Periodic solution of non-linear parabolic value problem

December 30, 2003

1 Description of the problem

On $\Omega =]0,1[^2$ consider the boundry value problem

$$\begin{aligned} \tfrac{d}{dt}u - \operatorname{div}(\alpha(|\mathbf{grad}\ u|)\mathbf{grad}\ u) &= 0 && \text{in } \Omega, \\ \alpha(|\mathbf{grad}\ u|)\cdot\mathbf{n} &= g\cdot\sin(\tfrac{2\pi}{T}t) && \text{on } \partial\Omega, \\ u(0,\cdot) &= u(T,\cdot), \end{aligned} \tag{1}$$

where $T > 0$ is the known period, $g \in H^{-\frac{1}{2}}(\partial\Omega)$, and $\alpha \in ([0;\infty]), 0 < \alpha_0 \le \alpha(s) \le \alpha_1$, with $\alpha'(s) > 0, \forall s \ge 0$. In a concrete electromagnetic application (ferromagnetic materials) the coefficent function α is well approximated by

$$\alpha(s) \approx \alpha_1 - \frac{\alpha_1 - \alpha_0}{1 + \exp\left(k(s - s_0)\right)},$$

where $s_0 > 0$ and $k > 0$.
The problem can be cast in a variational form as *seek* $u = u(t,\cdot) \in C^1([0;T], H^1(\partial\Omega))$ *such that*

$$\int_\Omega \frac{d}{dt}u\, v + \alpha(|\mathbf{grad}\ u|)\mathbf{grad}\ u \cdot \mathbf{grad}\ v\, dx = \sin(\frac{2\pi}{T}t)\int_{\partial\Omega} g v\, dS \quad \forall v \in H^1(\Omega). \tag{2}$$

It can be discretized on a regular triangular grid Ω_h with meshwidth $h = \frac{1}{N-1}, N \in \mathbb{N}$ the number of gridlines in each direction. More precisely, the grid arises by subdividing the squares of a tensor-product grid of meshwidth h into two triangles each. The finite element discretization of (2) employs piecewise linear, globally continuous finite elements on Ω_h. In the case of $\alpha \equiv 1$ this will give rise to the usual 5-point stencil for the Laplacian.

The evaluation of the discretized operator involves numerical quadrature. In detail the semi-discrete variational problem reads *seek* $u_h = u_h(t,\cdot) \in C^1([0;T], V_h)$ *such that*

$$\begin{aligned} \sum_K \Bigg(\frac{|K|}{3}\sum_{j=1}^{3} \frac{d}{dt}u_h(t,\mathbf{a}_j^K)v_h(\mathbf{a}_j^K) + \\ + |K|\alpha(|\mathbf{grad}\ u_h(t,\mathbf{c}_K)|)\mathbf{grad}\ u_h(t,\mathbf{c}_K)\cdot\mathbf{grad}\ v_h(\mathbf{c}_K)\Bigg) = \\ \sin(\frac{2\pi}{T}t)\sum_{e\subset\partial\Omega}\frac{|e|}{2}(g(\mathbf{p}_1^e)v_h(\mathbf{p}_1^e) + g(\mathbf{p}_2^e)v_h(\mathbf{p}_2^e))\,, \end{aligned} \tag{3}$$

for all $v_h \in V_h$. Here, $\mathbf{a}_j^K$ are the vertices of the triangle K, $\mathbf{c}_K$ is its center of gravity, and $\mathbf{p}_1^e$, $\mathbf{p}_2^e$ stand for the endpoints of the boundary edge e. In compact form, using $\vec{u}$ to denote the coefficient vector belonging to a finite element solution, this semidiscrete problem can be written as

$$\mathrm{M}\,\frac{d}{dt}\vec{u}(t) + \mathrm{A}(\vec{u})\,\vec{u} = \sin(\frac{2\pi}{T}t)\,\vec{f}\,,$$

with a diagonal mass matrix $\mathrm{M} \in \mathbb{R}^{N^2,N^2}$ and a stiffness matrix $\mathrm{A}(\vec{u}) \in \mathbb{R}^{N^2,N^2}$.

