Image Filtering, Warping and Sampling

Jason Lawrence

CS 4810: Graphics

Acknowledgement: slides by Misha Kazhdan, Allison Klein, Tom Funkhouser, Adam Finkelstein and David Dobkin
Outline

• Image Processing

• Image Warping

• Image Sampling
Image Processing

• What about the case when the modification that we would like to make to a pixel depends on the pixels around it?
  o Blurring
  o Edge Detection
  o Etc.
Multi-Pixel Operations

• In the simplest case, we define a mask of weights which tells us how the values at adjacent pixels should be combined to generate the new value.
Blurring

• To blur across pixels, define a mask:
  o Whose value is largest at the center pixel
  o Whose entries sum to one.

\[
\text{Filter} = \begin{bmatrix}
\frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\
\frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\
\frac{1}{16} & \frac{2}{16} & \frac{1}{16}
\end{bmatrix}
\]
Blurring

Pixel(x,y): red = 36  
green = 36  
blue = 0

Filter = \[
\begin{bmatrix}
\frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\
\frac{2}{16} & 4/16 & \frac{2}{16} \\
\frac{1}{16} & \frac{2}{16} & \frac{1}{16}
\end{bmatrix}
\]
Blurring

Pixel(x,y): red = 36
  green = 36
  blue = 0

Pixel(x,y).red and its red neighbors

<table>
<thead>
<tr>
<th>Y-1</th>
<th>X-1</th>
<th>X</th>
<th>X+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>36</td>
<td>109</td>
<td>146</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>36</td>
<td>109</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>36</td>
<td>73</td>
</tr>
</tbody>
</table>

Filter = 
\[
\begin{bmatrix}
\frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\
\frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\
\frac{1}{16} & \frac{2}{16} & \frac{1}{16}
\end{bmatrix}
\]

Original
Blurring

Original

New value for Pixel(x,y).red = 

\[
\begin{align*}
(36 \times \frac{1}{16}) & + (109 \times \frac{2}{16}) + (146 \times \frac{1}{16}) \\
(32 \times \frac{2}{16}) & + (36 \times \frac{4}{16}) + (109 \times \frac{2}{16}) \\
(32 \times \frac{1}{16}) & + (36 \times \frac{2}{16}) + (73 \times \frac{1}{16})
\end{align*}
\]

Pixel(x,y).red and its red neighbors

Filter = 

\[
\begin{bmatrix}
\frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\
\frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\
\frac{1}{16} & \frac{2}{16} & \frac{1}{16}
\end{bmatrix}
\]
Blurring

New value for Pixel(x,y).red = 62.69

Pixel(x,y).red and its red neighbors

Filter =

\[
\begin{bmatrix}
\frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\
\frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\
\frac{1}{16} & \frac{2}{16} & \frac{1}{16}
\end{bmatrix}
\]
Blurring

Original

Blur

New value for Pixel(x,y).red = 63

\[
\begin{bmatrix}
\frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\
\frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\
\frac{1}{16} & \frac{2}{16} & \frac{1}{16}
\end{bmatrix}
\]
Blurring

• Repeat for each pixel and each color channel

• **Note 1**: Keep source and destination separate to avoid “drift”.

• **Note 2**: For boundary pixels, not all neighbors are used, and you need to normalize the mask so that the sum of the values is correct.
Blurring

- In general, the mask can have arbitrary size:
  - We can express a smaller mask as a bigger one by padding with zeros.

\[
\begin{bmatrix}
1 & 2 & 1 \\
2 & 4 & 2 \\
1 & 2 & 1
\end{bmatrix}/16
\]

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 1 & 0 \\
0 & 2 & 4 & 2 & 0 \\
0 & 1 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}/16
\]
In general, the mask can have arbitrary size:

- We can have more non-zero entries to give rise to a wider blur.

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 1 & 0 \\
0 & 2 & 4 & 2 & 0 \\
0 & 1 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 & 1 & 2 & 1 & 0 \\
1 & 2 & 4 & 2 & 1 \\
2 & 4 & 8 & 4 & 2 \\
1 & 2 & 4 & 2 & 1 \\
0 & 1 & 2 & 1 & 0
\end{bmatrix} / 48
\]
Blurring

- A general way for defining the entries of an $n \times n$ mask is to use the values of a Gaussian:

  $$
  \text{Gaussian Mask}[i][j] = e^{-\frac{(x^2 + y^2)}{2\sigma^2}}
  $$

  - $\sigma$ equals the mask radius ($n/2$ for an $n \times n$ mask)
  - $x$ is $i$'s horizontal distance from center pixel
  - $y$ is $j$'s vertical distance from center pixel
  - Don’t forget to normalize!
Edge Detection

• To find the edges in an image, define a mask:
  o Whose value is largest at the center pixel
  o Whose entries sum to zero.

