Lab 2

Graded

1 Day, 19 Hours Late

Student

MJ Wu

Total Points

50 / 50 pts

Question 1

Q1 — **30** / 30 pts

1.1 **Q1a** — **10** / 10 pts

- ✔ **– 0 pts** Correct
- **– 5 pts** Incorrect Plots
- **– 2.5 pts** Some right ideas in explanation
- **– 5 pts** Incorrect Explanation

1.2 **Q1b** — **10** / 10 pts

- ✔ **– 0 pts** Correct
- **– 2.5 pts** Some correct graphs
- **– 2.5 pts** Some correct answers
- **– 5 pts** Incorrect graphs
- **– 5 pts** Incorrect answers

1.3 **Q1c** — **10** / 10 pts

- ✔ **– 0 pts** Correct
- **– 2.5 pts** Minor mistake/lack of explanation in 2nd question
- **– 5 pts** Some right ideas
- **– 10 pts** Incorrect

Question 2

Q2 **15** / 15 pts

2.1 **Q2a** **10** / 10 pts

- ✔ **– 0 pts** Correct
- **– 5 pts** Incorrect Plots
- **– 2.5 pts** Some Right Ideas in Explanation
- **– 5 pts** Incorrect Explanation

2.2 **Q2b** **5** / 5 pts

- ✔ **– 0 pts** Correct
- **– 2.5 pts** Incorrect Plot
- **– 2.5 pts** Incorrect Explanation

Question 3

Q3 **5** / 5 pts

- ✔ **– 0 pts** Correct
- **– 2.5 pts** Incorrect EMA
- **– 2.5 pts** Incorrect Explanation

No questions assigned to the following page.

EE 120 Lab 2: Applications of LTI Filtering

Signals and Systems at UC Berkeley

Acknowledgements:

- **Spring 2019** (v1.0): Dominic Carrano, Sukrit Arora, Babak Ayazifar
- **Fall 2019** (v2.0): Dominic Carrano
- **Spring 2020** (v2.1): Dominic Carrano
- **Fall 2020** (v2.1.1): Anmol Parande

In [204]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Background

Now that you're familiar with the Jupyter notebook environment and the Python language's scientific computing capabilities, you'll get to put your skills to use in exploring some applications of filters (LTI systems).

You'll see some examples of filters in this lab that you've probably also seen in class, such as the moving average or the edge detector. Here, we have a computer to do all the heavy lifting of computing convolutions and plotting things for us, so we can more easily explore how these systems behave when given longer or more complicated input signals.

No questions assigned to the following page.

Continuous (Interpolated with `plt.plot`) vs Discrete (Stem with `plt.stem`) Plots

While we're still considering our signals as discrete-time entities in this lab, many of them will be fairly long, containing hundreds or even thousands of signal values. As a result, you should **use `plt.plot` for plotting all signals in this lab, unless told otherwise**.

We used `plt.stem` almost exclusively in Lab 1, but stem plots have two major disadvantages which continuous-time plots (i.e. ones generated with `plt.plot`, which interpolates your signals) don't suffer from:

- It's difficult to overlay multiple signals on the same stem plot and visually compare them.
- Stem plots' rendering times scale *significantly* worse with the length of the signal being plotted.
  - Just to get a sense for how bad a stem plot is at scale, we compared times for plotting with `plt.stem` against `plt.plot` on a 2015 MacBook Pro. On average, for a length 1000 rect, `plt.plot` took ~100 ms and `plt.stem` took 2-3 seconds; for a length 10000 rect, `plt.plot` took ~200 ms and `plt.stem` took 1.5-2 ***minutes***.
  - Remember the demo in Q3c of Lab 1 where we repeatedly convolved rects and saw that we eventually get a Gaussian (the bell curve)? You probably noticed that it took longer and longer to display the successive results - the issue wasn't that the convolutions were taking long due to the signal length increasing (the convolution runtime was increasing with the signal length, but negligibly so); rather the repeated stem plotting was the bottleneck.

The two reasons listed above are some reasons why almost all plots you'll see in real world applications are interpolated. Keep in mind that there's actually a more complicated interplay going on between the continuous and discrete realms when doing digital processing of signals:

1. A continuous time signal $x(t)$ is sampled at some rate T, giving N samples at integer multiples of T as $x(nT)$. We then define $x[n] = x(nT)$, a discrete time signal, which we can work with on the computer; note that $n = 0, 1, \ldots, N-1$ since we only have N samples. Most practical signals are finite in duration, however, and so N can be chosen to be sufficiently high to capture the entire signal.
2. Processing is done on $x[n]$ on a computer, to produce some related signal $y[n]$.
3. $y[n]$ is plotted as a continuous-time signal via interpolation.

Much like the conventions discussed in Lab 1 of what "zero" means as a time index, this is one of the nuances of how signals and systems is done in practice that you get used to with experience.

In this lab, some of the signals we'll use are just made up "test signals" so we can experiment with different filters, in which case the above steps 1-3 aren't relevant. However, in other parts, we'll be using some real world data, and it's good to keep 1-3 straight in your head.

Question assigned to the following page: 1.1

Q1: The 1D Edge Detector

The 1D edge detector is used in signal processing to, as you may have guessed from the name, detect edges, or jumps in a signal's amplitude. It's referred to as 1D to distinguish it from 2D edge-detecting filters used in image processing — basic signals only containing amplitude versus time information are referred to as "1D" whereas images are often considered as "2D" signals. Videos are considered "3D" signals, with time (as you go from frame to frame) being the third dimension. Almost all signals you'll see in EE 120 will be 1D.

The impulse response of a 1D edge detector is defined as:

$$h(n) = \delta(n) - \delta(n-1)$$

In this question, we'll explore several important properties of the edge detector.

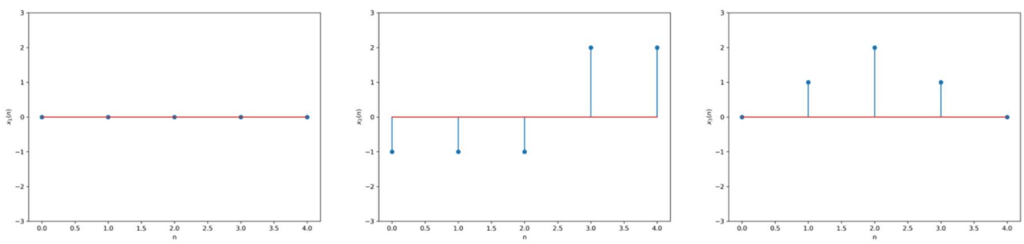
The filter works by taking the difference between every pair of adjacent signal values. If you have a sequence of constant values, the filter will repeatedly output zeros, as there's no "edge".

Similarly, if the signal has a zero (at time $n-1$) followed by a 1 (at time n), the filter will output a 1 (at time n), indicating an edge of "size" 1. The filter also encodes information about the edge's "direction" - if you instead had a 1 (at time $n-1$) followed by a 0 (at time n), the output (at time n) would be -1.

Because of this, the 1D edge detector can be thought of as the discrete-time equivalent of taking a derivative, an idea we'll explore in greater depth in part b of this question.

Q1a: Piecewise Constant Signals

We'll go through two examples with this filter. First, we'll try it out on a piecewise constant signal. We call a DT signal "piecewise constant" if it consists only of constant-height segments each spread over more than one sample. The first two signals below are piecewise constant; the third is not.



Question assigned to the following page: 1.1

Your Job

In the cell below:

- Using the time indices $\{0, 1, \ldots, 19, 20\}$(meaning we include the implicit zeros at $n = 0, 1, 2, 3, 4, 19, 20$), define the piecewise constant signal x as

$$x(n) = \sum_{k=5}^{9} \delta(n-k) + 3\sum_{k=10}^{13} \delta(n-k) + 2\sum_{k=14}^{18} \delta(n-k)$$

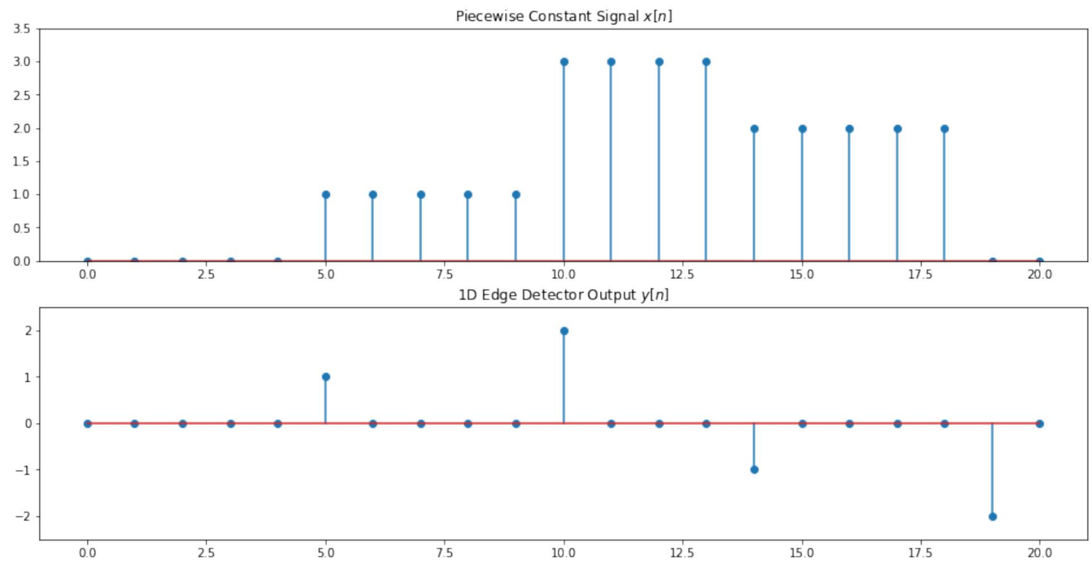
- Define the edge detector's impulse response h, but only at its nonzero points (so that the numpy array representing h contains two elements).
- Compute $y = x * h$ using "same" for the convolution mode.
- Finally, run the next cell to plot the results (plotting code has been provided for you).

