Opinionated Lessons in Statistics

by Bill Press

#46 Interpolation on Scattered Data

Professor William H. Press, Department of Computer Science, the University of Texas at Austin

Interpolation on Scattered Data in Multidimensions

In dimensions >2, the explosion of volume makes things difficult.

Rarely enough data for any kind of mesh.

Lots of near-ties for nearest neighbor points (none very near).

The problem is more like a machine learning problem:

Given a training set $\mathbf{x}_i$ with "responses" y_i, $i = 1 \ldots N$
Predict $y(\mathbf{x})$ for some new $\mathbf{x}$

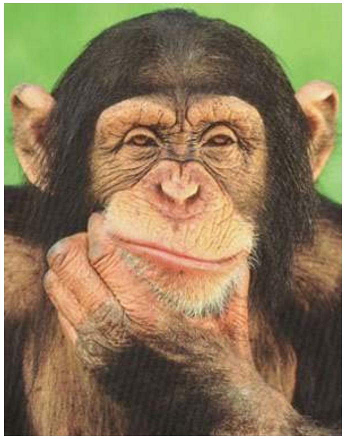
Example: In a symmetrical multivariate normal distribution in large dimension D, everything is at almost the same distance from everything else:

$$\mathbf{x}: \quad x_i \sim \text{Normal}(0,1)$$

$$\mathbf{x}_1 - \mathbf{x}_2: \quad (x_{1i} - x_{2i}) \sim \text{Normal}(0, \sqrt{2})$$

$$\|\mathbf{x}_1 - \mathbf{x}_2\|^2 \sim 2 \ \text{Chisq}(D)$$

$$\approx 2(D \pm \sqrt{2D})$$



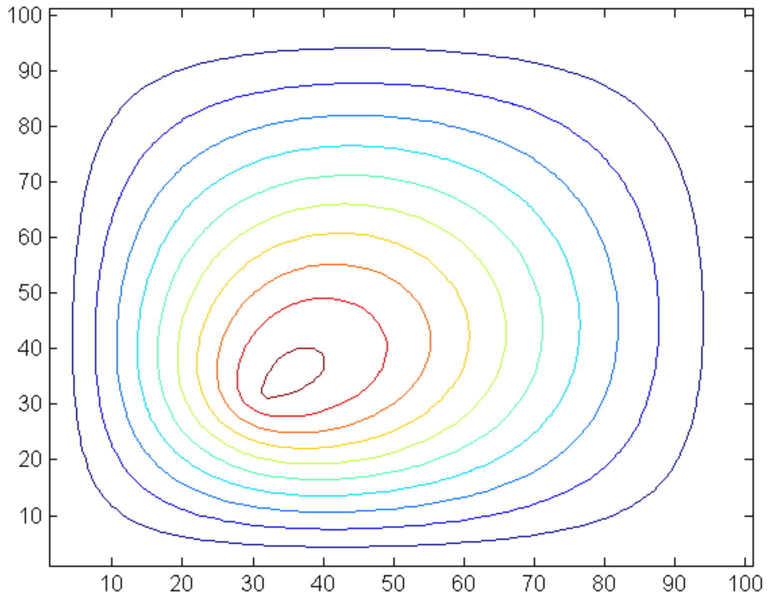
Let's try some different methods on 500 data points in 4 dimensions.

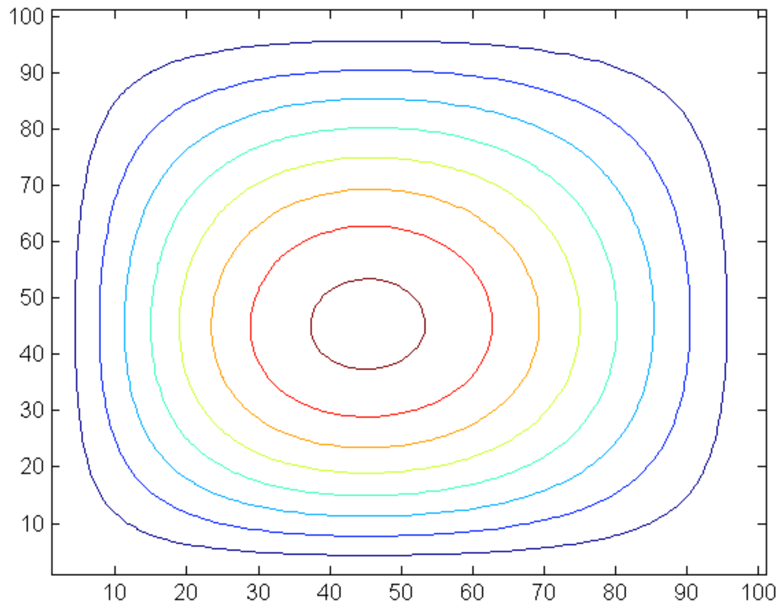
$500^{1/4} \approx 4.7$, so the data is sparse, but not ridiculous

```
testfun = @(x) 514.19*exp(-2.0*norm(x-[.3 .3 .3 .3])) ...
    *x(1)*(1-x(1))*x(2)*(1-x(2))*x(3)*(1-x(3))*x(4)*(1-x(4));
```

