Approximation Methods in Optimization —

The basic idea is that if you have a function that is noisy and possibly expensive to evaluate, then that function can be sampled at a few points and a fit of it created. Optimization is then performed not on the original function, but on the cheap to evaluate and smooth fit. As the optimization progresses to different areas of the design space, new fits are created.

These fits go by many names such as approximation models, output predictors, surrogate models, and response surfaces.

When Newton's method is used in nonlinear optimization, what it actually does is find a stationary point (minimum, maximum or saddle point) of the objective by modeling the objective as a quadratic. It uses this quadratic to predict the location of the stationary point and then makes another quadratic fit at the predicted location. By iterating in this fashion, the stationary point is found very efficiently. A variable metric, or quasi-Newton method is similar, except that it does not require second derivative information and has certain safeguards that guarantee convergence to a minimum.

A quadratic is the simplest function that has a minimum, and therefore it is a logical starting point for an approximate model. As in the Newton method, there has to be an algorithm for updating the fit as the optimization progresses including safeguards to prevent a bad fit from throwing everything off. One such method is the trust region framework. It simply says that the fit is not to be trusted beyond some distance from the points that were used to create it, so bounds are enforced during optimization on the fit to stay within the trust region.

The difficult part of the trust region approach is deciding on an update scheme that will not impede progress toward a minimum. One such algorithm is described in detail in the following paper by Alexandrov, et al.: http://www.icase.edu/Dienst/UI/2.0/Describe/ncstrl.icase/TR-97-50

Quadratic Response Surfaces —

In order to create a quadratic fit of a function, that function has to be sampled at some number of points. There have to be enough points sampled to solve for all the coefficients in the equation of a quadratic,

$$f = a + b_i x_i + c_{ij} x_i x_j.$$

Noting that the c matrix has to be symmetric, the number of unknowns (the scalar a, the vector b, and the matrix c) turns out to be (N+1)(N+2)/2 in N dimensions. So if the function is sampled at that many points, the unknown coefficients can by found as the solution of a linear system of equations.

As you might guess, the points have to be placed in the design space in such a way as to not make the system of equations singular. This turns out not to be a trivial problem when in spaces of many dimensions— a random scattering of points will fail to correctly span the space a significant amount of the time.

One way to pick the points is to evaluate the function at some guessed location, then perturb that location along each coordinate, both in the positive and negative directions, and then perturb, in the positive direction, all combination of pairs of coordinates. So in two dimensions this will yield the stencil of points in the next figure.

As can be seen from the figure, this set of points is not symmetrical. The fit will likely be more accurate in the first quadrant than elsewhere, and this could hurt the efficiency of an optimization algorithm that is based on such fits.

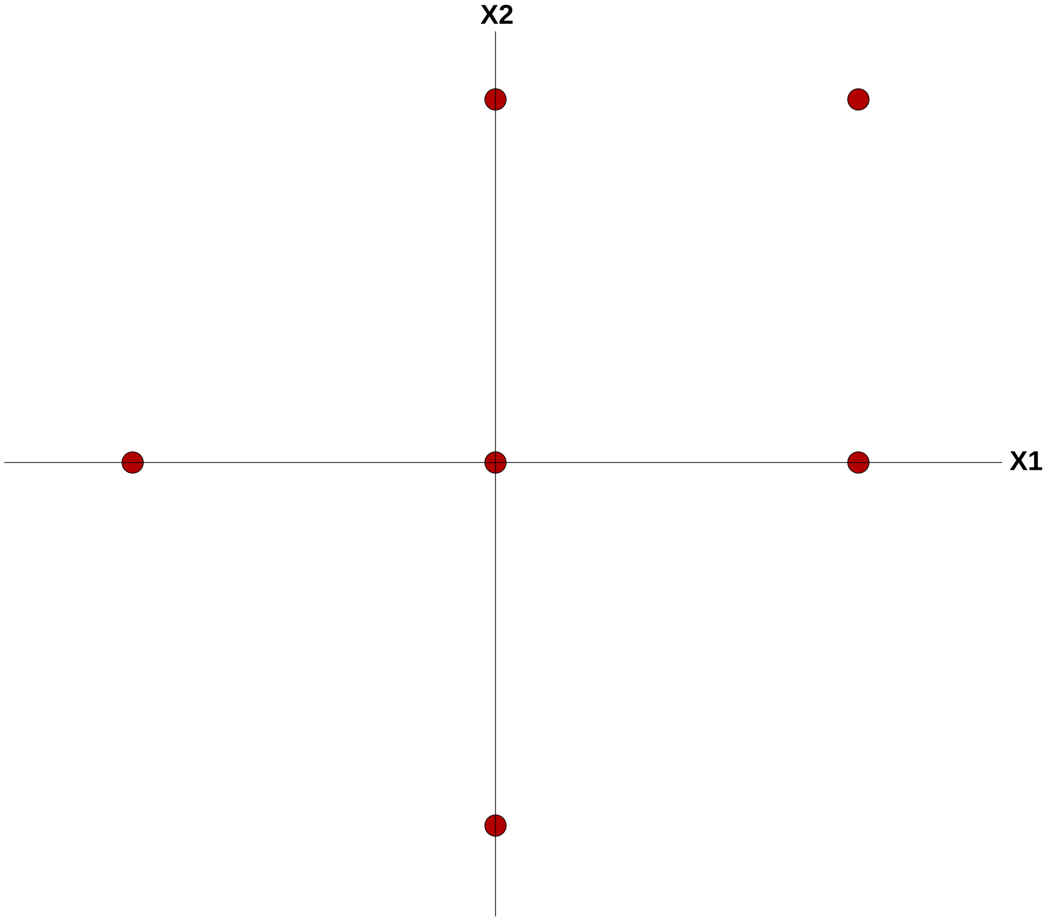


Figure 1: A stencil of points in two dimensions that can be used to sample a function in order to make a quadratic fit of that function.

The “best” stencil of points is one that most evenly spans the space. This can be quantified by minimizing the condition number of the system of equations used to solve for the coefficients of the quadratic. The next few figures show the results of this minimization in 2D and 3D.

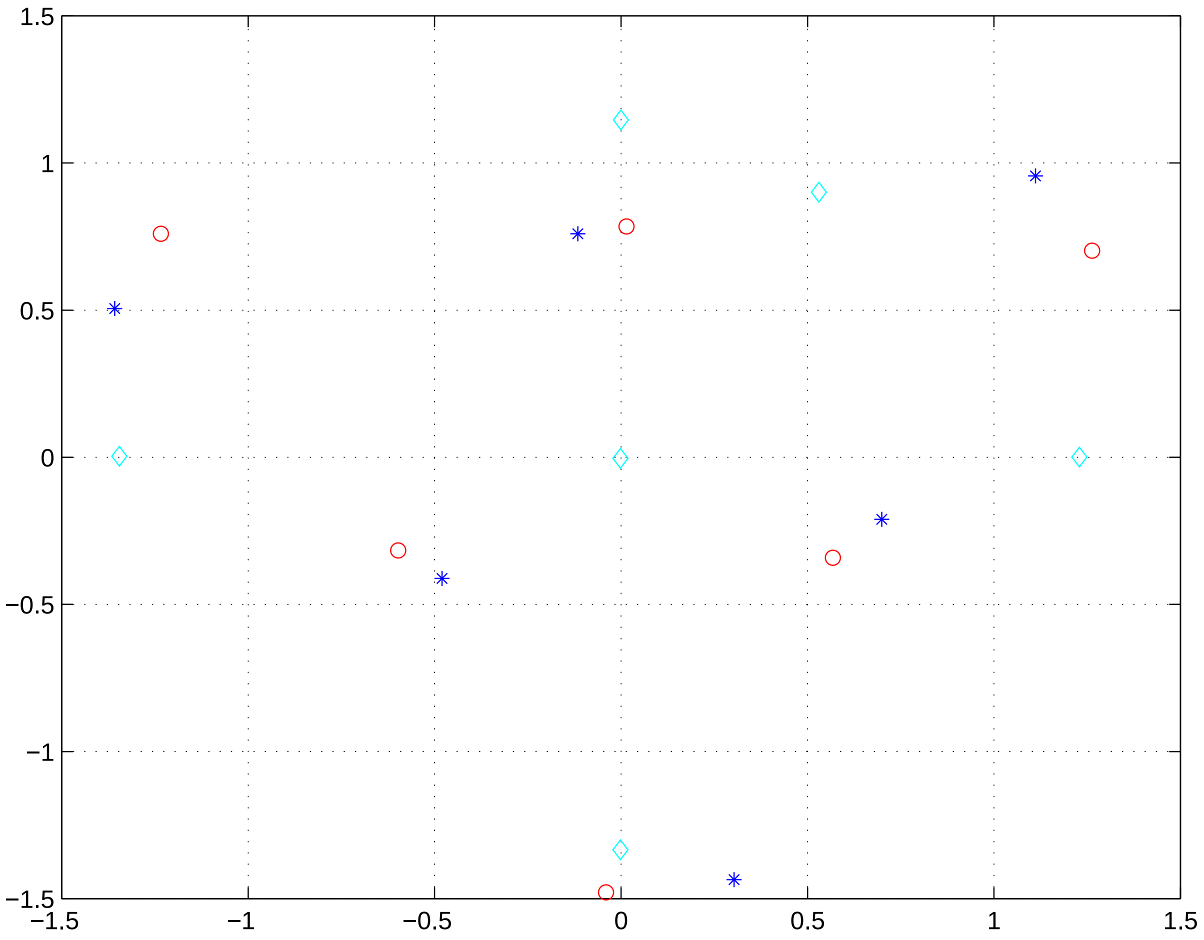


Figure 2: The original result of an optimization to minimize the condition number of the matrix that is used to solve for the coefficients of a quadratic fit in two dimensions. The blue asterisks and red circles are NPSOL and the Matlab quasi-Newton optimizer. The cyan diamonds are from another Matlab optimizer that failed to do anything useful.

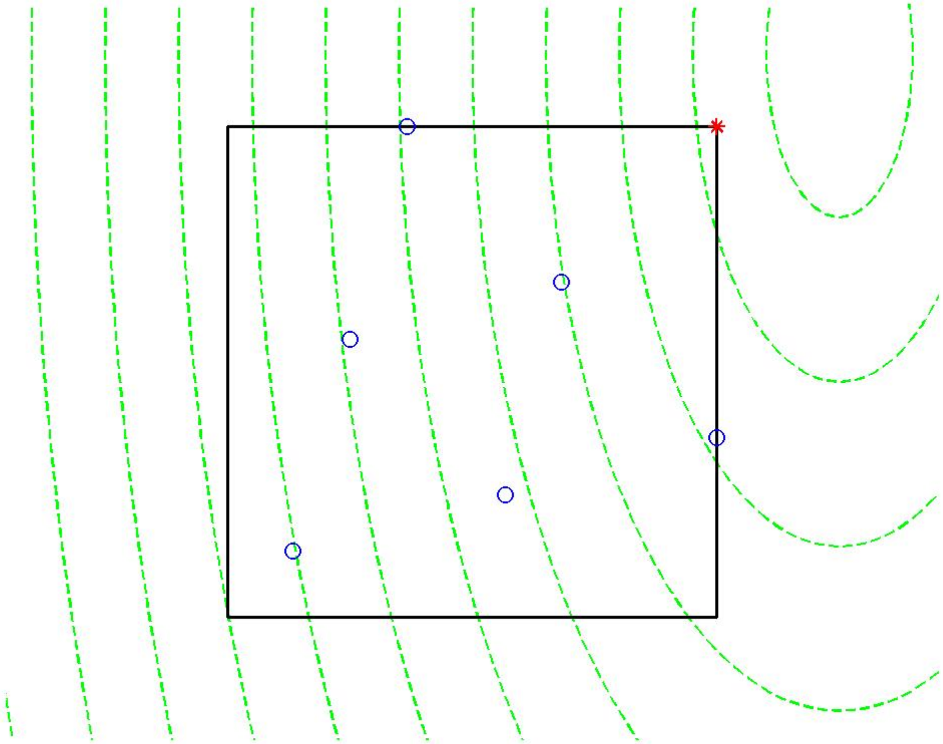
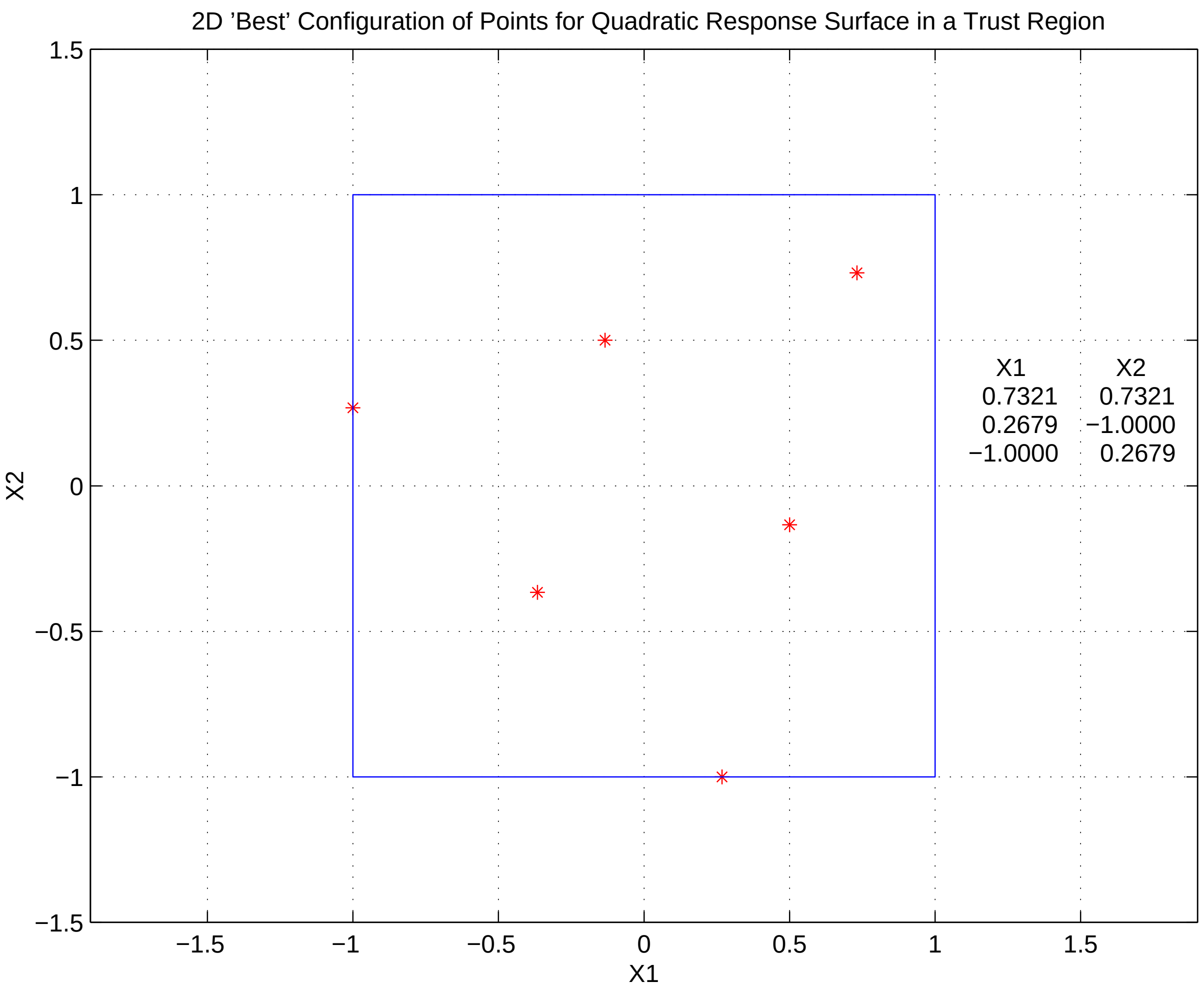
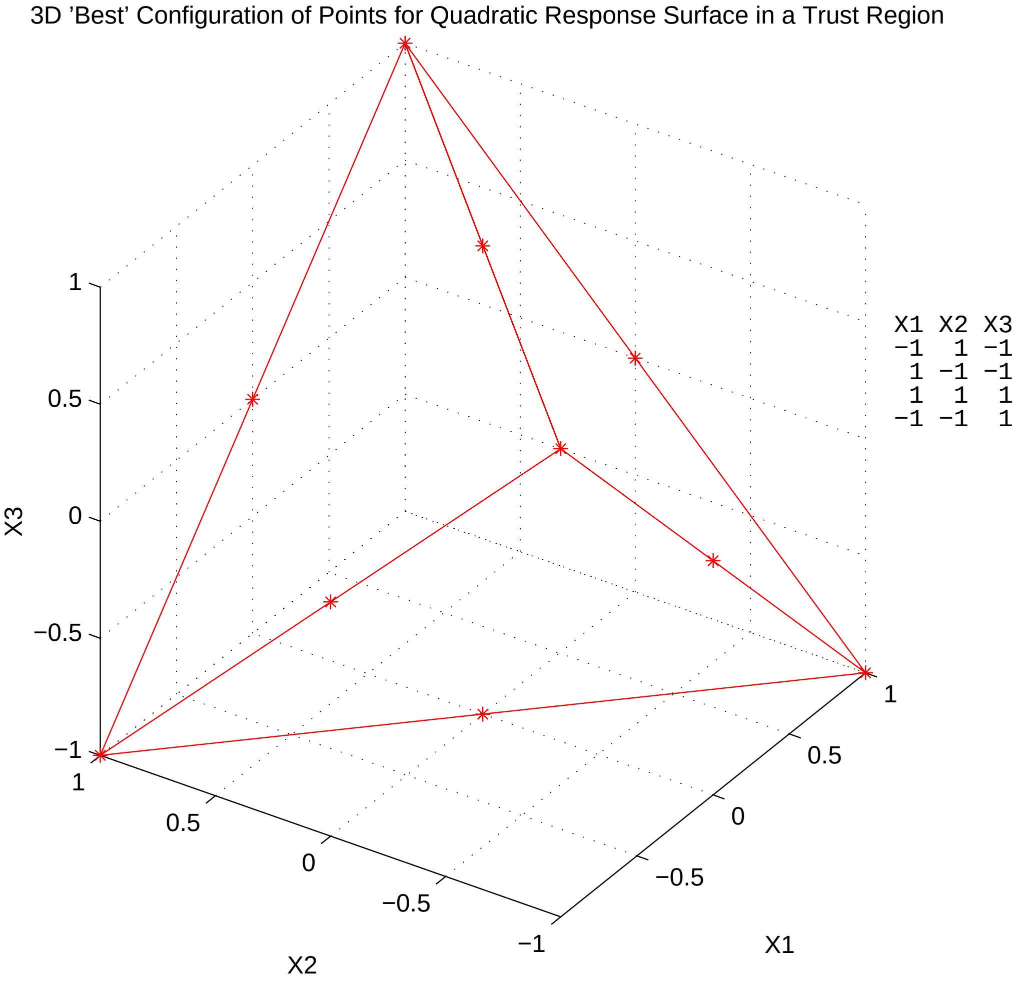


Figure 3: How the stencil of points found in the previous figure is used: The function is evaluated at the 6 points and a quadratic passing through those values is found (the green dashed lines). The minimum of that fit inside a trust region (black square) is found (red asterisk). That point is then predicted to be better than the original starting point.



