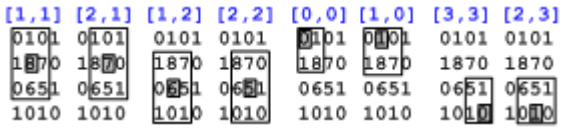
Image Processing: Filters

Lineare Tiefpassfilter

1) ungewichtete Mittelwertfilter = Averaging Filters

Einfachste und populärste Form = 3x3 Filter: ersetzt jedes Pixel durch den Mittelwert seiner 8er Nachbarschaft:
Zu jedem Grauwert die 8 Nachbargrauwerte addieren und Summe durch 9 teilen und runden.
Bildrandbehandlung: In den 0_{ten} und den letzten Zeilen/Spalten nur die Nachbarn addieren,
die existieren und durch eine entsprechend niedere Zahl teilen.
Beispiel 3x3 ungewichtet mit Indexreihenfolge zuerst x, dann y:

```
      0101
old = 1870
      0651
      1010
```




```
new[1,1] = (0 + 1 + 0 + 1 + 8 + 7 + 0 + 6 + 5)/9 = 28/9 = 3.11 = rounded 3
new[2,1] = (1 + 0 + 1 + 8 + 7 + 0 + 6 + 5 + 1)/9 = 29/9 = 3.22 = rounded 3
new[1,2] = (1 + 8 + 7 + 0 + 6 + 5 + 1 + 0 + 1)/9 = 29/9 = 3.22 = rounded 3
new[2,2] = (8 + 7 + 0 + 6 + 5 + 1 + 0 + 1 + 0)/9 = 28/9 = 3.11 = rounded 3
new[0,0] = (               0 + 1 +     1 + 8)/4 = 10/4 = 2.50 = rounded 3
new[1,0] = (           0 + 1 + 0 + 1 + 8 + 7)/6 = 17/6 = 2.83 = rounded 3
new[3,3] = (5 + 1 +    1 + 0               )/4 =  7/4 = 1.75 = rounded 2
new[2,3] = (6 + 5 + 1 + 0 + 1 + 0          )/6 = 13/6 = 2.16 = rounded 2
etc.
```

```
      3333
new = 3333
      3332
      2222
```

Die Grauwertunterschiede sind fast verschwunden,
new ist fast glatt geworden.

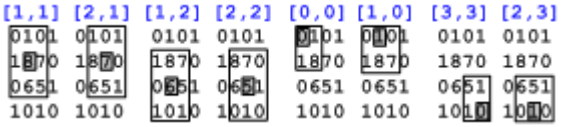
siehe auch: **Kovalevsky/ImProc/L01_Noise/Noise_e.htm**

C# Code: **Simple averaging filter.htm** **Simple averaging filter.pdf** **Demo: Simple averaging filter.exe**

2) gewichteter Mittelwertfilter

dämpft die Filterwirkung durch Vergabe eines Gewichts > 1 an das Mittenpixel:
Gegen seine 8 Nachbarn wird jedes Mittelpixel durch einen Faktor > 1 "gestärkt" und es wird durch einen entsprechend höheren Divisor geteilt.
Beispiel 3x3 mit Mittengewicht = 8,
was bedeutet, dass jedes Pixel soviel Stimmrecht hat, wie seine 8 Nachbarn zusammen:

```
      0101
old = 1870
      0651
      1010
```




```
new[1,1] = (0 + 1 + 0 + 1 + 8*8 + 7 + 0 + 6 + 5)/16 = 84/16 = 5.25 = rounded 5
new[2,1] = (1 + 0 + 1 + 8 + 7*8 + 0 + 6 + 5 + 1)/16 = 77/16 = 4.81 = rounded 5
new[1,2] = (1 + 8 + 7 + 0 + 6*8 + 5 + 1 + 0 + 1)/16 = 69/16 = 4.31 = rounded 4
new[2,2] = (8 + 7 + 0 + 6 + 5*8 + 1 + 0 + 1 + 0)/16 = 60/16 = 3.75 = rounded 4
new[0,0] = (               0*8 + 1 +     1 + 8)/11 = 10/11 = 0.91 = rounded 1
new[1,0] = (           0 + 1*8 + 0 + 1 + 8 + 7)/13 = 24/13 = 1.85 = rounded 2
new[3,3] = (5 + 1 +    1 + 0*8                )/11 =  7/11 = 0.63 = rounded 1
new[2,3] = (6 + 5 + 1 + 0 + 1*8 + 0           )/13 = 20/13 = 1.54 = rounded 2
etc.
```

```
      1331
new = 2551
      1442
      1121
```

Die Grauwertunterschiede sind noch deutlich,
new ist weit weniger unscharf als beim ungewichteten Mittelwertfilter.

