HW6 for GPGN605: Nonlinear Inversion

Xinming Wu

CWID: 10622240

x Center for Wave Phenomena, Colorado School of Mines, Golden, CO 80401, USA

1 SOFTWARE

All my Java codes for my home work projects are available here:

`https://github.com/xinwucwp/inversionTheory/tree/master/inversionTheory`

Please visit this link for more details.

2 TOTAL OBJECTIVE FUNCTION

The total objective function is defined as

$$\text{Min } \phi = \phi_d + \beta\phi_m \equiv \|\mathbf{W}_d(F[\mathbf{h}] - \mathbf{d}_o)\|_2^2 + \beta\|\mathbf{W}_m\mathbf{h}\|_2^2, \tag{1}$$

where $F[\cdot]$ is the forwarding modeling, $\mathbf{h}$ is the model vector, $\mathbf{W}_d$ is the weighting matrix for the data, and $\mathbf{W}_m$ is the weighting matrix for the model. For these two weighting matrices, we have

$$\mathbf{W}_d^\top\mathbf{W}_d = \begin{bmatrix} 1/\sigma_1^2 & 0 & \dots & 0 \\ 0 & 1/\sigma_2^2 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \dots & 1/\sigma_n^2 \end{bmatrix}_{n\times n}, \tag{2}$$

where n is the number of the samples in the data.

My Java code to compute $\mathbf{y} = \mathbf{W}_d^\top\mathbf{W}_d\mathbf{x}$ (without explicitly forming matrices) is:

```
// apply W'dWd operator
  private void applyWdWd(float[] x, float[] y) {
    float wd = 1.0f/_dSigma;
    float ws = wd*wd;
    mul(ws,x,y);
  }
```

The weighting matrix for the model is

$$\mathbf{W}_m^\top \mathbf{W}_m = \alpha_s \mathbf{W}_s^\top \mathbf{W}_s + \alpha_x \mathbf{W}_x^\top \mathbf{W}_x$$
$$= \alpha_s d_m \mathbf{I}_{m\times m} + \frac{\alpha_x}{d_m} \begin{bmatrix} 2 & -1 & & 0 \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ 0 & & -1 & 2 \end{bmatrix}_{m\times m}, \tag{3}$$

where $\alpha_s = 0.00002$, $\alpha_x = 1.0$ and d_m is the discretization interval for the model.

My Java code to compute $\mathbf{y} = \mathbf{W}_m^\top \mathbf{W}_m \mathbf{x}$ (without explicitly forming matrices) is:

```
// apply W'mWm operator
  private void applyWmWm(float dm, float[] x, float[] y) {
    int n = x.length;
    float ws = _alphaS*dm;
    float dx = _alphaX/dm;
    float d11 = dx;
    float d12 = dx;
    float d22 = dx;
    for (int i=1; i<n;++i) {
      float xa = 0.0f;
      float xb = 0.0f;
      xa += x[i ];
      xb -= x[i-1];
      float ya = d11*xa+d12*xb;
      float yb = d12*xa+d22*xb;
```

```
    y[i  ] += ya;
    y[i-1] -= yb;
  }
  add(mul(ws,x),y,y);
}
```

3 FORWARD MODELING

My forward modeling Java code translated from "vdyke.m" is shown as below:

```
// compute predicted data
  public static void forward(float[] mk, float[] dk) {
    zero(dk);
    int n = mk.length;
    for (int i=0; i<n; ++i) {
      float zbi = mk[i];
      float zti = _zt[i];
      float xci = _xc[i];
      float[] dki = forward(xci,zti,zbi);
      add(dki,dk,dk);
    }
  }
  // forward for each prism
  public static float[] forward(float xc, float zt, float zb) {
    // construct the "polygon" representing the vertical dyke
    int np = 4;
    float[] xp = new float[np];
    float[] zp = new float[np];
    float swd1 = 0.5f*_wd;
    float swd2 = 0.0001f*_wd;
    xp[0] = xc-swd1; zp[0] = zt-swd2;
    xp[1] = xc+swd1; zp[1] = zt+swd2;
    xp[2] = xp[1];  zp[2] = zb+swd2;
```

```
    xp[3] = xp[0]; zp[3] = zb-swd2;
    float gcons = 0.006672f;
    int nd = _xo.length;
    float[] sum = zerofloat(nd);
    for (int ip=0; ip<np; ++ip) {
      int ipp = ip+1;
      if (ip==np-1) ipp = 0;
      float xpi = xp[ip];
      float zpi = zp[ip];
      float xpe = xp[ipp];
      float zpe = zp[ipp];
      float dxi = abs(xpe-xpi);
      float dzi = abs(zpe-zpi);
      if (dzi<=0.000001f) zpi +=0.00001f*dxi;
      float[] x1 = sub(xpi,_xo);
      float[] x2 = sub(xpe,_xo);
      float[] z1 = sub(zpi,_zo);
      float[] z2 = sub(zpe,_zo);
      float[] r1 = add(mul(x1,x1),mul(z1,z1));
      float[] r2 = add(mul(x2,x2),mul(z2,z2));
      float[] bt = sub(z2,z1);
      float[] alpha = div(sub(x2,x1),bt);
      float[] beta  = div(sub(mul(x1,z2),mul(x2,z1)),bt);
      float[] factor = div(beta, add(1.0f,mul(alpha,alpha)));
      float[] term1 = mul(0.5f,log(div(r2,r1)));
      float[] term2 = sub(atan(z2,x2), atan(z1,x1));
      float[] update = sub(term1, mul(alpha,term2));
      sum = add(sum, mul(factor,update));
    }
    float sca = 2.0f*_dc*gcons;
    return mul(sca,sum);
  }
```

