CID: 02052671

In [1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Question 1

(a)

In [2]:
```python
def simulate_AR4(r1, f1, r2, f2, num_samples, burn_in=1000):

    #1 over root of z
    a = r1 * np.exp(1j * 2 * np.pi * f1)
    b = r1 * np.exp(-1j * 2 * np.pi * f1)
    c = r2 * np.exp(1j * 2 * np.pi * f2)
    d = r2 * np.exp(-1j * 2 * np.pi * f2)
    #calculate the phi values
    phi1 = a + b + c + d
    phi2 = -((a + b) * (c + d) + a * b + c * d)
    phi3 = a * b * (c + d) + c * d * (a + b)
    phi4 = -a * b * c * d
    #simulate the AR(4) process including burn in
    ar_process = np.zeros(num_samples + burn_in)
    for t in range(4, num_samples + burn_in):
        ar_process[t] = (
            phi1 * ar_process[t-1] +
            phi2 * ar_process[t-2] +
            phi3 * ar_process[t-3] +
            phi4 * ar_process[t-4] +
            np.random.normal()
        )
    #discard the burn in
    ar_process = ar_process[burn_in:]

    return ar_process
```

Mathematical Description: $z_1 = 1/a$, $z_2 = 1/b$, $z_3 = 1/c$, $z_4 = 1/d$

$a = r_1 e^{i2\pi f_1}$, $b = r_1 e^{-i2\pi f_1}$, $c = r_2 e^{i2\pi f_2}$, $d = r_2 e^{-i2\pi f_2}$

$1 - \phi_{1,4} z - \phi_{2,4} z^2 - \phi_{3,4} z^3 - \phi_{4,4} z^4 = (1 - az)(1 - bz)(1 - cz)(1 - dz)$

Therefore, $\phi_{1,4} = a + b + c + d$, $\phi_{2,4} = -((a + b)(c + d) + ab + cd)$, $\phi_{3,4} = ab(c + d) + cd(a + b)$, $\phi_{4,4} = -abcd$

(b)

In [3]:
```python
def S_AR(frequencies, phis, sigma2):

    #find the value of p
    p = len(phis)
    #create an array to store values of sdf
    spectral_density = np.zeros_like(frequencies)
    #calculate the sdf of AR(p) process for given frequencies and store in the array
    for i, f in enumerate(frequencies):
        sum_term = np.sum([phis[j] * np.exp(-1j * 2 * np.pi * (j + 1) * f) for j in rang
        denominator = np.abs(1 - sum_term)**2
        spectral_density[i] = sigma2 / denominator
```

```python
    return spectral_density
```

(c)

```python
def periodogram(X):

    N = len(X)
    #fourier transform
    X_fft = np.fft.fft(X)
    #compute the spectral estimate
    Pxx = np.abs(X_fft) ** 2 / N
    #fourier frequencies
    f = np.fft.fftfreq(N)
    f = np.fft.fftshift(f)
    #fftshift to put it in right order
    Pxx = np.fft.fftshift(Pxx)

    return f, Pxx
```

```python
def direct(X, p):
    N = len(X)
    taper = []
    #calculate [pN]
    k = int(p*N)
    #create a list of taper
    for t in range(1, N+1):
        if 1 <= t <= k / 2:
            taper.append(1/2 * (1 - np.cos(2 * np.pi * t / (k + 1))))

        elif k / 2 < t < N + 1 - k / 2:
            taper.append(1)

        elif N + 1 - k / 2 <= t <= N:
            taper.append(1/2 * (1 - np.cos(2 * np.pi * (N + 1 - t) / (k + 1))))
    #normalise taper
    taper = taper / np.linalg.norm(taper)
    #apply taper
    X_tapered = X * taper
    #fourier transform of the tapered time series
    spec_est = np.fft.fftshift(np.fft.fft(X_tapered))
    #fourier frequencies
    f = np.fft.fftfreq(N)
    f = np.fft.fftshift(f)
    return f, np.abs(spec_est) ** 2
```

(d)

```python
#initialise values for simulations
num_realizations = 5000
N = 64
sigma2 = 1
frequencies_of_interest = [6/64, 8/64, 16/64, 26/64]
r1 = r2 = 0.8
f1 = 6/64
f2 = 26/64
p_values = [0.05, 0.1, 0.25, 0.5]
#arrays to store results
periodogram_results = np.zeros((num_realizations, N))
direct_results = {p: np.zeros((num_realizations, N)) for p in p_values}

for i in range(num_realizations):
    ar_process = simulate_AR4(r1, f1, r2, f2, N)
```

```python
    #compute periodogram
    freq, periodogram_result = periodogram(ar_process)
    periodogram_results[i, :] = periodogram_result
    #compute direct spectral estimates for different taper percentages
    for p in p_values:
        freq_direct, direct_result = direct(ar_process, p)
        direct_results[p][i, :] = direct_result

a = r1 * np.exp(1j * 2 * np.pi * f1)
b = r1 * np.exp(-1j * 2 * np.pi * f1)
c = r2 * np.exp(1j * 2 * np.pi * f2)
d = r2 * np.exp(-1j * 2 * np.pi * f2)

#coefficients for the AR(4) process
phi1 = a + b + c + d
phi2 = -((a + b) * (c + d) + a * b + c * d)
phi3 = a * b * (c + d) + c * d * (a + b)
phi4 = -a * b * c * d

freq_list = freq.tolist()

#list and dictionary to store bias values
bias_periodogram = []
bias_direct = {p: [] for p in p_values}

#calculate the sample percentage bias for each frequency and p values
for f in frequencies_of_interest:
    true_spectral_density = S_AR([f], [phi1, phi2, phi3, phi4], sigma2)[0]
    f_i = freq_list.index(f)
    mean_periodogram = np.mean(periodogram_results[:, f_i])
    bias_periodogram.append((mean_periodogram - np.real(true_spectral_density)) / np.rea

for p in p_values:
    for f in frequencies_of_interest:
        f_i = freq_list.index(f)
        true_spectral_density = S_AR([f], [phi1, phi2, phi3, phi4], sigma2)[0]
        mean_direct = np.mean(direct_results[p][:, f_i])
        bias_direct[p].append((mean_direct - np.real(true_spectral_density)) / np.real(t
```

```
C:\Users\Juno\AppData\Local\Temp\ipykernel_7928\3591646313.py:18: ComplexWarning: Castin
g complex values to real discards the imaginary part
  ar_process[t] = (
```