3) schneller Mittelwertfilter = Fast Averaging Filter

Mit Hilfe eines zusätzlichen temporären Hilfsbildes `H[old.Height,old.Width]` kann man die Zahl der Additionen pro Pixel von `N*M` auf `2*M+2*N` drücken und damit den Mittelwertfilter enorm beschleunigen.
(Bei Mittengewichtung = Fog-Filter braucht man dazu noch ein zweites temporäres Hilfsbild `V[old.Height,old.Width]`.)
Algorithmus ohne Mittengewicht:

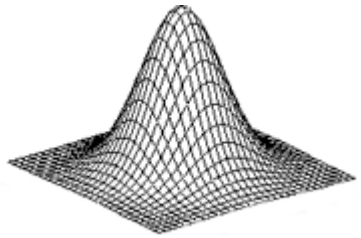
1. Schritt: Am linken Bildrand in jeder Zeile die ersten `M` Pixel von `old[y,x]` addieren und Summe nach `H[y,M/2]` speichern.
2. Schritt: Jede Summe ein `x` nach rechts schieben: `H[y,x+1] = H[y,x]-old[y,xleft]+old[y,xright]`, bis man am rechten Bildrand angekommen ist.
3. Schritt: Am oberen Bildrand in jeder Spalte die ersten `N` Pixel von `H[y,x]` addieren und Summe `sum` dividieren, runden und ausgeben: `new[y,x] = round( sum/(M*N) )`..
4. Schritt: Jede Summe ein `y` nach unten schieben: `sum = sum-H[yupper,x]+H[ylower,x]` und Summe `sum` dividieren, runden und ausgeben: `new[y,x] = round( sum/(M*N) )`, bis man am unteren Bildrand angekommen ist.

Algorithmus mit Mittengewicht (erzeugt Nebel = Fog) mit zweitem Hilfsbild `V[y,x]`:

1. Schritt: wie ohne Mittengewicht.
2. Schritt: wie ohne Mittengewicht.
3. Schritt: Am oberen Bildrand in jeder Spalte die ersten `N` Pixel von `H[y,x]` addieren und Summe nach `V[y,N/2]` speichern.
4. Schritt: Jede Summe ein `y` nach unten schieben: `V[y+1,x] = V[y,x]-H[yupper,x]+H[ylower,x]`, bis man am unteren Bildrand angekommen ist.
5. Schritt: Zu jedem `V[y,x]` das Produkt `weight*old[x,y]` addieren.
6. Schritt: Hilfsbild `V[y,x]` dividieren, runden und ausgeben: `new[y,x] = round( V[y,x]/(M*N+weight) )`.

C# Code: **Fast averaging filter.htm** **Fast averaging filter.pdf** **Demo: Fast averaging filter.exe**
C# Code: **Fast fog filter.htm** **Fast fog filter.pdf** **Demo: Fast fog filter.exe**

4) Gaußfilter = Gauss Filter



Beim Mittelwertfilter haben alle Nachbarn gleiches Stimmrecht unabhängig von ihrer Distanz zum Mittelpixel. Das ist unklug, denn nahen Nachbarn sollte man mehr Gehör schenken als entfernten. Man könnte den Kehrwert des Abstandes als Gewicht benutzen aber besser ist es, diesen Abstand `d` nicht linear, sondern mit Hilfe einer rotationssymmetrischen Gaußschen 3D-Glocke in Stimmgewichte umzurechnen (→ **Normalverteilung**):
`kernel[yy,xx] = e^-d*d/a`, wobei `e=2.718..` die Eulersche Zahl,
`a` eine Konstante und `d` der variable Abstand von der Mitte der Glocke ist.
Diese Glocke hat die Höhe `weight[N/2,N/2] = 1.0 = 100%`.
Der Divisor `a` im Exponenten bestimmt die Steilheit der Flanken der Glocke.
Man wählt die Steilheit am besten so, dass auch die entferntesten Nachbarn in den 4 Ecken des Kernels noch 0.01 = 1% Stimmrecht bekommen
(Sonderfall: bei 3x3-Kernels müssen sie mehr bekommen ≈ 16%).

