Networks Assignment 4

Synchronization and MAC protocols

Made By: Mohamed Said Ibrahem.

ID: 57

CS431: Computer Networks and Communications

Assignment 4: Synchronization and MAC protocols

In class we've learned about different MAC (medium access control) protocols that determine when devices get to transmit (these protocols *control access* to the *medium*). For all but the simplest MAC protocols, the participating devices need to be synchronized in time. The way this is done in practice is to have one node transmit a signal (called a "preamble" or "beacon") and the other nodes will listen for this signal. Since wireless signals travel at the speed of light, all nodes can agree that they will hear the preamble at nearly exactly the same time.

First, we will explore how nodes can listen for a specific preamble to achieve time synchronization. Then, we will wrap up with a Monte Carlo simulation of the capacities of the slotted-ALOHA MAC protocol.

```
[ ] # By Mohamed Said (https://github.com/mohamed-said-ibrahem)

    # For more info about MAC https://www.youtube.com/watch?v=509AmnKtx-c
    # Plotting library for the Python programming language and its numerical mathematics extension NumPy.
    # It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK+.
    %matplotlib inline
    # Libraries Imports.
    from matplotlib import pyplot as plt
    import random
    import numpy as np
```

Part 1: Correlation and Preambles

A preamble is a waveform sent at the beginning of a packet that is used to indicate the start of a packet. The preamble waveform is agreed to ahead-of-time, so the signal carries no data. Receivers will listen for the preamble, and when it's detected the receivers will start to demodulate the rest of the packet. Preambles can also be used to synchronize multiple clients for MAC protocols that require synchronization.

In this part, we will explore how to detect and synchronize to a preamble. Both detection and synchronization can be done using a **correlation**. A correlation of two discrete signals x and y is an infinite-length discrete signal which is defined as

$$(x \star y)[k] = \sum_{n=-\infty}^{\infty} x^*[n] \cdot y[n+k]$$

In other words, at each index k, you take conjugate of the first signal x and offset the second signal y by k indices and then sum the resulting two signals. Intuitively, the correlation measures the similarity of two signals for every offset of the one signal relative to the other. For signals with finite length, the correlation is zero for sufficiently large $|k|$, so often we'll consider the correlation to be finite as well.

Problem 1.1: Autocorrelation (5 pts)

The autocorrelation of a signal x is the correlation of x with itself. For a uniform random signal x think about what the autocorrelation would look like.

- **Plot the signal and its autocorrelation. Use `numpy.correlate` with `mode='full'`, and plot the absolute value of the autocorrelation.**

- Plot the signal and its autocorrelation. Use `numpy.correlate` with `mode='full'`, and plot the absolute value of the autocorrelation.