an exponential, off-center in the unit cube, and tapered to zero at its edges

```
[x1 x2] = meshgrid(0:.01:1,0:.01:1);
z = arrayfun(@(s1,s2) testfun([s1 s2 .3 .3]), x1, x2);
contour(z)
```



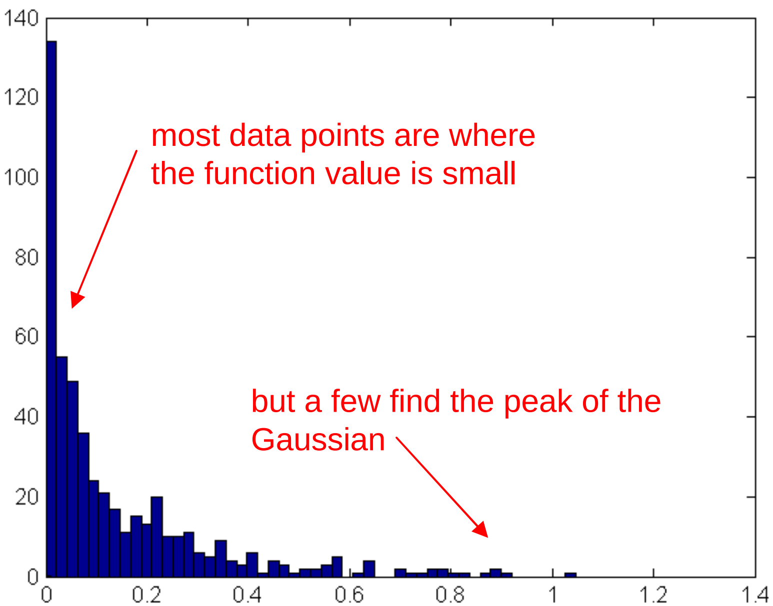


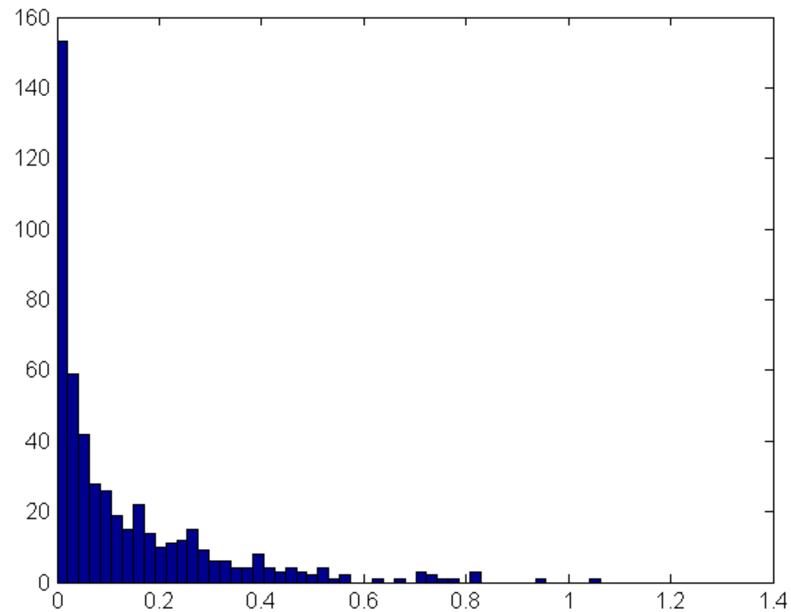

```
z = arrayfun(@(s1,s2) testfun([s1 s2 .7 .7]), x1, x2);
contour(z)
```

Generate training and testing sets of data.
The points are chosen randomly in the unit cube.

```
npts = 500;
pts = cell(npts,1);
for j=1:npts, pts{j} = rand(1,4); end;
vals = cellfun(testfun,pts);
hist(vals,50)
```

```
tpts = cell(npts,1);
for j=1:npts, tpts{j} = rand(1,4); end;
tvals = cellfun(testfun,tpts);
hist(tvals,50)
```





If you have only one sample of real data, you can test by leave-one-out, but that is a lot more expensive since you have to repeat the whole interpolation, including one-time work, each time.

