MTH 655, Winter 2013, LAB4

The goal of this assignment is twofold. We explore i) mesh (in) sensitivity and convergence of Newton method for semilinear BVPs and semismooth Newton methods for problems under constraints and ii) smoothing properties of relaxation methods such as (weighted) Jacobi, Gauss-Seidel, and SOR.

Advanced students **turn in 3, 4iii-iv), and 5**. Less advanced students can **turn in three of 1, 2, 4(i-iii), and 5**.

---

1. *Consider the BVP problem*

$$-u''(x) = f(x), x \in (0,1); u(0) = u(1) = 0. \tag{1}$$

Recall that after FD (or FE) discretization, you solve the system $A\mathbf{u} = \mathbf{f}$ *where* A *is the discrete Laplacian, and* f *is the load vector. In a FE setting, this is equivalent to minimization of* $\int_0^1 \left(\frac{1}{2}(u')^2 - fu\right) dx$ *over an appropriate space (ask me for details if you are interested). The discrete counterpart of the minimization principle is*

$$\inf_{\mathbf{u}\in V} \frac{1}{2}\mathbf{u}^t A\mathbf{u} - \mathbf{u}^t f. \tag{2}$$

where $V = R^n$.

To solve (2) *under inequality constraints where, e.g., we impose*

$$V = \{\mathbf{u} \in R^n : u_j \geq u_j^*, \ \forall j\} \tag{3}$$

with $\mathbf{u}^*$ *a given vector, we use the* semi-smnooth Newton method *as described in class. We define a Lagrange multiplier* $\lambda = Au - f \geq 0$, *and rewrite the constraint as* $\min(u - u^*, \lambda) = 0$.

The provided code `semicon.m` is a template for solving (2) with (3), where $f(x) \equiv -1$, $u^*(x) \equiv -0.1$.

(i) Test whether the semismooth Newton method is *mesh-sensitive* by running it for $n = 10, 20, \ldots$ and observing the number of iterations needed. Consider a fixed $n = 100$, and extract and plot the convergence history, i.e., the magnitude of residual depending on the iteration number. Discuss.

(ii) **Extend** the code and implement a solution to (2) with

$$V = \{\mathbf{u} \in R^n : u_j \leq u_j^*, \ \forall j\} \tag{4}$$

$f(x) \equiv 1$, $u^*(x) = 1/4 * sin(3\pi x)$. (Note that your constraint is now rewritten as $\min(u^* - u, -\lambda) = 0$, and your u must be below the constraint, and $\lambda \leq 0$. You must remember to modify the appropriate parts of residual and Jacobian calculations).

---

2. Generalize (1) to the semilinear problem

$$-u''(x) + g(u(x)) = f(x), x \in (0,1); u(0) = u(1) = 0. \tag{5}$$

where $g(\cdot)$ *is a monotone increasing function with a primitive* $G(\cdot)$. *One can show that the solution minimizes the functional* $\int_0^1 \frac{1}{2}\left((u')^2 + G(u) - fu\right) dx$ *and that the discrete formulation is* $A\mathbf{u}+\mathbf{g}(\mathbf{u}) = \mathbf{f}$ *with* $(\mathbf{g}(\mathbf{u}))_j = g(u_j), j = 1, \ldots n$ *corresponding to*

$$\inf_{\mathbf{u}\in V} \frac{1}{2}\mathbf{u}^t Au + \mathbf{G}(\mathbf{u}) - \mathbf{u}^t f. \tag{6}$$

where $(\mathbf{G}(\mathbf{u}))_j = G(u_j), j = 1, \ldots n$.

The provided template `semilinear.m` shows how to solve the problem over V with $g(u) = u^3$.

Test the *mesh (in)dependence* of the Newton's method and the convergence of iteration. Experiment with $g(u) = \alpha g(u)$ for different α, also those for which $g(u)$ is not monotone increasing.

3. Combine Pbms 1 and 2 (i.e., solve (5) with constraints) and experiment with various constraints and values of α. Include the case $\alpha = 0$ and the case where the constraint is not active. Discuss the behavior of Newton's method.

---

4. *Recall the family of stationary iterative methods and relaxation methods discussed in class. They exhibit particular smoothing properties in that they damp higher frequency modes of the error.*

You can use the provided MATLAB code as a template to set up the iteration matrices for these methods for the purpose of testing their properties. For "real" implementation, see problem 5.

```
%% set-up A for a given n
A = zeros(n,n);
for j = 1:n A(j,j)=2;end; for j = 2:n A(j,j-1)=-1;end; for j = 1:n-1 A(j,j+1)=-1;end;
%% extract D, L, U in the splitting of A
D = diag(diag(A)); L = tril(A)-D; U = triu(A)-D;
%% set-up iteration matrices and vectors for a given omega
Gj = inv(D)*(-L-U); cj = inv(D)*f;
Ggs = inv(D+L)*(-U); cgs = inv(D+L)*f;
Gsor = inv(D/omega+L)*((1/omega-1)*D-U); csor = inv(D+omega*L)*omega*f;
```

i) **Explore** the smoothing properties of those methods by applying them to solve $A\mathbf{u} = 0$ for $n = 64$ when initial guess is a highly varying function: a multiple of one of the eigenvectors of A. You can use the template:

```
h = 1/(n+1);x=(1:n)'*h;u = sin(k*pi*x)';
for m=1:10 u= G*u; plot(x,u); end
```

ii) **Explore** how many iterations you need to for the residual (for this case, the magnitude of u) to be less than $\tau = 10^{-2}$ for the modes $k = 30, 15, 4$. Try $\omega = 2/3$, and experiment with other values of ω. Compare performance of the various methods.

iii) Mix up the three modes $k = 30, 15, 4$, use some interesting amplitudes and compare performance as in ii).

iv) Do ii)-iii) and find the optimal ω for the case when the differential equation is $-(k(x)u')' = f$ and $k(\cdot)$ is piecewise constant so that the problem has an interface. The appropriate entries of A must change to reflect this, e.g, $A_{jj} = k_1 + k_2, A_{j,j-1} = -k_1, A_{j,j+1} = -k_2$.

---

5. **Implement in FORTRAN or C:** Jacobi method or SOR method to solve the problem as in LAB3. Compare the solution obtained with the direct solver DGESV and your Jacobi solver. How many iterations are needed to solve the problem of size $n = 10, 20, 100, 1000$ to get the residual r to satisfy $\| r \|_\infty \leq \tau = 10^{-1}$?

For SOR, use optimal parameter ω which can be found by trial and error for a given n. (There is also theory which you can ask me about). The search for optimal ω can be implemented using

```
%% G is some iteration matrix dependent on omega
radius = max(abs(eig(G)));
```

The following code in MATLAB would execute the SOR iteration:

```
%% ucode starts with an initial guess, iteration continues indefinitely %% you must
implement a way to stop it
while 1
iter = iter + 1;
ucodeold = ucode;
for j=1:n
suml = 0;
for k=1:j-1
suml = suml + A(j,k)*ucode(k);
```

```
end
sumu = 0;
for k=j+1:n
sumu = sumu + A(j,k)*ucodeold(k);
end
ucode(j) = omega*(f(j)- sumu - suml)/A(j,j) +(1-omega)*ucodeold(j);
end
end
```