X1	X2
0.7321	0.7321
0.2679	−1.0000
−1.0000	0.2679

Figure 4: The same 2D stencil of points arranged to maximize the area of the triangular stencil inside a square trust region (with the coordinates of the three vertices in the table on the right.)



3D 'Best' Configuration of Points for Quadratic Response Surface in a Trust Region

X1	X2	X3
−1	1	−1
1	−1	−1
1	1	1
−1	−1	1

Figure 5: Same idea as the previous figure, but for 3D. The shape is now a tetrahedron and it fits perfectly inside a cubic trust region. There are four vertices, and six mid-points along the edges for a total of 10 points needed in 3D to compute a quadratic fit.

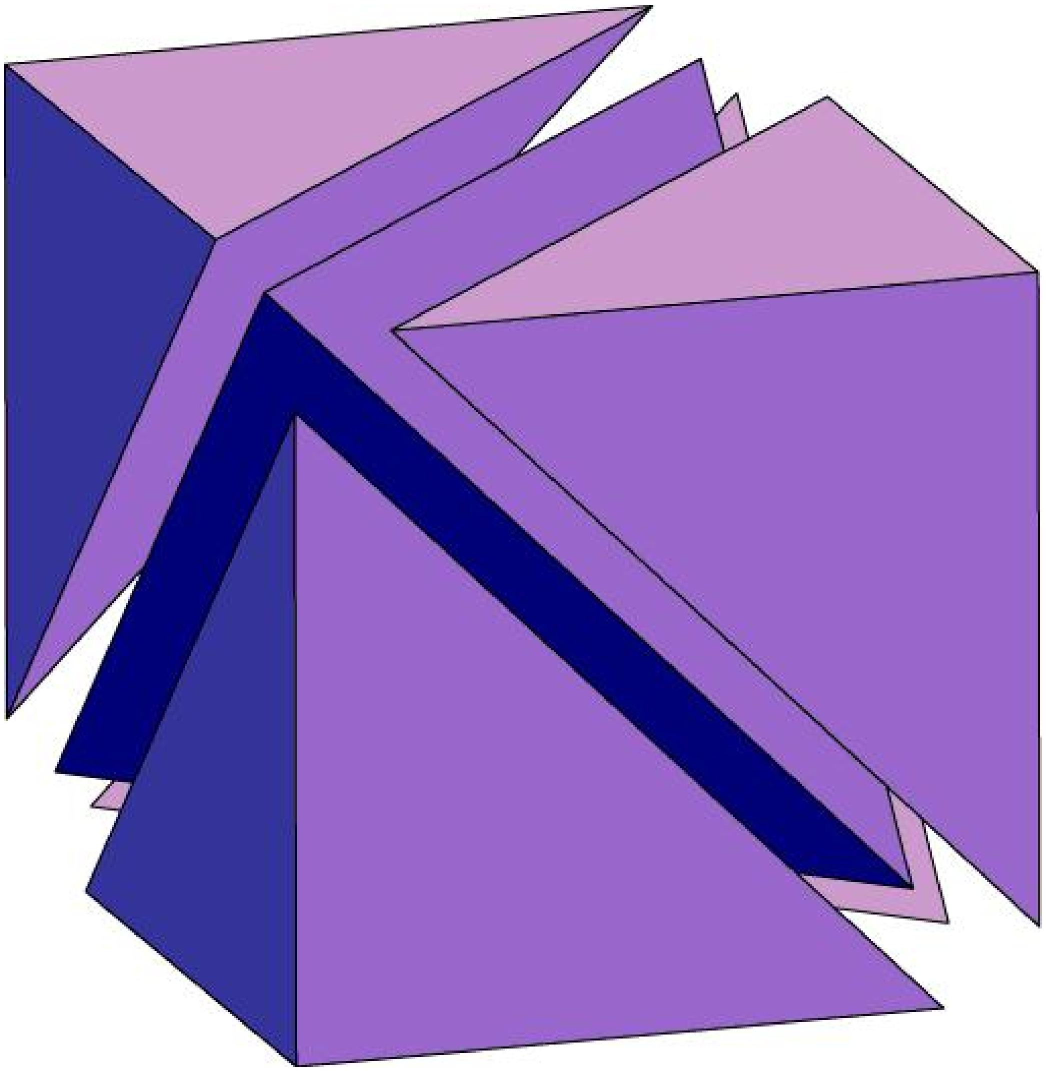


Figure 6: An isometric view of the tetrahedron from the previous figure. Imagine the cube is cut into the five pieces shown. The tetrahedron is at the center.

Following are the coordinates of the vertices of the triangle and tetrahedron from the last few figures. This generalizes to N-dimensions as the coordinates of a regular simplex. Those vertices plus all their mid-points are the “best” locations to evaluate a function to create a quadratic approximation of it. Note that it isn’t until 7 dimensions that the simplex fits inside an N-cube neatly.

```
2D:
xx =
   -0.7321   -0.7321
    1.0000   -0.2679
   -0.2679    1.0000

3D:
xx =
   -1.0000    1.0000   -1.0000
    1.0000   -1.0000   -1.0000
    1.0000    1.0000    1.0000
   -1.0000   -1.0000    1.0000
```

4D:

```
xx =
   -0.6321   -1.0000    1.0000   -0.0464
   -0.6321   -0.0464   -1.0000    1.0000
    0.8963    0.7299    0.7299    0.7299
   -0.6321    1.0000   -0.0464   -1.0000
    1.0000   -0.6835   -0.6835   -0.6835
```

5D:

```
xx =
   -0.7976   -0.7976    0.7976   -0.7976   -0.7976
    1.0000   -1.0000   -0.1819   -0.3843    1.0000
   -0.3843    1.0000   -1.0000   -1.0000    0.1819
    1.0000    1.0000    1.0000    0.1819   -0.3843
    0.1819   -0.3843   -1.0000    1.0000   -1.0000
   -1.0000    0.1819    0.3843    1.0000    1.0000
```

```
6D:
xx =
    1.0000   -1.0000   -1.0000    0.5912    0.1409   -0.6547
    0.6547    1.0000    0.1409    1.0000   -0.5912    1.0000
    0.1409   -1.0000    0.6547   -1.0000   -1.0000    0.5912
   -1.0000   -0.5912    1.0000    1.0000    0.6547   -0.1409
    0.7956    0.7956    0.7956   -0.7956    0.7956   -0.7956
   -0.5912    0.1409   -1.0000   -0.6547    1.0000    1.0000
   -1.0000    0.6547   -0.5912   -0.1409   -1.0000   -1.0000
7D:
xx =
    -1    -1    -1    -1    -1    -1     1
    -1    -1     1    -1     1     1    -1
     1    -1    -1     1    -1     1    -1
    -1     1     1     1    -1     1     1
     1    -1     1     1     1    -1     1
    -1     1    -1     1     1    -1    -1
     1     1     1    -1    -1    -1    -1
     1     1    -1    -1     1     1     1
```

Example — Fuselage Design

A CFD Euler code was combined with a boundary-layer solver to compute the flow on a wing-body. The problem is that the presence of the fuselage spoils the laminar flow that can normally be maintained on a thin, low sweep wing in supersonic flow. The goal is to reshape the fuselage at the wing-body junction to regain laminar flow on the wing.

Three design variables were used initially, with quadratic response surfaces and a trust region update algorithm.

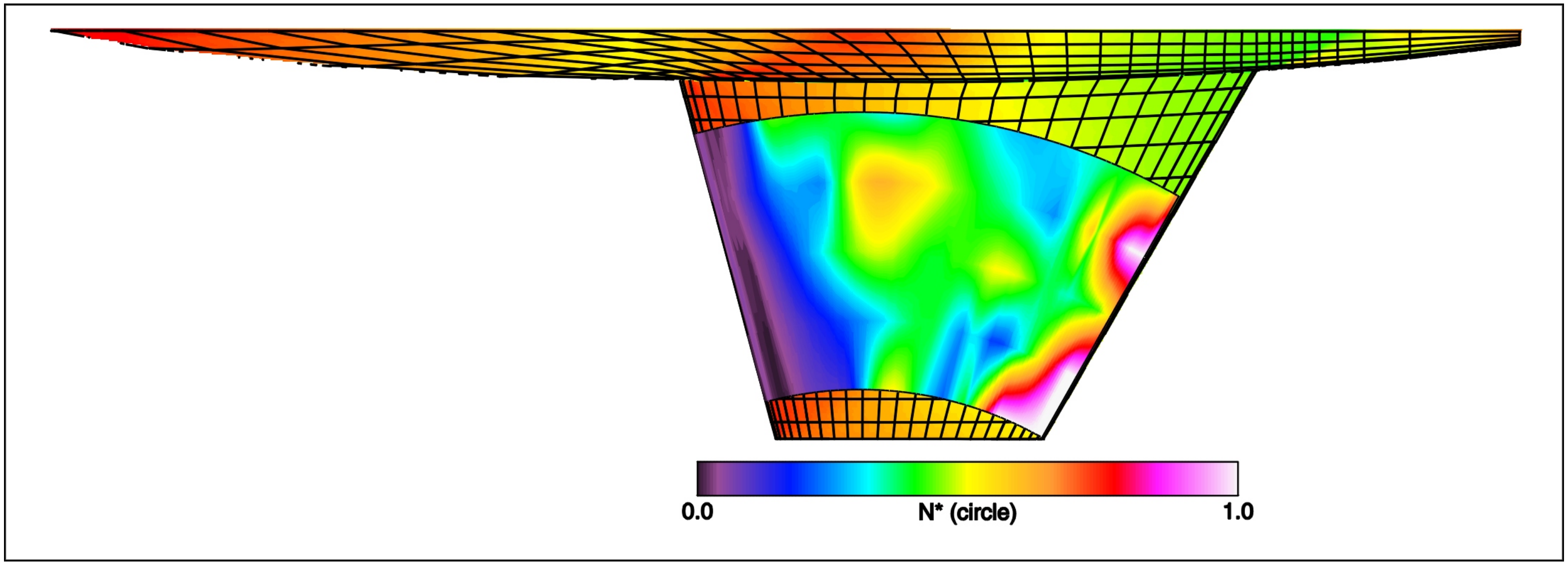


Figure 7: The flow is from left to right. Thinking of this as an airplane, this is the top view showing only the left half. The boundary-layer solution appears superimposed on the inviscid Euler pressures on the surface grid.

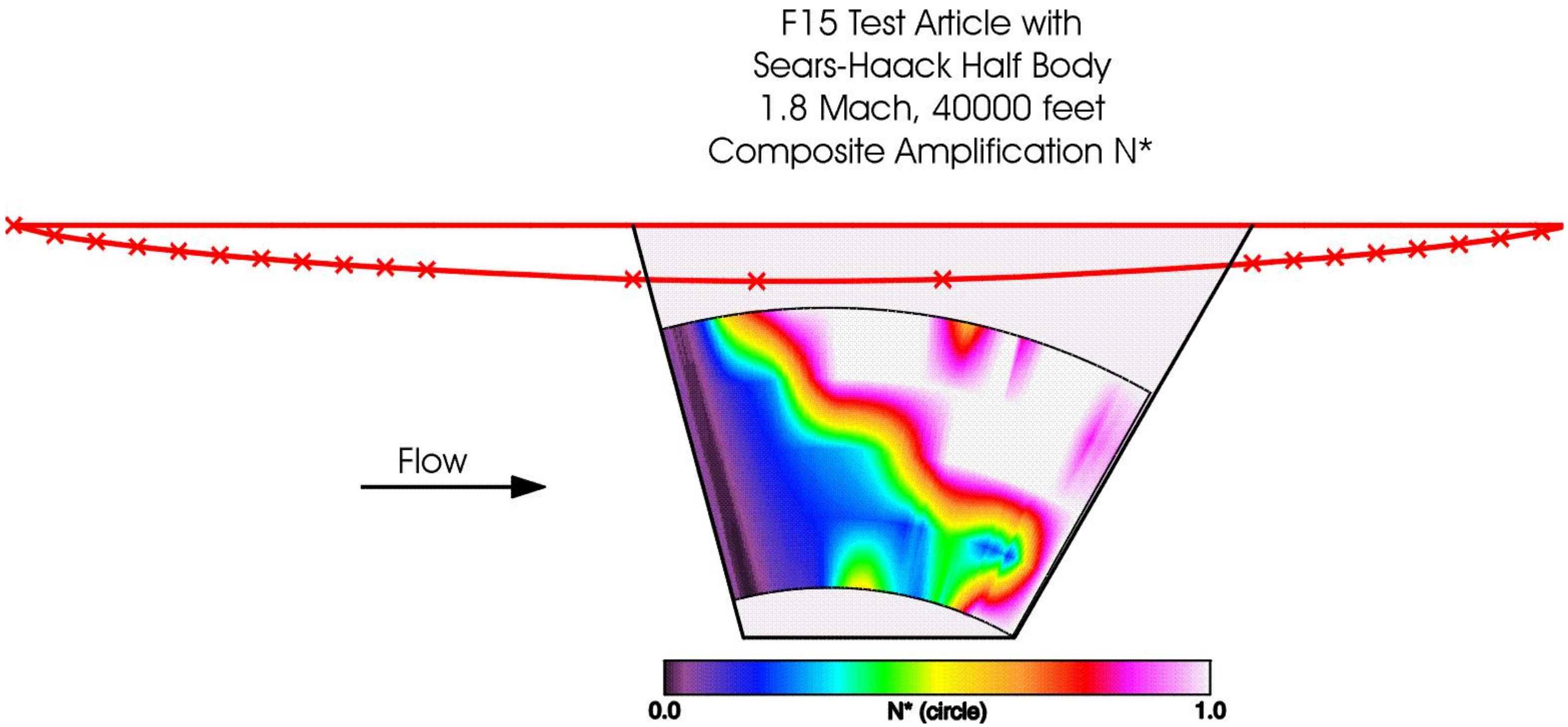


Figure 8: Here is the starting point: Just slapping a Sears-Haack body onto a wing results in early transition – the white areas in the boundary-layer solution. N* is the measure of laminar instability, with 1.0 – white – being the prediction of transition. The flow is then turbulent from the first occurrence of N*=1 to the trailing edge irrespective of further values of N*.

