CONTENTS

ビジュアルヘルプ – 画像処理（１）

本ドキュメントの操作は、ほとんどの場合、Image Processing Tutorial エクスペリメントを使って行います。

画像処理

画像処理とは、2D、3D、あるいは 4D のウェーブの形で存在する画像データに適用できるほとんどの操作を表す広範な用語です。
画像処理は、たとえデータが画像とはまったく関係がない場合でも、適切な分析ツールを提供することがあります。
ヘルプ Image Plots で、画像の表示に関連する操作について説明しています。
ここでは、画像の処理に使用できる変換、分析操作、および特別なユーティリティツールに焦点を当てます。

このセクションと並行して、Image Processing Tutorial エクスペリメント（File→Example Experiments→Tutorials）を使用できます。
このエクスペリメントには、導入資料に加え、サンプル画像およびこのヘルプファイルに記載されているコマンドの大半が含まれています。
コマンドを実行するには、ヘルプファイル内でコマンドを選択し、Ctrl キーを押しながら Enter キーを押します。

画像の変換

画像の変換の基本的な２つの分類は、色変換とグレースケール／値変換です。
色変換には、ある色空間から別の色空間への色情報の変換、カラー画像からグレースケールへの変換、およびグレースケール画像を偽色で表現することが含まれます。
グレースケール／値変換には、例えば、ピクセルレベルのマッピング、数学的演算、形態学的演算などが含まれます。

色変換

カラー画像には多くの標準的なファイル形式があります。
カラー画像が 2D ウェーブとして保存される場合、それには関連付けられた、あるいは暗黙のカラーマップがあり、各ピクセルの RGB 値は、2D ウェーブの値をカラーマップにマッピングすることで得られます。

画像が 3D ウェーブの場合、各画像平面は個別の赤、緑、青の色成分に対応します。
ウェーブが符号なしバイト型（/B/U）の場合、各平面の値の範囲は [0,255] です。
ウェーブが符号なしバイト型でない場合、値の範囲は [0,65535] です。

3D ウェーブには他に２種類あります。
１つ目は RGBA に対応する４層で構成され、「A」はアルファ（透明度）チャンネルを表します。
２つ目は３以上の平面を含む場合があり、その場合各平面はグレースケール画像で、次のコマンドで表示できます。

```
ModifyImage imageName, plane=n
```

複数のカラー画像を１つの 4D ウェーブに格納でき、各チャンクが個別の RGB 画像に対応します。

異なる種類の画像間の変換を行うためのツールのほとんどは、ImageTransform コマンドで見つけることができます。
例えば、カラーマップを持つ 2D 画像ウェーブを 3D RGB 画像ウェーブに変換できます。
下記では、「Red Rock」という名前の 2D 画像から、「Red RockCMap」という名前のカラーマップウェーブの RGB 値を使って、M_RGBOut という名前の３層 3D ウェーブを作成します：

```
ImageTransform /C='Red RockCMap' cmap2rgb 'Red Rock'
NewImage M_RGBOut        // 結果の 3D ウェーブは M_RGBOut
```

注記： Image Processing Tutorial エクスペリメントの画像はルートデータフォルダーに保存されていないため、チュートリアルのエクスペリメントのコマンドの多くにはデータフォルダーパスが含まれています。本ドキュメントでは、データフォルダーパスを追加しています。
ここで表示されているコマンドを実行したい場合は、「Image Processing Tutorial Help」ウィンドウのコマンドを使ってください。
データフォルダーの詳細については、ヘルプ Data Folders を参照してください。

色情報を除去し画像をグレースケールに変換する必要がある状況も多いです。
これは通常、元のカラー画像をグレースケール処理で処理または分析する場合に発生します。
例えば、先ほど生成した RGB 画像を使った例を示します。

```
ImageTransform rgb2gray M_RGBOut
NewImage M_RGB2Gray     // グレースケール画像を表示
```

グレーへの変換は YIQ 規格に基づいており、グレー出力ウェーブは Y チャンネルに対応します。

```
gray=0.299*red+0.587*green+0.114*blue.
```

別の変換定数セット {ci} を使いたい場合は、コマンドラインで変換を実行できます。

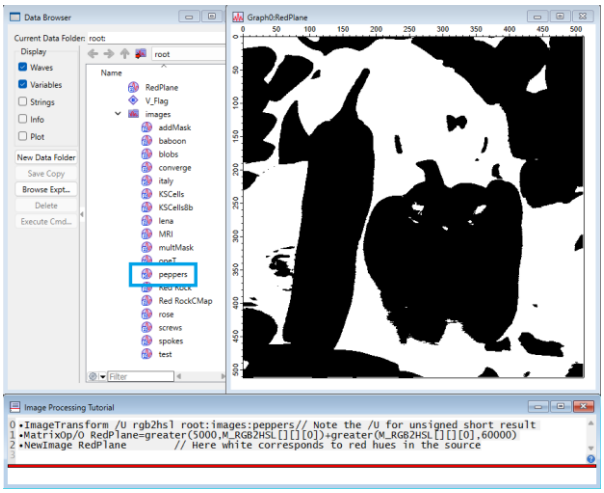
```
gray2DWave=c1*image[p][q][0]+c2*image[p][q][1]+c3*image[p][q][2]
```

大きな画像の場合、この操作は遅くなる可能性があります。
より効率的な方法は次のとおりです：

```
Make/O/N=3 scaleWave={c1,c2,c3}
ImageTransform/D=scaleWave scalePlanes image          // M_ScaledPlanes を作成
ImageTransform sumPlanes M_ScaledPlanes
```

一部のアプリケーションでは、画像内の領域の色から情報を抽出することが望ましい場合があります。
そこで、画像を RGB から HSL カラースペースに変換し、結果の 3D ウェーブの最初の平面（色相）に対して操作を行います。
次の例では、RGB 画像ウェーブ peppers を HSL に変換し、色相平面を抽出し、赤の色相が非ゼロとなる二値画像を生成します。

```
ImageTransform /U rgb2hsl root:images:peppers      // 符号なし整数の結果のため、/U に注目
MatrixOp/O RedPlane=greater(5000,M_RGB2HSL[][][0])+greater(M_RGB2HSL[][][0],60000)
NewImage RedPlane                                  // ここでは白はソースの赤色に対応する
```



結果の画像は白黒画像となり、元の画像で主に赤色だった領域に対応するピクセルが白で表示されます。
この白黒画像を使って、赤色領域と他の色相領域を区別できます。
上記の 2 行目のコマンドラインは、0 から 65535 の範囲の彩度値を、「赤みがかった」範囲内の色であれば 1

に、それ以外の範囲であれば 0 に変換します。
5000 未満の値を選択しているのは、赤の彩度が色相環の 0°（または 360°）の両側に現れるためです。

色相ベースの画像セグメンテーションは、キーワード hslSegment を使った ImageTransform コマンドを通じてサポートされています。
同じ操作は、RGB から CIE XYZ (D65 ベース) への変換、XYZ から RGB への変換といった色空間変換もサポートします。
詳細は「色の処理」と「HSL セグメンテーション」のセクションを参照してください。

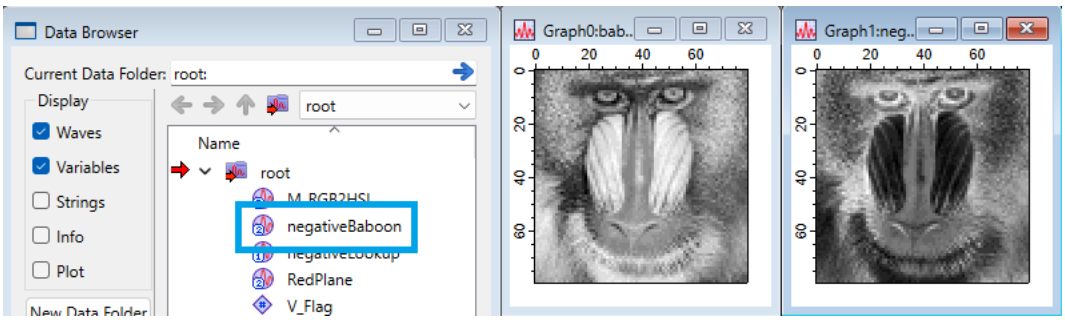
グレースケール変換または値変換

これらの変換は、2D ウェーブまたは高次元ウェーブの個々のレイヤーにのみ適用されます。
2D ウェーブ自体には色情報が含まれていないため、「グレースケール」と呼ばれます。
グレースケール変換は、レベルマッピングと数学的演算に分けられます。

明示的なルックアップテーブル

明示的なルックアップテーブル（LUT）を使って画像のネガを作成する、手間のかかる方法の例を次に示します。

```
Make/B/U/O/N=256 negativeLookup=255-x                    // ルックアップテーブルを作成
Duplicate/O root:images:baboon negativeBaboon
negativeBaboon=negativeLookup[negativeBaboon]            // ルックアップ変換
NewImage root:images:baboon
NewImage negativeBaboon
```



この例では、negativeBaboon 画像は標準的な線形 LUT で表示された派生ウェーブです。
元の baboon 画像を使い、負の LUT で表示しても同じ結果が得られます：

```
NewImage root:images:baboon
Make/N=256 negativeLookup=1-x/255                // Negative slope LUT from 1 to 0
ModifyImage baboon lookup=negativeLookup
```

