THE COLLEGE OF WILLIAM AND MARY

Programming Assignment 2

Alexander Powell

April 16, 2015

1

The following shows the modified code for F.m implemented in MATLAB

```
function y = F(x)
% Computes the system of nonlinear equations from the given differential
% equation.
    y = zeros(length(x),1);

    a = 1;
    b = 3;
    h = 0.1;
    alpha = answ_func(a);
    beta  = answ_func(b);
    xi = zeros(((b - a)/h) + 1,1);
    N = ((b - a)/h) - 1;
    for i = 1:((b - a)/h + 1)
        xi(i,1) = a + (i - 1) * h;
    end

    yi = zeros(((b - a)/h) + 1, 1);
    for j = 1:(length(yi))
        yi(j,1) = answ_func(xi(j,1));
    end

    for k = 1:N
```

```
        if (k == 1)
            y(k) = 2 * yi(k+1) - yi(k+2) + (h^2) * func(x(1),yi(k+1),...
            (yi(k+2) - yi(k))/(2 * h)) - alpha;
        elseif (k == N)
            y(k) = (-1) * yi(k) + 2 * yi(k+1) + (h^2) * func(x(N),yi(k+1),...
            (yi(k+2) - yi(k))/(2 * h)) - beta;
        else
            y(k) = (-1) * yi(k) + 2 * yi(k+2) - yi(k+2) + (h^2) * func(x(k),...
            yi(k+1),(yi(k+2) - yi(k))/(2 * h));
        end
    end
end
```

Below is the calculation for the exact Jacobian, or exJF.m.

```
function JF = exJF( x )
% Exactly calculates the Jacobian matrix of F.

    a = 1;
    b = 3;
    h = 0.1;
    xi = zeros(((b - a)/h) + 1,1);
    N = ((b - a)/h) - 1;
    for i = 1:((b - a)/h + 1)
        xi(i,1) = a + (i - 1) * h;
    end
    yi = zeros(((b - a)/h) + 1, 1);
    for j = 1:(length(yi))
        yi(j,1) = answ_func(xi(j,1));
    end

    JF = zeros(N,N);

    for i = 1:N
        for j = 1:N
            if ((i == (j - 1)) && (j >= 2) && (j <= N))
                JF(i,j) = -1 + (h/2) * fyp(x(i),yi(i+1),...
                (yi(i+2)-yi(i)/(2 * h)));
            elseif (i == j)
                JF(i,j) = 2 + (h^2) * fy(x(i),yi(i+1),...
                (yi(i+2)-yi(i)/(2 * h)));
            elseif ((i == j + 1) && (j >= 1) && (j <= (N - 1)))
                JF(i,j) = -1 - (h/2) * fyp(x(i),yi(i+1),...
                (yi(i+2)-yi(i)/(2 * h)));
            else
```

```
                JF(i,j) = 0;
            end
        end
    end
    JF = sparse(JF);
end
```

Also, it uses the answ_func.m function in MATLAB to calculate the exact value of the equation

$$f(x) = x^2 + \frac{16}{x}$$

. Shown below is that code.

```
function out = answ_func(x)
% This function models the originial differential equation.
    out = (x^2) + (16/x);
end
```

Furthermore, it always uses the originial differential equation, func.m calculated using the code below.

```
function out = func(x,y,yp)
    out = (1/8) * (32 + 2 * x^3 - y * yp);
end
```

The exact Jacobian is computed using the partial derivatives of f_y and f'_y. They are calculated in MATLAB by the following.

```
function out = fy(x,y,yp)
    out = (-1 * yp) / 8;
end
```

```
function out = fyp(x,y,yp)
    out = (-1 * y) / 8;
end
```

When $h = 0.1$ the infinity norm of the errors, defined by $|\omega_i - y(x_i)|$, is 0.002462. When $h = 0.01$ the infinity norm of the errors is 1.231×10^{-5}.
Our results behave like $O(h^2)$ which makes sense because if Newton's method converges, then the order of convergence is 2. However, we should keep in mind that the convergence of the iterative method is dependent on the given initial iterate.

Newton's method took 8 iterations to converge for $h = 0.1$ and 14 iterations for $h = 0.01$. This makes sense because we fed in a very good approximation to the Newton iterative method. When other iterative techniques like the Chord method and Shamanskii method were used, relatively similar results were found. The Chord method took 9 iterations when $h = 0.1$ and 17 when $h = 0.01$. Likewise, Shamanskii's method using $M = 2$ resulted in 8 iteration when $h = 0.1$ and 12 when $h = 0.01$.
The following table displays the times it took for the following computations to complete. The results were calculated using MATLAB's built-in tic and toc methods.

Table 2.1: Iteration Times for Algorithms

h	Newton	Chord	Shamanskii
0.1	0.00348	0.002876	0.047709
0.01	0.01203	0.038763	0.134587

Based on these rates of convergence, all methods do a pretty good job given the initial iterate. If any of the three was slower than the others it would probably by Shamanskii's algorithm, as it took a few more iterations to converge. Also, it should be noted that all of these results were found using the tolerance as 10^{-8}.