Writing $J(\vec{u}) := \mathrm{A}(\vec{u})\,\vec{u}$, we obtain as Jacobian

$$DJ(u_h)w_h v_h = \sum_K |K|\alpha(|\mathbf{grad}\ u_h(\mathbf{c}_K)|)\mathbf{grad}\ w_h(\mathbf{c}_K)\cdot\mathbf{grad}\ v_h(\mathbf{c}_K) + \\ + |K|\frac{\alpha'(|\mathbf{grad}\ u_h(\mathbf{c}_K)|)}{|\mathbf{grad}\ u_h(\mathbf{c}_K)|}(\mathbf{grad}\ u_h(\mathbf{c}_K)\cdot\mathbf{grad}\ w_h(\mathbf{c}_K))(\mathbf{grad}\ u_h(\mathbf{c}_K)\cdot\mathbf{grad}\ v_h(\mathbf{c}_K)).$$

2 MATLAB implementation

The crucial MATLAB function is (File evaluation.m)

```
function [y,J] = evaluation(N,u,t,T)
```

`N` : resolution of the mesh, meshwidth $h = 1/(N-1)$,
`u` : coefficient column vector $\vec{u} \in^{N^2}$,
`t` : current time,
`T` : periodicity.

This function returns

`y` : $y = \mathrm{M}^{-1}(\sin(\frac{2\pi}{T}t)\,\vec{f} - \mathrm{A}(\vec{u})\,\vec{u})$
`J` : Jacobian $-\mathrm{M}^{-1}\,DJ(u_h) \in^{N^2,N^2}$ as sparse matrix

Auxiliary MATLAB functions are

- `function [va,da] = alpha(s)`

  which returns the values `va`= $\alpha(s)$ and `da`= $\alpha'(s)$. Altering this function changes the strength of the non-linearity.

```
function [va,da] = alpha(x)

alf1 = 1.0; alf0 = 0.001; k = 10; s0 = 1;
efn = exp(k*(x-s0));
den= 1+efn;
va = alf1 - (alf1-alf0)./den;
da=k*(alf1-alf0)*efn./(den.*den);
```

- `function g = gfun(x,y)`

  which provides the Neumann data g.

3 Simple shooting method

Now set, using the notations introduced above,

$$F(\vec{u},t) := \mathrm{M}^{-1}(\sin(\frac{2\pi}{T}t)\vec{f} - \mathrm{A}(\vec{u})\vec{u})\ .$$

Then the semi-discretized two-point boundary value problem corresponding to (1) can be stated as

$$\frac{d}{dt}\vec{u} = F(\vec{u},t) \quad , \quad \vec{u}(0) = \vec{u}(T)\ . \tag{4}$$

Writing $\Phi : [0;T] \times \mathbb{R}^{N^2} \mapsto \mathbb{R}^{N^2}$ for the evolution operator associated with (4) we can formulate the problem as non-linear equation

$$\Phi(T, \vec{u}_0) = \vec{u}_0 \; . \tag{5}$$

This equation can be solved by the Newton's method

$$\begin{array}{rcl} (\frac{d\Phi}{d\vec{u}_0}(\vec{u}_0^{(m)}, T) - Id)\vec{\delta}^{(m)} & = & -(\Phi(\vec{u}_0^{(m)}, T) - \vec{u}_0^{(m)}) \\ \vec{u}_0^{(m+1)} & = & \vec{u}_0^{(m)} + \vec{\delta}^{(m)}. \end{array} \tag{6}$$

