CS513, Fall 2000
Prof. Amos Ron
Assignment #3 Solutions
Prepared by Houssain Kettani

1. The first hint in the problem forces s to be invertible. The second hint refers to the MATLAB function `meshgrid`. We can make use of that as follows:

```
function [r] = lattice(m, n, s)
% function [r] = lattice(m, n, s)
% m and n are integers and s is a 2-by-2 invertible integer
% matrix.

[x,y] = meshgrid(1:m, 1:n);
pts = [x(:)'; y(:)'];
% Now pts is a 2-by-(m*n) matrix whose columns list all points
% in the square [1,m]x[1,n].

% Check which columns of pts have integer solutions to
% s*? = pts(:,i)
sloc = inv(s)*pts;
ind = find(all(sloc == round(sloc)));

r = pts(:, ind);
```

   We invoke this function with the arguments requested in the problem:

   (a)
```
lattice(4, 6, [2, 0; 0, 2])
ans =
     2     2     2     4     4     4
     2     4     6     2     4     6
```
   (b)
```
lattice(5, 6, [1, 1; 1, -1])
ans =
  Columns 1 through 12
     1     1     1     2     2     2     3     3     3     4     4     4
     1     3     5     2     4     6     1     3     5     2     4     6
  Columns 13 through 15
     5     5     5
     1     3     5
```

2. We get a particularly simple and neat solution to this problem by making use of the `eval` and `diff` functions as mentioned in the hint.
   Additionally, we use the `vectorize` function so we can supply vectors as arguments and not just scalars.

```
fs = 'x^2*exp(-3*x^2)+(x/40)^2';
dfs = diff(fs);
ddfs = diff(fs,2);
% the following command makes the f(x) accepts x as a vector
f = vectorize(fs);
df = vectorize(dfs);
ddf = vectorize(ddfs);

x = -10:0.01:10;
% evaluate f at x
f = eval(f);
df = eval(df);
ddf = eval(ddf);

subplot(2,2,1)
plot(x,f)
title('A plot of f(x)')
xlabel('x')
ylabel('f(x)')

subplot(2,2,2)
plot(x,df)
title('A plot of f''(x)')
xlabel('x')
ylabel('f''(x)')

subplot(2,2,3)
plot(x,ddf)
title('A plot of f''''(x)')
xlabel('x')
ylabel('f''''(x)')
% find where the function attains its global maximum value
disp('The maximum value of f(x) occurs at')
x_max = x(find(f==max(f)))
```

