어셈블리프로그램 설계 및 실습

프로젝트 결과 보고서

과제제목: term project

제출일자: 2019년 11월 30일 (토)

학 과: 컴퓨터정보공학부

담당교수: 이준환 교수님

학 번: 2018202074

성 명: 김상우

1. Introduction

-제작할 프로젝트

부동소수점 데이터로 이루어진 N\*N 정방행렬에 대한 역행렬을 구현하는 것이다. 단 역행렬이 없는 경우는 고려하지 않으며, 이 때 코드는 최대한 적은 state로 움직여야 한다.

-예정일정

|  |  |  |  |
| --- | --- | --- | --- |
| 기간 | 11.07-11.14 | 11.15-11.21 | 11.22-11.28 |
| 알고리즘 구상 및 제안서 작성 |  |  |  |
| 코드 작성 |  |  |  |
| 코드 검증 및 수정 |  |  |  |
| 최종 결과 보고서 작성 |  |  |  |

프로젝트를 넉넉하게 진행하기 위해 위와 같이 일정을 생각하였다.

-최종 일정

|  |  |  |  |
| --- | --- | --- | --- |
| 기간 | 11.07-11.14 | 11.15-11.21 | 11.22-11.28 |
| 알고리즘 구상 및 제안서 작성 |  |  |  |
| 코드 작성 |  |  |  |
| 코드 검증 및 수정 |  |  |  |
| 최종 결과 보고서 작성 |  |  |  |

원래 일정에선 코드 작성이 2주차의 끝났어야 하나 프로젝트가 다수 겹치게 되면서 일정이 달라지게 되고, 결과적으로 3주차에 일정이 몰리게 되었다. 이로 인해 발생한 시간적인 오류는 일정을 힘들게 만들었다.

1. Project Specification

기본적으로 제공되는 행렬의 경우 DCD의 형식으로 input되며 DCD의 가로,세로 사이즈는 첫 번째 input으로 판단할 수 있다. 행렬 값들은 처음에 행렬의 크기를 위해 제공되는 값은 정수의 형태로, 이후 행렬내의 값들은 실수의 형태로 입력된다. 실수는 총 32bit로 1bit가 sign bit, 8bit가 exponent, 23bit가 Mantissa인 형태로 구성되어 있다.

Bit:0일 때 양수 1일 때, 음수를 의미

Exponent:2^(x-127)의 값

Mantissa1.xxxxx를 의미

최종적으로는 (-1)^(Bit) \* 2^(Exponent-127) \* (1.Mantissa)를 의미하게 된다.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| bit | exponent | | | | | | | | Mantissa | | | | | | | | | | | | | | | | | | | | | | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

그 결과로 나오는 역행렬의 구성 역시 이러한 형태의 실수로 존재해야 한다.

최대 오차율은 2^-10으로 제한된다.

MUL과 DIV를 사용해서는 안되며(이를 대체할 수 있는 코드를 구현을 진행하여야 한다.), 프로그램은 주석을 포함하여 다음과 같이 종료해야 한다. ✓ MOV pc, #0 ;Program end

주어진 Testbench에서 모두 돌아가야 하며 다음과 같은 Label을 사용하여야한다.

Label: Matrix\_data : DCD를 통해 N의 값과 임의의 데이터 NxN개가 저장되어 있는 메모리의 첫 주소 값. 이때 행렬 안의 숫자들을 의미하며 부동소수점으로 제공되며 -2000 ~ 2000 사이의 부동소수점 값을 원소로 가진다.

Label: Result\_data : Matrix\_data의 데이터들을 이용하여 구해진 역행렬이 저장되는 데이터 공간의 시작 주소 값. 저장시에는 1 word 단위로 저장하며, 0x60000000 번지에서 시작된다.

1. Algorithm

-곱셈기

곱셈기의 경우 아래의 이론을 바탕으로 제작되었다.

**Booth Multiplication**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Xi | Xi-1 | Operation | Description | yi |
| 0 | 0 | Shift Only | String of zeros | 0 |
| 0 | 1 | Add and shift | End of a string of ones | 1 |
| 1 | 0 | Subtract and shift | Beginning of a string of ones | 1’ |
| 1 | 1 | Shift only | String of ones | 0 |

Booth Multipliction은 위의 규칙을 따르는 곱셈의 방법이다.

