Contents

1 Problem 1 2
1.1 a) ......................................................... 2
1.2 b) ......................................................... 2

2 Problem 2 3
2.1 a) ......................................................... 3
2.2 b) ......................................................... 4
2.3 c) ......................................................... 4
   2.3.1 Fibonacci search .................................... 4
   2.3.2 Golden Section search ............................... 4
   2.3.3 Newton-Raphson search ............................ 5
   2.3.4 Comparison ......................................... 5

3 Problem 3 6
3.1 Application of the Golden Section Search ..................... 6
3.2 Application of the Newton-Raphson Search .................... 6

4 Problem 4 6
4.1 a) ......................................................... 6
4.2 b) ......................................................... 7
4.3 c) ......................................................... 7

A Bounding 8
A.1 M-file .................................................. 8
A.2 Output .................................................. 9

B Fibonacci search 10
B.1 M-file .................................................. 10
B.2 Output .................................................. 14

C Golden Section search 18
C.1 M-file .................................................. 18
C.2 Output .................................................. 21
1 Problem 1

1.1 a)

In the general case, where all we know is that \( a < p < q < b \), the positioning of \( p \) and \( q \) might be quite arbitrary. They may be near the centre, as with the Fibonacci and Golden Section searches, or they may be quite near to \( a \) and \( b \) respectively, as follows:

\[
\begin{array}{cccc}
| & \vdots & \vdots & | \\
\hline
a & r? & p & q \\
\hline
& & & b
\end{array}
\]

In this latter case, fixing \( r < p \) leads to a quite unbalanced search. Any choice of \( r \) not equal to \( a \), \( p \) or \( q \) would allow us to use \( f(r) \) with \( f(p) \) to narrow the search exactly as we did using \( f(q) \) and \( f(p) \) for the first iteration. So why would we choose an \( r \) that unbalances the search in this way?

The answer is that we have a little bit more information than at the first iteration, namely that we know \( f \) has a net increase from \( p \) to \( q \). This may indicate the minimum is more likely to be in \( [a, p] \) than in \( [p, q] \) (i.e. that \( f \) is strictly increasing from \( p \) to \( q \)). Therefore, we would like the next test to be able to narrow the search to exactly \( [a, p] \). If \( r > p \), it is unable to do this. Setting \( r < p \) is a heuristic which we suspect will manage to narrow down the search to \( [a, p] \), and thus the search converges faster.

It is worthwhile to note that the Fibonacci and Golden Section searches both choose \( p, q \) and subsequent \( r \)s so that this heuristic can be used while also keeping the search approximately balanced. (Perfectly balanced in the case of Fibonacci.)

1.2 b)

The Fibonacci search divides the interval to be searched into two overlapping regions of equal length, \( [a, q] \) and \( [p, b] \). At the next iteration, the search will have been reduced to one of these regions. The function is evaluated at both \( p \) and \( q \), but one of these will be reused, so each iteration corresponds to a single evaluation of \( f \).

W.L.O.G, say the next iteration narrows the search to \( [a, q] \) and \( p \) is reused. Then, again W.L.O.G, say the search narrows to \( [a, p] \). Say we started our search with \( n \) available \( f \)-calculations on an interval of length \( (b-a) \). In order to perform
this search, we are performing a subsearch with \( n - 1 \) available \( f \)-calculations on an interval of length \((q - a)\), followed by a subsearch with \( n - 2 \) available \( f \)-calculations on an interval of length \((p - a)\). Since the search is balanced, the particular sub-intervals we’ve chosen have not affected the lengths in question. In light of this, define \( l_k \) to be the length of the interval in this search when \( k \) \( f \)-calculations remain available. Now,

\[
\begin{align*}
(q - a) &= l_{n-1} \\
&= (b - p) \\
l_{n-1} + l_{n-2} &= (q - a) + (p - a) \\
&= (b - p) + (p - a) \quad \text{by (1)} \\
&= (b - a) \\
&= l_n
\end{align*}
\]