• Edge pixels are those whose value is larger (or smaller) than those of its neighbors.

Original

Detected Edges

Filter = \[
\begin{bmatrix}
-1 & -1 & -1 \\
-1 & 8 & -1 \\
-1 & -1 & -1
\end{bmatrix}
\]
Edge Detection

Pixel\((x,y)\): red = 36
          green = 36
          blue = 0

Filter = \[
\begin{bmatrix}
  -1 & -1 & -1 \\
  -1 &  8 & -1 \\
  -1 & -1 & -1 \\
\end{bmatrix}
\]
## Edge Detection

**Original**

Pixel\((x,y)\): red = 36  
green = 36  
blue = 0

<table>
<thead>
<tr>
<th></th>
<th>X - 1</th>
<th>X</th>
<th>X + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y - 1</td>
<td>36</td>
<td>109</td>
<td>146</td>
</tr>
<tr>
<td>Y</td>
<td>32</td>
<td>36</td>
<td>109</td>
</tr>
<tr>
<td>Y + 1</td>
<td>32</td>
<td>36</td>
<td>73</td>
</tr>
</tbody>
</table>

Pixel\((x,y)\).red and its red neighbors

Filter = \[
\begin{bmatrix}
-1 & -1 & -1 \\
-1 & 8 & -1 \\
-1 & -1 & -1 \\
\end{bmatrix}
\]
Edge Detection

New value for \(\text{Pixel}(x,y)\).\text{red} =
\[
\begin{align*}
36 \times -1 &+ 109 \times -1 + 146 \times -1 \\
32 \times -1 &+ 36 \times 8 + 109 \times -1 \\
32 \times -1 &+ 36 \times -1 + 73 \times -1 
\end{align*}
\]

Filter =
\[
\begin{bmatrix}
-1 & -1 & -1 \\
-1 & 8 & -1 \\
-1 & -1 & -1 
\end{bmatrix}
\]
Edge Detection

New value for Pixel(x,y).red = -285

<table>
<thead>
<tr>
<th></th>
<th>X-1</th>
<th>X</th>
<th>X+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y-1</td>
<td>36</td>
<td>109</td>
<td>146</td>
</tr>
<tr>
<td>Y</td>
<td>32</td>
<td>36</td>
<td>109</td>
</tr>
<tr>
<td>Y+1</td>
<td>32</td>
<td>36</td>
<td>73</td>
</tr>
</tbody>
</table>

Pixel(x,y).red and its red neighbors

Filter = \[
\begin{bmatrix}
-1 & -1 & -1 \\
-1 & 8  & -1 \\
-1 & -1 & -1 \\
\end{bmatrix}
\]
Edge Detection

Original

New value for Pixel(x,y).red = 0

<table>
<thead>
<tr>
<th></th>
<th>X-1</th>
<th>X</th>
<th>X+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y-1</td>
<td>36</td>
<td>109</td>
<td>146</td>
</tr>
<tr>
<td>Y</td>
<td>32</td>
<td>36</td>
<td>109</td>
</tr>
<tr>
<td>Y+1</td>
<td>32</td>
<td>36</td>
<td>73</td>
</tr>
</tbody>
</table>

Pixel(x,y).red and its red neighbors

Filter =
\[
\begin{bmatrix}
-1 & -1 & -1 \\
-1 &  8 & -1 \\
-1 & -1 & -1 \\
\end{bmatrix}
\]
Edge Detection

New value for Pixel(x,y).red = 0

\[
\begin{bmatrix}
-1 & -1 & -1 \\
-1 & 8 & -1 \\
-1 & -1 & -1 \\
\end{bmatrix}
\]
Outline

• Image Processing
• Image Warping
• Image Sampling
Image Warping

• Move pixels of image
  - Mapping
  - Resampling

Source image

Warp

Destination image
Overview

• Mapping
  o Forward
  o Reverse

• Resampling
  o Point sampling
  o Triangle filter
  o Gaussian filter
Mapping

- Define transformation
  - Describe the destination \((x,y)\) for every location \((u,v)\) in the source (or vice-versa, if invertible)
Example Mappings