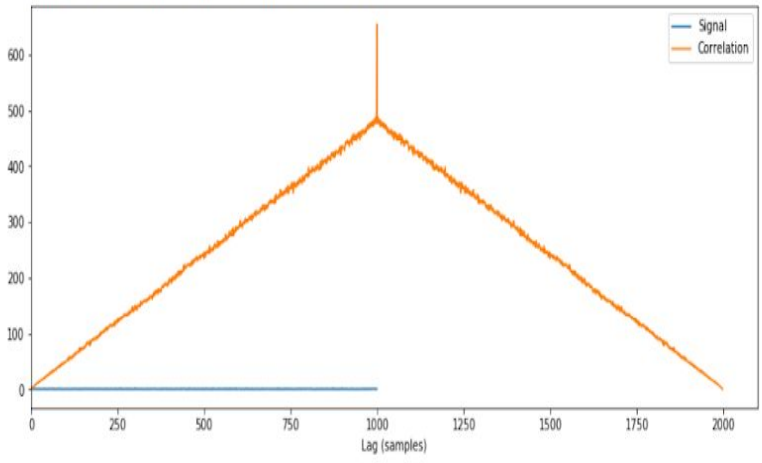
```
[ ] X_len = 1000
    X = np.random.random(X_len) + 1j * np.random.random(X_len)
    # Autocorrelation of a signal x is the correlation of x with itself
    correlation = np.correlate(X, X, mode='full')
```

```
[ ] # r is the same as the correlation at this case.
    r = np.correlate(X, X, mode='full')[len(X)-1:]
    print(X.shape, r.shape)
```

```
(1000,) (1000,)
```

```
[ ] # Plotting.
    plt.figure(figsize=(14, 5))
    # Plot the signal and its autocorrelation.
    plt.plot(X, label='Signal')
    plt.xlabel('Lag (samples)')
    # Plotting the absolute value of the autocorrelation.
    plt.plot(abs(correlation), label='Correlation')
    # For more info : https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.legend.html
    # The simplest legend can be created with the plt.legend() command, which automatically creates a legend for any labeled plot elements.
    plt.legend()
    plt.xlim(0, 2100)
    plt.show()
```

```
/usr/local/lib/python3.6/dist-packages/numpy/core/_asarray.py:85: ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```



So how do we use the correlation to detect a preamble? Let x be the preamble signal, and let y be the signal that the receiver receives. For now, we'll let y be finite length. If the preamble is not received, then we can model y as a random signal, and the correlation between x and y will also be random. However, if the receiver did receive the preamble in the signal y, then we can model y as x with some offset plus noise. The correlation between x and y will then have a large amplitude at the time in y when the preamble started. So if we look for large peaks in the correlation we can detect and then also synchronize to a given preamble.

So how do we use the correlation to detect a preamble? Let x be the preamble signal, and let y be the signal that the receiver receives. For now, we'll let y be finite length. If the preamble is not received, then we can model y as a random signal, and the correlation between x and y will also be random. However, if the receiver did receive the preamble in the signal y, then we can model y as x with some offset plus noise. The correlation between x and y will then have a large amplitude at the time in y when the preamble started. So if we look for large peaks in the correlation we can detect and then also synchronize to a given preamble.

Problem 1.2: Preamble detection using correlation (10 pts)

Implement preamble detection and synchronization using the correlation.

- **Fill in the `detect_preamble` function below. The function should take two signals and return `None` if the preamble is not found, otherwise it should return the index where the preamble starts.**

```
[ ] # For More Tuturials you can check https://www.youtube.com/watch?v=PHYCkgQcQ34  to know more about Beacon :)
    # Compare the correlation magnitude against this value to determine whether there is a preamble or not
    threshold = 100
    def detect_preamble(preamble, signal):
      # Absolute value of correlation between preamble and signal.
      correlation = list(abs(np.correlate(preamble, signal, mode='full')))
      maxValue = max(correlation)
      if maxValue >= threshold:
        # Index where preamble starts. # Get maxValue index inside the correlation list and then subtract it from lenth of signal.
        return len(signal) - 1 - correlation.index(maxValue)
      return None
```

```
[ ] # This cell will test your implementation of `detect_preamble`
    preamble_length = 100
    signal_length = 1000
    preamble = (np.random.random(preamble_length) + 1j * np.random.random(preamble_length))
    signalA = np.random.random(signal_length) + 1j * np.random.random(signal_length)
    signalB = np.random.random(signal_length) + 1j * np.random.random(signal_length)
    preamble_start_idx = 123
    signalB[preamble_start_idx:preamble_start_idx + preamble_length] += preamble
```

```
[ ] # This cell will test your implementation of `detect_preamble`
    preamble_length = 100
    signal_length = 1000
    preamble = (np.random.random(preamble_length) + 1j * np.random.random(preamble_length))
    signalA = np.random.random(signal_length) + 1j * np.random.random(signal_length)
    signalB = np.random.random(signal_length) + 1j * np.random.random(signal_length)
    preamble_start_idx = 123
    signalB[preamble_start_idx:preamble_start_idx + preamble_length] += preamble

