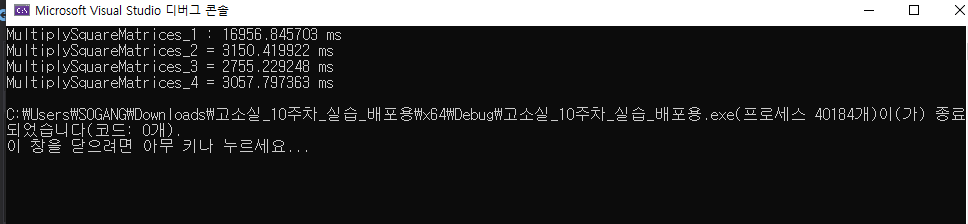
**고급소프트웨어실습 10주차 보고서**

**20191657 최세은**

**실습 1.**

**행렬 계산의 속도 향상을 위하여 (1) 자신이 적용한 방법과 (2) 어떠한 근거로 자신이 적용한 방법이 더 효율적일지, 그리고 (3) 어떤 m 값에 대해 loop unrolling 방법이 가장 효과적이었는지를 요약할 것**



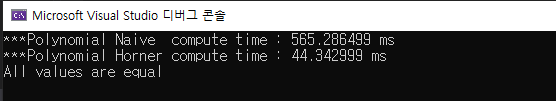
(1) 우선 MultiplySquareMatrices\_1 에서는 행렬의 곱셈을 응집성 없이 수행했기 때문에 시간이 오래 걸린다. 그리고 MultiplySquareMatrices\_2 에서는 시간을 줄이기 위해 전치 행렬을 사용해서 응집성을 갖는 행렬의 곱셈을 구현하였다. MultiplySquareMatrices\_3과 4에서는 loop unrolling 기법을 사용하여 행렬의 곱셈을 수행했다.

(2) c++에서 행렬에 접근할 때 row major로 접근한다. 그러나 MultiplySquareMatrices\_1 에서는 column major로 메모리에 접근했기 때문에 접근 오류가 일어나 여러 번 블록에 접근을 해야한다. 그러므로 MultiplySquareMatrices\_2가 더 효율적인 함수이다. 또한 loop unrolling을 사용하게 되면 반복문에서의 비교 연산 횟수가 줄어드므로 더 효율적으로 코드가 실행된다.

(3) m이 8일 때와 32일 때 확인해 본 결과 8일 때가 더 효과적이었다.

**실습 2.**

**(iv) 다음 적절한 함수를 작성하여 위에서 각각 두 함수를 통하여 구한 y[i] 값이 같음을 보여라.**

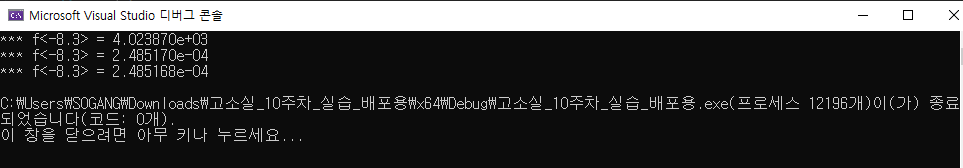


**(v) 이제 DEGREE는 10 정도, 그리고 N X는 1,048,576개 이상 크게 설정한 후, 위 두 함수의 수행 시간을 비교하라. 과연 여러분도 다음과 같은 정도의 차이를 보이는 실험 결과를 얻었는가?**

얻었다.

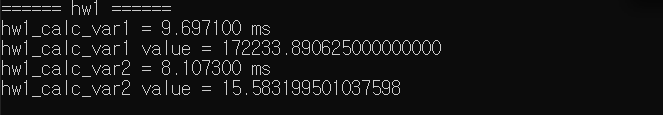
horner’s rule을 사용해서 계산하는 것이 더 효율적임을 이번 실습을 통해 알 수 있었다. 일반적으로 덧/뺄셈보다 곱/나눗셈을 수행하는 데 더 오래 걸리고, 특히 나눗셈이 더 오래 걸린다. 그래서 이번 실습에서 일반적인 방법으로 계산해서 더하는 방법과 horner’s rule을 사용해서 계산하는 방법의 걸린 시간을 비교하니 후자가 훨씬 적은 시간이 들었다. horner’s rule을 사용하면 y[i]를 구할 때 x[i]에 대해 거듭제곱 연산을 수행하지 않아 곱셈의 횟수가 줄어들기 때문이다.

