Computational Vision
U. Minn. Psy 5036

Linear systems, Convolutions and Optical blur

Linear Systems

Linear system models are important to vision for modeling: optical, retinal sampling, and neural transformations of image data. The basic requirements for a system **T** to be linear are:

T(a+b) = T(a) + T(b)
T(λa) = λT(a)

For continuous models, the so-called *linear superposition operator* for an image transformation can be expressed as:

$$r(x,y) = \iint l(x',y')W(x,y;x',y')dx'dy'$$

Exercises

1. As an pencil and paper exercise, show that the above 2D linear superposition operator satisfies the above two criteria.

2. Use *Mathematica* to verify that matrix multiplication on vectors satisfies the two criteria for a linear sytem: superposition and homogeneity. The matrix **WW** is the transformation applied to vector inputs **aa** or **bb**.

```
In[544]:= aa = Table[a[i],{i,1,3}];
          bb = Table[b[i],{i,1,3}];
          WW = Table[W[i,j],{i,1,3},{j,1,3}];
```

Answer to 2--or at least one way of doing it

```
In[547]:= Simplify[WW.aa + WW.bb - WW.(aa + bb)]
          Simplify[WW.(k aa) - k WW.aa]

Out[547]= {0, 0, 0}

Out[548]= {0, 0, 0}
```

Convolutions

Often we can assume that the transformation is space invariant. This is a reasonable assumption if we restrict ourselves to small optical patches (so-called isoplanatic patches), uniform retinal sampling, or subgroups of neural systems whose receptive fields subtend small regions of the visual field.

For a space invariant system, W(x,y; x',y') is equal to W(x-x',y-y') and the linear operation becomes a *continuous convolution*: from -infinity to +infinity :

$$r(x,y) = \iint l(x',y')W(x-x',y-y')dx'\,dy'$$

For computations, we usually model the system operation as a *discrete matrix* multiplication. Let **W** be the matrix, **r** and **l** are vectors:

r = Wl

2D convolution is used to model:

1) optical blur,
2) discrete retinal sampling by the photoreceptors, and
3) the linear part of "neural image" transformations by ganglion cell arrays, and in general populations of visual cortical cells
4) A pre-process for edge-detection, in models of color constancy, and motion detection.

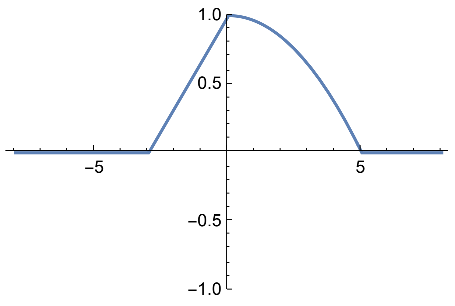
So it is worth spending time to understand it.

Continuous 1-D convolution

```
In[549]:= a = 3; b = 5;

Clear[filter];  (* When you play with building up definitions,  you should clear the function to make sure you don't have c
filter[x_]:= N[(1-Abs[x]/a)] /; x>-a && x <0
filter[x_]:= N[(1-x^2/b^2)] /; x < b && x>=0
filter[x_]:= 0.0 /; x<=-a || x >= b
```

```
In[557]:= Plot[filter[x], {x, -8, 8}, PlotRange → {-1, 1}]
```

Out[557]=


/; means "such that". Mathematica uses notation borrowed from the C programming language for logical operations such as:

OR => ||

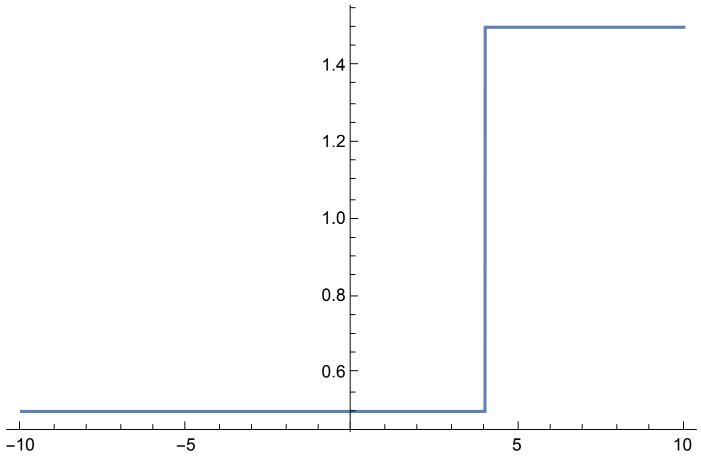
and

AND => &&

```
In[558]:= edge[x_] := N[If[x<4,1/2,1.5]];

In[559]:= Plot[edge[x], {x, -10, 10}]
```