In [205]:
```
# TODO your code here
h = np.array([1,-1])
x = list()
for i in range(21):
    if i <= 9 and i>=5:
        x.append(1)
    elif i > 9 and i < 14:
        x.append(3)
    elif i > 13 and i <= 18:
        x.append(2)
    else:
        x.append(0)
y = np.convolve(x, h, 'same')
n = np.linspace(0, 20, 21)
```

Question assigned to the following page: 1.1

In [206]:
```
# Plot results
plt.figure(figsize=(16, 8))

plt.subplot(2, 1, 1)
plt.stem(n, x)
plt.ylim([0, 3.5])
plt.title("Piecewise Constant Signal $x[n]$")

plt.subplot(2, 1, 2)
plt.stem(n, y)
plt.ylim([-2.5, 2.5])
plt.title("1D Edge Detector Output $y[n]$")

plt.show()
```

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:5:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will
be added as a LineCollection instead of individual lines. This signi
ficantly improves the performance of a stem plot. To remove this war
ning and switch to the new behaviour, set the "use_line_collection"
keyword argument to True.
  """
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:10:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will
be added as a LineCollection instead of individual lines. This signi
ficantly improves the performance of a stem plot. To remove this war
ning and switch to the new behaviour, set the "use_line_collection"
keyword argument to True.
  # Remove the CWD from sys.path while we load stuff.
```



Question assigned to the following page: 1.1

Q: x is a piecewise-constant signal, changing a total of four times: once from 0->1, then 1->3, then 3->2, and finally 2->0. How many points of the edge detector's output are nonzero (that is, how many edges are detected)?

A: 4 edges are detected as edges are the points which values change.

Q: One of the hottest areas of signal processing in the past ~15 years has been the study of sparse (mostly zero) signals, including both acquisition and representation, known as *compressed sensing*. A key part of compressed sensing algorithms is applying some *sparsifying transform* to signals that retain all the signal's information (i.e., the original signal could be completely recovered from the transformed one) but result in a new signal that is mostly zero.

Suppose you're interested in developing compressed sensing algorithms for piecewise constant signals, and are in need of a way to sparsify them. How would you do so by only using the first signal value, $x(0)$, and using an LTI filter of your choice as the sparsifying transform? Explain both what LTI filter you would use, and how to recover the original signal from the filtered one. You can ignore noise that would be present for real world signals — assume the signal truly is piecewise-constant like the ones above. Also, you may assume $x(n) = 0$ for $n < 0$, since this is basically what we're doing in the digital setup.

A: Since the signal is piecewise constant, we could use the 1D Edge Detector to detect any change from the original signal and generate a new signal that is mostly zero if the values of the signal are mostly the same. We could also recover the original signal from the new sginal from the output.
If the ouput value is zero, it means the values are the same at the point. If the output value is non-zero, it tells you how much the value decreases if it is negative, and how much the value increases if it is positive.

Question assigned to the following page: 1.2

Q1b: The Edge Detector as a DT Differentiator

The 1D edge detector is sometimes also referred to as a *moving difference* filter, since it operates by subtracting adjacent points. This has a particularly nice connection to the idea from calculus of taking a derivative of a function, and many properties of the 1D edge detector can be discovered by considering it as a discrete-time analogue of taking a signal's derivative.

Recall that given a function $f : \mathbb{R} \to \mathbb{R}$, the derivative of f is defined as

$$f'(t) = \lim_{\Delta t \to 0} \frac{f(t + \Delta t) - f(t)}{\Delta t}$$

and as we make Δt smaller and smaller, we get better and better approximations of $f'(t)$. But in DT, all arguments to our signals have to be integers, so Δt has to be an integer. The smallest positive nonzero value it can take, then, is 1, in which case we recover the formula for the 1D edge detector.

Equipped with this knowledge, we'll revisit some of the results you saw in calculus, but using DT signals analogous to some of the real-valued functions you're familiar with.

Your Job

In the cell below:

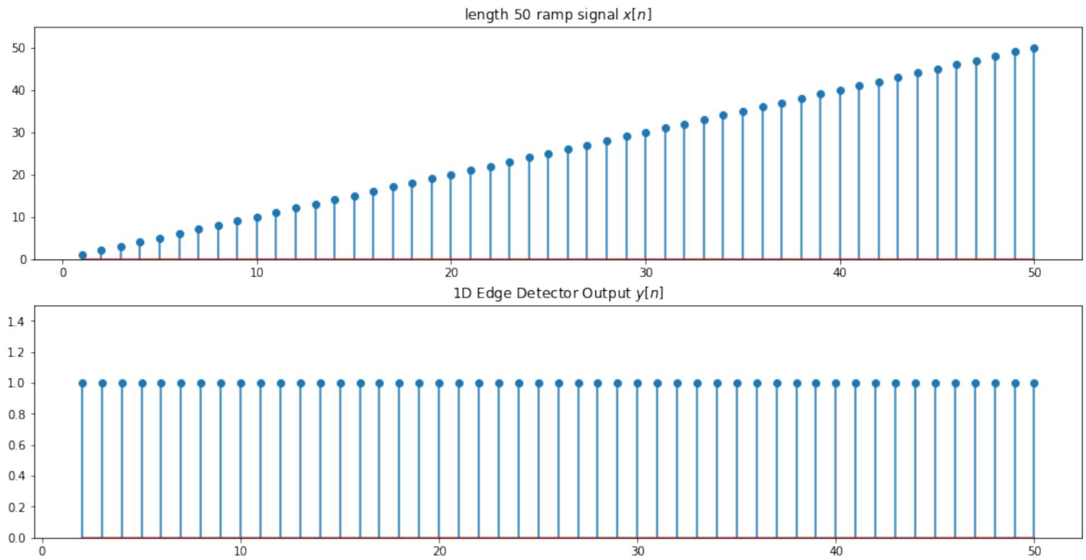
- Create a length 50 *ramp signal* x, defined as
$$x(n) = \sum_{k=1}^{50} k\delta(n-k) = \delta(n-1) + 2\delta(n-2) + 3\delta(n-3) + \ \ldots \ + 50\delta(n-50)$$
- Compute $y = x * h$ in "valid" mode (we only care about points of full overlap), where h is the same impulse response from Q1a.
- Plot x and y in separate figures, plot x on the top figure and y on the bottom one, being sure to:
  - **Use `plt.stem` to make stem plots**.
  - Give your plots reasonable titles.

Question assigned to the following page: 1.2

In [207]:
```
# TODO your code here
h = np.array([1,-1])
x = []
for i in range(50):
    x.append(i+1)
y = np.convolve(x, h, 'valid')
n = np.linspace(1, 50, 50)

plt.figure(figsize=(16, 8))

plt.subplot(2, 1, 1)
plt.stem(n, x)
plt.ylim([0, 55])
plt.title("length 50 ramp signal $x[n]$")
plt.subplot(2, 1, 2)
plt.stem(n[1:50], y)
plt.ylim([0 , 1.5])
plt.title("1D Edge Detector Output $y[n]$")
plt.show()
```

Question assigned to the following page: 1.2

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:12:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will
be added as a LineCollection instead of individual lines. This signi
ficantly improves the performance of a stem plot. To remove this war
ning and switch to the new behaviour, set the "use_line_collection"
keyword argument to True.
  if sys.path[0] == '':
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:16:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will
be added as a LineCollection instead of individual lines. This signi
ficantly improves the performance of a stem plot. To remove this war
ning and switch to the new behaviour, set the "use_line_collection"
keyword argument to True.
  app.launch_new_instance()
```



Q: What continuous function $f(t)$ does the DT signal $x(n)$, the input to our filter, remind you of?

A: $f(t) = t$

Q: What is its derivative $f'(t)$? Does the filter output $y(n)$ match?

