Lab 5

1. int

```c
#include <stdio.h>

//unsigned int a = 4294967295;
int a = -2147483648;
int b[32];
int c;

void bit()
{
        printf("%d \n", a);
        for(int i = 0; i < 32; i ++)
        {
                asm(
                "movq $1, %rax \n"
                "andq a(%rip), %rax \n"
                "movq %rax, c(%rip) \n"
                "shrq $1, a(%rip) \n"
                );
                b[31 - i] = c;
        }
        for(int i = 0; i < 32; i ++)
        {
                printf("%d", b[i]);
        }
}
```

Examples

Number	Two’s complement
-2147483648	10000000000000000000000000000000
-2147483646	10000000000000000000000000000010
-3	11111111111111111111111111111101
2147483648(int)	10000000000000000000000000000000 (equals to -2147483648)
4294967295(unsigned int)	11111111111111111111111111111111
2147483648(unsigned int)	10000000000000000000000000000000

2. float

```c
#include <stdio.h>

float a = 1.5;
//double a = 1.5;
int c;
int b[sizeof(a) * 8];

void bit()
{
        printf("%f \n", a);
        for(int i = 0; i < sizeof(a) * 8; i ++)
        {
                asm(
                "movq $1, %rax \n"
                "andq a(%rip), %rax \n"
                "movq %rax, c(%rip) \n"
                "shrq $1, a(%rip) \n"
                );
                b[sizeof(a) * 8 - i - 1] = c;
        }
        for(int i = 0; i < sizeof(a) * 8; i ++)
        {
                if(sizeof(a) == 4 && (i == 0 || i == 8)) printf("%d ", b[i]);
                else if(sizeof(a) == 8 && (i == 0 || i == 11)) printf("%d ", b[i]);
                else printf("%d", b[i]);
        }
}
```

Examples

Number	Two’s complement
1.5(float)	0 01111111 10000000000000000000000
1.5(double)	0 01111111111 1000000000000000000000000000000000000000 00000000000000
3.14(float)	0 10000000 10010001111010111000011 (=3.140001049041748046875)
+(-)0 (float)	0(1) 00000000 00000000000000000000000
+(-)∞	0(1) 11111111 00000000000000000000000
NaN	0 11111111 10000000000000000000000

3. Mantissa overflow

```
float d = 1073741824;
d += 1.0;
printf("%f", d);
```

output: 1073741824.0

4. Non-associativity

```
a*(b-c)!=a*b-a*c: -0.1299999803 != -0.1299999654
a/e*b*c)!=a*b*c/e: 0.0519999973 != 0.0520000011
```

5. Assembly listing

```
float a = 3.1;
double b = 3.4;

int main()
{
        double c = b + 1.2;
        float d = a + 4.3;
}
```

```
a:
        .long   1078355558
        .globl  b
        .align 8
        .type   b, @object
        .size   b, 8
b:
        .long   858993459
        .long   -1073007821
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movsd   b(%rip), %xmm1
        movsd   .LC0(%rip), %xmm0
        addsd   %xmm1, %xmm0
        movsd   %xmm0, -8(%rbp)
        movss   a(%rip), %xmm0
        cvtss2sd        %xmm0, %xmm1
        movsd   .LC1(%rip), %xmm0
        addsd   %xmm1, %xmm0
        cvtsd2ss        %xmm0, %xmm0
        movss   %xmm0, -12(%rbp)
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .section        .rodata
        .align 8
.LC0:
        .long   858993459
        .long   1072902963
        .align 8
.LC1:
        .long   858993459
        .long   1074869043
```

- movsd - moves scalar double-precision floating-point value(b to %xmm1(first 64 bits are rewritten))
- movss – same as movsd, but performs with float values
- addsd – adds two double values
- cvtss2sd - converts one single-precision floating-point value to one double-precision floating-point value(%xmm0 to %xmm1 - first 32 bits of %xmm0 are rewritten as double in first 64 bits of %xmm1)