Define \( F_n \) such that \( F_n(\alpha) \) is the largest interval that can be reduced to size \( \alpha \) using \( n \) iterations of the Fibonacci search. Taking the relationship between search size and subsearch size from (2), we see that

\[
F_n(\alpha) = F_{n-1}(\alpha) + F_{n-2}(\alpha) \quad \text{(3)}
\]

Furthermore, zero or one \( f \)-calculation cannot reduce an interval at all, so

\[
F_0(\alpha) = F_1(\alpha) = \alpha \quad \text{(4)}
\]

Given only two \( f \)-calculations, the Fibonacci search splits an interval almost in half. Thus

\[
F_2(\alpha) = 2\alpha \quad \text{(5)}
\]

Equations (3), (4) and (5) show that in fact

\[
F_n(\alpha) = F_n\alpha \quad \text{(6)}
\]

Where \( F_n \) is the \( n \)th Fibonacci number.

To reduce interval \([a, b]\) to size \( \alpha \), we must be able to reduce an interval of at least size \((b - a)\), which is to say we must have \((b - a) \leq F_n(\alpha)\). Given that \( \alpha \) and \( F_n(\alpha) \) represent closed intervals, to satisfy tolerance \( \epsilon \) we must pick some \( \alpha < 2\epsilon \). Thus, we arrive at the equation

\[
(b - a) \leq F_n\alpha < 2\epsilon F_n
\]

2 Problem 2

2.1 a)

The function used in this question is plotted in figure 1.
2.2 b)
After five iterations, the upper bound found for $x^*$ is 5.65, which is completely out of the domain of the function. Therefore, the domain has not been reduced in the following applications of the Fibonacci and Golden Section searches.

Matlab code and output showing how this upper bound was calculated can be found in appendix A.

2.3 c)
2.3.1 Fibonacci search
The estimated minimum found on interval $[1, 5]$ with tolerance 0.001 was $x^* = 2.798214$ with the $f(x^*) = -0.336508$. It was found after 17 $f$-calculations. The code and step by step output for the algorithm can be found in appendix B.

2.3.2 Golden Section search
The estimated minimum found on interval $[1, 5]$ with tolerance 0.001 was $x^* = 2.798040$ with $f(x^*) = -0.336508$. It was found after 18 $f$-calculations. The code and step by step output for the algorithm can be found in appendix C.
2.3.3 Newton-Raphson search

The estimated minimum found by the Newton-Raphson with tolerance 0.001 on the value of \( \frac{df}{dx} (x^*) \) was \( x^* = 2.7977 \) with \( f(x^*) = -0.3365 \). This was found after 5 iterations. That is, 5 calculations each of \( \frac{df}{dx} \) and \( \frac{d^2f}{dx^2} \), followed by one calculation of \( f \).

The code and output for the algorithm can be found in appendix D.

2.3.4 Comparison

In order to compare execution time, the code for each algorithm was stripped down (all printing statements were removed), and the algorithm was run five or six times in batches of 1000. This ensures the true execution time of the algorithm is measured, reducing superfluous overhead, and allowing for computer cacheing of repeated code. The outputs of these time trials can be found in appendix E.

In this case, the Golden Section search was actually the fastest. It averaged about 0.365 seconds for a thousand executions. This was followed closely by the Newton-Raphson search, which averaged about 0.436 seconds for the same. Last came the Fibonacci search, at over one second for a thousand executions.

It is important to note that because the function \( f \) is relatively quick to calculate, these times represent the overhead of each algorithm more than its efficiency for dealing with slow-to-calculate functions. If the function and its derivatives had taken, e.g., an hour to calculate, then the times would have been proportional to the number of function calculations - Golden Search would be the slowest at 18 hours, followed by Fibonacci at 17 hours, well behind Newton-Raphson at 11 hours.

In terms of accuracy, both the Fibonacci and Golden searches have produced the same minimum value for \( f \), and very close values for \( x^* \). The Newton-Raphson search has produced very similar values, but the result for \( f \) it has produced is not quite as good. It is worthwhile to note that the tolerance value given to the Newton-Raphson search is not equivalent to that given the others, as it is a tolerance on the value of \( \frac{df}{dx} (x^*) \), and not on \( x^* \). Therefore, given that it produced a comparable accuracy with far less function calculations, this is still a commendable result!