A: The derivative $f'(t)$ = 1. Yes, it matches with the filter output *y(n)*

Let's do another! In the cell below:

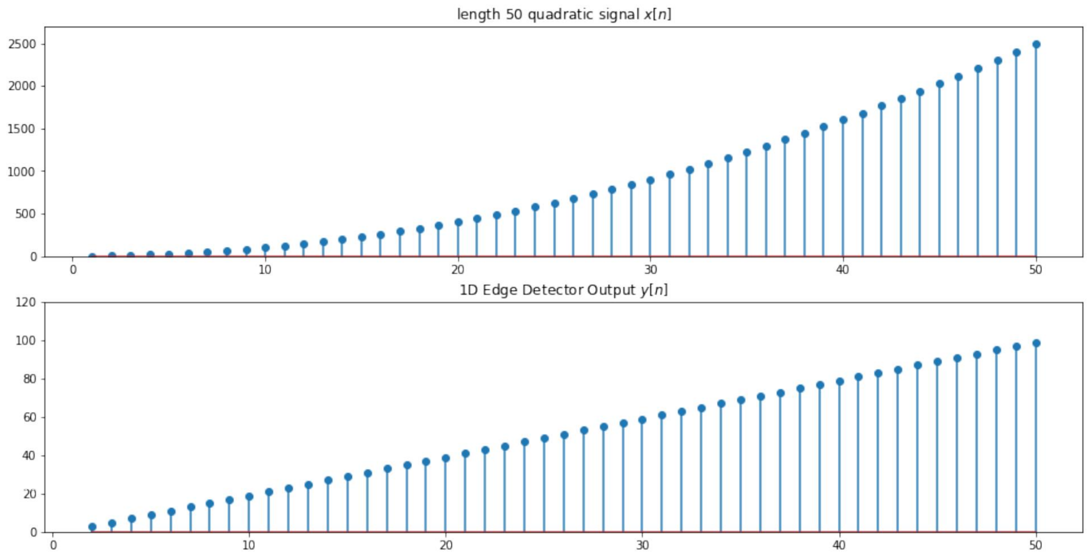
- Create a length 50 quadratic signal $x(n) = n^2$ for $n = 1, \ldots, 50$.
- Compute $y = x * h$, again in "valid" mode. Remember, h is the (unpadded, length 2) edge detector impulse response.
- Plot x and y the same as in the previous example, in separate figures with x above y.

In [208]:

```
# TODO your code here
h = np.array([1, -1])
x = []
for i in range(50):
    x.append((i+1)**2)
y = np.convolve(x, h, 'valid')
n = np.linspace(1, 50, 50)
plt.figure(figsize=(16, 8))
plt.subplot(2, 1, 1)
plt.stem(n, x)
plt.ylim([0, 2700])
plt.title("length 50 quadratic signal $x[n]$")
plt.subplot(2, 1, 2)
plt.stem(n[1:50], y)
plt.ylim([0, 120])
plt.title("1D Edge Detector Output $y[n]$")
plt.show()
```

Question assigned to the following page: 1.2

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:10:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will
be added as a LineCollection instead of individual lines. This signi
ficantly improves the performance of a stem plot. To remove this war
ning and switch to the new behaviour, set the "use_line_collection"
keyword argument to True.
  # Remove the CWD from sys.path while we load stuff.
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:14:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will
be added as a LineCollection instead of individual lines. This signi
ficantly improves the performance of a stem plot. To remove this war
ning and switch to the new behaviour, set the "use_line_collection"
keyword argument to True.
```



Q: What continuous function $f(t)$ does the DT signal $x(n)$, the input to our filter, remind you of?

A: $f(t) = t^2$

Q: What is its derivative $f'(t)$? Does the filter output $y(n)$ (roughly) match this shape? You can ignore any vertical offsets or rescaling differences.

A: The derivative $f'(t) = 2t$. Yes, it matches with the filter output *y(n)*

Question assigned to the following page: 1.2

Let's do one more example, but with a sine wave instead of another polynomial. In the cell below:

- Create a 101-point sine wave x, defined for $n = 0$ to $n = 100$ inclusive, with a period of 50 points. This corresponds to an angular frequency of $\omega_0 = 2\pi/50$, defined for you below, so that $x(n) = \sin(\omega_0 n)$ is the desired signal.
- Compute $y = x * h$, again in "valid" mode. Remember, h is the (unpadded, length 2) edge detector impulse response.
- Plot x and y the same as in the previous example, in separate figures with x above y.

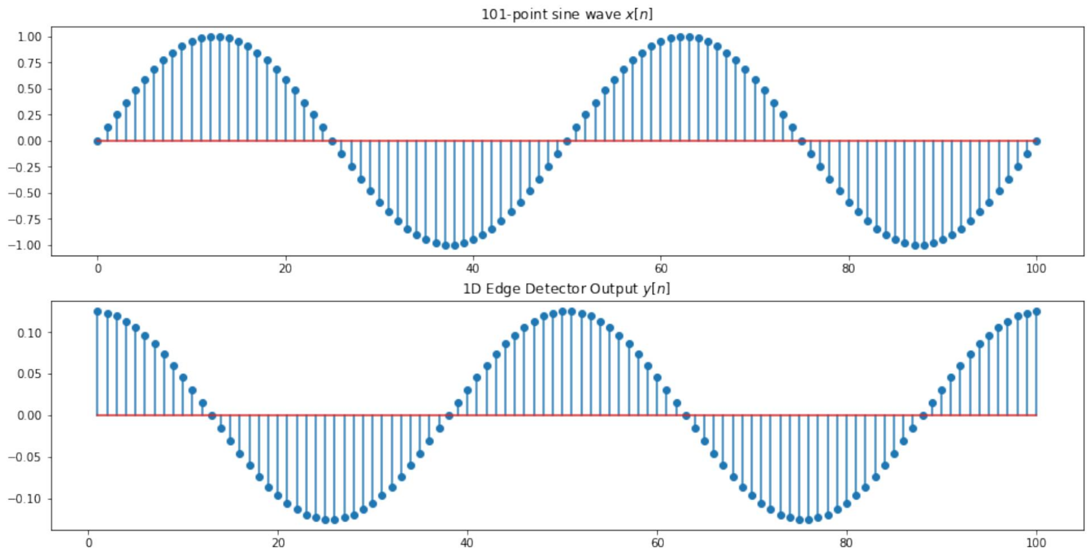
This one is a bit more interesting to analyze.

In [209]:

```
# TODO your code here

T = 50
w = 2 * np.pi / T
h = np.array([1, -1])
n = np.linspace(0, 100, 101)
x = np.zeros(101)
for i in range(101):
    x[i] = np.sin(w * i)
y = np.convolve(x, h, 'valid')
plt.figure(figsize=(16, 8))
plt.subplot(2, 1, 1)
plt.stem(n, x)
plt.title("101-point sine wave $x[n]$")
plt.subplot(2, 1, 2)
plt.stem(n[1:101], y)
plt.title("1D Edge Detector Output $y[n]$")
plt.show()
```

Question assigned to the following page: 1.2

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:13:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will
be added as a LineCollection instead of individual lines. This signi
ficantly improves the performance of a stem plot. To remove this war
ning and switch to the new behaviour, set the "use_line_collection"
keyword argument to True.
  del sys.path[0]
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:16:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will
be added as a LineCollection instead of individual lines. This signi
ficantly improves the performance of a stem plot. To remove this war
ning and switch to the new behaviour, set the "use_line_collection"
keyword argument to True.
  app.launch_new_instance()
```



Q: What continuous function $f(t)$ does the DT signal $x(n)$, the input to our filter, remind you of?

A: $f(t) = sin(2\pi/50 * t)$

Q: What is its derivative $f'(t)$? Does the filter output $y(n)$ match the *rough shape* of this derivative?

A: $f'(t) = 2\pi/50 * sin(2\pi/50 * t)$

Questions assigned to the following page: 1.2 and 1.3

Q: An important property of LTI systems — including our 1D edge detector — is that they cannot create new frequencies. Our input can be decomposed as

$$\sin(\omega_0 n) = \frac{e^{i\omega_0 n} - e^{-i\omega_0 n}}{2i}$$

and thus only contains the frequencies $\pm\omega_0$. Visually, does it appear that our output contains any new frequencies, or not? Don't overthink this.

A: It does not

Q1c: "Differentiating" Noise

So far, we've explored two important features of the edge detector:

1. As the name suggests, it finds the input's edges — points where the signal value changes.
2. It acts like a differentiator, returning an approximate derivative of the input signal.

Now, we'll analyze its behavior when given a more chaotic input, namely *Gaussian noise*.

Gaussian Noise

In general, signal processing tasks involve some information-bearing signal x that we want to somehow process. For example, x may be human speech representing a voice command given to Siri, which goes through some speech recognition algorithm so Siri can interpret and carry out the command.

However, we rarely (if ever) have access to x directly. Instead, we have to work with $\tilde{x} = x + z$, where z is another signal — typically thought of as being random — referred to as *noise*. This noise corrupts the information-bearing signal x. This presents an extra layer of complexity in real-world signal processing tasks: will our algorithms still work as intended when operating on the noised signal $\tilde{x}$ instead of the clean signal x?

The answer depends on z, of course, so we need a reasonable model for how the noise behaves. Typically, we assume z is *Gaussian noise*. Don't worry too much about what this means. The important takeaway is that we're generating a signal that will look like a series of values randomly jumping back and forth across some average value, typically taken to be zero.

Question assigned to the following page: 1.3

Your Job

The signal `noise` is defined for you below.

In [210]:
```
noise = np.random.normal(0, 5, 1000)
```

Now, add code to apply the (unpadded, i.e. the numpy array is length 2) edge detector to it, and store the result in `noise_filt` . Again, use "valid" as the convolution mode. Plotting code has been provided for you. You don't have to do much coding for this part.