Vor der Filterung mit einem `N*N`-Gaußfilter muss man somit erst einen Kernel mit passendem `a` berechnen, so dass in der größten Entfernung vom Kernelmittelpunkt d_{max} `= √2*N/2` = halbe Diagonale ein Funktionswert von `0.01` entsteht.
1. Schritt: Berechne `a` aus d_{max} und dem Sollwert `0.01`: `a = -2*(N/2)*(N/2) / Math.Log( 0.01 );`
2. Schritt: Berechne die Kernelmatrix `kernel[yy,xx]` und deren Gewichtssumme `sum_kernel`:

```
for ( yy=0; yy < N; yy++ )
  for ( xx=0; xx < N; xx++ )
  { double d = Math.Sqrt( (xx-N/2)*(xx-N/2) + (yy-N/2)*(yy-N/2) );
    sum_kernel += kernel[yy,xx] = (float)( Math.Exp( - d*d / a ) );
  }
```

3. Schritt: Faltung = Convolution = 4 geschachtelte `for`-Schleifen (siehe unten).

Fast Gauss Filter: Die Statistik lehrt, dass die Hintereinanderausführung von Faltungen mit einem kleinen gleichverteilten Kernel konvergiert zu einer Faltung mit einem großen normalverteilten Kernel = Gaußfilter. Diese Tatsache lässt sich leicht verifizieren: Eine Faltung mit einem `N*N`-Gauß-Filter ergibt ein fast identisches Ergebnis wie 3 hintereinander ausgeführten Faltungen mit 3 identischen `(N/2)*(N/2)` ungewichteten Mittelwertfiltern. Dieser Weg spart Rechenzeit, wenn 1) `N≥9` und 2) der Fast Averaging Filter benutzt wird und ist bekannt unter dem Namen Fast Gauss Filter.

C# Code: **Simple Gauss filter.htm** **Simple Gauss filter.pdf** **Demo: Simple Gauss filter.exe**
C# Code: **Fast Gauss filter.htm** **Fast Gauss filter.pdf** **Demo: Fast Gauss filter.exe**

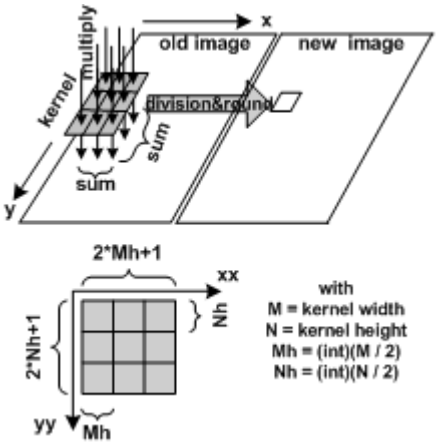
Zweck der Tiefpassfilter:
1) Rauschunterdrückung
2) Erzeugung künstlicher Leerbilder zur Shadingkorrektur
Um die Verwaschung = Blurring zu erhöhen, kann man:
1) Filtergröße auf 5x5, 7x7, 25x25 etc. erhöhen.
(Aus Symmetriegründen sind Filter meistens ungeradzahlig und quadratisch.)
2) Filter kaskadieren = mehrmals hintereinander anwenden.

Bildrandbehandlung ist kompliziert oder zeitaufwendig. Es gibt vier Strategien:
1) Die Bildränder ignorieren. Damit schrumpft das Ergebnisbild → schwarzer Rand.
2) Zuerst 1), dann die ersten und letzten Zeilen/Spalten von 1) in den Außenrand kopieren
→ von Prof. Miszalok empfohlen.
3) Zur Laufzeit die Filterkerne am Rand asymmetrisch verkleinern und die Division anpassen
→ langsam, Filterwirkung nimmt am Rand ab.
4) 8 Sonderfallbehandlungen codieren für jeden der 4 Ränder plus jede der 4 Ecken → für Perfektionisten.