Shepard Interpolation

The prediction is a weighted average of all the observed values, giving (much?) larger weights to those that are closest to the point of interest.

It's a smoother version of "value of nearest neighbor" or "mean of few nearest neighbors".

$$y(\mathbf{x}) = \frac{\sum_{i=0}^{N-1} y_i \phi(|\mathbf{x} - \mathbf{x}_i|)}{\sum_{i=0}^{N-1} \phi(|\mathbf{x} - \mathbf{x}_i|)}$$

$$\phi(r) = r^{-p}$$

the power-law form has the advantage of being scale-free, so you don't have to know a scale in the problem

In D dimensions, you'd better choose $p \geq D+1$, otherwise you're dominated by distant, not close, points: volume ~ no. of points ~ r^D

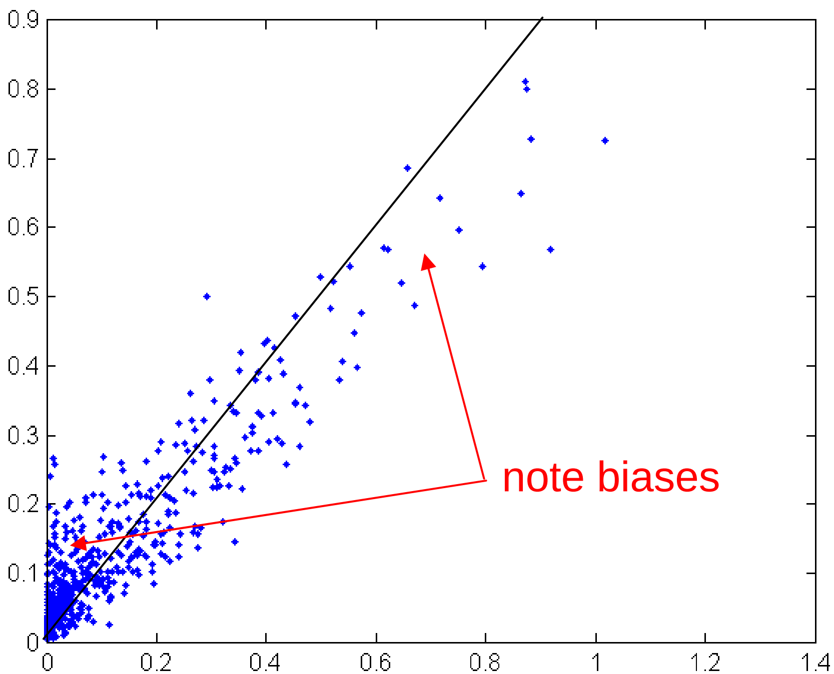
Shepard interpolation is relatively fast, O(N) per interpolation.
The problem is that it's usually not very accurate.