Out[559]=



A 1-D convolution operation is written:

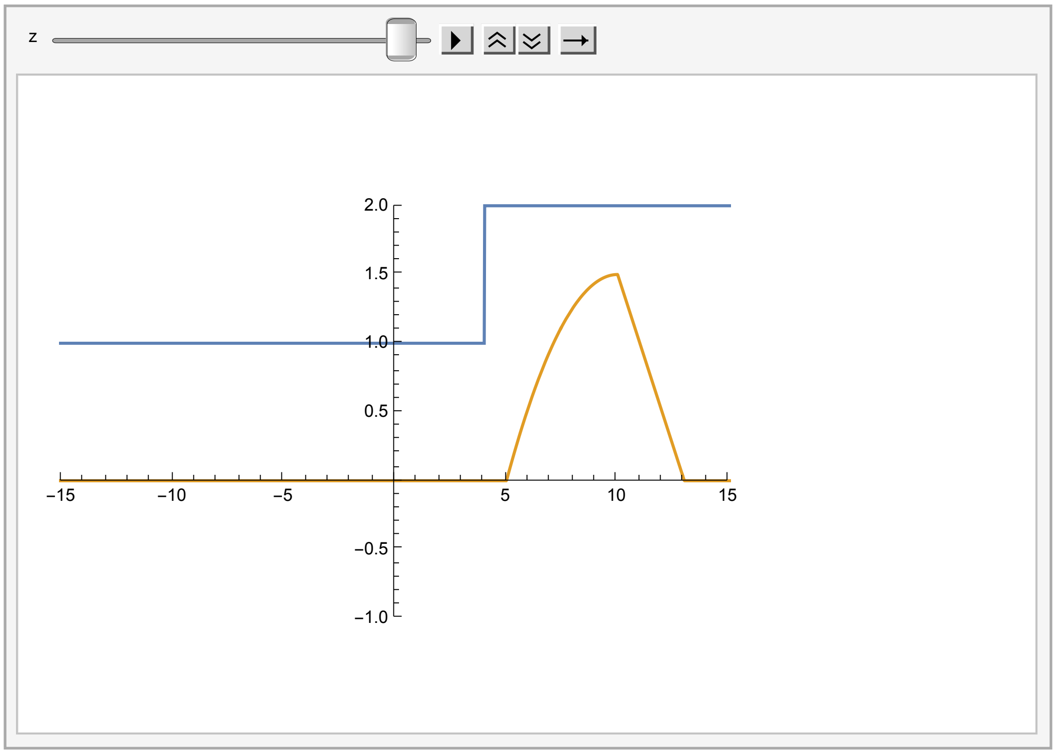
$$r(x) = \int_{-\infty}^{+\infty} l(x')\,w(x - x')\,dx'$$

$$r(x) = \int_{-\infty}^{+\infty} edge(z)\,filter\,(x - z)\,dz$$

It is useful to visualize this as the area underneath the curve formed by the product of edge and the left-right reversed filter as it slides along the x-axis.

In[560]:=
```
Animate[Plot[{edge[x] + .5, edge[x] filter[z - x]},
  {x, -30, 30}, PlotRange → {{-15, 15}, {-1, 2}}], {z, -10, 10, 0.5}]
```

Out[560]=

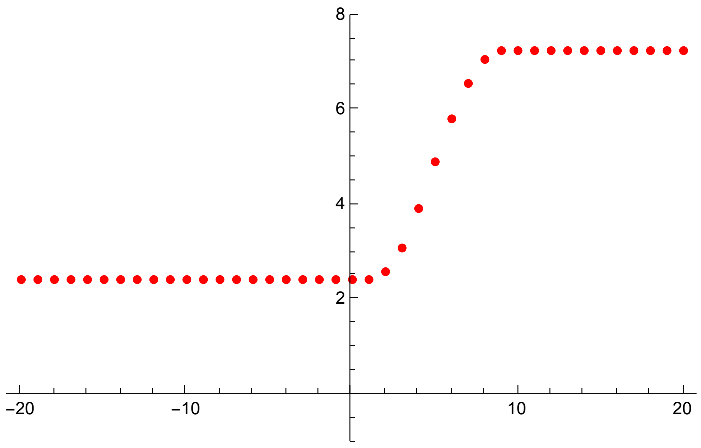


We can find the area under each of the above curves by using *Mathematica*'s built-in numerical integration capability, **NIntegrate[]**. It helps if we specify the range of necessary integration as precisely as possible. Inspection of the above curves shows that **{z-b,z+a}** is an appropriate range for **x**.

