Math 151B HW 5 Project Report

Justin JR Wang

5/15/14

User Guide:

fsolver.m : This function implements the solver functions in this program. It takes in arguments that make up the nonlinear system F(**x**)=**0**, then passes them to the actual solver functions. Let F, a function from R^n to R^n, be continuously differentiable with Jacobian matrix J(**x**). This function solves the nonlinear system F(**x**)=**0**, using Newton's method, Broyden's method, Steepest Descent method, and Homotopy method.

The calling syntax is:

```
function [ x, iter, runtime ] = fsolver( x0, fun, paras )
```

Inputs:

x0 is the initial guess for the solution by default
fun is a structure array by default containing the following fields:
- **.F** is the given function F(**x**): R^n to R^n
- **.J** is the Jacobian matrix J(**x**): R^n to R^nxn if available

paras is a structure array containing the following fields:
- **.option** specifies the method being used, which can be strings ‘newton’, ‘broyden’, ‘steep’, and ‘homotopy’
- **.tol** is the tolerance if available
- **.maxiter** is the maximal number of iterations allowed if available
- **.N** is only used in the homotopy method, and empty in other methods
- **.hoption** specifies the IVP solver used in the homotopy method, and empty in other methods

Outputs:
x is the obtained approximate solution
iter returns the number of iterations actually performed in those iterative methods so that you are able to monitor the algorithm or even adjust maxiter. It is defined as ‘nan’ in the homotopy method
runtime is the running time for the method being used, which are obtained by using the MATLAB stopwatch timers tic and toc. To make the comparison fair, the following two lines:
tic;
runtime = toc;
should be placed in fsolver at specific lines where each method starts and ends. In addition, the stopping criterion should be placed somewhere specific in order to get an approximate solution of the desired accuracy if maxiter is large enough.

newton.m : This function implements the Newton’s method.

Choose **x^**(0) in R^n. For k=1,2,...

$$\vec{x}^{(k)} = \vec{x}^{(k-1)} - J\big(\vec{x}^{(k-1)}\big)^{-1} F\big(\vec{x}^{(k-1)}\big)$$

The calling syntax is:

```
function [ x, iter, runtime ] = newton( x0, fun, tol, maxiter )
```

Inputs:
x0 is the initial guess for the solution by default
fun is a structure array by default containing the following fields:
- **.F** is the given function F(**x**): R^n to R^n
- **.J** is the Jacobian matrix J(**x**): R^n to R^nxn if available

tol is the tolerance
maxiter is the maximal number of iterations allowed

Outputs:
x is the obtained approximate solution
iter returns the number of iterations actually performed in those iterative methods so that you are able to monitor the algorithm or even adjust maxiter. It is defined as 'nan' in the homotopy method
runtime is the running time for the method being used, which are obtained by using the MATLAB stopwatch timers tic and toc.

broyden.m : This function implements the Broyden's method.

Choose $\vec{x}^{(0)} \in \mathbb{R}^n$, and let $A_0 = J(\vec{x}^{(0)})$ and $B_0 = A_0^{-1}$. For $k = 0, 1, \ldots$

$$\begin{cases} \vec{s}_{k+1} = -B_k F(\vec{x}^{(k)}) \\ \vec{x}^{(k+1)} = \vec{x}^{(k)} + \vec{s}_{k+1} \\ \vec{y}_{k+1} = F(\vec{x}^{(k+1)}) - F(\vec{x}^{(k)}) \\ B_{k+1} = B_k + \dfrac{(\vec{s}_{k+1} - B_k \vec{y}_{k+1}) \vec{s}_{k+1}^T B_k}{\vec{s}_{k+1}^T B_k \vec{y}_{k+1}} \end{cases}$$

The calling syntax is:

function [x, iter, runtime] = broyden(x0, fun, tol, maxiter)

Inputs:
x0 is the initial guess for the solution by default
fun is a structure array by default containing the following fields:
- **.F** is the given function F(**x**): R^n to R^n
- **.J** is the Jacobian matrix J(**x**): R^n to R^nxn if available
tol is the tolerance
maxiter is the maximal number of iterations allowed

Outputs:
x is the obtained approximate solution
iter returns the number of iterations actually performed in those iterative methods so that you are able to monitor the algorithm or even adjust maxiter. It is defined as 'nan' in the homotopy method
runtime is the running time for the method being used, which are obtained by using the MATLAB stopwatch timers tic and toc.

steep.m : This function implements the Steepest Descent method.

We consider normalizing the gradient to speed up. See Algorithm 10.3 for the selection of alpha hat.