**실습 3.**

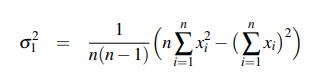
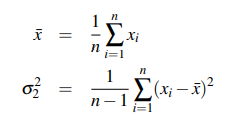


이번 실습에서의 x는 음수이다. 따라서 테일러 급수식에 맞추어 무한 급수를 전개해보면 양수+음수+양수+음수+… 의 형태이다. 이 때 다음 항을 계산할 때 점점 소수점 아래 자리수는 늘어나는 데 유효숫자의 자리를 정해져 있기 때문에 오차가 생길 수밖에 없다. 이 때 float 대신 double 자료형을 사용하면 유효숫자의 자리수가 더 늘어나기 때문에 오차가 줄어든다. 또한 위 실습 2에서 사용했었던 horner’s rule을 사용하면 곱셈 연산의 횟수가 줄어들어 연산 시간이 줄어들 것이다.

**과제 1.**



**(ii) 두 분산 값 계산 방법의 결과가 상당히 차이가 나게 해주는 샘플 데이터를 생성한 후, 계산 결과를 비교분석하라. 분산을 어떻게 계산한 것이 더 정확한 것으로 판단되는가? (즉 수학적으로 동일한 두 식이 컴퓨터상에서는 얼마나 다를 수 있는지를 스스로 파악하는 것이 목적임. 일반적으로 후자의 방법이 더 위험하다고 알려져 있음. 참고로 분산 값은 음수일 수 없음)**



위 두 방법 중 첫번째 방법으로 분산을 계산하는 것이 더 정확했다. 두번째 방법으로 계산했을 때에는 두 식의 값이 매우 커져 정확도가 낮아질 수 있고, 둘의 값이 비슷해져 비슷한 수들로 뺄셈을 하게 될 가능성이 있었다.

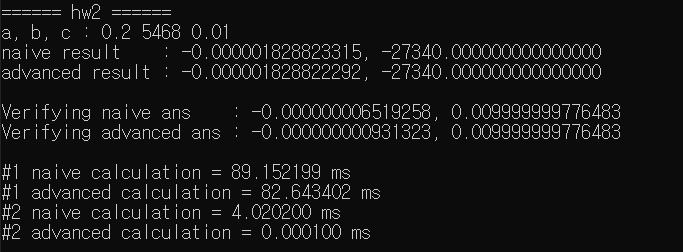
**(iii) 충분히 큰 n에 대하여 두 방법 중 어떤 방법이 더 빠르게 분산 값을 계산하는가?**

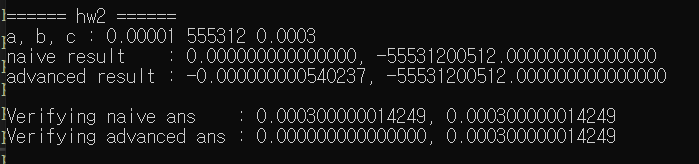
대체적으로 두번째 방법이 더 빠르게 분산 값을 계산했다.

**과제 2.**

**(ii) 위 프로그램이 심각한 문제를 야기하는 상황을 세 가지 발생시켜라. 즉 그런 문제를 일으킬 a, b, 그리고 c값을 적절히 설정한 후, 위에서 구한 두 근을 다시 f(x)에 대입하여 0이 나오는지 확임함으로서 심각한 문제가 발생하였다는 것을 증명하라.**

**(iii) 다음 그러한 문제를 완화시킬 수 있는 방법을 사용하여 위의 2차 방정식을 풀어주는 함수를 새롭게 구현한 후, 위의 문제와 동일한 과정를 거쳐 (즉 자신이 구한 근에 대해 f(x) 함수 값을 구하여), 위에서 심각한 문제를 야기한 세 경우 각각에 대해 자신의 두 번째 함수가 안정적으로 근을 구했음을 밝혀라.**



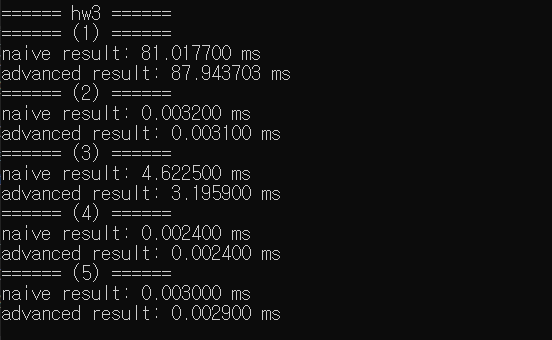


근의 공식 중 에서 4ac의 값이 매우 작을 경우 비슷한 값끼리 뺄셈을 수행하게 되어 오차가 발생할 수 있다. 위 두 결과를 보면 값이 작을수록 오차가 커졌다.

