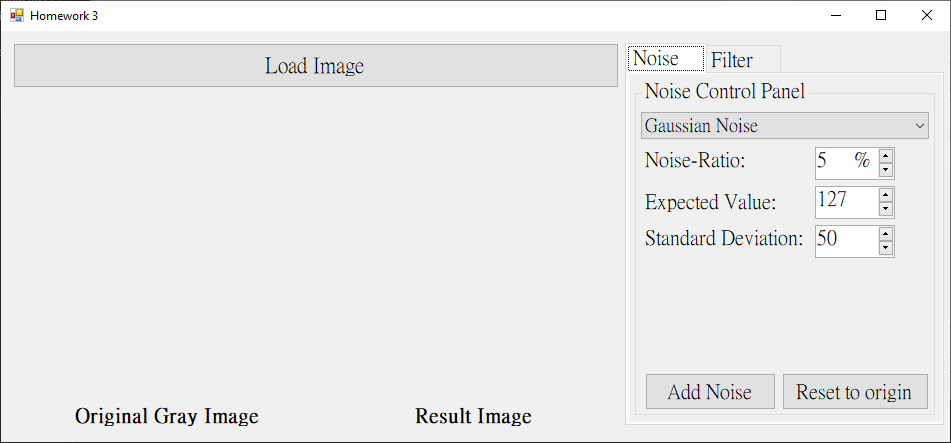
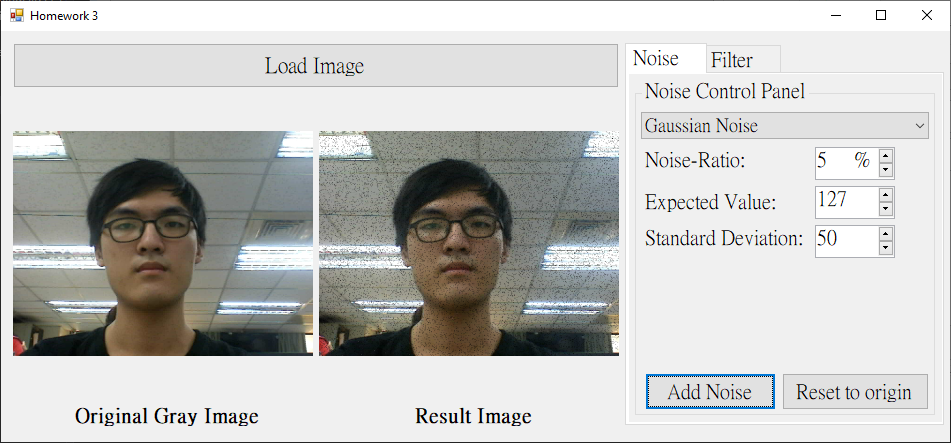
四資工三甲 C108151131 賴冠綸

1. 程式執行畫面：
   1. 程式起始畫面



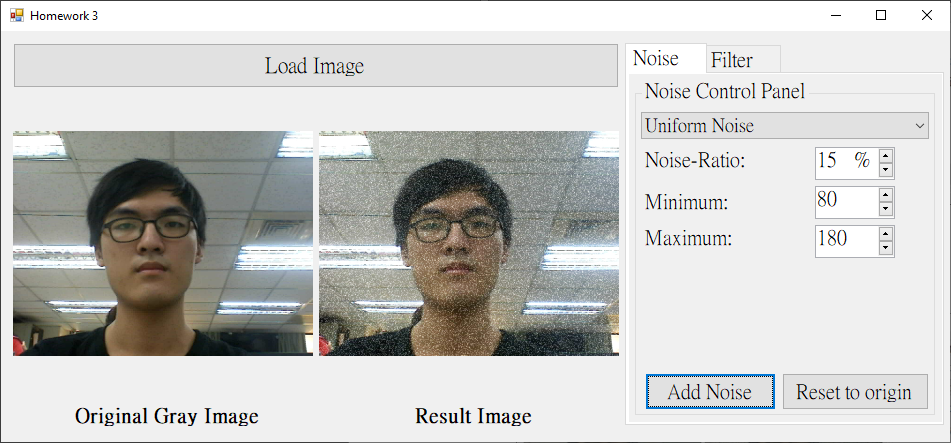
* 1. 增加雜訊（雜訊可相互疊加，**可自訂參數**）
     + 1. Gaussian Noise