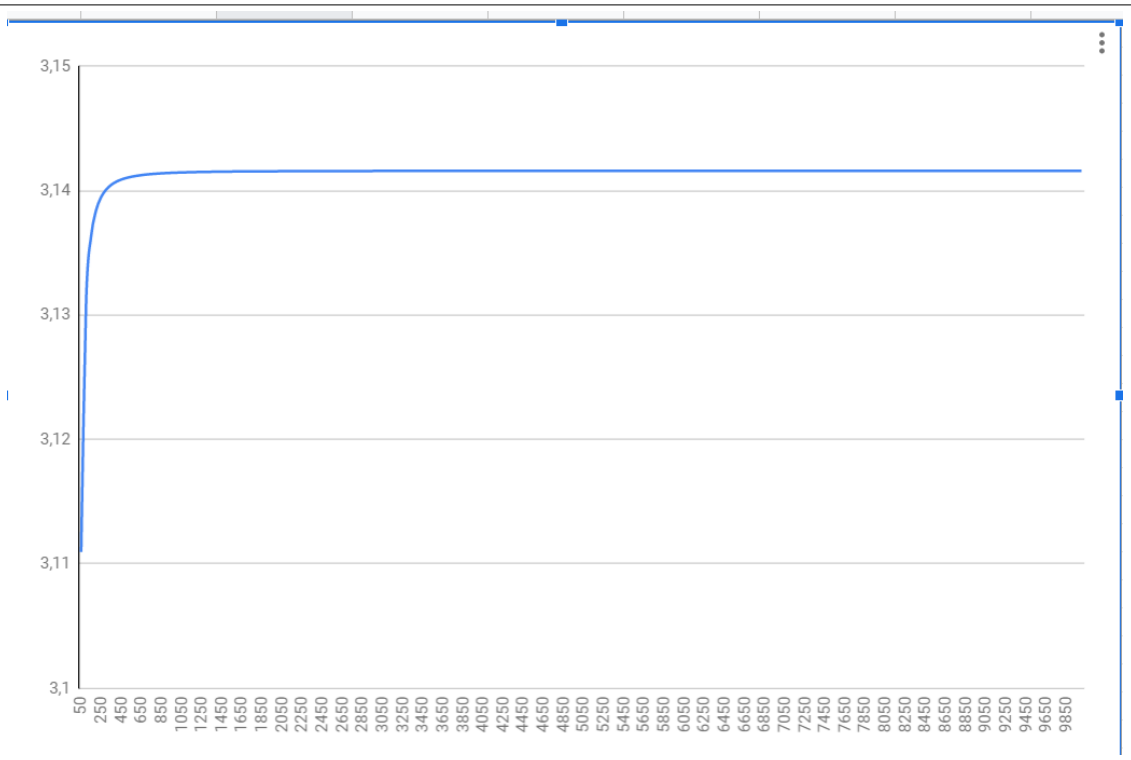
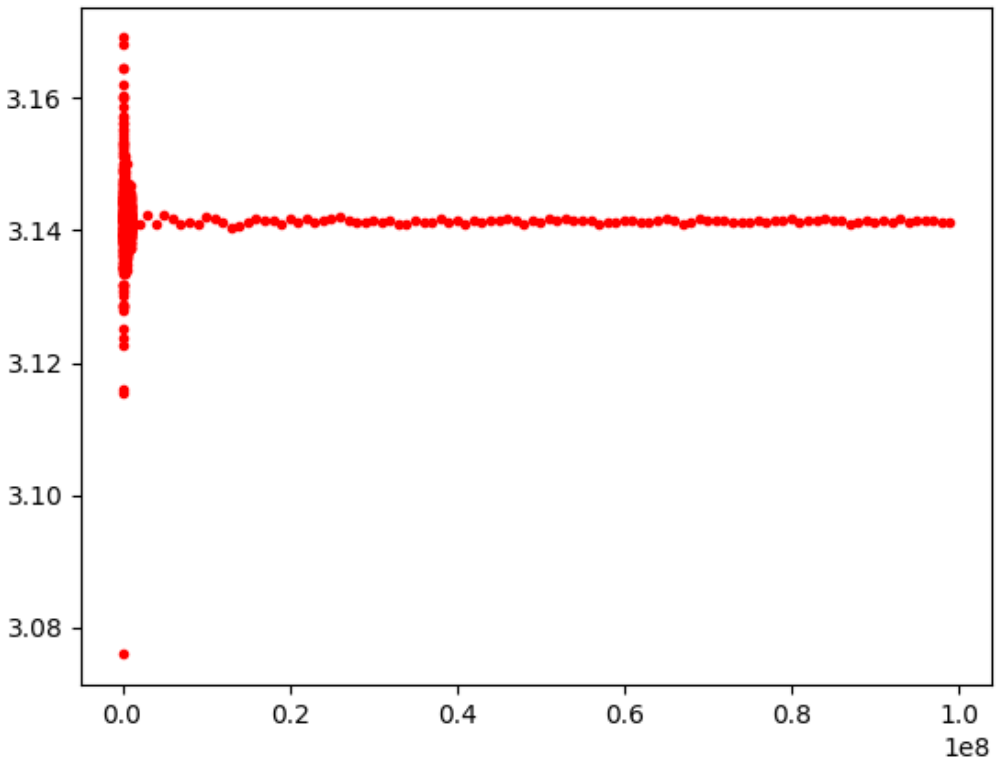
Interestingly, float and double numbers are written as huge numbers, e.g. float a is written as 10783555588. That is because floating-point number representation(in binary) is converted to decimal. As for doubles, it’s value is divided in 2 rows: first row is for last 32 bits of double(mantissa), second row is for first 32 bits(sign + exp + mantissa)

