• Overview

• Noise Models
  – Gaussian
  – Rayleigh
  – Gamma
  – Exponential
  – Uniform
  – Impulse

• Noise Removal
  – Arithmetic mean
  – Geometric mean
  – Harmonic mean
  – Contraharmonic mean
  – Median filter
  – Max, Min filters
  – Midpoint filter
  – Alpha trimmed mean

• Conclusion
Overview

• The objective of image restoration is to recover an image which has been degraded by some known process
  – f(x,y) → degrade → g(x,y) → restore → f'(x,y)

• By using a mathematical model of degradation we can derive a restoration algorithm that produces an optimal result
  – minimize the difference between f(x,y) and f'(x,y)
Overview

• The key is finding an appropriate model of the image degradation that can be inverted
  – Additive noise
    • $g(x,y) = f(x,y) + n(x,y)$
  – Linear blurring
    • $g(x,y) = f(x,y) \ast h(x,y)$
  – Linear blurring and additive noise
    • $g(x,y) = f(x,y) \ast h(x,y) + n(x,y)$
  – Linear blurring and complex noise
    • $g(x,y) = [f(x,y) \ast h(x,y)][1 + m(x,y)] + n(x,y)$
Noise Models

• Noise can be added to images during image acquisition (faulty CCD elements), during image transmission (channel interference) or during image processing (compression)

• We normally assume that noise is additive and not correlated with the image

• The range and distribution of noise values can vary significantly from one image to the next
Noise Models

- **Gaussian noise:**

\[
p(z) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(z-\bar{z})^2}{2\sigma^2}}
\]
• Rayleigh noise:

\[
p(z) = \begin{cases} 
\frac{2}{b} (z - a) e^{-(z-a)^2 / b} & \text{for } z \geq a \\
0 & \text{for } z < a
\end{cases}
\]
Noise Models

- Gamma noise:

\[ p(z) = \begin{cases} 
  \frac{a^b z^{b-1}}{(b-1)!} e^{-az} & \text{for } z \geq 0 \\
  0 & \text{for } z < 0 
\end{cases} \]
• Exponential noise:

\[ p(z) = \begin{cases} 
  ae^{-az} & \text{for } z \geq 0 \\
  0 & \text{for } z < 0 
\end{cases} \]
Noise Models

- Uniform noise:

\[ p(z) = \begin{cases} 
\frac{1}{b-a} & \text{for } a \leq z \leq b \\
0 & \text{otherwise}
\end{cases} \]
• Impulse noise (salt and pepper):

\[
p(z) = \begin{cases} 
  P_a & \text{for } z = a \\
  P_b & \text{for } z = b \\
  0 & \text{otherwise}
\end{cases}
\]
Noise Models
Noise Models
Noise Models

```c
void IM_TYPE::NoiseUniform(int range)
{
    // Error checking
    if (range < 0) range = 0;

    // Calculate current pixel range
    PIXEL Min, Max;
    MinMax(Min, Max);

    // Process all pixels
    const int RANGE = 1000000;
    for (int index = 0; index < NumPixels; index++)
    {
        float noise =
            rand() % RANGE / (RANGE - 1.0);
        float result = Data1D[index] +
            noise * range – range / 2;
        if (result < Min)
            Data1D[index] = Min;
        else if (result > Max)
            Data1D[index] = Max;
        else
            Data1D[index] = (PIXEL) result;
    }
}
```
Noise Models

```cpp
void IM_TYPE::NoiseGaussian(float stddev)
{
    // Error checking
    if (stddev < 0) stddev = 0;

    // Calculate current pixel range
    PIXEL Min, Max;
    MinMax(Min, Max);

    // Process all pixels
    const int COUNT = 5;
    const int RANGE = 1000000;
    const float SCALE = 0.129;
    for (int index = 0; index < NumPixels; index++)
    {
        // Get gaussian random value between [0..1]
        float noise = 0;
        for (int count = 0; count < COUNT; count++)
            noise += rand() % RANGE / (RANGE - 1.0);
        noise /= COUNT;

        // Add noise to image
        float result = Data1D[index] * (noise-0.5) * stddev / SCALE;
        Data1D[index] = (PIXEL) result;
    }
}
```
void IM_TYPE::NoiseImpulse(float fraction) 
{
    // Error checking
    if (fraction < 0) fraction = 0;
    if (fraction > 1) fraction = 1;

    // Calculate current pixel range
    PIXEL Min, Max;
    MinMax(Min, Max);

    PIXEL Salt = (PIXEL)(0.9 * (Max - Min + 1));
    PIXEL Pepper = (PIXEL)(0.1 * (Max - Min + 1));

