¿

Solving least-squares inverse problems

[1] The adjoint operator does not recover the input to the forward operator unless the operator happens to be unitary. If we want a better reconstruction than that given by the adjoint, we must use some form of inverse operator. Given a data vector $\mathbf{y}$ and an operator $\mathbf{A}$, we wish to find a model $\mathbf{x}$ such that $\mathbf{A}(\mathbf{x}) = \mathbf{y}$.

If the operator $\mathbf{A}$, is a matrix we could solve the problem in the following way,

$$\begin{aligned} \mathbf{A}\mathbf{x} &= \mathbf{y} \\ \mathbf{A}'\mathbf{A}\mathbf{x} &= \mathbf{A}'\mathbf{y} \\ \mathbf{x} &= (\mathbf{A}'\mathbf{A})^{-1}\mathbf{A}'\mathbf{y} \end{aligned}$$

If $\mathbf{A}$ is not represented as a matrix but it is implemented as an operator we can still solve this problem but we need another operator that is the "inverse" to $\mathbf{A}'\mathbf{A}$, i.e. for any $\mathbf{x}$ we require an operator $\mathbf{B}$ such that $\mathbf{B}(\mathbf{A}'(\mathbf{A}(\mathbf{x}))) = \mathbf{x}$ for all $\mathbf{x}$. The operator $\mathbf{B}$ can then be regarded as the inverse of $\mathbf{A}'\mathbf{A}$.

This assumes that the inverse of $(\mathbf{A}'\mathbf{A})$ exists. In many cases the matrix is singular or near enough to singular that this method will not work. We have to deal with problems that may be overdetermined, underdetermined and/or inconsistent. In these cases it is often best to choose to solve a different problem, the "least-squares" problem. We try to find a model that minimizes the L_2 norm of the difference between the predicted data and the observed data.

$$\min_{\mathbf{x}} ||\mathbf{A}\mathbf{x} - \mathbf{y}||^2$$

LEAST SQUARES SOLVERS

These concepts are implemented in solver objects. All solver objects are derived from the abstract base class `LSSolver`. The class header `lssolver.h` shows the public definitions. An instance of this class (a solver object) is constructed from an operator that maps a model space to a data space.

[1] Dave Nichols

```
#ifndef LSSOLVER_H
#define LSSOLVER_H
// abstract base class for least squares solver,
// the solve operation takes a RHS
// and solves the problem min ( || op( x ) - rhs ||^2

#include <foperator.h>
#include <floatspacea.h>

class LSSolver {

public:
    LSSolver( foperator& op );
    virtual ~LSSolver();

    virtual floatspacearray Solve(  const floatspacearray & rhs )=0;

protected:
    foperator * op;
};
#endif
```

It has one member function, `Solve`. This function takes as its argument a `floatspacearray` that is the right hand side of the least-squares problem and returns a `floatspacearray` that is the solution to the least squares problem. It is a pure virtual function, this means that it is not defined in this class. A class that is derived from this base must implement the function.

Iterative solvers

In our work we have operators that have a forward and adjoint operation defined, so we are interested in solution methods that only require these operations. There is a whole class of algorithms that use iterative methods to approximate the solution to the least squares problem. Many iterative methods can be implemented using the following set of operations,

- Application of the forward operator to an instance of the model space.
- Application of the adjoint operator to an instance of the data space.
- Scaling and addition of members of the model space
- Scaling and addition of members of the data space
- Inner products of instances of the model space
- Inner products of instances of the data space

All of the operations are defined for *any* of our operator and space classes. We can write solver classes that use these algorithms and they will be usable with any operator and space in our class library.

The first class derived from `LSSolver` is `IterSolver`. This defines the class of solvers that solve a least-squares problem by an iterative algorithm. What distinguishes these solution methods is that they start from an initial estimate of the solution $\mathbf{x}_0$ and produce a sequence of solutions that are better approximations to the true solution (we hope).