The Fibonacci search was by far the most complicated in terms of implementation. This stemmed from the fact that not only did the number of iterations have to be pre-calculated, but also the algorithm required special cases for \( n \in 1, 2 \). The next most complicated was the Golden search, which had a similar main body to Fibonacci. Lastly, there was the Newton-Raphson search, which was by far the simplest. Here it has been made even simpler than the algorithm in the notes, with no loss of function.
3 Problem 3

Both the Newton-Raphson and Golden Section searches assume the function to be minimised is unimodal, and this function is not, as it has local minima at both $x = 0$ and $x = 5$. Furthermore, Newton-Raphson requires a continuous derivative, but this function’s derivative has a discontinuity at $x = 2$. Therefore, these methods are not strictly applicable to minimising this function.

3.1 Application of the Golden Section Search

The first calculations for the Golden Section search are at approximately $x = 0$ and $x = 2$. Since $x = 0$ is the true minimum, this point will have, and continue to have, the smaller $f$-value in the Golden Section search comparison. Thus the Golden Section search will focus in on it. In fact, after one iteration, the search is entirely in the region $[-3, 2]$, for which the function is unimodal, so the search will find the minimum appropriately.

3.2 Application of the Newton-Raphson Search

The Newton-Raphson search actually finds the minimum of a parabola in one iteration (it was designed that way). On the other hand, if the second derivative of the function at a point is zero, the search must halt and give up without having found a minimum. Thus, it will find the minimum of this function in one iteration if it is started in the interval $[-3, 2]$, but fail on the first iteration if started in the interval $[2, 5]$.

4 Problem 4

4.1 a)

Two assumptions must be made for this problem to be reasonably interpreted:

1. We must assume that all chairs made will be bought.
2. We must assume that the discounts are calculated based on the number of chairs sold per day to all customers of one type, rather than being calculated per customer.

In both cases this is due to lack of information. Without assumption 1, the profit (and therefore the optimal production levels) will depend on the supply demand curve for the chairs, about which we know nothing. Without assumption 2, we have no way of knowing how much discount is applied to each chair. In particular, if an infinity of customers arrives and buys one chair each with the discount $15 \times (x_2 = 1)$, the store makes a profit on each chair, and thus achieves an infinity of profits if it manufactures an infinity of chairs.

With these assumptions in place, the profit calculation for the factory is as follows:
\[ f(x_1, x_2) = -\text{overhead} + \]
\[ (\text{bulk: price} - \text{discount} - \text{chair cost})x_1 + \]
\[ (\text{residential: price} - \text{discount} - \text{chair cost})x_2 \]
\[ = -100 + (70 - 4x_1 - 15)x_1 + (150 - 15x_2 - 15)x_2 \]
\[ = -4x_1^2 - 15x_2^2 + 55x_1 + 135x_2 - 100 \]  
\[ (8) \]

4.2 b)

\[ \nabla f = \begin{bmatrix} \frac{df}{dx_1} \\ \frac{df}{dx_2} \end{bmatrix} = \begin{bmatrix} -8x_1 + 55 \\ -30x_2 + 135 \end{bmatrix} \]  
\[ (9) \]

Thus, the unique solution to \( \nabla f = 0 \) is \((x_1, x_2) = (55/8, 9/2)\).

4.3 c)

Now \(-f\) is a quadratic function having the same critical points as \(f\), and having co-efficient matrix \(B\) where

\[ B = \nabla^2 (-f) \]
\[ = \begin{bmatrix} 8 & 0 \\ 0 & 30 \end{bmatrix} \]  
\[ (10) \]

Which is clearly positive definite having positive eigen values. Therefore \(-f\) is convex. Being quadratic it is \(C^1\). Thus we can conclude that \(-f\) has a global minimum at \((55/8, 9/2)\). This means that \(f\) has a global maximum at \((55/8, 9/2)\). That maximum is \(f(55/8, 9/2) = 392.8125\).