This gives the following answer:

```
x_max =
   -0.5706    0.5706
```

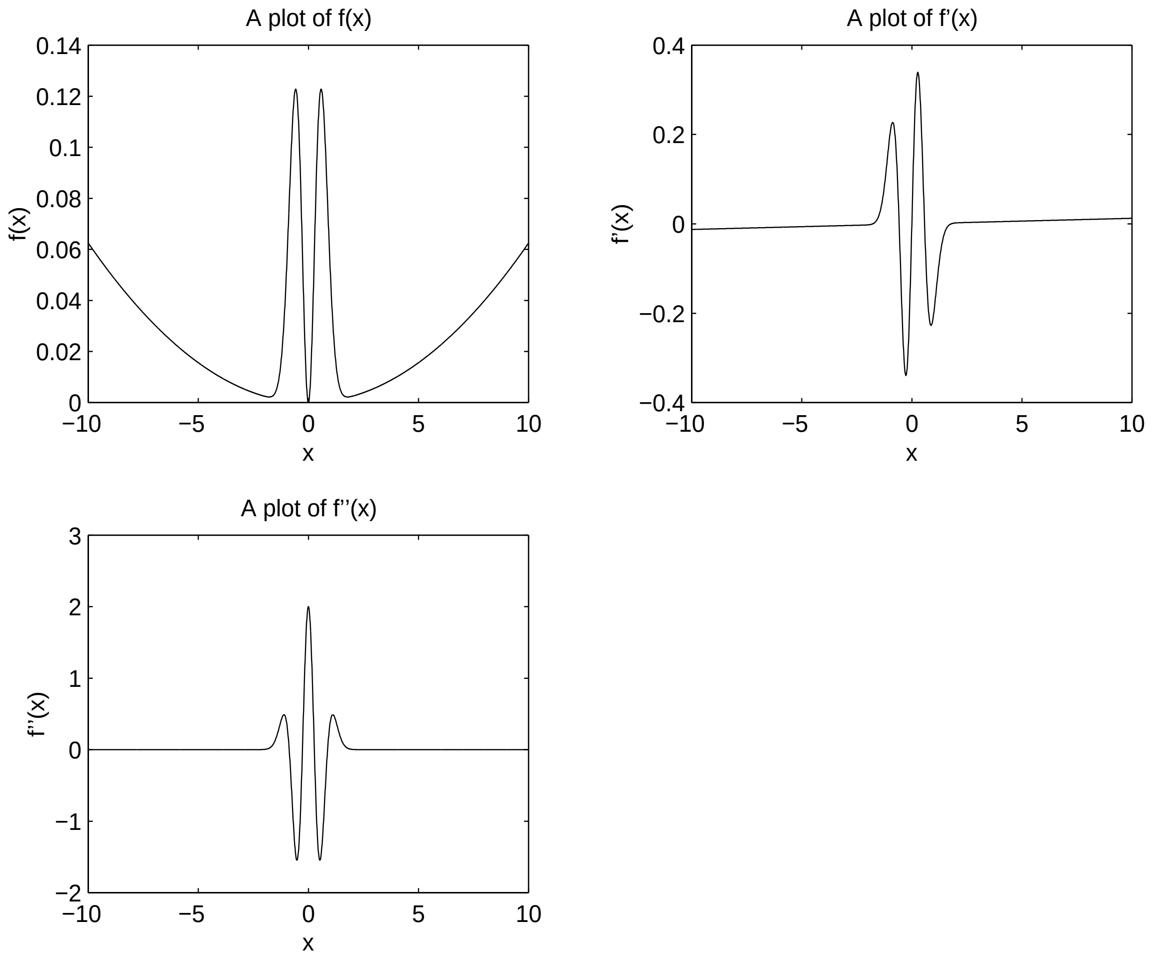

Figure 1:A plot of the corresponding functions of Q.2

3. In the previous homework, we observed that the complexity of sloppy_qr is asymptotically $2.5n^{3.9}$. To improve upon this code, we need to see what its biggest problem is. This turns out to be the two matrix multiplications QR and HR. For matrices of order n, matrix multiplications take n^3 operations. Since we run each of these steps n times, we are immediately guaranteed that our problem will be $O(n^4)$.

   The solution to this problem is simply not to explicitly calculate H! Using the definition of H we can just write

$$Q = Q * (I - 2 * w * w')$$
$$R = (I - 2 * w * w') * R$$

   Applying the distributive property of matrix multiplication we get

$$Q = Q - 2 * Q * w * w'$$
$$R = R - 2 * w * (w' * R)$$

   Notice the parentheses in the equation for R. This will be very important! Now analyze the number of operations these expressions take (just look at Q):

- scalar times nxn matrix: n^2 (produces nxn matrix)
- matrix times nx1 vector: n^2 (produces nxn matrix)
- vector times 1xn vector: n^2 (produces nxn matrix)
- matrix times nxn matrix: n^2 (produces nxn matrix)

All told, computing Q takes only $O(n^2)$ time. R can be analyzed similarly, but it is in doing so that we realize the importance of the parentheses. Without these, MATLAB would compute "2*w*w'*R" from left to right, and we would end up multiplying a matrix by a matrix. By forcing "w'*R" to be computed first, we make sure that we avoid the matrix multiplication and just have operations which take n^2 time.
By applying the same methodology discussed in the previous homework (see figure 2), we observe that the complexity of the modified sloppy_qr (better_qr) and qr codes is $10n^3$ and $0.6n^3$ respectively.

