Introduction to Neural Networks
U. Minn. Psy 5038

Sculpting energy landscapes: interpolation and gradient descent

■ **Initialization**

```
<< Statistics`MultinormalDistribution`
```


Using energy and gradient descent to derive update rules

Earlier we studied how to set up the weights in a discrete response TLU network for the correspondence problem in stereopsis. The weights were determined by an analysis of the constraints needed to find a unique correspondence between the pixels in the left and right eyes. We didn't compute energy in that example, but pointed out that energy could play a useful role as an index to describe how well the network's state vector was moving towards the correct answer in state space. In particular, the energy function contains information about the stable points in state space.

In effect, we sculpted the energy landscape by hand-wiring the weights according to the constraints that were determined heuristically.

In the TIP examples, we reconstructed stored letters from partial information. In that case, the weights were determined by Hebbian learning. So the energy landscape was sculpted by state vectors to be stored.

But we can also do things the other way around. Rather than figuring out the weights for a Hopfield-style network that has a known relationship to an energy function, we first specify the energy function, and then figure out an update rule that will descend the energy landscape.

We followed an analogous strategy when we set up an error function in terms of weights, and then did gradient descent to find the weights that minimized the error to learn the weights. But one can also set up the analog to the error function (weights variable), that is an energy function of the state vector (weights fixed), and then use gradient descent to derive a rule to find minima of this energy function. From the point of view of neural networks, this update rule may look nothing like what neurons do. But it may be the best way to start--that is, by sculpting the energy function directly, not worrying about "weights", and then see what emerges in terms of an update rule.

We are going to follow this strategy in this notebook on a simple problem of interpolation. It will turn out that our update rule is the simplest neural model--a linear summer. However, this kind of analysis provides a starting point for more complicated energy functions with correspondingly non-linear update rules.

The energy function is sometimes referred to as a cost function or objective function.

In the second main part of this notebook (Deriving learning rules), we return to the problem of deriving learning rules from cost functions over weights using gradient descent. We apply it to the problem of self-organization where the goal is to learn a decorrelating set of weight vectors, which (unlike PCA) are not necessarily orthogonal.

Our main tool is gradient descent. Given an objective or energy function, there are often better tools for finding the minimum (e.g. see Hertz et al.). But gradient descent is a simple and intuitive starting point.

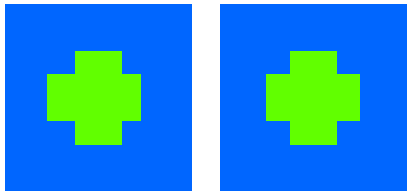
Interpolation problems in perception

A major theoretical problem in vision has to do with the fact that local changes in image intensities are usually ambiguous in natural images. A change in shading can mean an change in shape (or a change in illumination, e.g. a cast shadow). A change in image color can mean a change in the reflectivity of a surface (or a change in the illumination--two quite different causes). Changes of intensity of pixels in time provide information about surface structure, and the viewer's relation to that surface. In order to solve problems such as those above, researchers have studied special cases: shape-from-shading, reflectivity from color (color constancy), optic flow field from the flow of intensities, and more. One recurring theme in these problems is that the data available in the image does not fully constrain the estimate of the surface or surface properties one would like to compute.

Earlier we studied the random dot stereogram in which each image had many local features densely packed, but there was ambiguity in matching left eye pixels to those in the right. Here another stereo example. This time there are few features sparsely packed.

```
backwidth = 50; backheight = 50;
x0 = backwidth / 2; y0 = backheight / 2;
vwidth = backwidth / 4; vheight = backheight / 8;
vxoff = backwidth / 2; vyoff = backheight / 2;
hwidth = backwidth / 8; hheight = backheight / 4;
hxoff = backwidth / 2; hyoff = backheight / 2;
gleft = Show[
   Graphics[{Hue[.6], Rectangle[{0, 0}, {backwidth, backheight}], Hue[.27],
     Rectangle[{x0 - vwidth, y0 - vheight}, {x0 + vwidth, y0 + vheight}], Hue[
      .27], Rectangle[{x0 - hwidth, y0 - hheight}, {x0 + hwidth, y0 + hheight}]}],
   AspectRatio → Automatic, DisplayFunction → Identity];

gright = Show[Graphics[
    {Hue[.6], Rectangle[{0, 0}, {backwidth, backheight}], Hue[.27], Rectangle[
      {x0 - vwidth - 1, y0 - vheight}, {x0 + vwidth - 1, y0 + vheight}], Hue[.27],
     Rectangle[{x0 - hwidth, y0 - hheight}, {x0 + hwidth, y0 + hheight}]}],
   AspectRatio → Automatic, DisplayFunction → Identity];
Show[GraphicsArray[{gright, gleft}], DisplayFunction → $DisplayFunction];
```