5% 雜訊；期望值=127；標準差=50



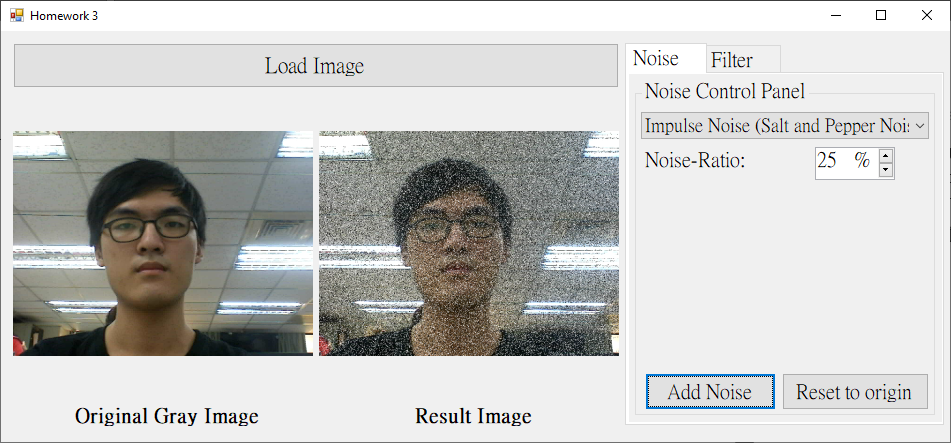
* + - 1. Uniform Noise

15% 雜訊；最小值=80；最大值=180



* + - 1. Impulse Noise (Salt-and-Pepper Noise)

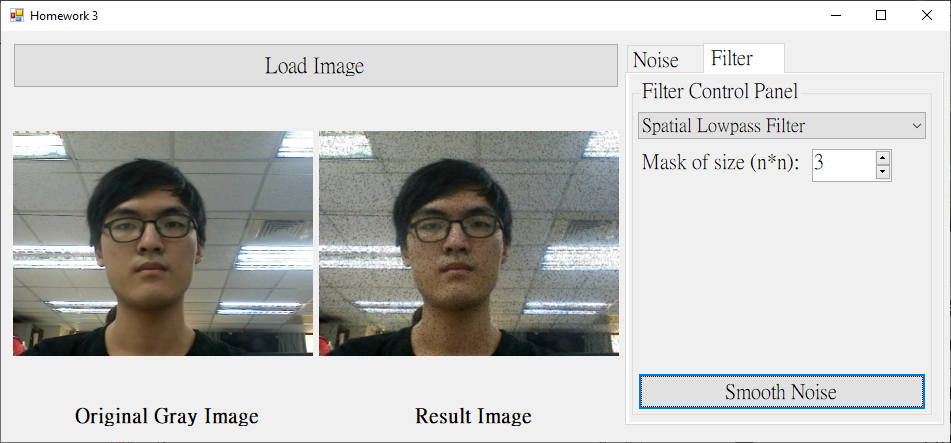
25% 雜訊



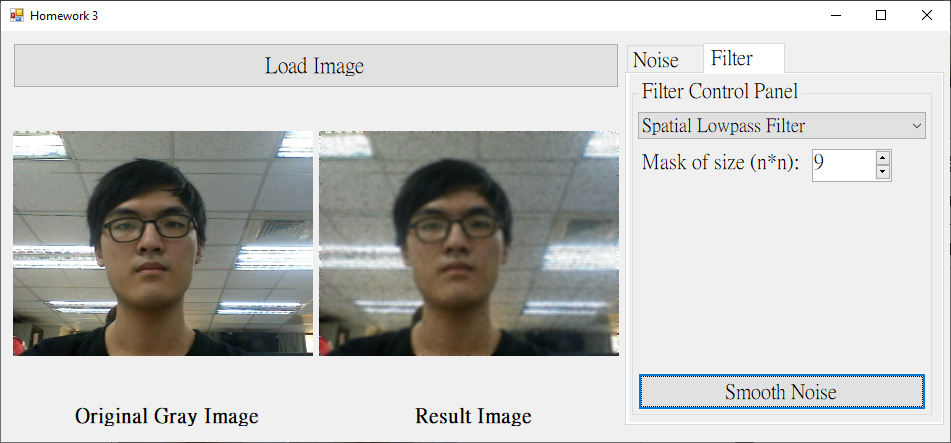
* 1. 濾波器（效果可相互疊加）
     + 1. Spatial Lowpass Filter

