



University College Dublin
An Coláiste Ollscoile, Baile Átha Cliath

WINTER EXAMINATIONS 2005

SCMXF0025 – MSc in Computational Science – Year 1

SCHDF0125 – H.Dip. in Computational Science (Secondary Curriculum)

MAPH P401 — NUMERICAL ALGORITHMS

— SOLUTIONS —

Prof C.J. Budd

Prof A. Ottewill

Dr Derek O'Connor*

Time Allowed: 3 hours

Instructions for Candidates

Answer any **five** (and only five) questions.

Instructions for Invigilators

Candidates may use calculators and tables.

Question 1.

(a). Describe the *Gaussian Elimination* algorithm with partial pivoting and explain what happens when a non-zero pivot cannot be found at stage k .

Solution 1. (a).

See Notes Chapter 5.

(b). Find the factors L, U , and P for the matrix A below

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}$$

Generalize your result for any $n \times n$ matrix of this form. Does this result indicate a potential difficulty in solving $Ax = b$?

Solution 2. (b)

The Gaussian elimination operations for each $k = 1, 2, \dots, n - 1$ are : add row k to rows $k + 1, k + 2, \dots, n$. This means that all the multipliers m_{ki} are 1 and so all the elements beneath the diagonal of L are -1 . No pivoting is needed. Hence $P = I$.

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 \\ -1 & -1 & -1 & 1 & 0 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 4 \\ 0 & 0 & 0 & 1 & 8 \\ 0 & 0 & 0 & 0 & 16 \end{bmatrix}, \quad P = I.$$

Generalizing to an $n \times n$ matrix, L and P are the same and the last column of U has the general element $u_{in} = 2^{i-1}$, $i = 1, 2, \dots, n$.

(c). Gaussian Elimination with *complete pivoting* interchanges both rows and columns of A at each pivot step. This gives the factorization $PAQ = LU$, where P and Q are permutation matrices, L is unit lower triangular, and U is upper triangular. Write an algorithm to solve $Ax = b$ using this factorization.

Solution 1 (c).

Given the factorization $A \rightarrow L, U, P, Q$ we get $PAQQ^{-1}x = Pb$. Letting $x' = Q^{-1}x$ and $b' = Pb$ we have $PAQx' = b'$, or $LUx' = b'$. The steps in solving $Ax = b$ are now

1. Calculate $b' = Pb$
2. Solve $Ly = b'$, using Forward Substitution, giving $y = L^{-1}b' = L^{-1}Pb$.
3. Solve $Ux' = y$, using Back Substitution, giving $x' = U^{-1}y = U^{-1}L^{-1}Pb$.
4. Calculate $x = Qx' = QU^{-1}L^{-1}Pb$. Not needed with partial pivoting, where $Q = I$.

Question 2.

(a). Using Newton's method for zero-finding, derive the iteration formula $x_{k+1} := T(x_k)$ for calculating $1/a$. Over what range of starting values x_0 does the reciprocal sequence converge to $1/a$?

Solution 2 (a).

We may define the reciprocal of some number $a > 0$ as a zero of the function $f(x) = 1/x - a$. Using this function in Newton's method

$$\boxed{x_{k+1} := x_k - \frac{f(x_k)}{f'(x_k)}} \quad (1)$$

gives

$$x_{k+1} := x_k - \frac{(1/x_k - a)}{(-1/x_k^2)} = x_k + (x_k - ax_k^2)$$

Thus we get the Newton iteration function for reciprocals :

$$\boxed{x_{k+1} := x_k(2 - ax_k) \triangleq T(x_k)}. \quad (2)$$

Notice that this iteration function uses multiplication and subtraction only.