• Scale by factor:
  \[ o_x = \text{factor} \times u \]
  \[ o_y = \text{factor} \times v \]
Example Mappings

- Rotate by $\theta$ degrees:
  
  \[
  x = u \cos \theta - v \sin \theta \\
  y = u \sin \theta + v \cos \theta
  \]
Example Mappings

- Shear in X by \textit{factor}:
  \[
  x = u + \text{factor} \times v
  \\
y = v
  \]

- Shear in Y by \textit{factor}:
  \[
  x = u
  \\
y = v + \text{factor} \times u
  \]
Other Mappings

• Any function of u and v:
  \[ o_x = f_x(u,v) \]
  \[ o_y = f_y(u,v) \]

Fish-eye

“Swirl”

“Rain”
Image Warping Implementation I

- Forward mapping:

for (int u = 0; u < umax; u++)
  for (int v = 0; v < vmax; v++)
    float x = f_x(u, v);
    float y = f_y(u, v);
    dst(x, y) = src(u, v);
Forward Mapping

- Iterate over source image
Forward Mapping – BAD!

- Iterate over source image

Many source pixels can map to the same destination pixel.

Rotate -30
Forward Mapping – BAD!

- Iterate over source image

Many source pixels can map to same destination pixel

Some destination pixels may not be covered
Image Warping Implementation II

- Reverse mapping:

```c
for (int x = 0; x < xmax; x++)
    for (int y = 0; y < ymax; y++)
        float u = f^{-1}_x(x,y);
        float v = f^{-1}_y(x,y);
        dst(x,y) = src(u,v);
```

Source image | Destination image

(u,v) | f | (x,y)
Reverse Mapping – GOOD!

- Iterate over destination image
  - Must resample source
  - May oversample, but much simpler!
Resampling

- Evaluate source image at arbitrary \((u, v)\)

\((u, v)\) does not usually have integer coordinates

Source image \(\rightarrow\) Destination image
Overview

- Mapping
  - Forward
  - Reverse

- Resampling
  - Nearest Point Sampling
  - Bilinear Sampling
  - Gaussian Sampling
Nearest Point Sampling

• Take value at closest pixel:

\[
i\text{int } iu = \text{trunc}(u+0.5);
\]
\[
i\text{int } iv = \text{trunc}(v+0.5);
\]
\[
dst(x,y) = src(iu,iv);
\]
Bilinear Sampling

- Bilinearly interpolate four closest pixels
  
  \[ a = \text{linear interpolation of src}(x_1, y_1) \text{ and src}(x_2, y_1) \]
  
  \[ b = \text{linear interpolation of src}(x_1, y_2) \text{ and src}(x_2, y_2) \]
  
  \[ \text{dst}(x, y) = \text{linear interpolation of "a" and "b"} \]

\[
\begin{align*}
x_1 &= \text{floor}(x) \\
x_2 &= x_1 + 1 \\
y_1 &= \text{floor}(y) \\
y_2 &= y_1 + 1 \\
dx &= x - x_1 \\
dy &= y - y_1 \\
a &= \text{src}(x_1, y_1) \times (1 - dx) + \text{src}(x_2, y_1) \times dx \\
b &= \text{src}(x_1, y_2) \times (1 - dx) + \text{src}(x_2, y_2) \times dx \\
\text{dst}(x, y) &= a \times (1 - dy) + b \times dy
\end{align*}
\]
Bilinear Sampling

• Bilinearly interpolate four closest pixels
  \[ a = \text{linear interpolation of src}(x_1,y_1) \text{ and src}(x_2,y_1) \]
  \[ b = \text{linear interpolation of src}(x_1,y_2) \text{ and src}(x_2,y_2) \]
  \[ \text{dst}(x,y) = \text{linear interpolation of } "a" \text{ and } "b" \]

\[
\begin{align*}
x_1 & = \text{floor}(x) \\
x_2 & = x_1 + 1 \\
y_1 & = \text{floor}(y) \\
y_2 & = y_1 + 1 \\
dx & = x - x_1 \\
dy & = y - y_1 \\
a & = \text{src}(x_1,y_2) \times (1 - dx) + \text{src}(x_2,y_2) \times dx \\
b & = \text{src}(x_1,y_1) \times (1 - dx) + \text{src}(x_2,y_1) \times dx \\
\text{dst}(x,y) & = a \times (1 - dy) + b \times dy
\end{align*}
\]