Binary의 형식으로 수를 표현하는 방식에서 X=xn-1\*2^(n-1)+xn-2\*2^(n-2)+…+x0\*2^0를 따르므로

X=xn-1\*2^(n-1)+X’라고 할 때, X’’=2^n-X(X’’는 X의 보수가 된다.)라 가정하면

이를 바탕으로 (-X’’) = X-2^n = X’-2^(n-1) 임을 알 수 있다.

위의 내용을 바탕으로 A\*X= A\*(X’-2^(n-1))=A\*X’-A\*2^(n-1)을 유추해 낼 수 있다.

이 공식을 바탕으로 진행되는 것이 multiplier가 음수일 때의 곱셈을 수행할 수 있다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Xn-1 | Xn-2 | Booth rule  (consider Xn-1=0) | Compensation for the sign bit | Yn-1 |
| 1 | 0 | Shift only | -A | -1 |
| 1 | 1 | Add and shift | -A | 0 |

위에서 확인할 수 있듯이 한 번에 2bits씩 확인된다.

다만 이를 위해서는 Mantissa들끼리만의 곱셈이 성립된다. 이를 해결하기 위해서 LSL과 LSR을 이용해 실수내의 bit, exponent, mantissa를 분리해내었다.

bit의 경우 (-)\*(+) 땐 음수가, 같은 부호일 땐 양수임을 이용, CMP와 MOVEQ를 이용해 bit를 결정했다. exponent는 2^A\*2^B가 2^(A+B)임을 이용해 exponent는 더해주었다. Mantissa는 위의 booth multipication을 이용,각 Mantissa를 24비트로 변경해주고, 이를 booth multiplication을 진행해주었다. 이 때, 48bit가 나올수가 있는 bit를 처리하기 위해 2개의 레지스터를 사용하였으며, 하위 24bit의 MSB를 LSR을 통해 찾아내어 이를 진행하였다. (Muliplicand에서 LSB찾을 때도 마찬가지이다.)

이후 발생된 상위 24bit와 하위 24bit의 결과를 #0x800000과 #1000000의 사이가 될 때까지 정상화해주었다. 이 과정에서#0x800000 이하일 때는 LSL, #1000000이상일 때는 LSR이 사용되었으며 LSL이 될 때마다 exponent를 1감소, LSR이 될 때마다 exponent를 1 증가 시켜주었다.

최종 결과에서 0x8000000을 빼주어 Mantissa을 완성하고 bit, exponent, Mantissa로 float 를 제작하였다.

Block Diagram은 다음과 같이 그려진다.

-나눗셈기

곱셈기와 비슷한 양상으로 제작하였다. bit의 경우 (-)\*(+) 땐 음수가, 같은 부호일 땐 양수임을 이용, CMP와 MOVEQ를 이용해 bit를 결정했다. exponent는 2^A/2^B가 2^(A-B)임을 이용해 더 큰 exponent를 cmp로 찾고 빼주어 exponent는 더해주었다. Mantissa의 경우 exponent의 차이만큼 exponent가 작았던 Mantissa(B)의 값을 LSR해주었다. 이후 아래의 알고리즘을 사용하였다. 이때 레지스터 2개를 사용한다.

1. 레지스터에 exponent값이 큰 Mantissa(A)를 레지스터에 준다.

