Missing-data restoration

[1]The interpolation of missing data is important to reduce the effort and risk in solving inverse problems. In this chapter, we solve two simple least-squares problems for missing data. In the first case, since the filter is known, the problem is linear. In the second case, because the filter is also unknown we must solve a nonlinear problem. Solving the linear problem by conjugate gradient yields the same results as Claerbout's solution in PVI (**?**). For the nonlinear case I have chosen a different solving scheme from the one in PVI. The results of the two schemes depart slightly and can be explained by the difference in the numerical algorithms.

INTERPOLATING MISSING DATA WITH A KNOWN FILTER

A method for restoring missing data is to ensure that the restored data, after specified filtering, has minimum energy.

To illustrate this method, let m denote a missing value. The dataset on which the examples are based is $\mathbf{d} = (\cdots, m, m, 1, m, 2, 1, 2, m, m, \cdots)$. The forward modeling operator is the convolution with the given filter $\mathbf{f}$. Now the problem is to minimize $\mathbf{f} * \mathbf{d}$ as follows:

$$0 \quad \approx \quad \mathbf{f} * \mathbf{d} \tag{1}$$

Let $\mathbf{M}$ denote a mask operator that hides the known values in the data. This operator is a diagonal square matrix with 1 or 0 as elements. For example, suppose that we have data such as $(m, m, 1, m, 2, 1, m, m)$. Then the mask operator for these data looks like this:

$$\begin{pmatrix}
1 & . & . & . & . & . & . & . \\
. & 1 & . & . & . & . & . & . \\
. & . & 0 & . & . & . & . & . \\
. & . & . & 1 & . & . & . & . \\
. & . & . & . & 0 & . & . & . \\
. & . & . & . & . & 0 & . & . \\
. & . & . & . & . & . & 1 & . \\
. & . & . & . & . & . & . & 1
\end{pmatrix} \tag{2}$$

Implementing the mask operator, we can represent the data in this way:

$$\mathbf{d} \quad = \quad \mathbf{M}\mathbf{d} + (\mathbf{I} - \mathbf{M})\,\mathbf{d} \tag{3}$$

[1]Hyang-Im Oh

where $\mathbf{Md}$ is the missing part and $(\mathbf{I}-\mathbf{M})\,\mathbf{d}$ the known part. Defining $(\mathbf{I}-\mathbf{M})\,\mathbf{d}$ as $\mathbf{d}_k$ and rewriting the equation (1) yields

$$0 \quad \approx \quad \mathbf{f} * (\mathbf{Md}) + \mathbf{f} * \mathbf{d}_k \tag{4}$$

We can then separate the known part from the unknown in the equation (4) and write it in the form of $\mathbf{y} = \mathbf{Ax}$, as follows:

$$-\,\mathbf{f} * \mathbf{d}_k \quad \approx \quad \mathbf{f} * (\mathbf{Md}) \tag{5}$$

The left side, $-\mathbf{f} * \mathbf{d}_k$, is known. The operator is $(\mathbf{f}*)\,\mathbf{M}$. We simply use the C++ least square solver to get $\mathbf{d}$.

```
//   miss1.cc :  1-D missing data with known filter
#include "miss1.h"
floatspace miss1(floatspace & filtsp, floatspace & data, int niter) {

        Axislist ax = data.getaxislist();           // Get axis info of data

        fopMask    mymask(ax.list[0],data);         // Mask operator
        fopContran myconv(filtsp,0);                // Convolution operator
        fopChain   myop( myconv,mymask);            // Conv*Mask
        floatspace resid = -(myconv.Forward(data));  // Make residual
        floatspace outspace = (fophestenes(data,resid,myop,niter))(0,0);//Solver

        return outspace;
}
```

We are now ready to code the problem. Let me explain the program briefly. The two operators applied to the missing data are masking (`mymask`) and convolution (`myconv`). The CLOP library class `fopMask` is constructed by taking either a `floatspace` or an integer array as an argument together with an axis to apply the operator on (see man page ??). According to the position of zero(missing data) in the argument, 1 or 0 is assigned to the diagonal elements in the operator matrix like the one in the equation (2). The CLOP class `fopContran` implements a convolution with the transient end effect (see page ??). The convolution operator is constructed from the known filter coefficients in `filtsp`. Since the two linear operators are applied sequentially, we can chain them, which is a nice feature of CLOP. The class `fopChain` concatenates those two operators into a single operator `myop`. The chained operator is treated like any other linear operator. We need not write any new operator but can just concatenate to solve the problem.

