Solution Exercise 1: Short-time features

```python
import numpy as np
import scipy.signal.windows as windows
import matplotlib.pyplot as plt
import sys
import math

def compute_short_time_energy(signal, frame_length, hop_size, window_type='rectangular'):
    """Compute Short-Time Energy (STE) with a given window type."""
    num_frames = int(np.floor((len(signal)-frame_length)/hop_size)+1)
    print(f'num_frames: ', num_frames)
    energy = np.zeros(num_frames)

    if window_type == 'rectangular':
        window = np.ones(frame_length)
    elif window_type == 'hamming':
        window = windows.hamming(frame_length)
    elif window_type == 'hann':
        window = windows.hann(frame_length)
    else:
        raise ValueError("Unsupported window type. Choose 'rectangular' or 'hamming'.")

    for i in range(num_frames):
        start = i * hop_size
        end = min(start + frame_length, len(signal))
        frame = np.zeros(frame_length)
        frame[:end-start] = signal[start:end]  # Zero-padding if needed
        frame = frame * window
        energy[i] = np.sum(frame ** 2)
    return energy
```

Solution Exercise 1: Short-time features

```
def compute_short_time_magnitude(signal, frame_length, hop_size, window_type='rectangular'):
    """Compute Short-Time Energy (STE) with a given window type."""
    num_frames = int(np.floor((len(signal)-frame_length)/hop_size)+1)
    print(f'num_frames: ', num_frames)
    magnitude = np.zeros(num_frames)

    if window_type == 'rectangular':
        window = np.ones(frame_length)
    elif window_type == 'hamming':
        window = windows.hamming(frame_length)
    elif window_type == 'hann':
        window = windows.hann(frame_length)
    else:
        raise ValueError("Unsupported window type. Choose 'rectangular' or 'hamming'.")

    for i in range(num_frames):
        start = i * hop_size
        end = min(start + frame_length, len(signal))
        frame = np.zeros(frame_length)
        frame[:end-start] = signal[start:end]  # Zero-padding if needed
        frame = np.abs(frame) * window # Element-wise multiplication
        magnitude[i] = np.sum(frame)
    return magnitude
```

Solution Exercise 1: Short-time features

```python
def compute_short_time_zero_crossing(signal, frame_length, hop_size, window_type
    ='rectangular'):
    """Compute Short-Time Zero Crossing (STZ) with a given window type."""
    num_frames = int(np.floor((len(signal)-frame_length)/hop_size)+1)
    zc = np.zeros(num_frames)

    if window_type == 'rectangular':
        window = np.ones(frame_length)
    elif window_type == 'hamming':
        window = windows.hamming(frame_length)
    elif window_type == 'hann':
        window = windows.hann(frame_length)
    else:
        raise ValueError("Unsupported window type. Choose 'rectangular' or '
             hamming'.")

    for i in range(num_frames):
        start = i * hop_size
        end = min(start + frame_length, len(signal))
        frame = np.zeros(frame_length)
        frame[:end-start] = signal[start:end]  # Zero-padding if needed
        frame = frame*window
        frame = 0.5*np.abs(np.diff(np.sign(frame)))
        zc[i] = np.sum(frame)
    return zc
```

Solution Exercise 1: Short-time features

```python
def main():
    if len(sys.argv) < 2:
        print("Usage: python script.py <wavfile>")
        sys.exit(1)

    # Load the WAV file
    wav_file = sys.argv[1]
    import librosa
    signal, sample_rate = librosa.load(wav_file, sr=None)
    print(signal.shape)
    if signal.ndim > 1:
        signal = signal[:, 0]  # Convert to mono if stereo

    frame_length = int(0.05 * sample_rate)  # 25 ms window
    hop_size =  1 #int(0.01 * sample_rate)  # 10 ms hop size

    # Compute STE for rectangular and Hamming windows
    ste_mag = compute_short_time_magnitude(signal, frame_length, hop_size, 'hamming')
    ste_eng = compute_short_time_energy(signal, frame_length, hop_size, 'hamming')
    ste_zcr = compute_short_time_zero_crossing(signal, frame_length, hop_size, 'rectangular')


    t = np.linspace(0, ste_eng.shape[0]/sample_rate, len(ste_eng))

    plt.figure(figsize=(10, 5))
```