In [7]:
```python
bias_periodogram
```

Out[7]:
```
[-3.7030421566596625,
 -0.1418589417126524,
 11.678361600750788,
 -5.617685587232044]
```

In [8]:
```python
bias_direct
```

Out[8]:
```
{0.05: [-3.4865049467039366,
  -0.09027016551017428,
  8.36971692278166,
  -5.091795216780823],
 0.1: [-3.577660526232341,
  0.06248913579587513,
  5.333552850222512,
  -4.908174045397036],
 0.25: [-3.490053716937146,
  0.16231408429200803,
  0.7033769407675953,
  -4.551105361416603],
 0.5: [-3.245066291647209,
  0.4165414373315374,
```

```
  0.5917070627247296,
  -4.449752244689624]}
```

In [9]:
```python
r_values = np.linspace(0.8, 0.99, 20)
frequencies_of_interest = [6/64, 8/64, 16/64, 26/64]

bias_per_dict = {f: [] for f in frequencies_of_interest}

for f in frequencies_of_interest:
    for r in r_values:
        num_realizations = 500
        N = 64
        sigma2 = 1
        frequencies_of_interest = [6/64, 8/64, 16/64, 26/64]
        r1 = r2 = r
        f1 = 6/64
        f2 = 26/64
        p_values = [0.05, 0.1, 0.25, 0.5]
        # Arrays to store results
        periodogram_results = np.zeros((num_realizations, N))
        direct_results = {p: np.zeros((num_realizations, N)) for p in p_values}

        for i in range(num_realizations):
            ar_process = simulate_AR4(r1, f1, r2, f2, N)

            # Compute periodogram
            freq, periodogram_result = periodogram(ar_process)
            periodogram_results[i, :] = periodogram_result

            # Compute direct spectral estimates for different taper percentages
            for p in p_values:
                freq_direct, direct_result = direct(ar_process, p)
                direct_results[p][i, :] = direct_result

        a = r1 * np.exp(1j * 2 * np.pi * f1)
        b = r1 * np.exp(-1j * 2 * np.pi * f1)
        c = r2 * np.exp(1j * 2 * np.pi * f2)
        d = r2 * np.exp(-1j * 2 * np.pi * f2)

        # Coefficients for the AR(4) process
        phi1 = a + b + c + d
        phi2 = -((a + b) * (c + d) + a * b + c * d)
        phi3 = a * b * (c + d) + c * d * (a + b)
        phi4 = -a * b * c * d

        freq_list = freq.tolist()

        true_spectral_density = S_AR([f], [phi1, phi2, phi3, phi4], sigma2)[0]
        f_i = freq_list.index(f)
        mean_periodogram = np.mean(periodogram_results[:, f_i])
        bias_per_dict[f].append((mean_periodogram - np.real(true_spectral_density)) / np
```

```
C:\Users\Juno\AppData\Local\Temp\ipykernel_7928\3591646313.py:18: ComplexWarning: Castin
g complex values to real discards the imaginary part
  ar_process[t] = (
```

In [10]:
```python
bias_per_dict
```

Out[10]:
```
{0.09375: [-6.148270055614468,
  -6.827906998478922,
  7.303065927450672,
  -2.5632397455866855,
  -2.16084016324775,
  -3.326493178928062,
  -7.03867547144333,
  -15.439438221564039,
```

```
 -17.11176679954237,
 -13.588496041961434,
 -6.91689700313812,
 -18.053661041058696,
 -13.561925149570381,
 -21.19586774436256,
 -26.743646621044913,
 -30.465820763114245,
 -34.66844023101906,
 -46.35839422258904,
 -56.63494712362112,
 -73.62449677017847],
0.125: [-2.685729264770699,
 -0.24614333341138075,
 -3.417861489731814,
 7.322062155688366,
 12.442961921773696,
 1.7729310142851826,
 -2.477156150457722,
 5.561220824773704,
 10.494273200976357,
 8.682348523878378,
 14.155940853726204,
 16.593599795461138,
 26.92368808903825,
 20.632257601516628,
 18.355763315311904,
 29.890420301909444,
 44.25983257003065,
 41.6808499055258,
 73.2461868293526,
 77.2535762997158],
0.25: [6.302814253902198,
 17.31075859196092,
 16.67206906777839,
 13.445045156197708,
 11.73717331703432,
 22.177206871416136,
 16.03615642459404,
 15.80212191388548,
 32.75197458498146,
 22.90958573027817,
 26.111086228311493,
 23.079465032633774,
 40.6648621540279,
 27.71116178966542,
 43.27676029246826,
 53.78137275946777,
 58.32827508777291,
 75.40926746641925,
 85.5941898379032,
 111.31087747356672],
0.40625: [-15.061969179971438,
 -7.9582316066706005,
 -4.973543262072715,
 -3.410432973280595,
 -10.915517415777142,
 -5.776655225711476,
 -7.6077876959143875,
 -8.708426991649908,
 -7.006266040240895,
 -15.194956990812658,
 -20.615348094343805,
 -17.672072896561048,
 -13.35434654790383,
 -24.184755744340826,
```

```
-23.05764494429028,
-24.244225521654933,
-39.06118847501208,
-43.38710286372957,
-55.39962623466326,
-75.43225912676918]}
```

