Lab #6 for Geophysical Inverse Theory ESS523, Fall 2005, Univ. of WA
TA presenting this lecture : Andy Ganse
Course professor : Ken Creager

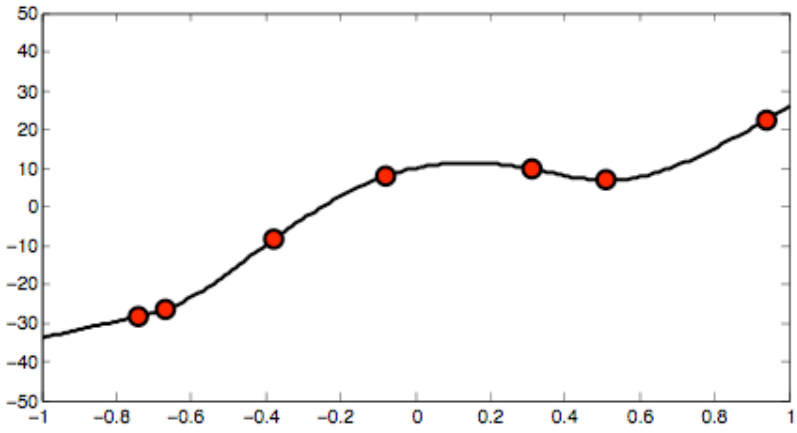
22 Nov 2005

A cubic spline curve-fitting example to demonstrate the use of a smoothing matrix in higher order Tikhonov regularization (e.g. as in Occam's and similar inversion methods).

We've learned that in inverse problems we desire to estimate a continuous model function based on a finite number of data points. This is an inherently underdetermined problem – without regularization, i.e. additional constraints on the model function, there are an infinite number of equally valid solutions. Different inversion methods use different variations of regularization. A commonly used one that's useful to know is called "Occam's inversion" (after Occam's Razor in philosophy, stating that all else being equal, the simplest answer is the best one – adapted for inversion in Constable, Parker, and Constable, in *Geophysics*, 1987). In Occam's inversion and similar methods, from those infinity solutions we choose the simplest one, meaning the one with the fewest model function features (like jumps and wiggles) that are required to still fit the data to within its noise. The regularization filters out undesirable model functions by using a model norm based on their 2nd derivative, so that if we minimize the norm, we minimize the curvature in the model function, making it as smooth as allowable by the data and data noise. In this lab we will learn how to set up such an inversion for our own projects by setting up a simple special case that estimates a cubic spline curve fit for a set of data points on a graph.

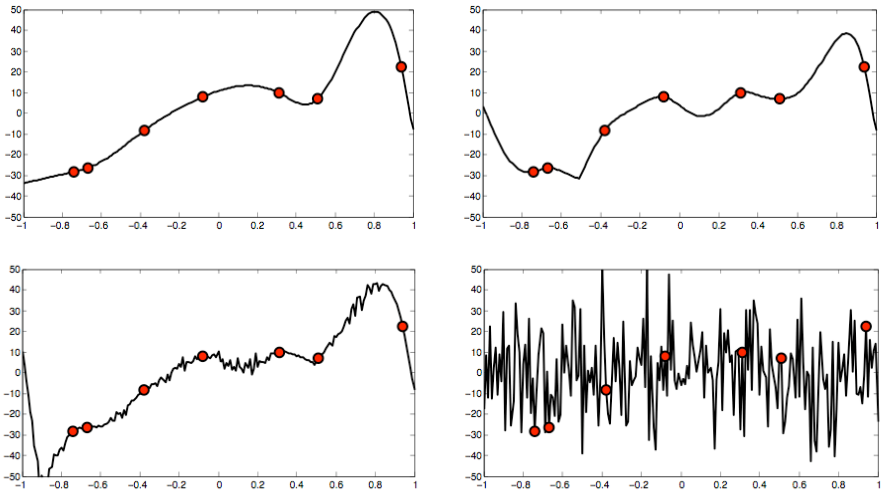
What is a cubic spline?

A cubic spline is defined as the smoothest curve that exactly fits a set of data points. So spline curves do not consider noise on the data points, although in this lab we will use a very small amount of noise to simulate an exact fit while setting up the technique for more general inverse problems.



This problem today is a special case of a linear inverse problem, because the data space and the model space are the same – the data points and the model function can be plotted on the same graph. Hopefully this will make more clear the concepts of model smoothness and associated uniqueness. We'll see shortly why it's a linear inverse problem.