Solution Exercise 1: Short-time features

```
time = np.linspace(0, len(signal)/sample_rate, len(signal))
    plt.subplot(4, 1, 1)
    plt.plot(time, signal, color='black')
    plt.title('Speech Signal')
    plt.xlabel('time in sec')
    plt.ylabel('Amplitude')
    #plt.xlim(0,math.ceil(len(signal/sample_rate)))
    plt.ylim(np.min(signal),np.max(signal))
    plt.legend()

    plt.subplot(4, 1, 2)
    plt.plot(t, ste_eng, color='red')
    plt.title('Short-Time Energy with Hamming Window')
    plt.xlabel('Frame Index')
    plt.ylabel('Energy')
    plt.xlim(0,math.ceil(ste_mag.shape[0]/sample_rate))
    plt.ylim(0,0.5)
    plt.legend()

    # Plot results
    plt.subplot(4, 1, 3)
    plt.plot(t, ste_mag, color='blue')
    plt.title('Short-Time Magnitude with Hamming Window')
    plt.xlabel('Frame Index')
    plt.ylabel('Magnitude')
    plt.xlim(0,math.ceil(ste_mag.shape[0]/sample_rate))
    plt.ylim(0,np.max(ste_mag))

    # Plot results
    plt.subplot(4, 1, 4)
    plt.plot(t, ste_zcr, color='blue')
    plt.title('Short-Time Zero Crossing with Rectangular Window')
    plt.xlabel('Frame Index')
    plt.ylabel('ZCR')
    plt.xlim(0,math.ceil(ste_zcr.shape[0]/sample_rate))
```

Solution Exercise 2: Pitch Detection with Autocorrelation

```python
    import numpy as np
import librosa
import matplotlib.pyplot as plt
import scipy.signal.windows as windows
from matplotlib.ticker import MultipleLocator


def autocorrelation(frame):
    """
    Compute the autocorrelation of a frame.
    """
    frame = frame - np.mean(frame)  # Remove DC component
    result = np.correlate(frame, frame, mode='full')
    return result[result.size // 2:] # positive value only

def compute_pitch(signal, sr, frame_size=1024, hop_size=512, fmin=50, fmax=500):
    """
    Compute pitch using short-time autocorrelation function.
    """
    pitches = []
    times = []

    for i in range(0, len(signal) - frame_size, hop_size):
        frame = signal[i:i + frame_size]
        frame =  frame * windows.hamming(frame_size)
        acf = autocorrelation(frame)

        # Find the first peak after lag 0
        min_lag = sr // fmax
        max_lag = sr // fmin
```

Solution Exercise 2: Pitch Detection with Autocorrelation

```
        acf[:min_lag] = 0  # Ignore low lags
        peak_index = np.argmax(acf[:max_lag])

        if acf[peak_index] > 0:
            pitch = sr / peak_index
        else:
            pitch = 0  # Unvoiced frame

        pitches.append(pitch)
        times.append(i / sr)

    return np.array(times), np.array(pitches)
```

Solution Exercise 2: Pitch Detection with Autocorrelation

```python
    # Load an example audio file
signal, sr = librosa.load('./audio/ih-pout-410Hz-8kHz.wav', sr=None)
# Compute pitch
times, pitches = compute_pitch(signal, sr, frame_size=240, hop_size=80)
# Plot pitch track
plt.figure(figsize=(10, 4))
plt.subplot(2, 1, 1)
plt.plot(times, pitches, label='Estimated Pitch', color='b', marker='o')
print(times)
plt.xlabel('Time (s)')
plt.ylabel('Pitch (Hz)')
plt.title('Pitch Estimation using Autocorrelation')
plt.legend()
plt.grid(True)
plt.subplot(2, 1, 2)
plt.plot(np.linspace(0, len(signal)/sr, len(signal)), signal, label='Speech Signal', color='r')
plt.xlabel('Time (s)')
plt.ylabel('Pitch (Hz)')
plt.legend()
plt.grid(True)
plt.show()
```