元のデータを変更しても構わない場合は、次を実行できます。

```
ImageTransform invert baboon
```

ヒストグラム均等化

ヒストグラム均等化は、グレースケール画像の値をマッピングし、結果として得られる値が利用可能な輝度範囲全体を活用するようにします。

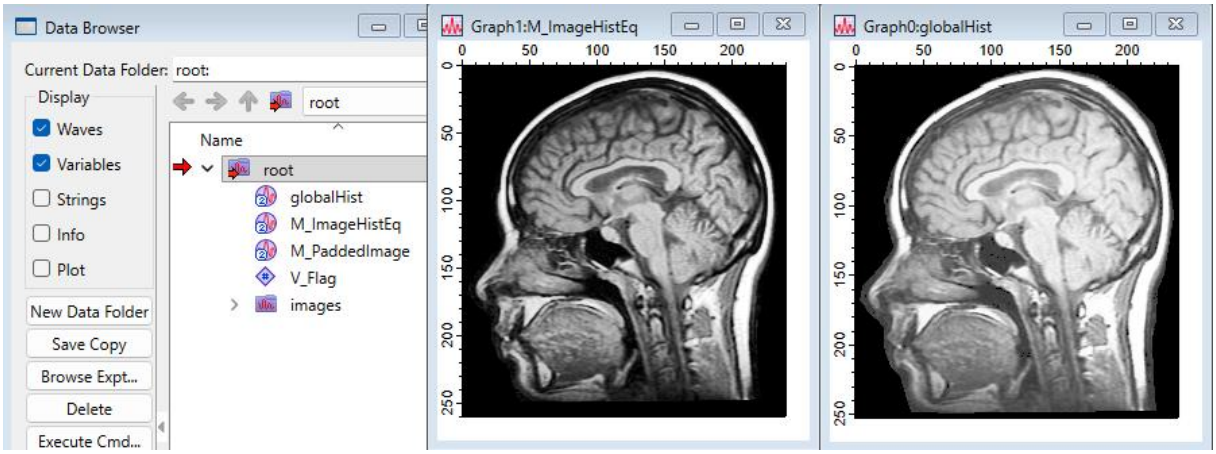
```
NewImage root:images:MRI
```

```
ImageHistModification root:images:MRI
NewImage M_ImageHistEq
```

適応ヒストグラム均等化

ImageHistModification コマンドは、ソース画像全体の累積ヒストグラムに基づいてルックアップテーブルを計算します。
その後、このルックアップテーブルが出力画像に適用されます。
ヒストグラムに空間的な変動が大きい場合、より局所的なアプローチが必要となる場合があります。
つまり、画像の異なる部分に対して個別にヒストグラム均等化を実行し、領域境界を越えて結果を整合させることで各領域の結果を統合するのです。
これは一般に「適応ヒストグラム均等化」と呼ばれます。

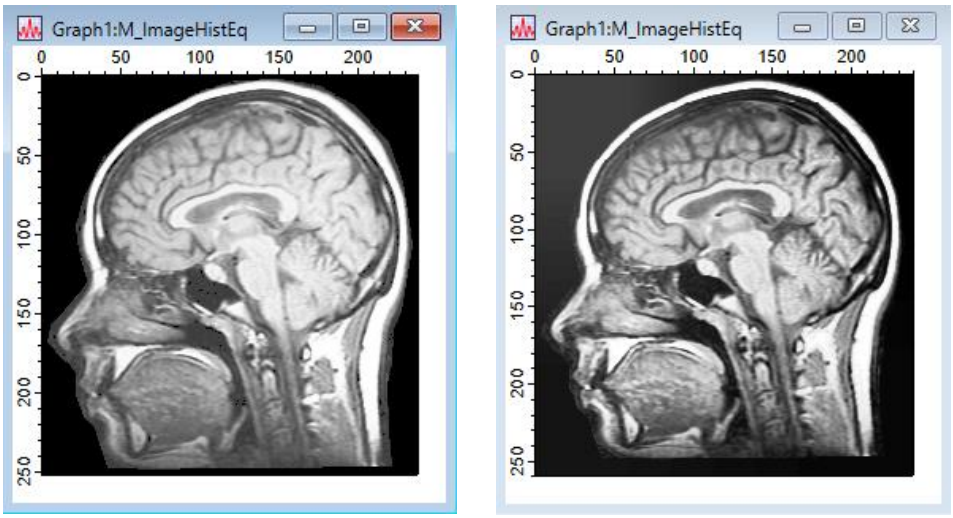
```
ImageHistModification root:images:MRI
Duplicate/O M_ImageHistEq, globalHist
NewImage globalHist
ImageTransform/N={2,7} padImage root:images:MRI              // 画像を分割可能にする
ImageHistModification/A/C=10/H=2/V=2 M_paddedImage
NewImage M_ImageHistEq
```



画像にパディングを施し、水平方向および垂直方向の画素数が、目的の間隔数で割り切れるようにしました。
結果として得られる適応ヒストグラムが、グローバルなヒストグラム均等化と十分に異なるものにならない場合は、処理する垂直方向および水平方向の領域数を増やすことができます。

```
ImageHistModification/A/C=100/H=20/V=20 M_paddedImage
```

グローバル **適応**



グローバルヒストグラムと適応ヒストグラムの結果を比較します。
適応ヒストグラムは画像の大部分でより良好な結果（コントラスト向上）を示しました。
クリッピング値（/C フラグ）の増加により、頭部の境界付近に軽微なアーティファクトが生じています。

閾値

閾値操作は、レベルマッピングクラスの重要なメンバーです。
これは、グレースケール画像をバイナリ画像に変換します。
Igor のバイナリ画像は、通常、符号なしバイト型のウェーブとして保存されます。
これは無駄に見えるかもしれませんが、速度と、各バイトのビットの一部を他の目的（たとえば、バイナリマスキングのためにビットをオンまたはオフにするなど）に使用できるという利点があります。
閾値処理は、二値化された閾値処理画像を生成するだけでなく、閾値の品質を測る指標となる相関値も提供できます。

ImageThreshold コマンドは、特定の閾値を指定するか、コマンド自体に閾値を決定させるかのいずれかで使用できます。
自動閾値決定には様々な方法があります。

Iterated: 閾値レベルを反復処理し、元の画像との相関を最大化します。

Bimodal: 画像ヒストグラムに二峰性分布を適合させようとする試みです。閾値レベルは 2 つのモードピークの間で選択されます。

Adaptive: 各ピクセルについて、同一走査線上の直近 8 ピクセルに基づいて閾値を算出します。これにより通常、走査線方向にドラッグラインが生じます。このアーティファクトは、以下の例に示すように補正が可能です。

Fuzzy Entropy: 画像を背景と対象物のピクセルからなるファジー集合とみなします。各ピクセルは確率をもっていずれかの集合に属します。アルゴリズムは、シャノンのエントロピー関数を用いて計算されるファジー度を最小化することで閾値値を得ます。

Fuzzy Means: 対象物に属するピクセルの確率と背景に属するピクセルの確率の積に基づくファジー度測定値を最小化します。

Histogram Clusters: データのヒストグラム分析を行い、画像をクラスターの集合として表現し、 2 つのクラスターが残るまで反復的に削減することで理想的な閾値を決定します。その後、閾値は下位クラスターの最高レベルに設定されます。この手法は A.Z. Arifin と A. Asano による論文（「参考文献」のセクションを参照）に基づくが、比較的平坦なヒストグラム

を持つ画像処理用に改良されています。画像ヒストグラムが 2 つ未満のクラスターしか生成しない場合、この手法による閾値決定は不可能であり、閾値は NaN に設定されます。Igor Pro 7.0 で追加されました。

Variance: 「対象」と「背景」の間の総分散を最大化することで理想的な閾値を決定します。
http://en.wikipedia.org/wiki/Otsu's_method
を参照してください。Igor Pro 7.0 で追加されました。

自動閾値設定の各手法には、それぞれ長所と短所があります。
特定の画像クラスに最適な手法を決定する前に、あらゆる異なる手法を試すことが有用な場合があります。
以降の例で Blob 画像に対する異なる閾値設定手法を示しています。

組み込み閾値手法の比較

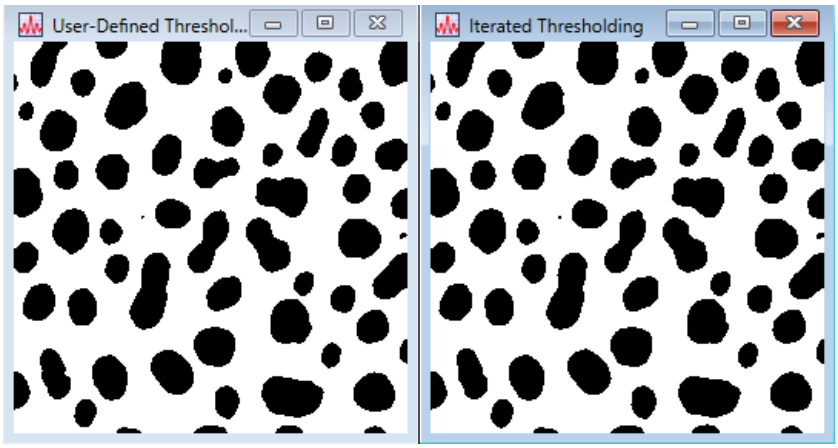
このセクションでは、自動閾値決定のための様々な手法を使った結果を示します。

表示されたコマンドは、File→Example Experiments→Tutorials→Image Processing Tutorial を選択して開くことができるデモエクスペリメントで実行しています。

```
// ユーザー定義の方法
ImageThreshold/Q/T=128 root:images:blobs
Duplicate/O M_ImageThresh UserDefined
NewImage/S=0 UserDefined; DoWindow/T kwTopWin, "User-Defined Thresholding"
```

```
// 反復法
ImageThreshold/Q/M=1 root:images:blobs
Duplicate/O M_ImageThresh iterated
NewImage/S=0 iterated; DoWindow/T kwTopWin, "Iterated Thresholding"
```

ユーザー定義 **反復法**




```
// 二峰性法
ImageThreshold/Q/M=2 root:images:blobs
Duplicate/O M_ImageThresh bimodal
NewImage/S=0 bimodal; DoWindow/T kwTopWin, "Bimodal Thresholding"
```

