Math 151B HW 4 Project Report

Justin JR Wang

5/14/14

User Guide

odesolver.m : This function implements the solver functions in this program. It takes in arguments that make up an initial value problem, then passes them to the actual solver functions. For the linear shooting method and the linear finite-difference method, the Runge-Kutta equations are in odesolver.
Solves the system of first-order IVPs using one-step methods or multistep
methods. Notice that f: [a,b]XR^k--->R^k is a vector-valued function and
yt returns a (N+1) X k matrix. At least, you should include the Runge-Kutta method of order four, and define a nested function rk4 in the

odesolver.m. Note that argf will be very useful when solving the z(x) -
involved IVP in the nonlinear shooting method (z(x) =z(x,t)) ie z''(x)=fy(x,y,y')*z(x) + fy'(x,y,y')*z'(x), where y,y' are from the y(x)-involved IVP and will be updated for each tk. You could use this
routine to get a solution to a high-order IVP.
The calling syntax is: function [yt] = odesolver(a, b, y0, h, f, argf, option)

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 fun, a structure array containing the fields .f, .dfdy, .dfy, and .r
argf contains related parameters for fun, which is optional
option can specify 'linshoot', 'nlinshoot', 'lindiff', and 'nlindiff'
Outputs:
- yt is the approximated solution to the initial value problem. It is a function f with respect to t, y(t).

bvpsolver.m : This function solves the two-point second order BVP. It can do both linear and nonlinear BVPs.
The calling syntax is : function [yx, error] = bvpsolver(xinit, yinit, fun, h, option)

Inputs:
xinit is a vector containing the endpoints of the interval ie a and b
yinit is a vector containing the function values at the endpoints ie alpha
and beta
fun is a structure array containing the following fields:
.f the function f(x,y,y') which can be defined as
fxydy=f(x,y,dy,argf) in
a separate function m-file where argf contains related parameters, or an
anonymous function f=@(x,y,dy,argf)(...); you may skip the argf if it is
not necessary
.dfdy the function fy'(x,y,y') which is reduced to P(x) in the linear case
.dfy the function fy(x,y,y') which is reduced to q(x) in the linear case
.r the function r(x) only in the linear case. Note that you may skip this
field fo rhte nonlinear case
For example, if the linear BVP has f(x,y,y')=x^2*y'+x*y+x in (1), then you

may define in the main.m
fun.f=@(x,y,dy)(x.^2.*dy+x.*y+x);
fun.dfdy=@(x) (x.^2);
fun.dfy=@(x)(x);
fun.r=@(x)(x);
or: fun=struct('f', @(x,y,dy)(x.^2.*dy+x.*y+x), 'dfdy', @(x)(x.^2), 'dfy', @(x)(x), 'r', @(x)(x));
the structure array simplifies syntax and groups arguments of similar
attributes, which is very convenient in coding.
h is the step size: note that xi=a+ih for i from 0 to N+1, and b-a=(N+1)h
when using the finite-difference method, while b-a=NJh when using the
shotting method. We determine N nodes in shotting methods, but N-1
interior nodes in finite-difference methods.
option is a string specifying hte method to be used which has the following choices: 'linshoot' - linear shooting method; 'nlinshoot' - nonlinear shooting method; 'lindiff' - linear finite-difference method;
'nlindiff' - nonlinear finite-difference method

Output:

- yx is a vector storing the approximations of the soln at xi
error is the global error defined as max i of abs(y(xi)-wi)

linshoot.m : This functions implements the linear shooting method to approximate the solution of the boundary-value problem:
-y'' + p(x)y' + q(x)y + r(x) = 0, a<=x<=b, y(a)=alpha, y(b)=beta:
the procedure is explained in the textbook in Algorithm 11.1.
%Note: equations 11.3 and 11.4 are written as first-order systems and,
%solved
The calling syntax is : function [yx] = linshoot(a,b,ya,yb,h,p,q,r)

Inputs:

a and b are endpoints of BVP
ya is boundary condition alpha
yb is boundary condition beta
h is the step size
p,q,r are the respective function handles of p(x), q(x) and r(x) of BVP

