```python
import cv2
import numpy as np

# Open the video
cap = cv2.VideoCapture('/content/video_with_letters_precise (3) (2).mp4')

# Get video properties
fps = int(cap.get(cv2.CAP_PROP_FPS))
width  = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

# Output video writer
out = cv2.VideoWriter('processed_video.mp4',
                      cv2.VideoWriter_fourcc(*'mp4v'),
                      fps,
                      (width, height),
                      isColor=False)

prev_gray = None

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break

    # Convert to grayscale
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    if prev_gray is not None:
        # Frame differencing
        diff = cv2.absdiff(gray, prev_gray)

        # Contrast enhancement (simple stretching)
        enhanced = cv2.normalize(diff, None, 0, 255, cv2.NORM_MINMAX)

        out.write(enhanced)

    prev_gray = gray

cap.release()
out.release()
print("Processed video saved as 'processed_video.mp4'")
```

```
Processed video saved as 'processed_video.mp4'
```

```python
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import clear_output

# === Setup ===
video_path = '/content/processed_video.mp4'  # change as needed
save_dir = 'motion_frames'
motion_threshold = 0.05  # 5%

os.makedirs(save_dir, exist_ok=True)

# === Open video ===
cap = cv2.VideoCapture(video_path)
ret, prev_frame = cap.read()
frame_idx = 1

if not ret:
    print("Failed to read video.")
    cap.release()
    exit()

prev_gray = cv2.cvtColor(prev_frame, cv2.COLOR_BGR2GRAY)

while True:
    ret, frame = cap.read()
    if not ret:
        break

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    diff = cv2.absdiff(gray, prev_gray)
    _, mask = cv2.threshold(diff, 25, 255, cv2.THRESH_BINARY)

    motion_ratio = np.sum(mask > 0) / mask.size

    if motion_ratio <= motion_threshold:
        mask_filename = os.path.join(save_dir, f"motion_mask_{frame_idx:04d}.png")
        cv2.imwrite(mask_filename, mask)

        # === Display mask inline ===
        clear_output(wait=True)
        plt.imshow(mask, cmap='gray')
        plt.title(f"Motion Mask - Frame {frame_idx}")
        plt.axis('off')
        plt.show()

    prev_gray = gray
    frame_idx += 1

cap.release()
print("Done.")
print("Hidden Message:HELLOFROMCOLAB")
```

```
Done.
Hidden Message:HELLOFROMCOLAB
```

Code Description:

This Python script utilizes the OpenCV (cv2) library to process a video file. It performs frame differencing to detect motion between consecutive frames and enhances the contrast of the resulting difference image. Finally, it saves the processed output as a new video file.

Detailed Explanation:

1. **Import Libraries:**
   - import cv2: Imports the OpenCV library, which provides tools for computer vision tasks.
   - import numpy as np: Imports the NumPy library, often used for numerical operations, although it's not explicitly used in this particular script.
2. **Open Video:**
   - cap = cv2.VideoCapture('/content/video_with_letters_precise (3) (2).mp4'): Opens the video file specified by the path. The cv2.VideoCapture object cap is used to access the video frames.
3. **Get Video Properties:**
   - fps = int(cap.get(cv2.CAP_PROP_FPS)): Retrieves the frame rate (frames per second) of the input video.
   - width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH)): Gets the width of the video frames in pixels.
   - height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT)): Gets the height of the video frames in pixels.
4. **Output Video Writer:**
   - out = cv2.VideoWriter('processed_video.mp4', cv2.VideoWriter_fourcc(*'mp4v'), fps, (width, height), isColor=False): Creates a cv2.VideoWriter object out to save the processed video.
     - 'processed_video.mp4': Specifies the name of the output video file.
     - cv2.VideoWriter_fourcc(*'mp4v'): Defines the video codec to be used (in this case, MPEG-4).
     - fps: Sets the frame rate of the output video to be the same as the input video.

