(time_indices, signal)
    plt.xlim(time_min - int(length * 0.2), time_max + int(length * 0.2))
    plt.ylim(0, 1.5)
    plt.xlabel('n')
    plt.ylabel('r(n)')
    plt.title(f'Length {length} Rectangular Signal')
    plt.show()
```

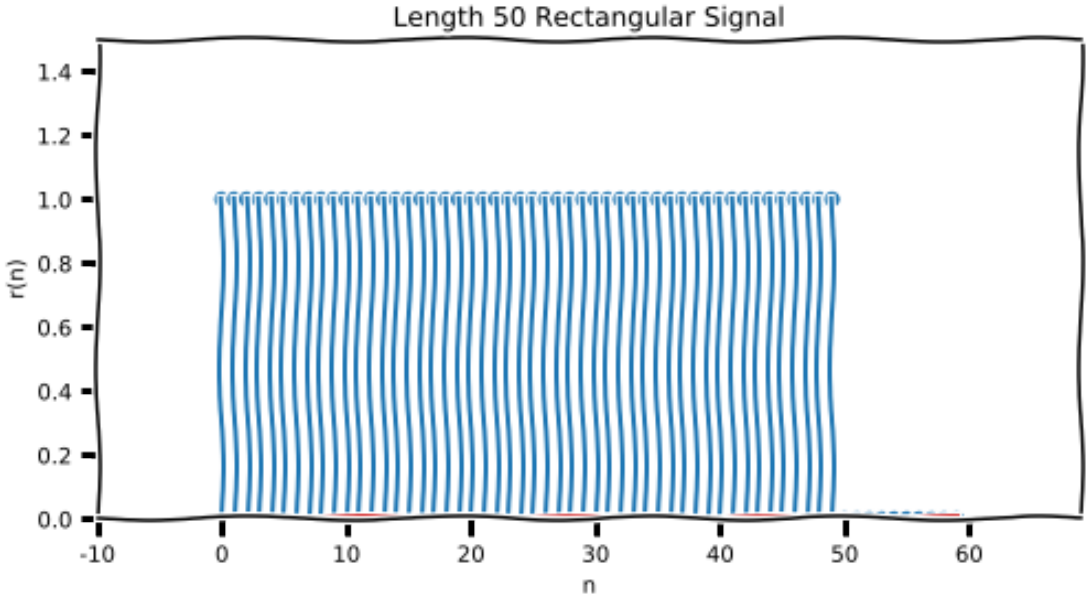
[14]:
```
box_signal(-2, 8, 5)
```



Now, fill in the cell below to *generate and plot* a **length 50** rectangular signal, with the **time indices ranging from 0 to 59**. Give your plot the title "Length 50 Rectangular Signal".

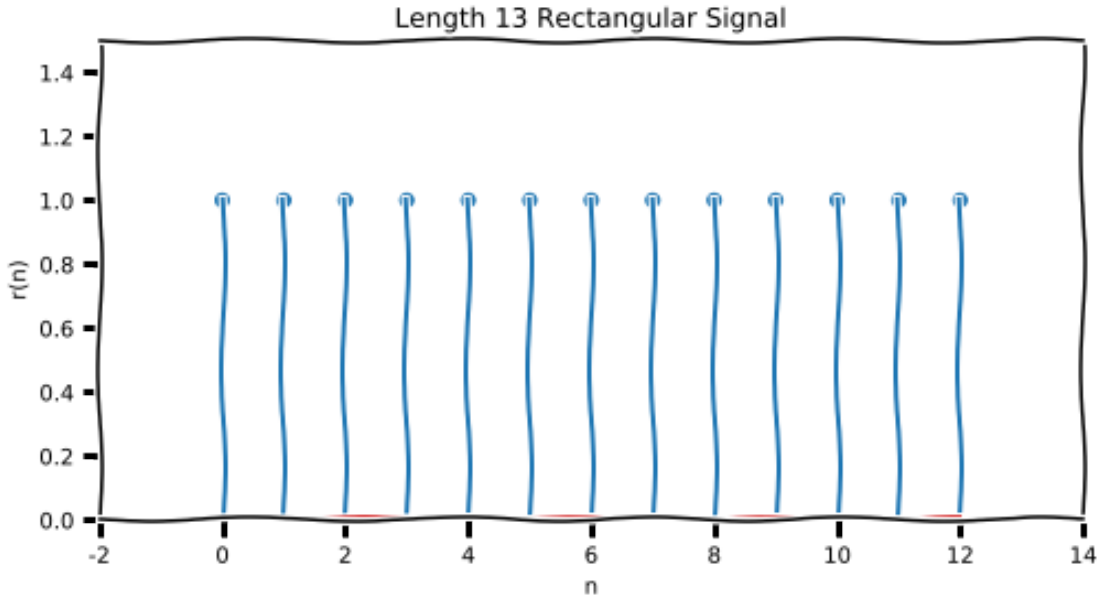
Hint: It's very tedious to type out 50 ones and 10 zeros. This is a great time to practice using the functions `np.concatenate`, `np.ones`, and `np.zeros`.

[15]:
```
box_signal(0, 59, 50)
```



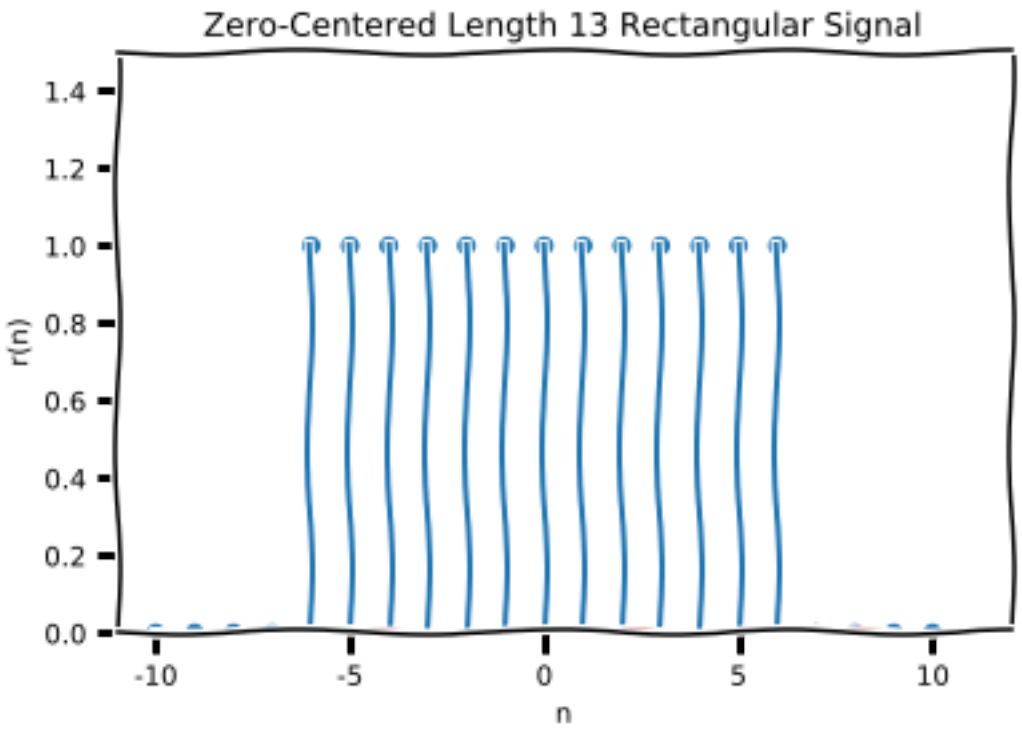
Now, fill in the cell below to *generate and plot* a **length 13** rectangular signal, with the **time indices ranging from 0 to 12**. Give your plot the title "Length 13 Rectangular Signal". Note that the indices 0 to 12 correspond to a total of 13 time points, so no padding of the implicit zeros is necessary.

[16]:
```
box_signal(0,12, 13)
```



Finally, fill in the cell below to *generate and plot* a **length 13** rectangular signal, with the **time indices ranging from -10 to 10**. This time, there's a twist: instead of using the definition above as we did in the previous part, make this signal zero-centered, so that the signal is nonzero from -6 to 6. Give your plot the title "Zero-Centered Length 13 Rectangular Signal".

[17]:
```
# TODO: Zero- entered length 13 rectangular signal code + plot for n=-10 to 10
time_min, time_max = -10, 10
time_indices = np.arange(time_min, time_max + 1)
signal = np.zeros(len(time_indices))
shift, length = 6, 13
signal[-time_min - shift: length - time_min - shift] = 1
plt.stem(time_indices, signal)
plt.xlim(time_min - 1, time_max + 2)
plt.ylim(0, 1.5)
plt.xlabel('n')
plt.ylabel('r(n)')
plt.title('Zero-Centered Length 13 Rectangular Signal')
plt.show()
```



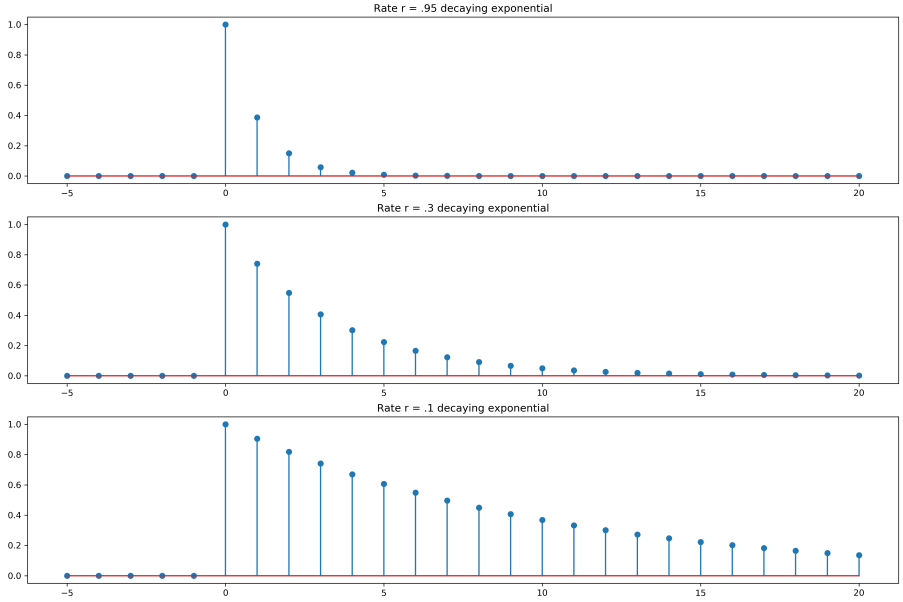
5.3 Q2b: The One-Sided Decaying Exponential Signal

We define the rate r, for all real numbers $r > 0$, one-sided decaying exponential signal as

$$x(n) = e^{-rn}u(n) = \begin{cases} e^{-rn} & n = 0, 1, 2, ... \\ 0 & \text{otherwise} \end{cases}$$

Again, we've adopted the convention of starting our signal at $n = 0$. Specifically, this one-sided decaying exponential is said to be *right-sided* since it decays as we go to the right (in fact, since the first nonzero point is at $n = 0$, we call the signal *causal*).

Here's a plot of the signal for a few different values of r, displayed for $n = -5$ to $n = 20$:



As we increase r, the signal decays to zero faster and faster. In the case of $r = 0$, we obtain the unit step, which doesn't decay at all.

5.3.1 Coping with infinite duration signals: Truncation

Note that for all $r > 0$, the signal has infinite duration, unlike our rectangular signal: we can pick an arbitrary $n > 0$, and $x(n)$ will be nonzero (in fact, it will be positive, since the exponential function is always positive).

This creates an issue for us, since we can't store an infinite number of signal values. Fortunately, there's an easy and well-justified solution: *truncation*. Just as calling the "int" function causes Python to cast floating-point numbers to integers by chopping off, or *truncating*, the decimal part of the number (e.g. `int(3.1415) = 3`, so that `.1415` is removed), we can ignore all the signal values past some point by not including them. For example, in the above plots, we truncate the signals at $n = 20$, so there are only 21 nonzero points in the output. The reason this is well-justified is that for most rates that aren't too small, exponentials decay very rapidly. In the first two plots above, for $n = 20$, the signal value appears to have already decayed to zero anyways! For the third, we'd simply need to use more data points (likely around 50-100) if we wanted to capture all signal values above ~.001 or so.

5.3.2 Your Job

Rather than recreate the same code for generating our signal several times as we did in Q2a, let's create a function for generating our one-sided decaying exponential signal. Then, we can just call

the function each time we want to generate the signal! Fill in the `decaying_expo` function below, and run the cell below it to make sure you wrote it correctly.

Your function should return two separate things, in this order:
1) The time indices.
2) The signal values.

If you're unsure of how to return multiple values from a Python function, you should go back to Q1.

Hint 1: The function `np.exp` will be of use, as `np.exp(x)` computes e^x for any number x. Recall from Q1 that numpy has been designed around *vectorization*: you can call `np.exp` (or, more generally, any numpy function that acts on numbers) on a numpy array and it will apply it element-wise! For example, `np.exp(np.array([1, .5, .25]))` would return a numpy array whose entries are $e^1, e^{.5}, e^{.25}$.

Hint 2: Your function needs to handle two separate cases: `n_start >= 0`, and `n_start < 0`. We recommend you define two variables: one for your time indices, and one for your signal. Then, based on `n_start`, assign them the appropriate values.

[18]:
```
def decaying_expo(rate, n_start, n_end):
    """
    Returns a right-sided decaying exponential signal, truncated to the␣
↪provided indices.