Try a motion analog: Show[gleft,DisplayFunction→ $DisplayFunction]; Show[gright,DisplayFunction→ $DisplayFunction]; and then group the two pictures, and play as a slow movie

If you can cross your eyes, so that the left image is in the right eye, and the right image in the left, you will see a green horizontal bar floating out in front of a green vertical bar. So-called "free-fusing" isn't easy, but when you've got it, you should see a total of three green crosses. The one in the middle is the one in which the two images are fused by your brain--and this is the one we are talking about.

The interesting point here is that even though there is no local information in the image to support the percept of a horizontal occluding bar, observers still see an illusory completion. It looks a bit like the figure below, except that the color of the horizontal bar is changed here slightly just for illustration.

```
Show[Graphics[{Hue[.6], Rectangle[{0, 0}, {backwidth, backheight}], Hue[.42],
    Rectangle[{x0 - hwidth, y0 - hheight}, {x0 + hwidth, y0 + hheight}], Hue[.27],
    Rectangle[{x0 - vwidth, y0 - vheight}, {x0 + vwidth, y0 + vheight}]}],
  AspectRatio → Automatic];
```

The visual system seems to *interpolate* a surface between salient points--in this case the salient points are the vertical edge segments of the horizontal bar.

(Side note: The perceptual interpolation in the above example is not straightforward. In fact, one's first guess might be that observers should not see the horizontal bar as a plane, but rather they would interpolate a surface that on the left is close to the viewer, but then descends back towards the depth of the vertical bar, stays there, and then comes back towards the viewer on the right. Why people usually don't see this has been studied by: Nakayama, K., & Shimojo, S. (1992).)

We will study a simpler case of interpolation--namely filling in a line between feature points. We will do this by constructing an energy function first, and then calculating an update rule by gradient descent. But the same principles apply to computational models of stereopsis, shape-from-shading, optic flow and other problems in early visual processing.

Energy: Data and smoothness terms

There are two constraints that will guide the problem of sculpting an energy function to reconstruct a line (or surface):

1. Fidelity to the data; 2) Smoothness of the fit.

Suppose $\{d_i\}$ are the data points, where the i's come from a subset, D of the total domain over which our reconstructed function, f is defined. Fidelity to the data can be represented by an energy term that is big if the estimate of f is too far away from the data:

$$E_d = \sum_{i \in D} \left(f_i - d_i \right)^2$$

Smoothness can be represented by an energy cost in which near-by values of the estimate, f, are required to be close:

$$E_s = \sum_i \left(f_i - f_{i+1} \right)^2$$

We combine two constraints by adding:

$$E = E_d + E_s = \sum_{i \in D} \left(f_i - d_i \right)^2 + \lambda \sum_i \left(f_i - f$$

λ is a free parameter that allows us to control how much the smoothing should dominate the data or fidelity term. Note we assume that the smoothness term is independent of the data, and is equivalent to assuming a Bayesian prior in statistics (Poggio et al., 1988; Kersten et al., 1987).

If we wish to start off with some initial guess for the values of f, we can successively improve our estimate by sliding down the slope of E in the steepest direction. This says that the rate of change of f_i in time should be proportional to the negative slope of E in the direction of f_i

$$\frac{df_i}{dt} = -\frac{\partial E}{\partial f_i}$$

■ **Proof, in case you are still wondering:**

$$\Delta E = \nabla E \bullet d\vec{f} = \left| \nabla E \right| \left| d\vec{f} \right| \cos\theta$$

For step Δt, the biggest decrease in E is when cosθ = -1. In other words, when the vectors $\overrightarrow{\nabla E}$ and $\overrightarrow{df}$ are pointing in opposite directions. So if we make a change df in proportion to -∇E for each time step Δt, we will reduce the energy.

$$\overrightarrow{df} \propto -\nabla E$$

In the limit Δt→0, we can set:

$$\left(\frac{\partial E}{\partial f_1}, \frac{\partial E}{\partial f_2}, \ldots \right) = -\left(\frac{df_1}{dt}, \frac{df_2}{dt}, \ldots \right)$$

As mentioned above, we've encountered gradient descent when we calculated the derivative of the error function with respect to the weights.