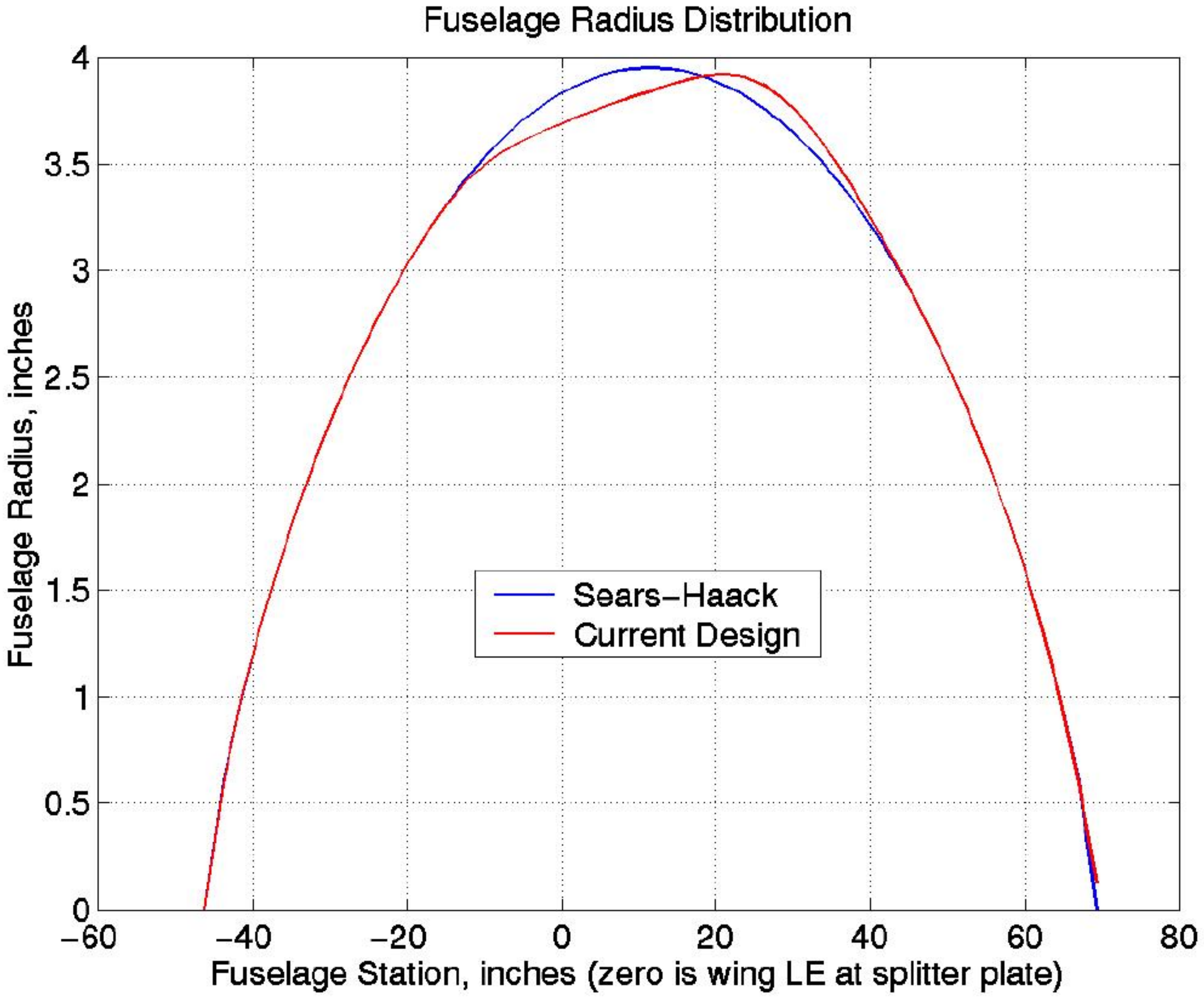


Figure 9: From the nose at left, to the tail at right, this is the radius of the original (blue) and re-designed (red) fuselage after two iterations.

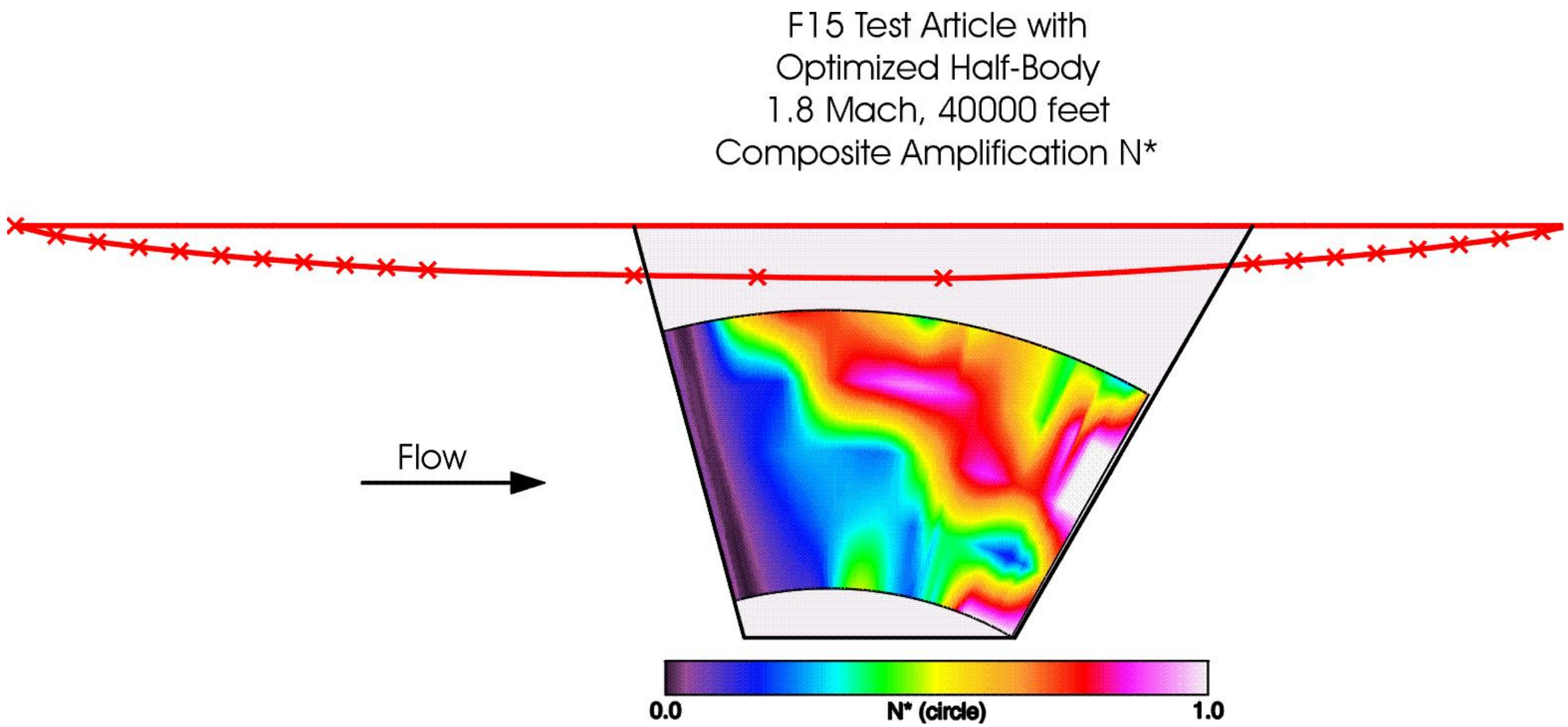


Figure 10: With only 3 design variables (the crosses on the fuselage outline that sit on the wing) and two iterations (nowhere near a converged optimization) the improvement is dramatic.

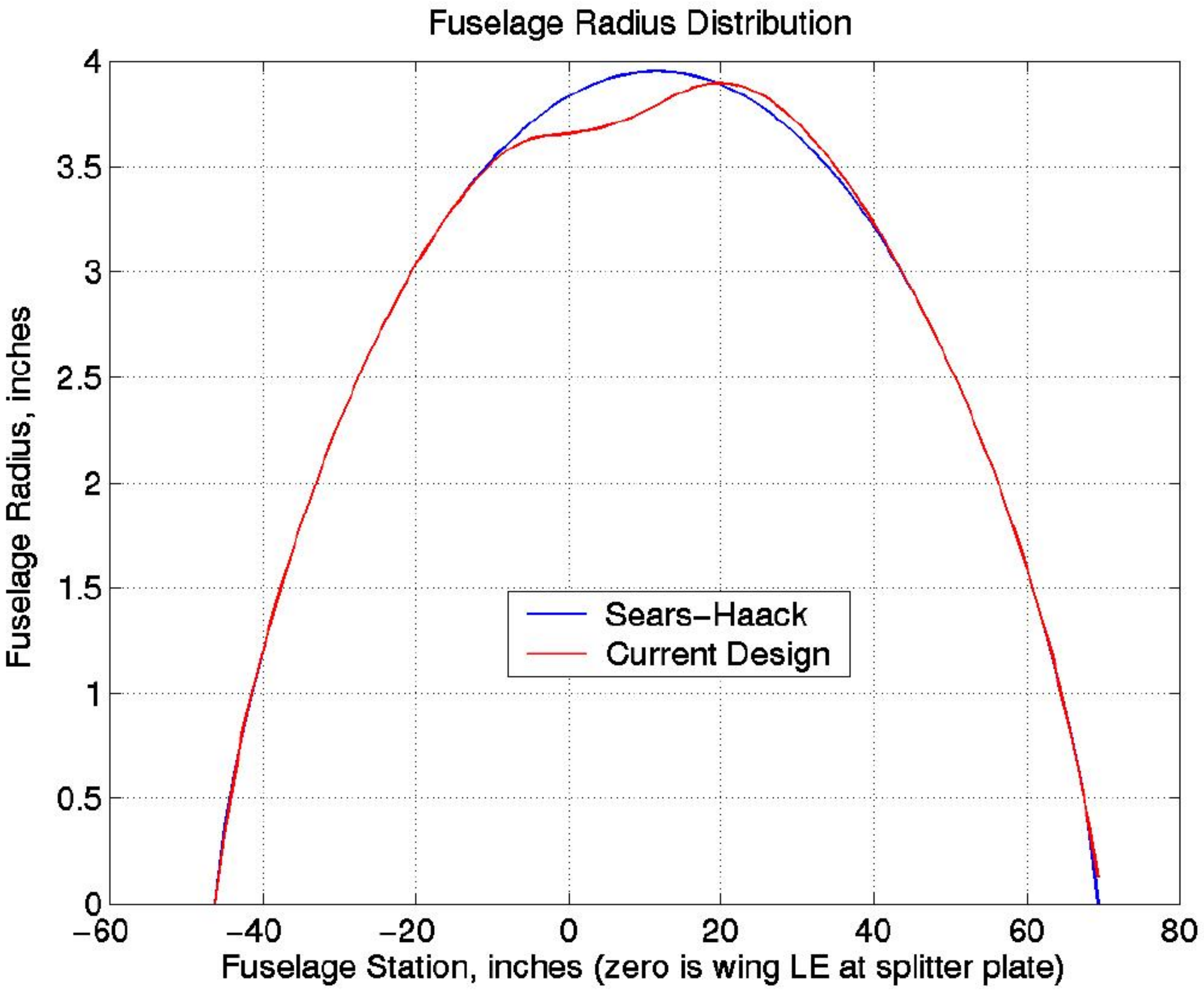


Figure 11: Now with five design variables, and a few more trust-region update cycles, a better solution is found.

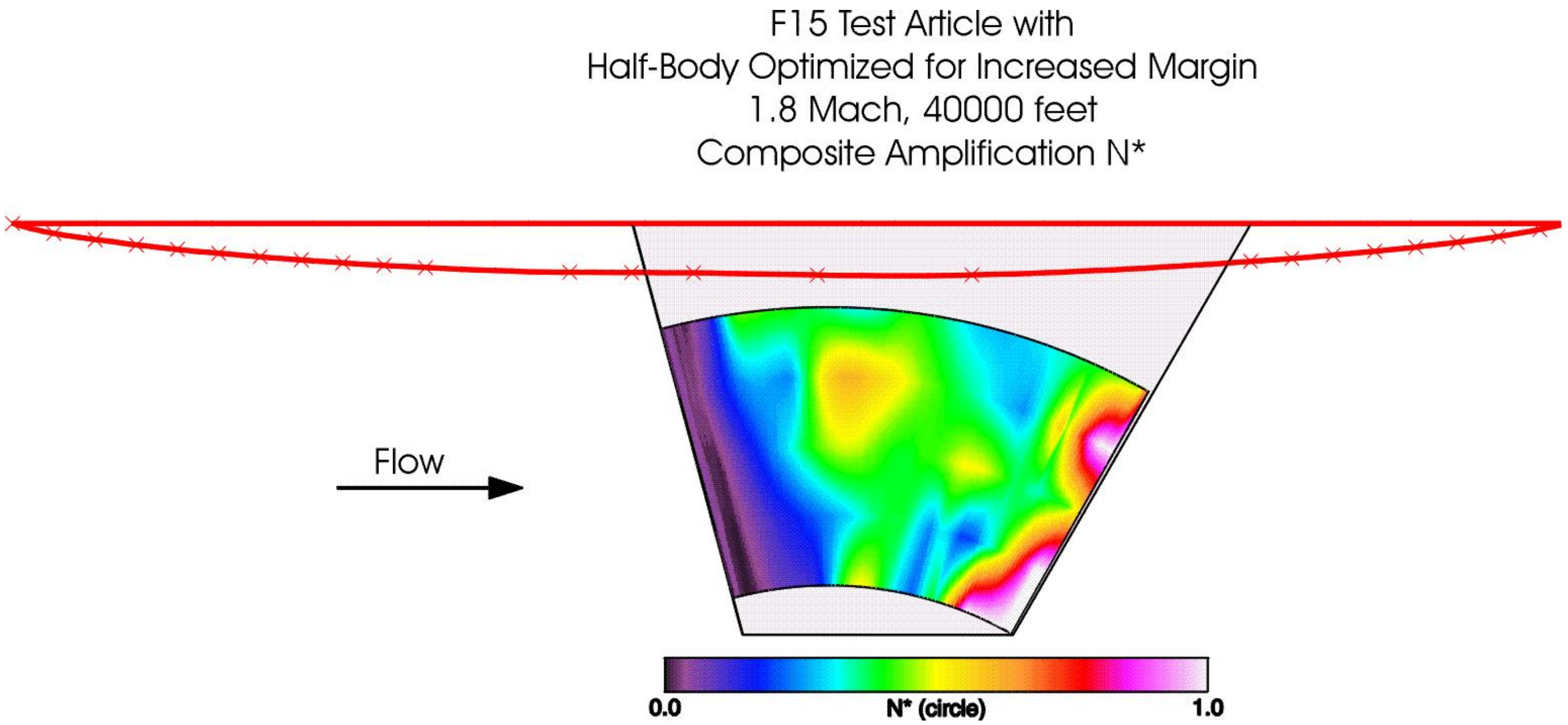


Figure 12: The boundary layer is much farther from transition to turbulent flow as can be seen by comparing the green and yellow colors on this wing with the red and violet colors two figures ago. Also notice how subtle the reshaping of the fuselage is.

Higher order fits —

Successive quadratic fitting during trust region updates can be inefficient because sampled data is routinely thrown away even if there are old points inside or near the new trust region. It would be very advantageous to keep those data and have some sort of global fit that builds up as new points are evaluated.

In one dimension, the most common way to do this is probably the cubic spline. Unfortunately the cubic spline, and nearly all other fitting methods lead to funny oscillations that are not always representative of the real behavior of the original function. This leads to many false local minima appearing in the fits which really hurts the efficiency of the optimization.

Cubic splines, polynomial fits, Fourier spectral methods and the osculatory method of Ackland all exhibit some amount of weird wiggling. One spline specifically designed to avoid unnatural oscillations is the Akima spline (see next figure). It was originally designed by H. Akima in 1969, and subsequently improved in 1991. The goal was to achieve a mathematical method of fitting that closely resembles what people do when they interpolate points by hand.