Make sure to test that the pixels \((x_1,y_1), (x_2,y_2), (x_1,y_2), \text{ and } (x_2,y_1)\) are within the image.
Gaussian Sampling

• Compute weighted sum of pixel neighborhood:
  - The blending weights are the normalized values of a Gaussian function.
Filtering Methods Comparison

• Trade-offs
  - Jagged edges versus blurring
  - Computation speed
Image Warping Implementation

- Reverse mapping:

```c
for (int x = 0; x < xmax; x++)
    for (int y = 0; y < ymax; y++)
        float u = f_x^{-1}(x,y);
        float v = f_y^{-1}(x,y);
        dst(x,y) = resample_src(u,v,w);
```

Source image  Destination image
Image Warping Implementation

• Reverse mapping:

```c
for (int x = 0; x < xmax; x++)
    for (int y = 0; y < ymax; y++)
        float u = f^{-1}_x(x,y);
        float v = f^{-1}_y(x,y);
        dst(x,y) = resample_src(u,v,w);
```

Source image  Destination image
Example: Scale

- Scale (src, dst, s):
  
  ```c
  float w = ?;
  for (int x = 0; x < xmax; x++)
    for (int y = 0; y < ymax; y++)
      float u = x / s;
      float v = y / s;
      dst(x,y) = resample_src(u,v,w);
  ```

```
```
```
```
```
```
```
Example: Scale

- Scale (src, dst, s):

  
  ```
  float \( w = \frac{1.0}{s} \);
  for (int x = 0; x < xmax; x++)
    for (int y = 0; y < ymax; y++)
      float \( u = \frac{x}{s}; \)
      float \( v = \frac{y}{s}; \)
      dst(x,y) = resample_src(u,v,w); 
  ```

\( w = \frac{1.0}{s} \)
Example: Rotate

- Rotate (src, dst, theta):

  \[
  \text{float } w \approx \,; \\
  \text{for } (\text{int } x = 0; \, x < \text{xmax}; \, x++) \\
  \quad \text{for } (\text{int } y = 0; \, y < \text{ymax}; \, y++) \\
  \quad \quad \text{float } u = x \cdot \cos(-\theta) - y \cdot \sin(-\theta); \\
  \quad \quad \text{float } v = x \cdot \sin(-\theta) + y \cdot \cos(-\theta); \\
  \quad \quad \text{dst}(x, y) = \text{resample_src}(u, v, w); \\
  \]
Example: Rotate

- Rotate \((\text{src}, \text{dst}, \theta)\):

  \[
  \text{float } w \equiv \ ?; \\
  \text{for (int } x = 0; \ x < \text{ xmax; } x++) \\
  \hspace{1em} \text{for (int } y = 0; \ y < \text{ ymax; } y++) \\
  \hspace{2em} \text{float } u = x \times \cos(-\theta) - y \times \sin(-\theta); \\
  \hspace{2em} \text{float } v = x \times \sin(-\theta) + y \times \cos(-\theta); \\
  \hspace{2em} \text{dst}(x,y) = \text{resample}_\text{src}(u,v,w); \\
  \]

- Rotate 30

\[
\begin{align*}
  x &= u \cos \theta - v \sin \theta \\
  y &= u \sin \theta + v \cos \theta
\end{align*}
\]
Example: Fun

• Swirl (src, dst, theta):