Let $g(\vec{x}) = \|F(\vec{x})\|_2^2$, and choose $\vec{x}^{(0)} \in \mathbb{R}^n$. For $k = 0, 1, \ldots$

$$\begin{cases} \widehat{\alpha} = \underset{\alpha>0}{\operatorname{argmin}}\, g\left(\vec{x}^{(k)} - \alpha \dfrac{\nabla g(\vec{x}^{(k)})}{\|\nabla g(\vec{x}^{(k)})\|_2}\right) \\ \vec{x}^{(k+1)} = \vec{x}^{(k)} - \widehat{\alpha} \dfrac{\nabla g(\vec{x}^{(k)})}{\|\nabla g(\vec{x}^{(k)})\|_2} \end{cases}$$

The calling syntax is:

function [x, iter, runtime] = steep(x0, fun, tol, maxiter)

Inputs:
x0 is the initial guess for the solution by default
fun is a structure array by default containing the following fields:
- **.F** is the given function F(**x**): R^n to R^n
- **.J** is the Jacobian matrix J(**x**): R^n to R^nxn if available
tol is the tolerance
maxiter is the maximal number of iterations allowed

Outputs:
x is the obtained approximate solution
iter returns the number of iterations actually performed in those iterative methods so that you are able to monitor the algorithm or even adjust maxiter. It is defined as 'nan' in the homotopy method
runtime is the running time for the method being used, which are obtained by using the MATLAB stopwatch timers tic and toc.

homotopy.m : This function implements the Homotopy method. Choose **x**^(0) in R^n, and obtain the IVP system:

$$\begin{cases} \dfrac{d}{d\lambda}\vec{x}(\lambda) = -J(\vec{x}(\lambda))^{-1}F(\vec{x}(0)), \quad \lambda \in [0,1]; \\ \vec{x}(0) = \vec{x}^{(0)}. \end{cases}$$

Let lambda_i = i/N for i = 0, 1, …, N, solve the IVP system numerically at lambda_i using some IVP solvers, e.g., midpoint method and RK4.
The stopping criterion for iterative methods is:

$$\left\|x^{(k+1)} - x^{(k)}\right\|_{\infty} < tol.$$

Note that the backslash or the left matrix division operator is considered in MATLAB to compute J(**x**)^-1*F(**x**).

The calling syntax is:

function [x, runtime] = homotopy(x0, fun, N, hoption)

Inputs:
x0 is the initial guess for the solution by default
fun is a structure array by default containing the following fields:
- **.F** is the given function F(**x**): R^n to R^n
- **.J** is the Jacobian matrix J(**x**): R^n to R^nxn if available

N is the last value of the iterations, starting at 0
hoption specifies the IVP solver used in the homotopy method

Outputs:
x is the obtained approximate solution
runtime is the running time for the method being used, which are obtained by using the MATLAB stopwatch timers tic and toc.

odesolver.m :

This function solves the system of first-order IVPs using one-step methods or multistep methods, including at least the midpoint method and the Runge-Kutta method of order four. Define two nested functions rk4 and midpoint in this routine. Therefore, either ‘rk4’ or ‘midpoint’ will be option.
The calling syntax is:

function [yt] = odesolver(a, b, y0, h, f, argf, hoption)

Inputs:
a is the initial time

b is the final time
y0 is the initial condition
h is the step size
f is the function the defines the IVP
argf contains related parameters for f, which is optional
hoption can specify 'rk4' , and 'midpoint'

Outputs:
yt is the approximated solution to the initial value problem

Implementation:

The problem encountered is solving the nonlinear system of equations
fun.F = @(x)[15*x(1)+x(2)^2-4*x(3)-13 ; x(1)^2+10*x(2)-x(3)-11 ; x(2)^3-25*x(3)+22];
From the results, it seems that Steepest Descent method is pretty bad, probably because the initial guess is somewhat close to the solution.

To test if my code worked, I did the example exercises with all four exercises, and since they are the same system, with the same F(x) and J(x), we expect the answer to be similar for the different methods, and after testing, we see that they are. I had some problems deciding how to declare fun.F and fun.J since each component is a function of x1, x2, x3. At first it was cumbersome working with cells, since I made fun.J and fun.F cells with anonymous functions for each component, and I did some nested for loops to evaluate at each component. It just made everything harder, and in the end, Jeffrey suggested that I just let F and J be an anonymous function that returns a vector and matrix, respectively.

fun.J = @(x) [15 2*x(2) -4 ; 2*x(1) 10 -1; 0 2*x(2)^2 -25];

Then we let x1 be x(1), and proceed by evaluating F and J at the vector x. Also, there was a better way to write g, the two norm of F, for the steepest descent method by doing this: g=@(x) sum((fun.F(x)).^2).