The constructor for the iterative solver has an optional integer parameter that is the maximum number of iterations to perform. The class adds a new version of the `Solve` function, one that takes both a right hand side and an initial solution. The `Solve` function defined in the `LSSolver` class is implemented by calling the new solve function with an initial solution that is all zeros.

The header `itersolver.h` shows that there are many more public functions for this class. The most important is the new `Solve` function. Note that it too is a pure virtual function, this means that we must derive classes from `IterSolver` that actually implement the new function.

```
class IterSolver : public LSSolver {

public:

IterSolver(foperator & fop , int maxit=10 ) :
        LSSolver( fop ), save_x_iter(0), save_resid(0),
        maxiter(maxit), numiter(0) , repstream(nil){};

virtual floatspacearray Solve(const floatspacearray & rhs,
                              const floatspacearray& init) = 0; //pure virtual

floatspacearray Solve(  const floatspacearray & rhs );

void ReportOn( ostream & rep )                  // report on this stream
        {repstream = &rep; }

void SetSaveResidStep( int save_step );         // set residual save interval
void SetSaveIterStep( int save_step );          // set iteration save interval

int GetIterCount(){ return numiter; }           // number of iterations used

floatspacearray GetResiduals( );                // retrieve all residuals
floatspacearray GetOneResid( int i );           // retrieve one residual

floatspacearray GetIterations();                // retrieve all iterations
floatspacearray GetOneIter( int i );            // retrieve one iteration
```

```
~IterSolver();

}
```

The rest of the functions are utility functions common to all iterative solvers. They allow the user to save the sequence of iterations and/or residuals, to retrieve the intermediate results and to control reporting of the progress of the algorithm.

A conjugate gradient solver

The first "concrete" solver class (one with all the functions defined) is `NormCGSolver`. It is derived from `IterSolver` and it implements the `Solve` routine by using the "conjugate gradient" method on the normal equations associated with the least squares problem. This class implements the algorithm describe by Claerbout in PVI (**?**). Although this is not a numerically well behaved algorithm, it is implemented here to provide direct comparisons with solutions illustrated in PVI. The header file `normcgsolver.h` shows that all that is defined is, a constructor, and the solve routine.

```
class NormCGSolver : public IterSolver {
public:

  NormCGSolver( foperator & fop, int maxit=10 ) : IterSolver( fop, maxit ){};

  floatspacearray Solve( const floatspacearray & rhs,
                         const floatspacearray & initial);

  floatspacearray Solve( const floatspacearray & rhs )
        { return IterSolver::Solve( rhs ); }

};
```

The implementation of the `Solve` routine is shown below. Note that at the end of each iteration it calls the functions `report()`, `save_one_iter()`, `save_one_resid()`, these functions are defined in the `IterSolver` class and are inherited by all classes derived from it.

```
#include <normcgsolver.h>

floatspacearray NormCGSolver::Solve( const floatspacearray & rhs,
                                     const floatspacearray & init  ) {
    floatspacearray x,grad,cgrad,resid;
    double alfa,beta,determ;
    double cgnorm,csnorm,cgxr,csxr,cgxcs;
```

```
    floatspacearray step =  init.empty();
    floatspacearray cstep = rhs.empty();

    rhs_norm = rhs.norm();
    x = init; resid = rhs - op->Forward(x);

    numiter = 0;
    do{
        grad = op->Adjoint(resid);
        cgrad = op->Forward(grad);
        if (numiter==0) {
            alfa = cgrad.dot(resid) / cgrad.norm2();
            beta = 0;
        } else {
            cgnorm = cgrad.norm2(); csnorm = cstep.norm2();
            cgxcs = cgrad.dot(cstep);
            cgxr = cgrad.dot(resid); csxr = cstep.dot(resid);
            determ = cgnorm * csnorm - cgxcs * cgxcs + 1.0E-15;
            alfa = (csnorm * cgxr - cgxcs * csxr) / determ;
            beta = (-cgxcs * cgxr + cgnorm * csxr) / determ;
        }
        step = alfa * grad + beta * step;
        cstep = alfa * cgrad + beta * cstep;
        x += step; resid -= cstep;

        // gather statistics, report, save iterations etc.
        resid_norm = resid.norm();
        report(); save_one_iter( x ); save_one_resid( resid );

    }while( (numiter++ < maxiter) && ( resid_norm/rhs_norm > 1e-8 ) );

    return x;
}
```