In its course we need the following evaluations

$$\Phi(T, \vec{v}) \quad \text{and} \quad (\frac{d\Phi}{d\vec{u}_0}(T, \vec{v}) - Id)\vec{s} \, ,$$

for $\vec{v}, \vec{s} \in \mathbb{R}^{N^2}$. Note that $\dfrac{d\Phi}{d\vec{u}_0}(T, \vec{v})$ can be obtained as the solution of the initial-value problem

$$\frac{d}{dt}\mathrm{Z} = \frac{dF}{d\vec{u}}(\vec{u}(t), t)\,\mathrm{Z} \quad , \quad \mathrm{Z}(0) = Id \; , \tag{7}$$

where $\vec{u}$ is the solution of

$$\frac{d}{dt}\vec{u} = F(\vec{u}, t) \quad , \quad \vec{u}(0) = \vec{v} \; .$$

Remark. It is very important to keep in mind that both (4) and (7) are *stiff* problems. Therefore explicit timestepping is not an option. At least semi-implicit methods have to be used. This is why the MATLAB routine `evaluatiuon.m` also provides the Jacobian.

3.1 Outline of the shooting algorithm

In this particular case we have to deal with

$$F(\vec{u}, t) = \mathrm{M}^{-1}(\vec{q}(t) - J(\vec{u})) \; . \tag{8}$$

The evolution $\Phi(t, \vec{u})$ will be approximated by means of a semi-implicit Euler scheme with fixed timestep $\tau > 0$, which reads, $k = 1, \ldots, N$, $N := T/\tau$,

$$\begin{array}{c} \vec{v}^0 = \vec{u} \; , \\ (\mathrm{M} + \tau \, DJ(\vec{v}^{k-1}))\delta\vec{v} = \tau(\vec{q}(t_k) - J(\vec{v}^{k-1})) \; , \\ \vec{v}^k = \vec{v}^{k-1} + \delta\vec{v} \; . \end{array} \tag{9}$$

Eventually, we end up with $\Phi(T, \vec{v}) = \vec{v}^N$. In lockstep with the integration of the initial value problem, we can approximately solve (7): for $k = 1, \ldots, N$ compute

$$\begin{array}{c} \vec{z}^0 = \vec{s} \, , \\ (\mathrm{M} + \tau \, DJ(\vec{v}^k))\vec{z}^k = \mathrm{M}\vec{z}^{k-1} \; . \end{array} \tag{10}$$

3.2 Implementation of the shooting algorithm

The used MATLAB functions are

```
function [Dphi_w,phi] = shootingII(N,u,T,n,w)
```

and

```
function phi = phi_shooting(N,u,T,n)
```

with

`N` : resolution of the FEM mesh,
`u` : column vector $\vec{v} \in \mathbb{R}^{N^2}$,
`T` : periodicity,
`n` : number of time-steps in the Euler method,
`w` : column vector $\vec{s} \in \mathbb{R}^{N^2}$.

The functions return

`Dphi_w` : $(\frac{d\Phi}{d\vec{u}_0}(T,\vec{v}) - Id)\vec{s}$,
`phi` : $-(\Phi(T,\vec{v}) - \vec{v})$.

Remark: The shooting function doesn't return the entire Matrix $(\frac{d\Phi}{d\vec{u}_0}(T,\vec{v}) - Id)$ because in the run of solving the linear problem (11) it is only used as a multiplication with a column vector $\vec{s} \in \mathbb{R}^{N^2}$ (consult section 4, Newton update).

4 Newton update

In each Newton step, one has to solve the linear equation

$$(\frac{d\Phi}{d\vec{u}_0}(\vec{u}_0^{(m)},T) - Id)\vec{\delta}^{(m)} \quad = \quad -(\Phi(\vec{u}_0^{(m)},T) - \vec{u}_0^{(m)}). \tag{11}$$

It is solved in an iterative conjugated gradient squared method. In its course we need in every iteration step the evaluation of the right and the left hand side of the equation, what is done by the above introduced functions. It is obvious that the more iteration steps are needed until sufficient convergence, the more time is needed to reach a satisfying result. The relative tolerance demanded for the abortion of the iteration method as well as the maximum of iteration steps affect the total computation time of the algorithm very sensitively.

4.1 Implementation of the Newton algorithm

The Newton iteration is provided by the MATLAB function

```
function u=newton4(N,u,TolN,T,n)
```

with

`N` : FEM mesh resolution,
`u` : starting vector of the Newton iteration,
`TolN` : relative tolerance of for the Newton residual,
`T` : periodicity of the initial equation,
`n` : number of time-steps in the shooting method.