In [11]:
```
#define a function that simulates the sample percentage bias of the spectral estimate fo
def repeat_dir(f):
    r_values = np.linspace(0.8, 0.99, 20)
    p_values = [0.05, 0.1, 0.25, 0.5]
    num_realizations = 500
    N = 64
    sigma2 = 1
    bias_dir = {p: [] for p in p_values}

    for r in r_values:
        r1 = r2 = r
        f1 = 6/64
        f2 = 26/64

        # Arrays to store results
        direct_results = {p: np.zeros((num_realizations, N)) for p in p_values}

        for i in range(num_realizations):
            ar_process = simulate_AR4(r1, f1, r2, f2, N)

            # Compute direct spectral estimates for different taper percentages
            for p in p_values:
                freq, direct_result = direct(ar_process, p)
                direct_results[p][i, :] = direct_result

        a = r1 * np.exp(1j * 2 * np.pi * f1)
        b = r1 * np.exp(-1j * 2 * np.pi * f1)
        c = r2 * np.exp(1j * 2 * np.pi * f2)
        d = r2 * np.exp(-1j * 2 * np.pi * f2)

        # Coefficients for the AR(4) process
        phi1 = a + b + c + d
        phi2 = -((a + b) * (c + d) + a * b + c * d)
        phi3 = a * b * (c + d) + c * d * (a + b)
        phi4 = -a * b * c * d

        freq_list = freq.tolist()

        for p in p_values:
            f_i = freq_list.index(f)
            true_spectral_density = S_AR([f], [phi1, phi2, phi3, phi4], sigma2)[0]
            mean_direct = np.mean(direct_results[p][:, f_i])
            bias_dir[p].append((mean_direct - np.real(true_spectral_density)) / np.real(

    return bias_dir
```

In [12]:
```
b_6 = repeat_dir(6/64)
```

```
C:\Users\Juno\AppData\Local\Temp\ipykernel_7928\3591646313.py:18: ComplexWarning: Castin
g complex values to real discards the imaginary part
  ar_process[t] = (
```

In [14]:
```
b_8 = repeat_dir(8/64)
```

```
C:\Users\Juno\AppData\Local\Temp\ipykernel_7928\3591646313.py:18: ComplexWarning: Castin
g complex values to real discards the imaginary part
  ar_process[t] = (
```

In [16]:
```
b_16 = repeat_dir(16/64)
```

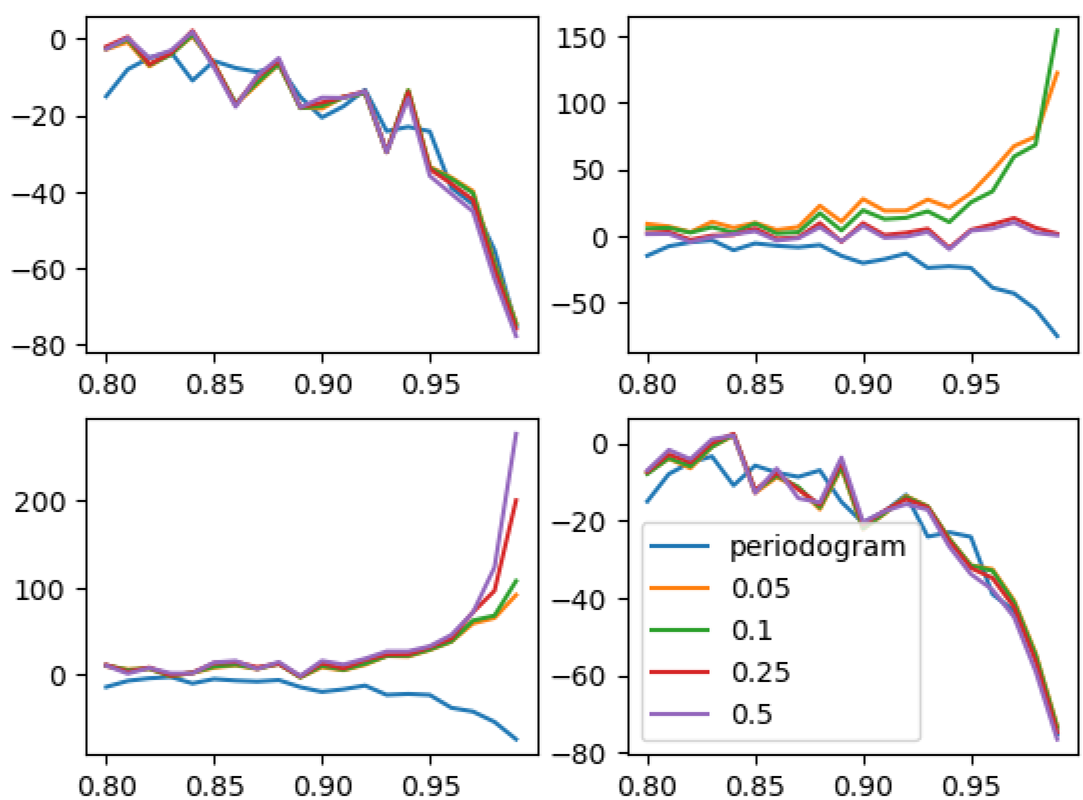
```
C:\Users\Juno\AppData\Local\Temp\ipykernel_7928\3591646313.py:18: ComplexWarning: Castin
g complex values to real discards the imaginary part
  ar_process[t] = (
```