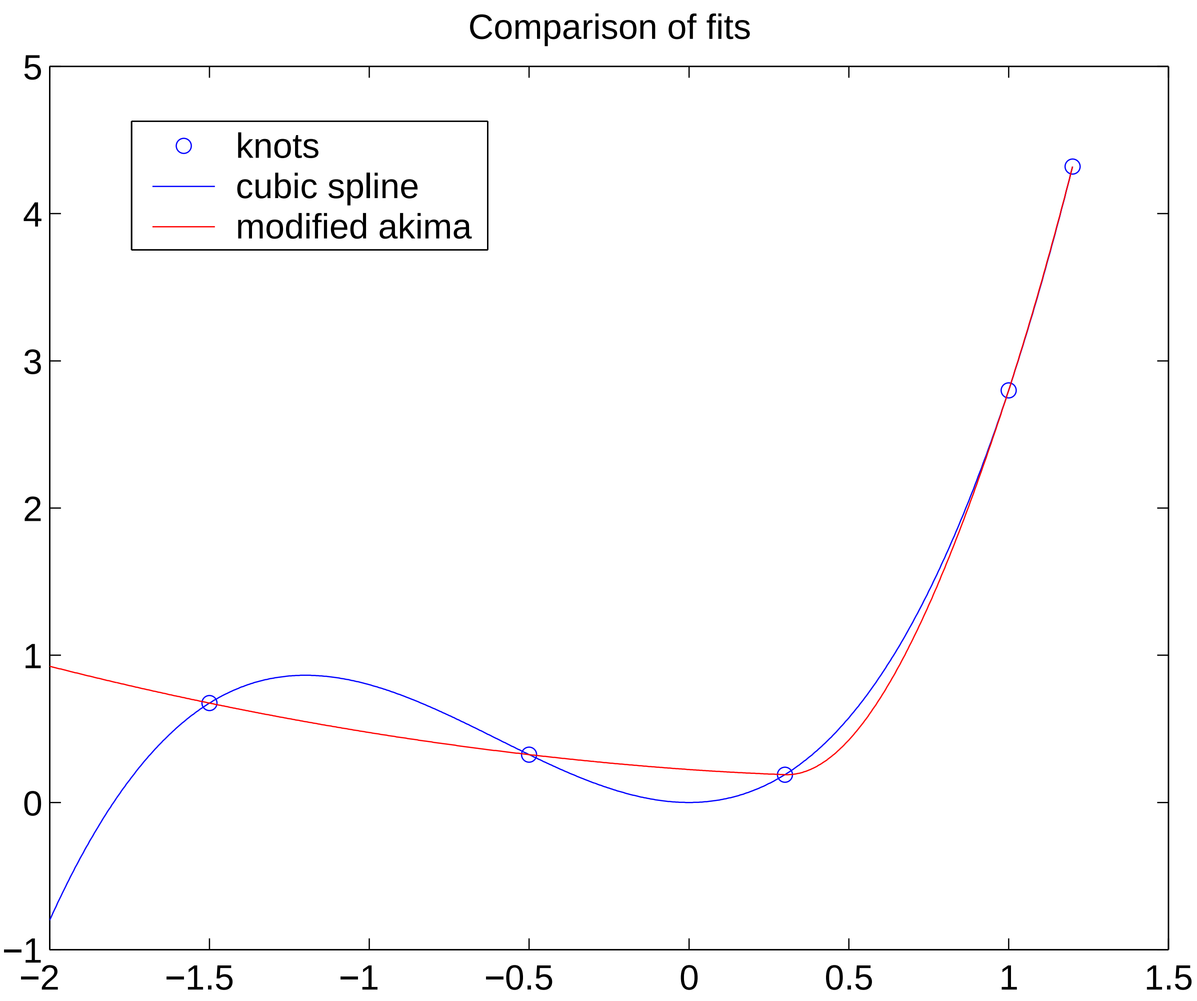
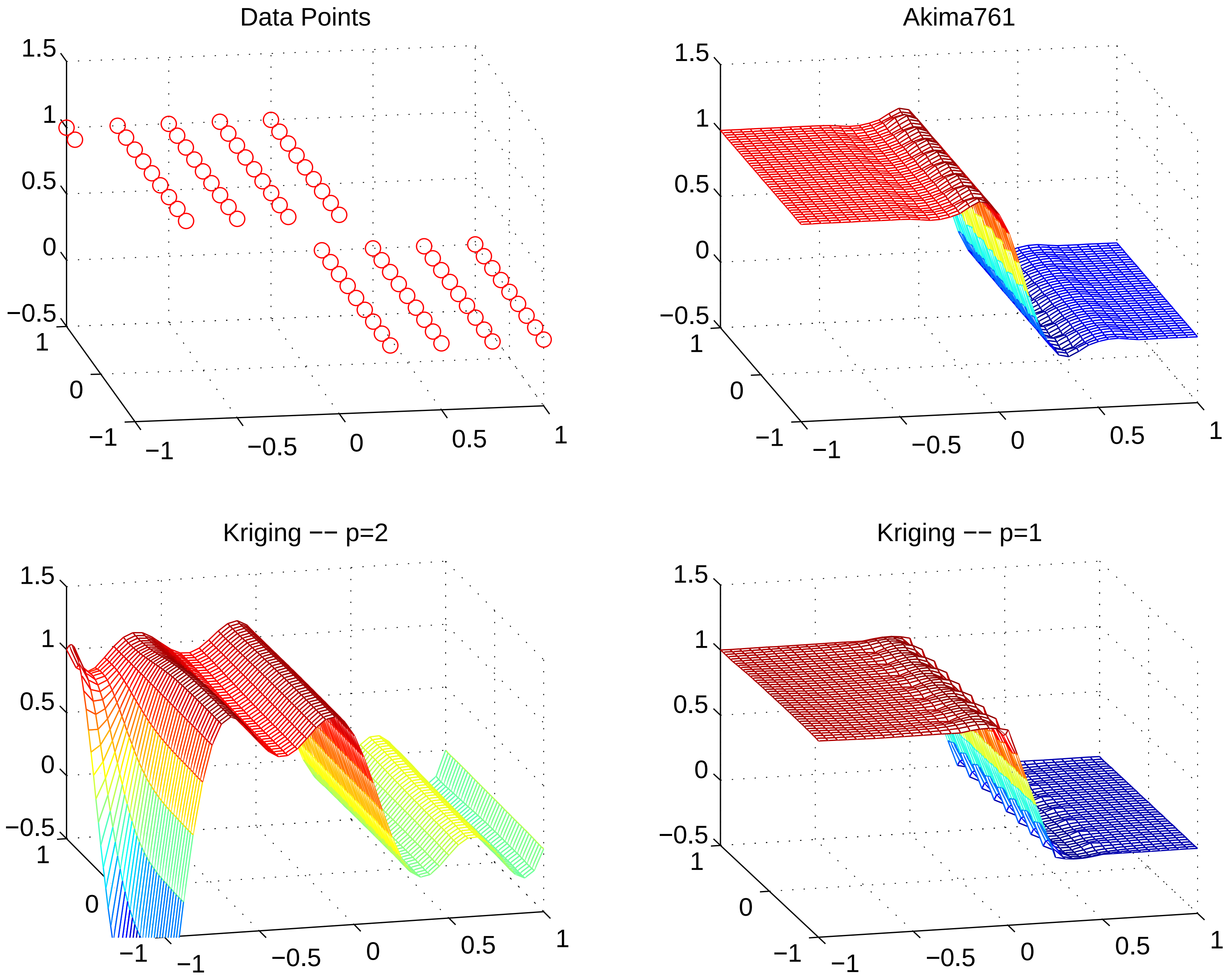


Figure 13: The Akima method of 1991 — ACM TOMS algorithm 697 (vol. 17 no. 3, http://www.acm.org/pubs/contents/journals/toms/) slightly modified to use a different "volatility" measure — is nearly always a more natural fit than any other spline I've seen.

Unfortunately, the Akima method is difficult to extend to higher dimensions. Akima has an algorithm for 2D when the data samples are on a Cartesian grid (which is too restrictive for most optimization algorithms) and another (Akima 761) that can handle scattered data. The next two figures show this method on two types of step functions. The results are not too encouraging. Not only are there extra oscillations, but the computer program is quite involved— and this is only two dimensions!

In fact, the lower plots in the following figures show another method that is at least as good as Akima's algorithm 761. This is kriging, the topic of the next section.

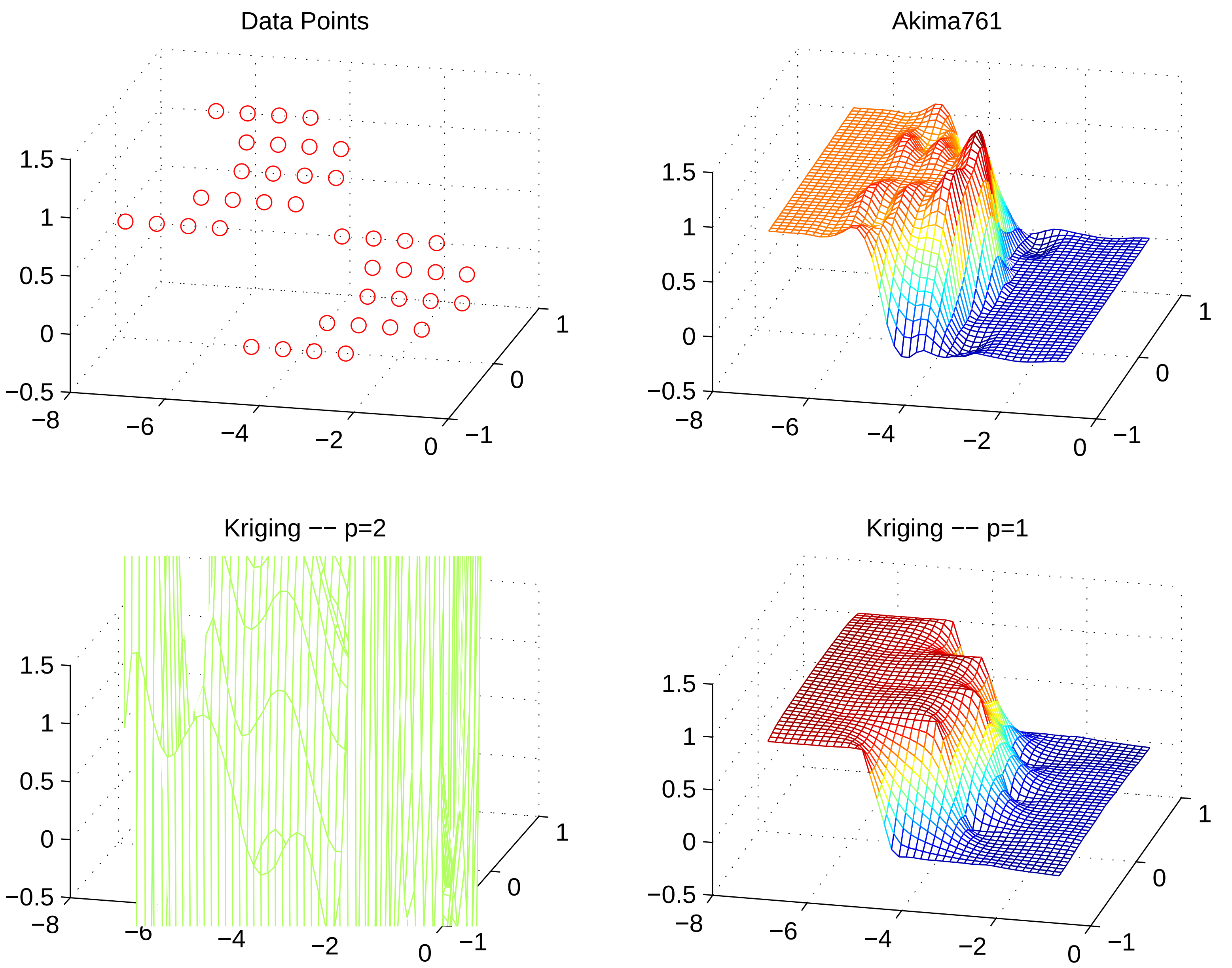


Data Points

Akima761

Kriging −− p=2

Kriging −− p=1



Data Points

Akima761

Kriging -- p=2

Kriging -- p=1

Kriging —

Kriging is an interpolation method that originated in the earth sciences to fit geological data. The field of study from which it evolved is called geostatistics, so kriging involves statistics and random function theory. Interestingly, the name kriging originated from one of the method's big proponents attributing the original idea to D. G. Krige, although it is now used in the lower case as both a noun and a verb.

Nevertheless, its biggest advantage is that it is very easy to implement in an arbitrary number of dimensions. The basic idea is to use a weighted linear combination of values at the sampled data locations to interpolate the function. What kriging tries to do is come up with the best linear estimator by minimizing the error of the estimation. Because the actual function is not known, the error is modeled with probability theory and then minimized, resulting in a linear system of equations for the weighting factors. These weights are different for each location at which the interpolation is performed.

Kriging has a few knobs you can twist to change the behavior of the fit. In fact, they are an entire function called the covariance. The covariance describes the probability that the function to be fitted has, at a given point, a value similar to known values nearby. Two covariance models are used in the examples here: the Gaussian (denoted as p=2) and the exponential (p=1). Both covariance models also vary in scale, i.e. they can be stretched out or squashed depending on the "range parameter," a.

Here are the equations used in the examples, where v_0 is the interpolated point at x_0 and the v_i's are values of the the n sample points taken at the locations x_i:

$$v_0 = \sum_{i=1}^{n} w_i v_i \tag{1}$$

$$\begin{bmatrix} C_{11} & C_{12} & \dots & C_{1n} & 1 \\ C_{21} & C_{22} & \dots & C_{2n} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ C_{11} & C_{12} & \dots & C_{nn} & 1 \\ 1 & 1 & \dots & 1 & 0 \end{bmatrix} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \\ \lambda \end{pmatrix} = \begin{pmatrix} C_{10} \\ C_{20} \\ \vdots \\ C_{n0} \\ 1 \end{pmatrix} \tag{2}$$

The weights, w_i, are found by solving the linear system (equation 2), and then are plugged into the first equation to find the value of the fitted function, v_0. (λ is the Lagrange multiplier resulting from the minimization of the fitting error that is in the the derivation of kriging.)

The C_{ij}'s are the covariances defined by:

$$C_{ij} = \exp\left\{ -1.5 \left(\frac{|x_i - x_j|}{a} \right)^p \right\}$$

The Euclidean norm in the covariance equation is the only place the number of dimensions appears. This is why this method is as easy to implement in one dimension as it is in 35 (or however many you want).

The basic theory of kriging and random functions can be found in a textbook by Edward H. Isaaks and R. Mohan Srivastava called *Applied Geostatistics*.

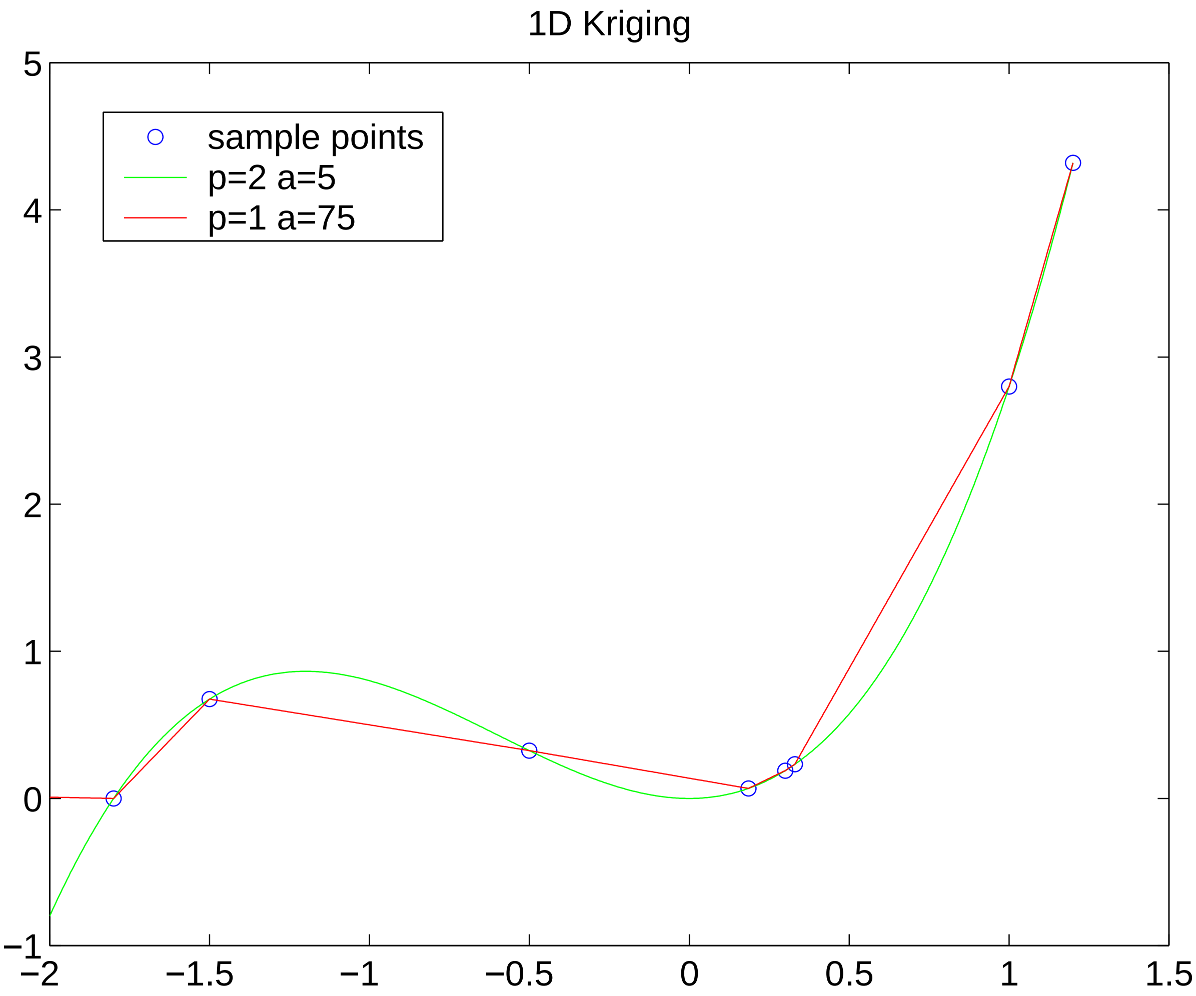


Figure 14: With an exponential covariance (p=1), the curve fit is linear in 1D and somewhat flat in higher dimensions. With a Gaussian covariance (p=2), the fit becomes equivalent to the cubic spline (only in 1D, but it is smooth in higher dimensions.)

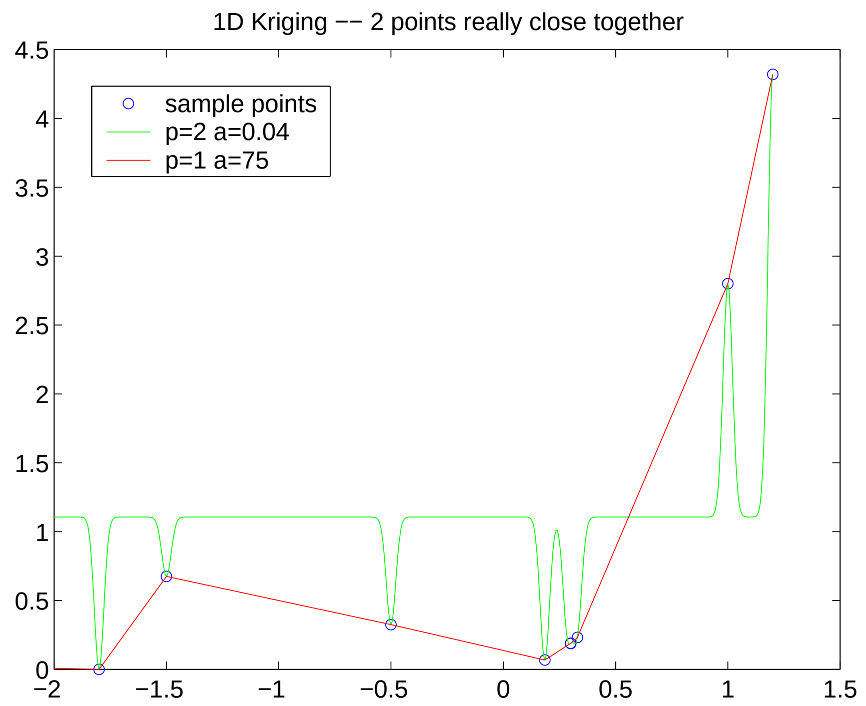


Figure 15: Same as the previous figure, but with two sample points taken very close together (superimposed on the plot). The linear system of equations becomes very close to singular in the p=2 case unless the range parameter, a, is reduced to very small values. The fit looks funny because most of the interpolated points are so distant from the sampled points that they just take the average value of all the sampled data (distance is scaled by the range parameter, so anything much greater than 0.04 units away is very far in this case).