2. A에서 B를 뺄 수 있으면 레지스터(B)에 1을 더한다. 0인 경우 exponent를 1 감소시킨다.

3. A를 LSL한다.(MSB제거되지 않도록 32bit 레지스터 사용)

4. B를 LSL한다.

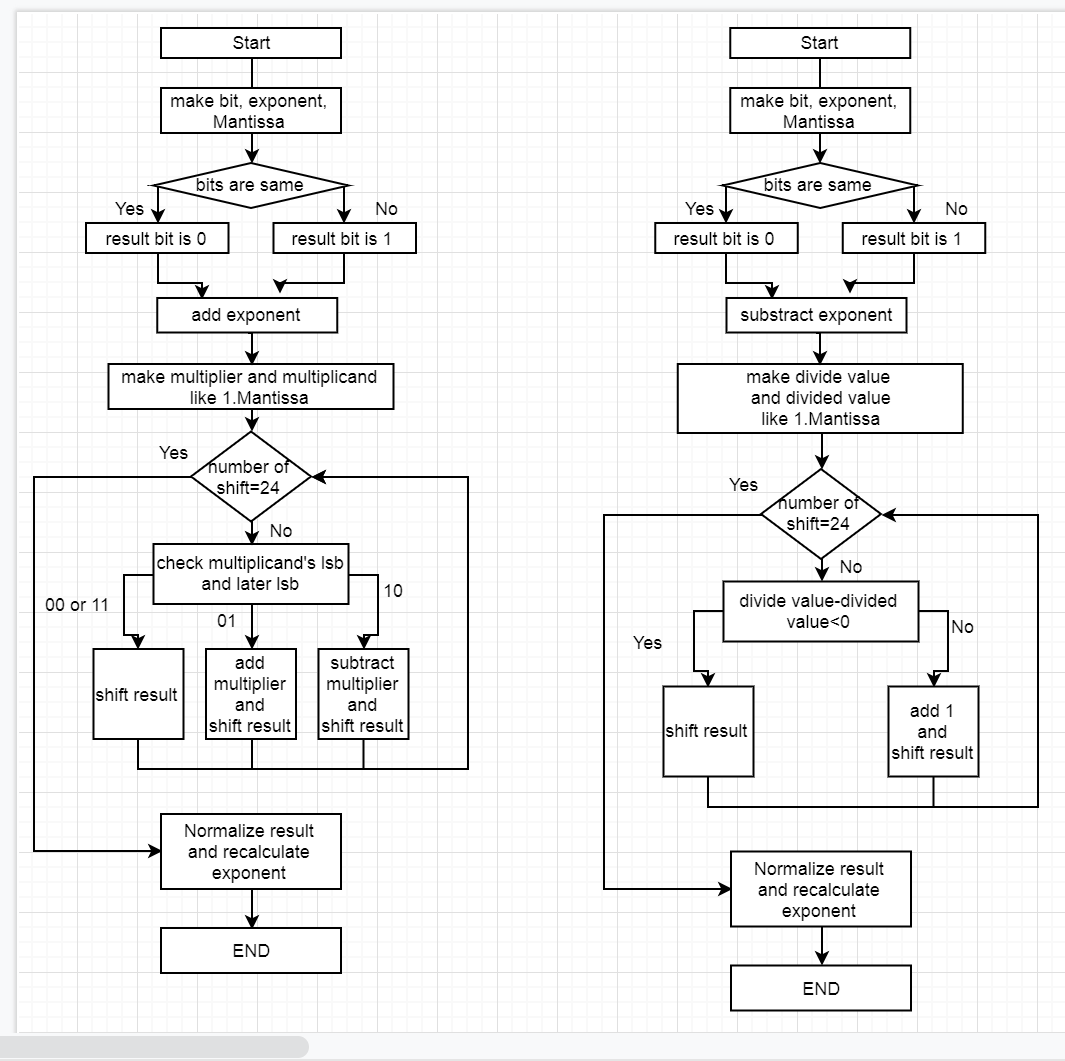
5. A에서 B를 뺄 수 있으면 레지스터(B)에 1을 더한다.

6. A를 LSL한다.(MSB제거)

7. B를 LSL하고 B를 LSL한 횟수가 24번이 아니라면 4번으로 돌아간다.

이후 레지스터 B에 저장된 24bit의 수중 MSB를 제외한 23bit를 Mantissa로서 사용한다.

곱셈기와 나눗셈기의 Diagram은 다음과 같다.



-전체 알고리즘

가우스 조던 방식의 아랫줄만 모두 0으로 바꾸어주고 이후 아래를 기반으로 위의 값들을 빼주는 것은 너무 많은 과정이 진행된다고 생각하여 다음과 같이 알고리즘을 수정했습니다.

|  |  |  |
| --- | --- | --- |
| A | B | C |
| D | E | F |
| G | H | I |

|  |  |  |
| --- | --- | --- |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

Main Matrix Sub Matrix

|  |  |  |
| --- | --- | --- |
| A | B | C |
| D | E | F |
| G | H | I |

|  |  |  |
| --- | --- | --- |
| 1 | B/A | C/A |
| D | E | F |
| G | H | I |

|  |  |  |
| --- | --- | --- |
| 1 | B/A | C/A |
| 0 | E-(DB/A) | F-(DC/A) |
| 0 | H-(GB/A) | I-(GC/A) |

1. 기존 Matrix와 똑 같은 크기의 Sub Matrix를 만든다. 아래 행동들은 SubMatrix에서도 진행한다.
2. MainMatrix의 첫 행의 첫 칸을 저장하고, 모든 Matrix의 첫 행의 값들을 해당 값으로 나누어 준다.