In [18]:
```
b_26 = repeat_dir(26/64)
```

```
C:\Users\Juno\AppData\Local\Temp\ipykernel_7928\3591646313.py:18: ComplexWarning: Castin
g complex values to real discards the imaginary part
  ar_process[t] = (
```

In [48]:
```
fig, axs = plt.subplots(2, 2)
axs[0, 0].plot(r_values, bias_per_dict[26/64], label='periodogram')
for p in p_values:
    axs[0, 0].plot(r_values, b_6[p], label= f"{p}")
axs[1, 0].plot(r_values, bias_per_dict[26/64], label='periodogram')
for p in p_values:
    axs[1, 0].plot(r_values, b_8[p], label= f"{p}")
axs[0, 1].plot(r_values, bias_per_dict[26/64], label='periodogram')
for p in p_values:
    axs[0, 1].plot(r_values, b_16[p], label= f"{p}")
axs[1, 1].plot(r_values, bias_per_dict[26/64], label='periodogram')
for p in p_values:
    axs[1, 1].plot(r_values, b_26[p], label= f"{p}")
plt.legend()
```

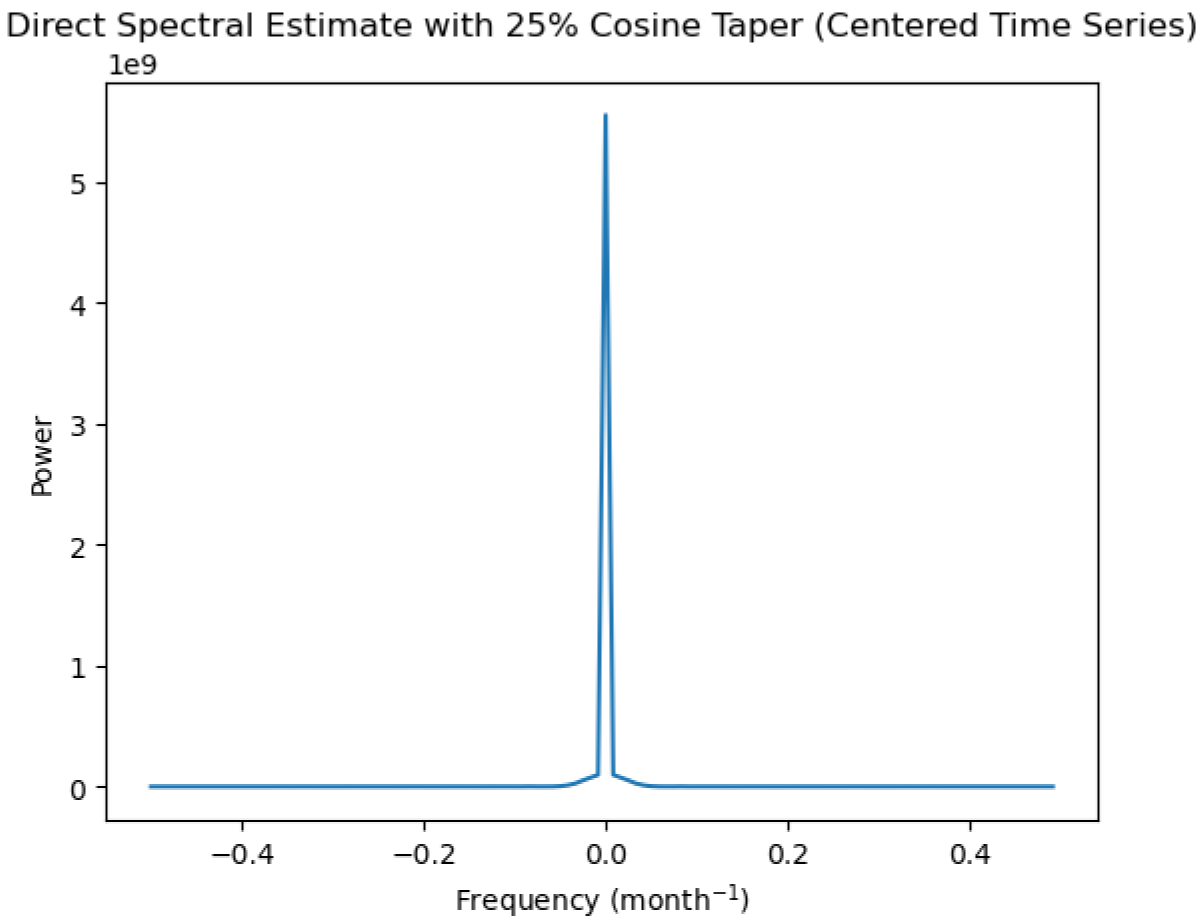
Out[48]:
```
<matplotlib.legend.Legend at 0x2232e587250>
```



The following four plots are the sample percentage bias of spectral estimates at four different frequencies, For the periodogram and 4 p values, we can see that the bias becomes more and more extreme as the value of r increases. Also, the bias of the spectral estimate becomes more negative as r increases for frequencies 6/64 and 26/64. On the other hand, the bias of the spectral estimate becomes positive as r increases for frequencies 8/64 and 16/64.