Taking the derivative:

$$\frac{df_i}{dt} \propto -\left(f_i - d_i \right) - \lambda \left(2 f_i - f_{i+1} - f_{i-1} \right)$$

(We use *Mathematica* below to verify the pattern of the terms in the derivative.) Or in discrete time steps of dt,

$$f_i(t + dt) = f_i + dt \left\{ -2\left(f_i - d_i \right) - 2\lambda \left(2 f_i - f_{i+1} - f_{i-1} \right) \right\}$$

(Note: There is a slightly messy issue of book-keeping in that, we have to pay attention to when we are updating f's that don't have data support. We show one way to handle that below.)

With a little inspection, you can see that the value of f at time t+dt, is just the weighted sum of the values of f at time t, and the data, d. A weighted sum calculation is exactly what our linear model of a neuron does. So we've derived an update rule for our energy function that could be implemented with a standard neural model.

What is the weight matrix? Are the diagonals zero? Is the matrix symmetric?

It turns out that as long as the energy function is a quadratic function of the f's, the update rule will be linear. Why?

The power of this approach is that one can construct more complicated energy functions, and derive gradient descent update rules that are not linear but can be used to find solutions. For example, the world we see is not constructed from one smooth surface, but is better modeled as a set of surfaces separated by discontinuities. An energy function can be constructed that explicitly models these discontinuities and their effects on the interpolated values. The energy function in this case is no longer quadratic, and the update rule computes more than a weight sum. This usually brings the additional problem of local energy minima that we will study in the next lecture.

For examples of constructing a more complicated energy functions with discontinuities, see: Kersten, D. J. (1991) and Kersten and Madarsmi (1995).

Example: Reconstructing a smooth line, interpolation using smoothness

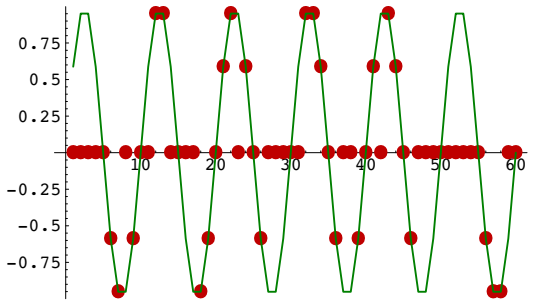
First-order smoothness

Suppose we have sampled a function at a discrete random set of points **xs**. Multiplying the sine function by the vector **xs** picks out the values at the sample points at each location of the vector where **xs** is one, and sets the others to zero.

```
size = 60;
xs = Table[Random[Integer,1], {i,1,size}];
data = Table[N[Sin[2 Pi (1/10) j] xs[[j]]],
{j, 1, size}];

g3 = ListPlot[Table[N[Sin[2 Pi (1/10) j]], {j, 1, size}],
        PlotJoined->True,DisplayFunction->Identity,
    PlotStyle->{RGBColor[0,.5,0]}];
g2 = ListPlot[data,PlotJoined->False,
PlotStyle->{RGBColor[.75,.0,0]}, Prolog-> AbsolutePointSize[5],
DisplayFunction->Identity];
```

```
Show[g2,g3,DisplayFunction->$DisplayFunction];
```



We would like to find a smooth function, **f[]**, approximation to the non-zero data points, given the assumption that we don't know what the underlying function actually is.

We have one constraint already--the fidelity constraint that requires that the function **f** should be close to the non-zero data,

d. We will use this to construct the "energy" term that measures how close they are in terms of the sum of the squared error.

We need another constraint--smoothness--to get the in-between points. There are many ways of doing this. If we had a priori knowledge that the underlying curve was periodic, we'd try fitting the data with some combination of sinusoids. Suppose we don't know this, but do have reason to believe that the underlying function is smooth in the sense of nearby points being close. As above, let's assume that the difference between nearby points should be small. That is, the sum of the squared errors, **f[i+1] - f[i]**, gives us the second part of our energy function.

Let's make up a small 8 element energy vector:

```
energyvector =
Table[(f[i+1] - f[i])^2 + s[i] (d[i] - f[i])^2,
            {i,1,8}];
```

```
energy = Sum[energyvector[[j]],{j,1,8}];
```

The **s[i]** term is the "filter" (the same as **xs** above) that only includes data points in the data part of the energy function. It is zero for i's where there are no data, and one for the points where there are data.