Solution Exercise 3: VUS detector

```
import numpy as np
import librosa
import matplotlib.pyplot as plt
import scipy.signal.windows as windows
from matplotlib.ticker import MultipleLocator
from scipy.interpolate import interp1d
import scipy.signal as signal
from matplotlib.ticker import MaxNLocator
def vus(s,  frame_size, hop_size):
    # Remove DC component
    s = s - np.mean(s)
    # Signal length
    D = len(s)
    # Frame Length
    L = frame_size
    # Frame shift
    U = hop_size
    # Window type
    win = windows.hamming(L)
    # Number of frames
    Nfr = int(np.floor((D-L)/U)+1)
    # Memory allocation (for speed)
    En = np.zeros(Nfr)
    ZCr = np.zeros(Nfr)
    T =  np.zeros(Nfr) # Next analysis time  instants
    for i in range(Nfr):
        start = i * U
        end = start + L
        frame = np.zeros(L)
        frame[:end-start] = s[start:end]*win
        En[i] = np.sum(frame**2)/L # it is defined as an average energy in the
             frame.
        ZCr[i] = np.sum(0.5*np.abs(np.diff(np.sign(frame))))
        T[i] = L/2 + i*U # Next analysis time instant
```

Solution Exercise 3: VUS detector

```
        # THRESHOLDS
    Ethres = np.mean(En)/2
    ZCRthres = (3/2)*np.mean(ZCr) - 0.3*np.std(ZCr)

    VUS = np.zeros(Nfr)

    # Classification for each frame
    for i in range(Nfr):
        if En[i] > Ethres:
            # Voiced
            VUS[i] = 1.0
        elif ZCr[i] < ZCRthres:
            # SILENCE
            VUS[i] = 0.0
        elif ZCr[i] > ZCRthres:
            # UNVOICED
            VUS[i] = 0.5

    return T, VUS
```

Solution Exercise 3: VUS detector

```
    s, sr = librosa.load('./audio/H.22.16k.wav', sr=None)
s = s - np.mean(s)
frame_rate = 0.01
frame_length = 0.03
frame_size = int(frame_length*sr)
hop_size = int(frame_rate*sr)
# Define frames
# Peak normalize the audio to range [-1, 1]
s = s / np.max(np.abs(s))
T, VUS_values = vus(s, frame_size, hop_size)
T, index = np.unique(T, return_index=True)
new_T = np.arange(0, len(s), 1)  # New points for interpolation (1:1:D in MATLAB
    )
interpolator = interp1d(T, VUS_values[index], kind='linear', fill_value=0,
    bounds_error=False)
VUS_i = interpolator(new_T)
t = np.linspace(0, len(s)/sr, len(s))
plt.plot(t, s, label='Speech Signal', color='r')
plt.plot(t, VUS_i, label='VUS', color='b')

plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.ylim(-1.0, +1.0)
plt.legend()
plt.grid(True)
plt.title("VUS discriminator")
plt.show()
```

Solution Exercise 4: Pitch Tracker

```
import numpy as np
import librosa
import matplotlib.pyplot as plt
import scipy.signal.windows as windows
from matplotlib.ticker import MultipleLocator
from scipy.interpolate import interp1d
from scipy.interpolate import CubicSpline
import scipy.signal as signal
from matplotlib.ticker import MaxNLocator

def vus(s,  frame_size=1024, hop_size=512):
    # Remove DC component
    s = s - np.mean(s)
    # Signal length
    D = len(s)
    # Frame Length
    L = frame_size
    # Frame shift
    U = hop_size
    # Window type
    win = windows.hamming(L)
    # Number of frames
    Nfr = int(np.floor((D-L)/U)+1)
    # Memory allocation (for speed)
    En = np.zeros(Nfr)
    ZCr = np.zeros(Nfr)
    T =  np.zeros(Nfr) # Next analysis time  instants
    for i in range(Nfr):
        start = i * U
        end = start + L
        frame = np.zeros(L)
        frame[:end-start] = s[start:end]*win
        En[i] = np.sum(frame**2)/L
        ZCr[i] = np.sum(0.5*np.abs(np.diff(np.sign(frame))))
```

Solution Exercise 4: Pitch Tracker

```
    T[i] = L/2 + i*U # Next analysis time instant

    # THRESHOLDS
    Ethres = np.mean(En)/1.2
    ZCRthres = (3/2)*np.mean(ZCr) - 0.3*np.std(ZCr)
    VUS = np.zeros(Nfr)
    # Classification for each frame
    for i in range(Nfr):
        if En[i] > Ethres:
            # Voiced
            VUS[i] = 1.0
        elif ZCr[i] < ZCRthres:
            # SILENCE
            VUS[i] = 0.0
        elif ZCr[i] > ZCRthres:
            # UNVOICED
            VUS[i] = 0.5
    return VUS
```