Note that for a finite set of data points there are all sorts of (in fact an infinite number of) possible continuous curve functions that fit all the data points exactly. For example, using the data points from the previous plot in repeated cases:



These functions may not all be pretty looking, but they all fit the data points exactly. Without further constraints, who's to say which curve is the "real" one? Notice here that in the type of curve fitting of today's lab (and in your own inverse problems) we are not determining ahead of time some coarse parameterization of the continuous model function such as by a polynomial of degree *N*. Instead, by choosing the constraint of "minimum 2nd derivative norm", we let the data points and the problem determine the amount of "wiggliness" of the curve by choosing the smoothest one that fits the data.

Now after just saying "we don't use a polynomial to do this", I should clarify where the word "cubic" comes into the name "cubic spline". It turns out that in meeting the definition I gave already, a cubic spline curve happens to be a string of concatenated cubic function segments, joined at each data point x value in such a way that the 2nd derivative doesn't change across the join point. The "cubic" part is not immediately obvious – it relates to the 2nd derivatives and can be seen in the development given in Parker's book. In fact, computer programs that use spline curves compute them in a more efficient way than we will here, by directly calculating each cubic segment. But that wouldn't help us learn how to set up an Occam's inversion! Lastly on this point, Parker develops an inversion to find a cubic spline curve in his book using the Gram matrix technique, and obtains the exact continuous spline function. But we'll use a different technique that is approximate but also more applicable to your other geophysical inverse problems.

Both the Parker book version and today's version of the problem are linear inverse problems (or purists might call our approach today a "discrete linear inverse problem"). In Parker's development, you want to estimate the coefficients of the representer functions, which are linearly parameterized with those coefficients. In today's lab it's even easier; we simply want the model curve equal to the data point values when at the data point x locations, and we specify this with a matrix of zeros and ones. Then the regularization constrains what goes between the data points. Let's look at that now.

Defining the A matrix:

In our method, given a set of data points at locations x_i, we will estimate the continuous spline curve as a finely discretized sequence of points at equally spaced locations xx_j. We want the points xx_j in the model curve to equal x_i at the appropriate locations, and we can specify that with a simple linear relation between the two using a matrix of zeros and ones:

$$\mathbf{d} = \mathbf{A}\,\mathbf{m} \quad \Longrightarrow \quad d_i = f(x_i),\ m_j = f(xx_j)$$

$$\begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_n) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & & 0 \\ \vdots & & & & & & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} f(xx_1) \\ f(xx_2) \\ f(xx_3) \\ f(xx_4) \\ f(xx_5) \\ f(xx_6) \\ \vdots \\ f(xx_m) \end{bmatrix}$$

This **A** matrix only controls information at the data point locations; it's zero everywhere else so it's a pretty sparse matrix. So **A** is non-invertible – it will require regularization such as the higher-order Tikhonov approach we will do with the smoothing matrix **L**. By the way, if we wished, we could have instead specified data points at x locations that were not on the xx grid. In that case we could design a slightly different (but still sparse) **A** that would constrain the data point values to be linear interpolations between the neighboring two $f(xx)$ values. But that's really tangential to our use of this lab as an example for setting up our own inversion projects; let's stick with the 0's and 1's for this lab assignment.

Defining the L matrix:

We need a mechanism to compute the 2nd derivative of the model curve function. As seen in class, rather than analytically compute expressions for the 2nd derivative of the model curve (as Parker does in his book) we will use a 2nd order finite difference operator matrix **L**. The product **Lm** (recall from above that **m** is the finely discretized model curve vector with $m_j=f(xx_j)$) approximates the 2nd derivative of the model curve. This **L**

matrix takes one of the two similar forms below. Most of either matrix is zeros, so for clarity in displaying them the zeros are left as blanks:

$$\mathbf{L} = \frac{1}{(\Delta xx)^2}\begin{bmatrix} a & b & c & d & & & & & \\ 1 & -2 & 1 & & & & & & \\ & 1 & -2 & 1 & & & & & \\ & & 1 & -2 & 1 & & & & \\ & & & & \ddots & & & & \\ & & & & 1 & -2 & 1 & & \\ & & & & & 1 & -2 & 1 & \\ & & & & d & c & b & a & \end{bmatrix} \quad m\text{x}m \text{ matrix}$$

or

$$\mathbf{L} = \frac{1}{(\Delta xx)^2}\begin{bmatrix} 1 & -2 & 1 & & & & & & \\ & 1 & -2 & 1 & & & & & \\ & & 1 & -2 & 1 & & & & \\ & & & & \ddots & & & & \\ & & & & 1 & -2 & 1 & & \\ & & & & & 1 & -2 & 1 & \end{bmatrix} \quad (m-2)\text{x}m \text{ matrix}$$

The difference between these two forms of **L** is in having (or not) the top and bottom rows to make a square matrix – note the upper form is square while the lower form is not. Since they both have *m* columns they both can multiply the model curve vector **m**, but the resulting product vectors will differ in length by two elements. Let's come back to these top and bottom (or missing) rows in a moment; first let's discuss the common features to both of these matrices.