Lineare Hochpassfilter

Linearer Hochpassfilter enthalten positive und negative Gewichte.
Sie ziehen homogene Flächen (meist auf 0) herunter, egal, ob sie hell oder dunkel sind und verstärkt die Grauwertdifferenzen = Anhebung der Kanten = Kantenfilter.
Einfachster Hochpassfilter = **Laplace Filter** enthält die Gewichte +1 und -1.
Er ist benannt nach dem Mathematiker Pierre Laplace um 1800, Paris.
Rechenvorschrift: Man berechnet für jedes Pixel die vier Grauwertdifferenzen zu seinen vier 4er Nachbarn und schreibe das Maximum nach `new`. Man verwerfe die 3 restlichen Differenzen.
Vorteile:
1) Bevorzugt steile Grauwertdifferenzen
2) Setzt alle homogenen Bildareale auf Null
3) einfach und schnell
Nachteil: Verstärkt das Rauschen
Andere häufig benutzte lineare Hochpassfilter sind benannt nach ihren Erfindern:
Sobel-, Prewitt-, Kirsch-Filter siehe unten.

Faltung = Convolution



Faltung = Convolution = lineare Filterung ist eine mathematische Formulierung für alle Arten linearer Filterung: Tiefpässe, Hochpässe und Mischungen zwischen beiden = Bandpässe.
Die Idee besteht darin, die Filtergewichte als kleine Matrix (meist quadratisch mit ungerader Spalten- und Zeilenzahl) namens Kern = engl. Kernel zu formulieren.
Die Rechenvorschriften für Filter werden dann zu einer Art gleitender Matrixmultiplikation mit Addition der Produkte und abschließender Normierung und Rundung auf eine ganze Zahl 0 ... 255.
Vorteile:
1) Alle Arten und Größen linearer Filter haben nur noch eine einzige Rechenvorschrift.
2) schnell durch Faltungs-ASICs = Application Specific Integrated Circuits speziell für Faltungen.
Nachteile:
1) Übertriebener Aufwand bei einfachen Kernels, die als Gewichte nur Einsen kennen.
2) Rechenvorschrift nicht anwendbar am Bildrand, wo die Kernelmatrix über das Bild hinausragt.

Rechenvorschrift: Lokal gleitende Multiplikation und Flächenintegral einer Bildmatrix `old[y,x]` mit einer (im Vergleich zur Bildmatrix kleinen) Filtermatrix `kernel[yy,xx]` mit zwei Indizes `0 <= xx <= 2*Mh` und `0 <= yy <= 2*Nh`.
Anschaulich ist der Kernel eine kleine Matrix `kernel[yy,xx]` auf durchsichtiger Folie, die über eine große Bildmatrix `old[y,x]` gelegt und durch zwei äußere `for`-Schleifen spalten-/zeilenweise weitergeschoben wird.
Schritt 1: Jede einzelne Zahl des Kernels wird mit dem darunter liegenden Grauwert von `old[y,x]` multipliziert.
Schritt 2: Danach werden alle Produkte zu einer Summe addiert.
Schritt 3: Die Summe wird durch einen Divisor `divisor ≥ 1` geteilt, gerundet und, falls negativ, mit einem `offset` in den Wertebereich zwischen 0 und 255 verschoben (notwendig bei eventuell negativen Summen).
Schritt 4: Das so erhaltene Ergebnis kommt in ein Ausgabebild `new` an die Stelle `[y,x]`.

Der Bildrand, wo die Folie `kernel[yy,xx]` über den Bildrand `old[y,x]` hinausragt bleibt undefiniert.

Eine vollständige Faltung mit Farbbehandlung und Randergänzung finden Sie unter **Simple Gauss filter.htm**.