The first way to determine the domain of convergence uses a corollary to Banach's fixed point theorem : the mapping $T : \mathbb{R}^1 \rightarrow \mathbb{R}^1$, is a contraction if $|T'(x)| < 1$ on some ball B . This means

$$\begin{aligned} |T'(x)| &= |2 - 2ax| < 1 \\ \Rightarrow |1 - ax| &< \frac{1}{2} \\ \Rightarrow -\frac{1}{2} &< 1 - ax < \frac{1}{2}. \end{aligned}$$

Thus the method will converge for any x_0 in the range $1/2a < x_0 < 3/2a$. This is a sufficient but not necessary condition, as can be seen in Figure 1 : the lower bound $1/2a < x$ can be loosened to $0 < x$. The upper bound can be loosened to $x < 2/a$, where the tangent to $f(x) = 1/x - a$ at $x = 2/a$ passes through the origin.

(b). Given $a = 5$ and starting at $x_0 = 1/10$ show how the iteration formula above converges to $0.2 = 1/5$ by calculating the first four steps, x_1, x_2, x_3, x_4 .

Illustrate your calculation steps by drawing the function $f(x) = 1/x - a$ and the five approximations steps x_0, x_1, \dots, x_4 .

Solution 2 (b).

| k | x_k | $x_{k+1} := x_k \star (2 - 5 \star x_k)$ |
|-----|-------------------|------------------------------------------|
| 0 | 0.1 | 0.15 |
| 1 | 0.15 | 0.1875 |
| 2 | 0.1875 | 0.19921875 |
| 3 | 0.19921875 | 0.199996948242188 |
| 4 | 0.199996948242188 | 0.19999999953434 |
| 5 | 0.19999999953434 | 0.2 |

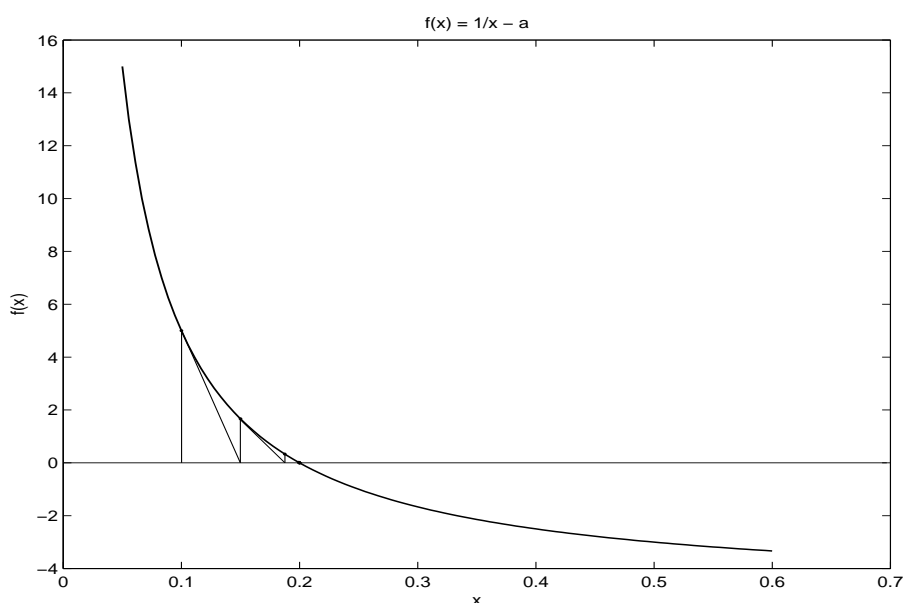


Figure 1: Reciprocal

(c). Let $e_n = |x_n - 1/a|$ be the error at the n th iteration formula derived in (a) above. Derive an expression for e_n as a function of e_{n-1} , the error at the previous step. Hence show that this iteration formula has *second order convergence*. Starting with an error $e_0 = |x_0 - 1/a| = 2^{-1}$, how many iterations are required to get x_n to full IEEE double precision.

Solution 2 (c).