Solution Exercise 4: Pitch Tracker

```python
def acf_peak_picking(frame, sr, fmin=70, fmax=500):
L = len(frame)

np.set_printoptions(precision=4, suppress=False)
corr = np.correlate(frame, frame, mode='full')
pos_corr = corr[corr.size // 2:] # positive axes only

up_thresh = 1/fmin
low_thresh = 1/fmax

# Find peaks in the autocorrelation function
peaks, _ = signal.find_peaks(pos_corr)
pks_value = pos_corr[peaks]
locs = peaks/sr # Convert location of peaks to time values

valid_locs = locs[(locs >= low_thresh) & (locs <= up_thresh)]
valid_peaks = pks_value[(locs >= low_thresh) & (locs <= up_thresh)]
argmax = np.argmax(valid_peaks)
f_0 = 1/valid_locs[argmax]
return f_0
```

Solution Exercise 4: Pitch Tracker

```python
def compute_pitch(s, sr, frame_size=240, hop_size=80, fmin=50, fmax=500):
    """
    Compute pitch using short-time autocorrelation function.
    """

    VUS = vus(s, frame_size=frame_size, hop_size=hop_size)
    s = s-np.mean(s) # Remove DC component
    # Compute pitch
    Nfr = int(np.floor((len(s)-frame_size)/hop_size)+1)
    f_acf = np.zeros(Nfr)
    T = np.zeros(Nfr)

    for i in range(Nfr):
        start = i*hop_size
        end = start + frame_size
        frame = np.zeros(frame_size)
        frame = s[start:end]*windows.hamming(frame_size)
        if VUS[i] == 1.0:
            f_acf[i] = acf_peak_picking(frame, sr)
        else:
            f_acf[i] = 0
        T[i] = frame_size/2 + i*hop_size
    return T, f_acf
```

Solution Exercise 4: Pitch Tracker

```
    import soundfile as sf
s, sr = sf.read('./audio/H.22.16k.wav')

# Peak normalize the audio to range [-1, 1]
s = s / np.max(np.abs(s))
T, f_acf = compute_pitch(s, sr, frame_size=240, hop_size=80)
T, index= np.unique(T, return_index=True)
new_T = np.arange(0, len(s), 1)  # New points for interpolation (1:1:D in MATLAB)
interpolator = interp1d(T, f_acf[index], kind='cubic', fill_value='extrapolate')
f_acf_i = interpolator(new_T)


# Classify the speech signal
f_acf_pos = f_acf[f_acf > 0]
f_acf_male = f_acf_pos[(f_acf_pos >= 70) & (f_acf_pos <= 160)]
f_acf_female = f_acf_pos[ (f_acf_pos > 160) & (f_acf_pos <= 275)]
f_acf_child = f_acf_pos[ (f_acf_pos > 275) & (f_acf_pos <= 500)]
result  = 'Adult Male'
max_len = len(f_acf_male)
if max_len < len(f_acf_female):
    result = 'Adult Female'
    max_len = len(f_acf_female)
if max_len < len(f_acf_child):
    result = 'Child'
print(f'The voice is ',result)
```

Solution Exercise 4: Pitch Tracker

```
# Plot pitch track
plt.figure(figsize=(10, 4))
plt.subplot(2, 1, 1)
t = np.linspace(0, len(s)/sr, len(s))
plt.plot(t, f_acf_i, label='Estimated Pitch', color='b')
plt.xlabel('Time (s)')
plt.ylabel('Pitch (Hz)')
plt.title('Pitch Estimation using Autocorrelation')
plt.legend()
plt.ylim(np.min(f_acf_i),np.max(f_acf_i)+100)
ax = plt.gca()
# Customize the y-axis tick locator
#ax.yaxis.set_major_locator(MaxNLocator(integer=True, prune='both', steps=[1, 2,
      5, 10]))
ax.yaxis.set_major_locator(MultipleLocator(50))
plt.grid(True)
plt.subplot(2, 1, 2)
plt.plot(t, s, label='Speech Signal', color='r')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()
plt.grid(True)
plt.show()
```