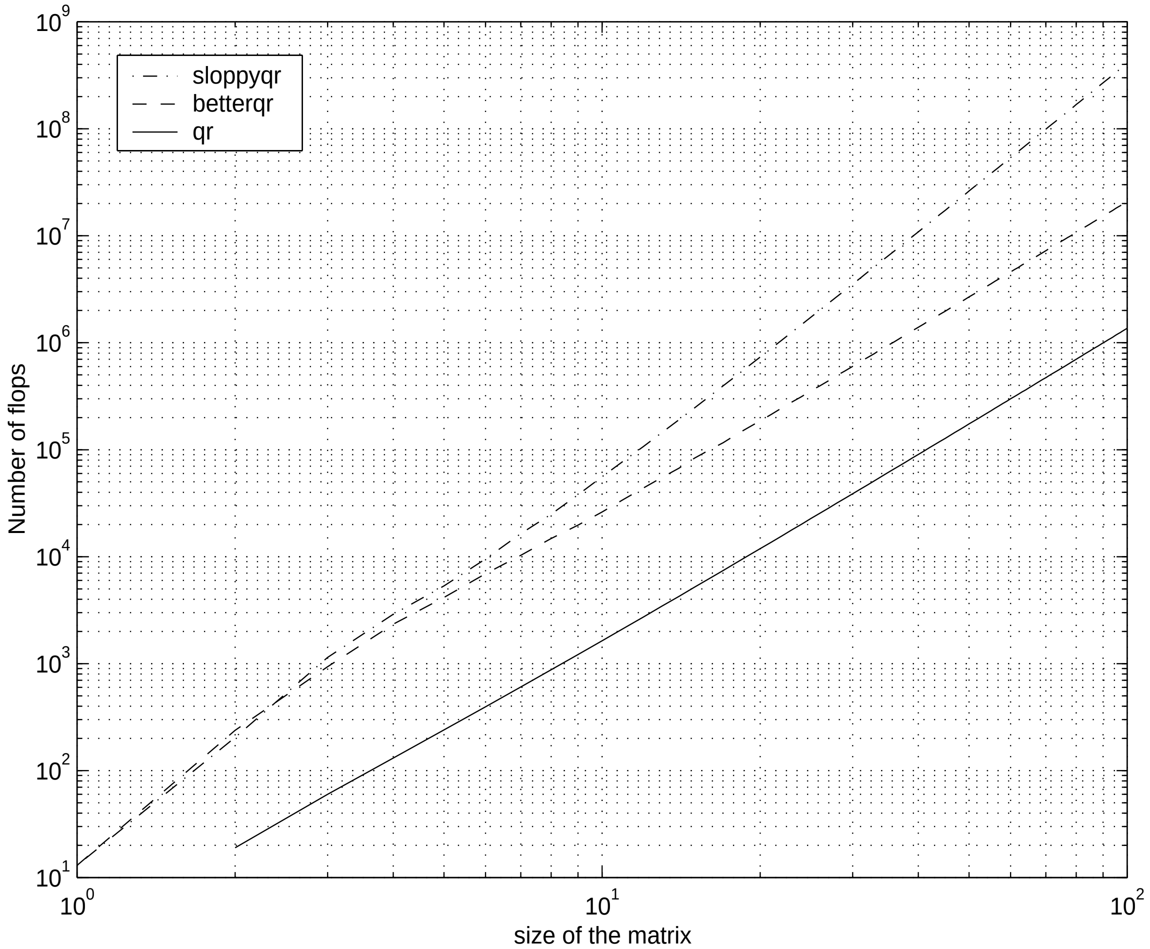

Figure 2:Comparison between order of complexity in sloppy_qr, its better_qr and qr

4. (a) The required MATLAB code is:

```
m = length(T);
A = (T'*ones(1,k+1)).^(ones(m,1)*(0:k));
```

(b) To achieve the required task, we use the following code:

```
% We approximate the function f(t) over the interval [a,b],
% by evaluating f(t) over m equaly spaced points from that interval,
% and minimize the mean squares error between f(t) and a polynomial
% of degree k. the variable f is a string variable.
function appro(a,b,m,k,f)
t = linspace(a,b,m);
% we use the technique discussed in the previous question
A = (t'*ones(1,k+1)).^(ones(m,1)*(0:k));
f = eval(vectorize(f))';
% Want to solve A'Ax = A'f, where x is a vector containing the coefficients
% of the required polynomial.
[Q,R] = qr(A);
% We modify the matrices Q and R, to get rid of redundant computations
Qh = Q(:,1:k+1);
Rh = R(1:k+1,:);
% So now we just need to solve Rh*x=Qh'*f
x = inv(Rh)*Qh'*f;
% The approximated function is af:
af = A*x;
plot(t,af,'--')
hold on
plot(t,f,'-.')
Error = norm(af-f)
```

(c) To answer this question, we type `appro(0,3,30,3,'tan(t/3)')` and `appro(0,3,30,3,'abs(t-1)')` which gives Figures 3 and 4 with Error 0.023 and 0.398 respectively.