A different conjugate direction solver

The class `HestSolver` implements the algorithm describes by Hestenes (**?**). Again all we define is a constructor and a `Solve` function. The implementation of this algorithm is shown in `hestsolver.cc`. This algorithm is said to be more stable than the simple conjugate-gradient method on normal equations described earlier.

```
#include <hestsolver.h>

floatspacearray HestSolver::Solve( const floatspacearray & rhs,
                                   const floatspacearray & init  ) {
```

```
    floatspacearray x,step,cstep,grad,resid;     // work space
    double gamma,gamma1,alpha,beta;

    rhs_norm = rhs.norm();                      // initial norm
    x = init; resid = rhs - op->Forward(x);     // initial residual

    grad = op->Adjoint(resid);
    step = grad;  gamma1 = step.norm2();

    numiter=0;
    do{
        cstep = op->Forward(step);
        alpha = gamma1 / cstep.norm2();
        x += step * alpha;
        cstep *= alpha; resid -= cstep;
        grad = op->Adjoint(resid);
        gamma = grad.norm2();
        beta = gamma / gamma1;
        gamma1 = gamma;
        step *= beta; step += grad;

        // gather statistics, report, save iterations etc.
        resid_norm = resid.norm();
        report(); save_one_iter( x ); save_one_resid( resid );

    }while( (numiter++ <maxiter) && ( resid_norm / rhs_norm > 1e-8 ) );

    return x;
}
```

These classes represent a first attempt to define least-squares solver objects. An inspection of the `Solve` function for the two classes shows that they have a lot of code in common. This usually means that the classes have been poorly designed and that some of this functionality belongs in the parent class. Another limitation is that it is difficult to fit non-linear solvers into this class heirarchy. These classes do not represent the final stages of our code development and they will probably change significantly before we are satisfied.

USING SOLVER OBJECTS

The program `SolvNmo.cc` illustrates the use of a solver object. The bulk of the code deals with reading and writing SEPlib data (our local data format). The core of the code constructs an NMO operator (`op`), it then constructs a solver from that operator (`solver`) and invokes the solver on the input data. The result of invoking the solver is written to the output. Note that the solver is instructed to report its progress on the stream associated with `stderr`.

```
/* SolvNmo vel=1.5 <infile.H >outfile.H
 * Applies inverse NMO operator to dataset, given velocity vel.
 */
#include <SEPinit.h>
#include <fopnmo.h>
#include <hestsolver.h>

main(int argc,char **argv) {
    SepInput * sepInput = SEPinit(argc,argv,SOURCE);// Seplib initialization

    int niter=5; sepInput->Getch("niter",niter);    // command line arguments
    float vel=1.5; sepInput->Getch("vel",vel);

    floatspace input(sepInput);                     // make input space
    Axislist inaxlist = input.getaxislist();        // retrive axis info

    fopNMO op(inaxlist[0],inaxlist[1],1/vel);       // make NMO operator

    HestSolver solver( op, niter );                 // construct solver object
    solver.ReportOn(cerr);                          // report progress on stderr    floatspacearray output = solver.Solve( input ); // solve the problem

     // write to output
    SepOutput *sepout = new SepOutput("stdout", output(0,0).sepData() );
    delete sepout;

    UnRef(sepInput); return 0; // cleanup
}
```

This program produces results identical to those shown in the section on NMO and inversion **??**.