    np.testing.assert_equal(detect_preamble(preamble, signalA), None)
    np.testing.assert_equal(detect_preamble(preamble, signalB), preamble_start_idx)
```

Handling Frequency Offsets

In practice, radios will disagree on their center frequencies. This is because a radio's center frequency is determined by a local oscillator (which is fed into the "mixer"), and these oscillators have some error. What this means is that a signal that's transmitted by one radio will be received with a "frequency offset" by another radio.

Unfortunately, if the receiving radio doesn't know what the frequency offset is, then this can cause the correlation to fail. Consider the demonstration below:

Handling Frequency Offsets

In practice, radios will disagree on their center frequencies. This is because a radio's center frequency is determined by a local oscillator (which is fed into the "mixer"), and these oscillators have some error. What this means is that a signal that's transmitted by one radio will be received with a "frequency offset" by another radio.

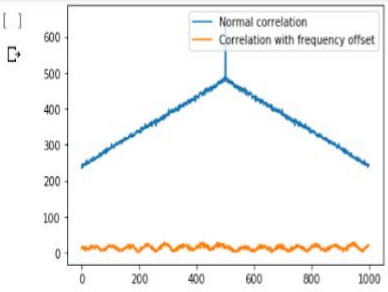
Unfortunately, if the receiving radio doesn't know what the frequency offset is, then this can cause the correlation to fail. Consider the demonstration below:

```
[ ] def shift_frequency(signal, freq_shift):
        """
        Applies a frequency shift to a signal. `freq_shift` is in radians per sample, and should be between 0 and 2π.

        A frequency offset is simply an addition of a linearly increasing phase. Note that you could have taken the DFT,
        circularly shifted the spectrum, and then taken the IDFT, and you would have gotten the same result.
        """
        return signal * np.exp(1j * freq_shift * np.arange(len(signal)))
```

```
[ ] X_with_freq_offset = shift_frequency(X, 0.1)
    X_autocorr = np.correlate(X, X, mode='same')
    X_autocorr_freq_offset = np.correlate(X, X_with_freq_offset, mode='same')
    plt.figure()
    plt.plot(abs(X_autocorr), label="Normal correlation")
    plt.plot(abs(X_autocorr_freq_offset), label="Correlation with frequency offset")
    plt.legend()
    plt.show()
```

[ ]



As you can see, the correlation with a frequency offset doesn't have a large peak anywhere. This is because the frequency offset causes the received signal to "look" significantly different from the transmitted signal, and so the correlation fails to detect the preamble.

The way this is handled in practice is by using a method called "Schmidl-Cox" synchronization. The intuition is that the frequency offset "corrupts" our signal such that we can't detect it using correlation, but if we were to send two copies of the preamble back-to-back, then we would receive two copies of the corrupted preamble back-to-back. Then, instead of searching for a specific signal, we can search for any signal that repeats in time.

Let L be the length of the repeated segment. Then the preamble has length $2L$. At each index d of the signal s, the Schmidl-Cox algorithm calculates the following two values:

$$P(d) = \sum_{m=0}^{L-1} s_{d+m}^* s_{d+m+L}$$

$$R(d) = \frac{1}{2} \sum_{m=0}^{2L-1} |s_{d+m}|^2$$

Then the metric calculated at each index is

$$M(d) = \frac{|P(d)|^2}{R(d)^2}$$

Intuitively, P is a correlation between the next L samples and the following L samples and R is a normalization factor. Once the value of M

Intuitively, P is a correlation between the next L samples and the following L samples and R is a normalization factor. Once the value of M increases beyond a threshold, the preamble is detected. Note that P also has a recursive form:

$$P(d+1) = P(d) + \left(s^*_{d+L} s_{d+2L}\right) - \left(s^*_d s_{d+L}\right)$$

Food for thought: How could we use a Schmidl-Cox preamble to estimate the frequency offset? This may not be immediatly obvious, but we will do this in Lab 4 to estimate and then correct frequency offsets for when we're receiving data.

Problem 1.3 Schmidl-Cox Preamble Detection (10 pts)

- **Implement the Schmidl-Cox preamble detector below.**

You can use either the iterative or recursive formulations for calculating P. Return the index d^* that maximizes $M(d)$ if $M(d^*)$ is greater than the threshold, otherwise return None.