There were some errors that have become easier to deal with, such as matrix size matching, and input arguments matching. Frequently, I would get some subscript indices must be real value errors, and this was usually because the functions that I were trying to use were actually scalars.

I tried to make my codes as efficient as possible so that the comparison is fair, especially to reduce the number of function evaluations. The running time varies with programs and computer configurations.

Solutions:

Solve the nonlinear system by applying the aforementioned methods with the initial guess x^(**0**)=**0** and compare the performance of each method in terms of the number of iterations, the running time, and the error, norm infinity of (**x^(**k)-**x***), where **x*** is the actual solution which can be estimated by Newton's method with tolerance 'eps' Use the homotopy method with the midpoint method and the Runge-Kutta method of order four respectively, where N=10,20,50. Which performs better?

```
newton_x =

   1.036400470365324
   1.085706550748083
   0.931191442454289


newton_iter =

     4


newton_runtime =

     6.100504068864156e-04


broyden_x =

   1.036400470315312
   1.085706550772970
   0.931191442333108
```

```
broyden_iter =

           6


broyden_runtime =

           6.767605206528173e-04


steep_x =

   1.048522955950043
   1.048545087492462
   0.921173857412267


steep_iter =

           10


steep_runtime =

   0.006433450926259


homotopy_x =

   1.031944863526471
   1.084914279318542
   0.914052981922199


homotopy_iter =

   NaN


homotopy_runtime =

   0.002139855187738
```

```
actual_x =

   1.036400470329211
   1.085706550741678
   0.931191442315390


actual_iter =

     8


actual_runtime =

     7.917675493632203e-04
```

Newton's method did 4 iterations, Broyden's did 6, and Steepest Descent did 10. So Newton's method had the fewest number of iterations, and Broyden's just a few more; however, Steepest Descent had many more iterations.

In terms of runtime, Newton's is 5.9194e-04; Broyden's is 6.6288e-04, Steepest Descent is 0.0064; and Homotopy is 0.0016. So Newton's and Broyden's both have fast runtimes, much faster than Steepest Descent and Homotopy; however, the Steepest Descent runtime is many times slower than Homotopy.

The Actual Solution which I estimated by Newton's method with tolerance 'eps' is:

```
newton_x =

   1.036400470329211
   1.085706550741678
   0.931191442315390


newton_iter =

     8


newton_runtime =
        3.492470661888090e-04
```

So now we'll calculate the error for each method:

```
error_newton =
        1.388988923878287e-10
error_broyden =

        3.062939593068137e-07
error_steep =
  0.037161463249216
error_homotopy =
  0.017185965976594
```

So the trend of the methods is similar when we compare error. Here, Newton's method and Broyden's method have errors that are very small, around the 10^-10ths place. It seems that with this given tolerance, Broyden's method had smaller error than Newton's. But both Broyden and Newton have much much smaller error than Steepest Descent and Homotopy. And again Steepest Descent had the highest error.

Now let's see how changing the number of iterations affects homotopy method. I change this parameter in fsolver.m.
Using the 'hoption' = rk4, when N=10 we get:

```
error_homotopy =
  0.017138460393191
homotopy_runtime =
  0.002139855187738
```

When N=20, we get:

```
error_homotopy =
  0.017138470805761
homotopy_runtime =
  0.002352965324928
```

When N=50, we get:

```
error_homotopy =
  0.017138471475133
homotopy_runtime =
  0.006468768045312
```

So as N increases, error increases, so N=10 performs better, the runtime is also best when N is low, as expected.

When we use the Midpoint Method to solve the IVP in homotopy, so hoption= 'midpoint'.

```
 >>paras.hoption='midpoint';
[ homotopy_x, homotopy_iter, homotopy_runtime ]=fsolver( [0;0;0], fun, paras )
>> error_homotopy=norm( homotopy_x-actual_x  ,Inf)
```

Then the results are:
When N=10:

```
error_homotopy =
  0.017185965976594

homotopy_runtime =

  0.001183576271846
```

When N=20:

```
error_homotopy =

  0.017150395610859

homotopy_runtime =

  0.002576640412262
```

When N=50:

```
error_homotopy =

  0.017140383470064

homotopy_runtime =

  0.002289877479782
```

So we see that in the Midpoint case, as N increases, the runtime increases, of course, and it increases at a fast rate from 10 to 20, and not as fast from 20 to 50. In terms of error, as N increases, error decreases! So unlike in the Runge-Kutta case, N=50 performs better in terms of error.