Question 2

In [54]:
```
file_path = '0226.xlsx'
data = pd.read_excel(file_path, header=None, names=['T', 'X'])
# Extract the gauge readings
X = data['X'].values
x, y = direct(X, 0.25)
# Plot the results
plt.plot(x, y)
plt.title('Direct Spectral Estimate with 25% Cosine Taper (Centered Time Series)')
plt.xlabel('Frequency (month$^{-1}$)')
plt.ylabel('Power')
plt.show()
```

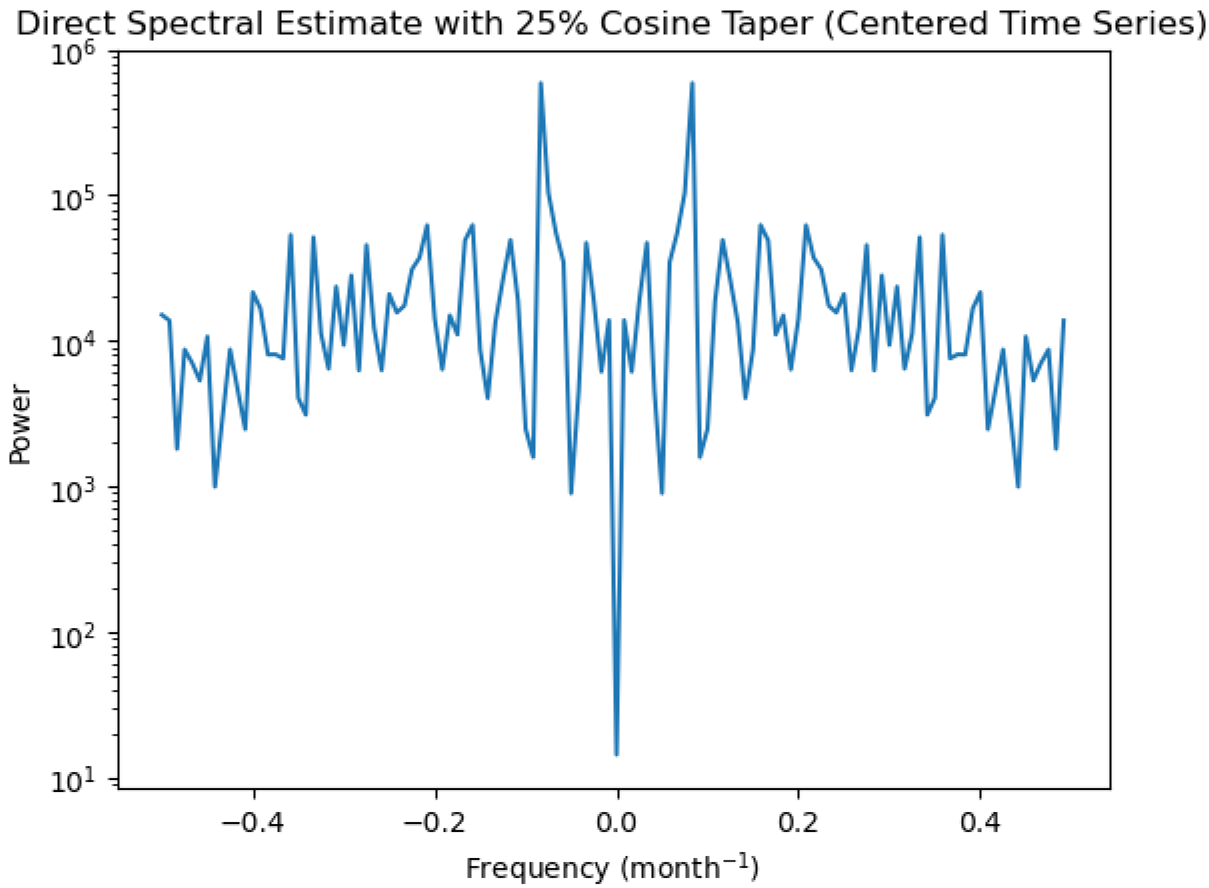



```
<Figure size 400x300 with 0 Axes>
```

If we do not centre the given time series, as we can see from the plot above, we will have a large peak at the zero frequency and we will not be able to capture the trend in freuqencies around 0.

The mean of the time series corresponds to the 0(or infinite) frequency term of the Fast Fourier transform and the peak affects the data around point 0. Also, we have to centre the data ensure zero mean time series when we apply spectral analysis.