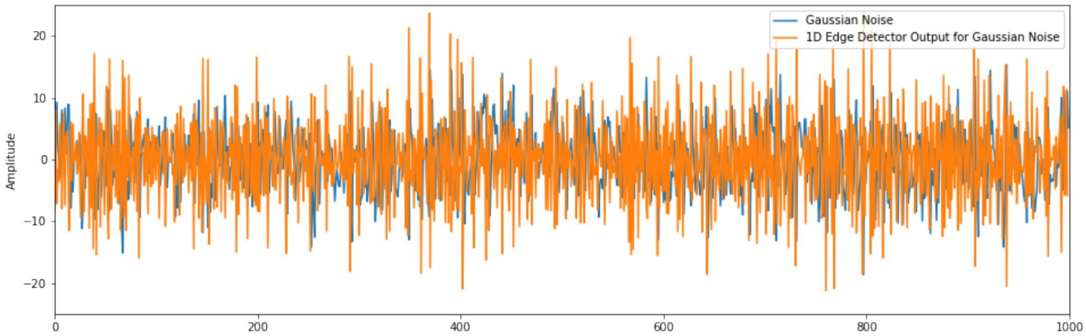
Note that we'll start using `plt.plot` from here on out. We'll be using longer signals (we generated 1000 samples of noise) and in addition to `plt.plot` being more efficient, it'll be much easier to visualize our results, as mentioned in "Background".

In [211]:
```
# TODO your filtering code here; store result in noise_filt
h = np.array([1, -1])
noise_filt = np.convolve(noise, h, 'valid')
```

The signals `noise_filt` and `noise` will not necessarily fit within the -25 to 25 y-limits we've set for the plot. Since it's random, it's possible we'll draw values that, either before or after filtering, will go outside these limits.

Question assigned to the following page: 1.3

In [212]:
```
# Plot results
plt.figure(figsize=(16, 5))
plt.plot(noise)
plt.plot(noise_filt)
plt.ylim([-25, 25])
plt.ylabel("Amplitude")
plt.legend(("Gaussian Noise", "1D Edge Detector Output for Gaussian No
ise"), loc="upper right")
plt.xlim([0,1000])
plt.show()
```



The plot should look kind of crazy. To help us understand what happened, we can turn to a salient summary statistic for random signals: *variance*.

You should see that both signals jump back and forth around zero. Each signal's variance will tell us, on average, how spread out these jumps are. Run the cell below to find the variance before and after applying the edge detector.

In [213]:
```
print("Noise variance before edge detector: {}".format(round(np.var(no
ise), 2)))
print("Noise variance after edge detector:  {}".format(round(np.var(no
ise_filt), 2)))
```

```
Noise variance before edge detector: 27.66
Noise variance after edge detector:  55.48
```

Q: What happens to the noise variance after we apply the edge detector? Does it increase or decrease, and by how much (e.g., does it double, triple, get cut in half, etc.)?

A: The noise variance increases more than twice its current number

Questions assigned to the following page: 2.1 and 1.3

Q: Qualitatively, and taking into account the variance calculation, does the edge detector amplify or suppress the "strength" of the noise? What implications might this have if we want to detect edges on a signal that has been corrupted by a large amount of noise?

A: The edge detector amplifies the strngth of the noise becasue if the noise is large enough to dominate the corrupted signal, then the changing might be due to noises instead of the real signal.

Q2: Data Smoothing

One of the most common uses of LTI filters is in data smoothing. The applications are numerous, ranging from noise reduction, to extracting trends from complex data, to interpolation, and more. In this question, we'll explore the simplest, yet perhaps most widely used, data smoothing method in a few of these application spaces: a moving average filter.

Q2a: Noise Reduction

The *simple* moving average filter is specified in terms of a single integer parameter, L, which represents the length of the filter. The filter takes the average of the L points before, and including, the current point of a signal, outputting that average. Formally, the filter's impulse response is:

$$h_{SMA}(n) = \frac{\delta(n) + \delta(n-1)\ +\ \dots\ +\ \delta(n-(L-1))}{L}$$

You probably recognize the impulse response, as it's just a length L rect normalized to sum to 1. Note that the definition we'll use here is for the *causal* moving average filter: the output at any point in time is only computed as an average of the current and previous signal values. Causal systems can not "see the future" and are always at rest until a signal arrives (i.e their impulse response is 0 for $n < 0$). This is, of course, very important in real-time systems since in real-time systems, we don't have access to the future.

The subscript "SMA" in the impulse response definition stands for *Simple Moving Average*, meaning all points are given equal weight in computing the average. This distinguishes the filter from more elaborate moving averages, such as the *Exponential Moving Average* (EMA) which gives more recent data points a higher weight. You'll get a chance to explore the EMA later in this Lab. We encourage you to check out reference [1] for more on the theory behind the use of moving average filters for noise reduction if interested.

Question assigned to the following page: 2.1

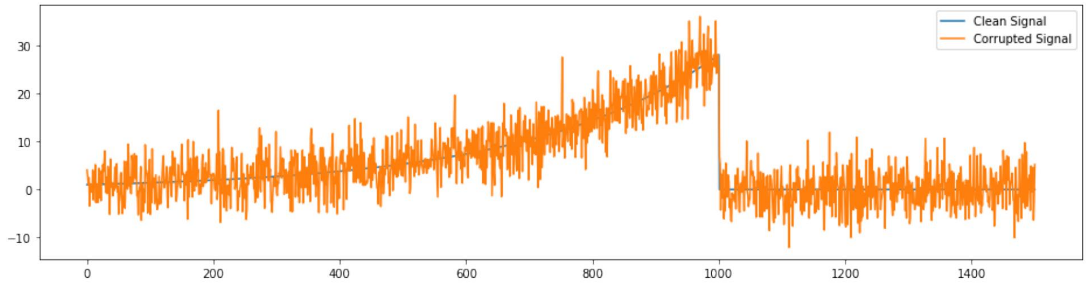
Your Job: Signal Generation

In the cell below, do the following:

- Generate the time indices $\{0, 1, \ldots, 999, 1000\}$.
- Generate the signal $x[n] = e^{n/300}$ over these time indices.
- Append 500 zeros onto the end of x, and extend your time indices to account for these extra data points. That is, your time indices should now be $\{0, 1, \ldots, 1000, 1001, \ldots, 1500\}$
- Generate noise z of the same dimensions as x, by using np.random.normal (https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.normal.html) with the parameters `loc=0, scale=4, size=np.shape(x)` .
  - As a reminder from Lab 1 Q1, `np.shape(x)` returns how many entries `x` has along each dimension (here, our signal is 1D, so `np.shape` just returns the number of entries), which we can pass into a numpy function to easily generate numpy arrays (i.e. signals) of the same size as some other numpy array.
- Create the noised signal $y = x + z$.
- On the same 16x4 figure, plot x and y. Using plt.legend (https://matplotlib.org/3.1.0/api/_as_gen/matplotlib.pyplot.legend.html), label x as "True Signal" and y as "Noised Signal". **Make sure you use `plt.plot` , *not* `plt.stem` , for plotting.** You should use the time indices you generated as the first argument to `plt.plot` .

Question assigned to the following page: 2.1

In [214]:
```
# TODO your code here
n = np.linspace(0, 1500, 1501)
x1 = []
x2 = np.zeros(500)
for i in range(1001):
    x1.append(np.exp(i/300))
x = np.concatenate((x1, x2))
z = np.random.normal(0, 4, np.shape(x))
y = x+z

plt.figure(figsize=(16, 4))
plt.plot(x, label='Clean Signal')
plt.plot(y, label='Corrupted Signal')
plt.legend()
plt.show()
```



If your code is correct, you should see an exponential that rises in amplitude from 0 to ~30 over the course of the first 1000 samples followed by 500 zeros. Overlayed on top of it, you'll see a signal that jumps up and down, **but, on average, follows the sample amplitude as the original signal.**

Your Job: Signal Denoising

Now that we've got a (not so) nice and noised up signal, let's try to denoise it with a moving average. Intuitively, since the noised signal follows, on average, the original one, we should be able to reduce the noise a bit by averaging adjacent data points. The hard part, and a great example of an engineering tradeoff, is figuring out the right number of points to average at a time.

The array `filt_sizes` containing the different sizes (the parameter L from above) we'll try out has been defined for you. In the cell below:

- Create a 20x35 figure. We will be creating a column of subplots, one for each filter length (note `len(filt_sizes)` is 7).
- For each filter size in `filt_sizes` ,
    - Create a simple moving average filter, h, of that size.

Question assigned to the following page: 2.1

    - Don't bother doing any zero padding of h. Instead, just construct the filter for all points where it is nonzero.
  - Compute $\hat{x} = y * h$, where y is the noised growing exponential signal you created above. Naming this variable `x_hat` in your code is fine. **Use "full" as the convolution mode.** If we don't use "full", the filter will be non-causal due to the way we defined the impulse response.
  - On a new subplot (that is, for each different moving average filter, we're using a separate subplot):
    - Plot x, the original (noiseless) signal. You can use the variable n from above for the time indices.
    - Plot $\hat{x}$, the moving averaged version of y you computed in the current loop iteration.
      - Since we didn't use "same" for the convolution (motivated by the desire to have a causal filter), we need an expanded set of time indices to plot with.
      - As a freebie, the code for this is `n_aug = np.concatenate((n, np.arange(n[-1], n[-1] + (len(x_hat) - len(n)))))` . This just adds on the extra indices to the end of the existing ones based on how much the convolution stretches the signal x by.
    - Use `plt.legend` to label x as "True Signal" and $\hat{x}$ as "Noised Signal after ?-point SMA", where "?" should be replaced by the current filter size. Python's [format (https://www.digitalocean.com/community/tutorials/how-to-use-string-formatters-in-python-3)](https://www.digitalocean.com/community/tutorials/how-to-use-string-formatters-in-python-3) function may be of use here.