이후 다른 행-(다른 행 첫 번째 칸의 값)\*(첫 행)의 형식으로 다른 행들의 첫 번째 값들이 0이 되게 함.

|  |  |  |
| --- | --- | --- |
| 1 | B/A | C/A |
| 0 | 1 | J |
| 0 | H-(GB/A) | I-(GC/A) |

|  |  |  |
| --- | --- | --- |
| 1 | 0 | C/A-(JB/A) |
| 0 | 1 | J |
| 0 | 0 | I-(GC/A)-(JB/A) |

1. MainMatrix의 N 행의 N 칸을 저장하고, 모든 Matrix의 N 행의 값들을 해당 값으로 나누어 준다.

이후 다른 행-(다른 행 첫 번째 칸의 값)\*(첫 행)의 형식으로 다른 행들의 첫 번째 값들이 0이 되게 함.

이 때 가우스 조던 알고리즘과는 다르게 위의 값들도 그 범위에 해당시킴.

1. 이를 마지막 행까지 반복한다. \* J= {F-(DC/A)}/{E-(DB/A)}

|  |  |  |
| --- | --- | --- |
| 1 | 0 | C/A-(JB/A) |
| 0 | 1 | J |
| 0 | 0 | 1 |

|  |  |  |
| --- | --- | --- |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

(마지막 Main Matrix), 이 때의 SubMatrix가 최종 결과가 된다.

위 공식을 이용하기 위해서 다음과 같은 값들을 저장할 필요가 있었습니다. 그렇기에 저는 추가적으로 address를 더 선언했습니다.

Address0: 행렬의 가로길이, 현재 사용된 y point, column간 길이, 사용중인 MainMatrix array, 사용중인 SubMatrix array, 나눠지는 수, 나눈 횟수, 곱해지는 수, 곱해진 수, 더해지는 array의 시작 주소, 더해지는 Sub array의 시작주소, 더해진 횟수가 순서대로 저장되는 일종의 변수 저장을 위한 메모리이다.

Address1:MainMatrix이다.

Address3:가우스 조던 시 더해지는 array를 저장한다.

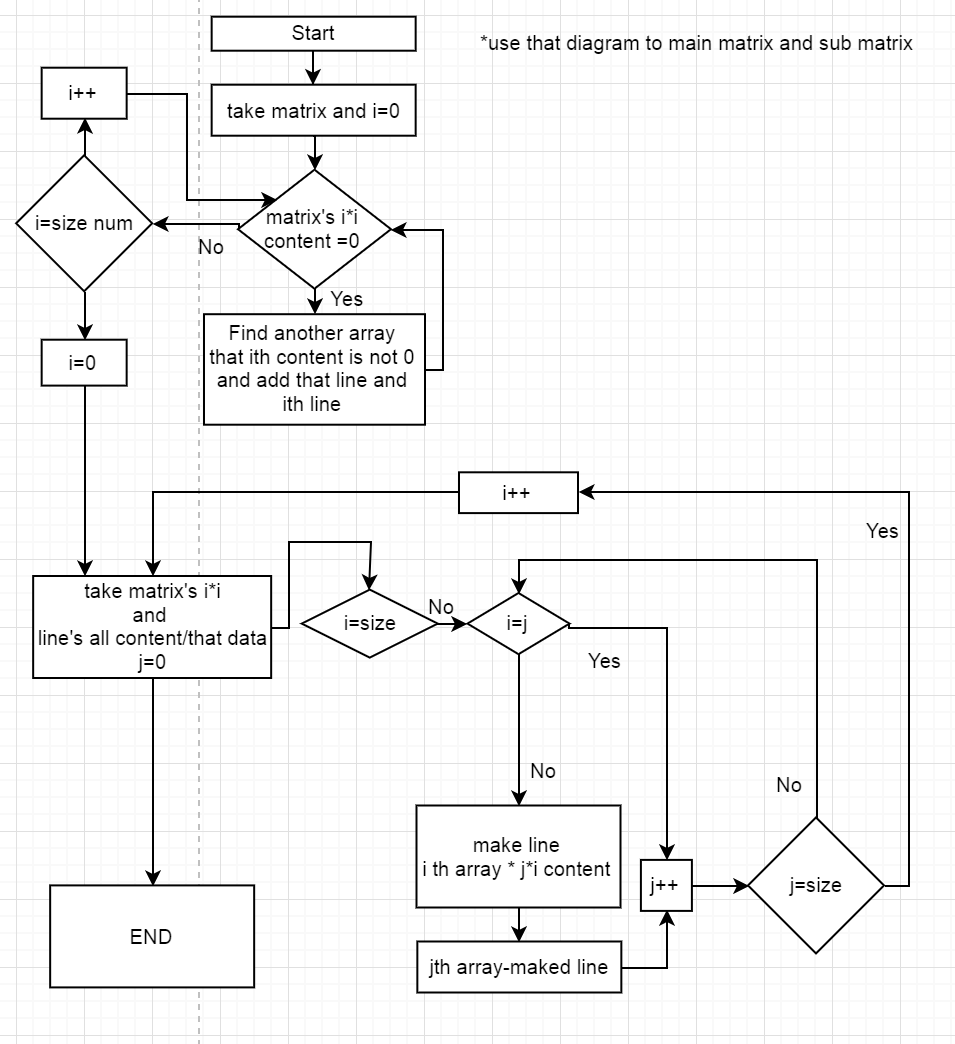
Address4:가우스 조던 시 더해지는 sub array를 저장한다.