Since a fractional number of chairs cannot be created, we must search for a constrained maximum out of the nearest four integer positions, \((6, 4), (6, 5), (7, 4)\) and \((7, 5)\) (moving further in any direction would result in a value that is definitely smaller than one of these).

\[ f(6, 4) = 380 \]
\[ f(6, 5) = 386 \]
\[ f(7, 4) = 389 \]
\[ f(7, 5) = 389 \]

So the optimum number of chairs to manufacture is 12, seven for sale to retail outlets, and five for sale to residential customers.
A Bounding

A.1 M-file

%BOUND Finds a bounding interval for the minimum of a unimodal function on
% the semi closed interval [start, infinity)
%
% Inputs:
% f = the function to find bounds to the minimum of
% T = the step size to start with when searching
% start = the position to start at
%
% Outputs:
% rbound = the right bound of the interval
% kval = the number of iterations performed to find the interval.

function [rbound, kval] = bound(f, T, start)

% Check the number of inputs is correct.
if (nargin ~= 3)
    error('Usage: bound(function, initial step size, initial left bound')
end

k = 1;
p = start;
q = start + T;

sprintf('f(%f) = %f', p, feval(f, p))

while (feval(f, p) > feval(f, q))
    sprintf('f(%f) = %f', q, feval(f, q))
    k = k + 1;
p = q;
    q = p + T * (2 ^ (k - 1));
end

sprintf('f(%f) = %f', q, feval(f, q))

rbound = q;
kval = k;
kval

A.2 Output

>> bound('Q2', 0.15, 1)
ans =
f(1.000000) = 0.540302

ans =
f(1.150000) = 0.355206

ans =
f(1.450000) = 0.083105

ans =
f(2.050000) = -0.224914

ans =
f(3.250000) = -0.305886

ans =
f(5.650000) = 0.142681

kval =

5

ans =
B Fibonacci search

B.1 M-file

% Matlab function m-file: fibonacciSearch.m
%
% Performs Fibonacci line search for minimizing a
% unimodal function of one variable on the interval [a,b] to within
% a given tolerance
%
% Modify as you wish!
%
% Input:
%
% functionToMinimise This should be a string which is the same as the
% name of the function m-file corresponding
% to the function that we wish to minimize.
% In particular, functionToMinimise should take a scalar input
% and return a scalar output.
%
% a, b Lower and upper limits of the initial search interval.
% Note: functionToMinimise should be unimodal on the interval [a,b]
%
% tolerance Any positive number (typically 'small', eg: 0.01).
% The final interval produced by the Fibonacci search algorithm
% will have length 2*tolerance or less.
%
% Output:
%
% xminEstimate the midpoint of the final interval containing
% xmin produced by the Fibonacci search algorithm
%
% fminEstimate the value of functionToMinimise at xminEstimate
%
% Input syntax example:
%
% fibonacciSearch('f', 0, 1, 0.01)
% or alternatively
% fibonacciSearch(@f,0,1,0.01)
%
% where a function m-file called
% f.m
%
% has been written (by you) and saved to the local working directory.

function [xminEstimate, fminEstimate] = fibonacciSearch(functionToMinimise, a, b, tolerance)

% Check that the correct number of input arguments have been passed to the
% function (this is not essential, but it helps to detect user error)
if (nargin ~= 4) % in Matlab the operator ~= means "not equal to"
    error('four input arguments are required')
end

% Check that input parameters have appropriate values
if(b <= a)
    error('b must be strictly greater than a')
end
if(tolerance <= 0)
    error('tolerance must be strictly positive')
end

% Step 1: Find smallest value of n such that (b-a)/fibonacciNumber(n) <= 2*tolerance,
% where fibonacciNumber(n) is the n`th number of the famous Fibonacci sequence
n = 0;
while ( (b-a)/fibonacciNumber(n) >= 2*tolerance )
    n = n+1;
end

