Homework 1

16.9.22

Chuan Lu, 13300180056, chuanlu13@fudan.edu.cn

Code of this assignment are in the attachment, and they can also be found in GitHub: https://github.com/Orcuslc/Learning/tree/master/Numerical%20Algorithm%20and%20Case%20Studies/Homework1/code

Problem 1.

Abstract.

About the computation of roots and its sensitivity on coefficients of a quadratic polynomial.

Introduction and Algorithm.

In middle school we learned that the roots of a quadratic polynomial $ax^2 + bx + c = 0$ can be expressed as $x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a}, x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$.

However in computation, when $4ac \ll b^2$, the so-called catastrophic cancellation would happen: if $b \geq 0$, then $\sqrt{b^2 - 4ac} \approx b$, which would cause a decreas in the number of significant digits of $-b + \sqrt{b^2 - 4ac}$. This statement also holds when $b \leq 0$.

In order to avoid this, we use a different but equivalent expression for the roots.

When $b \geq 0$, $\begin{cases} x_1 = \dfrac{-b - \sqrt{b^2 - 4ac}}{2a} \\ x_2 = \dfrac{-2c}{b + \sqrt{b^2 - 4ac}} \end{cases}$; when $b < 0$, $\begin{cases} x_1 = \dfrac{-b + \sqrt{b^2 - 4ac}}{2a} \\ x_2 = \dfrac{2c}{-b + \sqrt{b^2 - 4ac}} \end{cases}$

Verification: The correctness can be easily proved by using Vieta theorem.

Code.

The code can be found in the attachment($\mathrm{extract\_quadratic.m}$); We use the name MyFunc to refer this function in the following sections.

Results.

In the following sections, we always use x_1 to refer to the larger root, and x_2 to the smaller root.

For $(b = -56, c = 1)$, the result is in Table 1 below:

	x_1	x_2
$\mathrm{MyFunc}()$	55.9821	0.0179
$\mathrm{root}()$	55.9821	0.0179

Table 1: Result of $(b = -56, c = 1)$

For $(b = -10^8, c = 1)$, the result is in Table 2 below:

When $4ac \approx b^2$, the results are the same; but when $4ac \ll b^2$, the significant number of the smaller root using root is eliminated.

	x_1	x_2
MyFunc()	100000000	1.0000e-08
root()	$1.0000 * 10^8$	$0.0000 * 10^8$

Table 2: Result of $(b = -10^8, c = 1)$

Sensitivity.

According to Taylor series,

$$x(t+\delta) \approx x(t) + \delta \frac{dx}{dt}(t).$$

Therefore the sensitivity of x on t can be defined as $s_x(t) = dx/dt$.

In this problem, $F(x,b,c) = x^2 + bx + c$. According to implicit function theorem,

$$\frac{\partial x}{\partial b} = -\frac{F_x}{F_b} = -\frac{2x+b}{x} = -2 + \frac{b}{x},$$

$$\frac{\partial x}{\partial c} = -\frac{F_x}{F_c} = -(2x+b).$$

In consequence, when $|x| \ll |b|$, x is very sensitive to a little change in b. Similarly, when $|2x+b| \gg 1$, x is very sensitive to a little change in c.

Discussion and Conclusion. With the conclusion in Sensitivity part, we can make some explanation for the difference between results of MyFunc and root.

1. For the first case, when $b = -56, c = 1$, and the result $x_1 \approx 56, x_2 \approx 0.018$.

In this case $\frac{\partial x_1}{\partial b} \approx -3$, $\frac{\partial x_1}{\partial c} \approx -56$, and $\frac{\partial x_2}{\partial b} \approx -3113$, $\frac{\partial x_1}{\partial c} \approx 56$. Since the sensitivities are not large enough to have strong effects on a $\mathrm{format\ short}$ computation, the result of MyFunc and root seems the same.

2. For the second case, when $b = -10^8, c = 1$, and the result $x_1 \approx 10^8, x_2 \approx 10^{-8}$.

In this case $\frac{\partial x_1}{\partial b} \approx -3$, $\frac{\partial x_1}{\partial c} \approx -10^8$, and $\frac{\partial x_2}{\partial b} \approx 10^16$, $\frac{\partial x_2}{\partial c} \approx 10^8$.

Apparently, sensitivity of x_2 on both b and c are large enough to make a obvious change in significant numbers. It leads to that the result of x_2 being 0 when using root. In this scenario we may regard this numerial problem an ill-conditioned one; though the numerical algorithm by MATLAB may be a stable one, the results can be far from true ones.