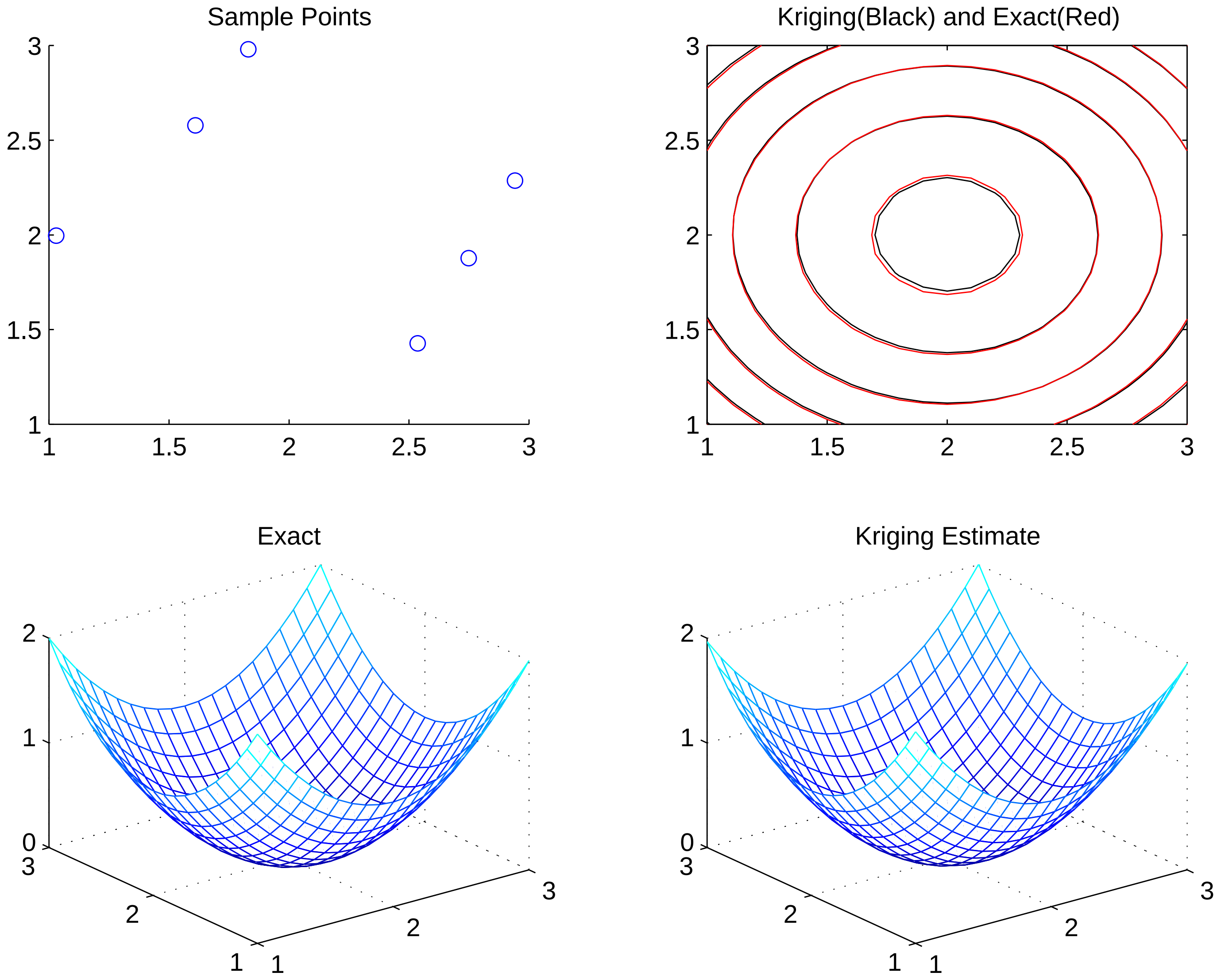


Figure 16: This is a kriging estimate of the parabola at bottom left evaluated a random set of six points at top left.

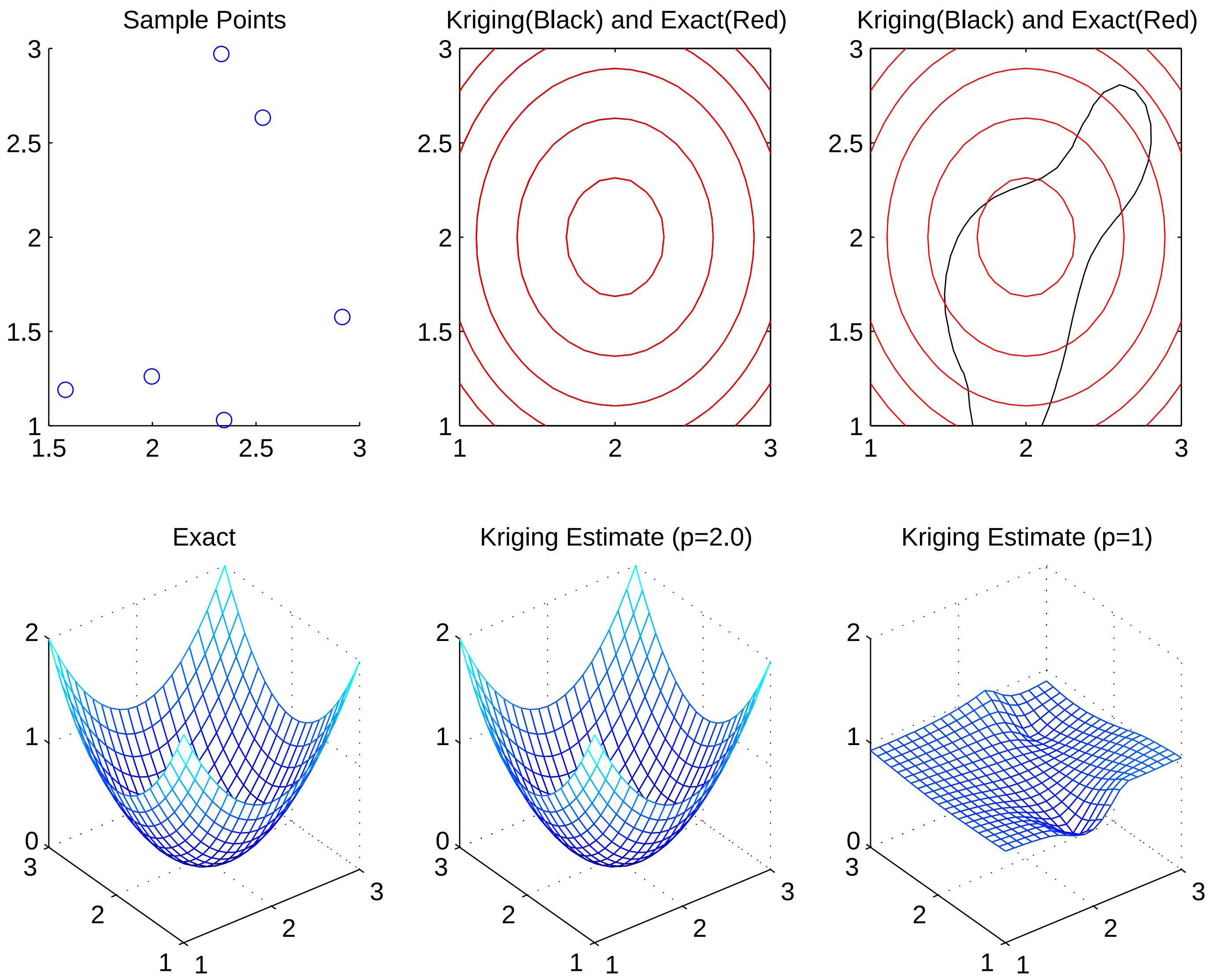


Figure 17: Same as the previous figure but also with the flatter p=1 fit. It doesn't result in anything like the parabola when evaluated at 6 random points.

Example — Optimization with kriging

The kriging estimate is used on a 2D problem to find the minimum of the Rosenbrock function. At each iteration, three points are evaluated and a kriging fit is made. The minimum of the fit within a trust region then becomes the center of the next iteration.

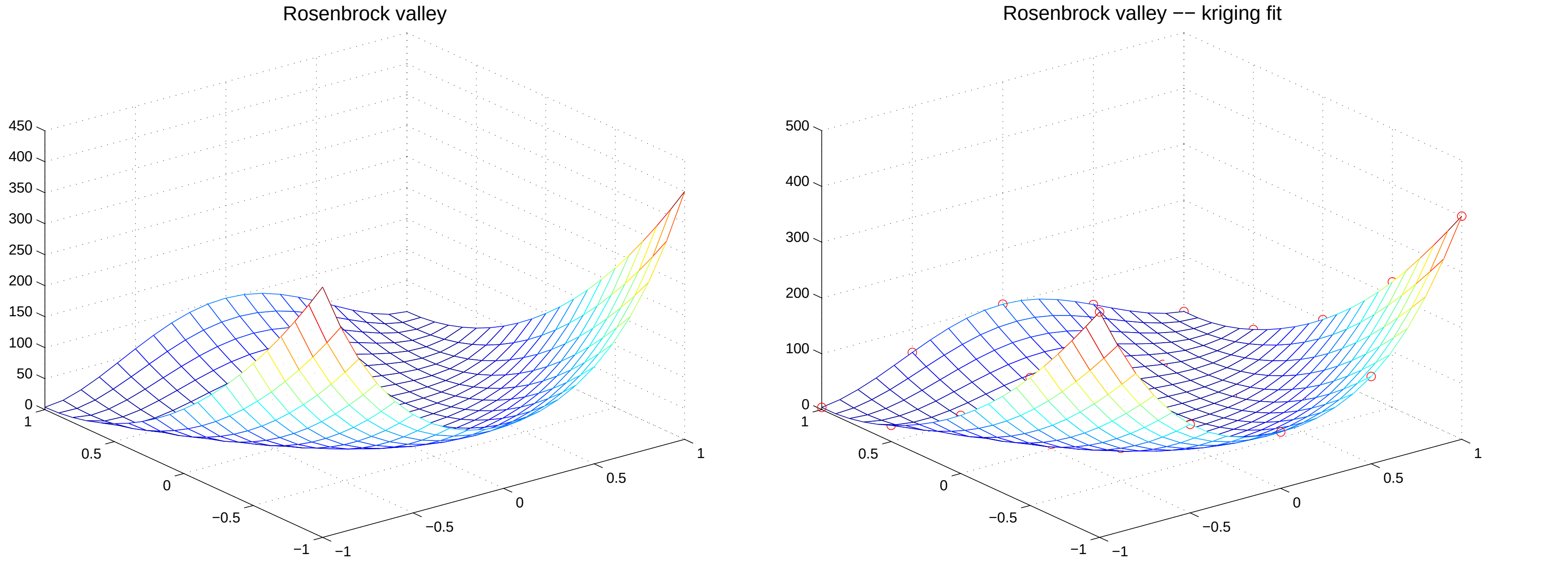


Figure 18: With 25 points and p=2, the kriging fit (at right) looks pretty good.

The following set of figures step through almost every iteration in the optimization.

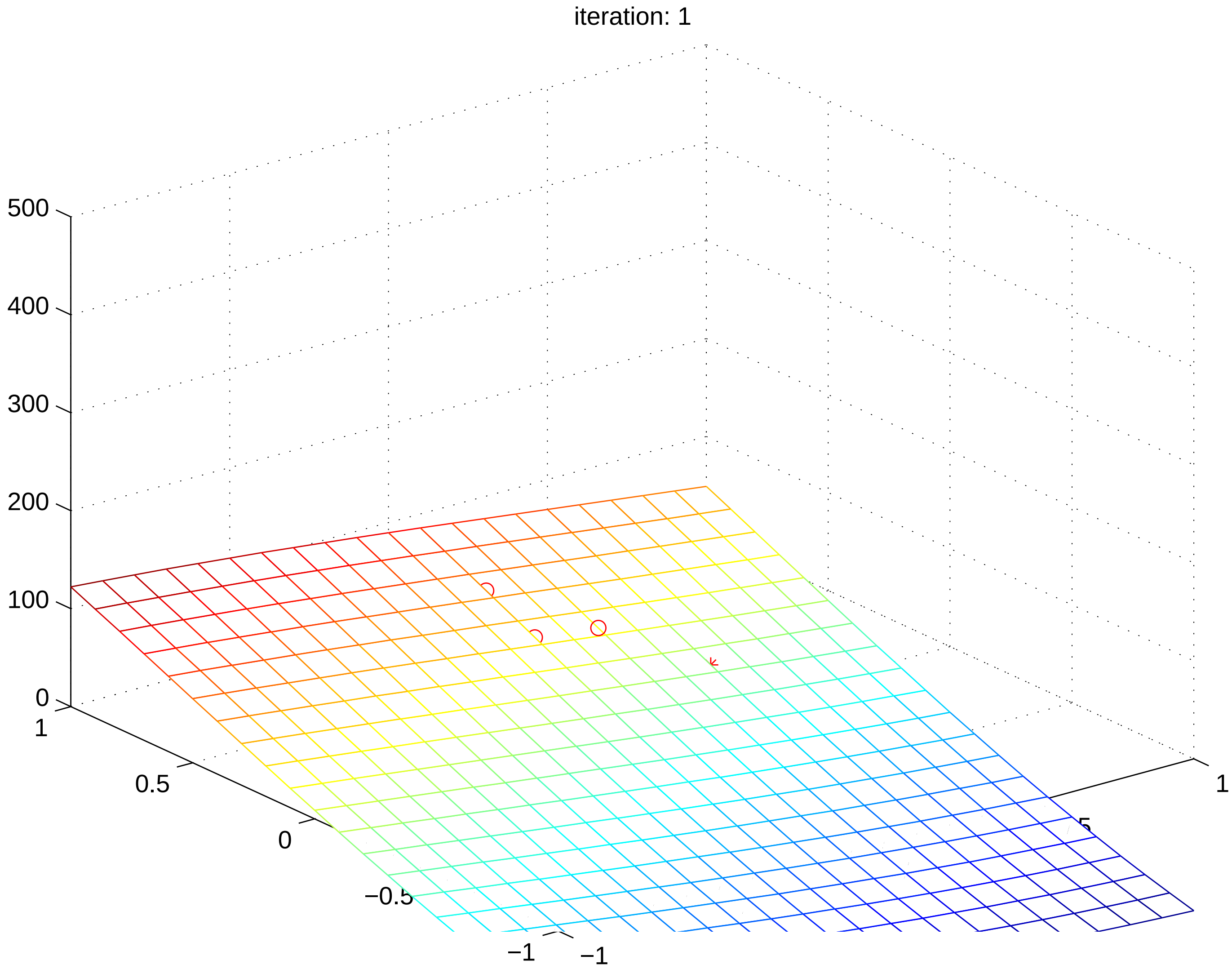
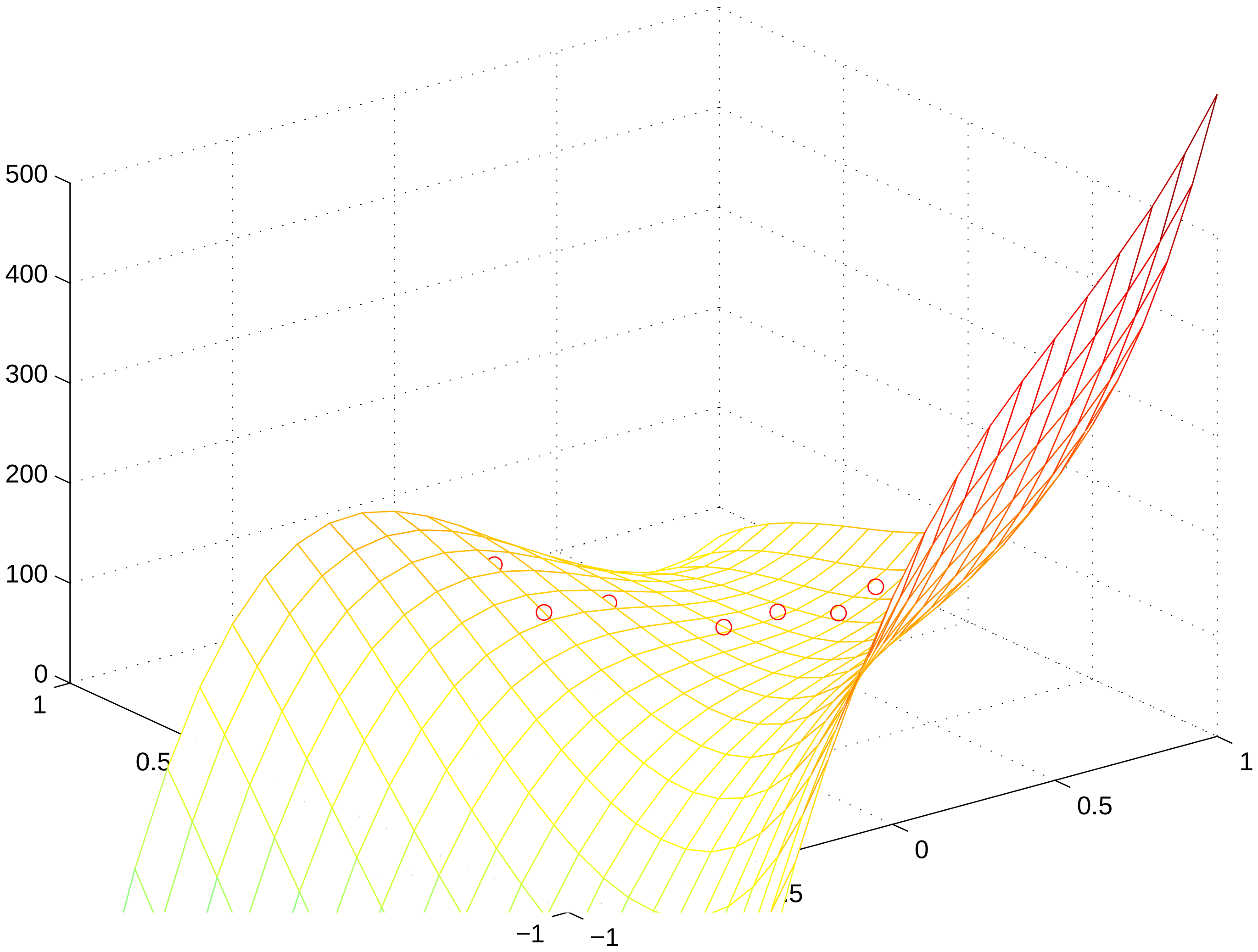