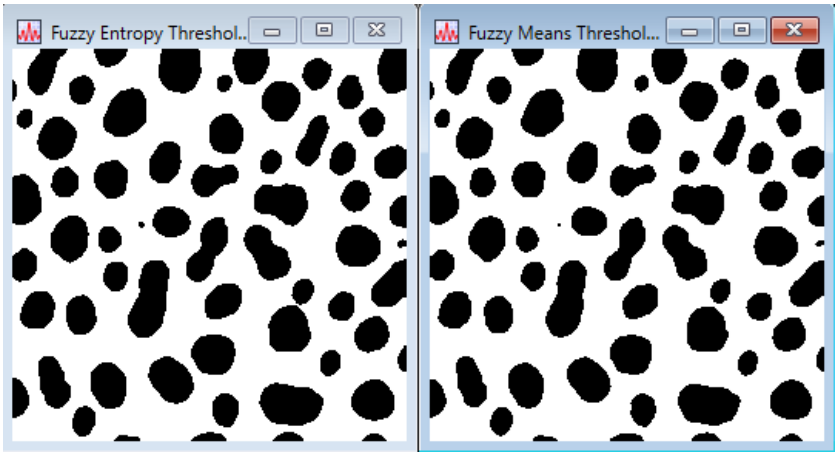
```
// 適応法
ImageThreshold/Q/I/M=3 root:images:blobs
Duplicate/O M_ImageThresh adaptive
NewImage/S=0 adaptive; DoWindow/T kwTopWin, "Adaptive Thresholding"
```

```
// ファジーエントロピー法
ImageThreshold/Q/M=4 root:images:blobs
```

```
Duplicate/O M_ImageThresh fuzzyE
NewImage/S=0 fuzzyE; DoWindow/T kwTopWin, "Fuzzy Entropy Thresholding"
```

```
// ファジー平均法
ImageThreshold/Q/M=5 root:images:blobs
Duplicate/O M_ImageThresh fuzzyM
NewImage/S=0 fuzzyM; DoWindow/T kwTopWin, "Fuzzy Means Thresholding"
```

ファジーエントロピー **ファジー平均**




```
// Arifin と Asano の方法
ImageThreshold/Q/M=6 root:images:blobs
Duplicate/O M_ImageThresh A_and_A
NewImage/S=0 A_and_A; DoWindow/T kwTopWin, "Arifin and Asano Thresholding"
```

```
// Otsu の方法
ImageThreshold/Q/M=7 root:images:blobs
Duplicate/O M_ImageThresh otsu
NewImage/S=0 otsu; DoWindow/T kwTopWin, "Otsu Thresholding"
```

Arifin と Asano **Otsu**

これらの例では、各 ImageThreshold コマンドに /C フラグを追加し、/Q フラグを削除することで、閾値の品質に関するフィードバックを得ることができます。
相関係数が履歴に表示されます。

このデータに対して、適応アルゴリズムと二峰性アルゴリズムの性能がかなり劣っていたことは、視覚的に容易に判断できます。

適応アルゴリズムの結果を改善するには、適応閾値処理を画像の転置にも適用し、列ベースの演算とします。
その後、 2 つの出力を論理 AND 演算で結合します。

空間変換

空間変換は、ウェーブ内におけるデータの位置を変更する一連の操作を表します。
これには、複数のキーワードを持つ ImageTransform コマンド、MatrixTranspose、ImageRotate、ImageInterpolate、ImageRegistration が含まれます。

画像の回転

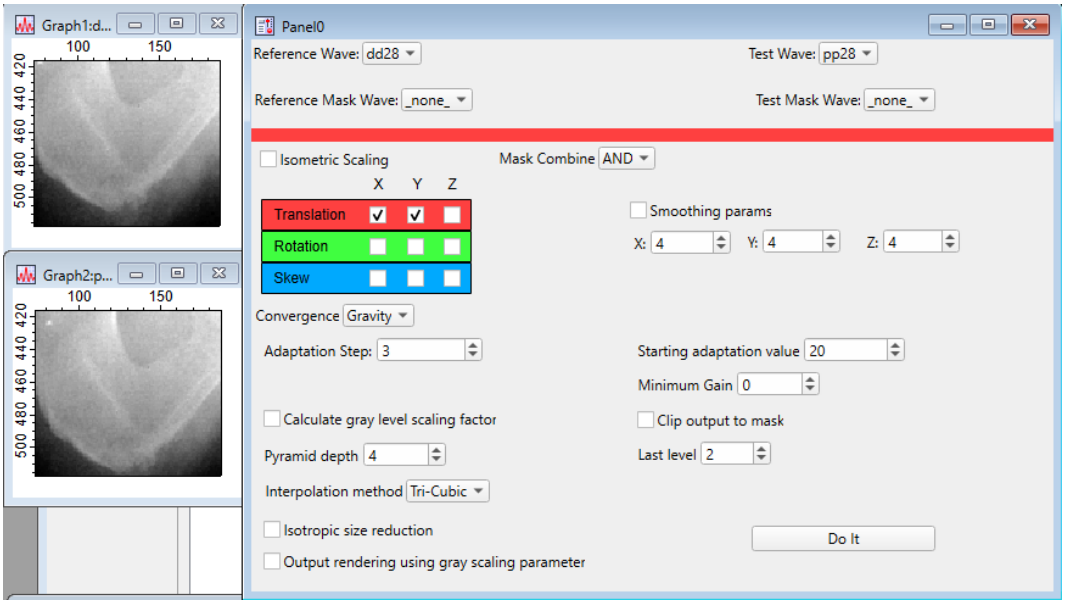
ImageRotate コマンドを使って画像を回転させることができます。
回転角度が 90 度の倍数ではない画像回転に関連して、注意すべき点が 2 つあります。
第一に、回転後の画像に元のピクセルをすべて収めるため、画像サイズは常に拡大されます（切り捨ては行われません）。
第二に、回転ピクセルは双線形補間を使って計算されるため、360/N 度の回転を N 回連続して行った場合、結果は一般的に元の画像とは一致しません。
複数回の回転を行う場合、すべての回転で元の画像を同一のソースとして使用できるよう、元の画像のコピーを保持することを検討すべきです。

画像の登録

多くの場合、同じ対象物の画像が複数存在し、それらの画像間の差異は撮影時間、異なる撮影機器、あるいは露出間の対象物の形状変化に起因します。
こうした画像間の比較を容易にするためには、画像の登録、すなわち互いに一致するように調整する必要があります。
ImageRegistration コマンドは、主要特徴が過度に異ならない場合に、テスト画像を基準画像に一致させるよう修正します。
このアルゴリズムはサブピクセル解像度を実現可能ですが、非常に大きなオフセットや大きな回転は扱えません。
アルゴリズムは、粗い詳細から細かい詳細へと進む反復処理に基づいています。
最適化には修正 Levenberg-Marquardt 法を使い、相対的な回転と平行移動のためのアフィン変換を生成します。
オプションで等尺性スケーリングとコントラスト調整も可能です。
このアルゴリズムは、回転中心が画像中心から大きく離れていない正方形画像で最も効果を発揮します。

ImageRegistration は、Thévenaz and Unser（1998）によって記述されたアルゴリズムに基づいています。

デモエクスペリメント：File→Example Experiments→Imaging→Image Registration Demo



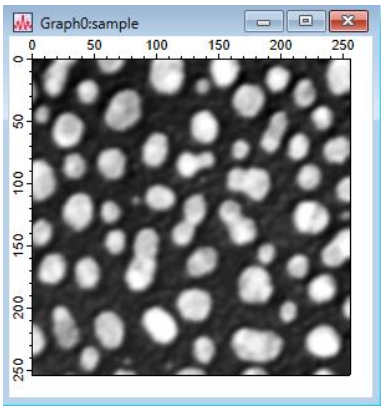
数学的変換

この変換クラスには、標準的なウェーブ代入、補間およびサンプリング、フーリエ変換、ウェーブレット変換、ハフ変換、ハートレー変換、畳み込みフィルター、エッジ検出器、形態学的演算子が含まれます。

標準的なウェーブの操作

グレースケール画像ウェーブは、すべてのウェーブ操作で使用可能な標準的な 2D ウェーブです。
例えば、単純な線形操作を実行できます。

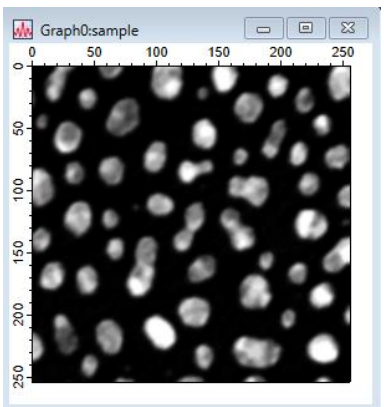
```
Duplicate/O root:images:blobs sample
Redimension/S sample                    // 単精度サンプルを作成
sample=10+5*sample                      // 線形操作
NewImage sample                         // 画像を表示したままにする
```



この線形変換に対して、画像の表示は不変であることに注意してください。

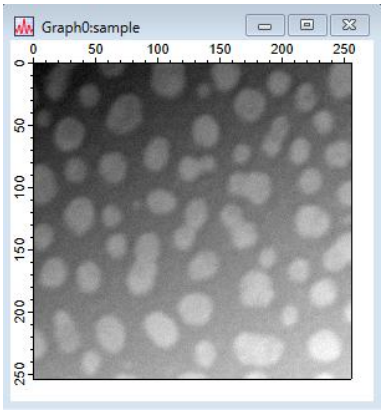
非線形操作も同様に簡単です。

```
sample=sample^3-sample                  // 特に役立つものではない
```



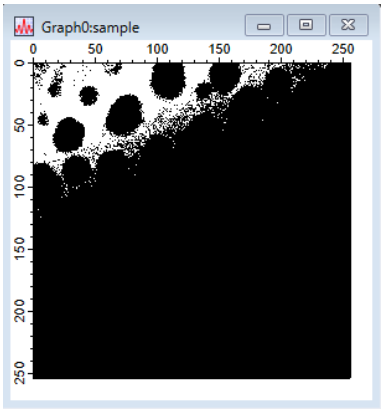
ノイズを追加したり背景を変更したりするには、簡単なウェーブの代入を使用できます。

```
sample=root:images:blobs          // オリジナルに戻す
sample+=gnoise(20)+x+2*y          // ガウスノイズと背景平面を追加
```