Note: $\hat{x}$ and h change with each new filter size; x and y do not.

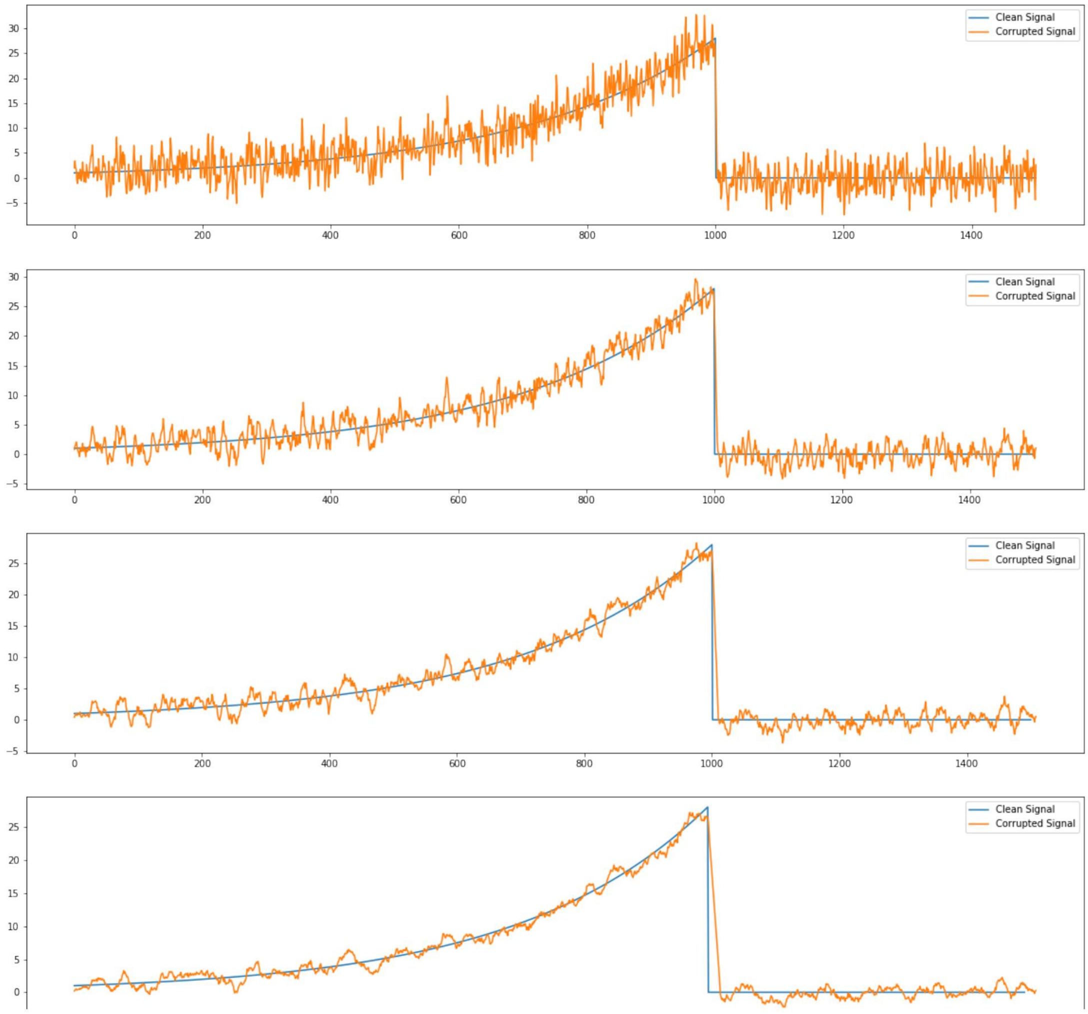
Hint 1: It will be most informative to display our subplots as a column. Thus, your call to `plt.subplot` should look something like `plt.subplot(len(filt_sizes), 1, i)` where `i` is a variable tracking which subplot we're currently on (i.e., it starts at 1, and should increase by 1 each time we create and apply a different moving average filter).

Hint 2: Make sure to call `plt.show()` *after* you've created all your subplots (i.e., it should NOT be called inside a loop). You only call `plt.show()` once per figure, i.e. after you're finished generating all the subplots.

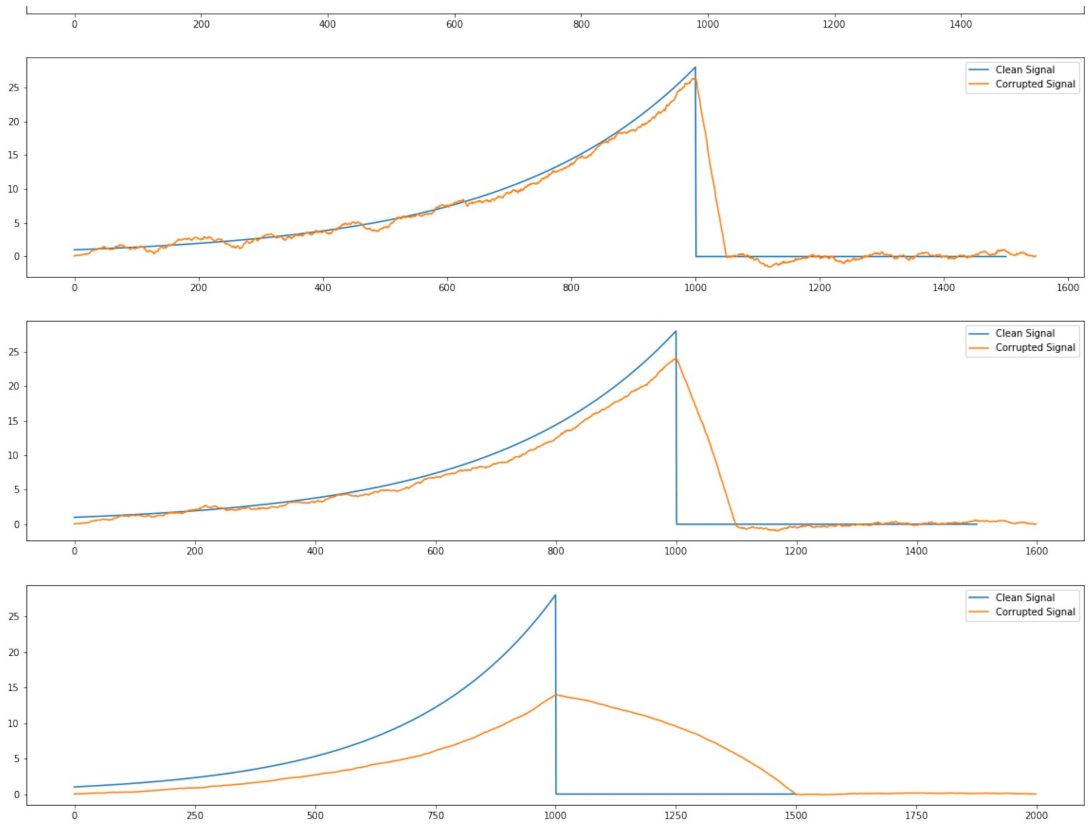
In [215]:
```
filt_sizes = [2, 5, 10, 20, 50, 100, 500]
```

Question assigned to the following page: 2.1

In [216]:
```
# TODO your code here
plt.figure(figsize=(20, 35))
n = np.linspace(0, 1500, 1501)

for i in range(len(filt_sizes)):
    j = filt_sizes[i]
    h = np.ones(j)/j
    x_hat = np.convolve(y, h, 'full')
    m = np.concatenate((n, np.arange(n[-1], n[-1] + (len(x_hat) - len(
n)))))

    plt.subplot(7, 1, i + 1)
    plt.plot(n, x, label='Clean Signal')
    plt.plot(m, x_hat, label='Corrupted Signal')
    plt.legend()

plt.show()
```



Question assigned to the following page: 2.1



A few sanity checks:

- In all cases, the true signal should go from 0 to 1500 and look the same.
- As we increase the filter length, the noised signal (post-filtering) will be stretched further and further out, occupying just over 1500 points for the length 2 and length 5 filters, and about 2000 points for the length 500 filter.
- The peak of the noised signal (post-filtering) should always line up with the true signal's peak. This is a good sanity check that the filter is causal: if it doesn't look ahead, then the filter's first encounter with the peak will be averaging the peak with the previous (positive) values, resulting in a peak in the output. After the filter sees the true signal's peak, it only sees zeros, so averaging in more and more zeros instead of the positive values on the exponential will lead to smaller and smaller output values.

Analyzing the plots

Question assigned to the following page: 2.1

Q: As we increase the filter length, do the first 400-500 points of the filtered signal get smoothed out (reducing the noise at these points) or blown up (amplifying the noise)? Ignore any scaling differences; that is, if the filtered signal looks the same as the true one except for being off by a constant scaling factor, that's fine.