Let $e_k = |x_k - 1/a|/1/a = 1 - ax_k$ be the relative error at the k th iteration of $x_{k+1} := x_k(2 - ax_k)$. Then we have

$$\begin{aligned}
 e_k = 1 - ax_k &= 1 - ax_{k-1}(2 - ax_{k-1}) \\
 &= 1 - 2x_{k-1} + ax_{k-1}^2 \\
 &= (1 - ax_{k-1})^2 = e_{k-1}^2
 \end{aligned} \tag{3}$$

This result, $e_k = e_{k-1}^2$, shows immediately that the sequence has 2nd-order convergence, which we normally expect of Newton's method. Using successive substitution we get

$$\begin{aligned} e_k &= e_{k-1}^2 = (e_{k-2}^2)^2 = e_{k-2}^{2^2} = (e_{k-3}^{2^2})^2 \\ &= e_{k-3}^{2^3} = \dots = e_0^{2^k}, \text{ and we get} \end{aligned}$$

$$\boxed{e_k = e_0^{2^k} = (1 - ax_0)^{2^k}} \quad (4)$$

A necessary and sufficient condition for the sequence $\{e_k\}$ to converge is $|ax_0 - 1| < 1$. This condition gives $|ax_0 - 1| < 1 \Rightarrow -1 < ax_0 - 1 < 1 \Rightarrow 0 < ax_0 < 2$, and we get the domain of convergence

$$\boxed{0 < x_0 < \frac{2}{a}} \quad (5)$$

We note that this condition is less stringent than (2). Indeed, this interval is twice the width of the interval in (2).

The number of iterations to full IEEE double precision is as follows :

Using equation (4) with $e_0 = 2^{-1}$ we get $e_k = e_0^{2^k} = (2^{-1})^{2^k} = 2^{-2^k}$. We have full precision when $e_k = 2^{-53}$. Solving $e_k = 2^{-53} = 2^{-2^k}$ for k gives $k = \lceil \log_2 53 \rceil = 6$ iterations

Question 3.

(a). Describe what is meant by a floating point number system $\mathbb{F}(b, p, e_{min}, e_{max})$. Show that every number $x \in \mathbb{F}$ is rational, i.e., $x = \frac{u}{v}$, and show how to determine the integers u and v for any x in floating point format.

Solution 3 (a).

A floating point number x is represented as

$$\begin{aligned} x = \pm .s \times b^e &= \pm .(s_1 s_2 \dots s_p) \times b^e, \quad s_i \in \{0, 1, \dots, b-1\} \\ &= \pm \left(\frac{s_1}{b^1} + \frac{s_2}{b^2} \dots \frac{s_{p-1}}{b^{p-1}} + \frac{s_p}{b^p} \right) \times b^e \\ &= \pm (s_1 b^{p-1} + s_2 b^{p-2} \dots s_{p-1} b^1 + s_p) \times \frac{b^e}{b^p} \\ &= m \times \frac{b^e}{b^p} = m \frac{n}{v} = \frac{u}{v}. \end{aligned}$$

where m, n, u , and v are integers. Thus, all floating point numbers are rational.

Given that we know b and p and can extract s and e from x , then $u = s \times b^p \times b^e = s \times b^{e+p}$ and $v = b^p$. MATLAB uses IEEE double precision which means $b = 2$ and $p = 53$. The MATLAB function $\log_2(x)$, used as $[s,e] = \log_2(x)$, gives s and e .

(b). Characterize the floating point numbers $x \in \mathbb{F}$ that cause x^2 to either underflow or overflow. What percentage of the numbers $x \in \mathbb{F}$ cause this problem?

Question 4.

(a). Describe how you would use *Monte Carlo sampling* to find the area of an n -polygon in two dimensions defined by the points $P = (p_1, p_2, \dots, p_n)$. You have the *computational geometry predicate* function $InPoly(P)$. Write a `MATLAB` function to do this calculation, being careful to describe how the polygon is represented in `MATLAB`.

Solution 4 (a).