In[561]:=
```
r1 =
Table[{z,NIntegrate[edge[x] filter[z - x],
                  {x,z-b,z+a}]}, {z,-20,20,1}];
```

In[563]:=
```
continousconvg = ListPlot[r1, PlotRange → {-1, 8}, PlotStyle → {RGBColor[1, 0, 0]}]
```

Out[563]=

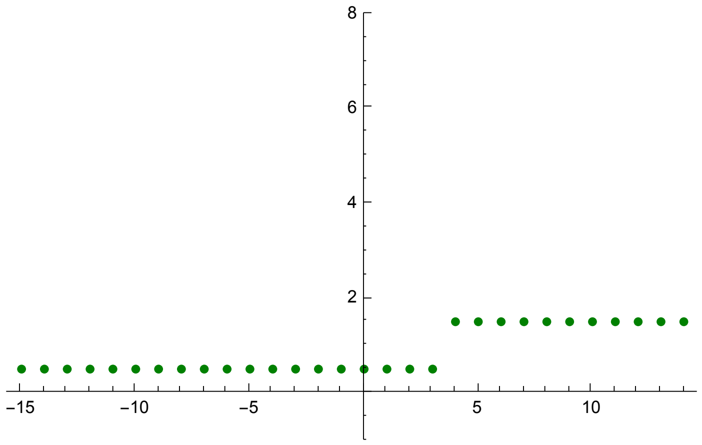


Discrete 1-D convolution as matrix multiplication

In this section, we will pass the 1-D "edge" through the filter again, but this time we will set it up as a discrete matrix operation. We'll apply a convolution matrix to a vector that represents a sharp edge, **I=edge**, at x = size/2.

```
In[569]:= size = 30; edge[x_] := N[If[x < 4, 1/2, 1.5`]];
          edgelist = Table[{x, edge[x]}, {x, -size/2, size/2 - 1}];
          edgevector = Transpose[edgelist][[2]]; edgelistg =
           ListPlot[edgelist, PlotRange → {-1, 8}, PlotStyle → {RGBColor[0, 0.5`, 0]}]
```

Out[569]=



Because the integration is finite, we have to make some decision about what to do at the boundaries. One choice is to assume that w and I are zero beyond the boundaries. Another is to assume that they are periodic with a shared common period, and we are just representing one period of the signal.

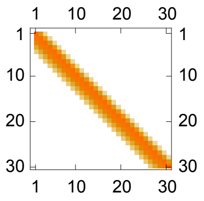
Now we construct a **size x size** matrix that will perform a convolution operation.

```
In[570]:= w    = Table[filter[i-j], {i,size},{j,size}];
```

A convenient way to view the matrix is to use **MatrixPlot[]**, () where you can see how each row is a shifted version of the preceding row.

```
In[571]:= MatrixPlot[w, ImageSize → Tiny]
```

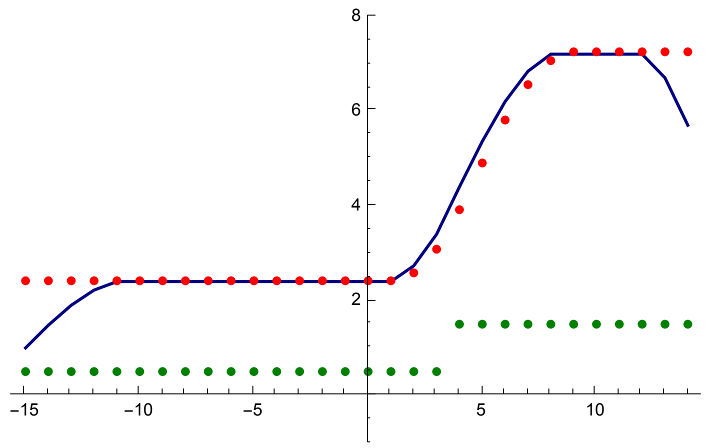
Out[571]=



Note that ArrayPlot[w], does almost the same thing, except that it reverses the order.

```
In[572]:= rvect = w.edgevector;
          r = Table[{i - size/2, rvect[[i + 1]]}, {i, 0, size - 1}]; matrixconvolg =
           ListPlot[r, Joined → True, PlotRange → {-1, 8}, PlotStyle → {RGBColor[0, 0, 0.5`]}];
```