    Parameters:
    rate    - The decay rate.
    n_start - The first time index to use in generating the signal.
    n_end   - The last time index to use in generating the signal.

    Returns:
    n       - The time indices the signal is generated for: n_start, n_start +␣
↪1, ..., n_end.
    sig     - The signal values for each input in n.

    If n_start is not less than n_end, a ValueError will be raised, as this␣
↪specifies an empty
    range of time indices to generate the signal over.
    """
    if n_start >= n_end:
        raise ValueError("n_start must be less than n_end")
    n = np.arange(n_start, n_end + 1)
    sig = np.exp(-rate * np.arange(n_start, n_end + 1))
    if n_start < 0:
        sig[:-n_start] = 0
    return n,sig
```

[19]:
```
def run_tests():
    num_passed = 0
```

```
    ### Index tests, non-negative n_start ###
    test1_yours, _ = decaying_expo(.95, 0, 20)
    test1_staff = np.arange(0, 21)
    test1_passed = np.allclose(test1_yours, test1_staff)

    test2_yours, _ = decaying_expo(.8, 3, 20)
    test2_staff = np.arange(3, 21)
    test2_passed = np.allclose(test2_yours, test2_staff)

    print("Testing indices are correct when n_start >= 0")
    print("Test 1 Passed: {}".format(test1_passed))
    print("Test 2 Passed: {}".format(test2_passed))

    ### Signal tests, non-negative n_start ###
    _, test3_yours = decaying_expo(.95, 0, 20)
    test3_staff = np.array([1.0,0.38674102345450123,0.14956861922263506,0.
↪057844320874838484,0.0223707718561656,0.008651695203120634,0.
↪003345965457471275,0.001294022105465849,0.0005004514334406108,0.
↪0001935450995580941 8,7.48518298877006e-05,2.894827329821157e-05,1.
↪119548484259096e-05,4.329753266092978e-06,1.674493209434269e-06,6.
↪475952175842209e-07,2.504516372327622e-07,9.685992250925397e-08,3.
↪745970556295258 4e-08,1.4487204867720514e-08,5.602796437537268e-09])
    test3_passed = np.allclose(test3_staff, test3_yours)

    _, test4_yours = decaying_expo(.8, 0, 20)
    test4_staff = np.array([1.0,0.44932896411722156,0.20189651799465538,0.
↪09071795328941247,0.04076220397836621,0.01831563888873418,0.
↪008229747049020023,0.003697863716482929,0.001661557273173934,0.
↪0007465858083766792,0.0003354626279025118 5,0.0001507330750954765,6.
↪772873649085378e-05,3.0432483008403625e-05,1.3674196065680938e-05,6.
↪14421235332821e-06,2.7607725720371986e-06,1.2404950799567113e-06,5.
↪573903692694596e-07,2.504516372327617e-07,1.1253517471925912e-07])
    test4_passed = np.allclose(test4_staff, test4_yours)

    _, test5_yours = decaying_expo(.3, 0, 20)
    test5_staff = np.array([1.0,0.7408182206817179,0.5488116360940265,0.
↪40656965974059917,0.30119421191220214,0.22313016014842982,0.
↪16529888822158656,0.1224564282529819,0.09071795328941251,0.
↪06720551273974978,0.049787068367863944,0.036883167401240015,0.
↪02732372244729257,0.02024191144580439,0.014995576820477703,0.
↪011108996538242306,0.00822974704902003,0.006096746565515638,0.
↪00451658094261267,0.003345965457471272,0.0024787521766663585])
    test5_passed = np.allclose(test4_staff, test4_yours)

    print("\nTesting signal values are correct when n_start >= 0")
    print("Test 3 Passed: {}".format(test3_passed))
```

```
    print("Test 4 Passed: {}".format(test4_passed))
    print("Test 5 Passed: {}".format(test5_passed))

    ### Index tests, negative n_start ###
    test6_yours, _ = decaying_expo(5, -4, 217)
    test6_staff = np.arange(-4, 218)
    test6_passed = np.allclose(test6_yours, test6_staff)

    test7_yours, _ = decaying_expo(np.pi, -1998, 2019)
    test7_staff = np.arange(-1998, 2020)
    test7_passed = np.allclose(test7_yours, test7_staff)

    print("\nTesting indices are correct when n_start < 0")
    print("Test 6 Passed: {}".format(test6_passed))
    print("Test 7 Passed: {}".format(test7_passed))

    ### Signal tests, negative n_start ###
    _, test8_yours = decaying_expo(1, -4, 14)
    test8_staff = np.array([0.0,0.0,0.0,0.0,1.0,0.36787944117144233,0.
↪1353352832366127,0.049787068367863944,0.01831563888873418,0.
↪006737946999085467,0.002478752176666358 5,0.0009118819655545162,0.
↪00033546262790251185,0.00012340980408667956,4.5399929762484854e-05,1.
↪670170079024566e-05,6.14421235332821e-06,2.2603294069810542e-06,8.
↪315287191035679e-07])
    test8_passed = np.allclose(test8_staff, test8_yours)

    _, test9_yours = decaying_expo(.5, -5, 15)
    test9_staff = np.array([0.0,0.0,0.0,0.0,0.0,1.0,0.6065306597126334,0.
↪36787944117144233,0.22313016014842982,0.1353352832366127,0.
↪0820849986238988,0.049787068367863944,0.0301973834223185,0.
↪01831563888873418,0.011108996538242306,0.006737946999085467,0.
↪004086771438464067,0.002478752176666358 5,0.0015034391929775724,0.
↪0009118819655545162,0.0005530843701478336])
    test9_passed = np.allclose(test9_staff, test9_yours)

    _, test10_yours = decaying_expo(4, -6, 16)
    test10_staff = np.array([0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.01831563888873418,0.
↪00033546262790251185,6.14421235332821e-06,1.1253517471925912e-07,2.
↪061153622438558e-09,3.775134544279098e-11,6.914400106940203e-13,1.
↪2664165549094176e-14,2.3195228302435696e-16,4.248354255291589e-18,7.
↪781132241133797e-20,1.4251640827409352e-21,2.6102790696677047e-23,4.
↪780892883885469e-25,8.75651076269652e-27,1.603810890548638e-28])
    test10_passed = np.allclose(test10_staff, test10_yours)

    print("\nTesting signal values are correct when n_start < 0")
    print("Test 8 Passed: {}".format(test8_passed))
    print("Test 9 Passed: {}".format(test9_passed))
```

```
    print("Test 10 Passed: {}".format(test10_passed))

    test_results = np.array([test1_passed, test2_passed, test3_passed,␣
↪test4_passed, test5_passed, \
                             test6_passed, test7_passed, test8_passed,␣
↪test9_passed, test10_passed])

    print("\n{0} out of {1} tests passed".format(sum(test_results), 10))

run_tests()
```

```
Testing indices are correct when n_start >= 0
Test 1 Passed: True
Test 2 Passed: True

Testing signal values are correct when n_start >= 0
Test 3 Passed: True
Test 4 Passed: True
Test 5 Passed: True

Testing indices are correct when n_start < 0
Test 6 Passed: True
Test 7 Passed: True

Testing signal values are correct when n_start < 0
Test 8 Passed: True
Test 9 Passed: True
Test 10 Passed: True