In `MATLAB` we will define a point as the vector $p = [x \ y]$, e.g. $p = [5.3 \ 3.4]$ and a polygon as an $n \times 2$ matrix $poly = [x_1 \ y_1; x_2 \ y_2; \dots; x_n \ y_n]$, e.g. $poly = [2 \ 3; 3 \ 6; 4 \ 6; 3 \ 8; 3 \ 7; 2 \ 5]$. As such, this is just a set of points. If we connect each adjacent pair of points by a line segment then we get an open polygon. To close the polygon we must add an extra point $p_{n+1} = p_1$. Thus we get the closed polygon $poly = [2 \ 3; 3 \ 6; 4 \ 6; 3 \ 8; 3 \ 7; 2 \ 5; 2 \ 3]$. Here is the `MATLAB` function for the area of a polygon. This was part of Assignment 2.

```
function area = HMPoly(polyg,nsamp)
%-----
% Estimate the area of a polygon Polyg = (p1,p2,...,pn)
% using the Hit-or-Miss Monte Carlo method.
% This generates sample points in the bounding box of Polyg.
% Derek O'Connor, UCD, Oct 2005
%
[xmin,ymin,xmax,ymax] = BoundingBox(polyg);
hits = 0;
for k = 1 : nsamp
    x = xmin + (xmax-xmin)*rand;
    y = ymin + (ymax-ymin)*rand;
    p = [x y];
    if InPoly(polyg,p)
        hits = hits + 1;
    end;
end;
area = (xmax-xmin)*(ymax-ymin)*(hits/nsamp);
%----- end HMPoly(polyg,nsamp) -----
```

(b). Modify the `MATLAB` function above to calculate the area enclosed by the intersection of two polygons P_1 and P_2 .

Solution 4 (b).

This is done by two simple modifications of the `MATLAB` function above. First we calculate the bounding box that contains both polygons. We then sample from this bounding box and if a point p falls within both P_1 and P_2 then it lies in the intersection of the two polygons, and counts as a hit. The modified function is shown below.

A little more thought reveals that the bounding box of either polygon must contain their intersection so that it is sufficient to calculate either box and sample from it. This will concentrate the sample points and so the estimate of the area of the intersection will be more accurate.

We could refine this by calculating both bounding boxes and then sampling from the smaller.

```

=====
function area = HMPolyInt(poly1,poly2,nsamp)
%-----
% Estimate the area of the intersection of two polygons
% using the Hit-or-Miss Monte Carlo method.
% This generates sample points in the bounding box of the two.
% Derek O'Connor, UCD, Jan 2006
%
[xmin,ymin,xmax,ymax] = BoundingBox(poly1,poly2);
hits = 0;
for k = 1 : nsamp
    x = xmin + (xmax-xmin)*rand;
    y = ymin + (ymax-ymin)*rand;
    p = [x y];
    if InPoly(poly1,p) & InPoly(poly2,p)
        hits = hits + 1;
    end;
end;
area = (xmax-xmin)*(ymax-ymin)*(hits/nsamp);
%----- end HMPolyInt(poly1,poly2,nsamp) -----

```

The ultimate refinement is this : sample from the intersection of the two bounding boxes. This is relatively easy to calculate and would save time if the intersection is empty.

(c). You have the MATLAB function `RanBit` which returns 1 with probability $\frac{1}{2}$ and 0 with probability $\frac{1}{2}$. Using this function only, write a function `Roll` that returns one of the set $\{1, 2, 3, 4, 5, 6\}$, each with probability $\frac{1}{6}$. Analyse your `Roll` function and find the average number of calls to `RanBit` it makes. Can you speed this up ?

Solution 4 (c).

Using `RanBit` three times in succession, we get one of eight possible bit strings $\{(000), (001), \dots, (110), (111)\}$ each with probability $\frac{1}{2^3} = \frac{1}{8}$. We interpret this set of bit strings as the integer set $\{0, 1, \dots, 7\}$. If we get 0 or 7 then we reject these and start over again. Thus we get one of the set $\{1, 2, 3, 4, 5, 6\}$ with equal probability = $1/8$. This is shown in Figure 4

