Introduction to Video Processing

HW4

Hong Shiang Lin

National Taipei University

hongshianglin@gm.ntpu.edu.tw

March 13, 2025

HW4: Fourier Transform for Translating Sinusoidal Video

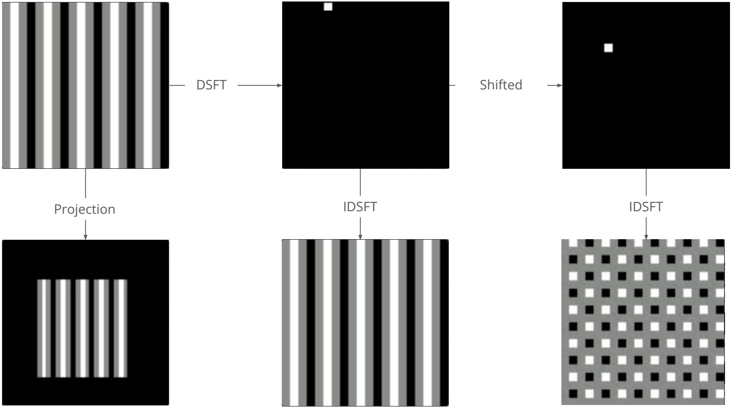


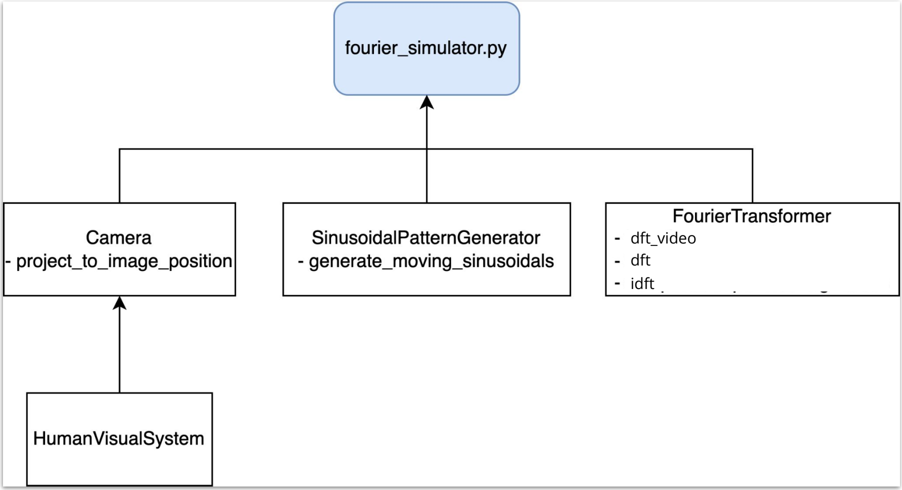
Figure: Overview of the workflow.

You should complete the following tasks:

- Complete the provided codes.
- Provide a HW report (see report guidelines).

Code Completion

(*Code Architecture*)



(Tracing on **fourier_simulator.py**)

```
from sinusoidal_pattern_generator import *
from fourier_transformer import *
from visualizer import *
import os.path

fx_vid = 0.5 # Cycles per mm along horizontal direction
fy_vid = 0.0 # Cycles per mm along vertical direction
vx_vid = 0.2 # Speed along horizontal direction mm/sec
vy_vid = 0.2 # Speed along vertical direction in mm/sec
Tb = 0 # Beginning of the time
dt = 1 # Frame time in seconds
Te = 30 # End of the time in seconds
fps = (Te - Tb) / dt
```

- The spatial temporal signal is a moving sinusoidal pattern.
- *fx_vid, fy_vid, vx_vid, vy_vid, Tb, dt, Te* characterize frequencies of the 3D sinusoidal signal.

(Continue)

```
# Physical Length of the image plane
h_mm = 10
w_mm = 10
d_mm = 15

# Height, width, and interval in pixels
height = 100
width = 100
intv = 5 # Interval for downsampling the frequency and spatial temporal
    variables
```

- *h_mm, w_mm, d_mm, height, width, intv* characterizes the resolution of the signal.