Now we have `resid`, `myop`, and `data` prepared in the code, which correspond to $\mathbf{y}$, $\mathbf{A}$, and $\mathbf{x}$, respectively, in $\mathbf{y} = \mathbf{Ax}$. The standard Hestenes solver routine uses CG to solve for the missing data. Here the maximum iteration number `niter` is set to be 10 to generate the Figures 1 through 4.

Using the optimization program `miss1()`, values are found to replace the missing m values so that the power in the filtered data is minimized. Figure 1 shows interpolation of the dataset with $1 - Z$ as a roughening filter. The interpolated data match the given data where they overlap.

Figures 1–3 illustrate that the rougher the filter, the smoother the interpolated data, and vice versa. Let us turn our attention from the residual spectrum to the

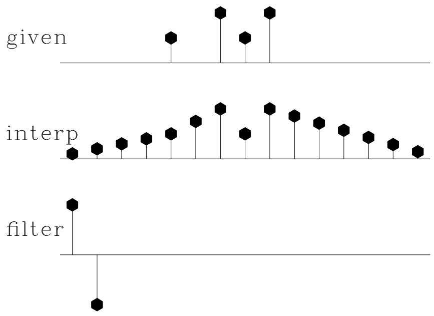

Figure 1: The top row shows the given data; the middle, the given data with interpolated values; the bottom,the filter $(1, -1)$ whose output has minimum power. mis-mlines [NR]

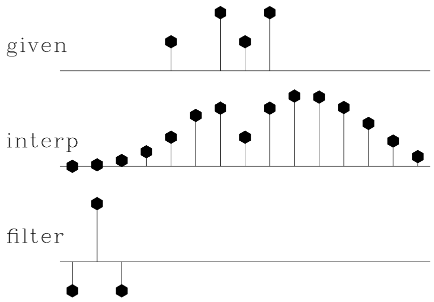

Figure 2: The top shows the same input data as in Figure 1. The middle is interpolated. The bottom shows the filter $(-1, 2, -1)$. The missing data seem to be interpolated by parabolas. mis-mparab [NR]

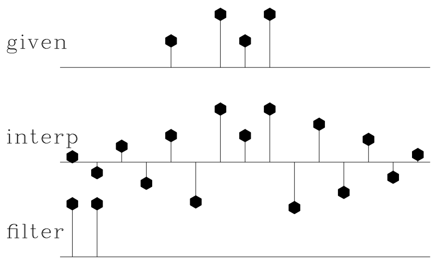

Figure 3: The bottom shows the filter $(1, 1)$. The interpolation is rough. Like the given data itself, the interpolation has much energy at the Nyquist frequency. But unlike the given data, it has little zero-frequency energy. mis-moscil [NR]

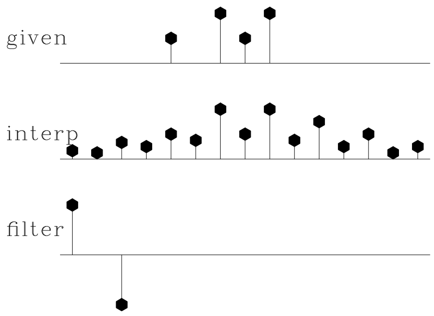

Figure 4: The top is the same as in Figures 1 to 3. The middle is interpolated. The bottom shows the filter $(1, 0, -1)$, which comes from the coefficients of $(1-Z)(1+Z)$. Both the given data and the interpolated data have significant energy at both the zero and the Nyquist frequencies. mis-mbest [NR]

residual itself. The residual for Figure 1 is the *slope* of the signal (because the filter $1 - Z$ is a first derivative), and the slope is constant (uniformly distributed) along the straight lines where the least-squares procedure is choosing signal values. So these examples confirm the idea that the least-squares method abhors large values (because they are squared). Thus, least squares tend to uniformly distribute residuals in both time and frequency to the extent that the constraints allow.

So far, we have concentrated on interpolating the missing data with a known filter. That is a linear problem. However, in real life, it is hard to determine the best filter. We usually have to treat it as an unknown and solve another optimization problem to determine which filter to use. Now the problem becomes a nonlinear one, which the next section discusses.

INTERPOLATING MISSING DATA WITH AN UNKNOWN FILTER

The material below is an attempt to reproduce the results of the missing data chapter in PVI. In Figures 1 through 4, the filters are taken as known, and the only unknowns are the missing data. Now, instead of having a predetermined filter, let us solve for the filter as well as for the missing data. The principle we need to use is that the output power is minimized while the filter is constrained to have one nonzero coefficient (otherwise all the coefficients would go to zero).