丟一個含5%的高斯雜訊影像進去。

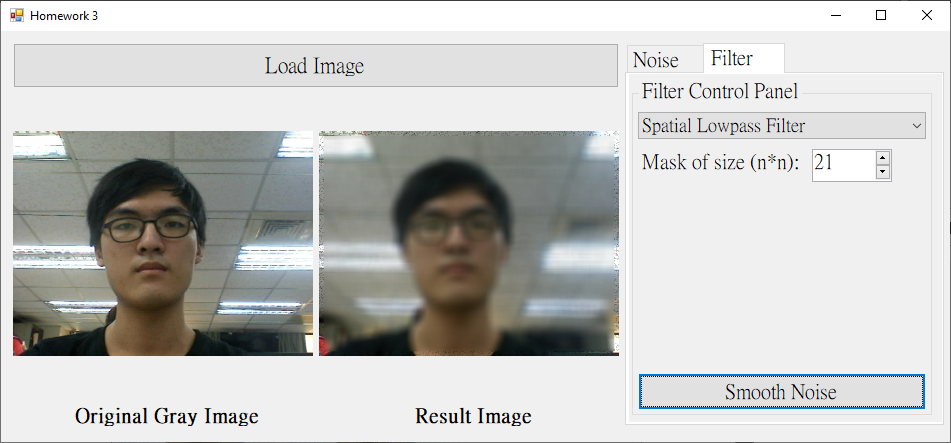
* + - * 1. Mask Size = 3x3



* + - * 1. Mask Size = 9x9

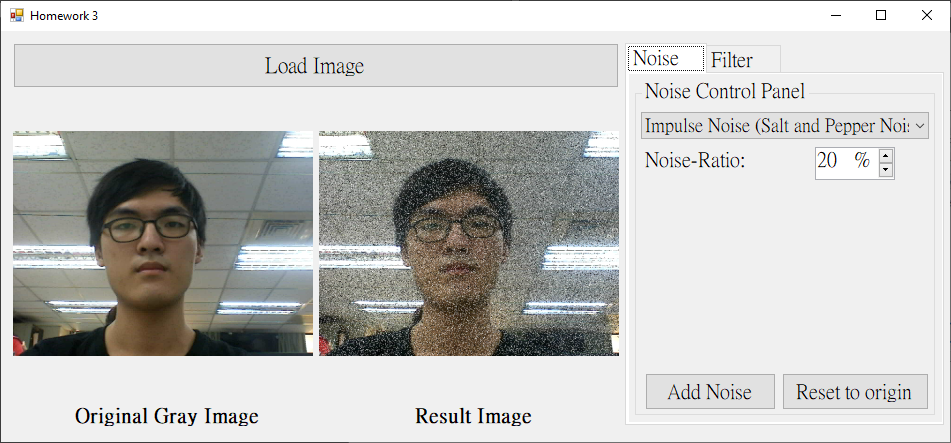


* + - * 1. Mask Size = 21x21

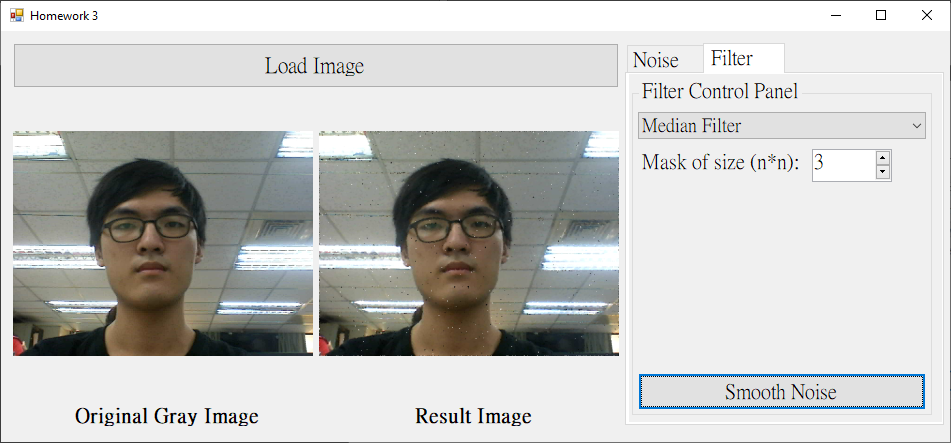


* + - 1. Median Filter

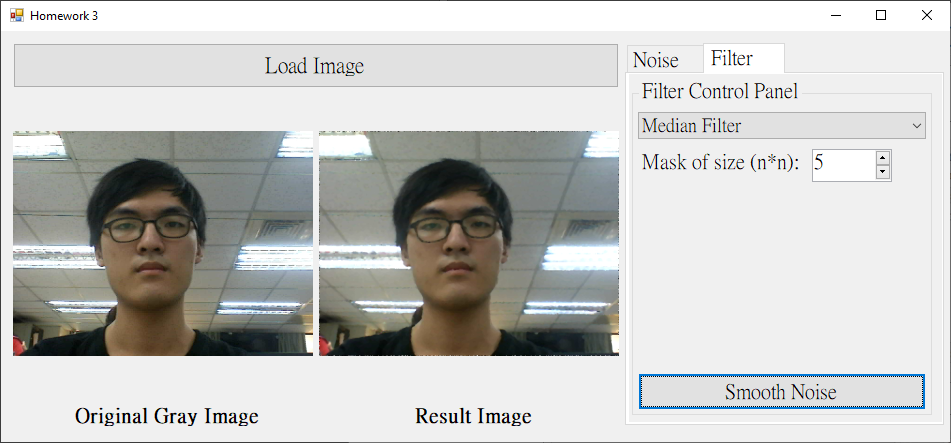
照題目2所說，丟一個含20%的脈衝雜訊影像進去。



* + - * 1. Mask Size = 3x3

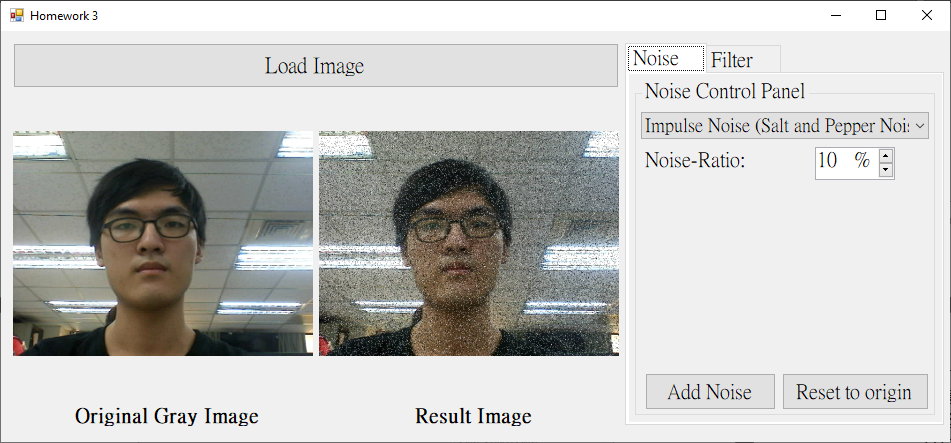


* + - * 1. Mask Size = 5x5

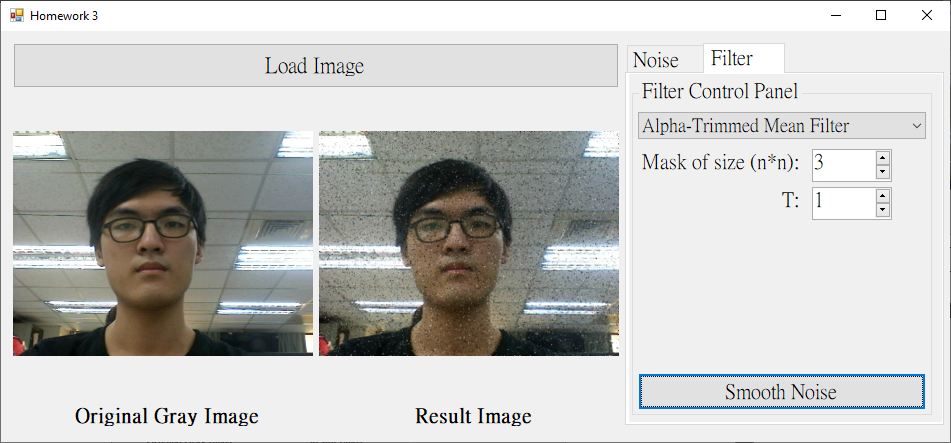


* + - 1. Alpha-Trimmed Mean Filter

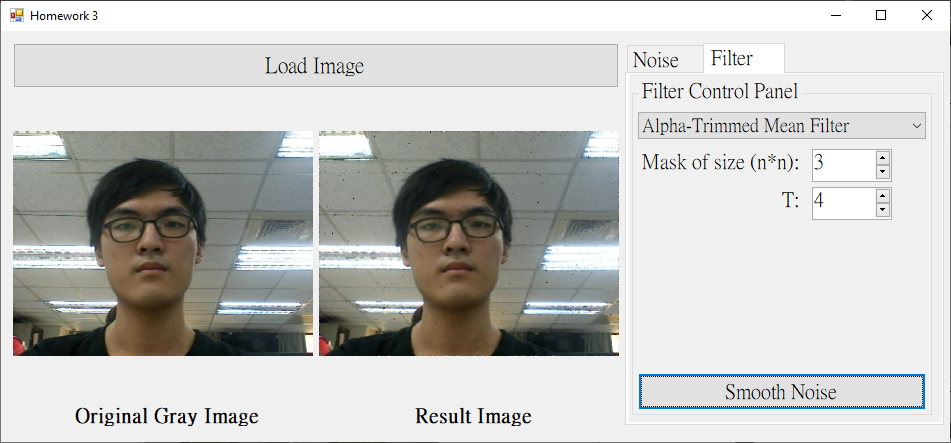
照題目3所說，丟一個含5%之期望值=0且標準差=400的高斯雜訊及10%的脈衝雜訊影像進去。



* + - * 1. T = 1



* + - * 1. T = 4 (=Median Filter)



1. 程式碼（僅附主要程式碼）
   1. 增加雜訊
      * 1. Gaussian Noise

|  |
| --- |