Output:

%output yx is a vector storing the approximations of the solution at xi

nlinshoot.m : This function implements the nonlinear shooting method.
It implements algorithm 11.2, nonlinear shooting method where newton's methodis used to update tk.
To approximate the solution of the nonlinear boundary-value problem

y'' = f(x,y,y'), a<=x<=b, y(a) = ya, y(b) = ya:

The calling syntax is: function [yx] = nlinshoot(a, b, ya, yb, h, fun, itermax, tol)

Inputs:

a and b are the endpoints of BVP
ya, and yb are boundary conditions alpha and beta
h is the step size used to calulate n, which is the number of subintervals
fun is the same as in bvpsolver.m
itermax is the maximal number of iterations allowed
tol is the tolerance in the stopping critera abs(y(xn+1, tk)-yb) <tol

Output:

output yx is the vector of approximations w(1,i) to y(xi) for each i=0,1,...,n

lindiff.m : This function implements the linear finite-difference method to solve the linear BVP (2). Its setup is similar to Algorithm 11.3 in the textbook; however, in order to solve the tridiagonal system of the form Aw=b, I first
construct the coefficient matrix A by using the command spdiags([d1,d2,d3], -1:1,N,N) which creates a NxN sparse band matrix.
Then I use the matrix left division mldivide, ie A/b, to obtain the solution to the linear system.
The calling syntax is: function [yx] = lindiff(a, b, ya, yb, h, p, q, r)

Inputs:

a and b are endpoints of BVP
ya is boundary condition alpha
yb is boundary condition beta
h is the step size

p,q,r are the respective function handles of p(x), q(x) and r(x) of BVP

Output:

output yx is a vector storing the approximations of the solution at xi

nlindiff.m : This function implements the nonlinear finite-difference method. It applies the finite-difference method and the Newton's method for nonlinear systems to solve the nonlinear BVP.

Note that itermax and tol are set by default in the written functions for the given exercises, since they are not passed to nlindiff.m through bvpsolver.m.

%the Jacobian matrix J cna be constructed similar to A in the linear case.

Be careful that J varies at each iteration which involves the function fy,

and fy'. Similar to lindiff, instead of solving a linear system with ethe

coefficient matrix J, the matrix left division '\' can be used.

the initial values for wi for i from 1 to N are chosen as what are used
in Algorithm 11.4 on page 693 step 2.
The calling syntax is: function [yx] = nlindiff(a, b, ya, yb, h, fun, itermax, tol)

Inputs:

a and b are endpoints of BVP
ya is boundary condition alpha
yb is boundary condition beta
itermax and tol are the maximal number of itermations allowed and the
tolernace respectively in the Newton's method for nonliner systems

Output:

output yx is a vector storing the approximations of the solution at xi

Implementation:

To test whether my code worked properly or not, I tested the examples in sections 11.1 to 11.4 of the textbook on pages 676,

683, 689, and 695, which are the linear shooting method, the nonlinear shooting method, the linear finite-difference method, and the nonlinear finite-difference method, respectively.
The sample code that I input into the command window are included as comments at the end of each of those four functions.
Here is the code:

```
%Test example on page 676 in chapter 11.1
%linshoot(1,2,1,2, .1, @(x)(-2/x), @(x)(2/x^2), @(x)
sin(log(x))/x^2)

%Test example on page 688 in chapter 11.3
%linshoot(1,2,1,2, .1, @(x)(-2/x), @(x)(2/x^2), @(x)
sin(log(x))/x^2)

%Test example on page 683 in chapter 11.2
%fun.f = @(x,y,dy) -(y./8).*dy+x.^3./4+4;
%fun.dfdy = @(x,y,dy) -(y./8);
%fun.dfy = @(x,y,dy) -(dy./8);
%fun.r = @(x,y,dy) x.^3./4+4;
%nlinshoot( 1, 3, 17, 43/3, .1, fun, 10, 10^-5 )

%Test example on page 695 in chapter 11.4
%fun.f = @(x,y,dy) -(y./8).*dy+x.^3./4+4;
%fun.dfdy = @(x,y,dy) -(y./8);
%fun.dfy = @(x,y,dy) -(dy./8);
%fun.r = @(x,y,dy) x.^3./4+4;
%nlindiff( 1, 3, 17, 43/3, .1, fun, 10, 10^-5 )
```

They all work perfectly!