위 내용들을 바탕으로 전체알고리즘은 다음과 같이 진행했습니다.

1. 반복을 통해 Address1에 MainMatrix의 값들을 받아오고, Address0에 필요한 값들을 넣어준다.
2. 반복을 통해 Sub Matrix를 만든다.
3. Nii의 값이 0인지 위에서부터 확인한다.   
   만약 Nii가 0이면 위에서부터 Nji가 0이 아닌 array를 찾고 이를 더한다.(이는 subMatrix에도 적용된다)  
   모든 Nii가 0이 아닐 때 까지 반복한다.
4. i=0으로 한다.
5. Nii의 값을 저장하고 그 행의 모든 값을 Nii로 나눠준다. (이는 subMatrix에도 적용된다)
6. 나눠진 행을 따로 저장하고 그 행에 다른 행의 i번째 값을 곱하고 그 다른 행에서 곱해진 행을 뺀다.  
   MainMatrix의 i번째 열이 하나를 제외한 모든 행이 0이 될 때까지 이를 반복한다.

(이는 subMatrix에도 적용된다)

1. i가 행렬의 가로 사이즈와 같지 않다면 i에 1을 더하고 5로 돌아간다.
2. Sub Matrix에 역행렬 완성

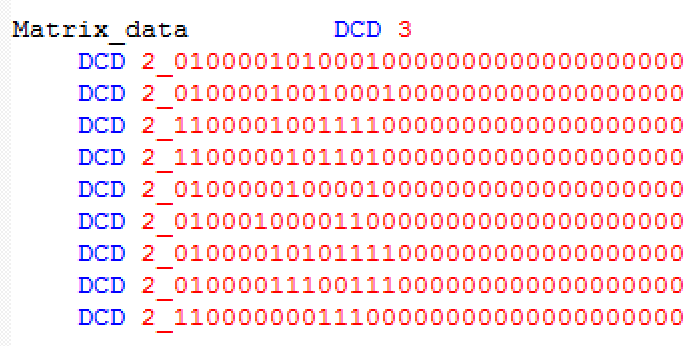


1. Performance & Result

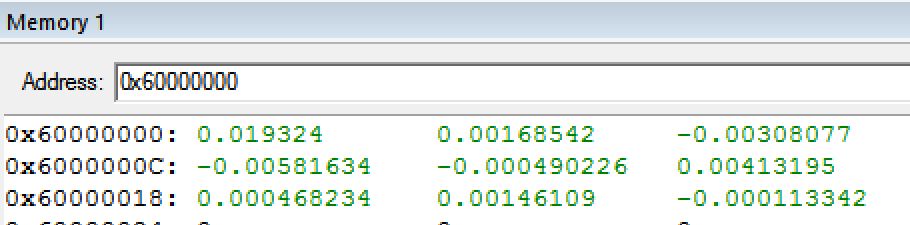
TestBench 1

Input

|  |  |  |
| --- | --- | --- |
| 68 | 49 | -62 |
| -14.5 | 8.5 | 704 |
| 94 | 312 | -3.75 |

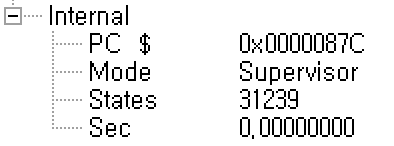


|  |  |  |
| --- | --- | --- |
| 0.019324 | 0.00168542 | -0.00308077 |
| -0.00581634 | -0.000490225 | 0.00413195 |
| 0.000468234 | 0.00146109 | -0.000113342 |