| private: Bitmap^ AddNoiseToImg\_Gaussian(int ratio, double ex\_val, double std\_dev, Bitmap^ origin)  {  Bitmap^ result = CopyOriginIamgeToResult(origin);  BitmapData^ resultBD = result->LockBits(Rectangle(0, 0, result->Width, result->Height)  , ImageLockMode::ReadWrite, result->PixelFormat);  Byte\* resultPtr = (Byte\*)((void\*)resultBD->Scan0);  int noiseAmount = resultBD->Width \* resultBD->Height \* ratio \* 0.01;  array<int>^ noiseArray = gcnew array<int>(noiseAmount); // c++/cli dynamic array  for (int i = 0; i < noiseAmount; i++) noiseArray[i] = 0;  srand(time(NULL));  for (int i = 0; i < noiseAmount; i++)  {  // X = sqrt(-2 \* ln(U)) \* cos(2 \* pi \* V) \* stdDev + expectedValue  double u = rand() / (double)RAND\_MAX;  double v = rand() / (double)RAND\_MAX;  double x = sqrt(-2 \* log(u)) \* cos(2 \* PI \* v) \* std\_dev + ex\_val;  int tmp = (int)round(x); // avoid out of the range  if (tmp > 255) noiseArray[255]++;  else if (tmp < 0) noiseArray[0]++;  else noiseArray[tmp]++;  }  // to check the selected postion whether is noise.  array<array<bool>^>^ isPositionNoised = gcnew array<array<bool>^> (resultBD->Height);  for (int i = 0; i < resultBD->Height; i++)  isPositionNoised[i] = gcnew array<bool>(resultBD->Width);  int curNoiseAmount = 0;  while (curNoiseAmount < noiseAmount)  {  int randX = rand() % resultBD->Width;  int randY = rand() % resultBD->Height;  if (!isPositionNoised[randY][randX])  {  Byte\* ptr = resultPtr + randX \* 3 + randY \* resultBD->Stride;  ptr[0] = noiseArray[curNoiseAmount];  ptr[1] = noiseArray[curNoiseAmount];  ptr[2] = noiseArray[curNoiseAmount];  isPositionNoised[randY][randX] = true;  curNoiseAmount++;  }  }  result->UnlockBits(resultBD);  return result;  } |