(Continue)

```
print('Generating the sinusoidal signal...')
pattern_generator = SinusoidalPatternGenerator(w_mm, h_mm, width, height, Tb, Te)
spatial_temporal_signal = pattern_generator.generate_moving_sinusoidals(fx_vid, fy_vid, fps, vx_vid, vy_vid, intv)

# Viz for Spatial Temporal Signals
visualize_spatial_temporal_signal(folder_out + '/spatial_temporal.mp4', spatial_temporal_signal, width, height, fps, intv)

# Viz the viewer perspective results
visualize_viewed_signal(folder_out + '/viewing.mp4', spatial_temporal_signal, width, height, fps, intv, w_mm, h_mm, d_mm)
```

- The spatial temporal signal is generated by *generate_moving_sinusoidals*.
- We utilize two visualization functions for visualizing original spatial temporal signal and its viewing version.

(Continue)

```python
xformer = FourierTransformer()

# Compute frequency response
print('Computing Fourier transform for the sinusoidal signal...')
frequency_signal = xformer.dft_video(spatial_temporal_signal)

# Cache the results for testing or debugging
cache_signal(folder_out + '/frequency.sg', frequency_signal)
'''
# Load cached signal if you do not want to compute the same transform again, used for debugging.
frequency_signal = load_signal(folder_out + '/frequency.sg', spatial_temporal_signal.shape)
'''

# Viz for Frequency Response
visualize_frequency_signal(folder_out + '/frequency_response.mp4', frequency_signal, width, height, fps, intv)
```

- Create the fourier transformer.
- Use *dft_video* for the video transformation.
- We can cache or load signal for debugging or testing.

(Continue)

```
# Reconstruct spatial temporal signal via inverse fourier transform
print('Recovering the spatial temporal signal by inverse Fourier
    transform...')
recovered_signal = xformer.idft(frequency_signal)

# Cache reconstructed signal
cache_signal(folder_out + '/reconstruction.sg', recovered_signal)

# Visualize reconstructed signal (SHOULD be visually the same as
    original spatial temporal signal)
visualize_recovered_signal(folder_out + '/reconstruction.mp4',
    recovered_signal, width, height, fps, intv)
```

- We use *idft* for performing the inverse Fourier transform.
- The reconstructed signal should be ideally the same as original spatial temporal signal.

(Continue)

```
# Shift the frequency signal
print('Shifting the frequency signal...')
dx = 0
dy = 5
dt = 0
resized_height, resized_width, frames = frequency_signal.shape[0],
    frequency_signal.shape[1], frequency_signal.shape[2]
shifted_frequency_signal = np.zeros([resized_height, resized_width,
    frames], dtype=complex)
# TODO #2: Implement a shifting operation to move the frequency
    responses. The shifted responses are stored in
# shifted_frequency_signal

# Viz for Shifted Frequency signal
visualize_frequency_signal(folder_out + '/shifted_frequency.mp4',
    shifted_frequency_signal, width, height, fps, intv)
```

- Implement the frequency response shifting here by using *dx, dy, dt*.

(Continue)

```
# Generate the corresponding spatial temporal signal, it SHOULD be
    changed.
print('Generating new spatial temporal signal according to the shifted
    frequency signal...')
edited_spatial_temporal_signal = xformer.idft(shifted_frequency_signal)

# Cache the new spatial temporal signal
cache_signal(folder_out + '/edited_spatial_temporal.sg', edited_spatial
    _temporal_signal)

# Visualize the new spatial temporal signal
visualize_recovered_signal(folder_out + '/edited_spatial_temporal.mp4',
     edited_spatial_temporal_signal, width, height, fps, intv)
```

- Perform inverse Fourier transform on the shifted frequency signal, we will see the changed spatial temporal signal.

