Хранение int > 0

```cpp
#include <iostream>
int x = 256;
int num = 9;
int binx[32];
int main()
{
    for(int i = 0; i < 32; i++)
    {
        asm(
                "movl x(%rip), %r8d \n"
                "rcl $1, %r8d \n"
                "pushf \n"
                "popq %rax \n"
                "andq $1, %rax \n"
                "movl %eax, num(%rip) \n"
                "movl %r8d, x(%rip) \n"
                );
        binx[i] = num;
    }
    for(int i = 0; i < 32; i++)
        std::cout<<binx[i]<<" ";
    return 0;
}
```

```
D:\AssemblerC++2021\cmake-build-debug\AssemblerC__2021.exe
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
Process finished with exit code 0
```

Хранение int < 0

```cpp
#include <iostream>
float x = -378453;
int num = 9;
int binx[32];
int main()
{
    for(int i = 0; i < 32; i++)
    {
        asm(
                "movl x(%rip), %r8d \n"
                "rcl $1, %r8d \n"
                "pushf \n"
                "popq %rax \n"
                "andq $1, %rax \n"
                "movl %eax, num(%rip) \n"
                "movl %r8d, x(%rip) \n"
                );
        binx[i] = num;
    }
    for(int i = 0; i < 32; i++)
        std::cout<<binx[i]<<" ";
    return 0;
}
```

```
D:\AssemblerC++2021\cmake-build-debug\AssemblerC__2021.exe
1 1 0 0 1 0 0 0 1 0 1 1 1 0 0 0 1 1 0 0 1 0 1 0 1 0 1 0 0 0 0 0
Process finished with exit code 0
```

unsigned int выглядит также как int>0

Просто для примера:

float x = 1245.563345 вид: 0 1 0 0 0 1 0 0 1 0 0 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1

±0 выглядит как 32 нуля независимо от знака(как и должно быть)

+∞ имеет вид: 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

-∞ имеет вид 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

NaN имеет вид: 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Значения получены с помощью двоичного представления результатов следующих операций: 1.1/0. -1.1/0. 1.1/0. - 1.1/0

Переполнение мантиссы:

float x = 0,2 вид: 0 0 1 1 1 1 1 0 0 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1

Неассоциативность арифметических операций (пример):

```
float a = pow( _X: 10,  _Y: 10);
x  = (a+1) - a;
y = (a - a) + 1;
std::cout<<x<<"\n"<<y<<"\n";
```

```
0.00000
1.00000
```

В первом случае 10^{10}+1 округлилось до 10^{10}, а во втором все понятно. Кстати без скобочек все работает также, потому что в любом случае они выполняются слева направо.

Арифметические операции с float и double.

```
#include <stdio.h>
float x=1.2, y=3.4, z=5.6;
int main()
{
    x = y + z;
    y = x - y;
    z = x / y;
    z = x * y;
}
```

```
.cfi_startproc
endbr64
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq    %rsp, %rbp
.cfi_def_cfa_register 6
movss   y(%rip), %xmm1
movss   z(%rip), %xmm0
addss   %xmm1, %xmm0
movss   %xmm0, x(%rip)
movss   x(%rip), %xmm0
movss   y(%rip), %xmm1
subss   %xmm1, %xmm0
movss   %xmm0, y(%rip)
movss   x(%rip), %xmm0
movss   y(%rip), %xmm1
divss   %xmm1, %xmm0
movss   %xmm0, z(%rip)
movss   x(%rip), %xmm1
movss   y(%rip), %xmm0
mulss   %xmm1, %xmm0
movss   %xmm0, z(%rip)
movl    $0, %eax
popq    %rbp
.cfi_def_cfa 7, 8
ret
```

Особенности:

Используется тип ss(Scalar-Single) для float sd(Scalar-Double) для double соответственно, отдельные регистры %xmm0...%xmm7

Объявление переменных различных типов

int

```
x:
        .long   1
        .globl  y
        .align 4
        .type   y, @object
        .size   y, 4
```

float

```
x:
        .long   1067030938
        .globl  y
        .align 4
        .type   y, @object
        .size   y, 4
```

double

```
x:
        .long   858993459
        .long   1072902963
        .globl  y
        .align 8
        .type   y, @object
        .size   y, 8
```