L in either form is normalized by the square of the step size Δxx, and follows the definition of 2nd finite differences. It is required that the xx_j are equally spaced here. Recall the finite difference discussion from Lab #4; the approximation of a first derivative in terms of forward and central differences can be written as below. This is slightly different notation than we used in Lab #4 but says the same thing:

$$1^{\text{st}} \text{ forward diff}, \ \mathcal{O}\{\Delta xx\} \qquad f'_j \approx \frac{f_j - f_{j+1}}{\Delta xx}$$

$$1^{\text{st}} \text{ central diff}, \ \mathcal{O}\{(\Delta xx)^2\} \qquad f'_j \approx \frac{f_{j+1} - f_{j-1}}{2\Delta xx}$$

The $\mathcal{O}\{\Delta xx\}$ refers to the accuracy of the finite difference approximation, based on the truncation of the Taylor series that it was derived from (you'll find pertinent details and a derivation in your favorite numerical methods textbook). Notice from the $\mathcal{O}\{\}$ that the central difference is more accurate than the forward difference, as was also mentioned in Lab #4.

Similarly, there is a finite difference approximation to the second derivative in terms of forward and central differences:

2^{nd} forward diff, $\mathcal{O}\{\Delta xx\}$

$$f_j'' \approx \frac{f_j - 2f_{j+1} + f_{j+2}}{(\Delta xx)^2}$$

2^{nd} central diff, $\mathcal{O}\{(\Delta xx)^2\}$

$$f_j'' \approx \frac{f_{j-1} - 2f_j + f_{j+1}}{(\Delta xx)^2}$$

We can see from the coefficients of these expressions where the rows of {1, -2, 1} enter into the matrix **L**, which multiplies the vector **m** composed of $m_j = f_j = f(xx_j)$. We also see that step size normalization factor in the denominator.

Now back to the top and bottom rows. The Aster/Borchers/Thurber book uses the second form for **L** here; our professor Ken uses the first form. They have differing numbers of rows but the same number of columns, so they both can work in our inversion formulas since only the product $\mathbf{L}^T\mathbf{L}$ is used (as seen in class and recalled shortly below), which is a square matrix of the same length as the model vector **m**. Notice that the central difference formulation relies on the neighboring points on either side to compute the approximate derivative. That's no problem except for at the edges of the **L** matrix, which correspond to the endpoints of the **m** vector that we are taking derivatives of. So what does one use for the non-existent neighboring point to approximate the 2^{nd} derivative at the edges?

There are several ways to address that problem. The first is the Aster/Borchers/Thurber approach which doesn't approximate the derivative at the edges at all, so that the 2^{nd} derivative vector is shorter than the model vector by one point on each side. In the L2-norm based inversion schemes one only uses $\mathbf{L}^T\mathbf{L}$ so that this difference in length doesn't matter. But maybe sometime you'll want to use some other norm like the L1-norm (not discussed in this lab) which doesn't use $\mathbf{L}^T\mathbf{L}$, or have some situation where you want to compute a 2^{nd} derivative approximation of the same length as the model vector. Or maybe you want to guarantee that $\mathbf{L}^T\mathbf{L}$ is invertible – it's not in this non-square **L** case – so that you can guarantee that $(\mathbf{A}^T\mathbf{A}+\nu^2\,\mathbf{L}^T\mathbf{L})$ in the normal equations will be invertible for an appropriately sized ν. (In practice this often may not be a problem however.) In those cases you'll need to add a top and bottom row to **L** and fill in the *a,b,c,d* elements according to your needs. Below in the table are some possibilities for *a,b,c,d* choices, but they are not the only choices you could do. Meanwhile, note that in this lab assignment we will only use the Aster/Borcher/Thurber version of **L**, i.e. the non-square one without the extra top and bottom rows.

