Spring 2014:
Computational and Variational Methods for Inverse Problems
CSE 397/GEO 391/ME 397/ORI 397
Assignment 5 (due 9 May 2014)

1. An inverse problem for Burgers' equation

Consider the inverse problem for the viscosity coefficient ν in the one-dimensional Burgers' equation (this equation is often taken as a one-dimensional surrogate for the Navier-Stokes equations). Using the spatial interval $[0, L]$ and the temporal interval $[0, T]$, the solution $u = u(t, x) : [0, T] \times [0, L]$ satisfies

$$u_t + uu_x - \nu u_{xx} = f \quad \text{in } (0, T) \times (0, L), \tag{1a}$$
$$u(t, 0) = u(t, L) = 0 \quad \text{for all } t \in [0, T], \tag{1b}$$
$$u(0, x) = 0 \quad \text{for all } x \in [0, L]. \tag{1c}$$

Here, $\nu = \nu(x) : [0, L] \to \mathbb{R}$ is the viscosity coefficient we wish to invert for, and $f = f(t, x)$ is a given source term. The conditions (1b) and (1c) are the boundary and the initial conditions, respectively.[1] We are given observations $u^{\text{obs}} = u^{\text{obs}}(t, x)$ for a latter portion of the time interval, i.e., for $t \in [T_1, T]$, $(T_1 > 0)$. To invert for the coefficient ν, we thus use the functional

$$J(a) := \frac{1}{2} \int_{T_1}^{T} \int_0^L (u - u^{\text{obs}})^2 \, dx dt + \frac{\beta}{2} \int_0^L \frac{d\nu}{dx} \frac{d\nu}{dx} \, dx \tag{2}$$

with a regularization parameter $\beta > 0$. To use an optimization method for (2), we require the gradient of J with respect to ν.

1. Derive a weak form of (1) by multiplying (1a) with a test function $p(t, x) : [0, T] \times [0, L]$ that satisfies Dirichlet boundary conditions analogous to (1b), and integrating over space and time. There is no need to impose the initial condition via a Lagrange multiplier (as we did in class for the initial condition inversion problem), since the inverse parameter (ν) does not appear in the initial condition. Thus $p(0, x)$ should be taken to be zero, similar to how we treat Dirichlet boundary conditions. Use integration-by-parts on the viscous term to derive the weak form of the Burgers' equation.

2. Using the Lagrangian functional, derive expressions for the adjoint equation and for the gradient of J with respect to ν. Give weak and strong forms of these equations. Note that ν as well as its variation $\tilde{\nu}$ are functions of space only, while u and the adjoint p are functions of space and time.

[1] Note that the boundary $\partial\Omega$ of the one-dimensional interval $\Omega = (0, L)$ are simply the points $x = 0$ and $x = L$, i.e., $\partial\Omega = \{0, L\}$

2. Inverse advection-diffusion inverse problem, continued from Assignment 4.

Here, we continue solution of the advection-diffusion inverse problem begun in Assignment 4. There we employed a steepest descent method to minimize the cost functional. Here, we will extend a state-of-the-art inexact Newton-CG method, the source code (described below) of which is available on the class webpage (`http://users.ices.utexas.edu/~omar/inverse_probs`). Please hand in printouts of your implementations (at least of the lines you modified) together with the results.

As a reminder, we solve the inverse problem for the advection-diffusion equation on $\Omega = [0,1] \times [0,1]$:

$$\min_m J(m) := \frac{1}{2} \int_\Omega (u - u^{obs})^2 \, dx + \frac{\beta}{2} \int_\Omega \nabla m \cdot \nabla m \, dx, \tag{3}$$

where $u(\boldsymbol{x})$ depends on the diffusivity $m(\boldsymbol{x})$ through

$$\begin{aligned} -\nabla \cdot (m \nabla u) + \boldsymbol{v} \cdot \nabla u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{4}$$

with the advective velocity $\boldsymbol{v}(\boldsymbol{x}) = (v_1, v_2)$, regularization parameter $\beta > 0$ and measurement data u^{obs}, which are obtained by solving the state equation with $m(x,y) = 2$ for $(x-0.5)^2 + (y-0.5)^2 \leq 0.04$ and $m(x,y) = 8$ otherwise (and adding noise).

1. In the solutions below, use an optimal regularization parameter β found from the discrepancy criterion.

2. Extend the COMSOL implementation `elliptic_GN_ip` of the Gauss-Newton method[2] by introducing the advection velocity $\boldsymbol{v} = (30, 0)$. Report the number of Gauss-Newton and of overall CG iterations for a discretization of the domain with 10×10, 20×20, 40×40 and 80×80 linear finite elements and give the number of unknowns used to discretize the coefficient function m (which is called a in the implementation) for each of these meshes.[3] Discuss how the numbers of iterations change as the parameter mesh is refined, i.e., as the parameter dimension increases.