```
[ ] schmidl_cox_threshold = 0.5
    # Schmidl-Cox preamble detector.
    # Try to follow the implementation.
    def detect_preamble_with_freq_offset(signal, short_preamble_len):
      lenth = short_preamble_len
      max_1 = -1
      max_2 = -1
      for d in range(len(signal) - 2*lenth - 1):
        p = sum(signal[d+m].conj() * signal[d+m+lenth] for m in range(lenth))
        r = sum(abs(signal[d+m])**2 for m in range(2*lenth)) * 0.5
        m = (abs(p) / r)**2

        if m >= schmidl_cox_threshold and m < max_1:
          return max_2
        else:
          max_1 = m
          max_2 = d

      return None
```

```
[ ] # This cell will test your implementation of `detect_preamble`
    short_preamble_length = 100
    signal_length = 1000
    short_preamble = np.exp(2j * np.pi * np.random.random(short_preamble_length))
    preamble = np.tile(short_preamble, 2)
    noise = np.random.normal(size=signal_length) + 1j * np.random.normal(size=signal_length)
    signalA = 0.1 * noise
    signalB = 0.1 * noise
    signalC = 0.1 * noise
    preamble_start_idx = 321
    signalB[preamble_start_idx:preamble_start_idx + len(preamble)] += preamble
    signalC[preamble_start_idx:preamble_start_idx + len(preamble)] += shift_frequency(preamble, 0.1)

    np.testing.assert_equal(detect_preamble_with_freq_offset(signalA, short_preamble_length), None)
    np.testing.assert_equal(detect_preamble_with_freq_offset(signalB, short_preamble_length), 321)
    np.testing.assert_equal(detect_preamble_with_freq_offset(signalC, short_preamble_length), 321)
```

Part 2: Slotted-ALOHA Capacity

In the idealized slotted aloha model, time is splitted into multiple slots and the packet transmission time is one full slot. All nodes are perfectly syncronized, and transmit at the begining of each slot. The transmission probability for each user for each slot is some value p.

For each time slot, three cases may happen. 1): There are more than one nodes transmits, thus resulting in a conflict slot. The receiver cannot receive them correctly. 2): There is only one node transmission and thus the receiver can receive it correctly. 3): No node transmits at the current slot.

In this section, we will verify the slotted-ALOHA capacity curve through Monte-Carlo simulation.

Problem 2.1 Slotted ALOHA Simulation (10 pts)

- **Fill in the function below.**

The function will take a number of users, a number of time slots, and a probability p that a given user will transmit on a given timeslot. The function will return the ratio of successful timeslots (timeslots in which a successful transmission occurred) to total number of timeslots.

```
[ ] # num_timeslots is not used here at my equation :|
    # For Pure ALOHA and Solotted ALOHA you can check this video https://www.youtube.com/watch?v=509AmnKtx-c
    # ALOHA is mac protocol (contention based).
    # 1. Node can send at time when they are ready to send.
    # 2. In case of collision they wait random amount of time and again resent.
    # 3. Feedback property or ACK is used to check successful transmission of frame.
    def simulate_slotted_aloha(num_users, num_timeslots, per_slot_user_prob):
      return num_users * per_slot_user_prob * ((1 - per_slot_user_prob)**(num_users -1))
```

▾ Problem 2.2 Verifying Slotted ALOHA capacity (5 pts)

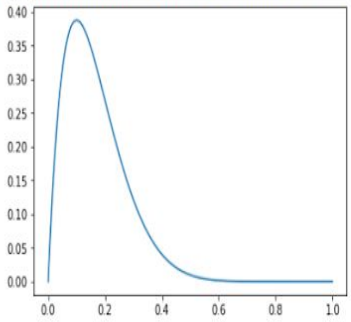
Run your simulation over 1000 values of p between 0 and 1 with 10 users and 10k timeslots.

- *Plot the timeslot success ratio as a function of p*

Verify that the resulting plot matches the theoretical curve.

```
[ ] p = np.arange(0, 1, 0.001) # 1000 value
    success = [simulate_slotted_aloha(10, 10000, i) for i in p]
    plt.figure()
    plt.plot(p, success)
```

```
[<matplotlib.lines.Line2D at 0x7f6861de5668>]
```



By Mohamed Said:

Code:

https://github.com/mohamed-said-ibrahem/Computer-Networks-IV-Synchronization-and-MAC-protocols