Some possibilities for *a,b,c,d* in the top and bottom rows of the L matrix:

{a, b, c, d}	Description
{1, -2, 1, 0}	Use the 2nd forward difference rather than 2nd central difference to approximate the derivative at the endpoints. Note from the equations on the previous page that the forward difference is less accurate than the central one used in the rest of the matrix, i.e. $\mathcal{O}\{\Delta xx\}$ vs. $\mathcal{O}\{(\Delta xx)^2\}$, but often this may be negligible.
{2, -5, 4, -1}	There does exist an $\mathcal{O}\{(\Delta xx)^2\}$ accuracy 2nd forward difference with four terms (this is it), derived in numerical methods textbooks by including one more term in the Taylor expansion, if you want to match the $\mathcal{O}\{(\Delta xx)^2\}$ accuracy of the rest of the **L** matrix.
{0, 0, 0, 0}	Set a boundary condition constraining *f''* to equal zero at the endpoints, so that there is zero curvature there.
{-2, 2, 0, 0}	Set a boundary condition constraining *f'* to equal zero at the endpoints, so that there is a zero gradient there.
{-2, 0, 0, 0}	Set a boundary condition constraining *f* to equal zero at the endpoints. **L** is invertible with this choice.
{-2, 1, 0, 0}	Extends minimum curvature integral from $-\infty$ to $+\infty$ and forces $f=0$ at all nodes outside grid. *f''* at boundary and first node outside grid can have non-zero curvature. **L** is invertible with this choice.

You can derive the boundary conditions listed above using the 1st and 2nd finite difference formulas on the previous pages, rearranging them to substitute for f_{j-1}, which is the inaccessible point past the endpoint. (Hint on the $f=0$ one, you somewhere specify f_j as the mean of f_{j-1} and f_{j+1}.)

With the exception of the last two, most of the choices in the table share the drawback that they cause at least one of the eigenvalues of **L** to be zero, in turn causing **L** and hence $\mathbf{L}^T\mathbf{L}$ to be non-invertible, potentially making trouble later when trying to invert $(\mathbf{A}^T\mathbf{A}+\nu^2\,\mathbf{L}^T\mathbf{L})$. Often in practice this may not be a problem; it depends on the null space of the **A** matrix. If it is a problem, you might get stuck with having to use a different choice of boundary condition.

Putting the pieces together to estimate the model curve:
Now let's assemble our various parts to compute our spline curve fit. We had two requirements for the estimation which we now have the pieces to specify:

1.) Exactly fit the data $\mathbf{d}=\mathbf{Am}$, or in our case we'll fit it very closely rather than exactly by assuming tiny data noise. This way we can set up the problem in a way that you can adapt for your own inverse problems.

2.) Minimize the norm of the 2nd derivative of **m**, i.e. minimize $\|\mathbf{Lm}\|_2$, making **Lm** close to the zero vector **0**.

We can combine those requirements like this:

$$\text{minimize} \quad \left\| \begin{bmatrix} \hat{\mathbf{A}} \\ \nu\mathbf{L} \end{bmatrix} \mathbf{m} - \begin{bmatrix} \hat{\mathbf{d}} \\ \mathbf{0} \end{bmatrix} \right\|^2 , \qquad \text{where} \quad \begin{aligned} \hat{\mathbf{A}} &= \tfrac{1}{\sigma}\mathbf{A} \\ \hat{\mathbf{d}} &= \tfrac{1}{\sigma}\mathbf{d} \end{aligned}$$

The σ is the standard deviation of some tiny amount of data noise that we'll make up for the problem, here the same for all data points – just make it a few orders of magnitude smaller than the values of your data points. (Recall from class that for a more general problem with data noise which isn't all the same for every data point, we could instead pre-multiply **A** and **d** by $\mathbf{C}^{-1/2}$, where **C** is the data noise covariance matrix.) The choice of tradeoff parameter ν will depend upon σ and is chosen via the discrepancy principle on the L-curve, although in this example problem there's handily a wide insensitive range for ν so we'll skip picking the best ν off the L-curve in this lab. You'll definitely need to do that in general however, according to one of the methods presented in class or in Aster/Borchers/Thurber.

We can solve the normal equations for the requirements above to find the least-squares fit for the model curve **m**:

$$\begin{bmatrix} \hat{\mathbf{A}} \\ \nu\mathbf{L} \end{bmatrix}^T \begin{bmatrix} \hat{\mathbf{A}} \\ \nu\mathbf{L} \end{bmatrix} \mathbf{m} = \begin{bmatrix} \hat{\mathbf{A}} \\ \nu\mathbf{L} \end{bmatrix}^T \begin{bmatrix} \hat{\mathbf{d}} \\ \mathbf{0} \end{bmatrix} \qquad \Longrightarrow \qquad \left[\hat{\mathbf{A}}^T \nu\mathbf{L}^T\right] \begin{bmatrix} \hat{\mathbf{A}} \\ \nu\mathbf{L} \end{bmatrix} \mathbf{m} = \left[\hat{\mathbf{A}}^T \nu\mathbf{L}^T\right] \begin{bmatrix} \hat{\mathbf{d}} \\ \mathbf{0} \end{bmatrix}$$

$$\tilde{\mathbf{m}} = (\hat{\mathbf{A}}^T\hat{\mathbf{A}} + \nu^2\mathbf{L}^T\mathbf{L})^{-1}\hat{\mathbf{A}}^T\hat{\mathbf{d}}$$