10 out of 10 tests passed
```

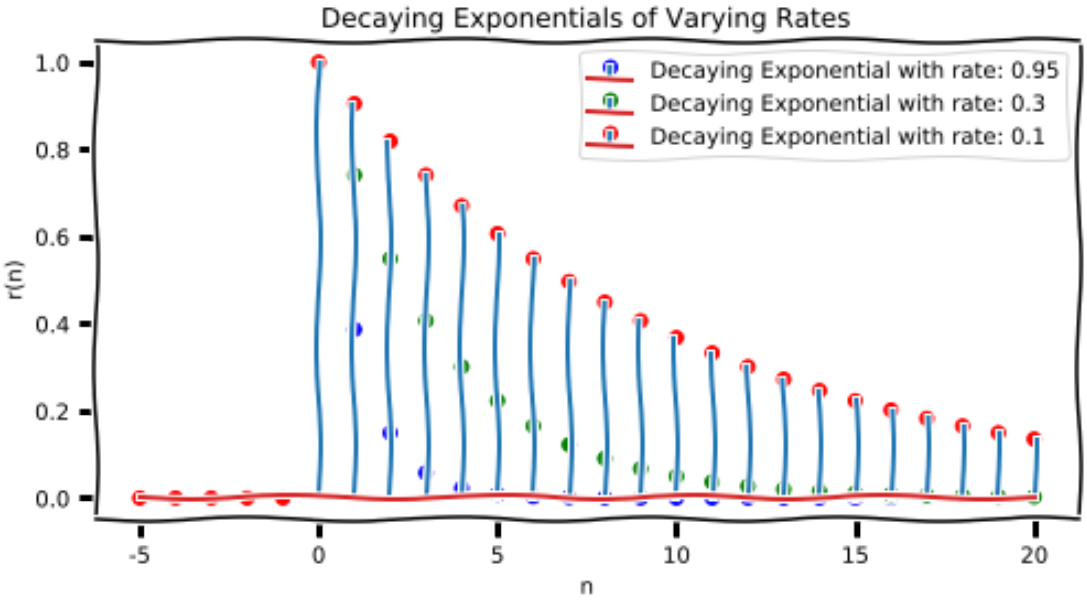
```
/home/oscar/.local/lib/python3.6/site-packages/ipykernel_launcher.py:20:
RuntimeWarning: overflow encountered in exp
```

Once all tests are passing, fill in the cell below to recreate the plot that was given to you above for the right-sided decaying exponentials.

Using your shiny new and tested function, this should be pretty simple; the only work is in generating the plots. If you're unsure of how to plot multiple signals below one another, head back to the **"Plotting multiple signals in different figures"** section of Q1. Alternatively, if you're feeling fancy, this would be a great opportunity to try your hand at using matplotlib's subplot function to generate a 3x1 grid of plots within the same figure. Either plotting method is acceptable.

[20]:
```
# TODO your signal/time index generation here
rates = [.95, .3, .1]
signals = [decaying_expo(rate, -5, 20) for rate in rates]
```

[21]:
```
# TODO your plotting code here
plt.figure(figsize=(8, 4))
color = ['bo', 'go', 'ro']
for signal, rate, color in zip(signals, rates, color):
    t_indices, sig = signal
    plt.stem(t_indices, sig, label=f'Decaying Exponential with rate: {rate}',␣
 ↪markerfmt=color)
    plt.xlabel('n')
    plt.ylabel('r(n)')
plt.legend()
plt.title('Decaying Exponentials of Varying Rates')
plt.show()
```



You can do a quick sanity check on your results by comparing to the original plot above - the two should match exactly (don't forget the plot titles)!

6 Q3: Convolution

As you know, LTI systems act by convolving an input with the system's impulse response. This is why LTI systems are so nice from an analysis point of view - an LTI system is completely characterized by its impulse response, which is typically measureable to a high degree of precision in practice - just send in an impulse!

From a certain point of view, EE 120 is an entire class on LTI system theory and extensions of it. So, naturally, convolution is an important operation for us, both in exploring applications in labs and understanding theoretical underpinnings in class. Accordingly, we've devoted an entire

question to helping you gain familiarity with computing convolutions in Python.

6.1 Convolution with numpy

The numpy function for convolution is np.convolve, which takes in three arguments: - `x`, a numpy array representing the signal $x(n)$. - `h`, a numpy array representing the signal $h(n)$. - `mode`, a string specifying how to truncate the convolution, if at all.

It returns $y(n) = (x * h)(n)$, with some subtle differences in how the boundaries are handled based on what you pass in for `mode`. There are three options for this parameter: `full`, `same`, and `valid`. Rather than explain the differences in words, we'll go through some actual examples.

6.1.1 `mode = "full"`

When you set `mode` to `full`, numpy will compute the entire convolution - if x is a length M array and h is a length N array, the returned array will have length $M + N - 1$. Essentially, this mode is exactly what you've seen so far in class.