Figure 19: Here is the first iteration: 3 points (the circles) results in a planar fit. The minimum within the trust region (not plotted) is the point that shows up as a red asterisk.

iteration: 3



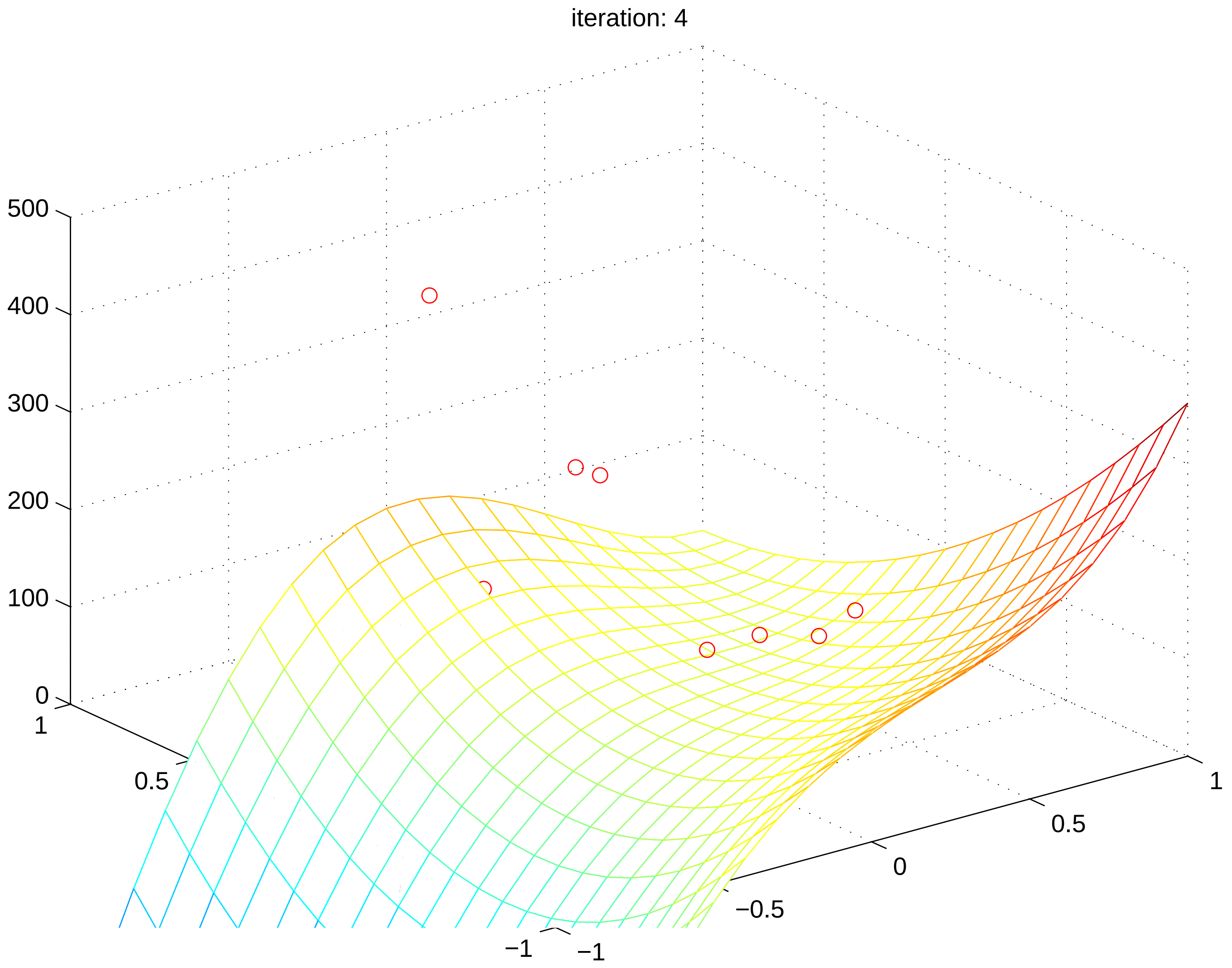
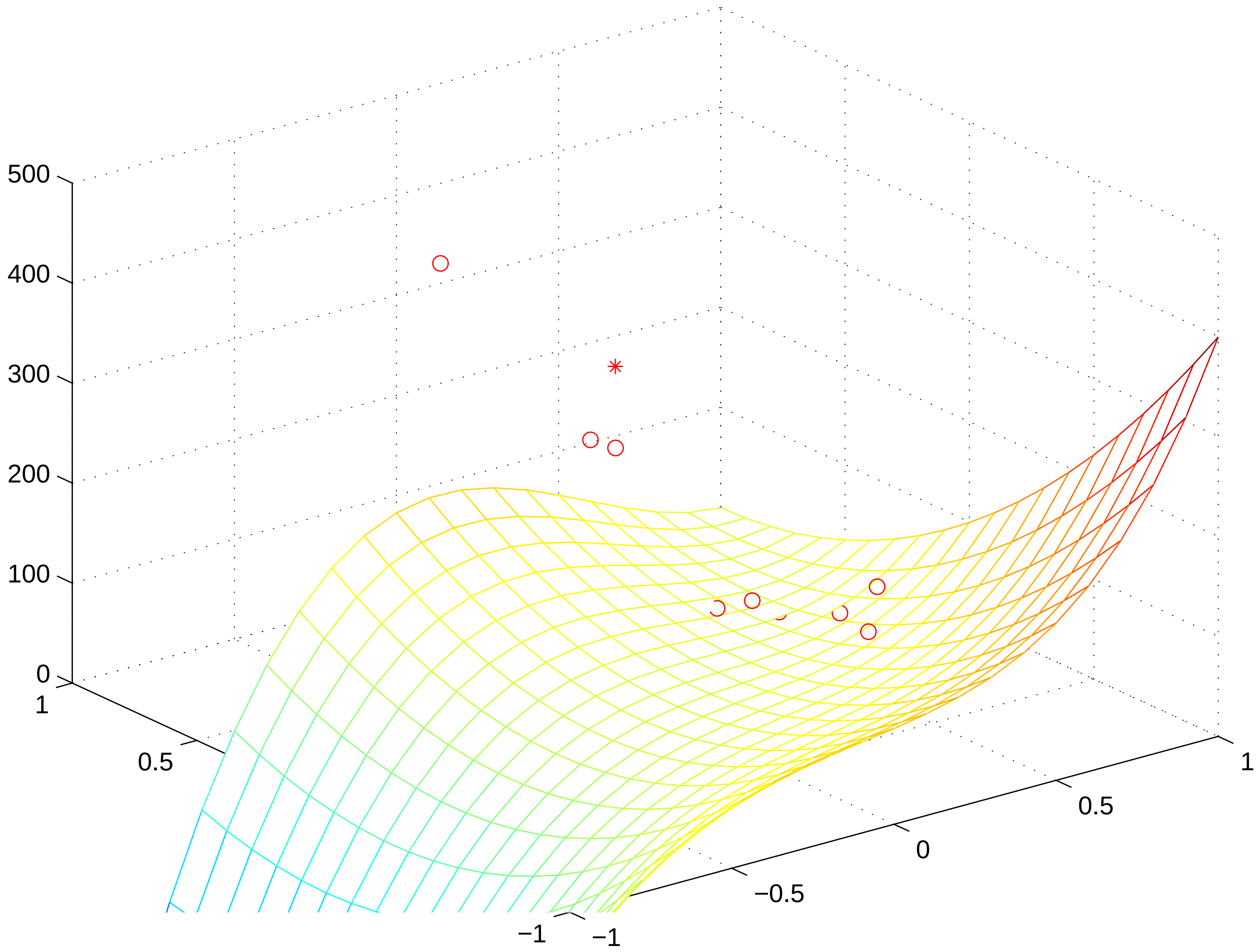
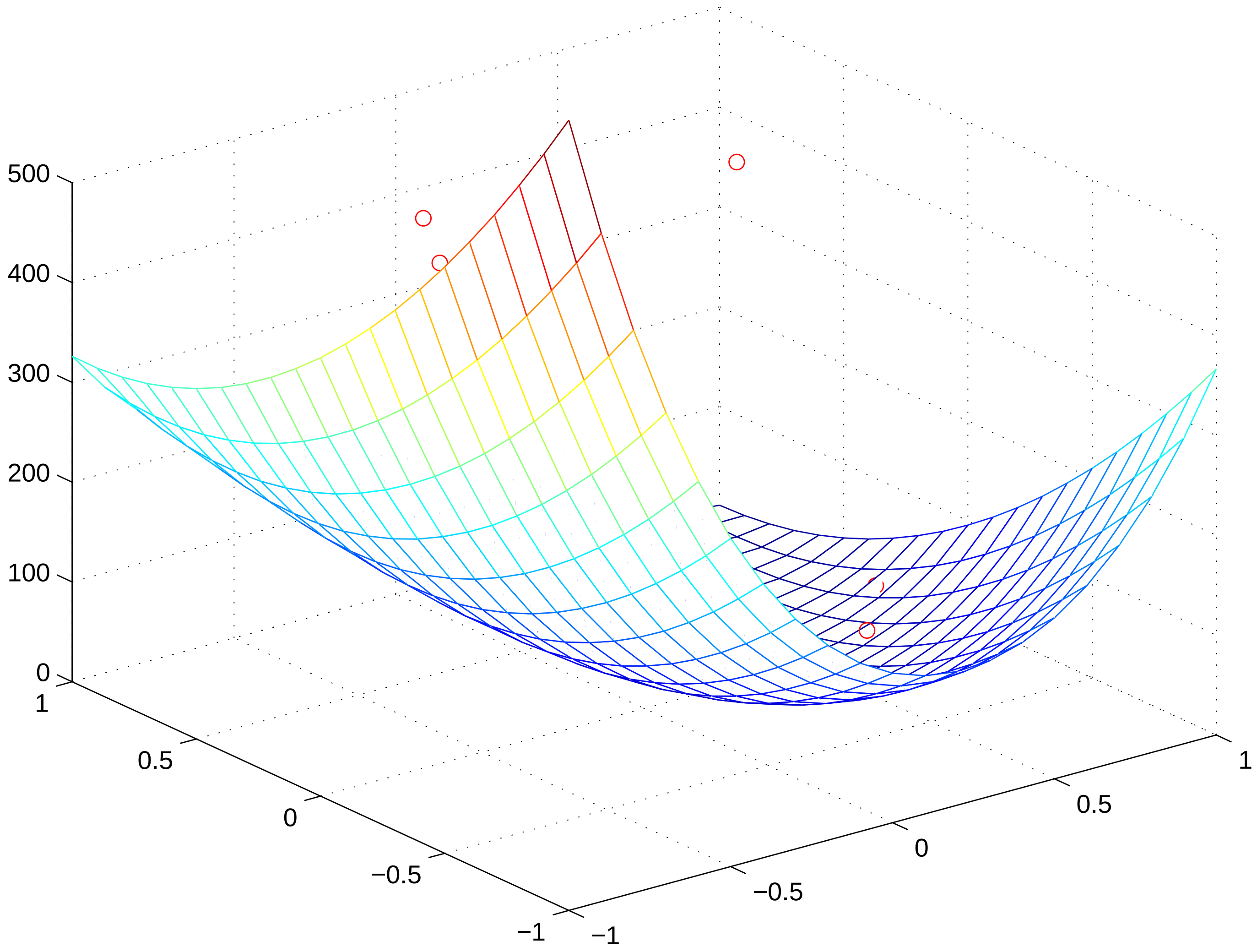


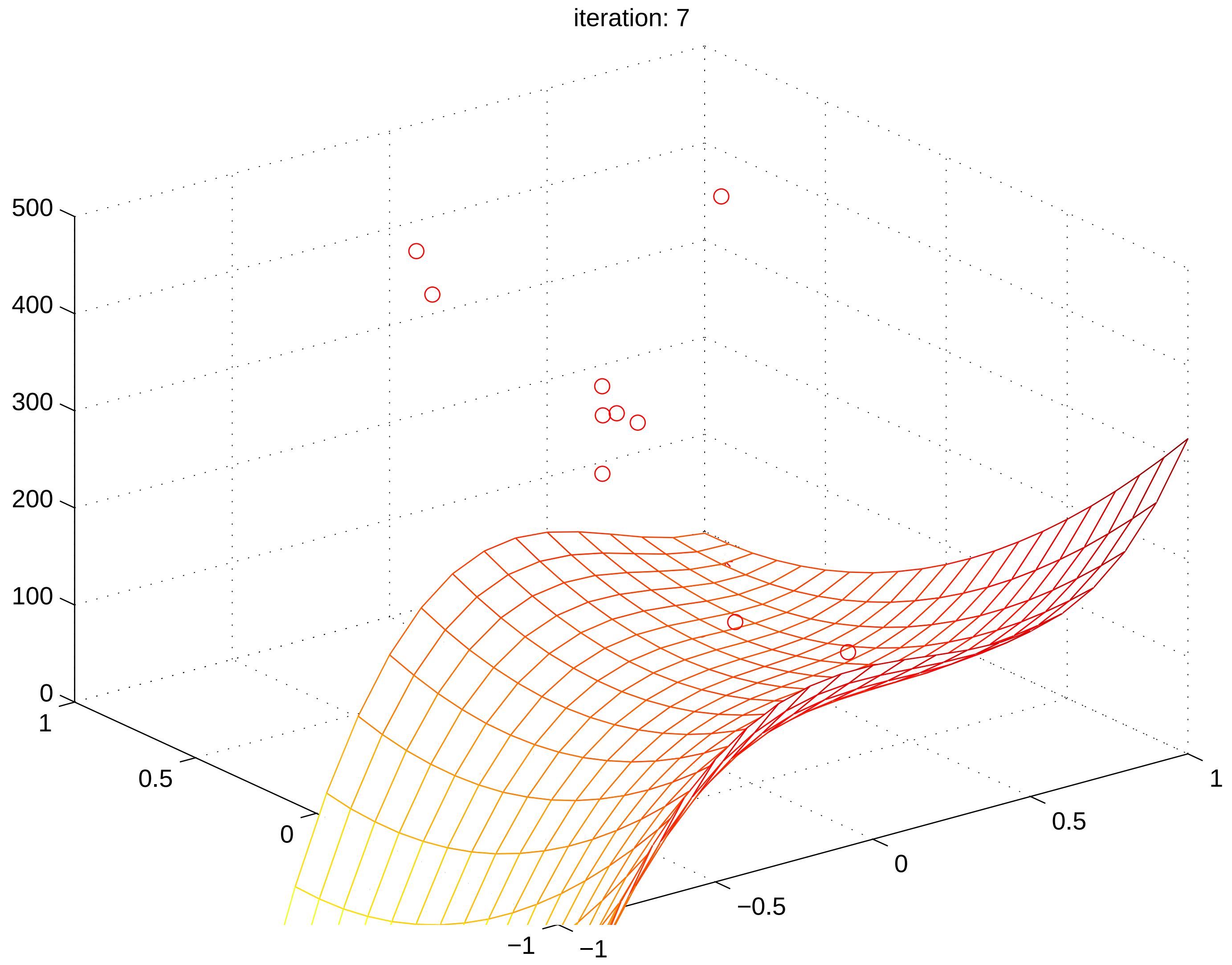
Figure 20: The points that look like they're floating in space are actually on the fit, they are just beyond the box in which Matlab computed the mesh.