We would like to find the **f[]** that makes this energy a minimum. We can do this by calculating the derivative of the energy with respect to each component of **f**, and moving the state vector in a direction to minimize the energy--i.e. in the direction of the negative of the gradient.

It can be messy to keep track of all the indices in these derivatives, so let's let *Mathematica* calculate the derivative for f[3]. From this we can see the pattern for any index.

```
D[energy, f[3]]
```

```
2 (-f[2] + f[3]) - 2 (-f[3] + f[4]) - 2 (d[3] - f[3]) s[3]
```

```
Simplify[%]
```

```
2 (-f[2] + 2 f[3] - f[4] - d[3] s[3] + f[3] s[3])
```

Now we could generate the full set of derivatives, set them equal to zero and solve for **f**, using standard linear algebra to solve a set of linear equations. This will work because the energy function is quadratic in elements of **f**, and thus the derivatives are linear in **f**.The interpolation function is then a matrix operation on the data. Life won't always be that easy, and the situation often arises in which the energy function is not quadratic.

So the alternative, which can work in the non-linear case, is to use what we introduced above--gradient descent. We can do this by expressing the derivative in terms of two matrix operations: One on the function to be estimated, f, and one on the data.

Let's set up these two matrices, **Tm** and **Sm** such that the gradient of the energy is equal to:

Tm . f - Sm . f. **Sm** will be our filter to exclude non-data points. **Tm** will express the "smoothness" constraint.

```
Sm = DiagonalMatrix[xs];
Tm = Table[0,{i,1,size},{j,1,size}];
For[i=1,i<=size,i++,Tm[[i,i]] = xs[[i]]+2];
For[i=1,i<size,i++, Tm[[i+1,i]] = -1];
For[i=1,i<size,i++, Tm[[i,i+1]] = -1];
```

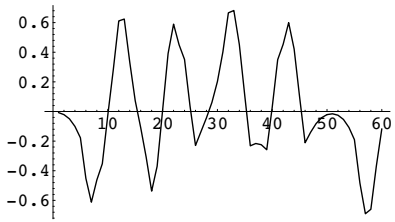
```
dt = 0.1;
Tf[f1_] := f1 - dt (Tm.f1 - Sm.data);
```

We will initialize the state vector to zero, and then run the network for 30 iterations:

```
f = Table[0,{i,1,size}];
result = Nest[Tf,f,30];
```

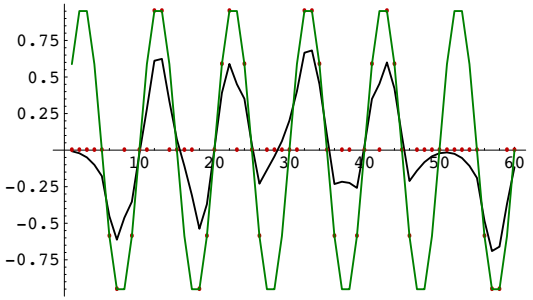
Here is our smoothed function:

```
g1 = ListPlot[result,PlotJoined->True];
```



You can see below that the function was not interpolated--the fit doesn't not pass through the data points. If we wanted more fidelity to the data, we could control this by increasing the weight given to the data part of the energy term relative to the smoothness part.


```
Show[{g1,g2,g3}];
```



Optional exercise:

Decrease the parameter controlling smoothness to see if you get better fidelity.

Sculpting for interpolation using second order smoothness constraints

This section elaborates the energy function (and weight matrix) to require that both first and second order differences be small.

```
Clear[energy,energyvector,f,d,s,data];
```

```
energyvector =
Table[(f[i+1] - f[i])^2 + (f[i+2] - 2 f[i] + f[i+1])^2 + s[i] (d[i] -
f[i])^2,{i,1,8}];
```

```
energy = Sum[energyvector[[j]],{j,1,8}];
```

By taking the derivative of the energy with respect to one of the interpolation depths, say f[3], we can see the pattern of the weights for the gradient descent update rule:

```
D[energy, f[3]]
```

```
2 (-f[2] + f[3]) + 2 (-2 f[1] + f[2] + f[3]) - 2 (-f[3] + f[4]) +
  2 (-2 f[2] + f[3] + f[4]) - 4 (-2 f[3] + f[4] + f[5]) - 2 (d[3] - f[3]) s[
```

```
Simplify[%]
```

```
2 (-2 f[1] - 2 f[2] + 8 f[3] - 2 f[4] - 2 f[5] - d[3] s[3] + f[3] s[3])
```