```c
float w ≅ ?;
for (int x = 0; x < xmax; x++)
  for (int y = 0; y < ymax; y++)
    float u = rot(dist(x,xcenter)*theta);
    float v = rot(dist(y,ycenter)*theta);
    dst(x,y) = resample_src(u,v,w);
```

![Swirl Diagram]
Outline

• Image Processing
• Image Warping
• Image Sampling
Sampling Questions

• How should we sample an image:
  o Nearest Point Sampling?
  o Bilinear Sampling?
  o Gaussian Sampling?
  o Something Else?
Image Representation

What is an image?

An image is a discrete collection of pixels, each representing a sample of a continuous function.
Sampling

Let’s look at a 1D example:

Continuous Function  Discrete Samples
Sampling

At in-between positions, values are undefined. How do we determine the value of a sample?

Discrete Samples
Sampling

At in-between positions, values are undefined.

How do we determine the value of a sample?

We need to reconstruct a continuous function, turning a collection of discrete samples into a 1D function that we can sample at arbitrary locations.

Discrete Samples
Sampling

At in-between positions, values are undefined.

How do we determine the value of a sample?

We need to reconstruct a continuous function, turning a collection of discrete samples into a 1D function that we can sample at arbitrary locations.

How do we define the in-between values?
**Nearest Point Sampling**

The value at a point is the value of the closest discrete sample.
Nearest Point Sampling

The value at a point is the value of the closest discrete sample.

The reconstruction:

- Interpolates the samples
- Is not continuous
Bilinear Sampling

The value at a point is the (bi)linear interpolation of the two surrounding samples.
Bilinear Sampling

The value at a point is the (bi)linear interpolation of the two surrounding samples.

Reconstructed Function

The reconstruction:
✓ Interpolates the samples
✗ Is not smooth

Discrete Samples
Gaussian Sampling

The value at a point is the Gaussian average of the surrounding samples.
Gaussian Sampling

The value at a point is the Gaussian average of the surrounding samples.

The reconstruction:
- Does not interpolate
- Is smooth
Image Sampling

• How do we reconstruct a function from a collection of samples?
Image Sampling

• How do we reconstruct a function from a collection of samples?

• To answer this question, we need to understand what kind of information the samples contain.
Image Sampling

• How do we reconstruct a function from a collection of samples?

• To answer this question, we need to understand what kind of information the samples contain.

• Signal processing helps us understand this better.
Fourier Analysis

* Fourier analysis provides a way for expressing (or approximating) any signal as a sum of scaled and shifted cosine functions.

The Building Blocks for the Fourier Decomposition
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

Initial Function

\[ f(\theta) \]

0\textsuperscript{th} Order Approximation

\[ f_0(\theta) = a_0 \cos(\theta + \phi_0) \]

0\textsuperscript{th} Order Component
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

\[ f_1(\theta) = a_1 \cos(\theta + \phi_1) \]

\[ f(\theta) \]
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

Initial Function

2\textsuperscript{nd} Order Approximation

\begin{align*}
\text{Initial Function} & \quad \text{2\textsuperscript{nd} Order Approximation} \\
\begin{array}{c}
f(\theta) \\
\end{array} & \quad \begin{array}{c}
\text{1\textsuperscript{st} Order Approximation} \\
\quad + \\
\text{2\textsuperscript{nd} Order Component} \\
\end{array}
\end{align*}

\begin{align*}
f_2(\theta) &= a_2 \cos(2(\theta + \phi_2)) \\
\end{align*}
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

Initial Function

3rd Order Approximation

2nd Order Approximation

\[ f_3(\theta) = a_3 \cos(3(\theta + \phi_3)) \]

3rd Order Component
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

\[ f_4(\theta) = a_4 \cos(4(\theta + \phi_4)) \]

Initial Function

4\(^{th}\) Order Approximation

3\(^{rd}\) Order Approximation

4\(^{th}\) Order Component
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

\[ f(\theta) = a_5(\cos 5(\theta + \phi_5)) \]

4th Order Approximation

5th Order Approximation
Fourier Analysis

• As higher frequency components are added to the approximation, finer details are captured.

\[ f(\theta) = a_6 \cos(6(\theta + \phi_6)) \]

\[ 6^{th} \text{ Order Component} \]
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

Initial Function

7th Order Approximation

\[ f(\theta) = a_7 \cos(7(\theta + \phi_7)) \]

6th Order Approximation

7th Order Component
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