(Tracing on **fourier_transformer.py**)

```
class FourierTransformer():
    def __init__(self):
        pass

    '''
    Inputs:
        - video: 3D numpy array with real numbers
    Outputs:
        - output_signal: 3D numpy array with complex numbers
    '''
    def dft_video(self, video):
        # Convert the input video type into complex type
        video_complex = video.astype(complex)
        return self.dft(video_complex)
```

- The *dft_video* convert the type from real to complex.

(Continue)

```
    '''
    Inputs:
        - input_signal: 3D numpy array with complex numbers.
    Outputs:
        - output_signal: 3D numpy array with complex numbers.
    '''
    def dft(self, input_signal):
        # Get width, height, frames of the input_signal.
        height, width, frames = input_signal.real.shape[0], input_signal.real.shape[1], input_signal.real.shape[2]

        # Get frequency response by computing fourier dft on the input signals
        output_signal = np.zeros([height, width, frames], dtype=complex)

        # TODO #3: Implement 3D fourier transform for output_signal according to the DSFT formulas provided in the slides.
        # You are NOT ALLOWED to use any third party API to execute the Fourier transform

        return output_signal
```

(Continue)

```
    '''
    Inputs:
        - input_signal: 3D numpy array with complex numbers
    Outputs:
        - output_signal: 3D numpy array with complex numbers.
    '''
    def idft(self, input_signal):
        # Get width, height, frames of the input_signal.
        height, width, frames = input_signal.real.shape[0], input_
            signal.real.shape[1], input_signal.real.shape[2]

        # Get frequency response by computing fourier dft on the input
            signals
        output_signal = np.zeros([height, width, frames], dtype=complex
            )

        # TODO #4: Implement 3D inverse Fourier transform for output_
            signal according to the IDSFT formulas provided in the
            slides.
        # You are NOT ALLOWED to use any third party API to execute the
             inverse Fourier transform

        return output_signal
```

(Tracing on **visualizer.py**)

```
def visualize_spatial_temporal_signal(filename, signal, width, height,
     fps, intv):
   fourcc=cv2.VideoWriter_fourcc(*'mp4v')
   video = cv2.VideoWriter(filename, fourcc, fps, (width, height))
   image = np.zeros([height, width, 3], dtype = np.uint8)
   # Frame index of viz.
   f_viz = 0
   frames = signal.shape[2]
   for f in range(frames):
       for y in range(height):
           if y % intv == 0:
               # Get corrsponding y in signal
               resized_y = y // intv
               for x in range(width):
                   if x % intv == 0:
                       # Get corresponding x in signal
                       resized_x = x // intv
                       (See next page)
   video.release()
```

- The visualization images have more pixels than original signal.
- *resized_x, resized_y*: corresponding spatial position in the signal.

(Continue)

```
# Get corresponding signal
b = signal[resized_y, resized_x, f]

# Dump to signal if current frame is filled.
if (f > f_viz):
    video.write(image)
    f_viz += 1

# Normalization
b_viz = (b + 1) / 2 # Mapping the value from
    (-1, 1) to (0, 1)

# Draw response with thickness as the specified
    interval
image[y:y+intv, x:x+intv, :] = int(b_viz * 255)
    # Mapping to integer
```

- Visualize the signal by filling square areas for every 3D variable.

(Continue)

(Tracing on **sinusoidal_pattern_generator.py**)

```
class SinusoidalPatternGenerator():
    '''
    Inputs:
        w_mm: plane width in milli meters
        h_mm: plane height in milli meters
        width: number of pixels on the horizontal direction
        height: number of pixels on the vertical direction
        Tb: Beginning of the time
        Te: End of the time
    '''
    def __init__(self, w_mm, h_mm, width, height, Tb, Te):
        self.w_mm = w_mm
        self.h_mm = h_mm
        self.width = width
        self.height = height
        self.Te = Te
        self.Tb = Tb
```

- A moving sinusoidal pattern in a plane.