* + - 1. Uniform Noise

|  |
| --- |
| private: Bitmap^ AddNoiseToImg\_Uniform(int ratio, int max, int min, Bitmap^ origin)  {  Bitmap^ result = CopyOriginIamgeToResult(origin);  BitmapData^ resultBD = result->LockBits(Rectangle(0, 0, result->Width, result->Height)  , ImageLockMode::ReadWrite, result->PixelFormat);  Byte\* resultPtr = (Byte\*)((void\*)resultBD->Scan0);  double probability = resultBD->Width \* resultBD->Height \* ratio \* 0.01;  // to check the selected postion whether is noise.  array<array<bool>^>^ isPositionNoised = gcnew array<array<bool>^>(resultBD->Height);  for (int i = 0; i < resultBD->Height; i++)  isPositionNoised[i] = gcnew array<bool>(resultBD->Width);  srand(time(NULL));  for (int i = 0; i < probability; i++)  {  int randVal = rand() % (max - min) + min;  int randX = rand() % resultBD->Width;  int randY = rand() % resultBD->Height;    if (!isPositionNoised[randY][randX])  {  Byte\* ptr = resultPtr + randX \* 3 + randY \* resultBD->Stride;  for (int channel = 0; channel < 3; channel++)  {  if (ptr[channel] + randVal > 255) ptr[channel] = 255;  else if (ptr[channel] + randVal < 0) ptr[channel] = 0;  else ptr[channel] += randVal;  }  isPositionNoised[randY][randX] = true;  }  }  result->UnlockBits(resultBD);  return result;  } |

* + - 1. Impulse Noise (Salt-and-Pepper Noise)

|  |
| --- |
| private: Bitmap^ AddNoiseToImg\_SaltPepper(int ratio, Bitmap^ origin)  {  Bitmap^ result = CopyOriginIamgeToResult(origin);  BitmapData^ resultBD = result->LockBits(Rectangle(0, 0, result->Width, result->Height)  , ImageLockMode::ReadWrite, result->PixelFormat);  Byte\* resultPtr = (Byte\*)((void\*)resultBD->Scan0);  int noiseAmount = resultBD->Height \* resultBD->Width \* ratio \* 0.01;  int pepperAmount = noiseAmount / 2; // black pixel amount  int saltAmount = noiseAmount / 2; // white pixel amount  srand(time(NULL));  int pepperCount = 0, saltCount = 0;  while (pepperCount < pepperAmount)  {  int randX = rand() % resultBD->Width;  int randY = rand() % resultBD->Height;  Byte\* ptr = resultPtr + randX \* 3 + randY \* resultBD->Stride;  if (ptr[0] != 0 && ptr[1] != 0 && ptr[2] != 0  && ptr[0] != 255 && ptr[1] != 255 && ptr[2] != 255)  {  ptr[0] = ptr[1] = ptr[2] = 0;  pepperCount++;  }  }  while (saltCount < saltAmount)  {  int randX = rand() % resultBD->Width;  int randY = rand() % resultBD->Height;  Byte\* ptr = resultPtr + randX \* 3 + randY \* resultBD->Stride;  if (ptr[0] != 0 && ptr[1] != 0 && ptr[2] != 0  && ptr[0] != 255 && ptr[1] != 255 && ptr[2] != 255)  {  ptr[0] = ptr[1] = ptr[2] = 255;  saltCount++;  }  }  result->UnlockBits(resultBD);  return result;  } |