Графики числа π от числа итераций

Код

```
float pi1=0;
float pi2=1;
float pi3=0;
float pi4=3;
outfile << fixed;
outfile.precision( prec: 20);
for(int i=0; i<N; i++)
{
    if(i % 2 == 0) //ЧЕТНЫЕ
    {
        pi1 += 1/(2*float(i)+1);
        pi2 *= (float(i)+2)/(float(i)+1);
        pi4 += 4/(2*float(i)+2)/(2*float(i)+3)/(2*float(i)+4);
    }
    else //НЕЧЕНТНЫЕ
    {
        pi1 -= 1/(2*float(i)+1);
        pi2 *= (float(i)+1)/(float(i)+2);
        pi4 -= 4/(2*float(i)+2)/(2*float(i)+3)/(2*float(i)+4);
    }

    pi3 += pow( x: -1, i)/(pow( x: 3, i)*(2*float(i)+1));

    outfile<<i<<" "<<pi1*4<<" "<<pi2*2<<" "<<""<<pi3*2*sqrt( x: 3)<<" "<<pi4<<"\n";
```

Количество элементов N =1000000, используемые формулы указаны ниже. Ось X прологарифмированная.

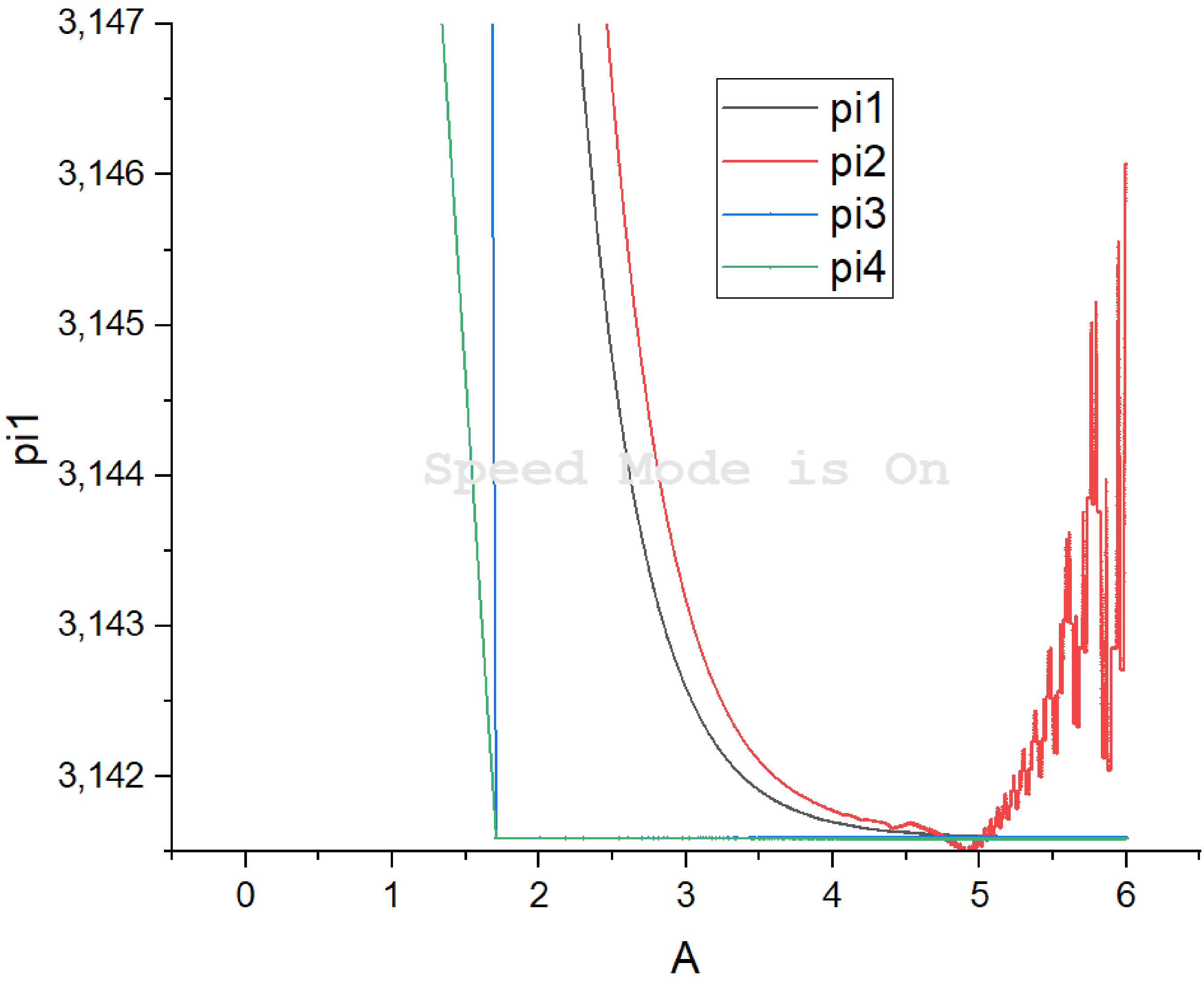
$$\frac{1}{1}-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\cdots=\frac{\pi}{4}$$ — pi1

$$\frac{\pi}{2}=\frac{2}{1}\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdot\frac{8}{7}\cdot\frac{8}{9}\cdots$$ — pi2

$$\frac{2}{\pi}=\frac{\sqrt{2}}{2}\cdot\frac{\sqrt{2+\sqrt{2}}}{2}\cdot\frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2}$$ — pi3

$$\pi=2\sqrt{3}\sum_{k=0}^{\infty}\frac{(-1)^k}{3^k\,(2k+1)}$$ pi4

*π = 3 + 4/(2*3*4) - 4/(4*5*6) + 4/(6*7*8) - 4/(8*9*10) + 4/(10*11*12) - (4/(12*13*14))*