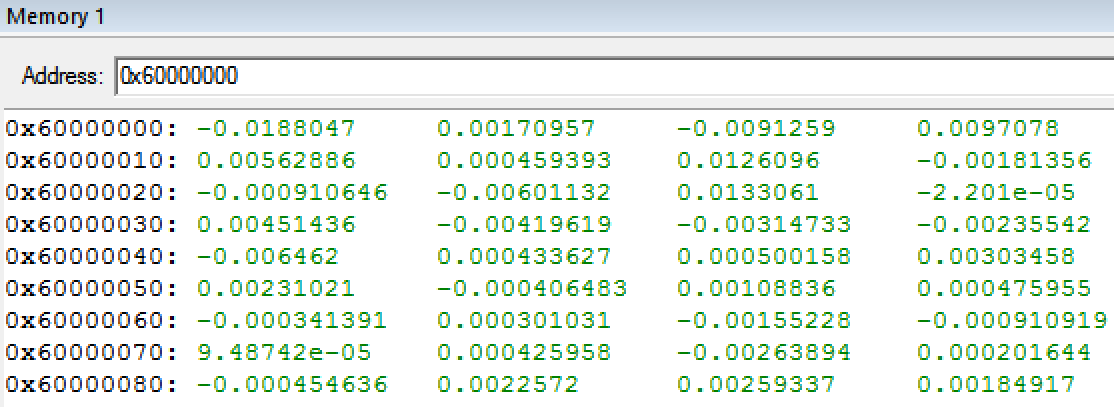
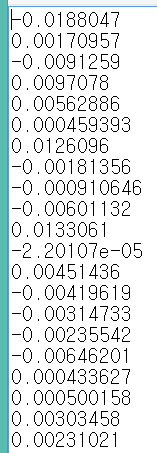
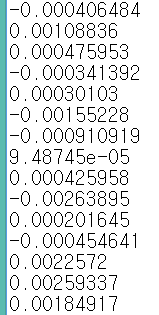
좌측은 실험에 의한 값, 우측은 우리가 기대한 값들의 모습이다.

3\*3 행렬을 집어넣었을 때 Testbench에서 기대한 값과 실제 값의 오차가 전혀 발생하지 않는 모습이다.

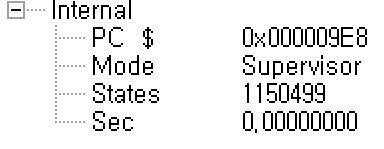


State는 다음과 같다. 곱셈기나 나눗셈기를 수정한다면 더 효율적으로 State를 줄일 수 있을 지도 모른다.

TestBench 2

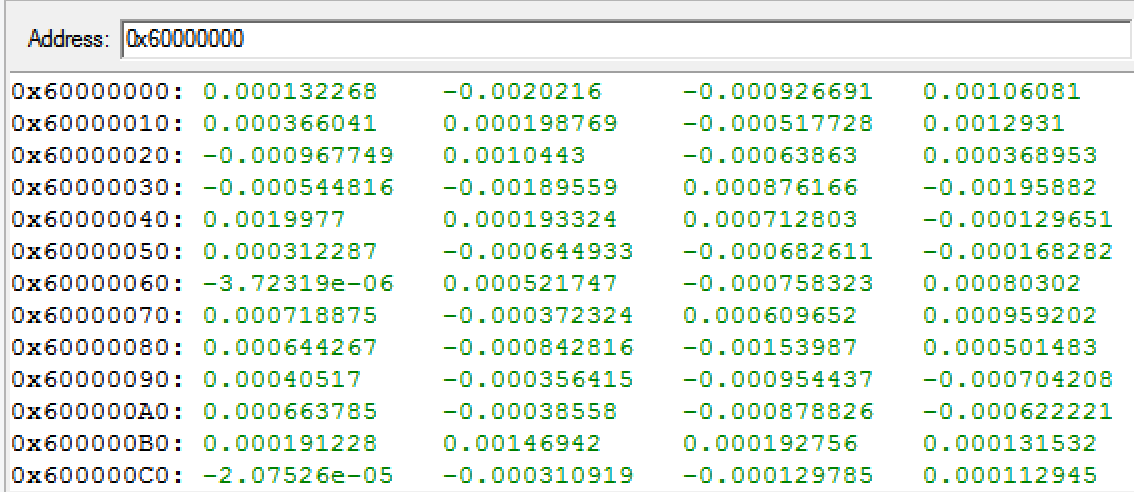
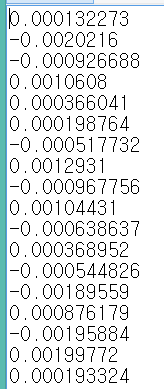
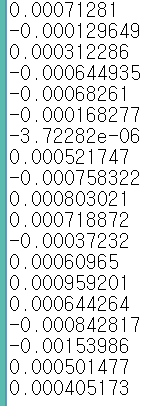
  

위쪽은 코드를 돌렸을 때, 아래쪽은 예상되는 값이다. (골든 Value)-(실험값)/(골든 Value)<2^-10임을 확인 할 수 있다. 이는 다른 값들도 마찬가지임을 테스트를 통해 확인할 수 있다.