* 1. 濾波器
     + 1. Spatial Lowpass Filter

|  |
| --- |
| private: Bitmap^ LowpassFilter(int maskSize, Bitmap^ noiseImg)  {  Bitmap^ result = gcnew Bitmap(noiseImg->Width, noiseImg->Height  , noiseImg->PixelFormat);  BitmapData^ noiseImgBD = noiseImg->LockBits(Rectangle(0, 0, noiseImg->Width  , noiseImg->Height), ImageLockMode::ReadOnly, noiseImg->PixelFormat);  BitmapData^ resultBD = result->LockBits(Rectangle(0, 0, result->Width, result->Height)  , ImageLockMode::ReadWrite, result->PixelFormat);  /\* Image maybe have unknown information \*/  int resultByteskip = resultBD->Stride - resultBD->Width \* 3;  Byte\* noiseImgPtr = (Byte\*)((void\*)noiseImgBD->Scan0);  Byte\* resultPtr = (Byte\*)((void\*)resultBD->Scan0);  array<array<int>^>^ mask = gcnew array<array<int>^> (maskSize);  for (int i = 0; i < maskSize; i++) mask[i] = gcnew array<int> (maskSize);  for (int i = 0; i < maskSize; i++)  for (int j = 0; j < maskSize; j++)  mask[i][j] = 1;  int blank = maskSize / 2;  for (int y = 0; y < noiseImgBD->Height; y++)  {  for (int x = 0; x < noiseImgBD->Width; x++)  {  // avoid out of the range  if (x >= blank && x < noiseImgBD->Width - blank  && y >= blank && y < noiseImgBD->Height - blank)  {  int bSum = 0, gSum = 0, rSum = 0;  for (int i = -blank, index\_i = 0; i <= blank; i++)  {  for (int j = -blank, index\_j = 0; j <= blank; j++)  {  Byte\* ptr = noiseImgPtr + (x + j) \* 3 + (y + i) \* noiseImgBD->Stride;  bSum += ptr[0] \* mask[index\_i][index\_j];  gSum += ptr[1] \* mask[index\_i][index\_j];  rSum += ptr[2] \* mask[index\_i][index\_j];  }  }  resultPtr[0] = bSum / (maskSize \* maskSize);  resultPtr[1] = gSum / (maskSize \* maskSize);  resultPtr[2] = rSum / (maskSize \* maskSize);  }  else // if is edge  {  Byte\* ptr = noiseImgPtr + x \* 3 + y \* noiseImgBD->Stride;  resultPtr[0] = ptr[0];  resultPtr[1] = ptr[1];  resultPtr[2] = ptr[2];  }  resultPtr += 3;  }  resultPtr += resultByteskip;  }  noiseImg->UnlockBits(noiseImgBD);  result->UnlockBits(resultBD);  return result;  } |