前述のいくつかの例で示したように、データからバイナリ画像を作成する必要が頻繁に生じます。
例えば、画像ウェーブサンプルにおいて 50 から 250 の値の間にある全てのピクセルを 255 に設定し、それ以外のピクセルを 0 に設定した画像を作成したい場合、次の１行のコマンドを使用できます。

```
MatrixOp/O sample=255*greater(sample,50)*greater(250,sample)
```



より効率的なウェーブの操作

このカテゴリには、特定の画像計算の性能を向上させるために設計されたいくつかの操作があります。
例えば、次のようなウェーブの代入を使って、複数平面画像から１つの平面（レイヤー）を取得できます。

```
Make/N=(512,512) newImagePlane
newImagePlane[][]=root:Images:peppers[p][q][0]
```

あるいは、次を実行することもできます。

```
ImageTransform/P=0 getPlane root:Images:peppers
```

または

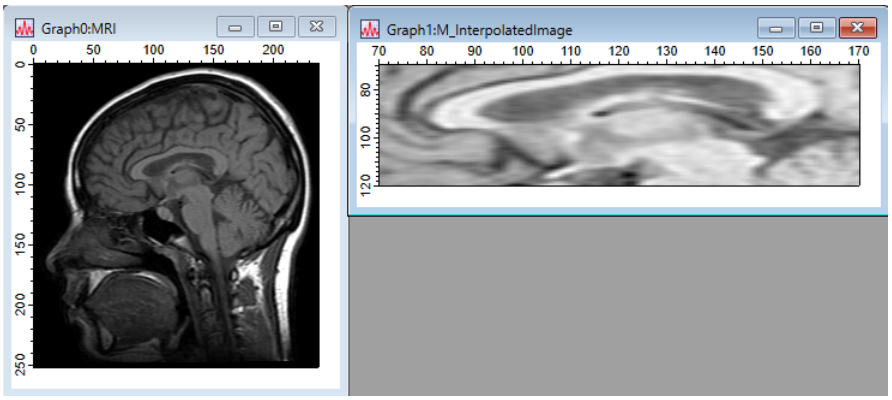
```
MatrixOp/O outWave=root:Images:peppers[][][0]
```

このサイズの画像では、単純なウェーブ代入よりも ImageTransform と MatrixOp の方がはるかに高速です。
詳細はヘルプ General Utilities: ImageTransform Operation および Using MatrixOp を参照してください。

補間とサンプリング

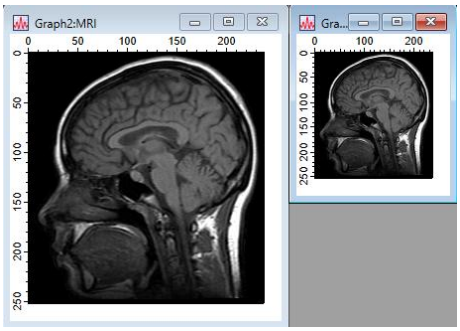
ImageInterpolate コマンドは補間ツールとしてもサンプリングツールとしても使用できます。
例えば、MRI 画像の一部から水平方向に４倍、垂直方向に２倍のサンプル数を持つ画像を作成します。

```
NewImage root:images:MRI
ImageInterpolate /S={70,0.25,170,70,0.5,120} bilinear root:images:MRI
NewImage M_InterpolatedImage
```



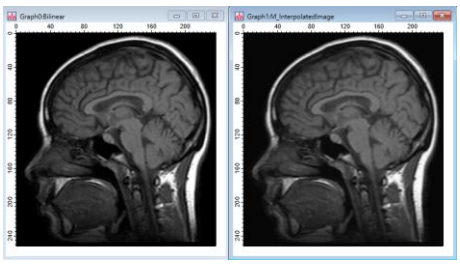
キーワードが示す通り、補間は双線形補間です。
同じコマンドで画像のサンプリングも行えます。
次の例では画像サイズを４分の１に縮小します：

```
NewImage root:images:MRI          // 比較を表示
ImageInterpolate /f={0.5,0.5} bilinear root:images:MRI
NewImage M_InterpolatedImage      // サンプリングされた画像
```



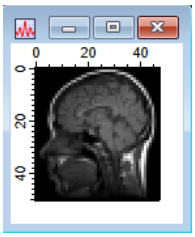
特定の画像のサイズを縮小する時には、最初にぼかし処理（例：MatrixFilter gauss）を適用すると効果的な場合があります。
これは、画像に細い（サンプルサイズより小さい）水平線または垂直線が含まれる場合に重要になります。

双線形補間が要件を満たさない場合、２次から５次のスプライン補間を使用できます。
以下は、画像の拡大縮小に用いた双線形補間と５次スプライン補間の比較です。



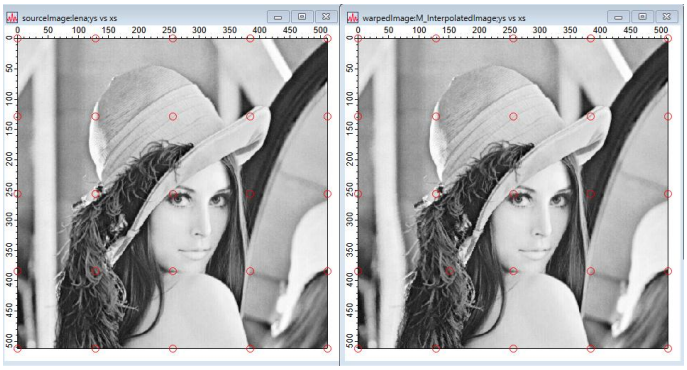
```
ImageInterpolate /f={1.5,1.5} bilinear root:images:MRI
Rename M_InterpolatedImage Bilinear
NewImage Bilinear
ImageInterpolate /f={1.5,1.5}/D=5 spline root:images:MRI
NewImage M_InterpolatedImage
```

サンプリングの別の形式として、ピクセル化された画像の作成があります。
ピクセル化された画像は、元の画像を n×y ピクセルの重なり合わない矩形に分割し、各矩形の平均ピクセル値を計算することで生成されます。



```
ImageInterpolate/PXSZ={5,5}/DEST=pixelatedImage pixelate, root:images:MRI
NewImage pixelatedImage
```

デモエクスペリメント：File→Example Experiments→Imaging→Image Warping Demo



（赤い○をドラッグすると画像が変化します）

高速フーリエ変換

フーリエ変換のイメージングへの応用に関する書籍は数多く存在するため、ここでは Igor Pro における FFT 演算の使用に関する技術的側面の一部のみを議論します。

留意すべき重要な点は、歴史的な理由から、デフォルト設定の FFT コマンドでは画像ウェーブが上書き・変更されることです。
FFT コマンドで宛先ウェーブを指定することも可能で、その場合はソースウェーブが保持されます。
２つ目に留意すべき点は、変換後のウェーブが複素データ型に変換されることであり、この変換に対応するためウェーブのポイント数も変更されることです。
３つ目の問題は、実数ウェーブに対して FFT 演算を実行した場合、結果が片側スペクトルとなることです。
つまり、結果を反射・複素共役することでスペクトルの残りの部分を取得する必要があります。

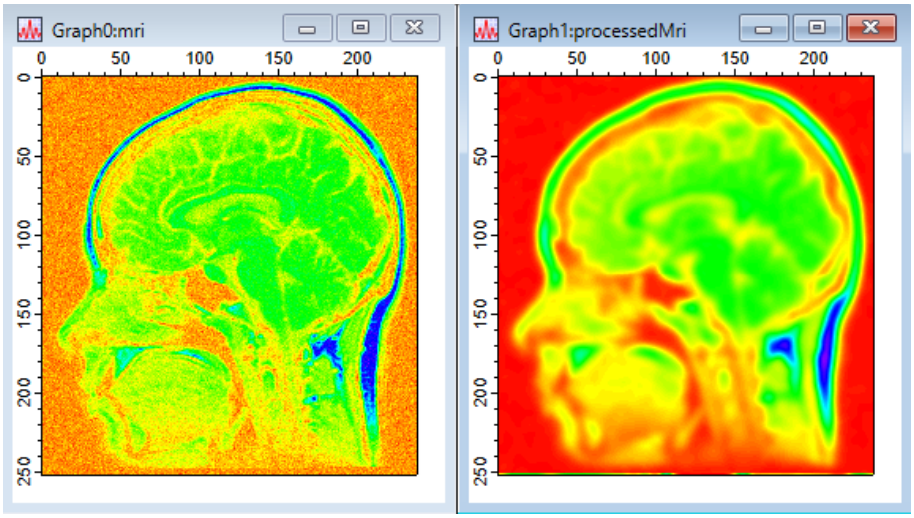
画像処理における FFT の典型的な応用例は、2N 行× M 列の実数ウェーブを変換することです。
FFT の結果は複素数で (N+1) 行× M 列となります。
元の画像ウェーブのウェーブスケーリングが dx と dy の場合、新しいウェーブスケーリングはそれぞれ 1/(N*dx) と 1/(M*dy) に設定されます。

以下の例は、画像処理における FFT の代表的な応用例をいくつか示しています。

畳み込みの計算

FFT を使って畳み込みを計算するには、入力ウェーブと畳み込みカーネルウェーブが同じ次元である必要があります（代替手法については MatrixOp コマンドのヘルプ convolve を参照）。
例えば、ガウス関数による畳み込みでノイズを平滑化する方法を考えてみます。

```
// ノイズの多い画像を作成して表示する
Duplicate /O root:images:MRI mri                    // これは符号なしバイト画像
Redimension/s mri                                   // 単精度に変換
mri+=gnoise(10)                                     // ノイズを追加
NewImage mri
ModifyImage mri ctab= {*,*,Rainbow,0}               // 偽色でノイズを表示

// フィルターウェーブを作成
Duplicate/O mri gResponse                           // ウェーブの高さが同じになるようにする
SetScale/I x -1,1,"" gResponse
SetScale/I y -1,1,"" gResponse

// 以下のガウシアンの幅を変更して、平滑化の量を設定
gResponse=exp(-(x^2+y^2)/0.001)

// 畳み込みを計算
Duplicate/O mri processedMri
FFT processedMri                                    // ソースを変換
FFT gResponse                                       // カーネルを変換
processedMri*=gResponse                             // （複素数）周波数空間における乗算
IFFT processedMri

// 比較のために結果を表示
SetScale/P x 0,1,"", processedMri                   // ラベルを削除
SetScale/P y 0,1,"", processedMri

// 結果を適切に中央に配置するために IFFT を入れ替える
ImageTransform swap processedMri
Newimage processedMri
ModifyImage processedMri ctab= {*,*,Rainbow,0}
```