A: the first 400-500 points of the filtered signal get smoothed out

Q: As we increase the filter length, what happens to our signal's sharp, high-frequency feature, the drop off from the top of the exponential back down to zero (at $n = 1000$)? Is this high-frequency feature preserved, or does it get more and more distorted? Explain, based on the moving average filter, why this makes sense.

A: The sharp is getting smoother and the amplitude is getting smaller. The feature is also getting more and more distorted. The moving average filter will include more and more zero as it increases which will change the average total signal.

Q: Putting your previous answers together, what advantage is there to using a longer moving average for noise reduction? What do we have to trade off (i.e., what disadvantage becomes more and more pronounced) in doing so?

A: Using a longer moving average will reduce noise and get a smoother signal; however, the disadvantage becomes more and more pronounced are less accurate signal and no high-frequecy feature.

Q: Assuming you equally prioritize minimizing distortion of sharp, high-frequency features while still getting a reasonable amount of noise reduction, which of the following moving average filter lengths would you pick from above for denoising this specific signal? There are multiple correct answers; just be sure to justify your choice.

A: The 50-point SMA filter is a good choice because it will keep the high-frequency feature and also reduces quite a bif of noise.

Question assigned to the following page: 2.2

Q2b: Extracting Trends from Data

In addition to its use in signal processing and statistics for noise reduction, the moving average filter is popular in analysis of financial data for highlighting trends in stock prices.

Here, we'll analyze one of the most common datasets in time series analysis: stock price data! Run the cell below to load it. We will be taking a look at Apple's stock data from mid 2017 to early 2019.

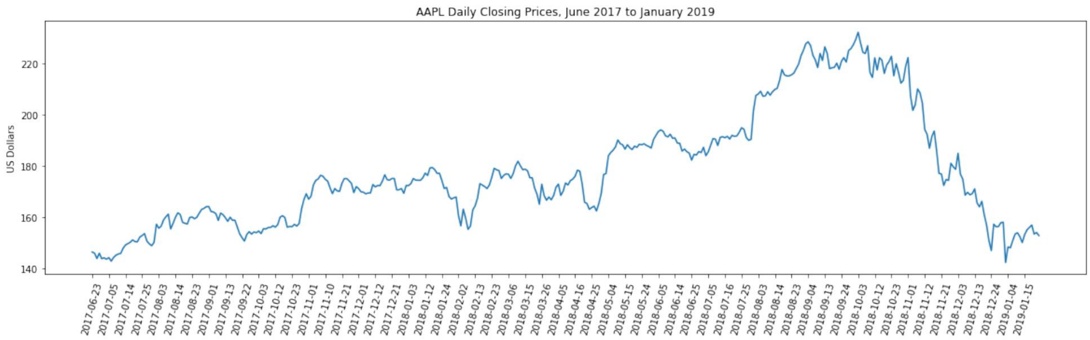
We got this data from Yahoo! finance. If you are interested in playing around with stock data on your own, you can click on a stock, navigate to the historical data tab, download a csv file, and use our code below to parse it. Acquiring data often plays second fiddle to all the fancy algorithms used on it despite being just as important a part of any engineering field that relies on it. However, here, we want to focus on the algorithms, not the minutiae of Yahoo! csv file formats, hence why we provide the code to read the data in.

In [217]:

```
# CSV hocus pocus
import csv

stock_dates = []
stock_prices  = []
with open('AAPL.csv', mode='r') as raw_data:
    csv_reader = csv.DictReader(raw_data)
    for row in csv_reader:
        data = row['Close']
        if not data == 'null':
            stock_prices.append(float(data))
            stock_dates.append(row['Date'])
stock_prices = np.array(stock_prices)
```

Question assigned to the following page: 2.2

In [218]:
```
# 400 most recent days
start = -400
end = -1
x = np.arange(len(stock_prices[start:end]))

# Plot roughly one data point per week (x[::7]) so we can display date
s
# without matplotlib going crazy about the labels overlapping
plt.figure(figsize=(20, 5))
plt.xticks(x[::7], stock_dates[start:end:7], fontsize=10, rotation=75)
plt.plot(x, stock_prices[start:end])
plt.title("AAPL Daily Closing Prices, June 2017 to January 2019")
plt.ylabel("US Dollars")
plt.show()
```



Your Task

Your job is to fill in the missing parts of the cell below to filter the noise with moving average filters of length 5, 25, and 75, and be sure to answer the question about interpreting the results below the cell. Unlike Q2a, most of the work has been done for you here.

In the cell below:

- Define the moving average impulse responses as `MA5, MA25, MA75` respectively.
  - Again, don't bother with any zero padding - just define them at their nonzero points.
- Filter your test data (`data`) with each of them using **convolution with the "same" mode**. We'll call the outputs `y5, y25` , and `y75` , respectively.

Question assigned to the following page: 2.2

Note: **Here, we are *not* using causal moving averages as we did in Q2a.** The motivation in doing so is that we want the filtered signals to temporally align with the original to make the results easier to interpret. While real-time filters have to be causal, we're working in an offline setting with pre-collected data here, so causality is less important, and interpretability is more useful. These properties are some of the tradeoffs involved in how data processing is done.

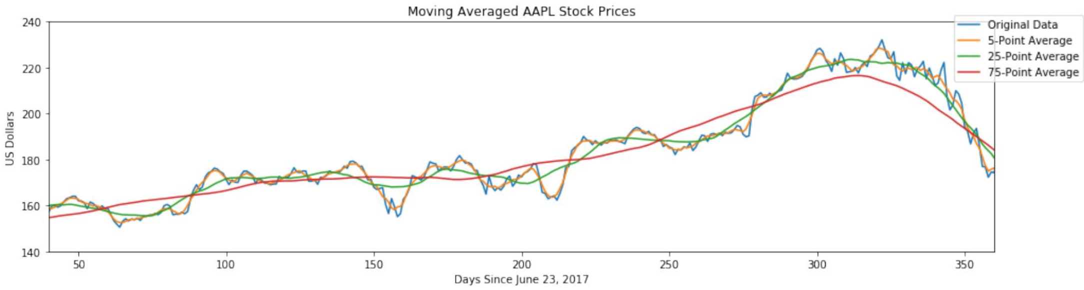
Plotting code has been provided for you. To generate the results, simply run the cell after adding your own code.

In [219]:
```
data = stock_prices[start:end]
```

In [220]:
```
## TODO your impulse response definitions here
MA5 = np.ones(5)/5
MA25 = np.ones(25)/25
MA75 = np.ones(75)/75
```

In [221]:
```
## TODO your filtering code here
y5 = np.convolve(data, MA5, 'same')
y25 = np.convolve(data, MA25, 'same')
y75 = np.convolve(data, MA75, 'same')
```

In [222]:
```
# Overlay stock prices
plt.figure(figsize=(16, 4))
plt.plot(np.arange(len(data)), data)
plt.plot(np.arange(len(data)), y5)
plt.plot(np.arange(len(data)), y25)
plt.plot(np.arange(len(data)), y75)

# Formatting mumbo jumbo to abstract away boundary issues
plt.xlim([40, 360])
plt.ylim([140, 240])
plt.legend(('Original Data', '5-Point Average', '25-Point Average', '7
5-Point Average'), bbox_to_anchor=(1.1, 1.05))
plt.ylabel("US Dollars")
plt.xlabel("Days Since June 23, 2017")
plt.title("Moving Averaged AAPL Stock Prices")
plt.show()
```



Q: As we use longer and longer moving average filters to process our signal (the stock data), does the filtered signal highlight longer term trends, or shorter term trends? Explain in 1-2 sentences.

A: It will highlight longer term trends as it takes more data and average the data, and hence it has the ability to reduce sudden change in the signal.

Q: Suppose that you had the closing price of Microsoft stock (MSFT) for every day it's existed, 1986 to present (~8000-10000 data points, one per day), and wanted to see how the company's stock price changed at the level of trends that occur over the course of a year using a moving average filter. How long of a moving average would you use? Why?

A: A 365 point moving average filter is a good choice because we could take daily data point and also average the data to get the trend for a year.

Question assigned to the following page: 3

Q3: The MACD Indicator

The Moving Average Convergence Divergence (MACD) Indicator is a trend-following momentum indicator that shows the relationship between two moving averages of a stock's price. In this question, we'll use the MACD indicator as a vehicle for introducing the exponential moving average as well as some more uses of signal processing in financial analysis.

The Exponential Moving Average

To calculate the MACD of a stock, we first have to understand a new kind of moving average, called the *Exponential Moving Average* (EMA). What we did earlier is known as an Simple Moving Average (SMA) - all points are given equal weight. An EMA, on the other hand, places a greater weight, and therefore significance, on the most recent data points. The benefit of the EMA over the SMA is that the EMA reacts faster to recent price changes.

Derivation

Let's take a look at where the EMA comes from. We can express the EMA using the following recursive Linear Constant Coefficient Difference Equation (LCCDE):

$$y(n) = \alpha \cdot x(n) + (1-\alpha) \cdot y(n-1)$$

where $y(n)$ is the filter output at day n, and $x(n)$ is the stock price at day n. The output, $y(n)$, is a linear combination of the previous output of the filter, $y(n-1)$, and the current signal value, $x(n)$.