In[573]:=
```
Show[{matrixconvolg, edgelistg, continousconvg}]
```

Out[573]=


The convolution matrix "blurs" out the edge and is in reasonable agreement with the approximation to the continous convolution. Note that the left and right boundary edges also got "blurred" by the matrix convolution.

Exercise: Blur the above edge with a line-spread function (LSF) that models diffraction-limited optics

Introduction to diffraction limited blur

Pin-hole optics are limited by diffraction. The image of a point though a circular aperture is an Airy pattern. We have to take care to define the function at x and y =0.

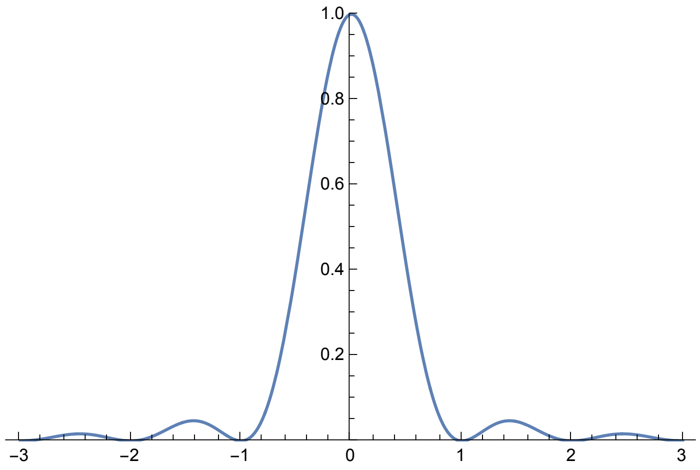
```
(*Cell inactive*)
Airy2D[x_,y_] :=
If[x==0 && y==0,1,
(2 BesselJ[1,Pi Sqrt[x^2+y^2]]/(Pi Sqrt[x^2+y^2]))^2]
Plot[Airy2D[x,0],{x,-3,3}];
```

For a single slit aperture, the "line-spread-function" for a diffraction limited system is a "sync" function squared:

In[574]:=
```
SincSq[x_]:= If[x==0,1.0,N[(Sin[Pi x]/(Pi x))^2]];
```

In[575]:=
```
Plot[SincSq[x], {x, -3, 3}, PlotRange → {0, 1}]
```

Out[575]=



Now as an exercise, construct a weight or filter matrix with the SincSq[] function, do a discrete convolution with the above edge, and plot up the results.

Convolutions and eigenvectors

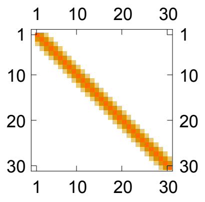
In[576]:=
```
size=30;
a = 3; b = 3;

Clear[filter];   (* When you play with building up definitions,  you should clear the function to make sure you don't have c
filter[x_]:= N[(1-Abs[x]/a)] /; x>-a && x <0
filter[x_]:= N[(1-Abs[x]/a)] /; x < b && x>=0
filter[x_]:= 0.0 /; x<=-a || x >= b
```

In[583]:=
```
ww  = Table[filter[N[i-j]], {i,size},{j,size}];
ee = Eigenvectors[ww];
```

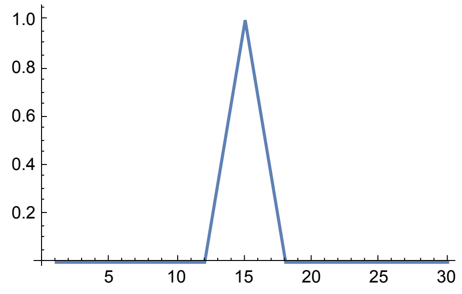
In[586]:=
```
MatrixPlot[ww, ImageSize → Tiny]
```