Now we simulate the sampled data, and then set up the weight matrix:

```
size = 60;
xs = Table[Random[Integer,1], {i,1,size}];
data = Table[N[Sin[2 Pi (1/20) j] xs[[j]]],
{j, 1, size}];

Sm = DiagonalMatrix[xs];
Tm = Table[0,{i,1,size},{j,1,size}];
For[i=1,i<=size,i++,Tm[[i,i]] = xs[[i]]+8];
For[i=1,i<size,i++, Tm[[i+1,i]] = -2];
For[i=1,i<size,i++, Tm[[i,i+1]] = -2];
For[i=1,i<(size-1),i++, Tm[[i+2,i]] = -2];
For[i=1,i<(size-1),i++, Tm[[i,i+2]] = -2];
```

We will give the smoothness term a little more weight:

```
dt = 0.1;
Tf[f1_] := f1 - dt (Tm.f1 - .3 Sm.data);
```

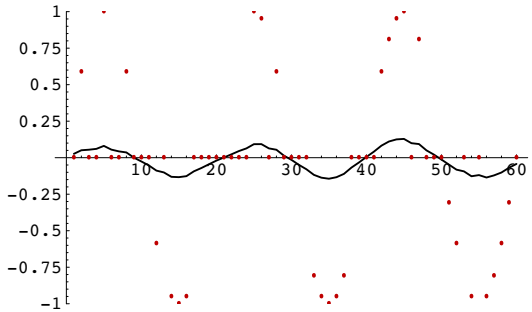
```
f = Table[0,{i,1,size}];
result = Nest[Tf,f,30];
```

```
g1 = ListPlot[result,PlotJoined->True,PlotRange->{-1,1},
    DisplayFunction->Identity];
```

```
g2 = ListPlot[data,PlotJoined->False,PlotRange->{-1,1},
PlotStyle->{RGBColor[.75,.0,0]}, Prolog-> AbsolutePointSize[5],
DisplayFunction->Identity];
```



```
Show[g1,g2,DisplayFunction->$DisplayFunction];
```



Deriving learning rules

In the previous section, we "sculpted" an energy function based on fidelity and smoothness constraints. The sculpting determined the parameters (or weights). Then we used gradient descent to derive an update rule from the energy function.

Can we do a similar thing to derive weight adjustment rules for learning? In other words construct something like an energy function, but where the weights are the variables, rather than the neural activities? We've already done that when we derived learning rules for Widrow-Hoff and back-prop. Widrow-Hoff produced a learning rule that was similar to the Hebb rule, but used the outer product to associate predicted error with the input. Let's look at another example, but this time in self-organization.

We re-visit the decorrelation problem which came up in the context of contingent adaptation. We'll see how to construct cost functions that embody desired constraints on the weights, and then use gradient descent to derive a self-organizing learning rule. As before, there is no guarantee that the learning rule will have neural plausibility; however, it does provide a solution to optimizing our constraints.

An example from non-orthogonal decorrelation

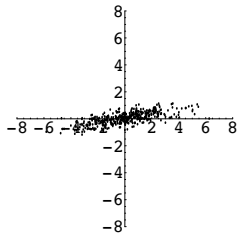
The generative model

Suppose we have a two-dimensional input. The elements are correlated, and one has a higher variance than the other. Similar to that used for the PCA demonstrations. Uses the Multinormal add-on package. Define the covariance matrix r:

```
(r = {{4, .6}, {.6, .2}};
ndist = MultinormalDistribution[{0, 0}, r]);
rv :=Random[ndist];
```

```
npoints = 400;
rvsamples = Table[rv,{n,1,npoints}];
```

```
g1 = ListPlot[rvsamples,PlotRange->{{-8,8},{-8,8}},
    AspectRatio->1, DisplayFunction->Identity];
Show[g1, DisplayFunction-> $DisplayFunction];
```



Find weights such that 1) they define a linear basis; 2) may be orthogonal, normalized, or non-orthogonal

■ **Algebra:**

```
xv = {x1, x2}; wv1 = {w11, w12}; wv2 = {w21, w22};
W = {{w11, w12}, {w21, w22}};
e = (xv - W.xv.W).(xv - W.xv.W);
```

■ **Rule 1: Find orthogonormal basis**

```
deltaW1[w11_, w12_, w21_, w22_, x1_, x2_] :=
 Evaluate[{{D[e, w11], D[e, w12]}, {D[e, w21], D[e, w22]}}]
```


■ **Rule 2: Find decorrelating basis**

The product of the outputs is added as a constraint. This can be minimized by having one or the other output high but not both at the same time.