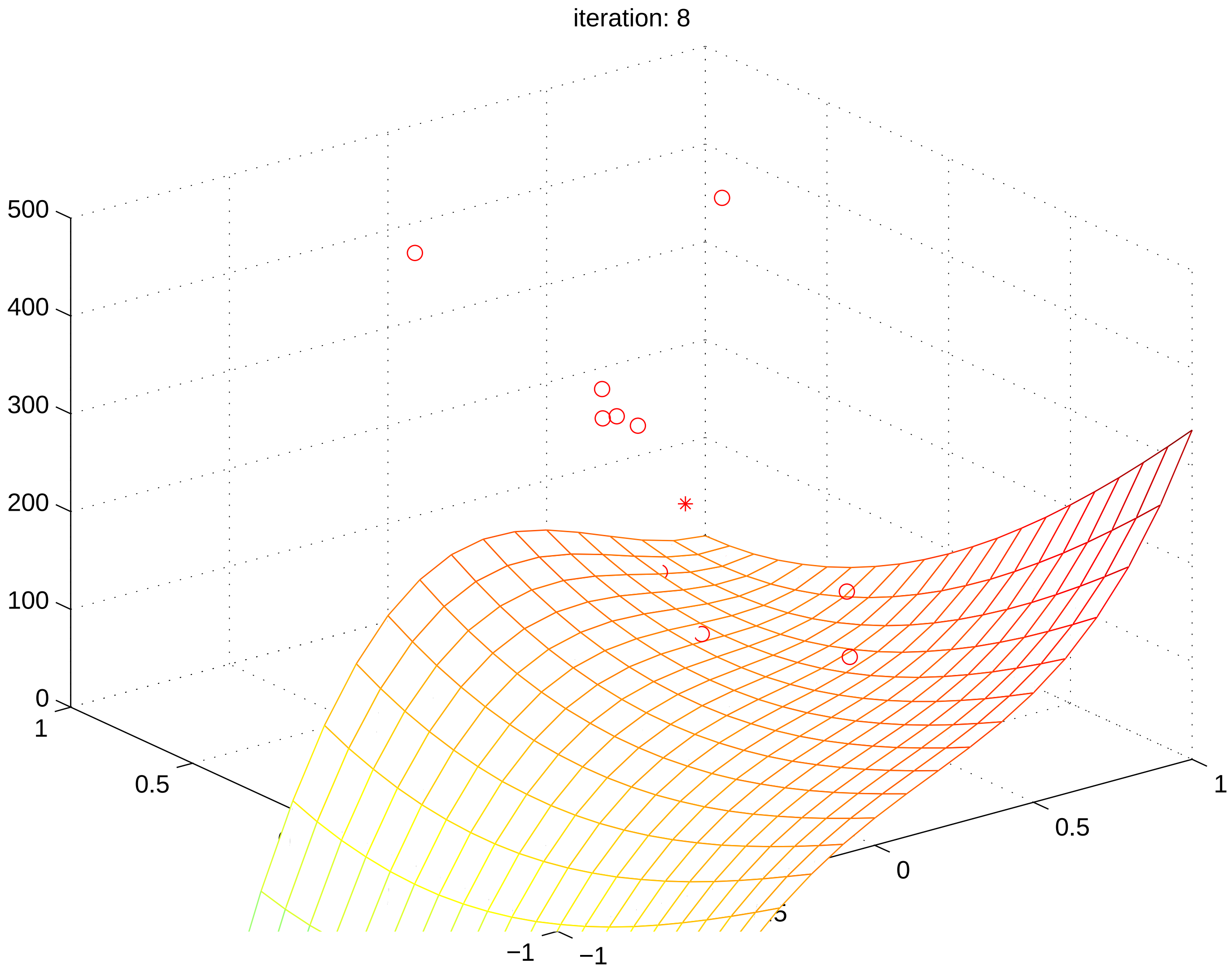
iteration: 5



iteration: 6




iteration: 7
500
400
300
200
100
0
1
0.5
0
−1
−1
−0.5
0
0.5
1


iteration: 8
500
400
300
200
100
0
1
0.5
−1
−1
1
0.5
0

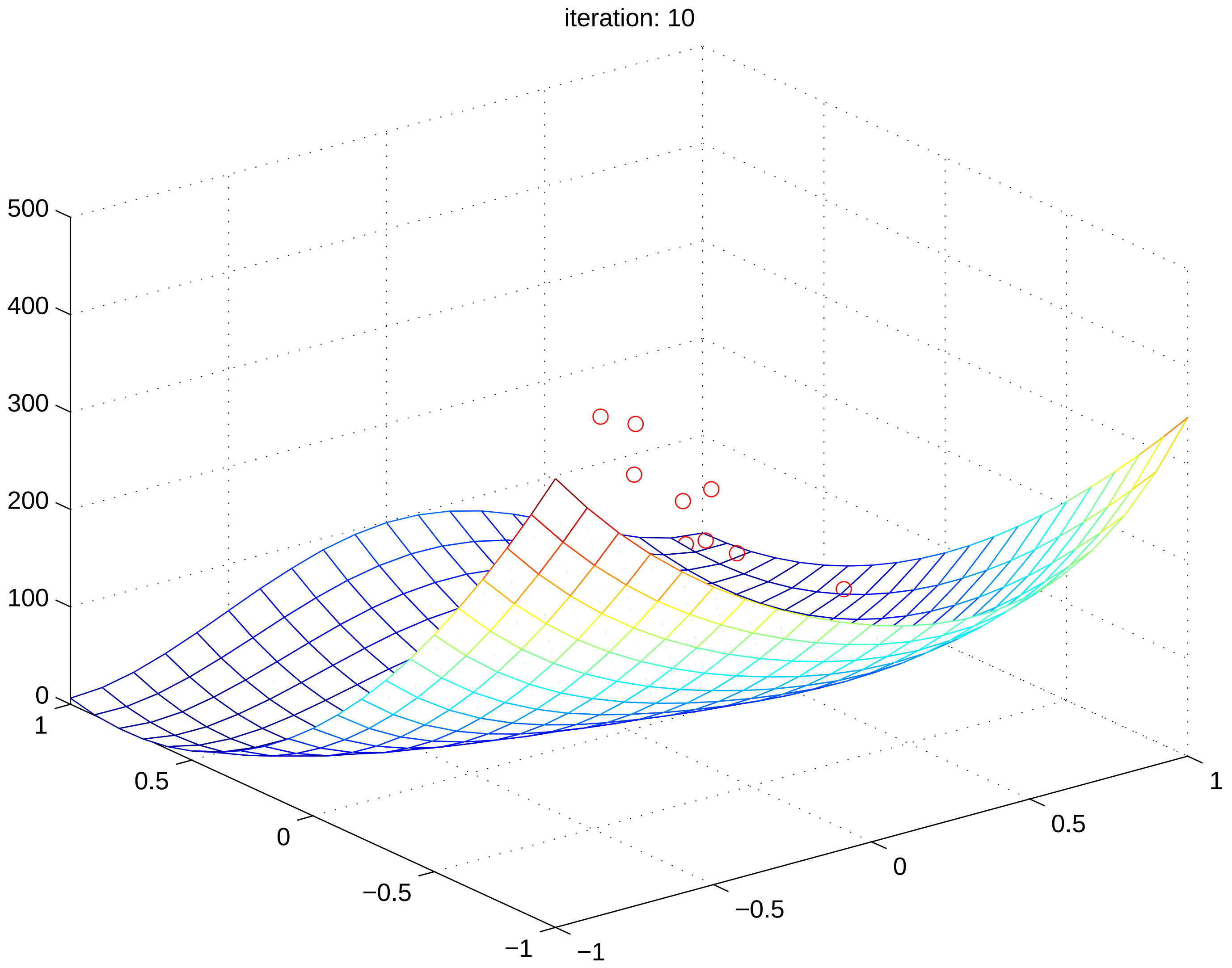
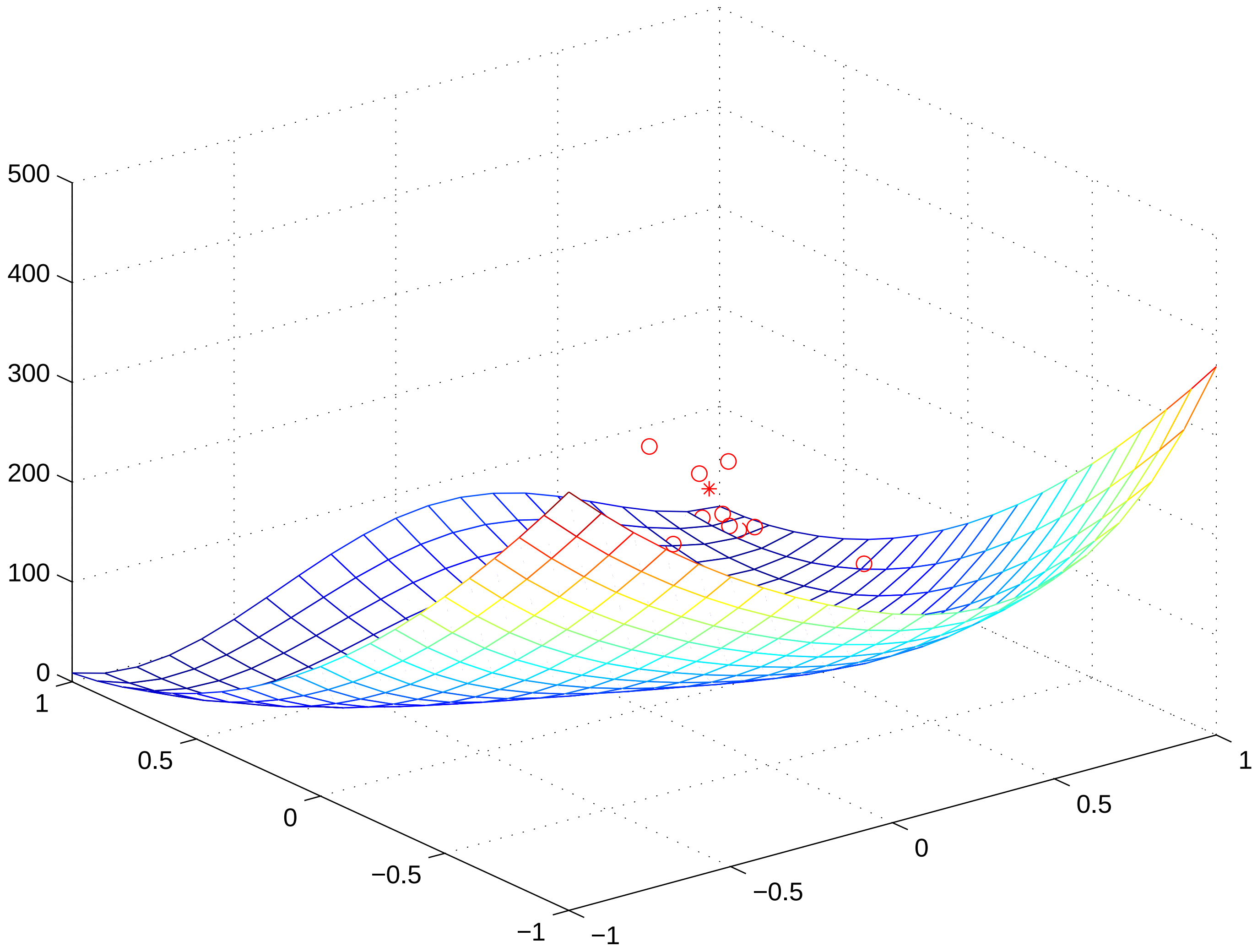


Figure 21: The fit at this and the next iteration now resembles the Rosenbrock function fairly closely.

iteration: 11



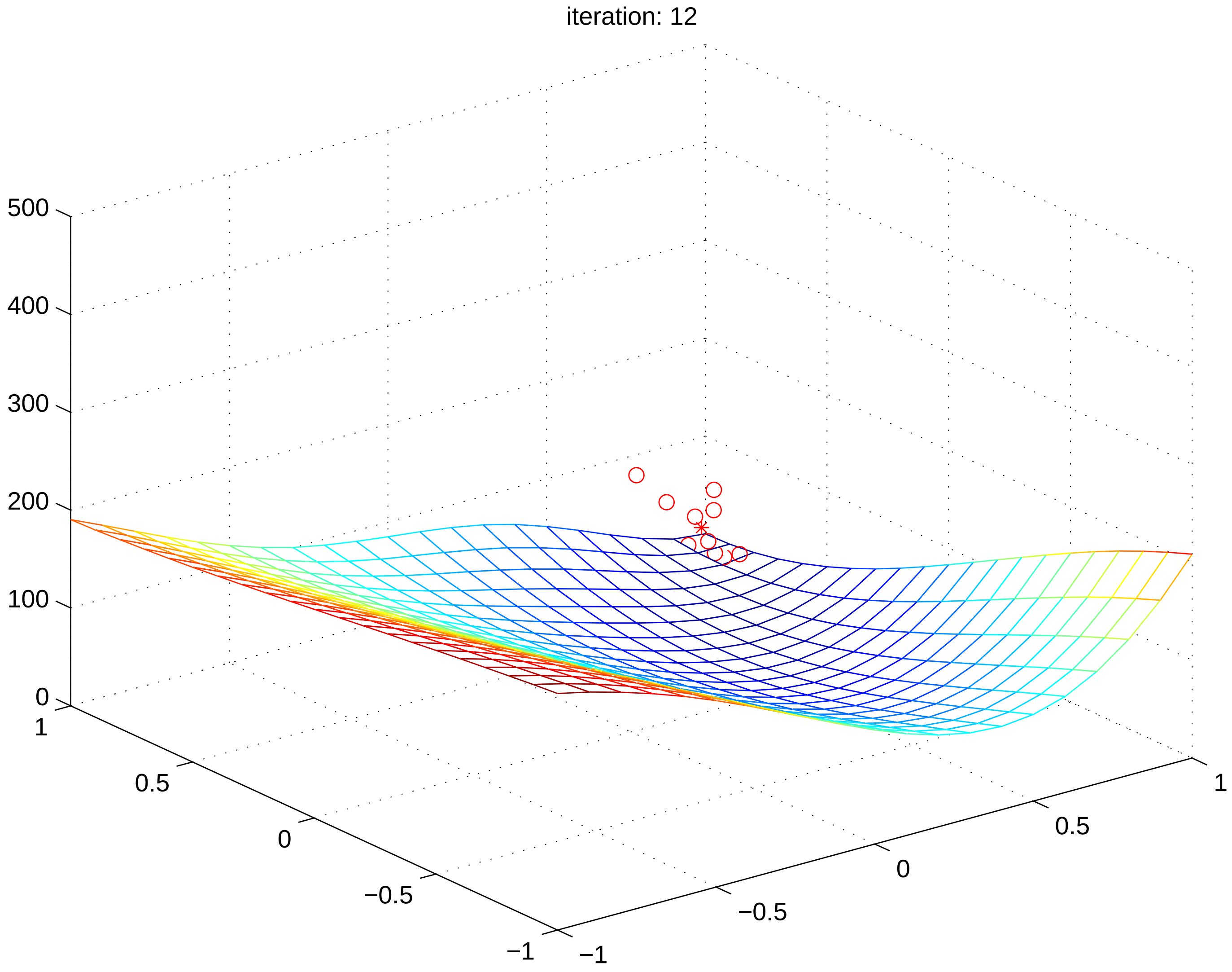
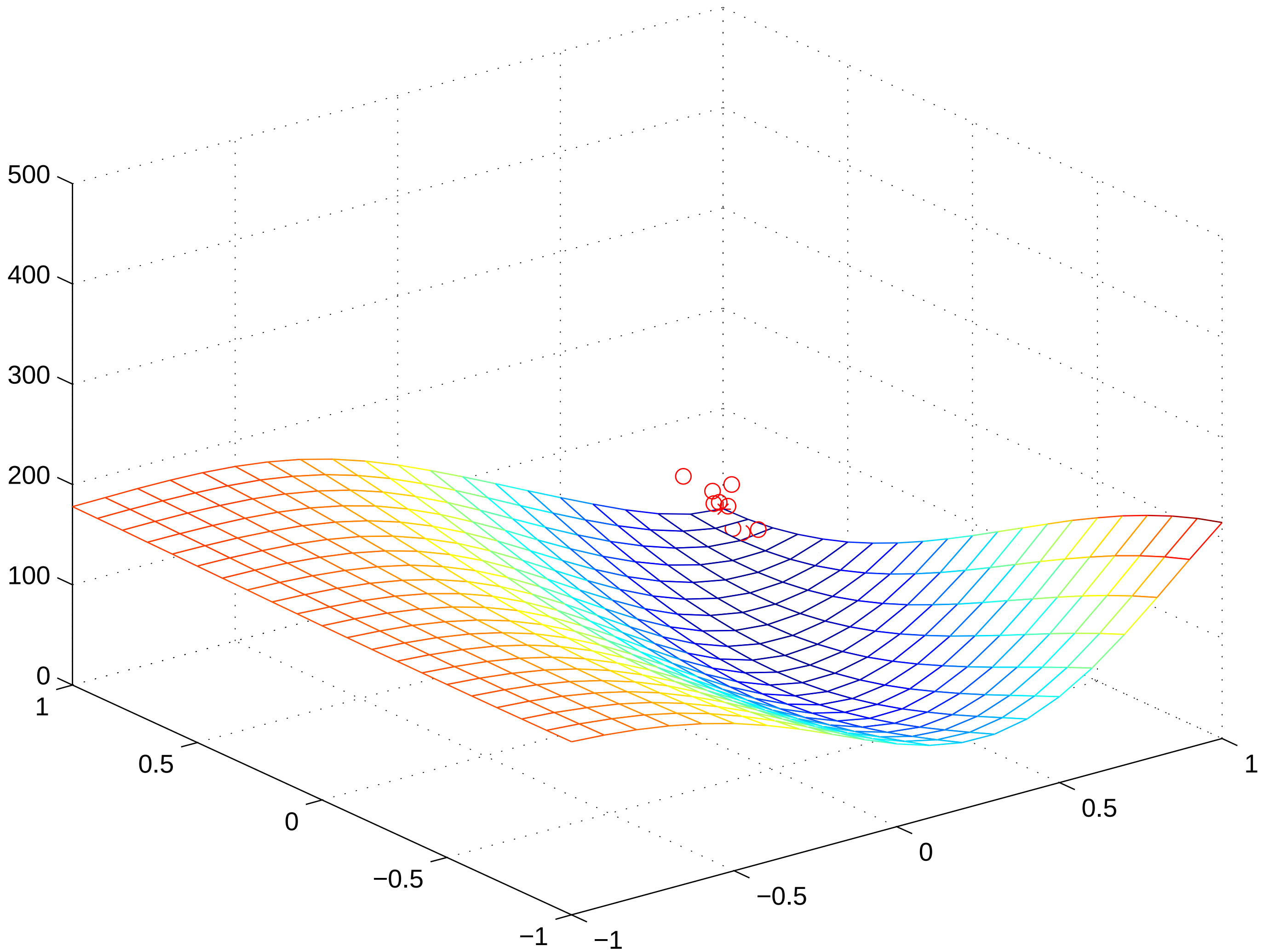
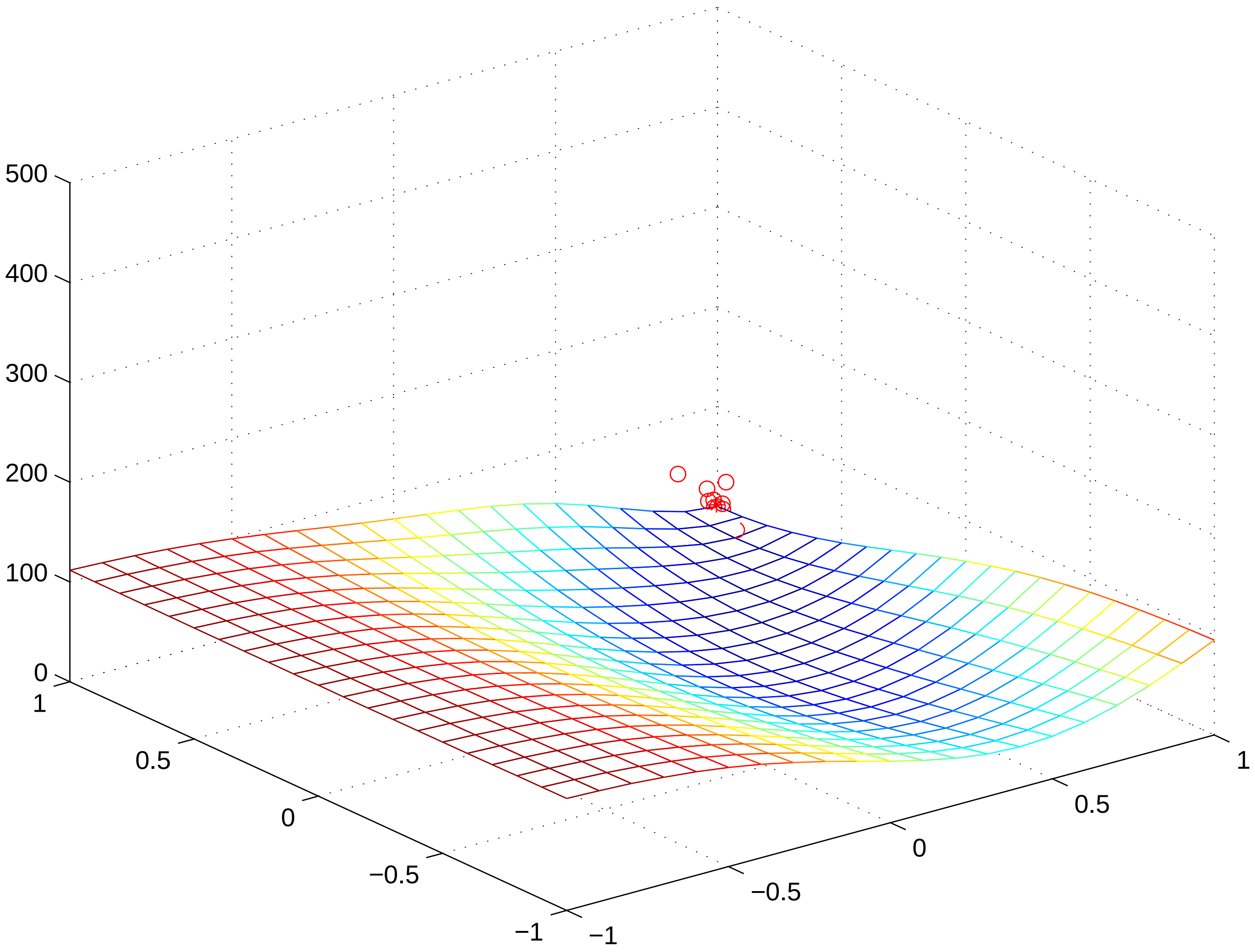