Now, why is this an "exponential" moving average? It might not be obvious when written it in this form. Let's write it out step by step to try to unravel the recursion. We'll assume a causal filter, and that the input signal x is zero for all $n < 0$, so that $n = 0$ represents the first point of stock data we have, with others filled in as zero.

Since we have a causal filter and $x(n) = 0$ for all $n < 0$, we know that $y(n) = 0$ for all $n < 0$, and:

$$\begin{aligned}
y(0) &= \alpha \cdot x(0) + (1-\alpha) \cdot y(-1) = \alpha \cdot x(0) \\
y(1) &= \alpha \cdot x(1) + (1-\alpha) \cdot y(0) = \alpha \cdot x(1) + (1-\alpha) \cdot \alpha \cdot x(0) \\
y(2) &= \alpha \cdot x(2) + (1-\alpha) \cdot y(1) = \alpha \cdot x(2) + (1-\alpha) \cdot (\alpha \cdot x(1) + (1-\alpha) \cdot \alpha \cdot x(0)) = \alpha \cdot x(2) + (1 - \\
&\vdots \\
y(n) &= \alpha \sum_{k=0}^{n} (1-\alpha)^k \cdot x(n-k)
\end{aligned}$$

Question assigned to the following page: 3

Aha! Now that we have rewritten it, it's clear why we call this an EMA: we are weighting each previous data point with a value that decreases exponentially as we go further and further back in time.

An LTI Perspective: The Impulse Response of an EMA Filter

We can input a Kronecker delta to the EMA system by setting $x(n) = \delta(n)$ to find its impulse response:

$$h(n) = \alpha \sum_{k=0}^{n} (1-\alpha)^k \cdot \delta(n-k) = \alpha(1-\alpha)^n u(n)$$

which is just what we would expect: a (one-sided) decaying exponential!

Your Job

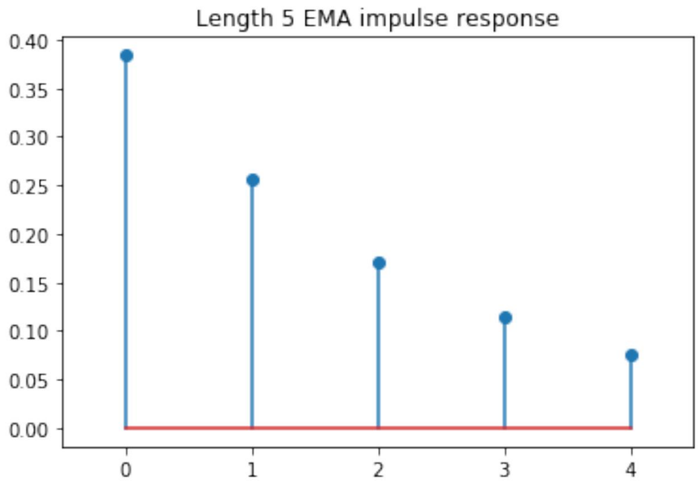
Fill in the `ema_filter` function below to create and return the impulse response of an EMA filter truncated after `length` points. The value of α has been determined for you already. **Make sure you normalize the impulse response to sum to 1.**

In [223]:
```
def ema_filter(length):
    alpha = 2/(length+1)
    # TODO: Create and return an EMA filter with length "length"
    h = np.ones(length)/length
    tot = 0

    for i in range(length):
        h[i] = alpha*(1-alpha)**i
        tot += alpha*(1-alpha)**i

    return h/tot
```

Question assigned to the following page: 3

In [224]:
```
# Run me to plot!
h = ema_filter(5)
plt.figure()
plt.title("Length 5 EMA impulse response")
plt.xlim([-.5, 4.5])
plt.stem(h)
plt.show()
```

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:6:
UserWarning: In Matplotlib 3.3 individual lines on a stem plot will
be added as a LineCollection instead of individual lines. This signi
ficantly improves the performance of a stem plot. To remove this war
ning and switch to the new behaviour, set the "use_line_collection"
keyword argument to True.
```



We can now convolve data with the impulse response to calculate the EMA! Note that if we flipped this signal (as we'd do in convolving) and slid it across some data signal, pointwise multiplying and summing to calculate our EMA, the strongest point would always be at the front, with an exponential drop-off as we go further and further back along the impulse response.

2 Quick Things to Note:

1. The LCCDE we originally gave describes an IIR filter, but we are using an FIR filter - after all, we can't store infinitely many values on a computer. In order to account for this, we truncate as is typically done, but we also renormalize so that the sum of the impulse response's coefficients, known as the DC gain, is 1.
2. The value for α that we have picked is motivated by reducing the output noise variance. We have a provided a reference below if you wish to read more about this.

Question assigned to the following page: 3

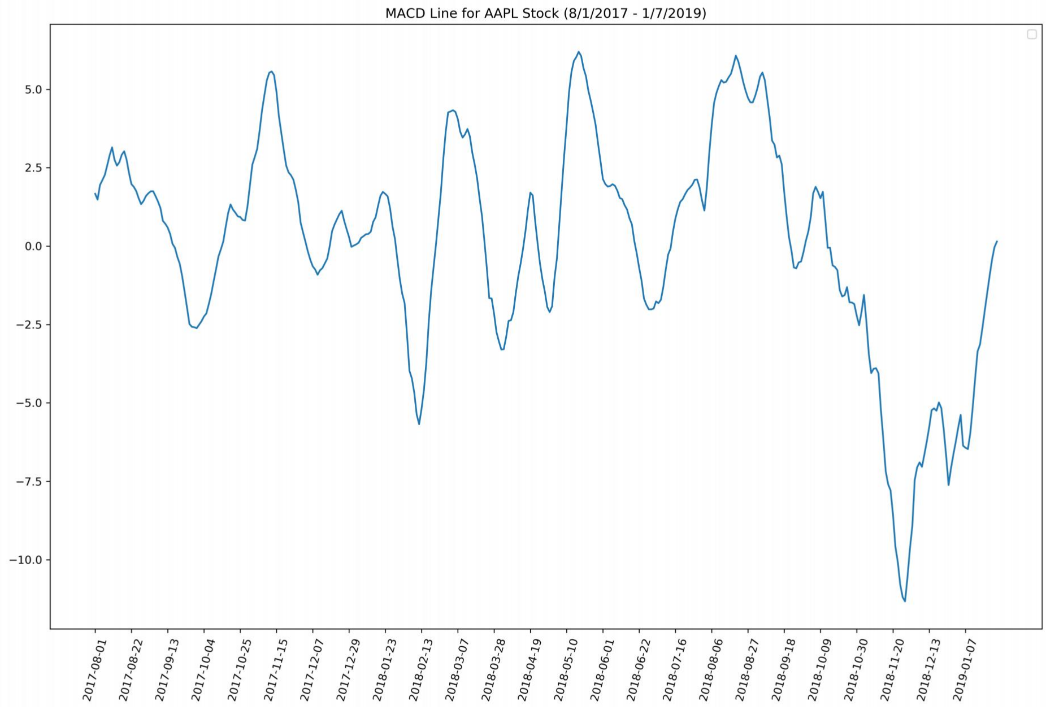
The MACD Line

The *MACD line* is calculated by taking the difference between the 26-day EMA of the stock and its 12-day EMA. In the cell below, we've defined an array of values, `data`, for you based on the same AAPL stock data we used earlier in this lab. In the cell below, calculate the MACD line. To do so,

- Create a length 26 EMA filter, h_{26}.
- Create a length 12 EMA filter, h_{12}.
- Calculate the 26-day EMA as $y_{26} = x * h_{26}$, where x is `data`. **Use "valid" as the convolution mode** - we only want points where our signals fully overlap.
- Calculate the 12-day EMA as $y_{12} = x * h_{12}$. Again, use "valid" as the convolution mode.
- Crop y_{12} by discarding its first 14 values, so that y_{26} and y_{12} have the same length.
  - The first value in y_{26} is an EMA taken over data points 1 through 26 (since the first point the signals fully overlap is at day 26), representing the exponential average at day 26, factoring in the previous 25 days. By cropping out the first 14 values of y_{12}, we ensure the first point in its output also corresponds to an EMA for day 26, but taken over the most recent 12 days rather than 26, essentially "aligning" the outputs.
- Compute the MACD line as $y_{12} - y_{26}$. **Store your result in a variable called `MACD`, as our plotting code uses this.**

After you finish, run the next cell to plot the results; here's the plot from the staff solution so you're able to check that your answer is correct before moving on:

Question assigned to the following page: 3

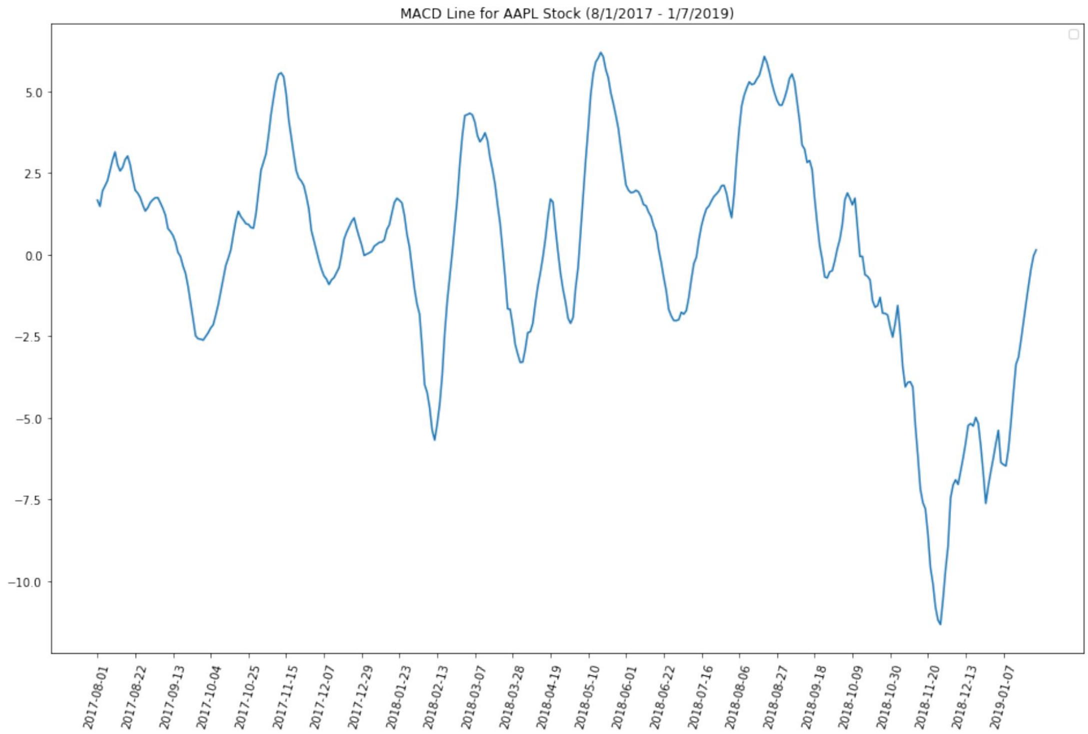


In [225]:
```
# Stock data
data = stock_prices[start:end]