Ну по графику понятно, что самые точные это формулы pi3 и pi4. Pi2 вообще дает примерно похожие показания на очень узком интервале. Pi1 увеличивает точность с ростом итераций.

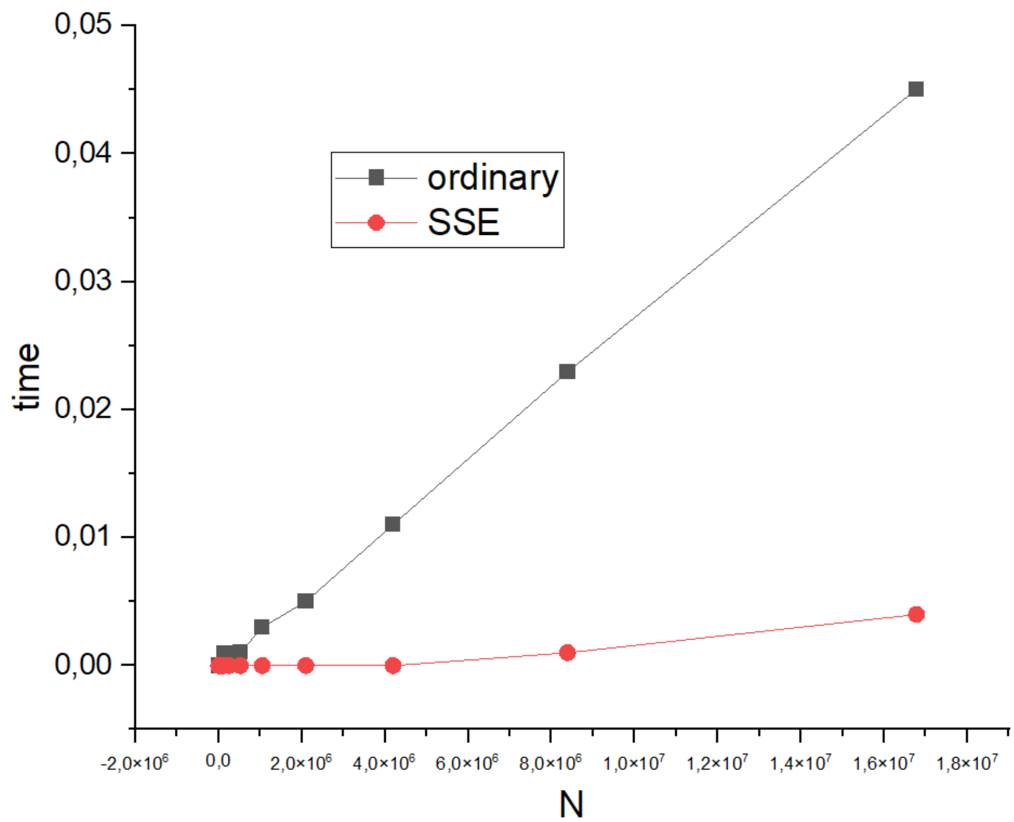
Среднее арифметическое с SSE и без

(Объяснение после картинок)

```cpp
random_device rd;
mt19937 mt(time(0)); //rd()
uniform_real_distribution<float> dist(0.0, 1000.0);
for(int i=0; i<n; ++i)
    arr[i] = dist(mt);

auto start = chrono::high_resolution_clock::now();
//Сумма элементов
//for(int i=0; i<n; ++i)
//    sum+=arr[i];
for(int i=0; i<n/4-1; i+=4)
{
    arr[i] = tmp[0];
    arr[i+1] = tmp[1];
    arr[i+2] = tmp[2];
    arr[i+3] = tmp[3];
    asm(
        "movups tmp, %xmm0 \n"
        "movups suma, %xmm1 \n"
        "addps %xmm0, %xmm1 \n"
        "movups %xmm1, suma \n"
        );
}
sum = suma[0]+suma[1]+suma[2]+suma[3];
sum = sum / float(n);
auto end = chrono::high_resolution_clock::now();

chrono::duration<double> diff = end - start;
//outfile << n << "            " << diff.count() << endl;
cout << diff.count() << endl;
delete [] arr;
```