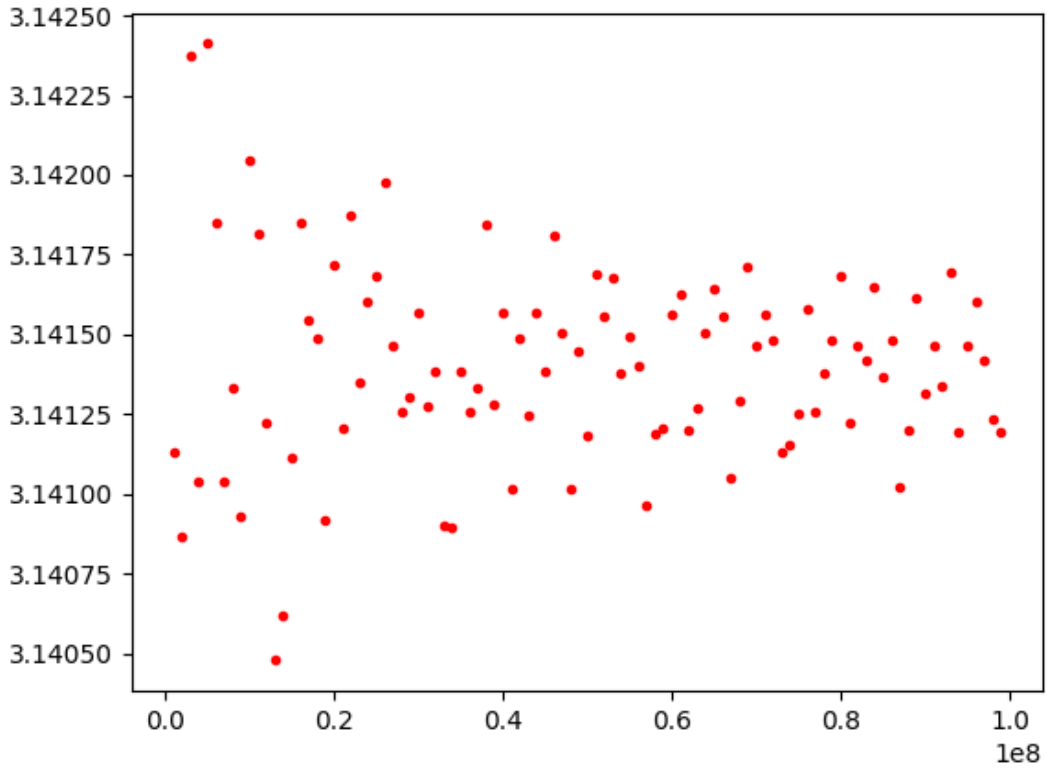
```c
#include <stdio.h>

float a = 3.1;
double b = -3.4;

int main()
{
        double c = b * 1.2;
        float d = a * 4.3f;
}
```