% display the value of n which will be used for the search
sprintf('using the first %d numbers of the fibonacci sequence',n)

% Step 2: initialize coordinates p and q, then
% evaluate the function to minimise at p and q
\[ p = b - \frac{\text{fibonacciNumber}(n-1)}{\text{fibonacciNumber}(n)} \cdot (b-a); \]
\[ q = a + \frac{\text{fibonacciNumber}(n-1)}{\text{fibonacciNumber}(n)} \cdot (b-a); \]

\[ \text{fp} = \text{feval(functionToMinimise, } p); \]
\[ \text{fq} = \text{feval(functionToMinimise, } q); \]
\[ \text{f_calculations} = 2; \]

% Step 3: perform repeated reductions of search interval using k as an
% iteration index

\[ [a,p,q,b] \]
\[ [f_p, f_q] \]
for \( k = n-1:-1:3 \)

\[ \text{f_calculations} = \text{f_calculations} + 1; \]

\[ \text{if } (\text{fp} \leq \text{fq}) \]

\[ b = q; \]
\[ q = p; \]
\[ \text{fq} = \text{fp}; \]

% given that we set \( q = p \), we also need to set \( f_q = f_p \)

% this is one of the reasons why the Fibonacci search is "efficient":
% it re-uses the result of an \( f \)-calculation from the
% previous iteration.

\[ p = b - \frac{\text{fibonacciNumber}(k-1)}{\text{fibonacciNumber}(k)} \cdot (b-a); \]
\[ \text{fp} = \text{feval(functionToMinimise, } p); \]

\[ \text{else} \]

\[ a = p; \]
\[ p = q; \]
\[ \text{fp} = \text{fq}; \]

% given that we set \( p = q \), we also need to set \( f_p = f_q \)

\[ q = a + \frac{\text{fibonacciNumber}(k-1)}{\text{fibonacciNumber}(k)} \cdot (b-a); \]
\[ \text{fq} = \text{feval(functionToMinimise, } q); \]

\[ \text{end} \]
\[ [a,p,q,b] \]
\[ [f_p, f_q] \]
end

% At this stage, the current search interval has \( p \) and \( q \) dividing
% the interval into equal thirds.
Step 4: in this step, the Fibonacci search algorithm reduces the search interval so that the inherited coordinate p or q is the midpoint, and the new coordinate lies a small distance to one side of the midpoint

\[ f_{\text{calculations}} = f_{\text{calculations}} + 1; \]

\[
\text{if}\ (fp \leq fq) \\
\quad b = q; \\
\quad q = p; \\
\quad fq = fp; \\
\quad p = b - 2\cdot\text{tolerance}; \\
\quad fp = \text{feval(functionToMinimise, p)}; \\
\text{else} \quad % \text{it must be true that } fp > fq \\
\quad a = p; \\
\quad p = q; \\
\quad fp = fq; \\
\quad q = a + 2\cdot\text{tolerance}; \\
\quad fq = \text{feval(functionToMinimise, q)}; \\
\text{end}
\]

Step 5: In this final step, the Fibonacci search algorithm reduces the interval to length to either
\% a) exactly 2*\text{tolerance} \\
\% b) less than 2*\text{tolerance} - specifically, (original interval length)/F_n

\[
\text{if}\ (fp \leq fq) \\
\quad b = q; \\
\text{else} \\
\quad a = p; \\
\text{end}
\]

% assign the output values of this function (as declared on line 30)
\[
\text{xminEstimate} = (a+b)/2; \quad % \text{the midpoint of the final interval} \\
\text{fminEstimate} = \text{feval(functionToMinimise,xminEstimate)};
\]

% Print out the endpoints of the final interval and the output values of
% the function (can be commented out)
sprintf(' The minimum lies in the interval [%f, %f] 
 xMinEstimate = midpoint = %f 
 fMinEstimate %f',a,b,xminEstimate,fminEstimate)
f calculations