3. To avoid "over-solving" the CG system in early Newton steps, where we are still far away from the solution and thus cannot benefit from the fast local convergence properties of Newton's method, the implementation uses the stopping criterion

$$\frac{\|H_k d_k + g_k\|}{\|g_k\|} \leq \eta_k.$$

   Here, H_k denotes the Hessian, g_k the gradient at the k-th iteration, and d_k the (inexact) Newton direction. Compare the behavior of the choices

   - $\eta_k = 0.5$,
   - $\eta_k = \min(0.5, \sqrt{\|g_k\|/\|g_0\|})$,

[2] Recall that the Gauss Newton method approximates the Hessian by dropping terms that depend on the adjoint variable p.

[3] If the 80×80 mesh too large for your computer, you can skip it.

- $\eta_k = \min(0.5, \|g_k\|/\|g_0\|)$,

where g_0 denotes the norm of the initial gradient (see also the 2nd assignment).

4. The ill-posedness of inverse problems is closely related to the spectrum (i.e., the eigenvalues) of the Hessian operator.[4] Compute the eigenvalues of the (Gauss-Newton) Hessian at the solution of the inverse problem for the 20×20 mesh. Since the (Gauss-Newton) Hessian is not available explicitly and can expressed only through its action on a vector, there are 2 possibilities to extract its eigenvalues:

   - Build the (Gauss-Newton) Hessian matrix explicitly by applying it to unit vectors and compute the eigenvalues of the resulting matrix.
   - *Preferred method:* Fortunately, there exist iterative methods, for example Lanczos, for computing eigenvalues of a matrix that require only the application of the matrix to vectors (these are known as matrix-free methods). Compute the largest 100 eigenvalues using `eigs` (which implements a Lanczos-like method) in Matlab[5]

   Plot the spectrum of the Hessian with and without regularization and discuss the result.

5. *Optional:* Replace Tikhonov regularization with total variation regularization[6] and report the results for different meshes.

Line-by-line description of inexact Gauss-Newton-CG implementation

Note that the coefficient m is called a in the implementation below. Moreover, the incremental variables are called `delta_u` and `delta_p`. Steps that are similar or identical to the steepest descent method (Assignment 4) are not described in detail again. Different from the implementation of the steepest descent method, which relies to a large extent on solvers provided by COMSOL, this implementation makes explicit use of the discretized operators (matrices) corresponding to the state and the adjoint equations (i.e., the blocks in the KKT matrix). For brevity of the description, we skip steps that are analogous to the steepest descent method. Note that the line numbers do not coincide with the provided implementation.

Note that there is an even shorter and easier way to implement the inexact Newton-CG method, which uses COMSOL's ability to symbolically linearize weak forms—basically, it computes the linearizations required for the incremental equations automatically. However, since the provided implementation is somewhat more explicit, it is easier to understand. Let Georg know if you are interested in the shorter implementation, which we recommend following if you intend to build an inverse problem solver with a similar toolkit for your own research.

After setting up the mesh, the finite element functions a, u and p corresponding to the coefficient, state and adjoint variables, as well as their increments are defined in lines 4 and 5.

[4] The eigenvalues usually decay rapidly such that inversion of the Hessian can be done in a stable manner only with the use of regularization.

[5] Note that the decaying property of the eigenvalues for ill-posed inverse problems is a reason why only very few steps of the conjugate gradient method are needed in each Newton step. This is because the CG method can be shown to eliminate error components of the solution in the direction of eigenvectors associated with large eigenvalues (for a more detailed discussion, see for instance J. Shewchuk: *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*).

[6] This adjustment will mainly affect the regularization matrix $\boldsymbol{R}$. Also, don't forget to adjust the cost function evaluation accordingly.

```
4 fem.dim = {'a' 'delta_a' 'delta_p' 'delta_u' 'p' 'u' 'ud'};
5 fem.shape = [1 1 1 1 1 1 1];
```