Klassische Faltungsformel in Informatiker-Schreibweise::

```
const int Mh = M/2, Nh = N/2, offset = ???; //M*N = kernel size, offset mostly = 0
float sum; const float divisor = ?.???; //divisor = sum of all kernel values
(mostly)
for ( int y=Nh; y < old.Height-Nh; y++ )
  for ( int x=Mh; x < old.Width-Mh; x++ )
  { sum = 0.0f;
    for ( int yy=-Nh; yy <= Nh; yy++ )
      for ( int xx=-Mh; xx <= Mh; xx++ )
        sum += kernel[yy+Nh,xx+Mh] * old[y+yy,x+xx];
    new[y,x] = offset + Convert.ToByte( sum / divisor );
  }
```

Häufige und berühmte Faltungskerne

1) ungewichtete Mittelwertfilter haben quadratische Abmessungen und ungeradzahlige Spalten/Zeilenzahl 3x3, 5x5 ... 25x25.
Man setzt alle Werte von `kernel[yy,xx] = 1`, `offset=0`, `divisor=1f/(Summe des Faltungskerns)`.
Ungewichtete Mittelwertfilter sind algorithmisch schnell, da alle Multiplikanten `kernel[yy,xx]` gleich `1` sind und deshalb die Multiplikationen überhaupt entfallen können.

2) gewichtete Mittelwertfilter reduzieren die starke Wirkung der ungewichteten Mittelwertfilter dadurch, dass in die Mitte von `kernel[yy,xx]` hohe Multiplikatoren und zu den Rändern hin niedrige Multiplikatoren gesetzt werden.
Die Normierung ist `offset = 0` und `divisor = 1f/(Summe aller Multiplikatoren)`.

$$k1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \qquad k2 = \begin{bmatrix} 1 & \sqrt{2} & 1 \\ \sqrt{2} & 4 & \sqrt{2} \\ 1 & \sqrt{2} & 1 \end{bmatrix} \qquad k3 = \begin{bmatrix} 1 & 2 & 3 & 2 & 1 \\ 2 & 4 & 6 & 4 & 2 \\ 3 & 6 & 16 & 6 & 3 \\ 2 & 4 & 6 & 4 & 1 \\ 1 & 2 & 3 & 2 & 1 \end{bmatrix}$$

Sonderfall: Steigt das Mittengewicht über alle Maßen an, dann verliert der Filter jede Wirkung:

$$k4 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 999 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

3) Kantenfilter sind lineare Hochpassfilter. Beispiele:

```
     +1 +2 +1       +1 +1 +1        +5 +5 +5
k1 =  0  0  0   k2 = 0  0  0   k3 = -3 -3 -3
     -1 -2 -1       -1 -1 -1        -3 -3 -3
```

`k1`, `k2` und `k3` liefern hohe positive Grauwerte an horizontalen Kanten zwischen einem hellen Gebiet oben von einem dunklen unten und hohe negative Grauwerte bei dunkel oben und hell unten.

4) Kombinationen von Kantenfiltern führen mehrere verschiedene Kantenfilter nacheinander aus und kombinieren deren Ergebnisse zu einem einzigen Ausgabebild.
Wichtigste Untergruppe: **Gradientenfilter** haben 2 symmetrische Filterkerne V1 und V2 mit je 9 Werten.
Die 2 Kerne führen zu 2 Ergebnissen `s1` und `s2` pro Pixel. Man kann `(s1,s2)` als Vektor in Richtung des stärksten Gefälles auffassen = Gradient.
Ins Ausgabebild schreibt man `sqrt(s1*s1 + s2*s2)` = Betrag des Gradienten.
Da der Gradient immer positiv ist sind die Ergebnisse oben-unten- und links-rechts-symmetrisch (anders als bei einfachen Kantenfiltern). Beispiele:

```
              +1 +2 +1       +1  0 -1
Sobel:   k1 =  0  0  0  k2 = +2  0 -2  //populärster Gradientenfilter
              -1 -2 -1       +1  0 -1
              +1 +1 +1       +1  0 -1
Prewitt: k1 =  0  0  0  k2 = +1  0 -1  //etwas schwächer als Sobel
              -1 -1 -1       +1  0 -1
              +5 +5 +5       +5 -3 -3
Kirsch:  k1 = -3 -3 -3  k2 = +5 -3 -3  //etwas stärker als Sobel
              -3 -3 -3       +5 -3 -3
```

Sie finden Bauanleitungen zu diesem Thema unter:
../../C_IPCis/C3_Filter/CIPCisFilter_d.htm,
../../C_IPCis/C4_Lowpass/CIPCisLowpass_d.htm,
../../C_IPCis/C5_Convolution/CIPCisConvolution_d.htm
und Sie können lauffähige EXEs downloaden:
../../C_IPCis/C3_Filter/CIPCisFilter.exe,
../../C_IPCis/C4_Lowpass/CIPCisLowpass.exe,
../../C_IPCis/C5_Convolution/CIPCisConvolution.exe.