B.2 Output

>> fibonacciSearch('Q2', 1, 5, 0.001)
ans =

using the first 17 numbers of the fibonacci sequence

ans =
1.0000  2.5279  3.4721  5.0000

ans =
-0.3234  -0.2724

ans =
1.0000  1.9443  2.5279  3.4721

ans =
-0.1877  -0.3234

ans =
1.9443  2.5279  2.8885  3.4721

ans =
-0.3234  -0.3352
ans =
    2.5279    2.8885    3.1115    3.4721

ans =
   -0.3352   -0.3212

ans =
    2.5279    2.7508    2.8885    3.1115

ans =
   -0.3361   -0.3352

ans =
    2.5279    2.6656    2.7508    2.8885

ans =
   -0.3334   -0.3361

ans =
    2.6656    2.7508    2.8034    2.8885

ans =
   -0.3361   -0.3365

ans =
    2.7508    2.8034    2.8359    2.8885
ans =
    -0.3365   -0.3363

ans =
    2.7508     2.7833     2.8034     2.8359

ans =
    -0.3365   -0.3365

ans =
    2.7833     2.8034     2.8158     2.8359

ans =
    -0.3365   -0.3365

ans =
    2.7833     2.7957     2.8034     2.8158

ans =
    -0.3365   -0.3365

ans =
    2.7833     2.7910     2.7957     2.8034

ans =
    -0.3365   -0.3365

ans =

16
The minimum lies in the interval $[2.797214, 2.799214]\) 
xminEstimate = midpoint = 2.798214 
fminEstimate = -0.336508 

f_calculations = 
17 

ans = 
2.7982
C  Golden Section search

C.1  M-file

% Matlab function m-file: goldenSearch.m
%
% Performs Golden Section line search for minimising a
% unimodal function of one variable on the interval [a,b] to within
% a given tolerance
%
% Created 30/03/07, for use in 620-361: Operations Research and Algorithms
% Modified from fibonacciSearch.m supplied as subject material
% Modify as you wish!
%
% Input:
%
% functionToMinimise This should be a string which is the same as the
% name of the function m-file corresponding
% to the function that we wish to minimise.
% In particular, functionToMinimise should take a scalar input
% and return a scalar output.
%
% a, b Lower and upper limits of the initial search interval.
% Note:functionToMinimise should be unimodal on the interval [a,b]
%
% tolerance Any positive number (typically "small", eg: 0.01).
% The final interval produced by the Golden Section search algorithm
% will have length 2*tolerance or less.
%
% Output:
%
% xminEstimate the midpoint of the final interval containing
% xmin produced by the Fibonacci search algorithm
%
% fminEstimate the value of functionToMinimise at xminEstimate
%
% Input syntax example:
%
% goldenSearch('f',0,1,0.01)
%
% or alternatively
%
% goldenSearch(@f,0,1,0.01)
% where a function m-file called
% f.m
% has been written (by you) and saved to the local working directory.

function [xminEstimate, fminEstimate] = goldenSearch(functionToMinimise, a, b, tolerance)

% Check that the correct number of input arguments have been passed to the
% function (this is not essential, but it helps to detect user error)
if (nargin ~= 4) % in Matlab the operator ~= means "not equal to"
    error('four input arguments are required')
end

% Check that input parameters have appropriate values
if(b <= a)
    error('b must be strictly greater than a')
end

if(tolerance <= 0)
    error('tolerance must be strictly positive')
end

% Store the golden section
g = (sqrt(5) - 1)/2;

% Store the desired interval size
targetIntervalSize = 2 * tolerance;

% Step 1: initialize coordinates p and q, then
% evaluate the function to minimise at p and q
p = b - g * (b-a);
q = a + g * (b-a);

k = 1;
fp = feval(functionToMinimise, p);
fq = feval(functionToMinimise, q);
f_calculations = 2;

% Step 3: perform repeated reductions of search interval until
% the desired interval size is reached.