Remarks. I used Python to solve the equation $x^2 - 1e^8 x + 1 = 0$:

```
import scipy.optimize as op
f = lambda x: x**2 - 1e8* x + 1
x1 = op.fsolve(f, x0 = 0)
x2 = op.fsolve(f, x0 = 1e9)
```

The result is: $x_1 = 1e^{-8}, x_2 = 1e^8$. So I am wondering, since Scipy could solve this equation perfectly, why can't MATLAB? The $\mathrm{roots}()$ function even doesn't provide a chance to choose the initial points.

Problem 2.

Abstract.

(a) Program row and column oriented algorithms and compare the time cost between them
(b) Program algorithms for update a matrix using different methods and compare the time cost.

Introduction and Algrithm.

For problem (a), we will use back substitution algorithm in row-oriented and column-oriented ways for solving an upper triangular system.

Algorithm 1.1, Row-oriented Back Substitution

Input: An upper triangular matrix $U \in \mathbb{R}^{n\times n}$, and a vector $b \in \mathbb{R}^n$.
Output: Updated $b \in \mathbb{R}^n$.

$b(n) = b(n)/U(n,n)$
for $i = n-1$ **to** 1 **by** -1
    $b(i) = (b(i) - U(i, i+1:n) \cdot b(i+1:n))/U(i,i)$
return b

Algorithm 1.2, Column-oriented Back Substitution

Input: An upper triangular matrix $U \in \mathbb{R}^{n\times n}$, and a vector $b \in \mathbb{R}^n$.
Output: Updated $b \in \mathbb{R}^n$.

for $j = n$ **to** 2 **by** -1
    $b(j) = b(j)/U(j,j)$
    $b(1:j-1) = b(1:j-1) - b(j) \cdot U(1:j-1, j)$
$b(1) = b(1)/U(1,1)$

These two algorithms do not need extra space.

For problem (b), the five algorithms are shown below.

Algorithm 2.1, Loop-update

Input: Three matrices $A, B, C \in \mathbb{R}^{n\times n}$.
Output: The updated $C \in \mathbb{R}^{n\times n}$.

for $i = 1$ **to** n
    for $j = 1$ **to** n
        for $k = 1$ **to** n
            $C(i,j) = C(i,j) + A(i,k) \cdot B(k,j)$
return C

Algorithm 2.2, Dot-update

Input: Three matrices $A, B, C \in \mathbb{R}^{n\times n}$.
Output: The updated $C \in \mathbb{R}^{n\times n}$.

for $i = 1$ **to** n
    for $j = 1$ **to** n
        $C(i,j) = C(i,j) + A(i,:) \cdot B(:,j)$
return C

ALGORITHM 2.3, DAXPY-UPDATE

Input: Three matrices $A, B, C \in \mathbb{R}^{n\times n}$.
Output: The updated $C \in \mathbb{R}^{n\times n}$.

```
1  for i = 1 to n
2      for j = 1 to n
3          C(:,i) = C(:,i) + A(:,j) · B(j,i)
4  return C
```

ALGORITHM 2.4, MATVEC-UPDATE

Input: Three matrices $A, B, C \in \mathbb{R}^{n\times n}$.
Output: The updated $C \in \mathbb{R}^{n\times n}$.

```
1  for i = 1 to n
2      C(:,i) = C(:,i) + A · B(:,i)
3  return C
```

ALGORITHM 2.5, OUTERDOT-UPDATE

Input: Three matrices $A, B, C \in \mathbb{R}^{n\times n}$.
Output: The updated $C \in \mathbb{R}^{n\times n}$.

```
1  for i = 1 to n
2      C = C + A(:,i) · (i,:)
3  return C
```

The template used here for these algorithms is clrscode3e.

Code and Implementation issues. The code can be found in the attachments(row_back_substitute.m, col_back_substitute.m, mmloop.m, mmdot.m, mmdaxpy.m, mmmatvec.m, mmouterdot.m). And the test script is test.m.

Code for checking and handling errors in each function is omitted, since I think in this problem it is of less importance.

The input and output of these algorithms are listed within the algorithms, and they only used scalar and vector multiplication contained in MATLAB.

Verification: It is not difficult to find that all these algorithms are equal to the first, triple-loop algorithm.

Result. All the data used are sampled by using rand(). To be specific, each number in each matrix is sampled with a rand().

For problem (a), the time cost is in Table 3 below. (I know that MATLAB is using some pre-compile skills to accelerate the computing speed, so I run the same test several times until the time cost converge to a stable status.)