\[ f(\theta) = a_8(\cos(\theta + \phi_8)) \]

\[ f_8(\theta) = a_8(\cos(8(\theta + \phi_8)) \]

Initial Function | 8th Order Approximation
---|---
\[ f(\theta) \]

7th Order Approximation

\[ f_8(\theta) = a_8(\cos(8(\theta + \phi_8)) \]
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

\[ f_9(\theta) = a_9 \cos(9(\theta + \phi_9)) \]

9th Order Component
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

\[ f_{10}(\theta) = a_{10} \cos(10(\theta + \phi_{10})) \]

10th Order Component
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

Initial Function

- $f(\theta)$

$10^{th}$ Order Approximation

- $f_{10}(\theta)$

$11^{th}$ Order Approximation

- $f_{11}(\theta) = a_{11}(\cos 11\theta + \phi_{11})$

$11^{th}$ Order Component
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

Initial Function

12th Order Approximation

11th Order Approximation

$12_{th}$ Order Component

$f_{12}(\theta) = a_{12} \cos(12(\theta + \phi_{12}))$
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

\[ f_{13}(\theta) = a_{13} \cos 13(\theta + \phi_{13}) \]

13th Order Component
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

\[ f_{14}(\theta) = a_{14} \cos(14(\theta + \phi_{14})) \]

14th Order Component
Fourier Analysis

- As higher frequency components are added to the approximation, finer details are captured.

Initial Function

15th Order Approximation

$14^{th}$ Order Approximation

$15^{th}$ Order Component

$f_{15}(\theta) = a_{15}(\cos 15(\theta + \phi_{15}))$
Fourier Analysis

• As higher frequency components are added to the approximation, finer details are captured.

\[ f_{16}(\theta) = a_{16}(\cos 16(\theta + \phi_{16})) \]

16\textsuperscript{th} Order Component
Fourier Analysis

- Combining all of the frequency components together, we get the initial function.

\[ f(\theta) = \sum_{k=0}^{\infty} f_k(\theta) = \sum_{k=0}^{\infty} a_k \cos(k(\theta + \phi_k)) \]

- \( a_k \): amplitude of the \( k \)th frequency component
- \( \phi_k \): shift of the \( k \)th frequency component
**Question**

- As higher frequency components are added to the approximation, finer details are captured.
- If we have $n$ samples, what is the highest frequency that can be represented?
Question

• As higher frequency components are added to the approximation, finer details are captured.

• If we have $n$ samples, what is the highest frequency that can be represented?

Each frequency component has two degrees of freedom:
  • Amplitude
  • Shift

With $n$ samples we can represent the first $n/2$ frequency components
Sampling Theorem

• A signal can be reconstructed from its samples, if the original signal has no frequencies above 1/2 the sampling frequency – Shannon’s Theorem

• The minimum sampling rate for band-limited function is called the “Nyquist rate”

A signal is band-limited if its highest non-zero frequency is bounded. The frequency is called the bandwidth.
Question

• What if we have only $n$ samples and we try to reconstruct a function with frequencies larger than the Nyquist frequency ($n/2$)?
Aliasing

- When a high-frequency signal is sampled with insufficiently many samples, it will be perceived as a lower-frequency signal. This masking of higher frequencies as lower ones is referred to as aliasing.
Aliasing

- When a high-frequency signal is sampled with insufficiently many samples, it will be perceived as a lower-frequency signal. This masking of higher frequencies as lower ones is referred to as **aliasing**.
Aliasing

- When a high-frequency signal is sampled with insufficiently many samples, it will be perceived as a lower-frequency signal. This masking of higher frequencies as lower ones is referred to as **aliasing**.
Aliasing

- When a high-frequency signal is sampled with insufficiently many samples, it will be perceived as a lower-frequency signal. This masking of higher frequencies as lower ones is referred to as aliasing.
Temporal Aliasing

- Artifacts due to limited temporal resolution
Sampling

• There are two problems:
  o You don’t have enough samples to correctly reconstruct your high-frequency information
  o You corrupt the low-frequency information because the high-frequencies mask themselves as lower ones.
Anti-Aliasing

Two possible ways to address aliasing:

• Sample at higher rate

• Pre-filter to form band-limited signal
Anti-Aliasing

Two possible ways to address aliasing:

• Sample at higher rate
  - Not always possible
  - Still rendering to fixed resolution

• Pre-filter to form band-limited signal
Anti-Aliasing

Two possible ways to address aliasing:

• Sample at higher rate
• Pre-filter to form a band-limited signal