[a,p,q,b]
[fp, fq]

while((b - a) >= targetIntervalSize)
    k = k + 1;
    f_calculations = f_calculations + 1;

    if (fp <= fq)
        b = q;
        q = p;
        fq = fp; % given that we set q = p, we also need to set fq = fp
        % this is one of the reasons why the Golden Section search is "efficient"
        % it re-uses the result of an f-calculation from the
        % previous iteration.
        p = b - g * (b-a);
        fp = feval(functionToMinimise, p);
    else
        a = p;
        p = q;
        fp = fq; % given that we set p = q, we also need to set fp = fq
        q = a + g * (b-a);
        fq = feval(functionToMinimise, q);
    end
end

% assign the output values of this function (as declared on line 27)

xminEstimate = (a+b)/2; % the midpoint of the final interval
fminEstimate = feval(functionToMinimise,xminEstimate);

% Print out the endpoints of the final interval and the output values of
% the function (can be commented out)
sprintf(' The minimum lies in the interval [%f, %f] \n xminEstimate = midpoint = %f \n fminEstimate %f',a,b,xminEstimate,fminEstimate)

f_calculations

C.2 Output

>> goldenSearch('Q2', 1, 4, 0.001)
ans =
    1.0000    2.1459    2.8541    4.0000

ans =
-0.2535   -0.3360

ans =
    2.1459    2.8541    3.2918    4.0000

ans =
-0.3360   -0.3004

ans =
    2.1459    2.5836    2.8541    3.2918

ans =
-0.3283   -0.3360

ans =
    2.5836    2.8541    3.0213    3.2918

ans =
\[-0.3360 \quad -0.3286\]

\[\text{ans} = \]
\[
\begin{array}{cccc}
2.5836 & 2.7508 & 2.8541 & 3.0213 \\
\end{array}
\]

\[\text{ans} = \]
\[
\begin{array}{cc}
-0.3361 & -0.3360 \\
\end{array}
\]

\[\text{ans} = \]
\[
\begin{array}{cccc}
2.5836 & 2.6869 & 2.7508 & 2.8541 \\
\end{array}
\]

\[\text{ans} = \]
\[
\begin{array}{cc}
-0.3344 & -0.3361 \\
\end{array}
\]

\[\text{ans} = \]
\[
\begin{array}{cccc}
2.6869 & 2.7508 & 2.7902 & 2.8541 \\
\end{array}
\]

\[\text{ans} = \]
\[
\begin{array}{cc}
-0.3361 & -0.3365 \\
\end{array}
\]

\[\text{ans} = \]
\[
\begin{array}{cccc}
2.7508 & 2.7902 & 2.8146 & 2.8541 \\
\end{array}
\]

\[\text{ans} = \]
\[
\begin{array}{cc}
-0.3365 & -0.3365 \\
\end{array}
\]

\[\text{ans} = \]
\[
\begin{array}{cccc}
\text{ans} = \\
\end{array}
\]
2.7508  2.7752  2.7902  2.8146

ans =
-0.3364  -0.3365

ans =
2.7752  2.7902  2.7996  2.8146

ans =
-0.3365  -0.3365

ans =
2.7902  2.7996  2.8053  2.8146

ans =
-0.3365  -0.3365

ans =
2.7902  2.7960  2.7996  2.8053

ans =
-0.3365  -0.3365

ans =
2.7960  2.7996  2.8018  2.8053

ans =
-0.3365  -0.3365

23
ans =
2.7960  2.7982  2.7996  2.8018

ans =
-0.3365  -0.3365

ans =
2.7960  2.7974  2.7982  2.7996

ans =
-0.3365  -0.3365

ans =
2.7974  2.7982  2.7987  2.7996

ans =
-0.3365  -0.3365

ans =
2.7974  2.7979  2.7982  2.7987

ans =
-0.3365  -0.3365

ans =
The minimum lies in the interval [2.797361, 2.798720]
xminEstimate = midpoint = 2.798040
fminEstimate = -0.336508