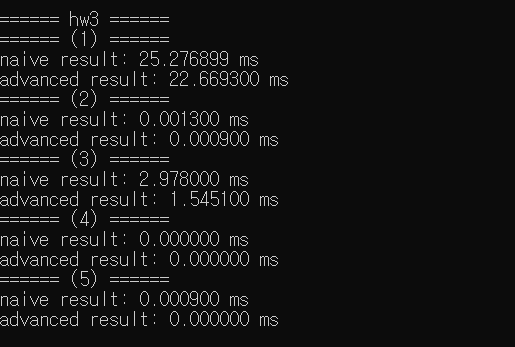
따라서 이 문제를 해결하기 위해 b가 양수이고 b2이 4ac 보다 훨씬 큰 경우 위 식을 로 바꾸어 계산하였다. 결과의 Verifying advanced ans와 Verifying naive ans를 보면 오차가 확실히 줄어든 것을 확인할 수 있었다.

**과제 3.**

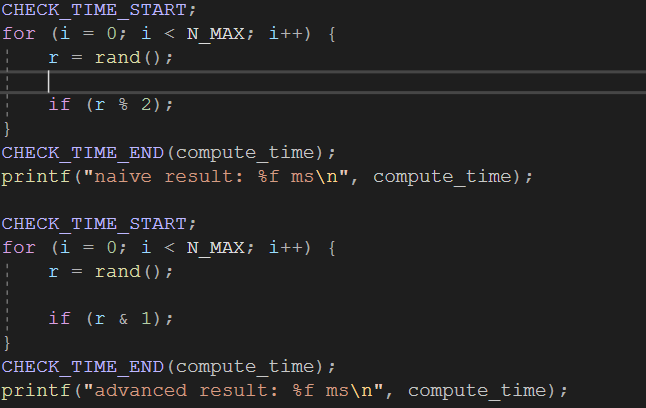
[debug]



[release]

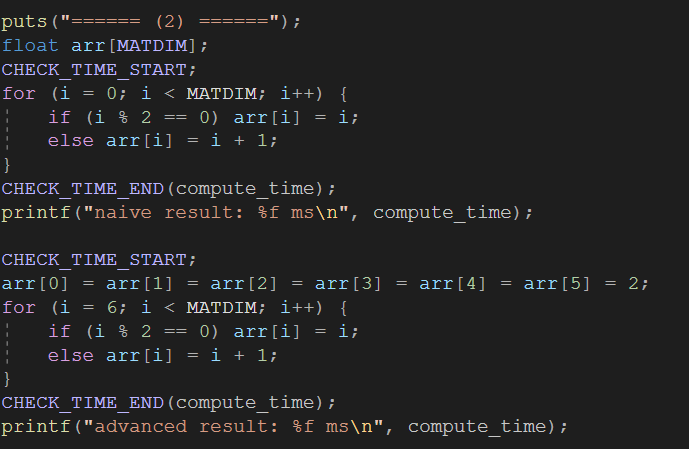


(1) 나눗셈 연산을 시프트 연산으로 바꾸기



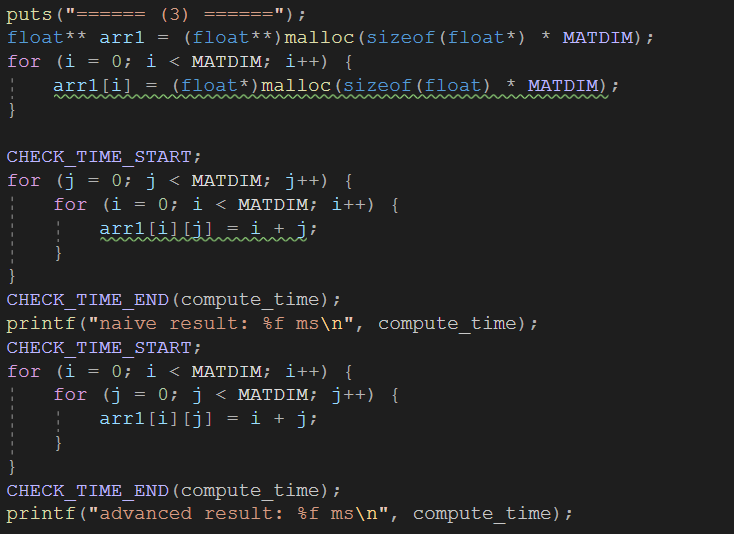
연산 중 나눗셈 연산이 특히 시간이 오래 걸리므로 시프트 연산으로 바꾸어 시간을 줄였다. 위 그림에서는 debug 모드에서 오히려 시프트 연산일 때 오래걸리는데, 실행해보면 90프로는 후자가 더 빠르게 나온다. 그리고 release 모드일 때가 전체 연산 시간과 두 시간의 차이 둘 다 줄어들었다.