Homogeneous Dirichlet conditions are used for the state and adjoint variables, as well as for their increment functions; see line 16. After initializing parameters, the weak forms for the construction of the synthetic data (lines 17 and 18), and the weak forms for the Gauss-Newton system are defined (lines 19–24).

```
16 fem.bnd.r = {{'delta_u''delta_p' 'u' 'ud'}};
17 fem.equ.expr.goal = '-(atrue*(udx*udx_test+udy*udy_test)-f*ud_test)';
18 fem.equ.expr.state = ['-(a*(ux*ux_test+uy*uy_test)-f*u_test)'];
19 fem.equ.expr.incstate = ['-(a*(delta_ux*delta_px_test+delta_uy '...
20         '*delta_py_test)+delta_a*(ux*delta_px_test+uy*delta_py_test))'];
21 fem.equ.expr.incadjoint = ['-(a*(delta_px*delta_ux_test+
22 delta_py*delta_uy_test)+delta_u*delta_u_test)'];
23 fem.equ.expr.inccontrol = ['-(gamma*(delta_ax*delta_ax_test+delta_ay '...
24         '*delta_ay_test)+(delta_px*ux+delta_py*uy)*delta_a_test)'];
```

As in the implementation of the steepest descent method, the synthetic data are based on a "true" coefficient `atrue`, and noise is added to the synthetic measurements; the corresponding finite element function is denoted by `ud`. The indices pointing to the coefficients for each finite element function in the coefficient vector `X` are stored in `AI,dAI,dPI,...`. After setting the coefficient a to a constant initial guess, the gradient is computed, which provides the right hand side in the Hessian equation. For this purpose, the state equation (lines 50–52) is solved.

```
50 fem.xmesh = meshextend(fem);
51 fem.sol = femlin(fem, 'Solcomp', {'u'}, 'U', X);
52 X(UI) = fem.sol.u(UI);
```

Next, the KKT system is assembled in line 57. Note that the system matrix `K` does not take into account Dirichlet boundary conditions. These conditions are enforced through the constraint equation `N*X=M` (where N and M are the left and right hand sides of the boundary conditions and can be returned by the `assemble` function).

```
56 fem.xmesh = meshextend(fem);
57 [K, N] = assemble(fem, 'U', X, 'out', {'K', 'N'});
```

Our implementation explicitly uses the individual blocks of the KKT system—these blocks are extracted from the KKT system using the index vectors `dAI, dUI, dPI`; see lines 58–61. Note that the choice of the test function influences the location of these blocks in the KKT system matrix. Since Dirichlet boundary conditions are not taken care of in `K` (and thus in the matrix `A`, which corresponds to the state equation), these constraints are enforced by a modification of `A` (see lines 58–61). The modification puts zeros in rows and columns of Dirichlet nodes and ones into the diagonals; see lines 62–68. Additionally, changes to the right hand sides are made using a vector `chi`, which contains zeros for Dirichlet degrees of freedom and ones in all other components (lines 69–70).

```
58 W = K(dUI,dUI);
59 A = K(dPI,dUI);
60 C = K(dPI,dAI);
61 R = K(dAI,dAI);
62 ind = find(sum(N(:, dUI),1)~=0);
63 A(:,ind) = 0;
64 A(ind,:) = 0;
```

```
65 for (k = 1:length(ind))
66   i = ind(k);
67   A(i,i) = 1;
68 end
69 chi = ones(size(A,1),1);
70 chi(ind) = 0;
```

The adjoint is used to compute the gradient (line 72). Note that `MG` denotes the coefficients of the gradient, multiplied by the mass matrix, which corresponds to the right hand side in the Gauss-Newton equation.

```
71 X(PI) = A' \ (chi.*(W * (X(UDI) - X(UI))));
72 MG =  C' * X(PI) + R * X(AI);
```

To solve the Hessian system and obtain a descent direction, we use MATLAB's conjugate gradient function `pcg`. The function `elliptic_apply`, which is specified below, implements the application of the Hessian to a vector. The right hand side in the system is given by the negative gradient multiplied by the mass matrix. We use a loose tolerance early in the CG iteration, and tighten the tolerance as the iterates get closer to the solution; see line 80.

```
80 tolcg = min(0.5, sqrt(gradnorm/gradnorm_ini));
81 P = R + 1e-10*eye(length(AI));
82 [D, flag, relres, CGiter, resvec] = pcg (@(V)elliptic_apply
83                  (V, chi, W, AF, C, R, X, AI, A0I, mu),-MG, tolcg, 300, P);
```

The vector `D` resulting from the (approximate) solution of the Hessian system is used to update the coefficients of the FE function a. A line search with an Armijo criterion is used to globalize the Gauss-Newton method.

We conclude the description of the Gauss-Newton implementation with giving the function `elliptic_apply`. This function applies the Hessian to a vector `D`.

```
1 function GNV = elliptic_apply(V, chi, W, AF, C, R, X)
2 du = A \ (chi .* (-C * V));
3 dp = A' \ (chi .* (-W * du));
4 GNV = C' * dp + R * V;
```