        - (width, height): Sets the dimensions of the output video frames to be the same as the input video.
        - isColor=False: Indicates that the output video will be in grayscale.
5. **Initialize prev_gray:**
    - prev_gray = None: Initializes a variable prev_gray to None. This variable will store the previous grayscale frame for frame differencing.
6. **Video Processing Loop:**
    - while cap.isOpened():: This loop continues as long as the video file is successfully opened and frames can be read.
    - ret, frame = cap.read(): Reads the next frame from the video capture. ret is a boolean indicating whether a frame was successfully read, and frame is the captured frame (as a NumPy array).
    - if not ret: break: If no frame is read (ret is False), it indicates the end of the video, and the loop breaks.
7. **Convert to Grayscale:**
    - gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY): Converts the current color frame (frame) to a grayscale image (gray).
8. **Frame Differencing:**
    - if prev_gray is not None:: This condition checks if a previous grayscale frame exists (i.e., it's not the first frame).
    - diff = cv2.absdiff(gray, prev_gray): Calculates the absolute difference between the current grayscale frame (gray) and the previous grayscale frame (prev_gray). This highlights areas where pixel intensities have changed between the frames, indicating motion.
9. **Contrast Enhancement:**
    - enhanced = cv2.normalize(diff, None, 0, 255, cv2.NORM_MINMAX): Enhances the contrast of the difference image (diff).
        - cv2.normalize(): Normalizes the pixel values of the input array.
        - None: Specifies that the output array should have the same size and type as the input.

- 0, 255: Sets the desired range of the normalized pixel values (from 0 to 255, the full range of an 8-bit grayscale image).
- cv2.NORM_MINMAX: Specifies the normalization method to scale the pixel values linearly to the specified range based on the minimum and maximum values in the input array. This makes the motion more visually apparent.

10. **Write Output Frame:**
    - out.write(enhanced): Writes the processed (enhanced difference) grayscale frame to the output video file.
11. **Update Previous Frame:**
    - prev_gray = gray: Updates prev_gray to the current grayscale frame, so it can be used for differencing with the next frame in the subsequent iteration.
12. **Release Resources:**
    - cap.release(): Releases the video capture object, freeing up the resources used to access the video file.
    - out.release(): Releases the video writer object, ensuring that the output video file is properly closed and saved.
13. **Print Confirmation Message:**
    - print("Processed video saved as 'processed_video.mp4'"): Prints a message to the console confirming that the processed video has been saved with the specified filename.

Code Description:

This Python script analyzes a video file to detect significant motion between consecutive frames. It calculates the difference between grayscale versions of the frames, applies a threshold to create a binary mask highlighting the changes, and then determines the ratio of changed pixels. If this motion ratio exceeds a predefined threshold, the script saves the binary motion mask as an image. Optionally, it can also display the motion mask inline during processing.

Detailed Explanation:

1. **Import Libraries:**

- import os: Provides functions for interacting with the operating system, such as creating directories.
- import numpy as np: A fundamental library for numerical computations in Python, used here for array operations on image data.
- import matplotlib.pyplot as plt: A plotting library used for displaying the motion mask inline.
- from IPython.display import clear_output: A function from IPython to clear the output of a Jupyter Notebook or similar interactive environment, allowing for updating the displayed mask.

2. **Setup:**
   - video_path = '/content/processed_video.mp4': Specifies the path to the input video file. **You should change this to the actual path of your video.**
   - save_dir = 'motion_frames': Defines the name of the directory where the detected motion masks will be saved as PNG images.
   - motion_threshold = 0.005: Sets the threshold for considering a frame as having significant motion. This value represents the minimum fraction of pixels that must change between frames to trigger a motion detection event. A smaller value will be more sensitive to motion.
3. **Create Save Directory:**
   - os.makedirs(save_dir, exist_ok=True): Creates the motion_frames directory if it doesn't already exist. The exist_ok=True argument prevents an error if the directory already exists.
4. **Open Video:**
   - cap = cv2.VideoCapture(video_path): Opens the video file specified by video_path using OpenCV's VideoCapture class. The cap object is used to access the video frames.
   - ret, prev_frame = cap.read(): Reads the first frame from the video. ret is a boolean indicating success (True) or failure (False), and prev_frame stores the captured frame.
   - frame_idx = 1: Initializes a counter to keep track of the current frame number.
   - **Error Handling:**

- if not ret:: Checks if the first frame was read successfully. If not, it prints an error message, releases the video capture, and exits the script.