The recursive MATLAB function is as follows :

```

=====
function face = Roll;
%-----
% Flipping a coin to generate the six faces of a die.
% RanBit returns 1 with prob. p and 0 with prob. 1-p
% Derek O'Connor, UCD, Jan 2006
test = RanBit*2^0;
test = test + RanBit*2^1;
test = test + RanBit*2^2;
if test == 0 | test == 7
    test = Roll;
end;
face = test;

```

The analysis of `Roll` is fairly simple : The first 3 statements `test = etc.` make 3 calls to `RanBit`. This produces an integer in $\{0, 1, \dots, 7\}$ each with probability $1/8$. Two out of

these 8 integers are rejected and Roll is called (recursively) again. The probability of calling Roll again is $\frac{2}{8}$. Thus the number of calls to RanBit is a random number. Let B stand for the average number of calls to RanBit. Then the average number of calls is

$$B = 3 + \frac{2}{8}B, \quad \text{or} \quad B = 4. \quad (6)$$

Although 3 calls to RanBit give us more numbers than we need, we have to make 4 calls to it on average.

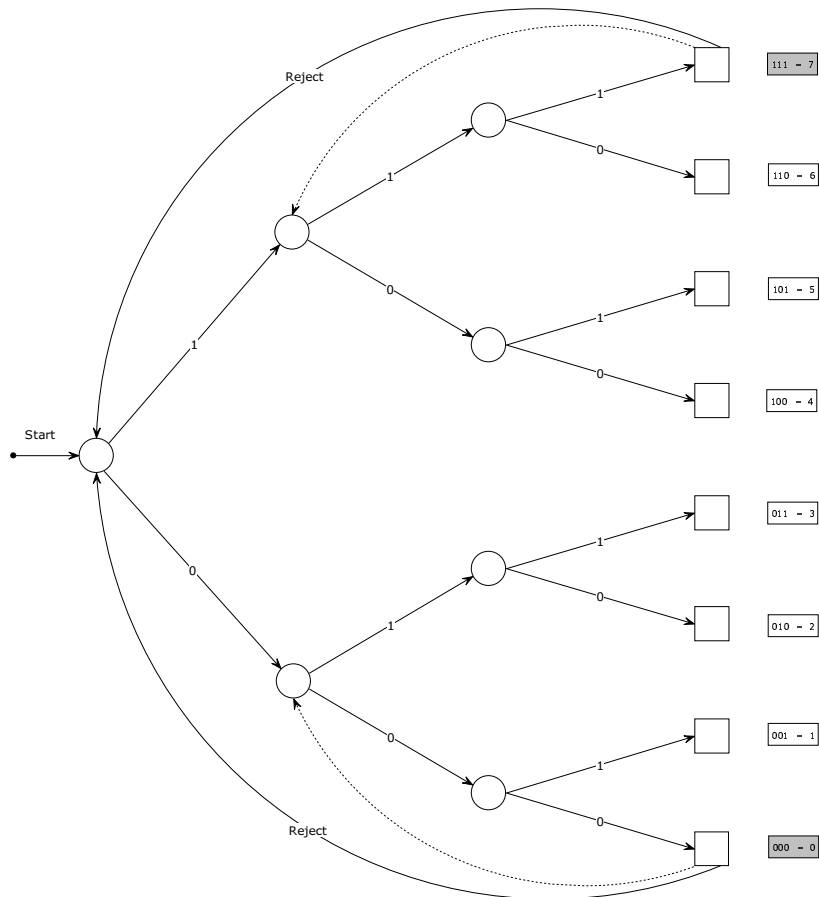


Figure 2: Rolling a Die using Random Bits

Can we speed up this function? Yes, by noticing that we do not need to repeat all 3 test = etc. statements. The bit strings (000) and (111) are equally probable and may be regarded as 0 and 1 outcomes of the first call to RanBit. This shortcut is shown as a dotted line in Figure 4. Equation (6) now becomes

$$B = 3 + \frac{2}{8}B', \quad \text{and} \quad B' = 2 + \frac{2}{8}B'. \quad (7)$$

The second equation gives $B' = \frac{8}{3}$, and this gives $B = 3\frac{2}{3}$.