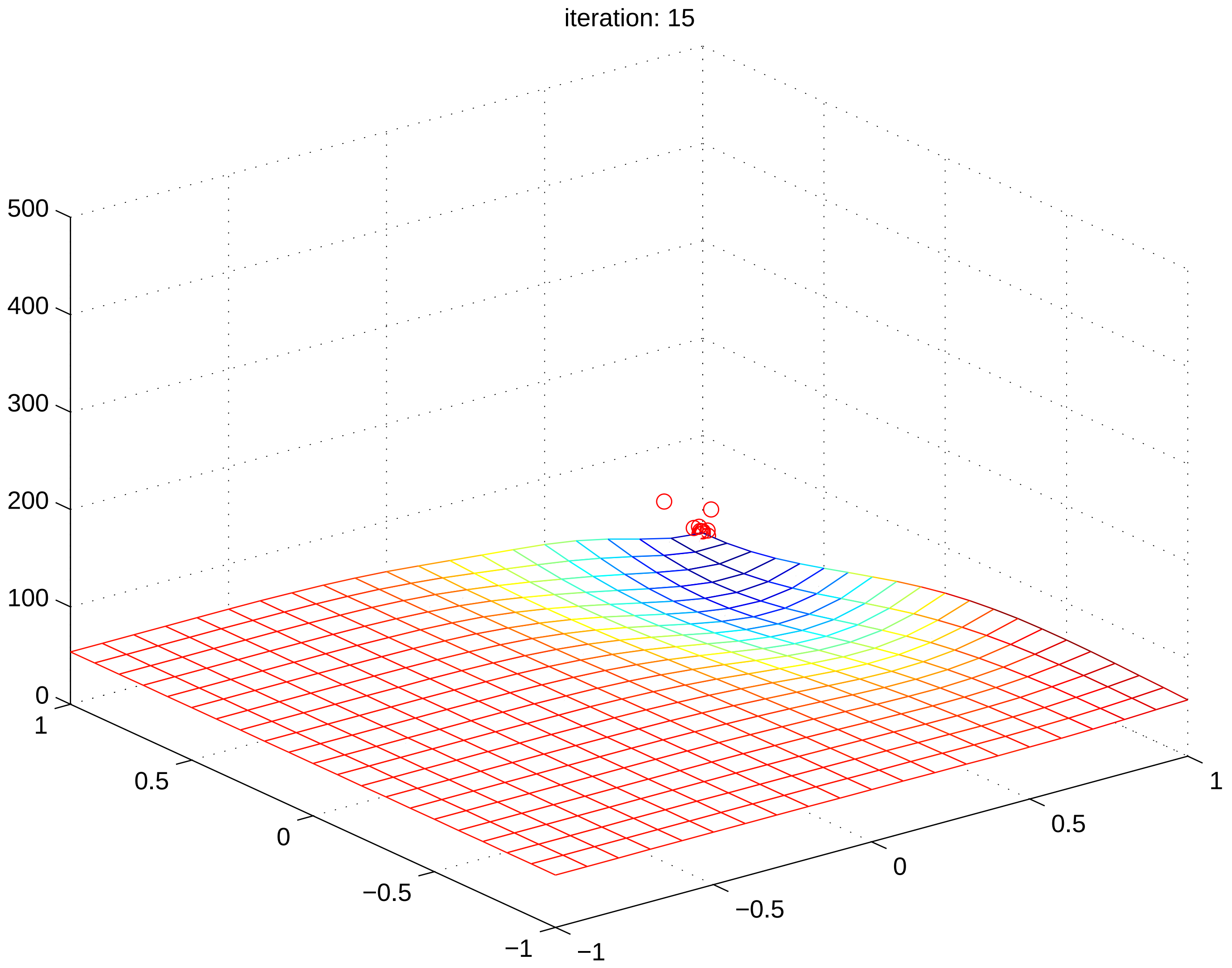
Figure 22: Here the fit is good near the solution, but is starting to lose accuracy farther away because not all the points from the previous iterations are kept.

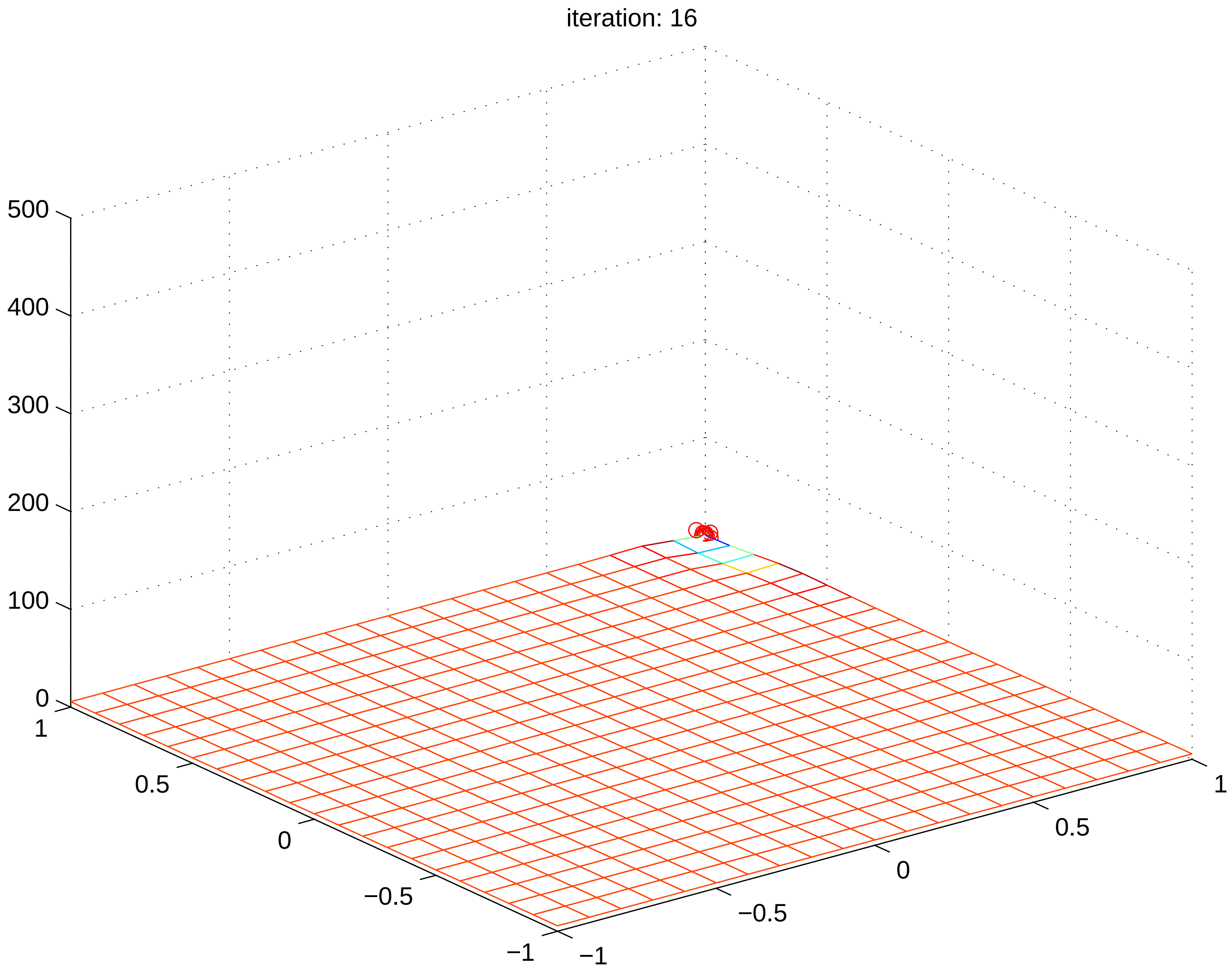
iteration: 13



iteration: 14




iteration: 15
500
400
300
200
100
0
1
0.5
0
−0.5
−1
−1
−0.5
0
0.5
1


iteration: 16
500
400
300
200
100
0
1
0.5
0
−0.5
−1
−1
−0.5
0
0.5
1

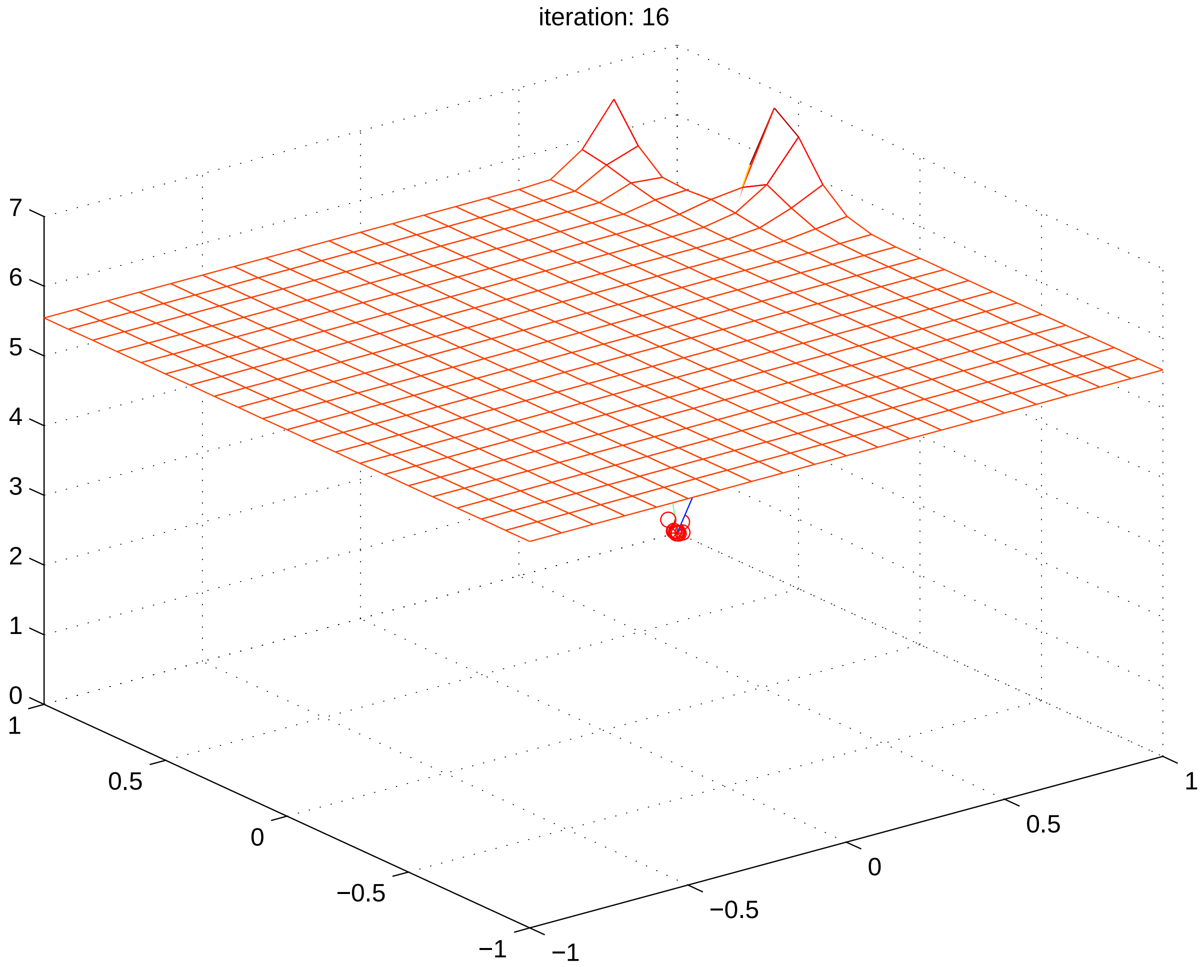


Figure 23: And the minimum is found. Note the cluster of points around the answer (1,1). Also note that all the points throughout the optimization were not kept. A maximum of 11 points were kept to prevent the kriging linear system from becoming singular when this tight cluster of points is mixed with the wide-spread cloud of points that it took to get here.