    // Process all pixels
    for (int index = 0; index < NumPixels; index++)
    {
        int value = rand() % NumPixels;
        if (value < fraction * NumPixels)
        {
            if (type == 's')
                Data1D[index] = Salt;
            else if (type == 'p')
                Data1D[index] = Pepper;
            else if (type == 'b')
                Data1D[index] = (value%2==0) ? Salt : Pepper;
        }
    }
}
Noise Models

• How can we determine which noise model is appropriate for our situation?
  – Capture image of uniform object
  – Calculate intensity histogram
  – Compare to noise models to select the most similar looking
  – Perform least mean square fit to obtain specific model parameters
Noise Removal

• What is the best way to remove noise?
  – Get more data!
  – Capture N images of the same scene
    \[ g_i(x,y) = f(x,y) + n_i(x,y) \]
  – Average to obtain new image
    \[ g_{\text{ave}}(x,y) = f(x,y) + n_{\text{ave}}(x,y) \]
  – The mean value \( n_{\text{ave}} \) is the same as \( n_i \)
  – The variance of \( n_{\text{ave}} \) is \( 1/N^{1/2} \) the variance of \( n_i \)
  – This can make a huge difference!
Noise Removal

- What is the second best way to remove noise?
  - Use all information available!
  - Derive noise model for the input image
  - Generate a synthetic image with the same noise distribution as the input image
  - Perform experiments on the synthetic image to select the noise removal method and parameters that minimize restoration error
  - Apply this method to the input image
Noise Removal

• Mean filters:
  – Arithmetic mean
    \[ f'(x, y) = \frac{1}{mn} \sum_{(x, y) \in S_{x, y}} g(x, y) \]
  – Geometric mean
    \[ f'(x, y) = \left[ \prod_{(x, y) \in S_{x, y}} g(x, y) \right]^{\frac{1}{mn}} \]
Noise Removal

- Mean filters:
  - Harmonic mean
    \[ f'(x, y) = \frac{mn}{\sum_{(x, y) \in S_{x,y}} g(x, y)^{-1}} \]
  - Contraharmonic mean
    \[ f'(x, y) = \frac{\sum_{(x, y) \in S_{x,y}} g(x, y)^{Q+1}}{\sum_{(x, y) \in S_{x,y}} g(x, y)^{Q}} \]
    
    \[ Q = 0 \rightarrow \text{arithmetic mean} \]
    
    \[ Q = -1 \rightarrow \text{harmonic mean} \]
Noise Removal

Original image

Gaussian noise

3x3 arithmetic mean

3x3 geometric mean
Noise Removal

3x3 contra-harmonic filter with $Q = 1.5$

Pepper noise with $P = 0.1$

Salt noise with $P = 0.1$

3x3 contra-harmonic filter with $Q = -1.5$
Noise Removal

• Order statistic filters:
  – Median filter

\[ f'(x,y) = \text{median}\{ g(x,y) \} \]

\( (x,y) \in S_{x,y} \)

  – Max and min filters

\[ f'(x,y) = \max \{ g(x,y) \} \]

\( (x,y) \in S_{x,y} \)

\[ f'(x,y) = \min \{ g(x,y) \} \]

\( (x,y) \in S_{x,y} \)
Noise Removal

- Order statistic filters:
  - Midpoint filter
    \[ f'(x,y) = \frac{1}{2} \left[ \max_{(x,y) \in S_{x,y}} \{g(x,y)\} + \min_{(x,y) \in S_{x,y}} \{g(x,y)\} \right] \]
  - Alpha trimmed mean filter
    \[ f'(x,y) = \frac{1}{mn - d} \sum_{(s,y) \in S_{x,y}} g_r(s,t) \]
  where \( g_r(s,t) \) represents the middle \((mn-d)\) points in the sorted neighborhood \( S_{x,y} \)
Noise Removal

Impulse noise
\[ P_a = P_b = 0.1 \]

1 iteration
3x3 median

2 iterations
3x3 median

3 iterations
3x3 median
Noise Removal

Pepper noise with $P = 0.1$

Salt noise with $P = 0.1$

3x3 max filter

3x3 min filter
Noise Removal

impulse noise with $P = 0.02$

2x2 median

3x3 median

4x4 median
Noise Removal

impulse noise with $P = 0.02$

max filter radius 1

max filter radius 2

max filter radius 4
Noise Removal

impulse noise with $P = 0.02$

min filter radius 1

min filter radius 2

min filter radius 4
Noise Removal

impulse noise with $P = 0.02$

mid filter radius 1

mid filter radius 2

mid filter radius 4
Noise Removal

impulse noise with $P = 0.05$

3x3 alpha trim mean with $a=0$

3x3 alpha trim mean with $a=4$

3x3 alpha trim mean with $a=8$
void IM_TYPE::Dilate(float radius)
{
    // Copy input image
    IM_TYPE in(*this);