[NOTE 1] Although the first statements of `Roll` produce an integer in $\{0, 1, \dots, 7\}$ each with probability $1/8$, the function produces an integer in $\{1, \dots, 6\}$ each with probability $1/6$.

[NOTE 2] It may seem that we need probabilities and averages in our analysis because we are using a random bit generator. This is not true, necessarily. Indeed, the analysis of most algorithms that contain an **if**-test requires probability statements.

```

% =====
function face = RollSC;
% -----
% Flipping a coin to generate the six faces of a die.
% RanBit returns 1 with prob. p and 0 with prob. 1-p
% Uses the shortcut described above.
% Derek O'Connor, UCD, Jan 2006
test = RanBit*2^0;
test = test + RanBit*2^1;
test = test + RanBit*2^2;
if test == 0 | test == 7
    test = RollP(test);
end;
face = test;
% =====
function face = RollP(test);
% -----
test = test + RanBit*2^1;
test = test + RanBit*2^2;
if test == 0 | test == 7
    test = RollP(test);
end;
face = test;

```

Question 5.

(a). Define what is meant by *forward* and *backward* error in numerically calculating a function $f(x)$ with an algorithm $A(x)$.

Solution 5 (a).

See Notes Chapter 2.

(b). Define the *condition* of a problem and show how it is related to the forward and backward error in (a) above. What is the condition of the problem $f(x) = \sqrt{x}$. Illustrate your answer by calculating this function for two nearby values of x .

Solution 5 (b).

See Notes Chapter 2.

(c). What is the condition [number] of the linear equations problem $Ax = b$? Show why the *residual* $r = b - A\hat{x}$ is not a good measure of the error in the solution $e = \hat{x} - x$.

Solution 5 (c).

See Notes Chapter 5.

Question 6.

(a). Describe two methods of transforming the matrix-vector equation $Ax = b$ into the fixed-point form $x = Cx + d$. Apply Banach's Fixed Point theorem to find sufficient conditions for the convergence of

$$x^{k+1} = Cx^k + d, \text{ where } x, d \in \mathbb{R}^n \text{ and } C \in \mathbb{R}^{n \times n}$$

Solution 6 (a).

See Notes Chapter 5.

(b). Describe the *Jacobi Method* and use it for 4 iterations to solve $Ax = b$, where

$$A = \begin{bmatrix} 20 & 1 & 1 \\ 1 & 10 & 1 \\ 1 & 1 & 10 \end{bmatrix}, \quad b = \begin{bmatrix} 22 \\ 12 \\ 12 \end{bmatrix}$$

Solution 6 (b).

(c). Describe the *Iterative Refinement* method and show how it is related to the Jacobi method by comparing their *residuals* at each step.

Solution 6 (c).

See Notes Chapter 5.

Question 7.

(a). All iterative methods generate a sequence of vectors $\{x_{k+1} = T(x_k)\}$ which must be stopped after a certain number of iterations. Generally we use a Cauchy-like test for convergence because the limit of the sequence is not known. One such test is

$$\|x_{k+1} - x_k\| \leq e_x + \epsilon_m \|x_k\|,$$

where e_x is set by the user and ϵ_m is *machine epsilon*. Show why this test works for all values of e_x , and why the simplified test $\|x_{k+1} - x_k\| \leq e_x$ can fail to stop the iterations.

Solution 7 (a).

See Notes Chapter 4.

(b). Setting $e_x = 0$ gives $\|x_{k+1} - x_k\| \leq \epsilon_m \|x_k\|$, which stops the iterations when full precision is reached. Explain why this test is not adequate when x_k close to or equal to 0.

Solution 7 (b).

See Notes Chapter 4.

(c). Explain why the sequence $\{H_n = \sum_{k=1}^n 1/k\}$ converges to a limit H_f in finite precision arithmetic. Write a `MATLAB` function to calculate this limit, with an appropriate test for convergence.

Solution 7 (c).

See Notes Assignment ?.