The function returns the values of the initial scalar field $u(0,\cdot)$ for the initial problem (1) at the vertices of the FEM mesh in the N^2 vector `u`. For further use `u` can be converged to matrix form.
The Newton update $\vec{\delta}^{(m)}$ from (11) is found by a conjugated gradient squared method, provided in

```
function [x,phi,flag,it,relres,resvec]=shoot_CGS(N,u,T,n,tol,maxit,psi,v0)
```

`N` : resolution of the mesh, meshwidth $h = 1/(N-1)$,
`u` : coefficient column vector $\vec{u} \in^{N^2}$,
`T` : periodicity,
`n` : number of time-steps in the semi-implicit Euler method,
`tol` : relative tolerance, that has to be reached until abortion (default is 10^{-6}),
`maxit` : maximum number of iterations (default is $min(30, N^2)$),
`psi` : optional default right hand side of the linear equation (11),
`v0` : starting vector of the iteration (default is zero).

This function returns

`x` : solution of the linear equation (i.e. the next Newton update)
`phi` : right hand side of the linear equation (11), `psi` if used,
`flag` :
- `0` then shoot_CGSconverged to the desired tolerance tol within maxit iterations without failing for any reason,
- `1` then shoot_CGS iterated maxit times but did not converge,
- `2` then a system of equations was ill-conditioned,
- `3` then shoot_CGS stagnated,
- `4` then one of the scalar quantities calculated during shoot_CGS became too small or too large to continue computing,

`relres` : the relative residual at the end of the iteration,
`resvec` : a vector of all residuals computed during iteration.

5 Example

For an example of the use of the algorithm the function `[u,A]=start_BVP` can be used, where the parameters for the algorithm can be set. In its course the result vector `u` of the `newton4` function is converted to the matrix `A`. A protocol of the evaluation is written to the text file *example.txt*, where run-time information can be observed.

6 MATLAB-files

For the use of the algorithm the following m-files have to be saved in the `bin` directory:
`aplha.m` : provides the non-linear function $\alpha(s)$ from the initial problem (1),
`evaluation.m` : for the FEM,
`gfun.m` : provides the Neumann data at boundary of the problem,
`massvec.m` : calculates the mass matrix of the FEM,
`newton4.m` : the Newton algorithm,
`phi_shooting.m` : calculates the right hand side of (11),
`shoot_CGS.m` : returns the Newton update,
`shootingII.m` : calculates the left and the right hand side of (11),
`start_BVP.m` : the example problem.

7 Performances

For $k = 10$ in the non-linearity function $\alpha(s)$ and the Neumann data $g = 10 \cdot \sinh(1 - (x + y))$ the run of the calculation gave showed the following performance, where N is the FEM resolution, n the number of time steps in the shooting algorithm, $TolN$ the demanded relative tolerance of the Newton residual and NNI the number of Newton iteration until convergence.

N	n	TolN	NNI
10	20	10^{-14}	3
	40	10^{-10}	3
	80	10^{-14}	3
	160	10^{-10}	2
	320	10^{-10}	2
15	40	10^{-10}	3
	80	10^{-10}	3
	160	10^{-10}	2
	320	10^{-10}	2
20	20	10^{-14}	6

Results for $k = 1$ and $g = 2$ are listed in the tabular below.

N	n	TolN	New. Iter.
10	20	10^{-14}	3
	40	10^{-10}	3
	80	10^{-10}	2
	160	10^{-10}	2
	320	10^{-10}	2
15	40	10^{-10}	2

For a third example $k = 3$, $g = 2 \cdot \cosh(1 - (x + y))$.

	n	TolN	New. Iter.
8	40	10^{-12}	3
	80	10^{-12}	2
	160	10^{-12}	2
	320	10^{-12}	2
	640	10^{-12}	2

The number of CGS iterations in every Newton step never became higher than 1. If in any particular example the `shoot_CGS` routine doesn't converge this fast anymore, one has to think about changing relative or absolute tolerance in the iterative method.