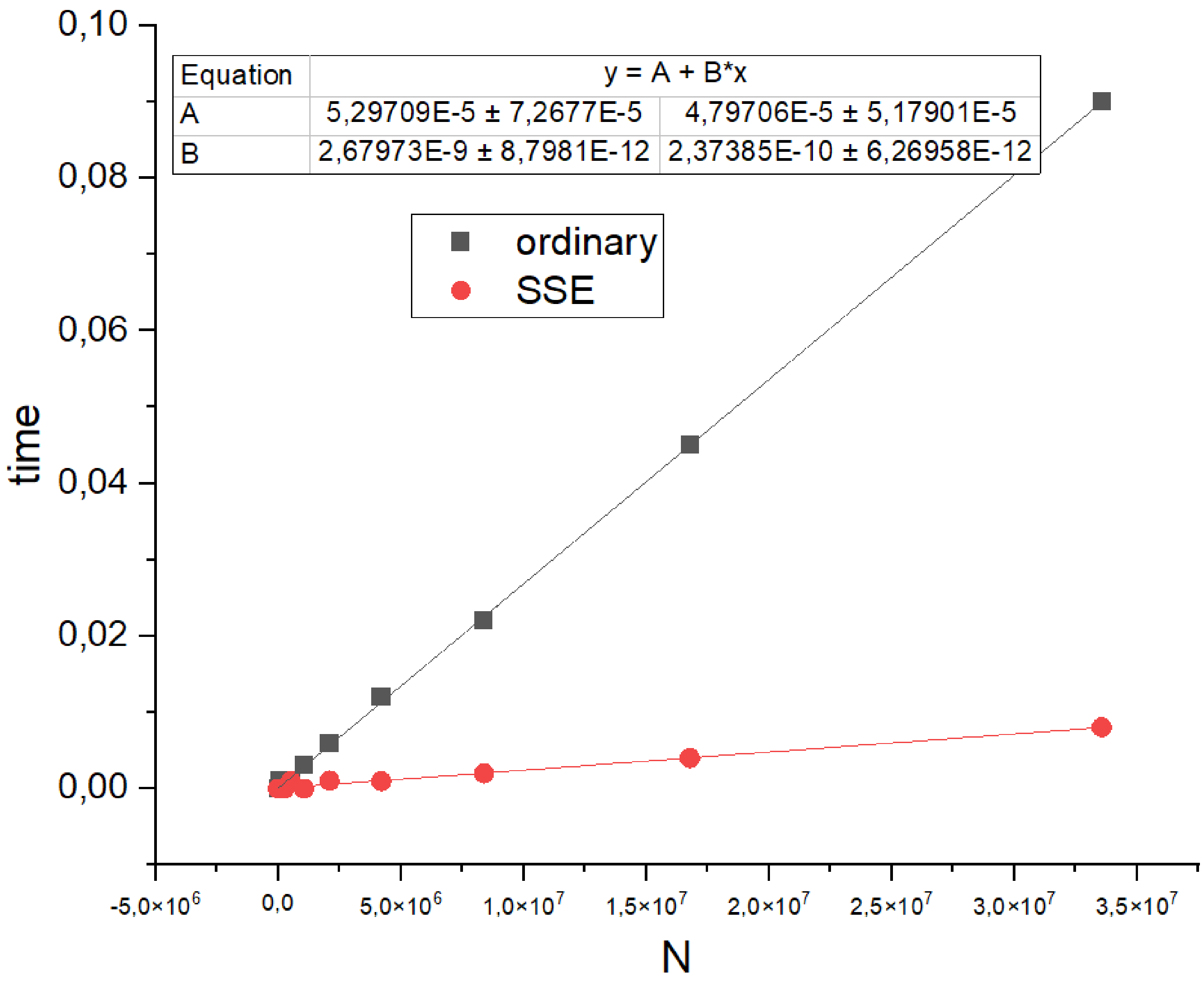
Размеры массива меняются от 16 до 16777216, умножаясь на 2.

Массив заполняется случайными числами.

ordinary алгоритм: проходим весь массив поэлементно и складываем все элементы по одному в переменную sum, потом делим на кол-во элементов.

SSE алгоритм: проходимся по массиву складываем каждые 4 числа в массив suma с помощью SSE, 4 значения в полученном массиве складываем и делим на кол-во элементов

Вот на картинке ниже сделал еще на 33000000 элементов массив, и аппроксимировал графики, вообщем углы наклона различаются ≈ в 9 раз.



Вот еще оставлю GitHub на всякий случай:

https://github.com/ArtemEvstafev/AssemblerC-2021.git