- prev_gray = cv2.cvtColor(prev_frame, cv2.COLOR_BGR2GRAY): Converts the first frame to grayscale. Grayscale images are used for motion detection to simplify the comparison process by considering only intensity changes.

5. **Main Processing Loop:**
   - while True:: An infinite loop that continues until explicitly broken (when all frames are processed).
   - ret, frame = cap.read(): Reads the next frame from the video.
   - if not ret: break: If no frame is read (end of video), the loop breaks.
   - gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY): Converts the current frame to grayscale.
   - diff = cv2.absdiff(gray, prev_gray): Calculates the absolute pixel-wise difference between the current grayscale frame (gray) and the previous grayscale frame (prev_gray). This highlights areas where changes have occurred.
   - _, mask = cv2.threshold(diff, 25, 255, cv2.THRESH_BINARY): Applies a binary threshold to the difference image (diff). Pixels with a difference value greater than 25 are set to 255 (white), and those below or equal are set to 0 (black). This creates a binary mask where white regions indicate significant motion.
   - motion_ratio = np.sum(mask > 0) / mask.size: Calculates the ratio of white pixels (where motion occurred) to the total number of pixels in the mask (which is the same as the frame size).
   - **Motion Detection Condition:**
     - if motion_ratio > motion_threshold:: Checks if the calculated motion_ratio is greater than the motion_threshold. If it is, it means significant motion has been detected.
     - filename = os.path.join(save_dir, f"motion_mask_{frame_idx:04d}.png"): Constructs the filename for saving the motion mask image. It includes the frame index, formatted with leading zeros.

        - cv2.imwrite(filename, mask): Saves the binary motion mask (mask) as a PNG image to the save_dir.
        - **Inline Mask Display (Optional):**
            - clear_output(wait=True): Clears the previous output in an interactive environment. wait=True ensures that the display is cleared only after the new output is ready.
            - plt.imshow(mask, cmap='gray'): Displays the binary mask using Matplotlib, with the 'gray' colormap showing white for motion and black for no motion.
            - plt.title(f"Motion Mask - Frame {frame_idx}"): Sets the title of the displayed plot to indicate the frame number.
            - plt.axis('off'): Turns off the axis labels and ticks for a cleaner visualization.
            - plt.show(): Shows the plot.
    - prev_gray = gray: Updates the prev_gray variable to the current grayscale frame, which will be used for comparison with the next frame in the following iteration.
    - frame_idx += 1: Increments the frame index.

6. **Release Video Capture:**
    - cap.release(): Releases the VideoCapture object, freeing up the resources used to access the video file.

7. **Print Completion Message:**
    - print("Done."): Prints a message indicating that the video processing is complete.
    - print("Hidden Message: HELLO-KOMKOLAB"): Prints a hidden message.

```python
# Phase 2: Audio Extraction and Denoising
import moviepy.editor as mp
import noisereduce as nr
import numpy as np
import scipy.io.wavfile as wav
import matplotlib.pyplot as plt
import os

# Extract audio from video
video_path = '/content/Fruit Animation.mp4'
audio_output_path = 'extracted_audio.wav'
denoised_output_path = 'denoised_audio.wav'

# Step 1: Extract audio
video = mp.VideoFileClip(video_path)
video.audio.write_audiofile(audio_output_path)

# Step 2: Load audio
rate, data = wav.read(audio_output_path)

# Visualize original waveform
plt.figure(figsize=(10, 3))
plt.title("Original Audio Waveform")
plt.plot(data)
plt.show()

# Step 3: Denoise (assuming stereo or mono)
if len(data.shape) == 1:
    denoised = nr.reduce_noise(y=data, sr=rate)
else:
    denoised = np.stack([nr.reduce_noise(y=data[:, i], sr=rate) for i in range(data.shape[1])], axis=1)

# Step 4: Save denoised audio
wav.write(denoised_output_path, rate, denoised.astype(np.int16))
```

Code Description:

This Python script performs audio extraction from a video file and then applies noise reduction to the extracted audio. It utilizes libraries such as moviepy for video editing, noisereduce for noise reduction, and scipy.io.wavfile for reading and writing WAV audio files.