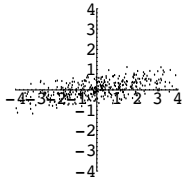
```
e3 = e + 2 * (xv.wv1) * (xv.wv2);
```

```
(*e3=(xv.wv1)*(xv.wv2);*)
```

```
deltaW3[w11_, w12_, w21_, w22_, x1_, x2_] :=
 Evaluate[{{D[e3, w11], D[e3, w12]}, {D[e3, w21], D[e3, w22]}}]
```

Try them out: learning simulation

```
npoints = 400;
rvsamples = Table[rv,{n,1,npoints}];
g1 = ListPlot[rvsamples,PlotRange->{{-4,4},{-4,4}},
    AspectRatio->1, DisplayFunction->Identity];
Show[g1, DisplayFunction-> $DisplayFunction];
```

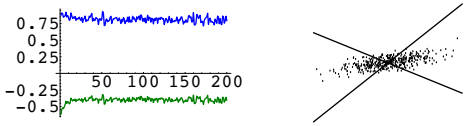



```
npoints = 2000; p1 = {}; α = 0.005;size = 2;
W0=Table[Random[],{i,1,size},{j,1,size}];
x=Table[0,{i,1,size}];
```

```
For[i=1,i<=npoints,i++,
        {x[[1]],x[[2]]}=rv;
        W0=W0-α
deltaW3[W0[[1,1]],W0[[1,2]],W0[[2,1]],W0[[2,2]],x[[1]],x[[2]]];
        If[Mod[i,50]==0,
                p1 = Join[p1,{{W0[[1,2]]/W0[[1,1]],
                    W0[[2,2]]/W0[[2,1]] }}]];

(*Print[W0,deltaWW[W0[[1,1]],W0[[1,2]],W0[[2,1]],W0[[2,2]],x[[1]],x[[2]]]]*
)
];
```

```
gg1=ListPlot[Map[#[[2]]&,p1], PlotJoined->True,DisplayFunction-
>Identity,PlotStyle->{RGBColor[0,.5,0]}];
gg2=ListPlot[Map[#[[1]]&,p1], PlotJoined->True,DisplayFunction-
>Identity,PlotStyle->{RGBColor[0,0,1]}];
gnetwork = Plot[{s p1[[Length[p1]]][[1]], s p1[[Length[p1]]][[2]]}, {s,
-6, 6}, PlotRange->{{-6,6},{-6,6}},AspectRatio->1,DisplayFunction-
>Identity];
(*Show[gnetwork,g1,DisplayFunction->$DisplayFunction,Axes->False];
Show[{gg1,gg2}, DisplayFunction->$DisplayFunction];*)
Show[GraphicsArray[{Show[{gg1,gg2}],Show[gnetwork,g1,Axes-
>False]}],DisplayFunction->$DisplayFunction];
```



References

Hertz, J., Krogh, A., & Palmer, R. G. (1991). *Introduction to the theory of neural computation* (Santa Fe Institute Studies in the Sciences of Complexity ed. Vol. Lecture Notes Volume 1). Reading, MA: Addison-Wesley Publishing Company.Poggio, T., Torre, V., & Koch, C. (1985). Computational vision and regularization theory. Nature, 317, 314-319.

Kersten, D., O'Toole, A. J., Sereno, M. E., Knill, D. C., & Anderson, J. A. (1987). Associative learning of scene parameters from images. Appl. Opt., 26, 4999-5006.

Kersten, D. J. (1991). Transparency and the Cooperative Computation of Scene Attributes. In M. Landy, & A. Movshon (Ed.), Computational Models of Visual Processing (pp. 209-228). Cambridge, Massachusetts: M.I.T. Press.

Kersten, D., & Madarasmi, S. (1995). The Visual Perception of Surfaces, their Properties, and Relationships. In I. J. Cox, P. Hansen, & B. Julesz (Eds.), Partitioning Data Sets: With applications to psychology, vision and target tracking, (pp. 373-389): American Mathematical Society.

Koch, C., Marroquin, J., & Yuille, A. (1986). Analog "neuronal" networks in early vision. Proc. Natl. Acad. Sci. USA, 83, 4263- 4267.) See also, Hertz et al., page 82-87.

Nakayama, K., & Shimojo, S. (1992). Experiencing and perceiving visual surfaces. Science, 257, 1357-1363.