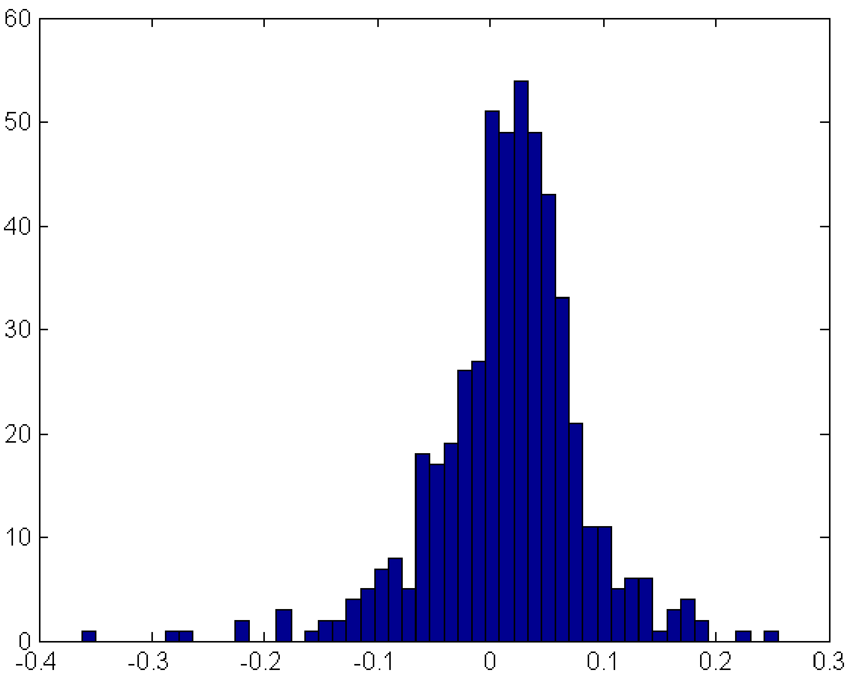
Shepard performance on our training/testing set:

```
    function val = shepinterp(x,p,vals,pts)
    phi = cellfun(@(y) (norm(x-y)+1.e-40).^(-p), pts);
    val = (vals' * phi)./sum(phi);
```

note value of p

```
shepvals = cellfun(@(x) shepinterp(x,6,vals,pts), tpts);
plot(tvals,shepvals,'.')
```





```
hist(shepvals-tvals,50)
```

Radial Basis Function Interpolation

This looks superficially like Shepard, but it is typically <u>much</u> more accurate.

However, it is also much more expensive:
$O(N^3)$ one time work + $O(N)$ per interpolation.

Like Shepard, the interpolator is a linear combination of identical kernels, centered on the known points

$$y(\mathbf{x}) = \sum_{i=0}^{N-1} w_i \phi(|\mathbf{x} - \mathbf{x}_i|)$$

But now we solve N linear equations to get the weights, by requiring the interpolator to go exactly through the data:

$$y_j = \sum_{i=0}^{N-1} w_i \phi(|\mathbf{x}_j - \mathbf{x}_i|) \quad \text{or} \quad \mathbf{\Phi}\,\mathbf{w} = \mathbf{y}$$

There is now <u>no</u> requirement that the kernel $\phi(r)$ falls off rapidly, or at all, with r.

Commonly used Radial Basis Functions (RBFs)

$$\phi(r) = (r^2 + r_0^2)^{1/2}$$ "multiquadric"

you have to pick a scale factor

$$\phi(r) = (r^2 + r_0^2)^{-1/2}$$ "inverse multiquadric"

$$\phi(r) = r^2 \log(r/r_0)$$ "thin plate spline"

$$\phi(r) = \exp\left(-\tfrac{1}{2} r^2 / r_0^2\right)$$ "Gaussian"

Typically very sensitive to the choice of r_0, and therefore less often used. (Remember the problems we had getting Gaussians to fit outliers!)

The choice of scale factor is a trade-off between over- and under-smoothing. (Bigger r_0 gives more smoothing.) The optimal r_0 is usually on the order of the typical nearest-neighbor distances.