4 JACOBIAN MATRIX

In my implementation, I do not explicitly form the Jacobian matrix $\mathbf{J}$ or the transpose of the Jacobian matrix $\mathbf{J}^\top$. My Java code computing $\mathbf{y} = \mathbf{J}\mathbf{x}$ and $\mathbf{y} = \mathbf{J}^\top\mathbf{x}$ are shown below:

```java
// apply J operator
  private void applyJacb(float[] dk, float[] mk, float[] x, float[] y) {
    zero(y);
    int nd = dk.length;
    int nm = mk.length;
    float[] dp = new float[nd];
    for (int im=0; im<nm; ++im) {
      mk[im] += _dh;
      forward(mk,dp);
      mk[im] -= _dh;
      for (int id=0; id<nd; ++id)
        y[id] += x[im]*(dp[id]-dk[id])/_dh;
    }
  }
  // apply J' operator
  private void applyJacbT(float[] dk, float[] mk, float[] x, float[] y) {
    zero(y);
    int nd = dk.length;
    int nm = mk.length;
    float[] dp = new float[nd];
    for (int im=0; im<nm; ++im) {
      mk[im] += _dh;
      forward(mk,dp);
      mk[im] -= _dh;
      for (int id=0; id<nd; ++id)
        y[im] +=x[id]*(dp[id]-dk[id])/_dh;
    }
  }
```

5 LINEAR SYSTEM IN EACH GAUSS-NEWTON ITERATION

In each iteration of the Gauss-Newton method, we solve a following linear system

$$(\mathbf{J}^\top {\mathbf{W}^\top}_d \mathbf{W}_d \mathbf{J} + \beta {\mathbf{W}^\top}_m \mathbf{W}_m)\mathbf{p} = \mathbf{J}^\top {\mathbf{W}^\top}_d \mathbf{W}_d \mathbf{J} \delta \mathbf{d} - \beta {\mathbf{W}^\top}_m \mathbf{W}_m \mathbf{h}^{(k)}. \tag{4}$$