In [55]:
```
X_centre = X - np.mean(X)
x_centre, y_centre = direct(X_centre, 0.25)
plt.plot(x_centre, y_centre)
plt.title('Direct Spectral Estimate with 25% Cosine Taper (Centered Time Series)')
plt.xlabel('Frequency (month$^{-1}$)')
plt.ylabel('Power')
plt.yscale('log')
plt.show()
```




```
<Figure size 400x300 with 0 Axes>
```

As we can see from the plot above, the dominant frequencies(i.e. frequencies corresponding to the peaks) are $\pm$ 0.08333333. . .. This is exactly $\frac{1}{12}$ and we can say that this time series is yearly periodic

In [22]:
```
np.where(y_centre == max(y_centre))
```

Out[22]:
```
(array([50, 70], dtype=int64),)
```

In [23]:
```
x[70]
```

Out[23]:
```
0.08333333333333333
```

Question 3

(a)

In [61]:
```
file_path = '0226.xlsx'
data = pd.read_excel(file_path, header=None, names=['T', 'X'])
x = data['X'].values
x = x - np.mean(x)
#MLE
def least_square(X, p, N):
    F = np.zeros((N - p, p))
    for r in range(N - p):
        F[r, :] = X[r : p + r][::-1]
    FT = np.transpose(F)
    FTF = np.dot(FT, F)

    return np.dot(np.dot(np.linalg.inv(FTF), FT), X[p : N])

#Yule-Walker
def yw(X, p, N):
    X = X.tolist()
```

```
    new_x = p*[0] + X[:N] + p*[0]

    return least_square(new_x, p, N + 2 * p)

#function to forecast 1 step ahead
def forecast(X, phis, k):
    p = len(phis)
    x = X[:k]
    x_i = x[-p:][::-1]

    return np.dot(phis, x_i)

#function to calculate the rmse of a data set
def rmse(e):
    e2 = np.mean(np.square(e))
    return e2**(1/2)
```

In [62]:
```
#RMSE of MLE method for p = 1 ~ 10
for p in range(1,11):
    x_est = []
    for t in range(60, 120):
        x_est.append(forecast(x, least_square(x, p, t), t))

    print(rmse(x[60:] - x_est))
```

```
147.93931279020893
149.57232855149869
150.45790759837493
149.25272758607022
147.50483956101826
155.9067079020586
150.0637706913615
152.61750063327094
154.06652853020702
154.55656813063243
```

In [63]:
```
#RMSE of Yule-Walker method for p = 1 ~ 10
for p in range(1,11):
    x_est = []
    for t in range(60, 120):
        x_est.append(forecast(x, yw(x, p, t), t))

    print(rmse(x[60:] - x_est))
```

```
147.56353626000615
149.18210323798633
149.73499767259008
148.53044623229374
146.46941421978505
153.22216253836785
147.65340858355663
149.63262907150434
150.11142044616398
150.18491041963057
```

p = 5 is best for both

(b)

In [64]:
```
freq = np.linspace(-1, 1, 1000)
phis = least_square(x, 5, 120)
```

In [65]:
```
#extension MLE method that estimates the sigma value
def least_square_sig(X, p, N):
```

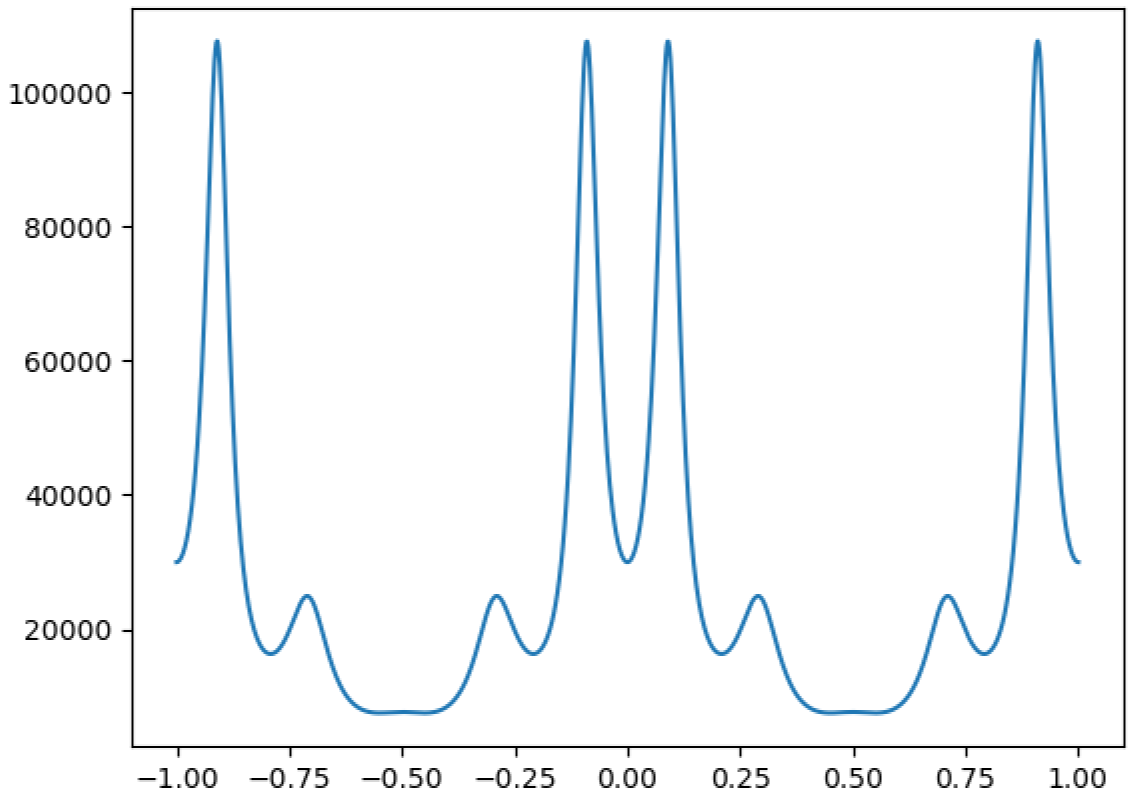
```
    F = np.zeros((N - p, p))
    for r in range(N - p):
        F[r, :] = X[r : p + r][::-1]
    FTF = np.dot(np.transpose(F), F)
    FT = np.transpose(F)
    phi = np.dot(np.dot(np.linalg.inv(FTF), FT), X[p:N])
    sigma = np.dot(np.transpose((X[p:N] - np.dot(F, phi))), (X[p:N] - np.dot(F, phi))) /

    return sigma
```