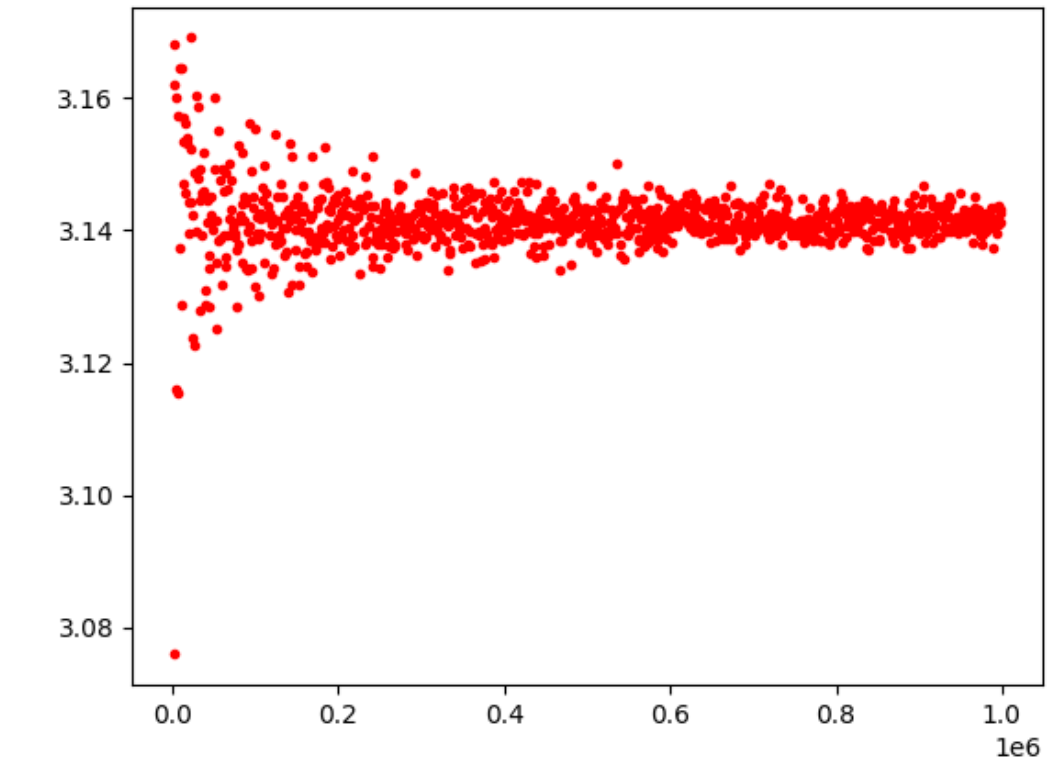
```
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6,-16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movsd   b(%rip), %xmm1
        movsd   .LC0(%rip), %xmm0
        mulsd   %xmm1, %xmm0
        movsd   %xmm0, -8(%rbp)
        movss   a(%rip), %xmm1
        movss   .LC1(%rip), %xmm0
        mulss   %xmm1, %xmm0
        movss   %xmm0, -12(%rbp)
        movl    $0, %eax
        popq    %rbp
```