  • You still don’t get your high frequencies, but at least the low frequencies are uncorrupted.
Fourier Analysis

• If we just look at how much information each frequency contributes, we obtain the power spectrum of the signal:
Fourier Analysis

- If we just look at how much information each frequency contributes, we obtain the power spectrum of the signal:

Initial Function

= Power Spectrum
Pre-Filtering

• Band-limit by discarding the high-frequency components of the Frequency decomposition.

Initial Power Spectrum

Band-Limited Power Spectrum
Pre-Filtering

• Band-limit by discarding the high-frequency components of the Fourier decomposition.

• We can do this by multiplying the frequency components by a 0/1 function:
Pre-Filtering

- Band-limit by discarding the high-frequency components of the Fourier decomposition.

- We can do this by multiplying the frequency components by a \(0/1\) function:

\[
\begin{align*}
\text{Initial Power Spectrum} & \quad \Rightarrow \\
\text{Band-Limited Spectrum} & \quad \Downarrow \\
\text{Frequency Filter} & \quad \begin{cases}
\sum_{k=0}^{n/2} a_k \cos(k (\theta + \phi_k)) \\
\sum_{k=0}^{\infty} a_k \cos(k (\theta + \phi_k))
\end{cases}
\end{align*}
\]
Fourier Theory

- A fundamental fact from Fourier theory is that multiplication in the frequency domain is equivalent to convolution in the spatial domain.
Convolution

- To convolve two functions $f$ and $g$, we resample the function $f$ using the weights given by $g$. 
Convolution

- To convolve two functions $f$ and $g$, we resample the function $f$ using the weights given by $g$. 

\[ (f * g)(\theta) \]
Convolution

- To convolve two functions $f$ and $g$, we resample the function $f$ using the weights given by $g$. 

\[ (f * g)(\theta) \]
Convolution

- To convolve two functions $f$ and $g$, we resample the function $f$ using the weights given by $g$. 

\[ f(\theta) \quad g(\theta) \quad (f \ast g)(\theta) \]
Convolution

- To convolve two functions $f$ and $g$, we resample the function $f$ using the weights given by $g$. 

$$f(\theta) \rightarrow (f * g)(\theta)$$
Convolution

• To convolve two functions $f$ and $g$, we resample the function $f$ using the weights given by $g$. 

\[
(f * g)(\theta) = \int f(\theta - \tau)g(\tau)\,d\tau
\]
Convolution

- To convolve two functions $f$ and $g$, we resample the function $f$ using the weights given by $g$. 
Convolution

• To convolve two functions $f$ and $g$, we resample the function $f$ using the weights given by $g$. 

\[ (f* g)(\theta) \]

\[ g(\theta) \]

\[ f(\theta) \]
Convolution

- To convolve two functions $f$ and $g$, we resample the function $f$ using the weights given by $g$. 

\[
(f * g)(\theta)
\]
Convolution

- To convolve two functions $f$ and $g$, we resample the function $f$ using the weights given by $g$. 

\[(f * g)(\theta)\]
Convolution

- To convolve two functions $f$ and $g$, we resample the function $f$ using the weights given by $g$. 
Convolution

• To convolve two functions $f$ and $g$, we resample the function $f$ using the weights given by $g$.

• Nearest point, bilinear, and Gaussian interpolation are just convolutions with different filters.
Convolution

• Recall that convolution in the spatial domain is the equal to multiplication in the frequency domain.

• In order to avoid aliasing, we need to convolve with a filter whose power spectrum has value:
  - 0 1 at low frequencies
  - 0 0 at high frequencies

![Diagram showing convolution process](image-url)
Nearest Point Convolution

Discrete Samples  *  Reconstruction Filter  =  Reconstructed Function

Filter Spectrum
Bilinear Convolution

Discrete Samples $\ast$ Reconstruction Filter = Reconstructed Function

Filter Spectrum
Gaussian Convolution

Discrete Samples \(*\) Reconstruction Filter = Reconstructed Function

Filter Spectrum
Convolution

• The ideal filter for avoiding aliasing has a power spectrum with values:
  - 1 at low frequencies
  - 0 at high frequencies

• The sinc function has such a power spectrum and is referred to as the ideal reconstruction filter:

\[
sinc(\theta) = \begin{cases} 
\frac{\sin(\theta)}{\theta} & \text{if } \theta \neq 0 \\
1 & \text{if } \theta = 0
\end{cases}
\]
The Sinc Filter

- The ideal filter for avoiding aliasing has a power spectrum with values:
  - 1 at low frequencies
  - 0 at high frequencies

- The sinc function has such a power spectrum and is referred to as the ideal reconstruction filter:

  ![Reconstruction Filter](image1)
  ![Filter Spectrum](image2)
The Sinc Filter

• Limitations:
  - Has negative values, giving rise to negative weights in the interpolation.
  - The discontinuity in the frequency domain (power spectrum) results in ringing artifacts known as the Gibbs Phenomenon.