Let's try a multiquadric with $r_0 = 0.1$

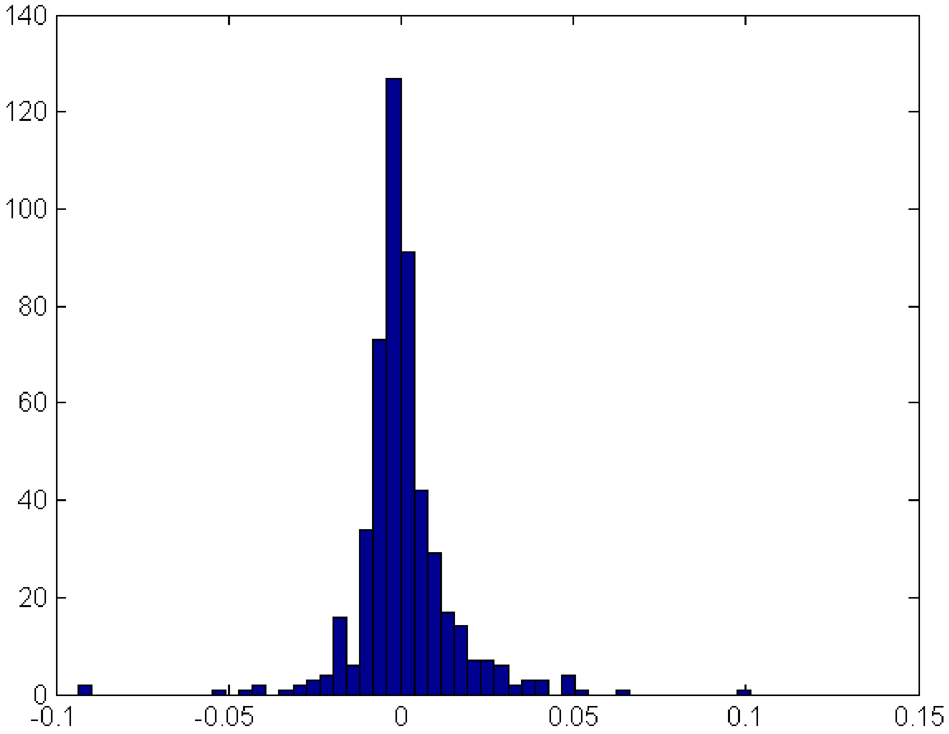
```
r0 = 0.1;
phi = @(x) sqrt(norm(x)^2+r0^2);
phimat = zeros(npts,npts);
for i=1:npts, for j=1:npts, phimat(i,j) = phi(pts{i}-pts{j}); end; end;
wgts = phimat\vals;
```

Matlab "solve linear equations" operator!

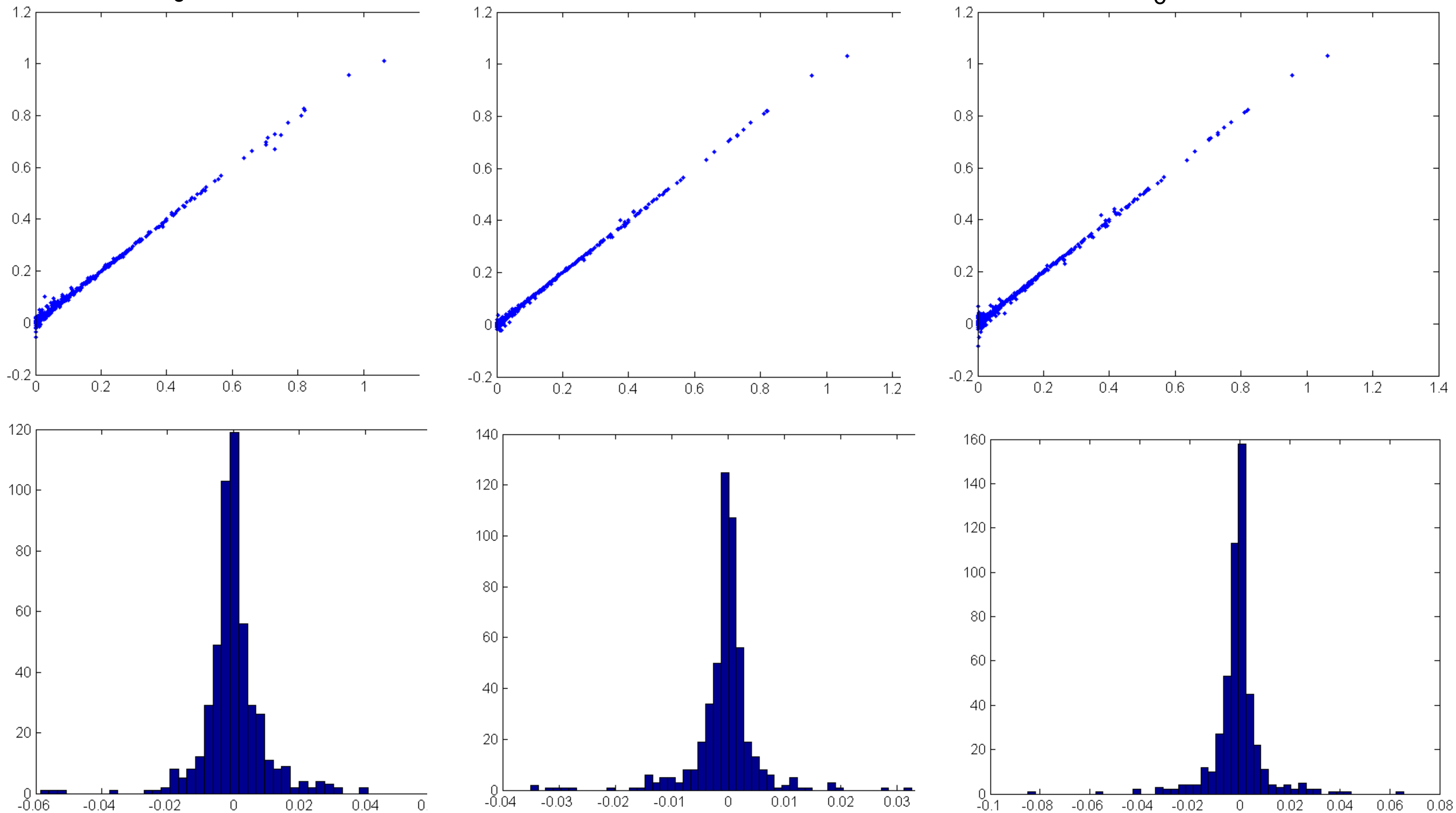
```
valinterp = @(x) wgts' * cellfun(@(y) phi(norm(x-y)),pts);
```

```
ivals = cellfun(valinterp,tpts);
hist(ivals-tvals,50)
stdev = std(ivals-tvals)
```

```
stdev =
    0.0145
```