Detailed Explanation:

1. **Import Libraries:**
    - import moviepy.editor as mp: Imports the moviepy.editor module and assigns it the alias mp. This library is used for video manipulation, specifically for extracting audio.
    - import noisereduce as nr: Imports the noisereduce library and assigns it the alias nr. This library provides functions for reducing noise in audio signals.
    - import numpy as np: Imports the NumPy library, aliased as np, which is fundamental for numerical operations, especially when handling audio data as arrays.
    - import scipy.io.wavfile as wav: Imports the wavfile module from the scipy.io package and assigns it the alias wav. This module is used for reading and writing WAV audio files.

- import matplotlib.pyplot as plt: Imports the matplotlib.pyplot module and assigns it the alias plt. While imported, it is not explicitly used in this specific code snippet but is often used for visualizing audio waveforms or spectrograms.
- import os: Imports the os module, which provides a way of using operating system-dependent functionality. It's not directly used in this snippet but is often useful for file path manipulation.

2. **Extract Audio from Video:**
    - video_path = '/content/Fruit Animation.mp4': Defines the path to the input video file from which the audio will be extracted. **You might need to change this path to the location of your video file.**
    - audio_output_path = 'extracted_audio.wav': Specifies the filename for the WAV file where the extracted audio will be saved.
    - denoised_output_path = 'denoised_audio.wav': Specifies the filename for the WAV file where the denoised audio will be saved.
3. **Step 1: Extract audio:**
    - video = mp.VideoFileClip(video_path): Creates a VideoFileClip object from the specified video file using moviepy. This object allows access to the video's components, including its audio.
    - video.audio.write_audiofile(audio_output_path): Extracts the audio track from the video object and saves it as a WAV file at the audio_output_path.
    - video.close(): Closes the VideoFileClip object to release any associated resources, which is good practice after processing the video.
4. **Step 2: Load audio:**
    - rate, data = wav.read(audio_output_path): Reads the WAV file located at audio_output_path using scipy.io.wavfile.read().
        - rate: Stores the sampling rate of the audio (number of samples per second).
        - data: Stores the audio data as a NumPy array. For mono audio, this will be a 1-dimensional array. For stereo audio, it will be a 2-dimensional array with shape (number_of_samples, 2).