In [66]:
```
sigma_est = least_square_sig(x, 5, 120)
sigma_est
```

Out[66]:
```
20724.28999802488
```

In [67]:
```
plt.plot(freq, S_AR(freq, phis, sigma_est))
plt.figure(figsize=(4,3))
```

Out[67]:
```
<Figure size 400x300 with 0 Axes>
```




```
<Figure size 400x300 with 0 Axes>
```

(c), (d)

In [68]:
```
v = x.tolist()

#forecasting values up to 12 steps
for t in range(120, 132):
    x_temp = forecast(v, phis, t)
    v = v + [x_temp]
```

In [69]:
```
t = [t for t in range(79,132)]
t1 = [t for t in range(120,132)]
```

In [70]:
```
resids = []
phis = least_square(x, 5, 120)

for t in range(5, 120):
```

```
    resids.append(x[t] - np.dot(phis, x[t-5:t][::-1]))

sigma_re = np.std(resids)
```

In [71]:
```
sigma_re
```

Out[71]:
```
143.95224226665493
```

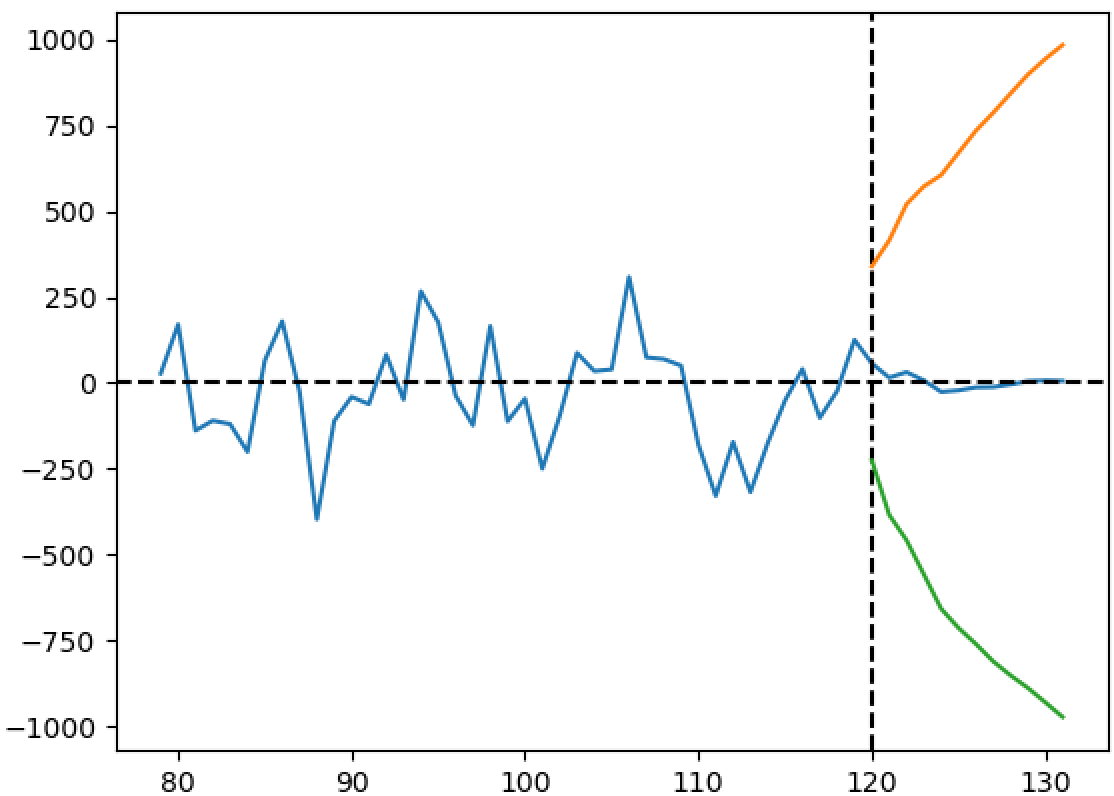
In [72]:
```
steps = []
for l in range(1, 13):
    steps.append(np.sqrt(l) * sigma_re)
```

In [73]:
```
steps = np.array(steps)
```

In [74]:
```
t = [t for t in range(79,132)]
```

In [75]:
```
plt.plot(t, v[79:132])
plt.axhline(y = 0, linestyle='dashed', color='k')
plt.axvline(x = 120, linestyle='dashed', color='k')
#Upper bound of estimation
plt.plot(t1, v[120:132] + 1.96 * steps)
#Lower bound of estimation
plt.plot(t1, v[120:132] - 1.96 * steps)
```