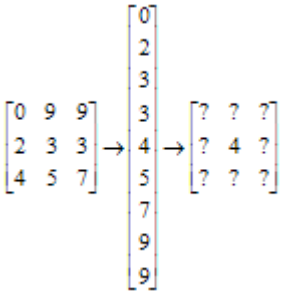
5) Bandpassfilter haben gleichzeitig Tiefpasseigenschaften (in der Mitte) und Hochpasseigenschaften (am Rand). Ihre Abmessungen (meistens größer als 9x9) und Gewichte sind auf spezielle Radaranlagen, Röntgen-Tomographen, Kernspintomographen sorgfältig angepasst.
Wichtigste Untergruppe: `Mexikanischer-Hut-Filter` hat rotationssymmetrisch hohe positive Werte in der Mitte von `kernel[yy,xx]`, die steil zu negativen Werten abfallen, um am Filterrand wieder zu niedrigen positiven Werten anzusteigen. Berühmt sind der 9x9 Filter von Ramachandran und Lakshminarayanan und der 9x9 Filter von Shepp und Logan. Beide Filter werden in der Computertomographie eingesetzt.

Nichtlineare Filter

Nichtlineare Filter heißen alle Filter, die **nicht** der Faltungsformel gehorchen.
Nichtlineare Filter besitzen auch ein über dem Originalbild `old` gleitendes rechteckiges Fenster mit ungeradzahliger Spalten- und Zeilenzahl, es gibt aber keinen Kernel `kernel[yy,xx]`.
Dieser wird ersetzt durch eine oder mehrere Rechenvorschriften,
die aus der Menge der überdeckten Grauwerte einen neuen Grauwert `new[y,x]` errechnen.

1) Medianfilter:

Populärster nichtlinearer Tiefpassfilter:
Er sortiert alle vom gleitenden `NxN`-Fenster `-N/2≤yy,xx≤+N/2` überdeckten Grauwerte `old[y+yy,x+xx]` in einen linearen Hilfs-Array der Länge `N*N` in aufsteigender Reihenfolge und nimmt den in der Mitte der Reihe stehenden Grauwert als Ergebnis `new[y,x]`.
Vorteil: Rauschunterdrückung ohne Verwaschung.
Nachteile:
1) Zerstörung von schmalen Strukturen und an Ecken.
2) Sortieren ist langsam.



C# Code: **Median filter.htm** **Median filter.pdf** **Demo: Median filter.exe**
Beispiel rechts: Ein Rechenschritt eines 3x3 Medianfilters

2) Sigmafilter:
ist ein selektiver Mittelwertfilter. Er mittelt nur Grauwerte, die sich höchstens um eine Spanne `sigma=σ` vom Grauwert des Zentralpixels unterscheiden und ignoriert die Abweichler.
Der Algorithmus ist fast identisch mit dem des normalen Mittelwertfilters.
Nur in der innersten `for`-Schleife muss man die Summierung folgendermaßen ändern:

```
if ( Math.Abs(old[y,x]-old[y+yy,x+xx]) <= sigma ) { sum += old[y+yy,x+xx]; divisor++; }
```

Beispiel `3x3`-Filter mit `σ=3` und Indexreihenfolge `y,x`:

```
2189
1278 → new[1,1]=(2+1+1+2+1+3)/6=10/6≈2;  new[1,2]=(8+9+7+8+9+7)/6=48/6=8;
1397
```

Zusammenfassung: Sehr wichtiger "intelligenter" Filter:
Beidseits der Grauwertkante werden die Grauwerte geglättet ohne die Kante abzuflachen.
Vorteil: Rauschunterdrückung ohne Verwaschung.
Nachteile:
1) Wenn σ zu niedrig, dann findet fast keine Mittelung statt,
wenn σ zu hoch, dann entsteht die gleiche Verwaschung wie bei den linearen Tiefpassfiltern.
2) Langsamer als der normale Mittelwertfilter. Fast Averaging gar nicht möglich.
3) Keine Gewichtung des Mittenpixels möglich.

C# Code: **Sigma filter.htm** **Sigma filter.pdf** **Demo: Sigma filter.exe**