    // Process all pixels
    int search = (int)(radius + 0.5);
    for (int y = 0; y < Ydim; y++)
        for (int x = 0; x < Xdim; x++)
        {
            // Find local max
            PIXEL max = in.Data2D[y][x];

            for (int dy = -search; dy <= search; dy++)
                for (int dx = -search; dx <= search; dx++)
                {
                    int dist = dx*dx + dy*dy;
                    if ((dist <= radius*radius) &&
                        (y+dy >= 0) && (y+dy < Ydim) &&
                        (x+dx >= 0) && (x+dx < Xdim) &&
                        (max < in.Data2D[y+dy][x+dx]))
                        max = in.Data2D[y+dy][x+dx];
                }
            Data2D[y][x] = max;
        }
}
void IM_TYPE::Erode(float radius)
{
    // Copy input image
    IM_TYPE in(*this);

    // Process all pixels
    int search = (int)(radius + 0.5);
    for (int y = 0; y < Ydim; y++)
        for (int x = 0; x < Xdim; x++)
        {
            // Find local min
            PIXEL min = in.Data2D[y][x];
            for (int dy = -search; dy <= search; dy++)
                for (int dx = -search; dx <= search; dx++)
                {
                    int dist = dx*dx + dy*dy;
                    if ((dist <= radius*radius) &&
                        (y+dy >= 0) && (y+dy < Ydim) &&
                        (x+dx >= 0) && (x+dx < Xdim) &&
                        (min > in.Data2D[y+dy][x+dx]))
                        min = in.Data2D[y+dy][x+dx];
                }
            Data2D[y][x] = min;
        }
}
void IM_TYPE::Midpoint(float radius)
{
    // Copy input image
    IM_TYPE in(*this);

    // Process all pixels
    int search = (int)(radius + 0.5);
    for (int y = 0; y < Ydim; y++)
    for (int x = 0; x < Xdim; x++)
    {
        // Find local min and max
        PIXEL min = in.Data2D[y][x];
        PIXEL max = in.Data2D[y][x];

        for (int dy = -search; dy <= search; dy++)
        for (int dx = -search; dx <= search; dx++)
        {
            int dist = dx*dx + dy*dy;
            if ((dist <= radius*radius) &&
                (y+dy >= 0) && (y+dy < Ydim) &&
                (x+dx >= 0) && (x+dx < Xdim))
            {
                if (min > in.Data2D[y+dy][x+dx])
                    min = in.Data2D[y+dy][x+dx];
                if (max < in.Data2D[y+dy][x+dx])
                    max = in.Data2D[y+dy][x+dx];
            }
        }
        Data2D[y][x] = (min + max) / 2;
    }
}
void IM_TYPE::AlphaMean
  (int xdim, int ydim, int Alpha)
{
    // Error checking
    if (xdim < 1) xdim = 1;
    if (ydim < 1) ydim = 1;
    if (Alpha < 0) Alpha = 0;

    // Copy input image
    IM_TYPE in(*this);

    // Define processing window
    int xlow = - xdim / 2;
    int xhigh = xlow + xdim;
    int ylow = - ydim / 2;
    int yhigh = ylow + ydim;
    int area = xdim * ydim;
    PIXEL *Data = new PIXEL[area];

    // Perform smoothing
    for (int y = 0; y < Ydim; y++)
      for (int x = 0; x < Xdim; x++)
        {
          // ...
// Copy pixels in window to temp array
int count = 0;
for (int dy = y+ylow; dy < y+yhigh; dy++)
for (int dx = x+xlow; dx < x+xhigh; dx++)
if ((dy >= 0) && (dy < Ydim) &&
    (dx >=0) && (dx < Xdim))
    Data[count++] = in.Data2D[dy][dx];

// Perform sort on temp array
quicksort(Data, 0, count - 1);

// Trim alpha/2 values from left and right
int left = Alpha/2;
int right = count-1-Alpha+Alpha/2;
if (Alpha >= count) left = right = count/2;

// Calculate average of remaining values
float total = 0;
for (int i = left; i<= right; i++)
total += Data[i];
Data2D[y][x] =
    (PIXEL)(total/(right-left+1));
}

// Release memory
delete []Data;
Conclusion

• In this section, we have focused on noise models for image restoration and spatial domain noise removal techniques.
• In the next section, we will consider more complex image degradation models and a variety of frequency domain restoration algorithms.