[22]:
```
np.convolve([1/2, 1/2], [1, 1, 1], 'full')
```

[22]:
```
array([0.5, 1. , 1. , 0.5])
```

[23]:
```
np.convolve([1, 2, 3], [0, 1, 0.5], 'full')
```

[23]:
```
array([0. , 1. , 2.5, 4. , 1.5])
```

What makes this mode useful? You *never* have to worry about data being cut out with `mode="full"` as you do with other convolution modes, as we'll see below! So, if you're ever concerned about the cropping issue, this is always a safe bet!

6.1.2 `mode = "same"`

This convolution mode only keeps the "middle" part of the result, cutting off data points from each edge until the result has the same length as the longer of the two inputs. This is the most popular convolution mode in practice - since most signals used in digital signal processing tasks are zero padded (for various reasons you'll learn about in later labs), the edges of convolutions often contain zeros, so it's okay to cut them out.

[24]:
```
np.convolve([1/2, 1/2], [1, 1, 1], 'same')
```

[24]:
```
array([0.5, 1. , 1. ])
```

[25]:
```
np.convolve([1, 2, 3], [0, 1, 0.5], 'same')
```

[25]:
```
array([1. , 2.5, 4. ])
```

Note how in both cases, some of our data was cut out! When you use `same`, you have to take care to ensure no data will be cut out in the result by zero-padding both sides of one of the signals. If, for the first convolution, we instead computed

[26]:
```
np.convolve([0, 1/2, 1/2, 0], [1, 1, 1], 'same')
```

[26]:
```
array([0.5, 1. , 1. , 0.5])
```

then all of our data is in tact.

What makes this mode useful? Using "same" allows you to deal with fixed size signals (i.e., all arrays are the same length) throughout your code, since a convolution in "same" mode of two length L signals returns a length L signal via cropping, and you can just zero pad to elimiate issues of data being cut out. Does that sound appealing to you? Same.

6.1.3 `mode = "valid"`

With `valid`, numpy will only keep the part of the convolution where your signals fully overlap.

[27]:
```
np.convolve([1/2, 1/2], [1, 1, 1], 'valid')
```

[27]:
```
array([1., 1.])
```

[28]:
```
np.convolve([1, 2, 3], [0, 1, 0.5], 'valid')
```

[28]:
```
array([2.5])
```

For the second signal, since the signals have the same length, they only full overlap at one point in time. Thus, we get only one value in our output.

What makes this mode useful? It's great when you want to filter some data, but the filter only makes sense when its input is entirely the data from your signal, and no zero padded values are included. For example, if you were applying a moving average filter to stock data, it doesn't make sense to take an average that's partially stock data and partially implicit zeros.

6.1.4 Final Comments on `mode`

For the labs, we'll typically tell you what convolution mode to use to avoid any confusion. In practice, the "same" mode is the most popular, as the two signals you're convolving are typically zero-padded to the same lengths; thus the output is the same length as the two inputs, which makes preallocation of memory for your signals easier to track (since everything has the same length).