(2) loop peeling



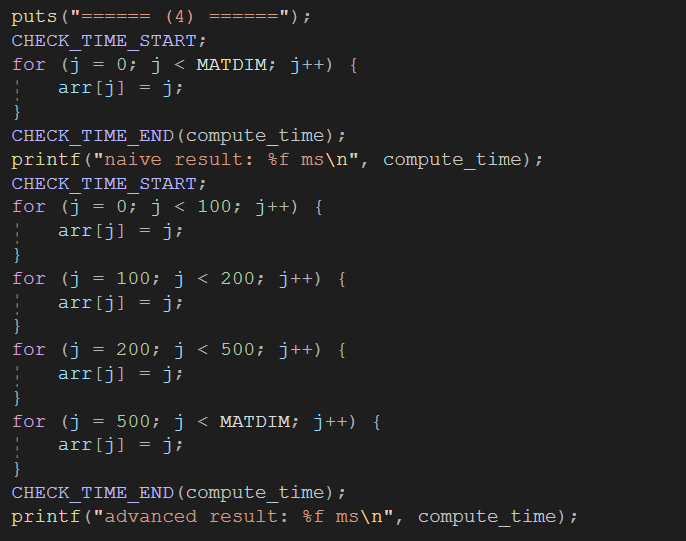
loop의 일부분(주로 처음이나 끝부분)을 따로 바깥으로 떼어내 반복 횟수를 줄이는 기법이다. 위 결과를 보면 후자일 때 시간이 줄어들긴 했으나 차이가 큰 편은 아니었다. 코드에서 반복 횟수를 5번만 줄여서 그런 것으로 예상된다. 그리고 release 모드일 때는 전체 연산 시간과 두 시간의 차이 둘 다 줄어들었다.

(3) loop interchange



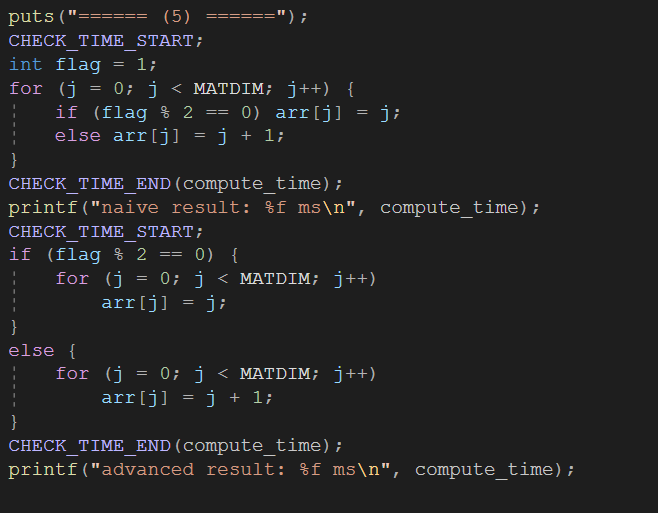
loop의 순서를 뒤바꾸는 기법으로 데이터의 locality를 개선하는 효과가 있으며, 위 코드에서는 이중 배열의 row와 col을 바꾸어 뒤바꿨다. 마찬가지로 여러 번 실행 시 대부분 후자가 실행 시간이 더 적게 나왔다. 그리고 release 모드일 때는 전체 연산 시간과 두 시간의 차이 둘 다 줄어들었다.

(4) loop splitting



하나의 loop를 다수의 loop로 쪼개는 기법으로, 실행 시 대체로 역시 후자의 실행 시간이 더 짧았으나 위 결과에서처럼 가끔씩 차이가 없게 나온다. 그리고 release 모드에서는 너무 시간이 줄어든 나머지 둘 다 0으로 측정되어 차이를 확인할 수 없었다.

(5) loop unswitching



loop 안의 조건을 (위 코드에서는 ‘flag % 2 == 0’) 바깥으로 옮기는 기법으로 loop의 바디가 단순화되어 총 조건문 검사의 수를 줄이는 효과가 있다. 역시 후자가 대체적으로 실행 시 시간이 더 적게 걸렸으며 release 모드일 때는 전체 연산 시간과 두 시간의 차이 둘 다 줄어들었다.

전체적으로 debug 모드일 때보다 release 모드일 때 실행 시간이 훨씬 줄었는데, 전자의 경우 실행 파일에 디버깅 정보를 포함하고 있기 때문에 더 큰 메모리를 사용하며 디버그에 필요한 정보를 실행 시 매번 체크하기 때문에 속도가 비교적 느리다. 그리고 후자의 경우 코드를 최적화하여 실행 파일 크기를 줄이기 때문에 이렇게 비교적 빠른 실행 시간을 보여준다.