# TODO calculate macd line
h26 = ema_filter(26)
h12 = ema_filter(12)

y26 = np.convolve(data, h26, 'valid')
y12 = np.convolve(data, h12, 'valid')[14:]

MACD = y12- y26
```

Question assigned to the following page: 3

In [226]:
```
# Plotting Code
x = np.arange(len(MACD))
fig = plt.figure(figsize=(16, 10))
plt.xticks(x[::15], stock_dates[start+26:end:15], fontsize=10, rotatio
n=75)
plt.plot(MACD)
plt.title("MACD Line for AAPL Stock (8/1/2017 - 1/7/2019)")
plt.legend()
plt.show()
```

```
No handles with labels found to put in legend.
```



The Signal Line

Now that we have the MACD line, we want to take the 9-day EMA of the MACD line to create the *signal line*. Once equipped with the MACD line and the signal line, we can analyze the stock data.

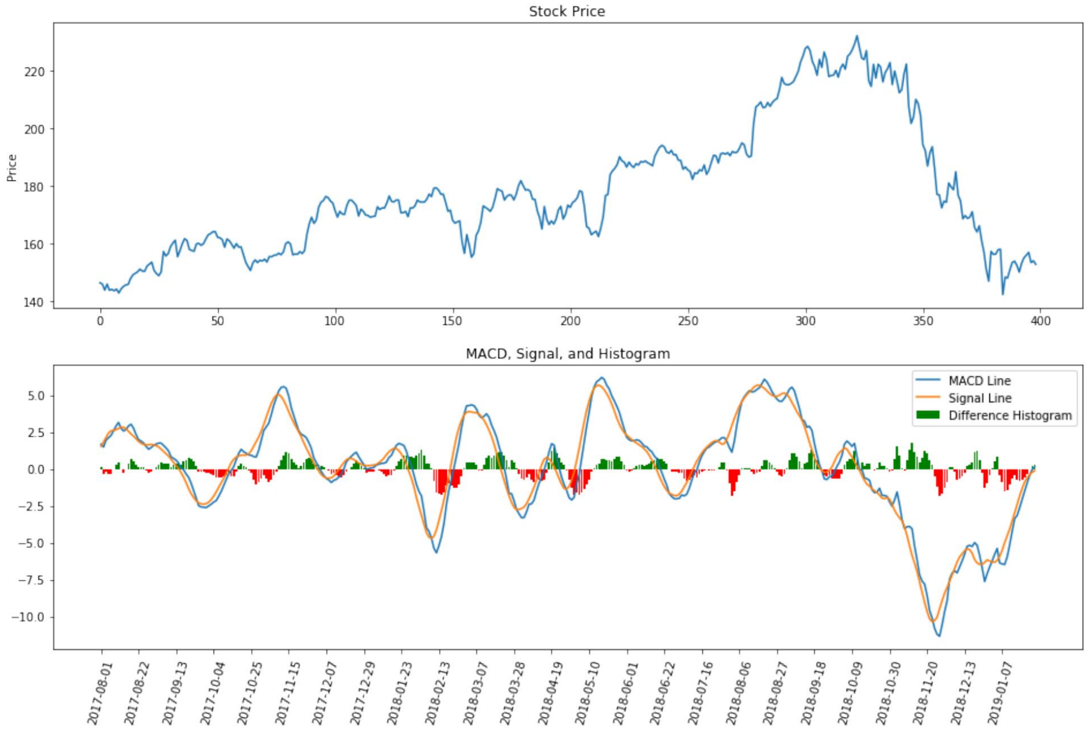
Add code in the top of the cell below to calculate the signal line by applying a 9-day EMA filter (use "same" as the convolution mode this time) to your MACD line, `MACD` , from above. Plotting code has been provided for you. **Store the result in a variable called `signal` , as this is what our plotting code uses.**

Question assigned to the following page: 3

In [227]:
```
# TODO your signal line calculation here
EMA_9Day = ema_filter(9)
signal = np.convolve(MACD, EMA_9Day, mode='same')
```

In [228]:
```
# Plotting Code
c = ['green', 'red']
colors = [c[bool(i)] for i in np.greater(signal, MACD)]
x = np.arange(len(signal))

plt.figure(figsize=(16,10))

plt.subplot(2,1,1)
plt.title("Stock Price")
plt.plot(data)
plt.ylabel("Price")

plt.subplot(2,1,2)
plt.title("MACD, Signal, and Histogram")
plt.xticks(x[::15], stock_dates[start+26:end:15], fontsize=10, rotatio
n=75)
plt.plot(MACD, label='MACD Line')
plt.plot(signal, label='Signal Line')
plt.bar(range(len(signal)),(MACD-signal), color=colors, label="Differe
nce Histogram")
plt.legend()
plt.show()
```

Question assigned to the following page: 3



If done correctly, you should see that the signal line resembles a smoothed, shifted (slightly to the left) version of the MACD line, and the difference histogram is green where the MACD Line is above the signal line, and red where the signal line is above the MACD line.

Interpretation of the MACD Indicator (from *Technical Analysis* [3])

In signal processing terms, the MACD is a filtered measure of velocity. The velocity has been passed through two first-order linear low-pass filters (the EMA filters). The signal line is that resulting velocity, filtered again. The difference between those two, the histogram, is a measure of the acceleration, with all three filters applied. A MACD crossover of the signal line indicates that the direction of the acceleration is changing. The MACD line crossing zero suggests that the average velocity is changing direction.

Question assigned to the following page: 3

Q: Using this analogy of velocity and acceleration, we can think of a stock's price as a the position of the car. We know its velocity and acceleration from the MACD and histogram, respectively. What do points where the difference histogram is positive tell us about how the stock is behaving, and how can we use this information to decide whether or not to invest in a stock? How about when the difference histogram is negative?

A: It seems that the difference histogram line represents the difference between the sginal line and MACD line. When the histogram line is positive, it means the MACD line is above the signal line which tells us that the stock price is going up, and the higher the green bar the faster the acceleration. When the histogram line is negative, it means the MACD line is below the signal line which tells us that the stock price is going down, and the higher the red bar downward the faster the deceleration.

References

[1] *The Scientist and Engineer's Guide to Digital Signal Processing. Chapter 15, Moving Average Filters.* [Link (https://www.analog.com/media/en/technical-documentation/dsp-book/dsp_book_Ch15.pdf)](https://www.analog.com/media/en/technical-documentation/dsp-book/dsp_book_Ch15.pdf)
[2] *Moving Average Convergence Divergence - Investopedia.* [Link (https://www.investopedia.com/terms/m/macd.asp)](https://www.investopedia.com/terms/m/macd.asp)
[3] *Technical Analysis*. [Link (http://www.mrao.cam.ac.uk/~mph/Technical_Analysis.pdf)](http://www.mrao.cam.ac.uk/~mph/Technical_Analysis.pdf)
[4] *Systems and Control Theory - Exponential Moving Average.* [Link (https://tttapa.github.io/Pages/Mathematics/Systems-and-Control-Theory/Digital-filters/Exponential%20Moving%20Average/Exponential-Moving-Average.html)](https://tttapa.github.io/Pages/Mathematics/Systems-and-Control-Theory/Digital-filters/Exponential%20Moving%20Average/Exponential-Moving-Average.html)