We want you to be aware of these subtle differences, as they can be confusing at first. It's worth paying attention to such nuances so you don't waste time trying to fix some boundary issue. And here's the great thing about the iPython notebook environment being so interactive: if you're ever unsure of what something would do, just create a new cell and try it out!

6.1.5 Your Job

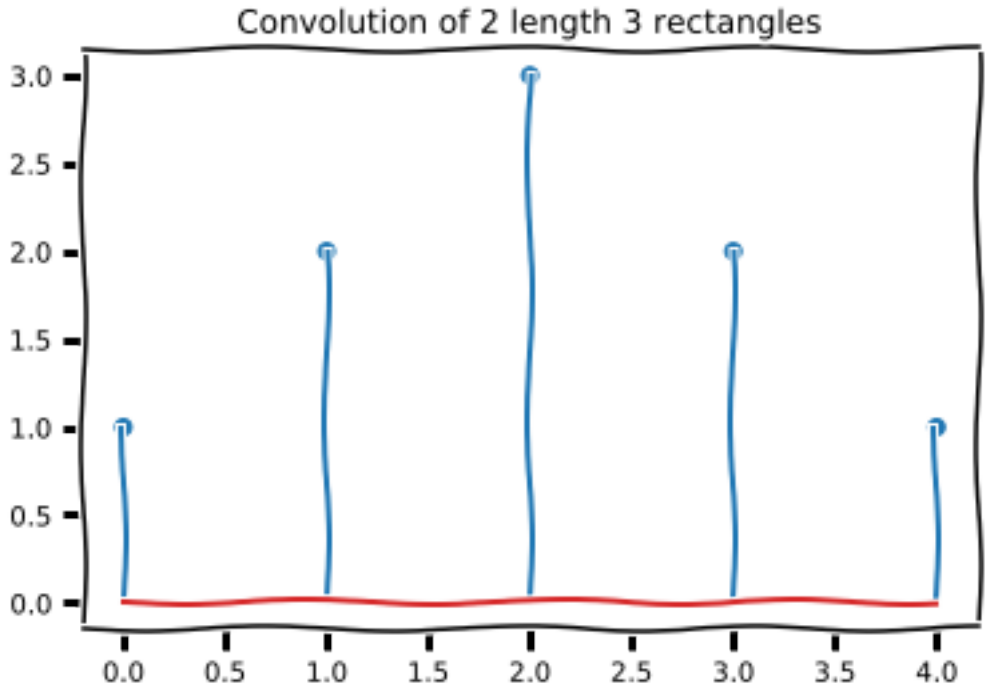
Now that you've seen the basics of how `np.convolve` works, it's time to try it out on your own. Don't worry about defining or tracking any time indices at all (or doing any zero padding) for this question - just generate the signals and convolve as specified by the instructions.

Make sure you use `plt.stem` for all plots in this question, so the results are displayed as DT signals. No need to do anything fancy with your plots either (e.g. labelling axes, etc.) beyond what you're asked for - just passing the signal straight into `plt.stem` and adding whatever title the question specifies is fine.

6.2 Q3a: Get Rect

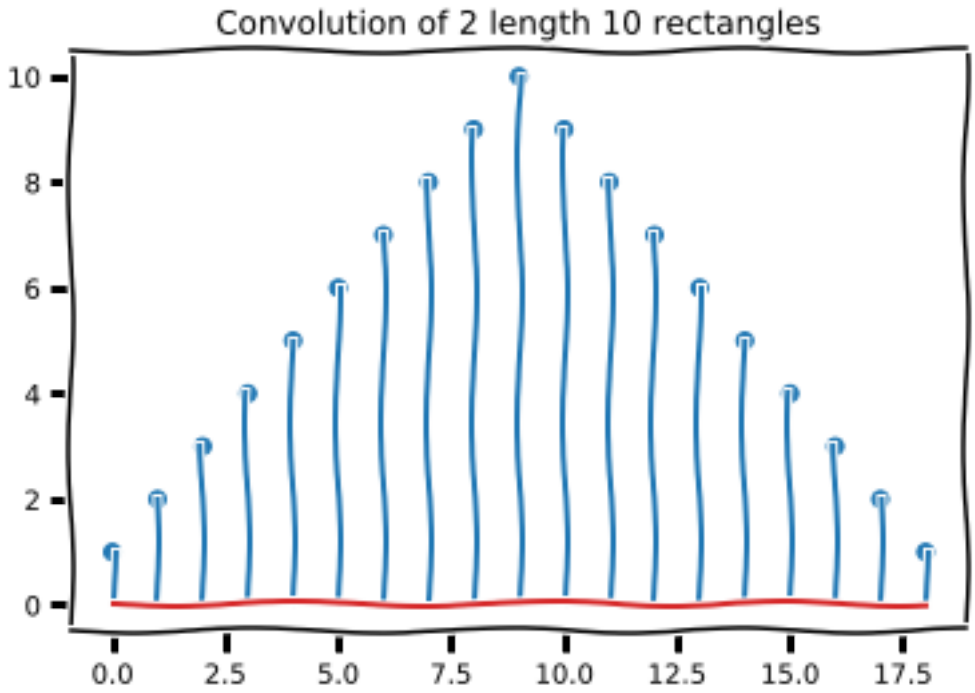
First, convolve a length 3 rect with a length 3 rect and plot it. Use "full" mode. Title it "Convolution of two length 3 rects".

[29]:
```
# TODO: convolution of two length 3 rects (mode="full")
rect_1, rect_2 = np.ones(3), np.ones(3)
plt.stem(np.convolve(rect_1, rect_2, mode='full'))
plt.title('Convolution of 2 length 3 rectangles')
plt.show()
```



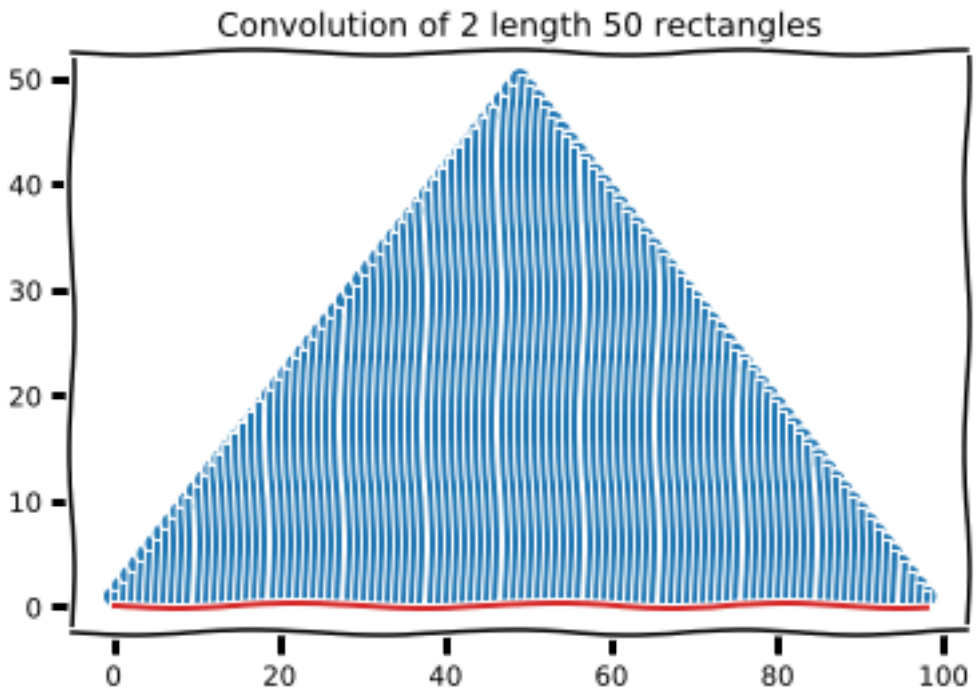
Now, convolve a length 10 rect with a length 10 rect and plot it, again in "full" mode. Title it "Convolution of two length 10 rects".

[30]:
```
# TODO: convolution of two length 10 rects (mode="full")
rect_1, rect_2 = np.ones(10), np.ones(10)
plt.stem(np.convolve(rect_1, rect_2, mode='full'))
plt.title('Convolution of 2 length 10 rectangles')
plt.show()
```



Finally, convolve a length 50 rect with a length 50 rect and plot it, again in ”full” mode. Title it ”Convolution of two length 50 rects”.

[31]:
```
# TODO: convolution of two length 50 rects (mode="full")
rect_1, rect_2 = np.ones(50), np.ones(50)
plt.stem(np.convolve(rect_1, rect_2, mode='full'))
plt.title('Convolution of 2 length 50 rectangles')
plt.show()
```



Q: In general, what shape is the convolution of two rects *of the same length*? A one-word answer is fine.

A: A rectangle

6.3 Q3b: Get Rect, Part II

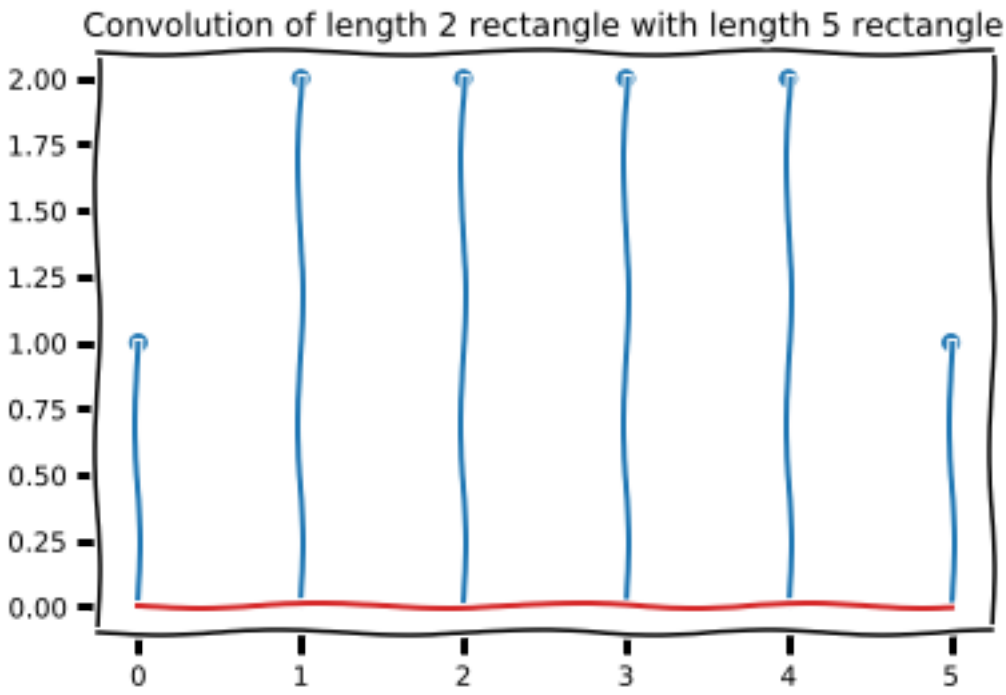
Now, we'll convolve rects that have *different* lengths and see what happens.

For this part, perform all convolutions in "full" mode, so you don't have to worry about signal values being cut out and can focus on the results. As a consequence, it's perfectly fine to not zero-pad any of your signals pre-convolution. Remember, "full" mode doesn't cut anything out. For example, if you were asked to convolve a length 2 rect with a length 4 rect, it's fine to use `np.array([1, 1])` and `np.array([1, 1, 1, 1])`, respectively, as the numpy array representations of your signals.