However, after test my program several times, I found that when n is small, which, to be

n	128	256	512	1024
row-oriented	0.000477	0.000957	0.002858	0.009390
col-oriented	0.000436	0.000912	0.002631	0.006705

Table 3: Time cost of solving a upper triangle system

exact, when $n < 50$, difference between solution of row-oriented and col-oriented algorithms $\|x_r - x_c\|$ is quite small. In this case $\|x_r - x_c\| \leq 10^{-4}\|U\|$. However, when n is

larger, the difference becomes larger. Table 4 shows the relative norm of difference vectors between result of the two als and result of directly using MATLAB's inner functions(i represents MATLAB's inner function).

$$\|\Delta_{error}\| = \frac{\|x_1 - x_2\|}{\|U\|}$$

n	128	256	512	1024
r-c	$5.948e^{-6}$	$1.504e^{27}$	$6.904e^{85}$	$4.179e^{168}$
r-i	$5.680e^{-6}$	$1.521e^{27}$	$2.952e^{85}$	$6.968e^{167}$
c-i	$1.091e^{-6}$	$1.693e^{25}$	$2.261e^{85}$	$4.875e^{168}$

Table 4: Relative norm of difference vector

For Problem (b), the time cost for als is shown in Table 5 below.

n	128	256	512	1024
loop	0.050033	0.425775	3.196842	39.002625
dot	0.026959	0.134890	0.787452	13.141148
daxpy	0.003710	0.023131	0.158915	1.182440
matvec	0.001167	0.005713	0.026144	0.375322
outerdot	0.007415	0.030481	0.426991	4.375097

Table 5: Time cost of updating a matrix

The norm of relative error (with MATLAB inner *) is shown in Table 6 below.

$$\|\Delta_{error}\| = \frac{\|\hat{C} - C\|}{\|C\|}$$

n	128	256	512	1024
loop	$1.841e^{-15}$	$2.614e^{-15}$	$1.408e^{-14}$	$2.660e^{-14}$
dot	$3.515e^{-15}$	$6.756e^{-15}$	$5.502e^{-15}$	$6.474e^{-15}$
daxpy	$1.144e^{-15}$	$2.614e^{-15}$	$1.408e^{-14}$	$2.660e^{-14}$
matvec	$1.178e^{-15}$	$1.643e^{-15}$	$1.385e^{-14}$	$2.628e^{-14}$
outerdot	$1.195e^{-13}$	$2.614e^{-15}$	$1.408e^{-14}$	$2.660e^{-14}$

Table 6: Norm of error in updating a matrix

Discussion. From result of problem (a), we found that column-oriented algorithm is slightly quicker that row-oriented algorithm in solving upper triangular system.

From the algorithm's point of view, in i^{th} loop, col-oriented al need to do (i-1) times' multiplication, (i-1) times' subtraction, and 1 division, while row-oriented al need to do (n-i) times multiplication, (n-i) times addition, and 1 division.

In consequence, C-O al need to do $\frac{(n-1)(n-2)}{2}$ times' multiplication and substraction, and (n-1) times' division; while R-O al need to do 1 more division than C-O. So I am curious why C-O cost only 2/3 time comparing with R-O when $U \in \mathbb{R}^{1024\times1024}$.

On the other hand, since the matrices are generated by random numbers, it is of much possibility that those matrices are nearly singular. So it is not difficult to understand why the

result of x has such a huge error.

For problem (b), result of the five algorithms are of good precision, which can be found with norm of errors.

Let's analysis the flops of the als. Loop update needs n^3 times of loops, each contains a multiplication and an addition. Dot update needs n^2 times of loops, each contains an addition and a Vectorized Dot. Daxpy update needs n^2 times of loops, each contains an Vectorized Addition and a Vectorized Scalar multiplication with a vector. MatVec update needs n times of loops, each contains a Vectorized Addition and a Vectorized Scalar multiplication with a matrix. OuterDot update needs n times of loops, each contains a Vectorized Addtion with a matrix and an outer dot of two vectors.

Though the total flops of the five als are the same, we all know that in MATLAB, for loops are much slower that vectorized computations. So the result of experiment is trivial.

Reference

- Jixiu Chen *et al*, *Mathematical Analysis II*, Page 174-175.
- D. P. O'Leary, *Scientific Computing with case studies*, Page 23-24.
- Gene H. Golub *et al*, *Matrix Computations*, Page 10-11.
- Shufang Xu *et al*, *Numerical Linear Algebra*, Page 11-13.