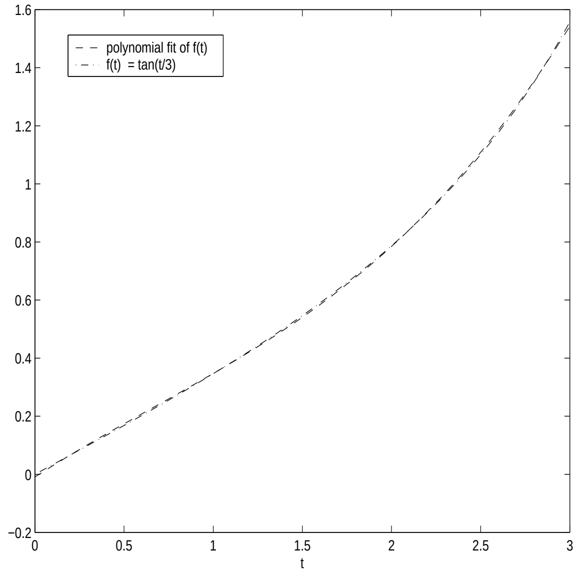

Figure 3: A plot of tan(t/3) and its polynomial fit

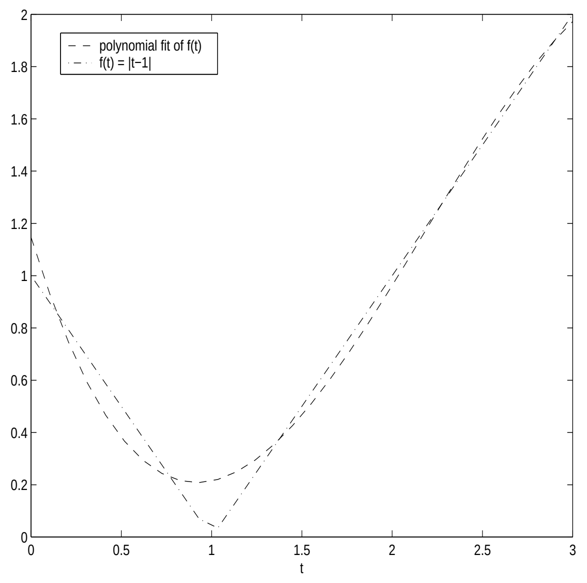

Figure 4: A plot of —t-1— and its polynomial fit

It is apparent that we have bad approximation when the function is not smooth.

What is wrong with `for` loop? You should use `for` loops very sparingly in MATLAB. Compare the following two solutions to question 2:

```
function [r] = lattice(m, n, s)
% function [r] = lattice(m, n, s)
% m and n are integers and s is a 2-by-2 invertible integer
% matrix.

% Instead of using the command
%    [x,y] = meshgrid(1:m, 1:n);
% we demonstrate the use of repmat:
x = repmat(1:m, n, 1);
y = repmat((1:n)', m, 1);
pts = [x(:)'; y(:)'];
sloc = inv(s)*pts;
ind = find(all(sloc == round(sloc)));
r = pts(:, ind);

function [r] = withForLoops(m, n, s)

r = zeros(2, m*n); %preallocate to minimize performance hit
count = 0;
sinv = inv(s);
for (j = 1:m)
  for (k = 1:n)
    v = inv(s)*[j; k];
    if (all(v == round(v)))
```

```
      count = count + 1;
      r(:, count) = [j; k];
    end
  end
end
r = r(:, 1:count);
```

And compare the floating point operations and the execution time of the two:

function call	flops	time (sec)
`lattice(512, 512, [1, 1; 1, -1])`	2097215	1.3
`withForLoops(512, 512, [1, 1; 1, -1])`	14024768	80.1

This is a 60-fold difference in execution time! You should usually consider converting long `for` loops into other operations that execute faster in MATLAB if you can. As a rule of thumb, if you can write your code in terms of vector or matrix multiplications and additions instead of looping over all the elements of vectors and matrices and vectorize your function calls, it will execute faster that way, and sometimes it may execute much faster. However, it can be quite tricky at times to write code with few (if any) `for` loops. In addition to vector and matrix operations, you should also keep in mind the command `repmat` which was used in the example above.