- mulsd – multiplies 2 double values
- mulss – multiplies 2 float values

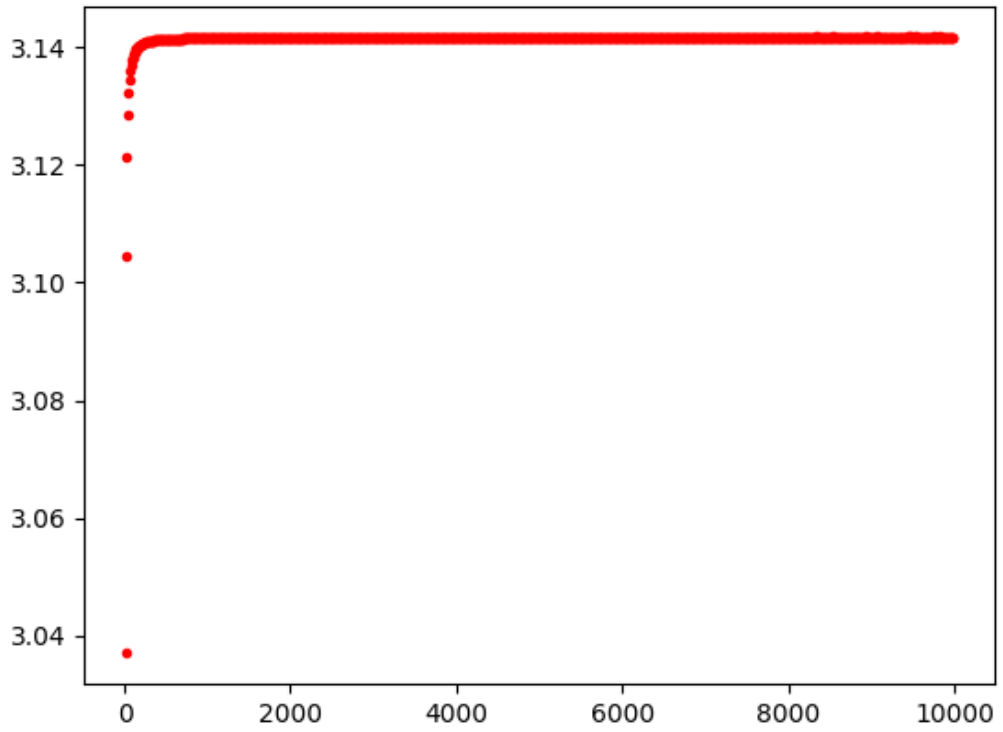
6. π(N)

Method	Plot
Wallis product	
Monte-Carlo method	

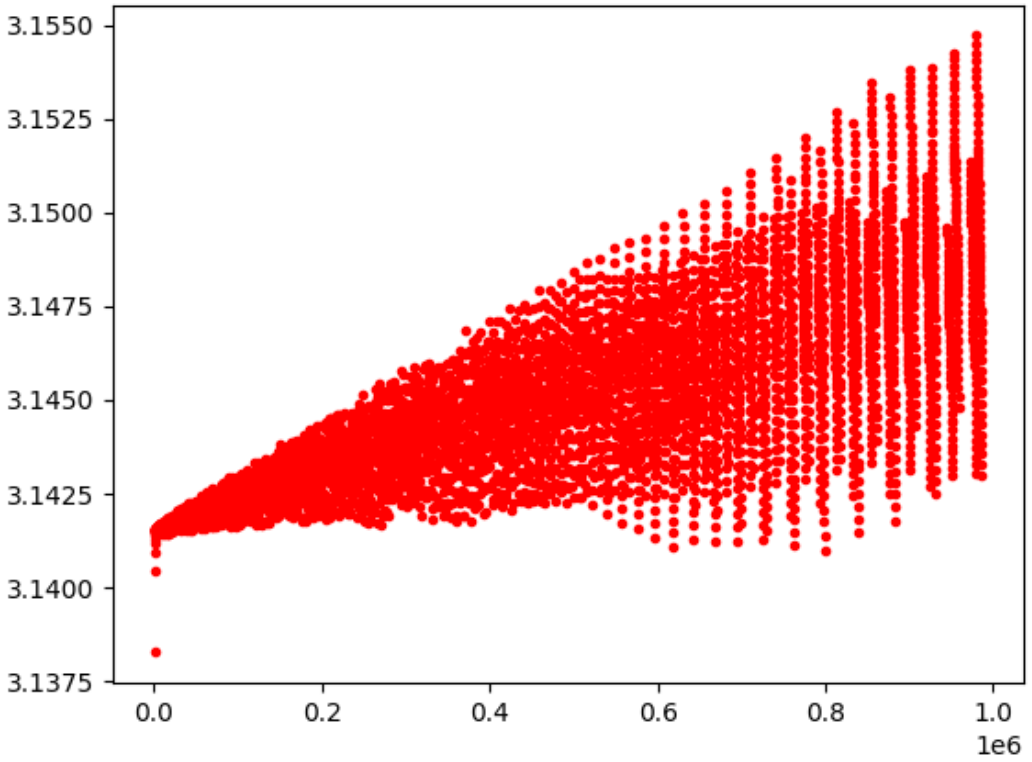

3.14250
3.14225
3.14200
3.14175
3.14150
3.14125
3.14100
3.14075
3.14050
0.0
0.2
0.4
0.6
0.8
1.0
1e8