Packing both the missing data and the filter into a CG vector

First, it is important to examine the theory and coding behind the Figures 5 through **??**. Initially, we define a roughening filter $\mathbf{f}$ and a data signal $\mathbf{d}$. The regression is $0 \approx \mathbf{f} * \mathbf{d}$ where the filter $\mathbf{f}$ has at least one coefficient constrained to be nonzero and the data contain both known and missing values. Suppose that the perturbations are $\Delta\mathbf{f}$ and $\Delta\mathbf{d}$. We neglect the nonlinear term $\Delta\mathbf{f} * \Delta\mathbf{d}$ as follows:

$$0 \approx (\mathbf{f} + \Delta\mathbf{f}) * (\mathbf{d} + \Delta\mathbf{d}) \tag{6}$$

$$0 \approx \mathbf{f} * \mathbf{d} + \mathbf{d} * \Delta\mathbf{f} + \mathbf{f} * \Delta\mathbf{d} + \Delta\mathbf{f} * \Delta\mathbf{d} \tag{7}$$

$$-\mathbf{f} * \mathbf{d} \approx \mathbf{d} * \Delta\mathbf{f} + \mathbf{f} * \Delta\mathbf{d} \tag{8}$$

Again let's take the mask operator to separate the known values from the missing part of the data, defining $\mathbf{M}_f$ as a mask operator for the unknown filter and $\mathbf{M}_d$ for the missing data. Then,

$$\mathbf{f} = \mathbf{M}_f\mathbf{f} + (\mathbf{I} - \mathbf{M}_f)\,\mathbf{f} \tag{9}$$

$$\mathbf{d} = \mathbf{M}_d\mathbf{d} + (\mathbf{I} - \mathbf{M}_d)\,\mathbf{d} \tag{10}$$

For simplicity, we can define

$$\mathbf{f}_k = (\mathbf{I} - \mathbf{M}_f)\,\mathbf{f} \tag{11}$$

$$\mathbf{d}_k = (\mathbf{I} - \mathbf{M}_d)\,\mathbf{d} \tag{12}$$

where $\mathbf{f}_k$ is the known part of the filter and $\mathbf{d}_k$ the known part of the data at some stage of interpolation. Thus, the perturbation terms $\Delta\mathbf{f}$ and $\Delta\mathbf{d}$ in equation (8) correspond to $\mathbf{M}_f\mathbf{f}$ and $\mathbf{M}_d\mathbf{d}$, respectively. Now the problem can be rewritten with the mask operators as follows:

$$-\mathbf{f}_k * \mathbf{d}_k \quad = \quad \mathbf{d}_k * (\mathbf{M}_f\mathbf{f}) \quad + \quad \mathbf{f}_k * (\mathbf{M}_d\mathbf{d}) \tag{13}$$

Rewriting the above equation in matrix form yields

$$-\mathbf{f}_k * \mathbf{d}_k \quad = \quad \begin{pmatrix} \mathbf{f}_k* & \mathbf{d}_k* \end{pmatrix} \begin{pmatrix} \mathbf{M}_d & 0 \\ 0 & \mathbf{M}_f \end{pmatrix} \begin{pmatrix} \mathbf{d} \\ \mathbf{f} \end{pmatrix} \tag{14}$$

where $\mathbf{f}_k*$ represents convolution with $\mathbf{f}_k$ and $\mathbf{d}_k*$ with $\mathbf{d}_k$.

Now we have a linearized problem exactly in the form of $\mathbf{y} = \mathbf{A}\mathbf{x}$, where

$$\mathbf{A} \quad = \quad \begin{pmatrix} \mathbf{f}_k* & \mathbf{d}_k* \end{pmatrix} \begin{pmatrix} \mathbf{M}_d & 0 \\ 0 & \mathbf{M}_f \end{pmatrix} \tag{15}$$

and

$$\mathbf{x} \quad = \quad \begin{pmatrix} \mathbf{d} \\ \mathbf{f} \end{pmatrix} \tag{16}$$

We solve the equation (14) for the unknown $\mathbf{x}$ using Hestenes solver, as in the preceding section. Then, $\mathbf{d}_k$ and $\mathbf{f}_k$ are updated and solved again, and so on.