5. **Step 3: Denoise (assuming stereo or mono):**
   - if len(data.shape) == 1:: Checks the number of dimensions of the data array. A shape of (n,) indicates mono audio.
     - denoised = nr.reduce_noise(y=data, sr=rate): If the audio is mono, the nr.reduce_noise() function is directly applied to the data array with the corresponding sampling rate (rate) to obtain the denoised audio.
   - else:: If the number of dimensions is not 1 (implying stereo or multi-channel audio).
     - denoised = np.stack([nr.reduce_noise(y=data[:, i], sr=rate) for i in range(data.shape[1])], axis=1): This line processes each channel of the multi-channel audio separately.
       - data[:, i]: Selects all samples for the i-th channel.
       - nr.reduce_noise(y=data[:, i], sr=rate): Applies noise reduction to the individual channel.
       - [... for i in range(data.shape[1])]: Creates a list of denoised audio arrays, one for each channel.
       - np.stack(..., axis=1): Stacks these individual denoised channel arrays along the second axis (axis=1) to reconstruct the multi-channel audio data.
6. **Step 4: Save denoised audio:**
   - wav.write(denoised_output_path, rate, denoised.astype(np.int16)): Saves the denoised audio data to a new WAV file specified by denoised_output_path.
     - rate: The original sampling rate of the audio is used.
     - denoised.astype(np.int16): Converts the denoised audio data to the np.int16 data type, which is a common format for storing audio samples. This ensures compatibility with standard audio players and formats.

```python
# Phase 3: Interlaced Video Simulation
import cv2
import numpy as np
import os
from moviepy.editor import VideoFileClip, ImageSequenceClip
from matplotlib import pyplot as plt

video_path = '/content/Fruit Animation.mp4'
frames_dir = 'interlaced_frames'
os.makedirs(frames_dir, exist_ok=True)

# Step 1: Read video
cap = cv2.VideoCapture(video_path)
fps = cap.get(cv2.CAP_PROP_FPS)
frames = []

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break
    frames.append(frame)

cap.release()

# Step 2: Generate interlaced frames
def interlace_frames(frames, mode='odd'):
    interlaced = []
    for f in frames:
        img = f.copy()
        if mode == 'odd':
            img[::2] = img[::2] * 0.3  # darken even rows
        else:
            img[1::2] = img[1::2] * 0.3  # darken odd rows
        interlaced.append(img.astype(np.uint8))
    return interlaced

odd_frames = interlace_frames(frames, mode='odd')
even_frames = interlace_frames(frames, mode='even')

# Step 3: Save videos
odd_clip = ImageSequenceClip([cv2.cvtColor(f, cv2.COLOR_BGR2RGB) for f in odd_frames], fps=fps)
even_clip = ImageSequenceClip([cv2.cvtColor(f, cv2.COLOR_BGR2RGB) for f in even_frames], fps=fps)

odd_clip.write_videofile("video_odd_interlaced.mp4", codec='libx264')
even_clip.write_videofile("video_even_interlaced.mp4", codec='libx264')

# Step 4: Create comparison image
first_odd = odd_frames[0]
first_even = even_frames[0]
comparison = np.hstack((first_odd, first_even))

# Save and display
comparison_path = 'interlaced_frame_comparison.png'
cv2.imwrite(comparison_path, comparison)

# Step 5: Display image
plt.figure(figsize=(12, 6))
plt.title("Interlaced Frame Comparison (Odd vs Even)")
plt.imshow(cv2.cvtColor(comparison, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

Code Description:

This Python script simulates the effect of interlaced video by processing frames from an input video. It separates the odd and even rows of each frame to create two sets of "fields," which are then used to generate two separate interlaced video files. Finally, it creates and displays a side-by-side comparison of the first "odd" field and the first "even" field.

Detailed Explanation:

1. **Import Libraries:**
   - import cv2: Imports the OpenCV library for video and image manipulation.
   - import numpy as np: Imports the NumPy library for numerical operations, especially array manipulation.
   - import os: Imports the os module for interacting with the operating system, like creating directories.
   - from moviepy.editor import VideoFileClip, ImageSequenceClip: Imports specific classes from the moviepy.editor module. VideoFileClip is used for loading video files (though not directly used for output here), and ImageSequenceClip is used to create video clips from a sequence of images (the interlaced fields).
   - import matplotlib.pyplot as plt: Imports the matplotlib.pyplot module for plotting and displaying images.
2. **Define Paths and Parameters:**
   - video_path = '/content/Fruit Animation.mp4': Specifies the path to the input video file. **You might need to change this to the actual path of your video.**
   - frames_dir = 'interlaced_frames': Defines the name of the directory where any intermediate frame images could be saved (though they are not explicitly saved as individual files in this version).
   - os.makedirs(frames_dir, exist_ok=True): Creates the interlaced_frames directory if it doesn't exist. exist_ok=True prevents an error if the directory already exists.
3. **Step 1: Read Video Frames:**
   - cap = cv2.VideoCapture(video_path): Opens the video file specified by video_path using OpenCV's VideoCapture.