3.16
3.14
3.12
3.10
3.08
0.0
0.2
0.4
0.6
0.8
1.0
1e6

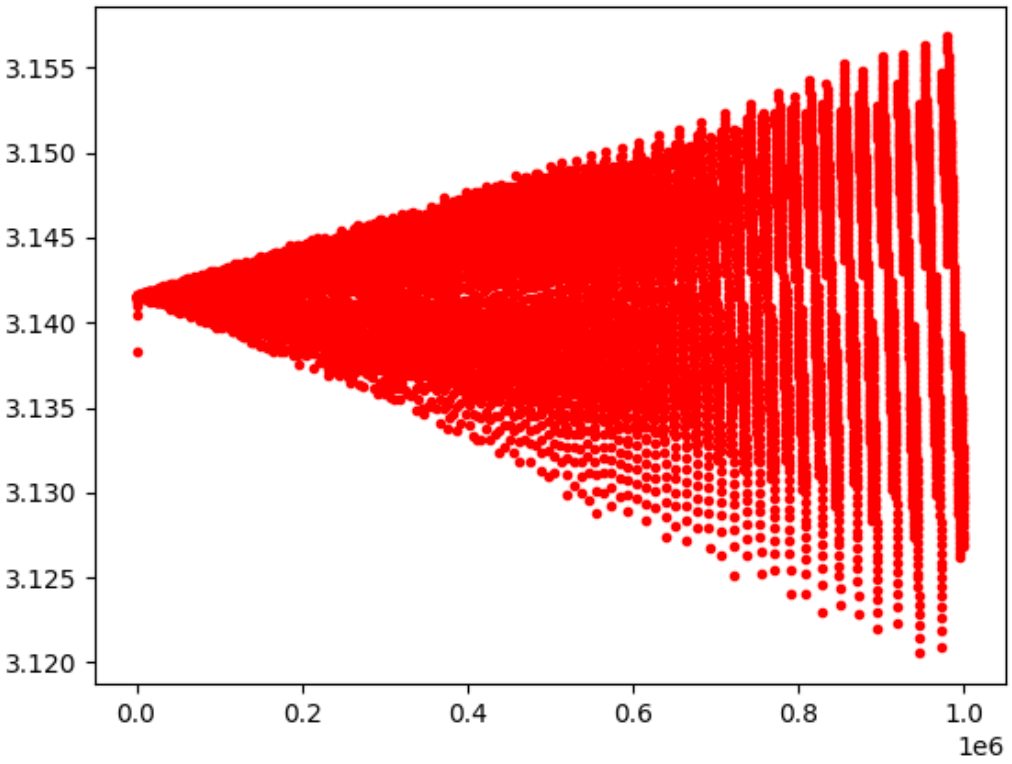
$$\int_{-1}^{1} \sqrt{1-x^2}$$



dx > $2 * 10^{-4}$



dx > $2 * 10^{-4}$



dx < 2 * 10^{-4} (more strict integration boundaries)

7. Denormalized numbers

```
#include <iostream>

#include <xmmintrin.h>

int main()
{
        //_mm_setcsr(_mm_getcsr() | 0x8040);
        float a = 1;
        int i = 0;
        while(a > 0)
        {
                a /= 2;
                i += 1;
                std::cout << a << "\n";
        }
        printf("%d", i);
}
```

	float	double
Normalized	1.17549e-38 0 127sonamkrlm@s	2.22507e-308 0 1023sonamkrlm
Denormalized	1.4013e-45 0 150sonamkrl	4.94066e-324 0 1075sonamkrl

First value is the minimum floating-point number; third value is the abs of maximum power of 2, which represents 0.

8. FTZ and DAZ

FTZ - sets denormal results from floating-point calculations to zero.

DAZ - sreats denormal values used as input to floating-point instructions as zero.