Out[586]=



```
In[587]:= ListPlot[ww[[15]], Joined → True]
```

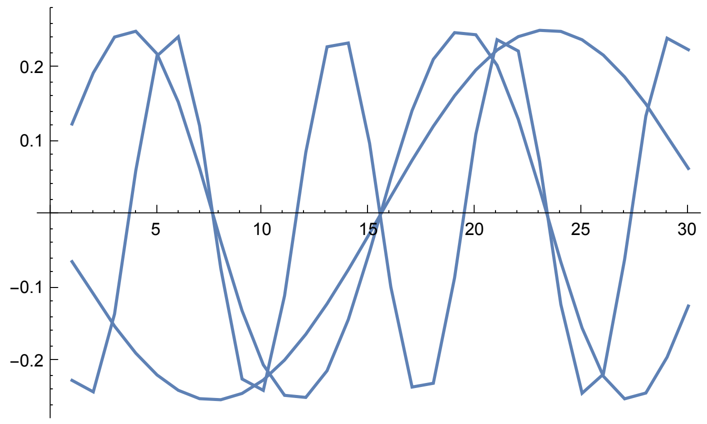
Out[587]=



Let's plot a few of the eigenvectors of w:

```
In[588]:= g1 = ListPlot[ee⟦2⟧, Joined → True];
g2 = ListPlot[ee⟦4⟧, Joined → True];
g3 = ListPlot[ee⟦8⟧, Joined → True];
Show[{g1, g2, g3}]
```

Out[589]=



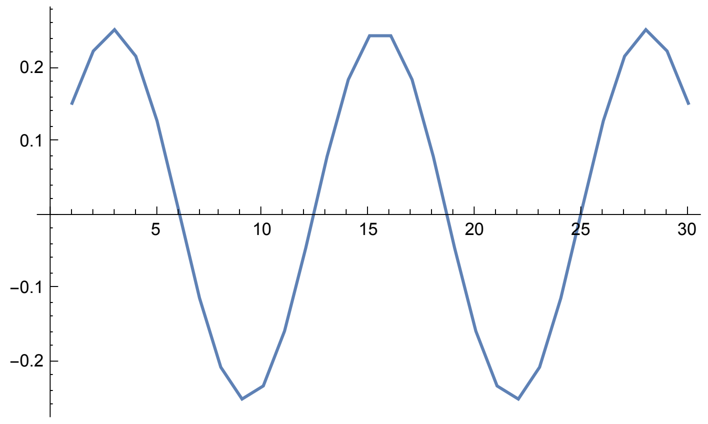
They sure look a lot like sinewaves, and in fact, they are pretty close. This is related to the fact that the sine-wave gratings keep the same form when imaged by the optics--even in the presence of aberrations. Why? Because the linear, shift-invariant model is a reasonable approximation.

Exercise

Verify that **ee[[5]]** is indeed an eigenvector of **ww** by showing that ww.ee[[5]] points in the same direction as e[[5]].

```
In[590]:= ListPlot[ee[[5]], Joined → True]
```

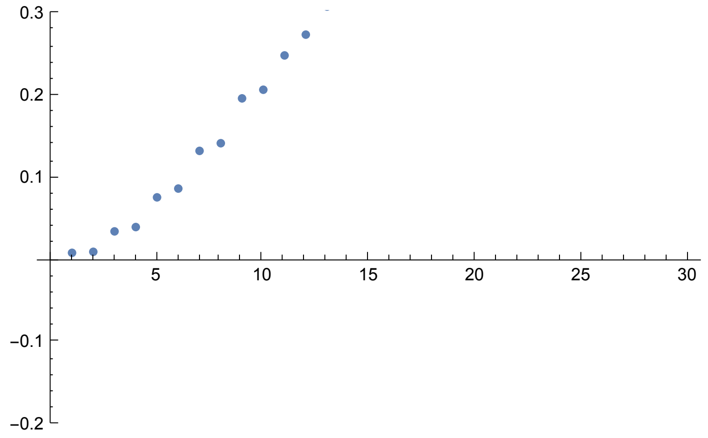
Out[590]=



Here is a plot of the sorted eigenvalues for **ww**:

```
In[591]:= lambda = Sort[Eigenvalues[ww]]; ListPlot[N[lambda], PlotRange → {-0.2`, 0.3`}]
```

Out[591]=



Later on we will use eigenvectors and eigenvalues in a completely different context when we discuss efficient ways of representing image information.