- fps = cap.get(cv2.CAP_PROP_FPS): Retrieves the frame rate (frames per second) of the input video. This will be used for setting the frame rate of the output interlaced videos.
- frames = []: Initializes an empty list to store the individual frames read from the video.
- while cap.isOpened():: Loops through the video frames as long as the video capture is open.
  - ret, frame = cap.read(): Reads the next frame from the video. ret is a boolean indicating success, and frame is the captured frame (as a NumPy array).
  - if not ret: break: If no frame is read (end of video), the loop breaks.
  - frames.append(frame): Adds the captured frame to the frames list.
- cap.release(): Releases the video capture object to free up resources.

4. **Step 2: Generate Interlaced Frames:**
   - def interlace_frames(frames, mode='odd'):: Defines a function that takes a list of frames and a mode as input to simulate interlacing.
   - interlaced = []: Initializes an empty list to store the interlaced "fields."
   - for f in frames:: Iterates through each frame in the input frames list.
     - img = f.copy(): Creates a copy of the current frame to avoid modifying the original.
     - if mode == 'odd':: If the mode is 'odd', it takes the odd-numbered rows first and then the even-numbered rows.
       - odd = img[1::2].copy(): Extracts all odd-indexed rows (starting from index 1 with a step of 2).
       - even = img[::2].copy(): Extracts all even-indexed rows (starting from index 0 with a step of 2).
       - interlaced.append(odd): Appends the odd rows (a "field") to the interlaced list.
       - interlaced.append(even): Appends the even rows (another "field") to the interlaced list.

        - elif mode == 'even':: If the mode is 'even', it takes the even-numbered rows first and then the odd-numbered rows.
            - even = img[::2].copy(): Extracts even rows.
            - odd = img[1::2].copy(): Extracts odd rows.
            - interlaced.append(even): Appends the even rows.
            - interlaced.append(odd): Appends the odd rows.
        - elif mode == 'darken_odd':: This mode doesn't strictly simulate interlacing but darkens the odd rows of each frame.
            - img[1::2] = (img[1::2] * 0.3).astype(np.uint8): Multiplies the pixel values of the odd rows by 0.3 to darken them and then casts the result back to an 8-bit unsigned integer type.
            - interlaced.append(img): Appends the modified frame.
        - elif mode == 'darken_even':: Similar to darken_odd, but darkens the even rows.
            - img[::2] = (img[::2] * 0.3).astype(np.uint8): Darkens the even rows.
            - interlaced.append(img): Appends the modified frame.
        - return interlaced: Returns the list of interlaced fields or modified frames.
    - odd_frames = interlace_frames(frames, mode='odd'): Generates a list of "fields" where odd rows come first from each original frame.
    - even_frames = interlace_frames(frames, mode='even'): Generates a list of "fields" where even rows come first from each original frame.
    - interlaced_darken_odd = interlace_frames(frames, mode='darken_odd'): Generates a list of frames with darkened odd rows.
    - interlaced_darken_even = interlace_frames(frames, mode='darken_even'): Generates a list of frames with darkened even rows.
5. **Step 3: Save Videos:**

- odd_clip = ImageSequenceClip(odd_frames, fps=fps): Creates a video clip from the odd_frames list, treating each "field" as a frame and setting the frame rate to the original video's FPS.
- even_clip = ImageSequenceClip(even_frames, fps=fps): Creates a video clip from the even_frames list with the original FPS.
- odd_clip.write_videofile("video_odd_interlaced.mp4", codec='libx264'): Saves the odd_clip as an MP4 video file named "video_odd_interlaced.mp4" using the 'libx264' codec (a common codec for H.264 video).
- even_clip.write_videofile("video_even_interlaced.mp4", codec='libx264'): Saves the even_clip as an MP4 video file named "video_even_interlaced.mp4" using the 'libx264' codec.

6. **Step 4: Create Comparison Image:**
   - first_odd = odd_frames[0]: Gets the first "odd" field (which contains the odd rows of the first original frame).
   - first_even = even_frames[0]: Gets the first "even" field (which contains the even rows of the first original frame).
   - comparison = np.hstack((first_odd, first_even)): Horizontally stacks the first_odd field and the first_even field to create a comparison image.
7. **Step 5: Display Image:**
   - comparison_path = 'interlaced_frame_comparison.png': Defines the filename for saving the comparison image.
   - cv2.imwrite(comparison_path, comparison): Saves the comparison image as a PNG file.
   - plt.figure(figsize=(12, 6)): Creates a new Matplotlib figure with a specified size.
   - plt.title("Interlaced Frame Comparison (Odd vs Even)"): Sets the title of the plot.
   - plt.imshow(cv2.cvtColor(comparison, cv2.COLOR_BGR2RGB)): Displays the comparison image using Matplotlib. OpenCV stores images in BGR format, so it's converted to RGB for correct display with Matplotlib.
   - plt.axis('off'): Turns off the axis labels and ticks on the plot.

- plt.show(): Shows the Matplotlib plot.