Out[75]:
```
[<matplotlib.lines.Line2D at 0x2233156e3d0>]
```



(e)

In [80]:
```
#RMSE of MLE method for p = 11 ~ 40
for p in range(11,41):
    x_est = []
    for t in range(60, 120):
        x_est.append(forecast(x, least_square(x, p, t), t))

    print(rmse(x[60:] - x_est))
```

```
157.483677933625
```

```
156.7394988874441
154.20221957304958
157.97829853185718
163.4172307459569
158.39142644553357
159.58191940285585
161.51055887794274
160.92168199754184
161.86923735654707
162.12640114234287
164.42969908815127
175.40568557724143
176.3203034048198
178.0247319070107
181.7810012943352
183.04500107876794
295.923101683606
372.06841414558994
538.032295722581
1021.6744817296089
1078.2512861494583
91064.65291530162
6758.672177164311
3113.3729177856953
15630.334997577993
3517.019719219868
5109.372297150826
1365.0745348958706
74295.74550429452
```

From the RMSE values above, we can see that the error does increase but not dramtically until approximately p = 25. But when p becomes even larger, the error value starts exploding to very high values.

In [81]:
```python
#RMSE of Yule-Walker method for p = 11 ~ 40
for p in range(11,41):
    x_est = []
    for t in range(60, 120):
        x_est.append(forecast(x, yw(x, p, t), t))

    print(rmse(x[60:] - x_est))
```

```
150.6523421309526
149.25431178556104
148.94079530734632
150.34504317783183
151.84646292940823
150.8894836920633
151.670958778916
153.15859633670078
152.95559937446282
152.55275856411205
151.29853936158722
152.2631397080339
155.26283391877018
154.97697778332676
154.95468642213368
154.64745244671323
155.55045468408488
155.41977624892934
154.4241907093383
154.80784870249545
155.27942001486
155.49147331431772
150.63797853573632
151.56512961638023
```

```
152.77968045749358
152.0073374617875
152.6277695279494
153.45162122251128
153.37905950812893
153.75953394053926
```

On the other hand for Yule-Walker, even though we can't see RMSE value lower than p = 5, we also do not see any explosion of error for higher p values.

In [42]:
```python
#forecast L steps ahead
def forecast_l(x, l, phis, N):
    v = x.tolist()
    v = v[:N]
    est = []
    for t in range(N, N + l):
        x_temp = forecast(v, phis, t)
        v = v + [x_temp]

    return x_temp
```

In [43]:
```python
estimate = forecast_l(x, 10, least_square(x, 5 ,60), 60)
```

The following codes will forecast the $\tilde{x}_{60} \sim \tilde{x}_{120}$ values with 10, 20, 30 step forecasts for up to p = 20. Then, the RMSE value for each p is calculated.

In [44]:
```python
for p in range(1, 21):
    ten_step = []
    for r in range(50, 110):
        ten_step.append(forecast_l(x, 10, least_square(x, p, r), r))

    print(rmse(x[60:] - ten_step))
```

```
157.94585545844254
157.96911322866652
157.96355917787324
157.205289643122
155.02211284254037
146.48178708707795
147.30761505225325
152.22662254439118
153.7981457590475
153.50041092248625
157.73043291706614
153.90630020883515
153.31952571908454
155.66679967528063
154.9668639005751
151.16225357352494
151.43030170325866
153.61606132095685
151.04746443550624
148.62943900427317
```

In [45]:
```python
for p in range(1,21):
    twenty_step = []
    for r in range(40, 100):
        twenty_step.append(forecast_l(x, 20, least_square(x, p, r), r))

    print(rmse(x[60:] - twenty_step))
```

```
157.96753921532488
157.967390875644
157.96739340415095
```

```
158.052391787671
157.8532936103267
148.9544586003528
149.63508775289114
155.57759265853127
153.44861225128545
152.58826134775322
157.74102655884624
161.7913856551494
165.34715263920856
172.22126179179392
165.79109934695512
165.29077106957402
185.03635472253467
210.83977074215906
3507.544867801912
169771726.0484336
```

In [46]:
```python
for p in range(1,21):
    thirty_step = []
    for r in range(30, 90):
        thirty_step.append(forecast_l(x, 30, least_square(x, p, r), r))

    print(rmse(x[60:] - thirty_step))
```

```
157.96739146952007
157.96739117825572
157.96739044359495
157.9754154340397
157.9326078408276
158.3675539596129
165.97198874269492
175.23861296364987
178.38502848130062
186.88754910764638
199.68540247621826
205.88647561991644
214.8861148910459
225.3761034539313
211.3391162840599
2842236468189040.5
4.2680319899185454e+57
6.157724230087809e+44
4.1982098130112614e+39
1.4863149408825037e+52
```

The accuracy of the forecast is similar up to p = 5. However, if p gets higher, we see some differences. For 10 step forecasting, the forecast error stays stable for all the p orders up to p = 20. In fact the second best estimation was at p = 20. For 20 step forecasting, the forecast error starts exploding approximately after p = 16. Finally, for the 30 step forecasting, we see that the errors are already higher than the lower step forecasts for low values of p. Also, after p = 15, the error explodes to incredibly high values.