注記： 実際には、より少ないコマンドで畳み込みを実行できます。上記の例では、より明確にするためにいくつかのコマンドが設計されています。また、ガウスフィルターを作成するために SetScale コマンドを使ったことにも注意してください。これは、ガウスがフィルター画像の中心で作成されるようにするためであり、ImageTransform の swap コマンドと互換性のある選択です。この例は理想的とは言えません。ガウシアンの性質（ガウシアンのフーリエ変換もガウシアンとなる）を利用すれば、次のように畳み込みを実行できるためです。

```
// 畳み込みを計算
Duplicate/O mri shortWay
FFT shortWay
shortWay*=cmplx(exp(-(x^2+y^2)/0.01),0)
IFFT shortWay
Newimage shortWay
ModifyImage shortWay ctab={*,*,Rainbow,0}
```

空間周波数フィルタリング

空間周波数フィルタリングの概念は、データを空間周波数領域に変換することにあります。
周波数領域に移行すれば、画像の空間周波数分布を修正し、その後逆変換を行って修正後の画像を得ることができます。

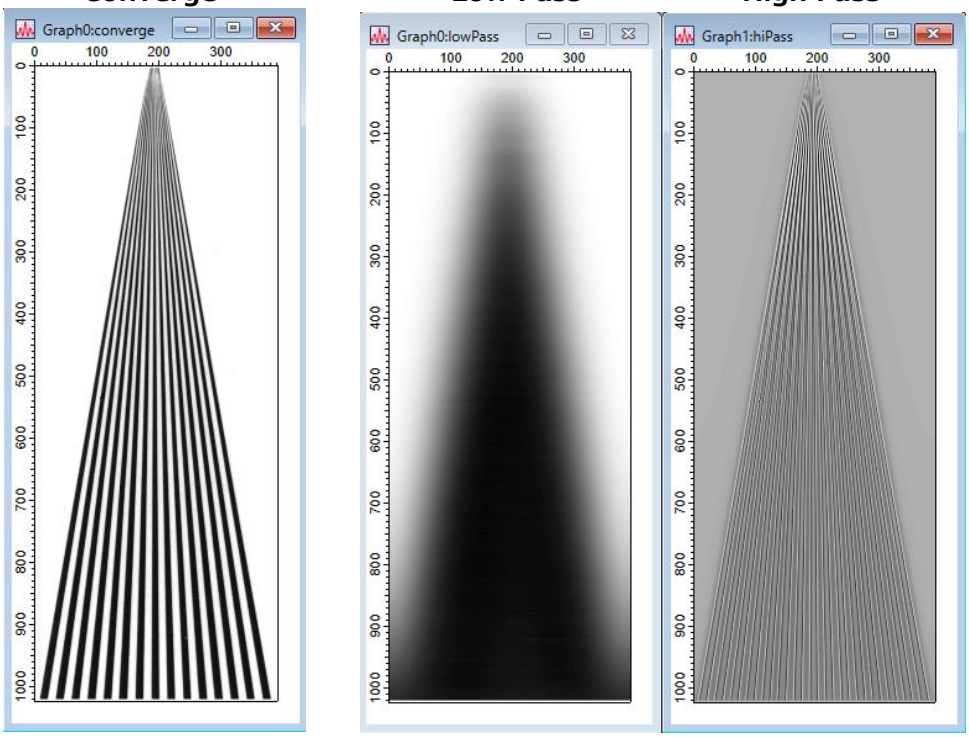
以下にローパスフィルタリングとハイパスフィルタリングの例を示します。
収束画像は太い黒線が一点に収束する構造を持ちます。
画像中央より下部の任意の位置に水平線プロファイルを描くと、15 個の長方形が連続して現れ、水平方向において広範囲の空間周波数が生じます。

ローパスフィルタリングとハイパスフィルタリング

```
// FFT の準備； SP ウェーブまたは DP ウェーブが必要
Redimension /s root:images:converge
FFT root:images:converge
Duplicate/O root:images:converge lowPass                // 周波数領域における新たな複素数ウェーブ
lowPass=lowPass*cmplx(exp(-(p)^2/5),0)
IFFT lowpass
SetScale/P x 0,1,"", lowPass                            // ラベルを削除
SetScale/P y 0,1,"", lowpass
NewImage lowPass                                        // 非最適ローパス
```

```
Duplicate/O root:images:converge hiPass
hiPass=hiPass*cmplx(1-1/(1+(p-20)^2/2000),0)
IFFT hiPass
SetScale/P x 0,1,"", hiPass                    // ラベルを削除
SetScale/P y 0,1,"", hiPass
NewImage hiPass                                // 非最適ハイパス
```

Converge **Low Pass** **High Pass**



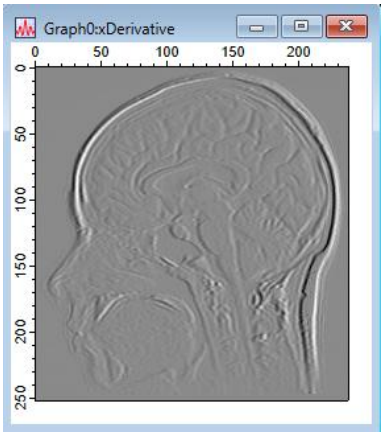
ローパスフィルターには、任意にガウス関数を選択しました。
実用上、正確な「カットオフ」周波数を選択すると同時に、リンギングなどの望ましくないフィルタリングアーティファクトを発生させないよう、十分に滑らかなフィルターを選択することが通常重要です。
上記で使ったハイパスフィルターは、低周波数を遮断するノッチフィルターに近いです。
どちらのフィルターも本質的に一次元フィルターです。

導関数の計算

フーリエ変換の導関数性質を使うことで、例えば画像の X 方向の導関数を次のように計算できます。

```
Duplicate/O root:images:mri xDerivative        // オリジナルを維持
Redimension/S xDerivative
FFT xDerivative
xDerivative *= cmplx(0,p)                      // 2pi の係数とウェーブのスケールを無視
IFFT xDerivative
SetScale/P x 0,1,"", xDerivative               // ラベルを削除
SetScale/P y 0,1,"", xDerivative
NewImage xDerivative
```

この手法は全てのアプリケーションで魅力的とは限りませんが、高次導関数の計算が必要な場合にはその利点が明らかになります。
また、この手法では行または列に関連付けられる可能性のあるウェーブスケーリングは考慮されていない点に留意してください。



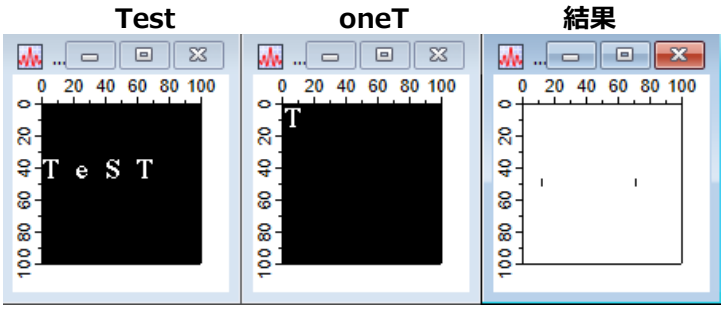
積分または和の計算

フーリエ変換のもう一つの有用な特性は、軸に沿った変換値が画像の積分に対応することです。
この目的のために FFT を使うことには、通常、利点はりません。
しかし、他の目的で FFT が計算される場合には、この特性を活用することができます。
これが有用となる典型的な状況は、相関係数（正規化相互相関）の計算です。

相関関係

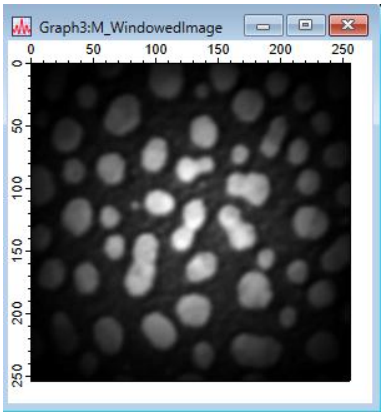
FFT は、特定の画像内で特定のサイズと形状を持つ物体を検出するために使用できます。
次の例は、テスト対象が画像内で検出される物体と同じスケールと回転角度を持つという点で、かなり単純です。

```
// テスト画像には「Test」という単語が含まれる
NewImage root:images:test                       // ２つの T を探す
Duplicate/O root:images:oneT oneT               // 探しているオブジェクト
NewImage oneT

Duplicate/O root:images:test testf              // FFT が上書きするため
FFT testf
Duplicate/O oneT oneTf
FFT oneTf
testf*=oneTf                                    // 「適切な」相関ではない
IFFT testf
SetScale/P x 0,1,"", testf                      // ラベルを削除
SetScale/P y 0,1,"", testf
ImageThreshold/O/T=1.25e6 testf                 // ノイズを除去（他の文字との重複による）
NewImage testf                                  // 結果は T の相関スポット
```



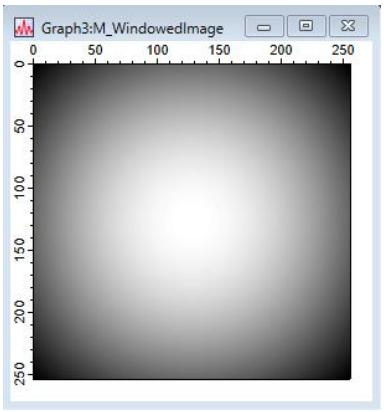
FFT を使う時、画像境界に近づくにつれてピクセル値が滑らかにゼロになるよう、ソース画像に組み込みのウィンドウ関数のいずれかを適用する必要がある場合があります。
ImageWindow コマンドは、Hanning、Hamming、Bartlett、Blackman、Kaiser の各ウィンドウをサポートしています。
通常、ImageWindow コマンドは次の例のように画像に直接作用します。

```
// FFT コマンドはいずれにせよ再次元化が必要なので、
// ここで実行して ImageWindow コマンドにおける
// 結果の量子化を低減しても差し支えない。
Redimension/s root:images:blobs
ImageWindow /p=0.03 kaiser root:images:blobs
NewImage M_WindowedImage
```