In this linear system, the matrix $(\mathbf{J}^\top {\mathbf{W}^\top}_d \mathbf{W}_d \mathbf{J} + \beta {\mathbf{W}^\top}_m \mathbf{W}_m)$ on the right-hand side is symmetric positive definite, therefore, we use the Conjugate Gradient (CG) method to solve this linear system. My Java code for the CG method is shown as below:

```java
package hw6;
public class CgSolver {
  public enum Stop {
    TINY,
    MAXI
  }
  public static class Info {
    private Info(Stop stop, int niter, double bnorm, double rnorm) {
      this.stop = stop;
      this.niter = niter;
      this.bnorm = bnorm;
      this.rnorm = rnorm;
    }
    public Stop stop;
    public int niter;
    public double bnorm;
    public double rnorm;
  }
  //linear operator A.
  public interface A {
    // input x
    // output y
    public void apply(Vec x, Vec y);
  }
```

```
public CgSolver(double tiny, int maxi) {
  _tiny = tiny;
  _maxi = maxi;
}
// a the linear operator that represents the matrix A.
// param b the right-hand-side vector.
// x the solution vector.
public Info solve(A a, Vec b, Vec x) {
  return solve(0.0,a,b,x);
}
// Solves the system of equation Ax = b with CG iterations.
public Info solve(double anorm, A a, Vec b, Vec x) {
  Vec q = b.clone();
  a.apply(x,q); // q = Ax
  Vec r = b.clone();
  r.add(1.0,q,-1.0); // r = b-Ax
  Vec d = r.clone();
  double bnorm = b.norm2();
  double rnorm = r.norm2();
  double xnorm = x.norm2();
  double rrnorm = rnorm*rnorm;
  logInit(bnorm,rnorm,xnorm);
  int iter;
  for (iter=0; iter<_maxi && rnorm>_tiny*(anorm*xnorm+bnorm); ++iter) {
    logIter(iter,rnorm,xnorm);
    a.apply(d,q);
    double dq = d.dot(q);
    double alpha = rrnorm/dq;
    x.add(1.0,d,alpha);
    xnorm = x.norm2();
    if (iter%50==49) { // if accumulated rounding error may be large, ...
      a.apply(x,q); // q = Ax
      r.add(0.0,b,1.0); // r = b
```

```
        r.add(1.0,q,-1.0); // r = b-Ax
      } else { // otherwise, use shortcut to update residual
        r.add(1.0,q,-alpha); // r -= alpha*q
      }
      double rrnormOld = rrnorm;
      rnorm = r.norm2();
      rrnorm = rnorm*rnorm;
      double beta = rrnorm/rrnormOld;
      d.add(beta,r,1.0);
    }
    logDone(iter,rnorm,xnorm);
    Stop stop = (iter<_maxi)?Stop.TINY:Stop.MAXI;
    return new Info(stop,iter,bnorm,rnorm);
  }
  ///////////////////////////////////////////////////////////////////////////
  // private
  private double _anorm; // estimate for norm(A); default is zero
  private double _tiny; // converged: norm(r)<tiny*(norm(A)*norm(x)+norm(b))
  private int _maxi; // upper limit on number of iterations
}
```

5.1 Left hand side

To compute the left-hand side, I do not form the matrix, and my Jave implementation is

```
private void applyLhs(
  float dm, float beta, float[] dk, float[] mk, float[] x, float[] y)
{
  int nm = mk.length;
  int nd = dk.length;
  float[] y1 = new float[nm];
  float[] y2 = new float[nm];
  float[] yd = new float[nd];
```

```
  applyJacb(dk,mk,x,yd);
  applyWdWd(yd,yd);
  applyJacbT(dk,mk,yd,y1);
  applyWmWm(dm,x,y2);
  add(y1,mul(beta,y2),y);
}
```

5.2 Right hand side

To compute the Right-hand side, I do not form the matrix, and my Jave implementation is:

```
private void makeRhs(
  float dm, float beta, float[] dk, float[] dok, float[] mk, float[] y)
  {
  int nm = mk.length;
  int nd = dk.length;
  float[] yd = new float[nd];
  float[] y1 = new float[nm];
  float[] y2 = new float[nm];
  applyWdWd(dok,yd);
  applyJacbT(dk,mk,yd,y1);
  applyWmWm(dm,mk,y2);
  sub(y1,mul(beta,y2),y);
}
```

6 RESULTS

6.1 Recovered models with different β

From the provided observed data as shown in Figure 1, I use 6 different β for the objection function as shown in Equation 1, and the corresponding recovered models are shown in Figure 2. From the results, we observe that:

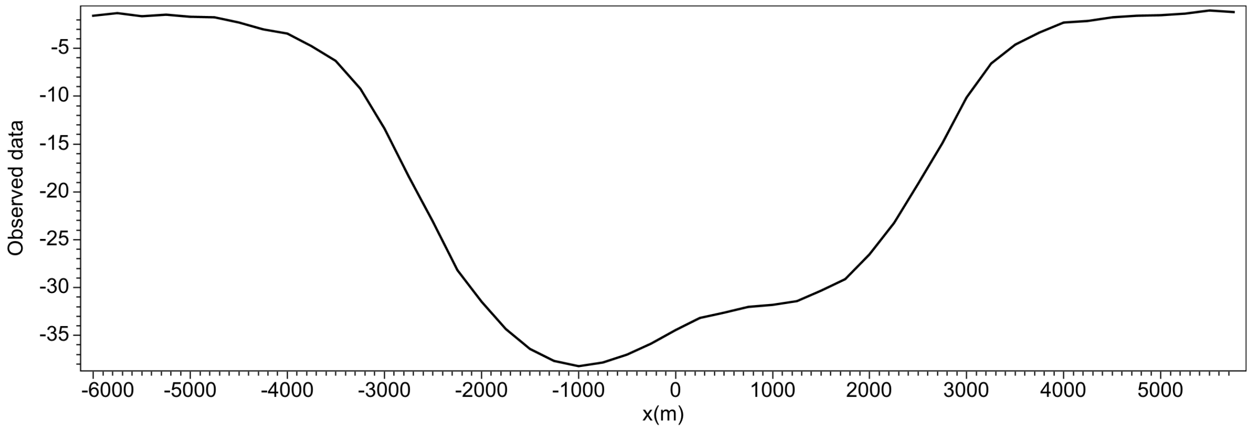

Figure 1. Observed data.