For all parts of this question, plot the convolution result, and give your plot a reasonable title.

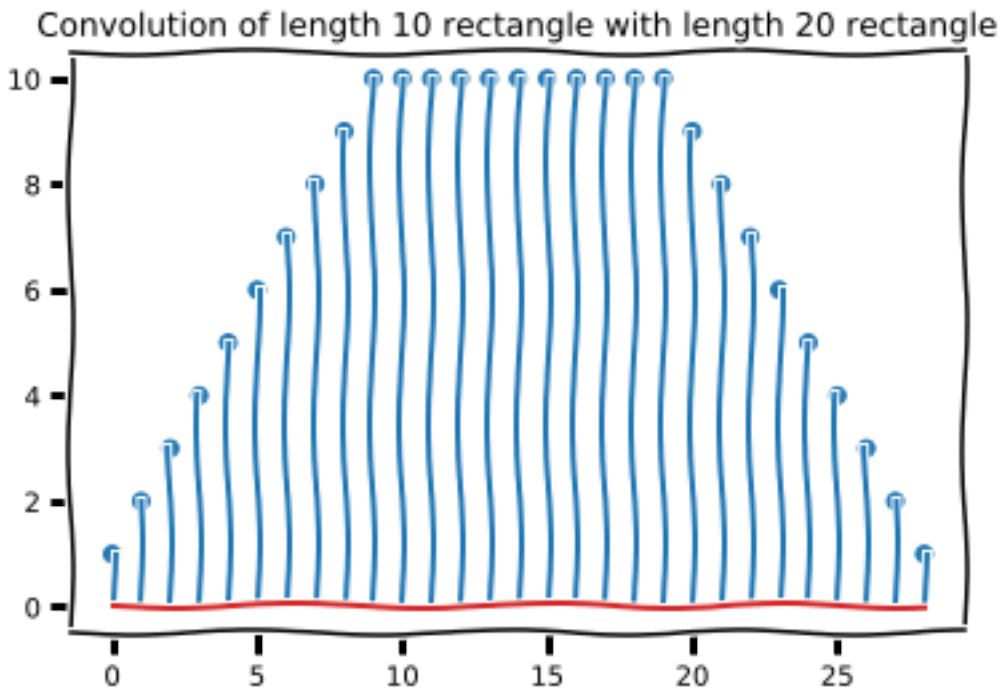
Start by convolving a length 2 rect with a length 5 rect.

[32]:
```
# TODO: convolve and plot length 2 and length 5 rect
rect_1, rect_2 = np.ones(2), np.ones(5)
plt.stem(np.convolve(rect_1, rect_2, mode='full'))
plt.title('Convolution of length 2 rectangle with length 5 rectangle')
plt.show()
```



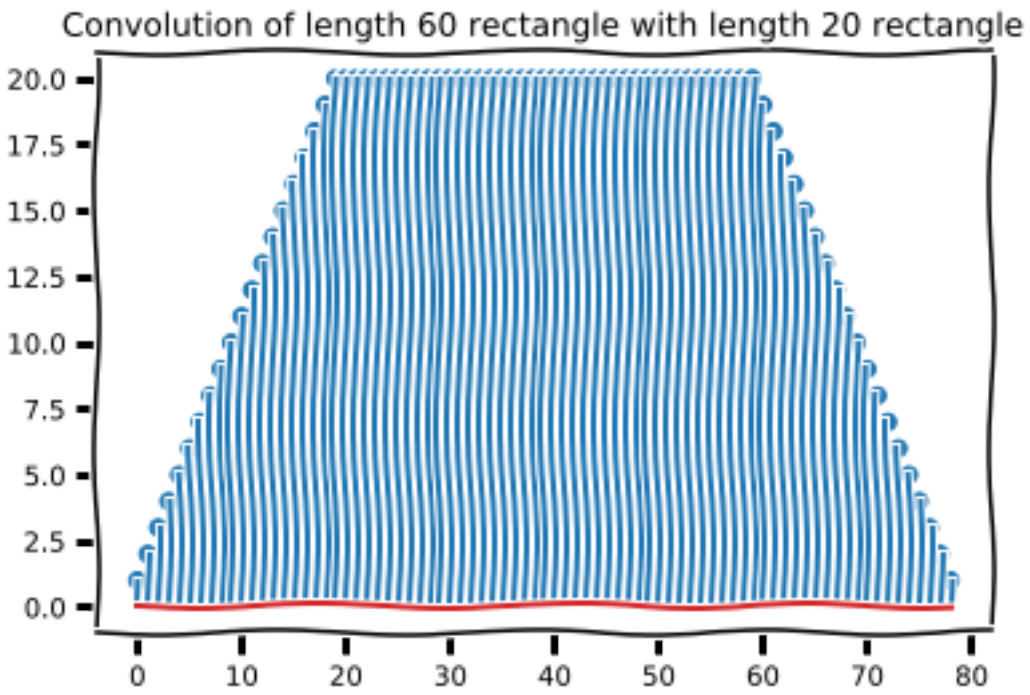
Now, convolve a length 10 and length 20 rect.

[33]:
```
rect_1, rect_2 = np.ones(10), np.ones(20)
plt.stem(np.convolve(rect_1, rect_2, mode='full'))
plt.title('Convolution of length 10 rectangle with length 20 rectangle')
plt.show()
```



Finally, convolve a length 60 and length 20 rect.

[34]:
```
rect_1, rect_2 = np.ones(60), np.ones(20)
plt.stem(np.convolve(rect_1, rect_2, mode='full'))
plt.title('Convolution of length 60 rectangle with length 20 rectangle')
plt.show()
```



Q: In general, the convolution of two rects of different lengths is a trapezoid, which you should see from your plots. We'll define the length of the trapezoid's top as the number of points for which it attains its maximum value. For example, the length of a triangle's top is 1, since it peaks at one data point and slopes downward on either side of the peak.

Which trapezoid would you expect to have a longer top: one obtained by convolving a length 10 rect and length 20 rect, or one obtained by convolving a length 15 rect and length 20 rect? Explain why in 1-2 sentences.

A: The region in which the dot-product of two is maximized when the length completely 20 rectangle overlaps the length 10 rectangle. This occurs over a larger interval when the length 20 rectangle overlaps the length 10 rectangle as opposed to the length 15 rectangle. Therefore, the top would be longer when convolving the length 10 rectangle with the length 20 rectangle.

6.4 Q3c: Get Rect, The Grand Finale

We saw in Q1a what happens when you convolve two rects that have the same length together. But what if we convolve 3 togther (by convolving two, then convolving the result with a third)? What about 4? 5? 10? 100? The result might surprise you.

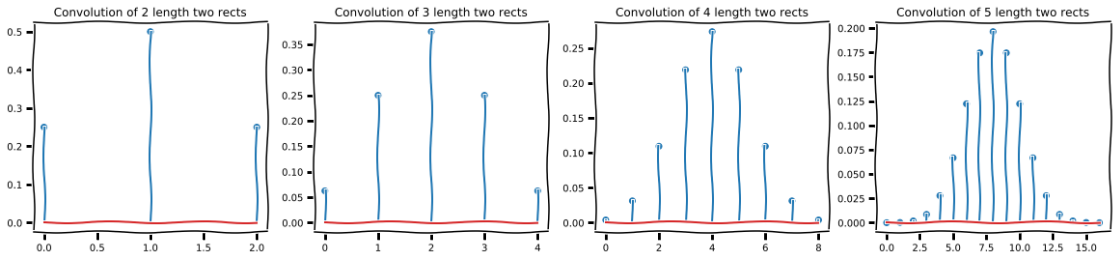
In doing so, to make sure the convolution results don't blow up to infinity, we'll normalize our rects to sum to 1. An interesting fact that you'll prove later in the semester is that if you convolve two signals that both sum to 1, the result also sums to 1. We'll use length two rects here, although you'll get a similar result if you use a larger size, or even if you vary the size of rects you convolve (e.g., you convolve with a length 2, then a length 3, then a length 7, then a length 2 again, and so

on), so long as the total number of convolutions performed is large.

You don't have to write any code for this question; just run the cells.

Let's see what happens when we convolve 2, 3, and 4 rects together.

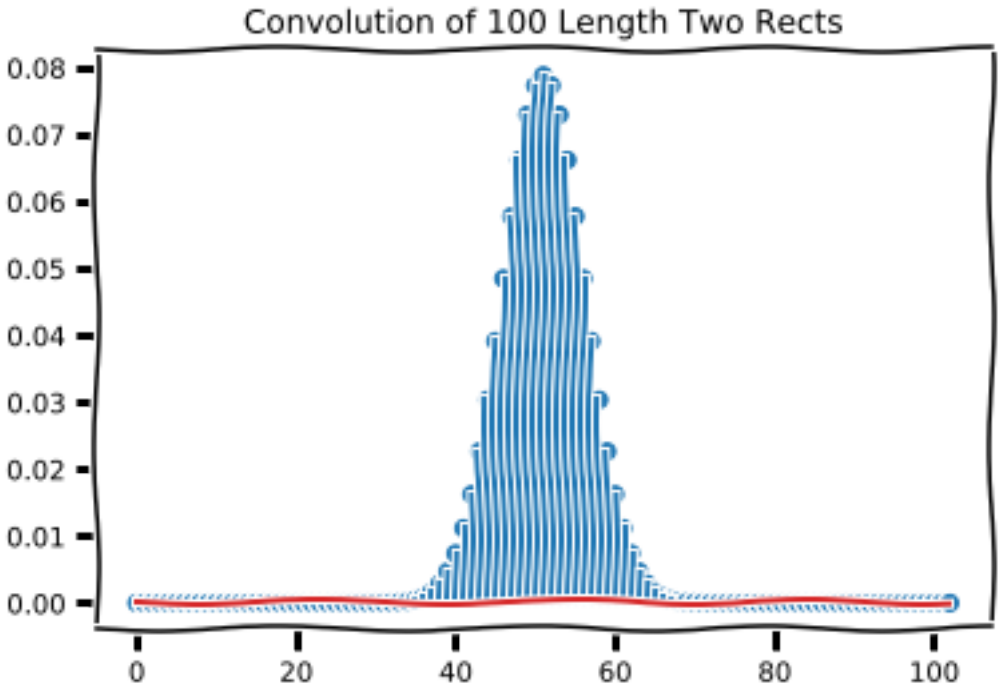
[35]:
```
x = np.array([1/2, 1/2])