$r_0 = 0.2$ $r_0 = 0.6$ $r_0 = 1.0$



$\sigma = 0.0099$ $\sigma = 0.0059$ $\sigma = 0.0104$

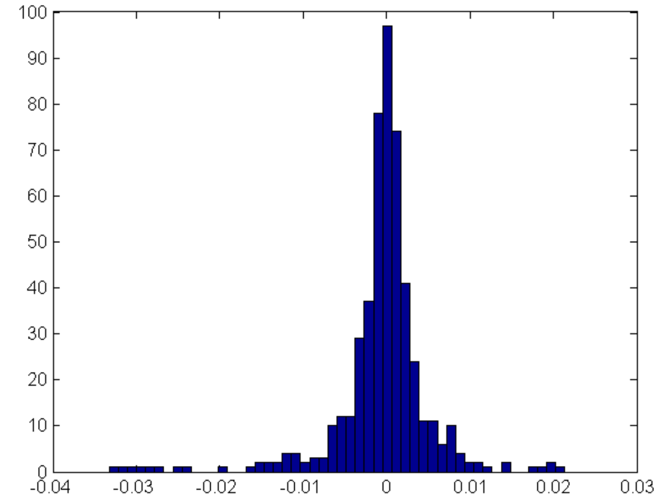
so $r_0 \sim 0.6$ is the optimal choice, and it's not too sensitive

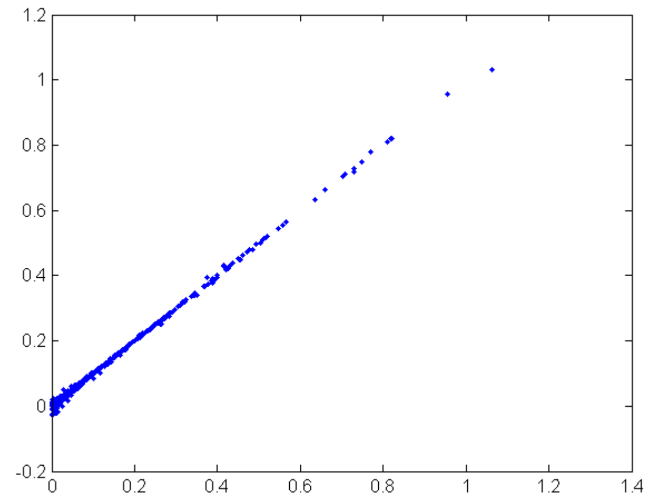
Try an inverse multiquadric

```
r0 = 0.6;
phi = @(x) 1./sqrt(norm(x)^2+r0^2);
[stdev ivals] = TestAnRBF(phi,pts,vals,tpts,tvals);
stdev
plot(tvals,ivals,'.')
stdev =
    0.0058
```

(performance virtually identical to multiquadric on this example)

$r_0 = 0.6$





$\sigma = 0.0058$

RBF interpolation is for interpolation on a smooth function, not for fitting a noisy data set.

By construction it exactly "honors the data" (meaning that it goes through the data points – it doesn't smooth them).

If the data is in fact noisy, RBF will produce an interpolating function with spurious oscillations.