PRECONDITIONED INVERSE PROBLEMS

Many iterative algorithms converge very slowly, an inversion for a 100×100 model has 10,000 free parameters. To guarantee complete solution we might need 10,000 iterations. This is impractical, in most cases we can only afford a few iterations. One way to make more effective use of the iterations that we can afford is to use preconditioners in our iterative methods. The aim in using a preconditioner is to solve a related problem that has faster convergence properties and then to convert the solution of the related problem to the solution of our problem.

Given a problem associated with the operator $\mathbf{A}$ we seek an operator $\mathbf{P}$ such that a problem involving the operator $\mathbf{AP}$ has better convergence properties that $\mathbf{A}$.

The original problem

$$\mathbf{A}(\mathbf{x}) = \mathbf{y}$$

is transformed to

$$\mathbf{AP}(\tilde{\mathbf{x}}) = \mathbf{y}$$

and

$$\mathbf{P}(\tilde{\mathbf{x}}) = \mathbf{x}$$

We solve the preconditioned problem to obtain $\tilde{\mathbf{x}}$ and then apply the preconditioner one more to obtain $\mathbf{x}$.

What makes a system converge faster? Mathematically we seek a preconditioner that "clusters the eigenvalues" of the system. Ideally the operator $\mathbf{A}\,\mathbf{P}$ would be unitary, then application of the adjoint of $\mathbf{AP}$ would solve the preconditioned problem, this would mean that $\mathbf{P}'$ was actually $(\mathbf{A}'\mathbf{A})^{-1}$. This suggests that an approximation to $(\mathbf{A}'\mathbf{A})^{-1/2}$ might make a good preconditioner. Some people use the inverse of the diagonal of $(\mathbf{A}'\mathbf{A})^{-1/2}$ as a preconditioner. The most effective preconditioners are usually problem specific, they rely on knowledge of the structure of the problem to design a good preconditioner.

In our C++ classes the use of preconditioners is a trivial process. Once we have a preconditioner we can convert an operator into a preconditioned operator by combining the two in a chain. We then solve the problem associated with the chained operator and apply the preconditioner to the result to obtain the final solution. We do *not* need to modify any of the solver classes to solve the preconditioned problem. Here is a code fragment for solving a system using a preconditioner.

```
foperator A( ... ) ;  // the operator
foperator P( ... ) ;  // the preconditioner

fopChain op( A, P );  // the preconditioned operator

floatspacearray rhs;  // the right hand side

HestSolver solver( op ); // a solver for the preconditioned operator

floatspacearray x1 = solver.Solve( rhs );  // solve preconditioned problem

floatspacearray x = P.Forward( x1 );  // apply preconditioner
```

THE "Process" UTILITY

In many of the sections of this document we write a new operator and we need a small driver program to test the operator and demonstrate its application. In all these programs we give the use the choice of applying the operator in a forward or adjoint manner or solving an inverse problem involving either the forward or adjoint operator. This is so common that we have written a small utility routine to simplify the process. This routine, `Process`, is passed a `foperator`, a `floatspacearray` to apply it to and flags to indicate how it should be used. It returns a `floatspacearray`. The declaration of the routine is shown in `Process.h`. The `adj` flag selects either the forward or adjoint operation. If the `inv` flag is zero, it just applies the operator and returns the result. If the `inv` flag is 1, it will solve an inverse problems using the `HestSolver` class.

```
#ifndef PROCESS_H
#define PROCESS_H

// This Utility function applies an operator to an input space in one
// of several ways. It returns a floatspacearray as the result.

#include <foperator.h>
#include <floatspacea.h>

extern floatspacearray Process(
        const floatspacearray& input,   // The input space
        foperator& op,                  // The operator to be applied
        int adj,                        // adj=0 use forward op, adj=1 adjoint
        int inv=0,                      // inv=1 solve inverse problem
        int maxiter=2 );                // maximum number of iterations

#endif
```