All these jobs can be nicely done in CLOP by using `spacearray` and `oparray` classes. In the program `missif()`, `x` is defined as `floatspacearray`, which is an array of `floatspace` **d** and `floatspace` **f**. In the same manner, we build the `fopArray` class of the convolution with the data and with the filter, say $(\mathbf{f}_k * \quad \mathbf{d}_k *)$. One of the interesting things about using the `fopArray` class is that applying it to `spacearray` class works the same way as the matrix multiplication. Thus, we have to make sure that all the products $\mathtt{fopArray}_{(\mathtt{i,j})} \cdot \mathtt{spacearray}_{(\mathtt{j,k})}$ have the same dimension to add them together. For example, $\mathtt{op1} \cdot \mathtt{space1}$ should be conformable with $\mathtt{op2} \cdot \mathtt{space2}$ in equation (17) to make the operation work as follows:

$$\begin{pmatrix} \mathtt{op1} & \mathtt{op2} \end{pmatrix} \begin{pmatrix} \mathtt{space1} \\ \mathtt{space2} \end{pmatrix} \quad = \quad \begin{pmatrix} \mathtt{op1} \cdot \mathtt{space1} + \mathtt{op2} \cdot \mathtt{space2} \end{pmatrix} \tag{17}$$

The second matrix of the mask operators in equation (14) are created by `fopDiagonal` class which takes operators and creates a square matrix with them in the diagonal.

```
//   missif.cc : 1-D missing data interpolation with unknown filter
#include "missif.h"
floatspacearray missif(const floatspace & f, const floatspace & d,
                                           int lag, int niter ) {
        floatspace data = d, filt = f;
        floatArray ff   = filt.getarray();
        Axislist   dax  = data.getaxislist();
        Axislist   fax  = filt.getaxislist();
        floatspacearray x(2,1),resid;
        x(0,0) = data;                        // array of data and filter
        x(1,0) = filt;
```

```
    if( ff(lag - 1) == 0.)
            seperr( "Error: filter(lag) = 0. \n");

    int * mask = new int[fax.list[0].length];
    for( int i=0; i < fax.list[0].length ; i++)
            if ( i != lag-1 )
                    mask[i] = 1;            // mask for filter

    fopDiagonal mymask(2);                  // diagonal array of Mask op
    fopMask     dmask(dax.list[0],data);
    fopMask     fmask(fax.list[0],mask);
    mymask.set(0,dmask);
    mymask.set(1,fmask);

    for( int iter=0; iter < niter;  iter++){    // outer nonlinear loop
                                                // update Operators
            fopContran * convf  = new fopContran(filt,0);
            fopContran * convd  = new fopContran(data,0);
            fopArray   * myconv = new fopArray(1,2);
            myconv->set(0,0,*convf);
            myconv->set(0,1,*convd);
            fopChain   * myop   = new fopChain(*myconv, mymask);

            resid = -(convf->Forward(data));    // make residual
            x = fophestenes(x, resid, *myop, 2);// Hestenes Linear Solver
            data += x(0,0);                     // update unknowns
            filt += x(1,0);

            delete convf; delete convd; delete myconv; delete myop;
    }
    delete mask;
    return x;
}
```

Two convolutions are combined into one `fopArray myconv(1,2)` operator, and two masking operators into `fopDiagonal mymask(2,2)`. Then we're ready to make a single operator $\mathbf{A}$, `myop` in the code by using `fopChain` to plug into the solver routine. Having built the operator , we only need to perform two basic loop calculations. First we linearize the problem and solve it, using the Hestenes solver in the inner iteration loop. Then we do the iteration and update the residual and the operator in the outer one. The residual is recalculated inside the outer nonlinear loop to avoid the accumulation of errors that would result from neglecting the nonlinear product $(\mathbf{M}_f\mathbf{f}) * (\mathbf{M}_d\mathbf{d})$.

In the program `missif()`, we need to determine two integers for the number of iterations in the inner loop and in the outer one. I have chosen 2 for the inner linear Hestenes solver to avoid breaking the linearity assumption by making a big change from the previous unknown. Otherwise, the output becomes unstable which results in diverging interpolated data. For the outer nonlinear loop, 22 is used.

Figures 5 through **??** show the interpolated data and filter made from `missif()`. In Figure 5 the filter is constrained to be of the form $(1, a_1, a_2)$. The result is the same as in PVI. The filter comes out slightly different from the $(1, 0, -1)$ in Figure 4, which is based on a subjective analysis. Constraining the filter to be of the form $(a_{-2}, a_{-1}, 1)$ in Figure 6 yields the same interpolated missing data as in Figure 5, as is the case in PVI. The next two figures show the interpolation for synthetic data consisting of a fragment of a damped exponential, and off to one side of it an

Figure 5: The top row shows known data. The middle includes the interpolated values. The bottom is the filter with the leftmost point constrained to be unity and other points chosen to minimize output power. mis-missif [NR]