* + - 1. Median Filter

|  |
| --- |
| private: Bitmap^ MedianFilter(int maskSize, Bitmap^ noiseImg)  {  Bitmap^ result = gcnew Bitmap(noiseImg->Width, noiseImg->Height  , noiseImg->PixelFormat);  BitmapData^ noiseImgBD = noiseImg->LockBits(Rectangle(0, 0, noiseImg->Width  , noiseImg->Height), ImageLockMode::ReadOnly, noiseImg->PixelFormat);  BitmapData^ resultBD = result->LockBits(Rectangle(0, 0, result->Width, result->Height)  , ImageLockMode::ReadWrite, result->PixelFormat);  /\* Image maybe have unknown information \*/  int resultByteskip = resultBD->Stride - resultBD->Width \* 3;  Byte\* noiseImgPtr = (Byte\*)((void\*)noiseImgBD->Scan0);  Byte\* resultPtr = (Byte\*)((void\*)resultBD->Scan0);  int blank = maskSize / 2;  int median = maskSize \* maskSize / 2;  for (int y = 0; y < noiseImgBD->Height; y++)  {  for (int x = 0; x < noiseImgBD->Width; x++)  {  // avoid out of the range  if (x >= blank && x < noiseImgBD->Width - blank  && y >= blank && y < noiseImgBD->Height - blank)  {  // vector template needs import library "vector"  std::vector<int> b;  std::vector<int> g;  std::vector<int> r;  for (int i = -blank; i <= blank; i++)  {  for (int j = -blank; j <= blank; j++)  {  Byte\* ptr = noiseImgPtr + (x + j) \* 3 + (y + i) \* noiseImgBD->Stride;  b.push\_back(ptr[0]);  g.push\_back(ptr[1]);  r.push\_back(ptr[2]);  }  }  // sort function needs import library "Algorithm"  std::sort(b.begin(), b.end());  std::sort(g.begin(), g.end());  std::sort(r.begin(), r.end());  resultPtr[0] = b[median];  resultPtr[1] = g[median];  resultPtr[2] = r[median];  }  else // if is edge  {  Byte\* ptr = noiseImgPtr + x \* 3 + y \* noiseImgBD->Stride;  resultPtr[0] = ptr[0];  resultPtr[1] = ptr[1];  resultPtr[2] = ptr[2];  }  resultPtr += 3;  }  resultPtr += resultByteskip;  }  noiseImg->UnlockBits(noiseImgBD);  result->UnlockBits(resultBD);  return result;  } |

* + - 1. Alpha-Trimmed Mean Filter

|  |
| --- |
| private: Bitmap^ AlphaTrimmedMeanFilter(int maskSize, int T, Bitmap^ noiseImg)  {  Bitmap^ result = gcnew Bitmap(noiseImg->Width, noiseImg->Height  , noiseImg->PixelFormat);  BitmapData^ noiseImgBD = noiseImg->LockBits(Rectangle(0, 0, noiseImg->Width  , noiseImg->Height), ImageLockMode::ReadOnly, noiseImg->PixelFormat);  BitmapData^ resultBD = result->LockBits(Rectangle(0, 0, result->Width, result->Height)  , ImageLockMode::ReadWrite, result->PixelFormat);  /\* Image maybe have unknown information \*/  int resultByteskip = resultBD->Stride - resultBD->Width \* 3;  Byte\* noiseImgPtr = (Byte\*)((void\*)noiseImgBD->Scan0);  Byte\* resultPtr = (Byte\*)((void\*)resultBD->Scan0);  int blank = maskSize / 2;  for (int y = 0; y < noiseImgBD->Height; y++)  {  for (int x = 0; x < noiseImgBD->Width; x++)  {  // avoid out of the range  if (x >= blank && x < noiseImgBD->Width - blank  && y >= blank && y < noiseImgBD->Height - blank)  {  // vector template needs import library "vector"  std::vector<int> b;  std::vector<int> g;  std::vector<int> r;  for (int i = -blank; i <= blank; i++)  {  for (int j = -blank; j <= blank; j++)  {  Byte\* ptr = noiseImgPtr + (x + j) \* 3 + (y + i) \* noiseImgBD->Stride;  b.push\_back(ptr[0]);  g.push\_back(ptr[1]);  r.push\_back(ptr[2]);  }  }  // sort function needs import library "Algorithm"  std::sort(b.begin(), b.end());  std::sort(g.begin(), g.end());  std::sort(r.begin(), r.end());  int bSum = 0, gSum = 0, rSum = 0;  for (int index = T ; index < maskSize \* maskSize - T; index++)  {  bSum += b[index];  gSum += g[index];  rSum += r[index];  }  resultPtr[0] = bSum / (maskSize \* maskSize - 2 \* T);  resultPtr[1] = gSum / (maskSize \* maskSize - 2 \* T);  resultPtr[2] = rSum / (maskSize \* maskSize - 2 \* T);  }  else // if is edge  {  Byte\* ptr = noiseImgPtr + x \* 3 + y \* noiseImgBD->Stride;  resultPtr[0] = ptr[0];  resultPtr[1] = ptr[1];  resultPtr[2] = ptr[2];  }  resultPtr += 3;  }  resultPtr += resultByteskip;  }  noiseImg->UnlockBits(noiseImgBD);  result->UnlockBits(resultBD);  return result;  } |