The Sinc Filter

- Limitations:
  - Has negative values, giving rise to negative weights in the interpolation.
  - The discontinuity in the frequency domain (power spectrum) results in ringing artifacts near spatial discontinuities, known as the Gibbs Phenomenon.
Summary

There are different ways to sample an image:
- Nearest Point Sampling
- Linear Sampling
- Gaussian Sampling
- Sinc Sampling

These methods have advantages and disadvantages.
Summary – Nearest

✓ Can be implemented efficiently because the filter is non-zero in a very small region.

? Interpolates the samples.

× Is discontinuous.

× Does not address the aliasing problem, giving bad results when a signal is under-sampled.

Discrete Samples \* Reconstruction Filter = Reconstructed Function
Summary – Linear

✓ Can be implemented efficiently because the filter is non-zero in a very small region.

? Interpolates the samples.

✗ Is not smooth.

✗ Partially addresses the aliasing problem, but can still give bad results when a signal is under-sampled.

Discrete Samples

* Reconstructed Function
Summary – Gaussian

✗ Is slow to implement because the filter is non-zero in a large region.

? Does not interpolate the samples.

✓ Is smooth.
✓ Addresses the aliasing problem by killing off the high frequencies.

![Diagram of Discrete Samples, Reconstruction Filter, and Reconstructed Function]
Summary – Sinc

✖ Is slow to implement because the filter is non-zero in a large region.
问他 Does not interpolate the samples.
✖ Assigns negative weights.
✖ Ringing at discontinuities.
✔ Addresses the aliasing problem by killing off the high frequencies.

![Diagram of Discrete Samples, Reconstruction Filter, and Reconstructed Function]
Summary

It appears that we have been mixing the sampling problem with the reconstruction problem.

However, our motivation for the choice of filter is the same in both cases. We want a filter whose spectrum goes to zero so that:

- **Sampling**: High frequency samples are killed off, the signal becomes band-limited, and we can sample discretely.
- **Reconstruction**: We do not end up reconstructing a function with high frequency components.
Image Sampling

Given a signal sampled at $m$ positions, if we would like to re-sample at $n$ positions we need to:

1. Reconstruct a function with maximum non-zero frequency no larger than $\min(m/2, n/2)$.
2. Sample the reconstructed function at the $n$ positions.
Image Sampling

Example:

Sampled $m=25$
Image Sampling

Example:

Sampled $m=25$

Reconstruction

Sampled $m=25$

Reconstruction
Image Sampling

Example:

Sampled $m=25$

Reconstruction

Sampled $n=25$

Sampled $n=10$

Reconstruction
Gaussian Sampling

Recall:

To avoid aliasing, we kill off high-frequency components, by convolving with a function whose power spectrum is zero at high frequencies.

We use a Gaussian for function reconstruction and sampling because it smoothly kills off the high frequency components.
Gaussian Sampling

Q: What variance Gaussian should we use?

A: The variance of the Gaussian should be between 0.5 and 1.0 times the distance between samples.
**Gaussian Sampling**

Q: What variance Gaussian should we use?

A: The variance of the Gaussian should be between 0.5 and 1.0 times the distance between samples.

Gaussians used for reconstructing and sampling a function with 20 samples.
Gaussian Sampling

Q: What variance Gaussian should we use?

A: The variance of the Gaussian should be between 0.5 and 1.0 times the distance between samples.
Gaussian Sampling

Scaling Example:

Q: Suppose we have data represented by 20 samples that we would like to down-sample to 5 samples. What variance should we use?
Gaussian Sampling

Scaling Example:

Q: Suppose we have data represented by 20 samples that we would like to down-sample to 5 samples. What variance should we use?

A: The distance between two adjacent samples in the final array corresponds to a distance of 4 units in the initial array. The variance of the Gaussian should be between 2.0 and 4.0.
Gaussian Sampling

Scaling Example:

Q: Suppose we have data represented by 20 samples that we would like to up-sample to 40 samples. What variance should we use?
Gaussian Sampling

Scaling Example:

Q: Suppose we have data represented by 20 samples that we would like to up-sample to 40 samples. What variance should we use?

A: Because the initial samples can’t represent frequencies higher than 10, we shouldn’t use a Gaussian with smaller variance since this would introduce high-frequency components into the reconstruction. The variance of the Gaussian should remain between 0.5 and 1.0.