(Continue)

```
'''
Inputs:
    fx: frequency on horizontal direction in cycles per mm
    fy: frequency on vertical direction in cycles per mm
    ft: frequency on temporal direction in frames per sec
    vx: velocity along horizontal direction in mm/sec
    vy: velocity along vertical direction in mm/sec
    intv: Interval for downsampling the frequency and spatial
        temporal variables
Outputs:
    video: temporal spatial signal. A list of 4D vectors: [b, x, y,
         t], where x, y, t represents the temporal
    sample, and b represents the brightness (singal) on (x, y, t)
'''
def generate_moving_sinusoidals(self, fx, fy, ft, vx, vy, intv):
    (See next page)
```

- The actual function to generate sinusoidal signal.

(Continue)

```python
        # Set video dimension
        pix_per_mm = self.width / self.w_mm # Resolution

        # Build time sequence with the interval (Tb, Te, dt)
        times = []
        dt = self.Te / ft
        t = self.Tb
        while t < self.Te:
            times.append(t)
            t += dt
```

- Build time sequence and set the resolution (*pix_per_mm*).

(Continue)

```
        # Pre-compute video samples based on the interval
        frames = len(times)
        resized_height = (self.height - 1) // intv + 1
        resized_width = (self.width - 1) // intv + 1
        video = np.zeros([resized_height, resized_width, frames])
        for f in range(frames): # Frame
            for y in range(self.height):
                if y % intv == 0: # Downsampling for reducing the
                    computing
                    for x in range(self.width):
                        if x % intv == 0: # Downsampling for reducing
                            the computing
                            (See next page)
```

- Downsampling the signal by *intv*.

(Continue)

```
# Compute input signal value as the brightness according to the specified sin pattern
x_mm = x / pix_per_mm # In mm
y_mm = y / pix_per_mm # In mm
t = times[f]
dx = vx * t
dy = vy * t
b = np.sin(fx * 2 * np.pi * (x_mm + dx) + fy * 2 * np.pi * (y_mm + dy)) # Should located in (-1, 1)
resized_y = y // intv
resized_x = x // intv
video[resized_y, resized_x, f] = b
```

- The brightness is changed by the three factors *fx, fy, dx, dy*.
- *dx* and *dy* are varied with the speed and time.

Report Guidelines

- Describe your implementation. (Do NOT paste code screen shot figures)
- Describe your understanding of theories.
- Describe your experiment settings and results.
  - Examine and analyze variations in results by adjusting the settings of the sinusodal patten.
  - Examine and analyze variations in results by adjusting the response shift of the frequency signal.
  - Share any additional noteworthy insights.
  - Raise any additional problems or sharing any additional insights.
- Describe your proofs.
- Write your division of labor (same rules as those of previous homework).
- Only .pdf file format is allowed.

Grading Criteria

- Richness of your experiments.
- Readability of the report.
- Clarity of your insights and the division of labor.

Scoring

- Code completion (80%)
- Report (20%)

Submission Guidelines

- Put all the codes into a folder *codes*.
- This time, the required videos/images are automatically put into the subfolder *results* in *codes*.
- Put all the experiemented videos (with additional experiment setting) into a folder *supplementary-material*.
- Put **videos**, **supplementary-material** (optional) and report.pdf into a folder **hw4** (i.e., they are under same level of the file tree)
- Compress **hw4** and specify the file format as hw4.zip.
- Upload hw4.zip onto Digital Platform 3.
- Deadline: 12:00 a.m. on March 27th.

Origin of The Latex Template

The latex template is downloaded from:
https://www.LaTeXTemplates.com

The Author: Vel (vel@latextemplates.com)

Temporary page!

LaTeX was unable to guess the total number of pages corre
there was some unprocessed data that should have been
the final page this extra page has been added to receive it
If you rerun the document (without altering it) this surplus
go away, because LaTeX now knows how many pages to exp
this document.