2D Convolution computation and the FFT (Fast Fourier Transform)

In this notebook, we've taken a preliminary look at linear filtering with simple examples in 1-D. Later, we will extend our techniques to efficiently handle 2-D images. Convolution can either be done in the space domain (as above), or in the Fourier domain. If we multiply the fourier transform of the image with the fourier transform of the filter (e.g. point spread function), and then take the inverse fourier transform of this product, we have the convolution of the image with the filter. Why bother? The main reason is that there are fast digital techniques, e.g. the Fast Fourier Transform or FFT, which make doing convolutions by multiplying in the fourier domain much more efficient.

Note: Convolutions vs. cross correlations

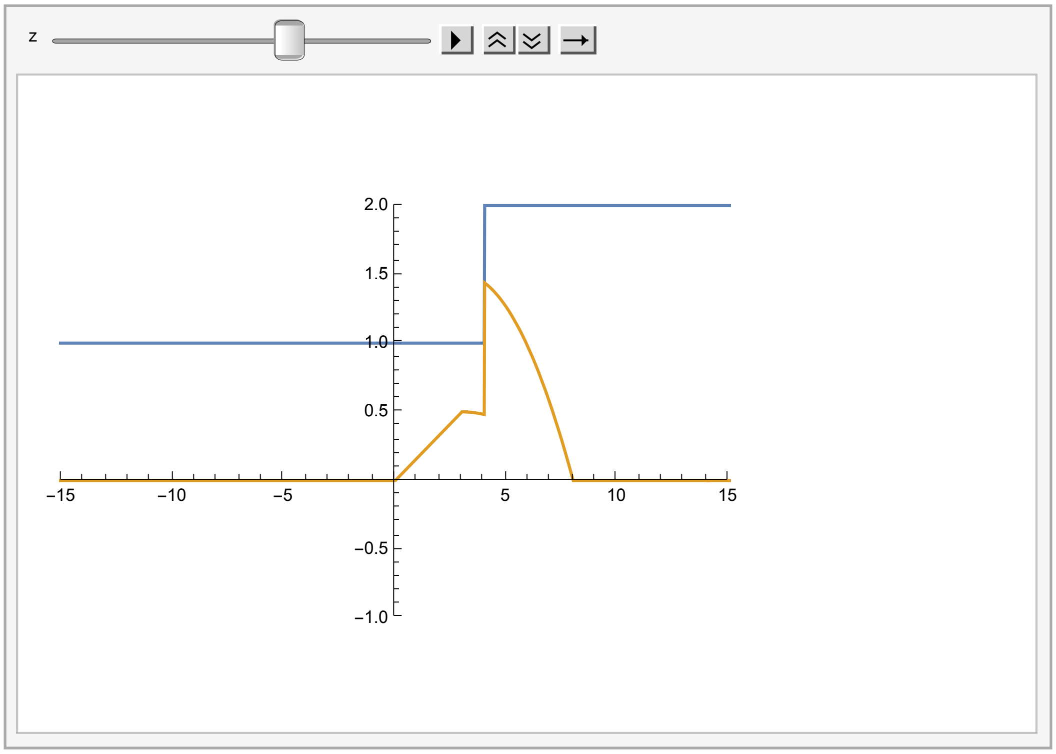
Convolution can be thought as two steps: 1) flip the filter about the vertical axis (i.e. x -> -x); 2) correlate the flipped filter with the signal. If we skip the flipping step, we just have correlation. The following illustrates the process of sliding the (unflipped) filter over the signal:

```
a = 3; b = 5;

Clear[filter];  (* When you play with building up definitions,  you should clear the function to make sure you don't have c
filter[x_]:= N[(1-Abs[x]/a)] /; x>-a && x <0
filter[x_]:= N[(1-x^2/b^2)] /; x < b && x>=0
filter[x_]:= 0.0 /; x<=-a || x >= b
```

```
In[601]:= Animate[Plot[{edge[x] + .5, edge[x] filter[x - z]},
            {x, -30, 30}, PlotRange → {{-15, 15}, {-1, 2}}], {z, -10, 10, 0.5}]
```

Out[601]=



So why bother flipping in the first place? First, it doesn't matter if the filter is symmetric. But the reason for flipping first is to make the operation *associative*. So if f, g, and h are filters that get sequentially applied to a signal **I**: f*h*g***I**, it doesn't matter how we group the operations. So f*(h*g)***I** = (f*h)*g***I**. This is very useful when we want to precompute a filter such as a ∇^2G operator for edge detection.