ウィンドウ関数がどのようなものかを確認するには：

```
// SP または DP ウェーブは必須
Redimension/S root:images:blobs
// ウィンドウデータを作成
ImageWindow/i/p=0.01 kaiser root:images:blobs
// これからサーフェスプロットを作成することもできる
NewImage M_WindowedImage
```



ウェーブレット変換

ウェーブレット変換は主に平滑化、ノイズ低減、非可逆圧縮に用いられます。
いずれの場合も、まず画像を変換し、次に変換されたウェーブに対して何らかの演算を行い、最後に逆変換を計算するという手順を踏みます。

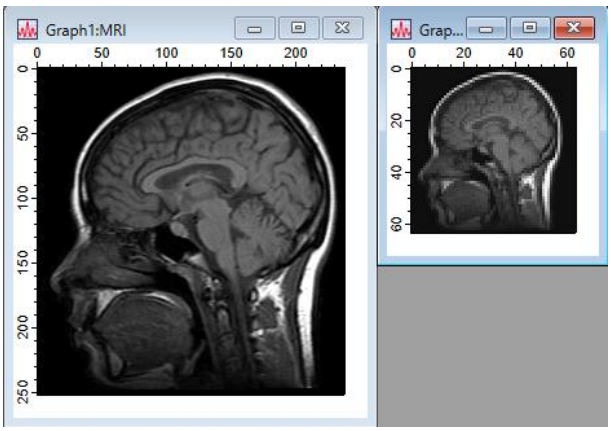
次の例は、ウェーブレット圧縮のプロシージャを示しています。
まず、画像のウェーブレット変換を計算します。
選択するウェーブレットと係数は、圧縮品質に大きく影響します。
圧縮された画像は、変換における低次係数に対応するウェーブの部分です（2D フーリエ変換におけるローパスフィルタリングと同じ）。
この例では、変換の 64x64 領域からウェーブを生成するために ImageInterpolate コマンドを使います。

```
DWT /N=4/P=1/T=1 root:images:MRI,wvl_MRI                    // perform the Wavelet transform
ImageInterpolate/S={0,1,63,0,1,63} bilinear wvl_MRI    // reduce size by a factor of 16
```

画像を再構築し圧縮品質を評価するため、圧縮画像を逆変換し結果を表示します。

```
DWT /I/N=4/P=0/T=1 M_InterpolatedImage,iwvl_compressed
NewImage iwvl_compressed
```

Original **Compressed**

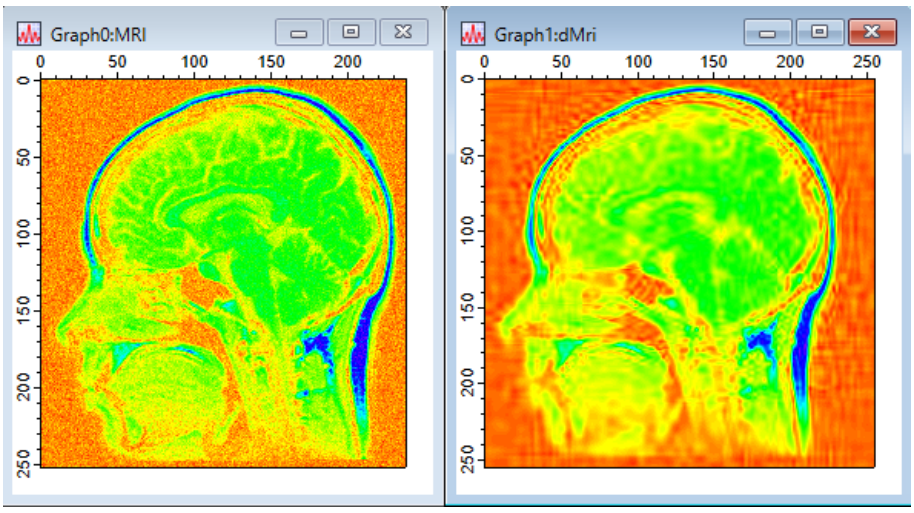


再構築された画像には圧縮に関連するアーティファクトがいくつか見られますが、FFT ベースのローパスフィルターとは異なり、ウェーブレット変換の利点は画像にかなりの高空間周波数成分が含まれている点です。
前述の１６倍という数値は完全には正確ではありません。
というのも、元の画像は１ピクセルあたり１バイトで保存されていたのに対し、圧縮画像は浮動小数点値で構成されているためです（したがって真の圧縮率は４倍に過ぎません）。

ウェーブレット変換のノイズ除去への応用例を示すため、まず標準偏差 10 のガウス分布ノイズを MRI 画像に付加します。

```
Redimension/S root:images:Mri                          // SP：バイポーラノイズを追加できる
root:images:Mri+=gnoise(10)                            // ガウスノイズを追加
NewImage root:images:Mri
ModifyImage Mri ctab={*,*,Rainbow,0}                   // 識別を容易にするための偽色
DWT/D/N=20/P=1/T=1/V=0.5 root:images:Mri,dMri          // ノイズ除去を強化するため /V を増やす
NewImage dMri                                          // ノイズが除去された画像を表示
ModifyImage dMri ctab={*,*,Rainbow,0}
```

Mri **dMri**



Hough 変換

Hough 変換は、画像空間における直線を変換空間内の 1 つのポイントに写像するマッピングアルゴリズムです。
これは線検出に最も頻繁に使われます。
具体的には、画像空間の各ポイントが変換空間内の正弦曲線に写像されます。
画像内のピクセルが直線上に並んでいる場合、これらのピクセルに関連する正弦曲線はすべて変換空間内の１つのポ

イントで交差します。
変換空間内の各ポイントで交差する正弦曲線の数を数えることで、線を検出できます。

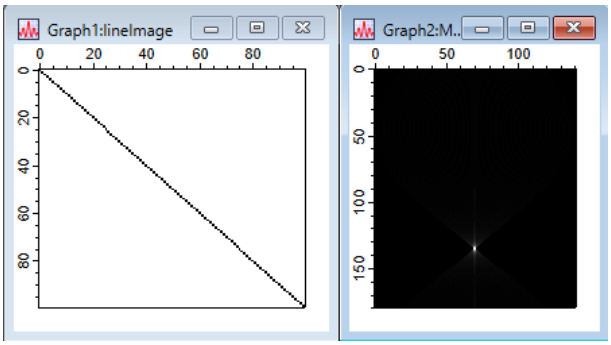
```
Make/O/B/U/N=(100,100) lineImage
lineImage=(p==q ? 255:0)                          // 45 度の一本の線
Newimage lineimage
Imagetransform hough lineImage
Newimage M_Hough
```

線の集まりの Hough 変換：

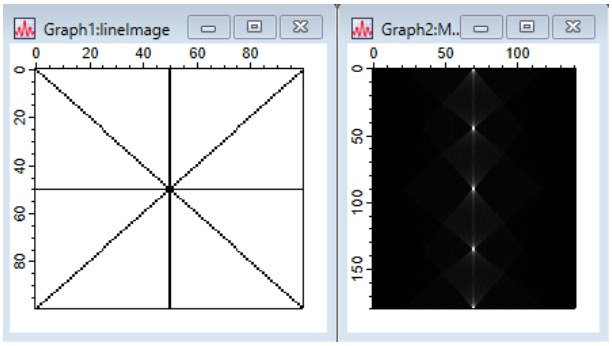
```
lineImage=((p==100-q)|(p==q)|(p==50)|(q==50)) ? 255:0
Imagetransform Hough lineImage
```

最後の画像には中央に一連の明るいピクセルが映っています。
最初と最後の点はそれぞれ 0 度と 180 度の線に対応します。
上から 2 番目の点は 45 度の線に対応します。

線１本



複数線



高速 Hartley 変換

Hartley（ハートレー）変換はフーリエ変換と似ていますが、実数値のみを用いる点が異なります。
この変換は以下で定義される *cas* カーネルに基づきます。

$$cas = \cos(vx) + \sin(vx)$$

離散 Hartley 変換は次式で与えられます。

$$H(u,v) = \frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} f(x,y)\left\{\cos\left(2\pi\left[\frac{ux}{M}-\frac{vy}{N}\right]\right)+\sin\left(2\pi\left[\frac{ux}{M}-\frac{vy}{N}\right]\right)\right\}$$

Hartley 変換には２つの興味深い数学的性質があります。
第一に、逆変換は正変換と同じで、第二に、パワースペクトルは次の式で与えられます。

$$P(f) = \frac{[H(f)]^2 + [H(-f)]^2}{2}$$

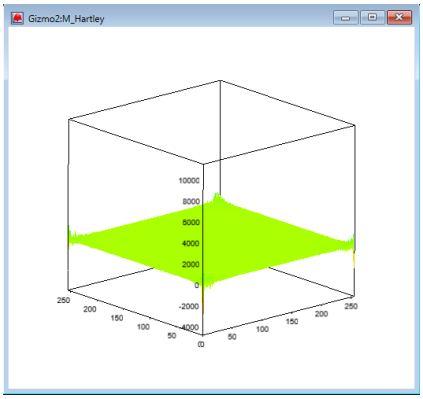
高速 Hartley 変換の実装は、ImageTransform コマンドの一部です。
これには、入力ウェーブが 2 のべき乗の次元を持つ画像であることが要求されます。

```
ImageTransform /N={18,3}/O padImage root:images:mri          // 256^2 画像を作成
ImageTransform fht root:images:mri
NewImage M_Hartley
```