definitions =

ans =

2.7980

D  Newton-Raphson search

D.1  M-file

% Matlab function m-file: newtonSearch.m
%
% Performs Newton-Raphson line search for finding a zero of an
% increasing function of one variable on the real numbers
%
% Created 30/03/07, for use in 620-361: Operations Research and Algorithms
% Modified from goldenSearch.m
% Modify as you wish!
%
% Input:
%
% functionToZero This should be a string which is the same as the
% name of the function m-file corresponding
% to the function that we wish to zero.
% In particular, functionToZero should take a scalar input
% and return a scalar output. Note that this should
% be strictly increasing.
%
% derivative The derivative of functionToZero. Note that this
% should be strictly positive.
%
% startingPoint The first estimate for \( x \)
%
% tolerance Any positive number (typically "small", eg: 0.01).
% 'functionToZero' will map the final \( x \) value produced
% to a value smaller than this limit.
% Output:
% xEstimate a point on the real axis for which
% 'functionToZero' is less than the tolerance
%
% Input syntax example:
% newtonSearch('f','dfdx',0.01)
% or alternatively
% newtonSearch(@f,@dfdx,0.01)
% where function m-files called
% f.m
dfdx.m
% has been written (by you) and saved to the local working directory.

function xEstimate = newtonSearch(functionToZero, derivative, startingPoint, tolerance)

% Check that the correct number of input arguments have been passed to the
% function (this is not essential, but it helps to detect user error)
if (nargin ~= 4) % in Matlab the operator ~= means "not equal to"
    error('four input arguments are required')
end

% Check that input parameters have appropriate values
if(tolerance <= 0)
    error('tolerance must be strictly positive')
end

k = 1;
a = startingPoint;

while(abs(feval(functionToZero, a)) >= tolerance)
    if(feval(derivative, a) < tolerance)
        sprintf('Derivative too small, a zero estimate could not be found')
    end
end
break;
end

a = a - feval(functionToZero, a) / feval(derivative, a);
k = k + 1;
end

xEstimate = a;

sprintf(' g(%f) = %f found after %d iterations', a, feval(functionToZero, a), k)

D.2 Output

>> newtonSearch('dQ2dx', 'ddQ2dxdx', 1, 0.001)
ans =
g(2.797655) = -0.000246 found after 5 iterations

ans =

2.7977

>> Q2(ans)
ans =

-0.3365

E Time Trials

>> tic, for i = 1:1000 ; silentNewtonSearch('dQ2dx', 'ddQ2dxdx', 1, 0.001); end ; toc
Elapsed time is 0.438610 seconds.

>> tic, for i = 1:1000 ; silentNewtonSearch('dQ2dx', 'ddQ2dxdx', 1, 0.001); end ; toc
Elapsed time is 0.436232 seconds.

>> tic, for i = 1:1000 ; silentNewtonSearch('dQ2dx', 'ddQ2dxdx', 1, 0.001); end ; toc
Elapsed time is 0.433551 seconds.

>> tic, for i = 1:1000 ; silentNewtonSearch('dQ2dx', 'ddQ2dxdx', 1, 0.001); end ; toc
Elapsed time is 0.434396 seconds.
Elapsed time is 0.437721 seconds.

```matlab
>> tic, for i = 1:1000 ; silentGoldenSearch('Q2', 1, 4, 0.001); end ; toc
Elapsed time is 0.371148 seconds.
```

Elapsed time is 0.365470 seconds.

```matlab
>> tic, for i = 1:1000 ; silentGoldenSearch('Q2', 1, 4, 0.001); end ; toc
Elapsed time is 0.366051 seconds.
```

Elapsed time is 0.364141 seconds.

```matlab
>> tic, for i = 1:1000 ; silentFibonacciSearch('Q2', 1, 5, 0.001); end ; toc
Elapsed time is 1.199157 seconds.
```

Elapsed time is 1.196196 seconds.

```matlab
>> tic, for i = 1:1000 ; silentFibonacciSearch('Q2', 1, 5, 0.001); end ; toc
Elapsed time is 1.199076 seconds.
```

Elapsed time is 1.193825 seconds.

```matlab
>> tic, for i = 1:1000 ; silentFibonacciSearch('Q2', 1, 5, 0.001); end ; toc
Elapsed time is 1.195606 seconds.
```