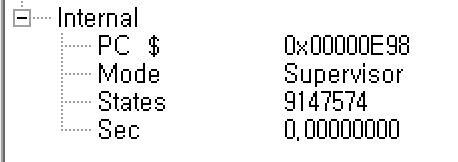
State은 다음과 같다. 2\*2나 5\*5를 활용해 state를 줄일 수 있을 것이다.

TestBench 3

위쪽은 코드를 돌렸을 때, 아래쪽은 예상되는 값이다. (골든 Value)-(실험값)/(골든 Value)<2^-10임을 확인 할 수 있다. 이는 다른 값들도 마찬가지임을 테스트를 통해 확인할 수 있다.

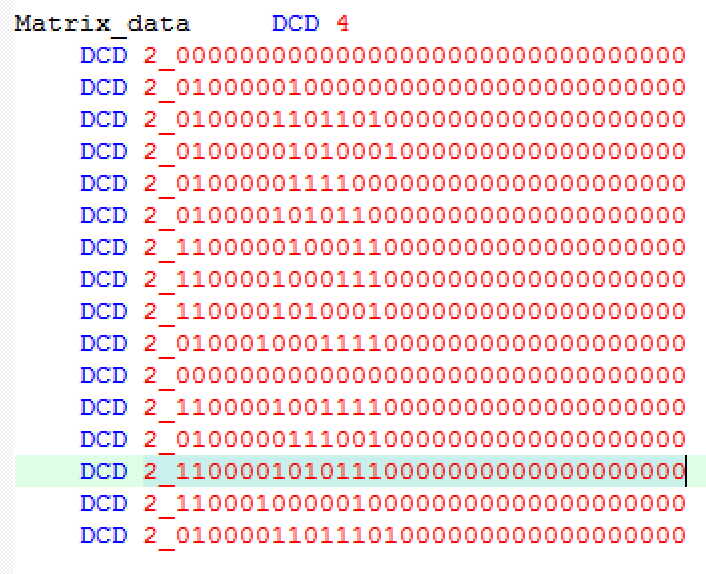
  

위쪽은 코드를 돌렸을 때, 아래쪽은 예상되는 값이다. (골든 Value)-(실험값)/(골든 Value)<2^-10임을 확인 할 수 있다. 이는 다른 값들도 마찬가지임을 테스트를 통해 확인할 수 있다.

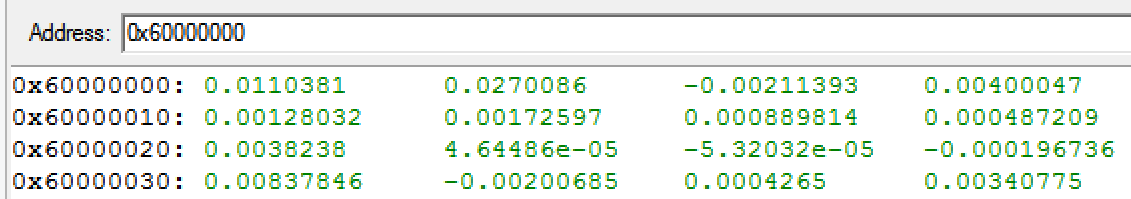
State은 다음과 같다. 4\*4나 5\*5를 이용해 줄일 수 있을 것이다.

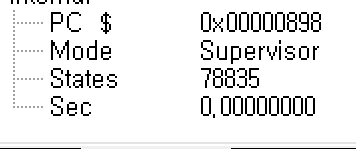
TestBench 4

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 8 | 232 | 12.25 |
| 28 | 88 | -9.5 | -46 |
| -68 | 992 | 0 | -62 |
| 25 | -92 | -576 | 244 |



|  |  |  |  |
| --- | --- | --- | --- |
| 0.0110382 | 0.0270086 | -0.00211393 | 0.00400047 |
| 0.0012803 | 0.00172597 | 0.000889814 | 0.00048721 |
| 0.0038238 | 4.6449e-05 | -5.32032e-05 | -0.000196736 |
| 0.00837845 | -0.00200685 | 0.000426501 | 0.00340775 |