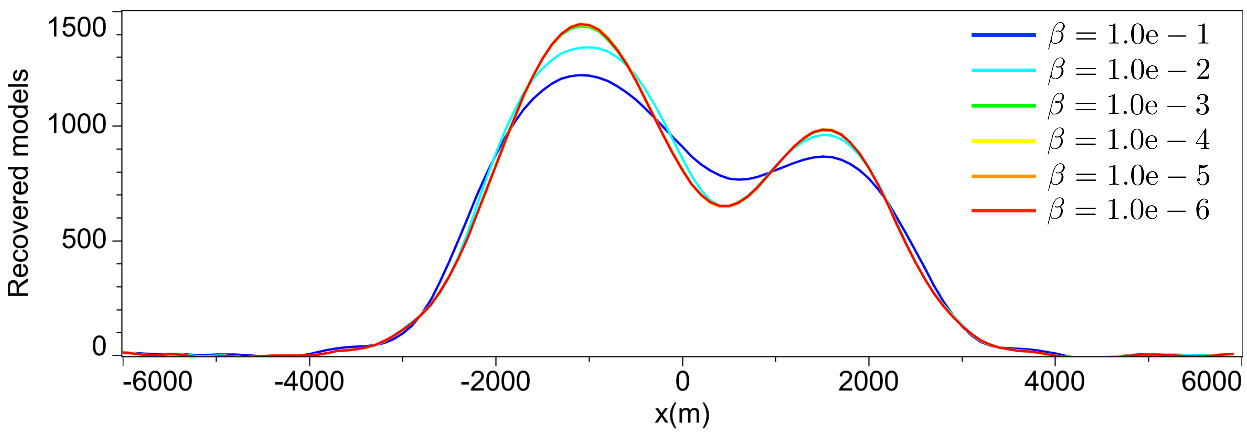

Figure 2. Six recovered models with 6 different β.

1) the recovered model is deeper using smaller β;

2) when $\beta > 1.0\mathrm{e}-4$, the recovered models are almost the same.

6.2 Object function with Gauss-Newton iterates

To evaluate the Gauss-Newton method, I use $\beta =$, and the total object function ϕ decreases with the Guss-Newton iterates as shown in Figure 3, from which we observe that the object function descreases dramatically at the first iteration.

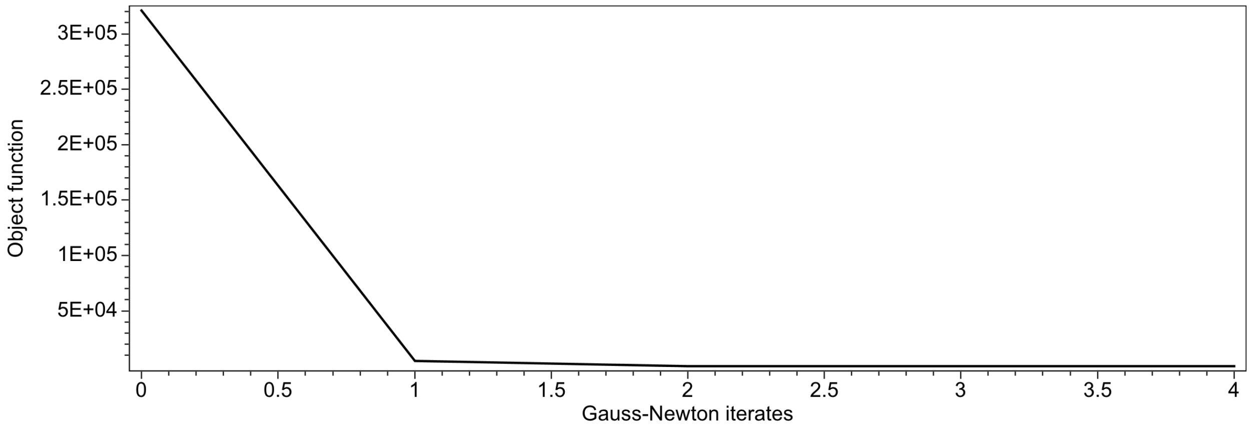

Figure 3. The total object function ϕ decreases with the Gauss-Newtom iterations.