実行結果は一面が黒い画像となります。
わかりやすくするために 3D 表示にします。

```
NewGizmo;DelayUpdate
AppendToGizmo DefaultSurface=M_Hartley
```

四隅に変化が見られます。

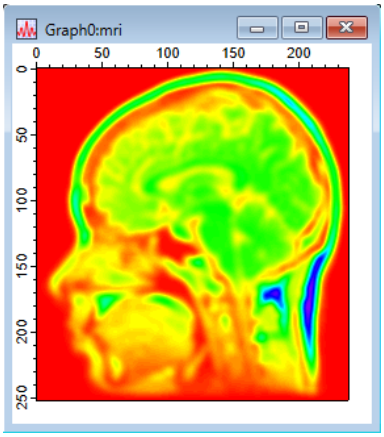


畳み込みフィルター

畳み込みコマンドは通常、画像と畳み込み演算を行うことで望ましい効果（単純な線形フィルタリング）を生成する２次元カーネルのクラスを指します。
場合によっては、FFT を使った畳み込み（先述の畳み込み例と同じ）の方が効率的です。
つまり、画像とフィルターウェーブの両方を変換し、周波数領域で変換を乗算した後、逆変換を IFFT を使って計算します。
FFT のアプローチは、13×13 ピクセルを超えるカーネルとの畳み込みにおいてより効率的です。
しかし、画像処理において重要な役割を果たす有用なカーネルの多くは、3×3 または 5×5 サイズです。
これらのカーネルは非常に小さいため、対応する線形フィルターを FFT を使わずに直接畳み込みとして実装することは、かなり効率的です。

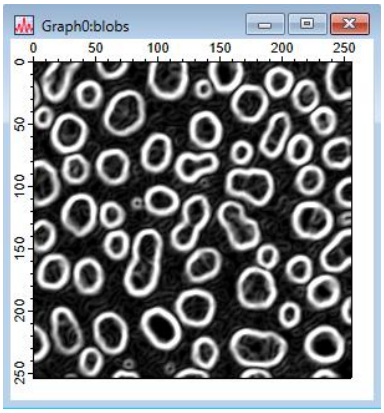
次の例では、両軸に沿って等しい空間周波数応答を持つローパスフィルターを実装します。

```
// 最初に畳み込みカーネルを作成
Make /O/N=(9,9) sKernel
SetScale/I x -4.5,4.5,"", sKernel
SetScale/I y -4.5,4.5,"", sKernel

// 空間周波数領域における rect(2*fx)*rect(2*fy) に相当する
sKernel=sinc(x/2)*sinc(y/2)

// 注意：MatrixConvolve はインプレースで実行
// 画像を先に保存してください！
Duplicate/O root:images:MRI mri
// 整数の切り捨てを避けるため
Redimension/S mri
MatrixConvolve sKernel mri
NewImage mri
// よく見えるようにするため
ModifyImage mri ctab= {*,*,Rainbow,0}
```



次の例は、ImageFilter コマンドで組み込みの畳み込みフィルターを使ってエッジ検出を実行する方法を示しています。

```
Duplicate/O root:images:blobs blobs
ImageFilter findEdges blobs
NewImage blobs
```



その他の注目すべき画像フィルターの例としては、Gauss、Median、Sharpen があります。
同じコマンドを 3D ウェーブにも適用できます。
Gauss3D、avg3D、point3D、min3D、max3D、median3D といったフィルターは、2D 版を 3x3x3 ボクセル近傍に拡張したものです。
最後の 3 つのフィルターは真の畳み込みフィルターではないことに注意してください。

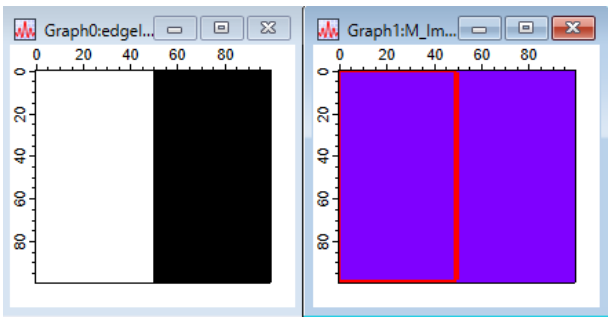
エッジ検出器

多くのアプリケーションでは、画像に現れるオブジェクトのエッジ／境界を検出する必要があります。
エッジ検出は、グレースケール画像からバイナリ画像を作成することで構成されます。
バイナリ画像のピクセルは、領域境界に属するかどうかに応じてオンまたはオフになります。
つまり、検出されたエッジはベクトル（1D ウェーブ）ではなく、画像によって記述されます。
領域境界を表すウェーブを取得する必要がある場合は、ImageAnalyzeParticles コマンドを使うことを検討するとよいでしょう。

Igor は 8 種類の組み込みエッジ検出器（メソッド）をサポートしており、ソース画像によって性能が異なります。
いくつかのメソッドでは、結果の品質に大きな影響を与える複数のパラメーターを指定する必要があります。
以下の例で、これらの選択の重要性を説明します。

新しいエクスペリメントで次を実行します。

```
// 単純な人工的なエッジ画像を作成し表示
Make/B/U/N=(100,100) edgeImage
edgeImage=(p<50? 50:5)
Newimage edgeImage

// 反復閾値検出を用いた単純な Sobel 検出器を試す
ImageEdgeDetection/M=1/N Sobel, edgeImage
NewImage M_ImageEdges
ModifyImage M_ImageEdges explicit=0            // このバイナリ画像をカラーで表示
ModifyImage M_ImageEdges ctab= {*,*,Rainbow,0}
```



この結果（赤い線）はほぼ予想通りです。
同様に良好な結果を示す他の例を次に示します。

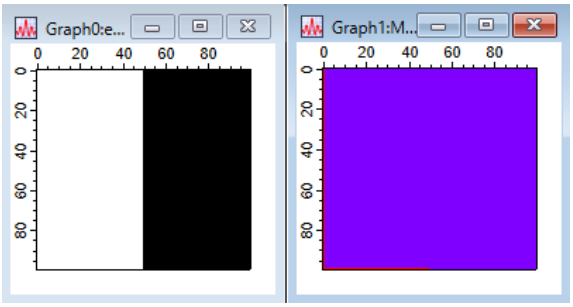
```
ImageEdgeDetection/M=1/N Kirsch, edgeImage          // 同じ出力ウェーブ
```

または

```
ImageEdgeDetection/M=1/N Roberts, edgeImage         // 同じ出力ウェーブ
```

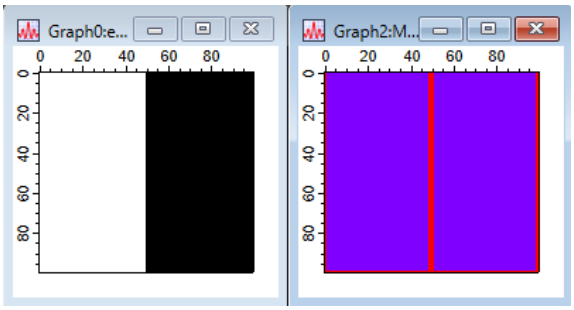
一見無害に見える /M=1 フラグは、このコマンドが反復的な自動閾値処理を使っていることを示唆しています。
上記の例ではうまく機能しているように見えますが、Frei フィルターを使うと完全に失敗します。

```
ImageEdgeDetection/M=1/N Frei, edgeImage
```



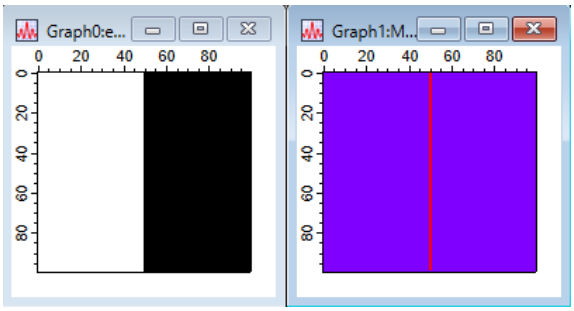
一方、二峰性フィッティングの閾値処理は次のようにはるかに良好に機能します。

```
ImageEdgeDetection/M=2/N Frei, root:edgeImage
```



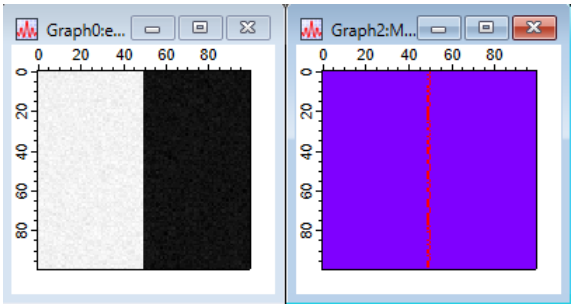
より特殊なエッジ検出器の中には、ノイズやグレースケールの変動がある場合に最も効果を発揮するものもあります。
単純な画像の場合、それらはエッジを完全に見逃す可能性があります。

```
ImageEdgeDetection/M=1/N/S=1 Canny, edgeImage
```



このフィルターの性能は、画像にわずかなノイズを加えると劇的に向上します。

```
edgeImage+=gnoise(1)
ImageEdgeDetection/M=1/N/S=1 Canny, edgeImage
```