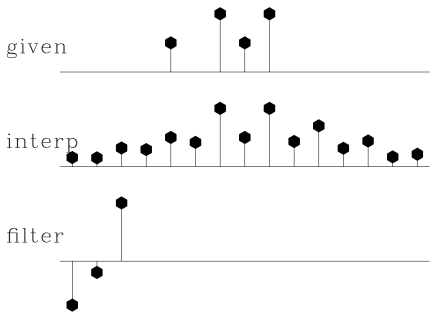

Figure 6: The filter here had its rightmost point constrained to be unity—i.e., this filtering amounts to backward prediction. The interpolated data seem to be identical, as with forward prediction. mis-backwards [NR]

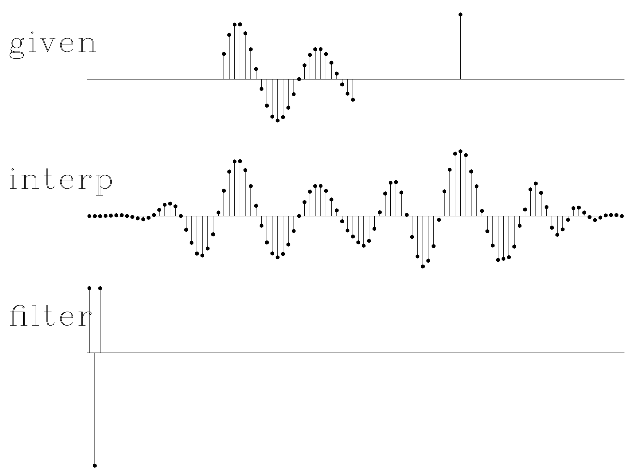

Figure 7: The top row shows synthetic data with the missing data represented by zeros. The middle includes the interpolated values. The bottom is a prediction-error filter. The filter may look symmetric but is not quite. mis-exp [NR]

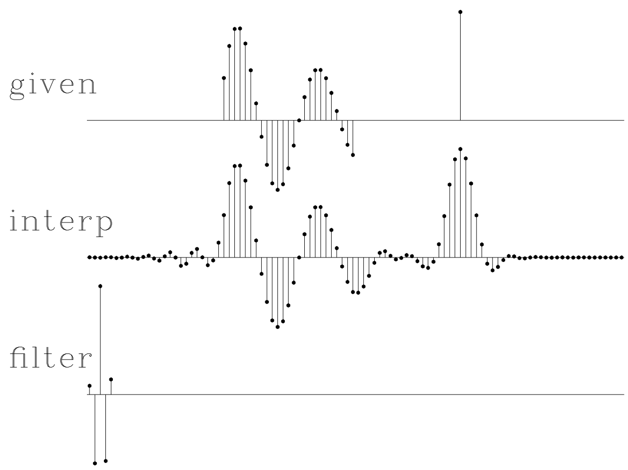

Figure 8: The top row shows synthetic data with the missing data represented by zeros. The middle includes the interpolated values. The bottom is the filter, an interpolation-error filter. mis-center [NR]

impulse function. Most of the energy is in the damped exponential. Figure 7 shows that the spectrum and the extended data are about what we would expect. From the extrapolated data, it is impossible to see where the given data end. In Figure **??** the filter is constrained in the middle, which produces output with a much shorter wavelength than in Figure 7. The boundary between real data and extended data is not nearly as well hidden as in Figure 7.

SUMMARY

Missing data can be restored by using the theorem that the output has minimum energy after specified filtering. The roughening filter can either be guessed or optimized. The former is a linear problem and the latter a nonlinear problem.

For the nonlinear problem, I have used a different scheme from Claerbout's. My code first linearizes the problem, then solves it explicitly and iterates. In contrast, the code in PVI hides the linear solving step inside the nonlinear iteration loop. The two schemes yield different results. The interpolated output from my program is very dependent on the number of iterations in the inner linear loop. The results can be improved by using a smaller number of iterations. For a larger number, such as five or more, the solution diverges. As mentioned earlier, it seems that a large number of iterations violates the linearity assumption. For now Claerbout's method seems to converge faster and gives a little better result when used for interpolating missing data. However, an interesting question remains as how to find in general the best scheme for a given nonlinear optimization problem.

Finally, coding the missing data problems in C++ is a pleasantly efficient process. We can simply use existing objects in the CLOP library. The programs in the preceding two sections look almost the same as the mathematics itself, in contrast to the verbose Fortran expressions for handling variables. Furthermore, we can separate the problem into two steps, making necessary operator objects and passing them to the solver routine, which can be designed by those who do professional numerical analysis.