plt.figure(figsize=(20, 4))
plt.subplot(1, 4, 1)
x = np.convolve(x, x, "full")
plt.stem(x)
plt.title("Convolution of 2 length two rects")

plt.subplot(1, 4, 2)
x = np.convolve(x, x, "full")
plt.stem(x)
plt.title("Convolution of 3 length two rects")

plt.subplot(1, 4, 3)
x = np.convolve(x, x, "full")
plt.stem(x)
plt.title("Convolution of 4 length two rects")

plt.subplot(1, 4, 4)
x = np.convolve(x, x, "full")
plt.stem(x)
plt.title("Convolution of 5 length two rects")
plt.show()
```

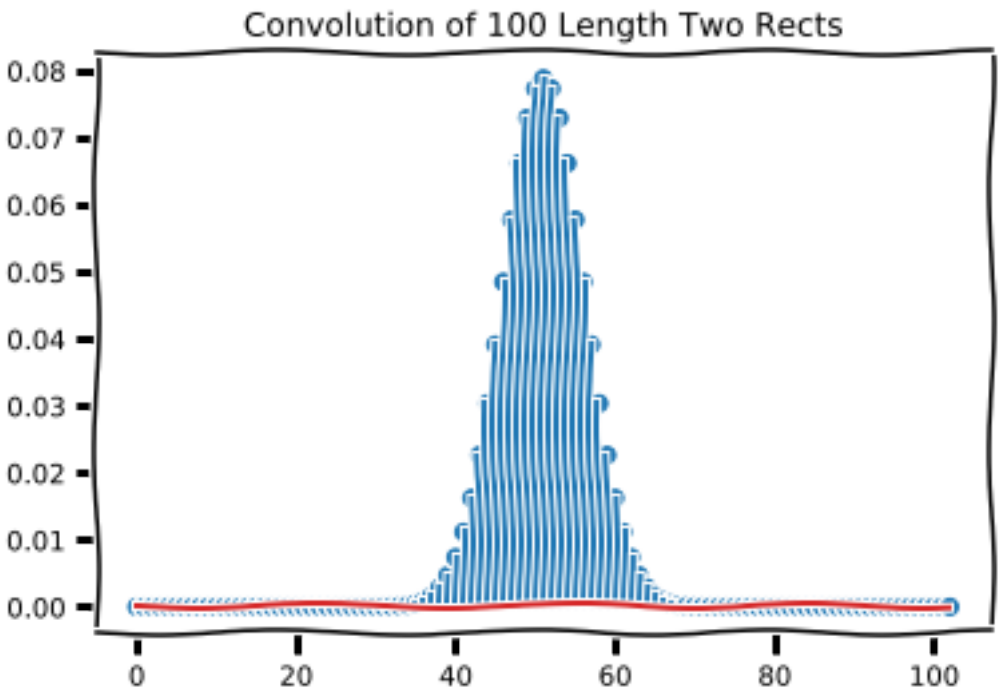


We can see the results getting smoother and smoother each time, in addition to the stretching that naturally occurs as a result of convolution. Perhaps you already have some intuition for what will happen if we keep doing this over and over again. Through a very cool feature of matplotlib, we can repeatedly convolve our rects together and update the same plot, seeing how the result "evolves". Run the cell below to convolve 100 rects together, displaying the result after each convolution.

[36]:
```
from IPython import display
```

```
numconv = 100 # change this to the number of convolutions you want to do
rect = np.array([1/2, 1/2])
y = np.convolve(rect, rect, "full")
for i in range(1, numconv+1):
    y = np.convolve(y, rect, "full")
    plt.clf()
    plt.stem(y)
    plt.title("Convolution of {} Length Two Rects".format(i))
    display.clear_output(wait=True)
    display.display(plt.gcf())
```





As we continue to convolve more and more rects together, we end up with a bell curve! So, if we wanted to apply a large number of filters that all had rectangular impulse responses (such as in the case of a moving average filter, for example), we could equivalently apply a *Gaussian filter*, i.e. one with a Gaussian (bell curve) as its impulse response. The Gaussian filter shows up in a wide variety of signal processing contexts for smoothing, and as you've seen here, it's the limiting case of applying a large number of rectangular filters (i.e., moving averages).

6.5 Optional reading: A probability theoretic view of Q3c

Believe it or not, it's no coincidence that we got a Gaussian-like signal by convolving a large number of rects. This is actually a consequence of one of the most celebrated results in probability theory, the Central Limit Theorem. This theorem says that, under certain conditions, as you add together more and more random variables, the result approaches a Gaussian distribution, i.e. the familiar bell curve. An interesting fact covered in EECS 126 is that if you add two independent random variables, the resultant distribution is the **convolution** of the summands' distributions. Thus, by convolving 100 rects together, we are effectively adding together 100 independent uniform random variables (the probability mass function of a uniform r.v. is a rectangle, since each value is equally probable), and we see that the result has a Gaussian distribution!

Note: We don't expect to you to know any probability theory for this class. If the preceding paragraph made no sense at all, don't worry about it. There are some interesting connections between signal processing and probability theory that will be mentioned from time to time in the labs for your enrichment, but you're not responsible for understanding any of their content.

7 References

[1] The official Python 3 language documentation. Link.
[2] The official numpy and scipy documentation. Link.
[3] The official matplotlib documentation. Link

Special thanks to the Berkeley Python Bootcamp 2013, Python for Signal Processing, EE 123 and and EECS 126 for providing a great starting point for Q1, Introduction to the Python Scientific Computing Stack.