After hours of debugging errors, in particular, indexing errors, and matrix size matching errors, all of the codes work perfectly and give the results given in the accompanying tables in the textbook. Some memorable errors are as follows: There was indexing nightmare in nlindiff.m because we first calculate w, the vector of the iterates that converge to the solution calculated using alpha and beta, which gets added to a correction vector calculated in the algorithm. At first, the values obtained were off in the tenths place, which I thought might be caused by an incorrect shift in adding the vectors, since there were vector addition issues, but in the end, it turned out that the main issue was calculating the vector w(i), and the indexing in the for loop. I know that for the finite difference methods, the book's algorithms tell you to make n=(b-a)/h -1, but I just used the formula without the -1, and wrote the rest of the function accordingly. Another memorable milestone in nlindiff.m was getting spdiags to work, since it's a pretty specific prewritten function and everything had to match. To get the Jacobian matrix J, I had to make spdiags work by taking the transpose so the size works with and use n-1 as the size.
Also, it took some time to figure out how to pass parts of the Runge-Kutta equations of linshoot.m and nlinshoot.m as written in the book algorithms into odesolver.m. I just made yt into a cell and passed the two necessary vectors for each method. I might have liked it better to just have the Runge-Kutta equations in linshoot.m and nlinshoot.m, and make them standalone functions. Another source of error was making sure the endpoints were

added to the solutions, alpha and beta, but usually this was an easy fix by just concatenating ya and yb into the solution vector.
I learned a useful debugging technique to make sure vectors and matrices match in size by using size( ). Just put a breakpoint below and write the sizes you want without semicolons to display them in the command window. Of course, sometimes you can just test stuff directly in the command window.

Solutions:

If you run my main.m, all the approximations to exercises 11.1.4b, 11.2.4b, 11.3.4b, 11.4.4b, which are yx1, yx2, yx3, and yx4, respectively are calculated using linshoot.m, nlinshoot.m, lindiff.m, and nlindiff.m, respectively. Also, their respective errors and actual solutions are calculated using bvpsolver.m.
I used itermax=10 for the nonlinear methods. The way the assignment is written, you can't use bvpsolver to pass itermax to nlinhsoot and nlindiff, I think.

I plotted the linear BVP example exercise in figure(1), and figure (3), along with the actual solution to 1 and 3.
Blue is Exercise 11.1.4b using linshoot.m.
Red is Exercise 11.3.4b using lindiff.m.
Black is the actual solution to 1 and 3.

I plotted the Non-linear BVP example exercise in figure(2)and figure (4), along with the actual solution to 2 and 4.
Blue is Exercise 11.2.4b using nlinshoot.m.

Red is Exercise 11.4.4b using nlindiff.m.
Black is the actual solution to 2 and 4.
It's a little hard to see through the overlapping, but they are surely there, since if you plot each of the lines individually, you can get a graph of it all by itself. But it's not too useful since we want them all on the same graph for sake of comparison.
In figure 1 especially, the blue and the black are hard to distinguish, but they give what looks like the same thing when run individually.
Basically figure (1) is exercise 1, and so on.
And here they are!:

Figure 1 is on the left and figure 3 is on the right.
As you can see, the blue line in figure 1 is really hard to distinguish, I changed it to red temporarily to better compare with 3 and it's the same deal, you can hardly see it since they overlap so much, so for this LINEAR BVP, we see that the shooting method is better than linear difference method.

Now, numerically,

```
>> error1

error1 =
   2.1242e-05
>> error3
error3 =
        0.0012
```

So error 1 is much smaller than error 3, which supports the conclusion.

Figure (2) is on the left and figure (4) is on the right.