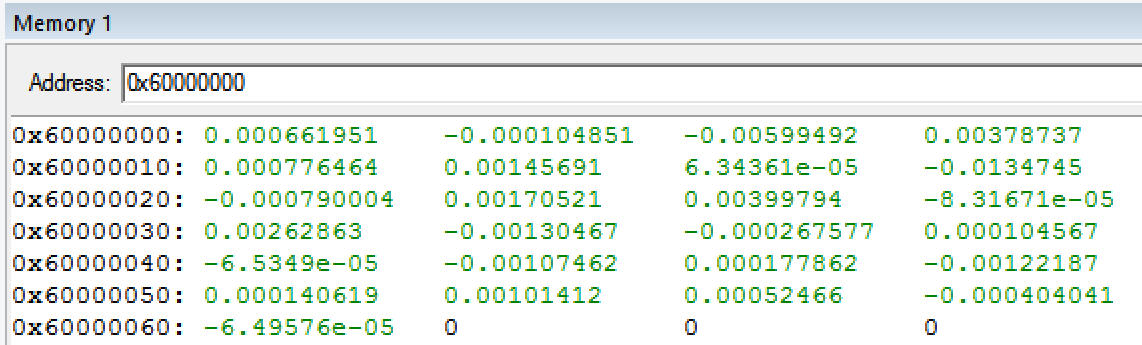
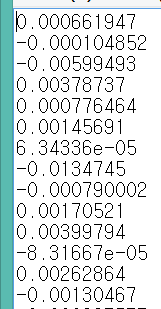
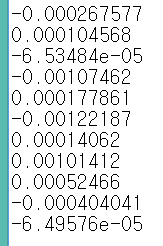


State는 다음과 같다. Sort가 상당한 State를 사용하는 것을 알 수 있다.

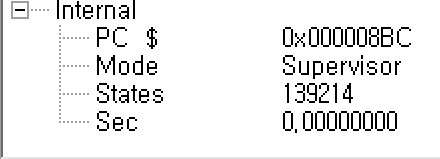
역으로 Sort 수정으로 State의 확보가 가능할지도 모른다.

위 아래를 비교해보면 6자리로 표현되는 수에서 하위 1,2bit에서만 차이가 날 뿐, 그 이상의 오차는 발생하지 않는 것을 확인할 수 있다. 제한된 2^-10=1/1024임을 생각해본다면 1/10000의 오차도 발생하지 않았기 때문에 성공적인 결과라고 할 수 있을 것이다.

TestBench 5

위쪽은 코드를 돌렸을 때, 아래쪽은 예상되는 값이다. (골든 Value)-(실험값)/(골든 Value)<2^-10임을 확인 할 수 있다. 이는 다른 값들도 마찬가지임을 테스트를 통해 확인할 수 있다

State는 다음과 같다. Size가 늘어남에 따라 State가 늘었다. 이 또한 TestBench 4와 같은 방식으로 State를 줄일 수 있을 것이다.

1. Conclusion

결과적으로 위 코드는 성공적으로 마무리 되었다. 최대 사이즈인 20\*20을 돌려본 결과, 하위 2자리 이상의 오차는 발생하지 않았으며, 그것 또한 2^-10보다 높은 오차를 내지 않음을 비교를 통해 확인하는 것이 가능했다. 단, 위 같은 알고리즘을 사용했을 경우, 실수가 표현할 수 있는 범위보다 낮은 값이 되거나 높은 값이 되어 overflow가 발생하였을 때 기존 가우스 조던 방식보다 오차 발생이 일어날 가능성이 크단 점이 문제가 되었다. 다만 State를 줄일 수 있었다. 이러한 과정을 통해 다양한 방법을 숙지하고 있으면 원하는 상황에 다양한 알고리즘을 구현할 수 있음을 알 수 있었다. 이는 앞으로의 공부를 위한 동기가 될 것이다.

동기로서만이 아니라 이번 프로젝트를 통해서 역행렬의 다양한 풀이법을 스스로 찾아볼 수 있었다.  
(예로 하위 행렬들을 먼저 구하고 상위 행렬을 구하거나 혹은 위 알고리즘처럼 계산을 단순화하는 방법 등) 앞으로도 행렬을 사용함에 있어서 좋은 복습 및 공부가 되었다고 생각한다.

그 외에도 Mul과 Div를 사용하지 않고 구현을 하였는데, 오히려 이를 이용하지 않았지 때문에 오차율을 낮추기 위해 코드를 여러 번 수정하며 더 나은 코드를 만들 수 있었던 것 같다.