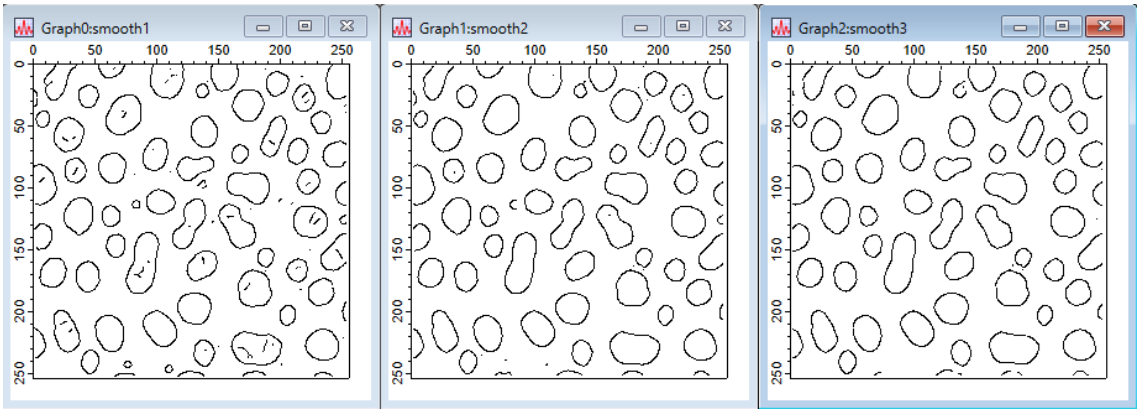
より特殊なエッジ検出器の使用

より特殊なエッジ検出器は、通常平滑化と微分を含む多段階処理で構成されます。
平滑化の効果を示す例を以下に示します。

Image Processing Tutorial エクスペリメントを使います。

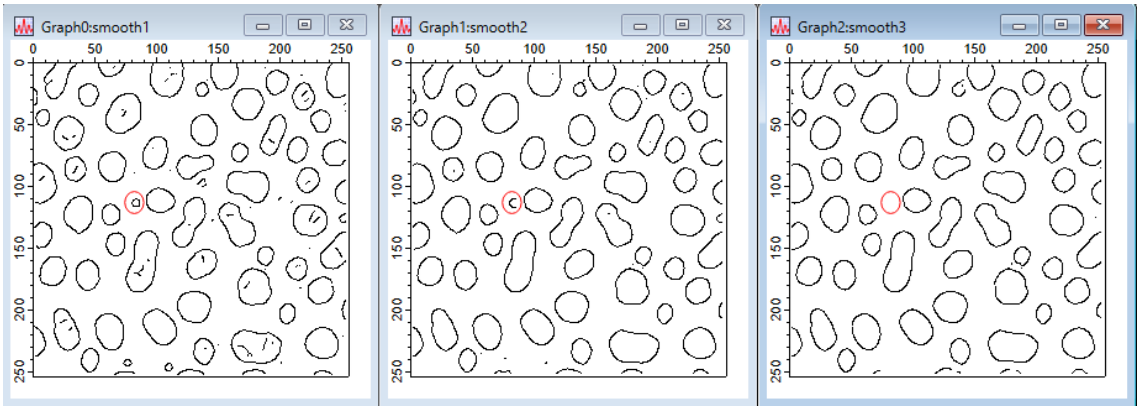
```
Duplicate/O root:images:blobs blobs
ImageEdgeDetection/M=1/N/S=1 Canny,blobs
```

```
Duplicate/O M_ImageEdges smooth1
ImageEdgeDetection/M=1/N/S=2 Canny,blobs
Duplicate/O M_ImageEdges smooth2
ImageEdgeDetection/M=1/N/S=3 Canny,blobs
Duplicate/O M_ImageEdges smooth3
NewImage smooth1
ModifyImage smooth1 explicit=0
NewImage smooth2
ModifyImage smooth2 explicit=0
NewImage smooth3
ModifyImage smooth3 explicit=0
```



上記の通り、３番目の画像（smooth3）は確かに１番目や２番目よりもはるかにクリーンですが、その結果は小さなブロブの一部を失う代償を得ています。
例えば、失われたブロブの１つを特定するには以下を実行します。

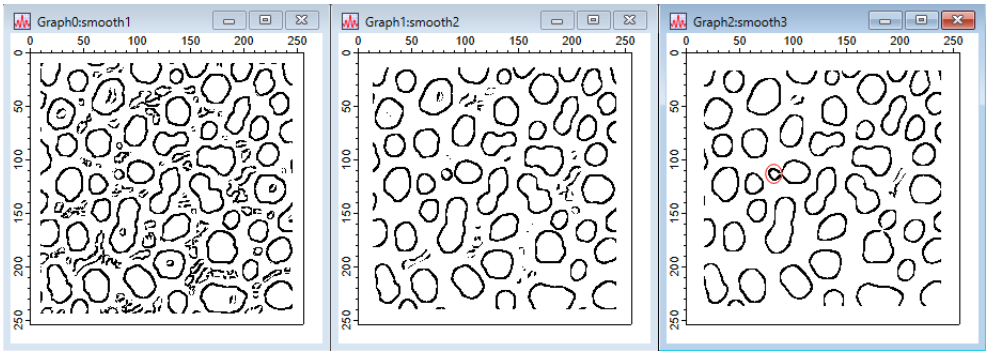
```
DoWindow/F Graph0
SetDrawLayer UserFront
SetDrawEnv linefgc= (65280,0,0),fillpat= 0
DrawOval 0.29,0.41,0.35,0.48
DoWindow/F Graph1
SetDrawLayer UserFront
SetDrawEnv linefgc= (65280,0,0),fillpat= 0
DrawOval 0.29,0.41,0.35,0.48
DoWindow/F Graph2
SetDrawLayer UserFront
SetDrawEnv linefgc= (65280,0,0),fillpat= 0
DrawOval 0.29,0.41,0.35,0.48
```



Marr 検出器と Shen 検出器を使って同様の画像セットを作成することは有益です。

```
// 注意：この処理の実行には時間がかかることがあります
Duplicate/O root:images:blobs blobs
ImageEdgeDetection/M=1/N/S=1 Marr,blobs
Duplicate/O M_ImageEdges smooth1
```

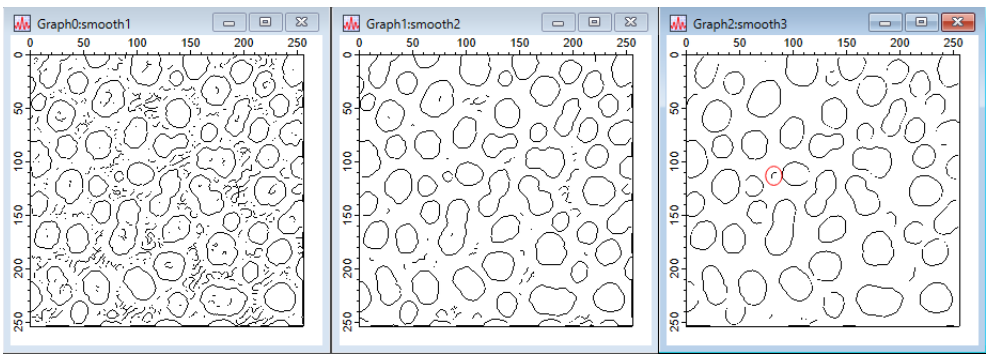
```
ImageEdgeDetection/M=1/N/S=2 Marr,blobs
Duplicate/O M_ImageEdges smooth2
ImageEdgeDetection/M=1/N/S=3 Marr,blobs
Duplicate/O M_ImageEdges smooth3
NewImage smooth1
ModifyImage smooth1 explicit=0
NewImage smooth2
ModifyImage smooth2 explicit=0
NewImage smooth3
ModifyImage smooth3 explicit=0
SetDrawLayer UserFront
SetDrawEnv linefgc= (65280,0,0),fillpat= 0
DrawOval 0.29,0.41,0.35,0.48
```



計算されたエッジの３つの画像は、畳み込みカーネルのサイズが大きくなるにつれてノイズが減少することを示しています。
また、Canny 検出器では消えていたブロブが、Marr 検出器では明瞭に確認できる点も特筆に値します。

次の例では、様々な平滑化係数を使った Shen-Castan 検出器を使います。
このエッジ検出アルゴリズムは標準的な閾値処理を使わない点に注意してください（閾値は /F フラグで指定する必要があります）。

```
Duplicate/O root:images:blobs blobs
ImageEdgeDetection/N/S=0.5 shen,blobs
Duplicate/O M_ImageEdges smooth1
ImageEdgeDetection/N/S=0.75 shen,blobs
Duplicate/O M_ImageEdges smooth2
ImageEdgeDetection/N/S=0.95 shen,blobs
Duplicate/O M_ImageEdges smooth3
NewImage smooth1
ModifyImage smooth1 explicit=0
NewImage smooth2
ModifyImage smooth2 explicit=0
NewImage smooth3
ModifyImage smooth3 explicit=0
SetDrawLayer UserFront
SetDrawEnv linefgc=(65280,0,0),fillpat=0
DrawOval 0.29,0.41,0.35,0.48
```

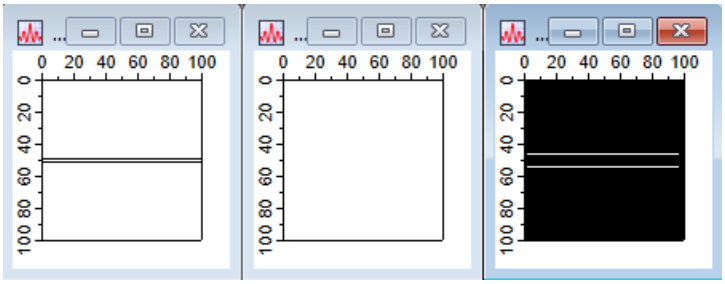


この例でわかるように、Shen 検出器は細い境界を生成しますが、時折途切れることがあります。
ノイズ低減はエッジ品質とのトレードオフです。

平滑化を使うエッジ検出器の問題点の１つは、互いに比較的近い位置にある２つのエッジが存在する場合に誤りを生じやすいことです。
次の例では、この点を示す人工的な画像を作成します。

```
Make/B/U/O/N=(100,100) sampleEdge=0
sampleEdge[][49]=255
sampleEdge[][51]=255
NewImage sampleEdge
ImageEdgeDetection/N/S=1 Marr, sampleEdge
Duplicate/O M_ImageEdges s2
NewImage s2
ImageEdgeDetection/M=1/S=3 Canny, sampleEdge
Duplicate/O M_ImageEdges s3
NewImage s3
```

Original **Marr** **Canny**



平滑化設定を 1 に設定した場合、マール検出器はエッジを完全に検出できないことに注意してください。
また、Canny 検出器では平滑化が増加するにつれて、エッジの位置が真のエッジから離れていきます。