The $\hat{\mathbf{A}}$ and $\hat{\mathbf{d}}$ are again the noise-normalized values defined above. The $\tilde{\mathbf{m}}$ is the maximum likelihood solution to the regularized problem for a given value of ν. In a general problem we would find the optimal ν by using the discrepancy principle on the L-curve, which means rerunning the problem a number of times, and each time solving for $\tilde{\mathbf{m}}$ and the norms of the data residuals and of the model smoothness. The rule of thumb that we learned for a useful range of ν is to start with the largest value as:

$$\nu_{largest} = \sqrt{\frac{max(eigenvalues(\mathbf{A}^T\mathbf{A}))}{max(eigenvalues(\mathbf{L}^T\mathbf{L}))}}$$

and then decrease exponentially over say 10 order of magnitude or so (yes that's a lot!). So for example you might have 10 ν points as:

$$\nu_k = \nu_{largest}/10^{k-1}$$

and then zoom in on the region of the optimal ν later. In this particular problem chosen for this lab assignment, it turns out that very little changes between $\nu_{largest}$ and the optimal

ν, so conveniently here we'll get away with just using $\nu_{largest}$ in this lab (you should try it and see for yourself if you're curious). Whether for this lab or another problem later, be aware of the convenient Matlab function `eig()` for computing $\nu_{largest}$.

Lastly, all the above process was for a linear inverse problem. But all the same material applies to the iterated, linearized approach for weakly nonlinear problems that we learned in Labs #4 and #5. The important difference is that in Labs #4 and #5 we were talking about a parameter estimation problem with just a few parameters and so we needed no regularization using (say) an **L** matrix. But when we have an actual inverse problem where we must estimate a continuous vector model function from a finite number of data points, we need to add in the regularization, which involves choosing an optimal ν. The remaining concern for the nonlinear problem is whether to estimate that optimal ν at each iteration or to run the whole set of iterations for each ν and choose the optimal ν at the end. The former is used in the Occam's inversion method, is more computationally efficient, and often works well in practice. The latter is the Gauss-Newton method, and is less computationally efficient but can be justified more rigorously regarding the definition of the optimal ν as the maximum likelihood of ν. The upshot is that folks use both methods, and that both methods use the Occam's Razor principle via regularization with the 2nd finite difference operator matrix **L**.

Okay, now let's end by listing the steps of the assignment for this lab, which is basically to compare the 2nd finite difference to the exact derivative of some polynomial, to fit our smoothest curve to some made-up points, and then to compare this curve to that produced by the Matlab `spline()` function.

Assignment:

Part A:

1.) Make up some 5th or 6th degree polynomial.
2.) Compute your polynomial's 2nd derivative by hand.
3.) Create a vector of many closely and evenly spaced xx_j values (say, 100 of them).
4.) Calculate the polynomial values f_j and 2nd derivative values g_j at your xx_j.
5.) Create a few versions of your **L** matrix (based on the table) and multiply them by the f_j to approximate the 2nd derivative.
6.) Plot the f_j, g_j, and your few versions of 2nd finite differences on top of each other, against the xx_j. Compare. How do the edge values compare?

Part B:

1.) Forget about the above polynomial completely now!
2.) Make up around 10 or so data points h_i, with their x-axis values x_i drawn from among the xx_j above (this will let us use just 0's and 1's in the **A** matrix rather than worrying about interpolating).

3.) Make up a data noise standard deviation σ that is very small compared to the magnitude of your h_i.
4.) Now compute the smoothest curve to fit those data points:
    a. Create your **A** matrix (hint: use the indices *i* and *j* that you used in Part B step 2).
    b. Normalize **A** and **d** by your data noise standard deviation σ.
    c. Create your (*m*-2)x*m* **L** matrix, where *m* is the length of **xx**, i.e. the number of *j* indices.
    d. Calculate $v_{largest}$.
    e. Calculate $\tilde{\mathbf{m}}$ which is your vector of $f(xx_j)$, the discretized smoothest curve fit to the data points (or at least its very close approximation).
5.) Use Matlab's `spline()` function to compute a spline curve that way and plot along with your $\tilde{\mathbf{m}}$ result above. Your $\tilde{\mathbf{m}}$ should be the smoothest possible curve fit, so if the Matlab curve doesn't exactly match, is $\tilde{\mathbf{m}}$ at least smoother? Remark on any differences.