Biostatistics 615 - Statistical Computing

Lecture 3 Floating-point representation & Errors

Jian Kang

September 15, 2015

Summary of the previous lecture

- Binary representation of a nonnegative integer in C++
- Basic syntax
  - Comment statement
  - Header files (iostream)
  - Statement "using"
  - The main function (return type and return statement)
  - Output/input operators (`cin, cout, << , >>`)
  - Prototype / declaration of functions and variables
  - The "for" loop and statement "if ... else ... "
  - Bit-wise operators (shift and AND)
- Compile and run using GNU C++ complier
- Environment for homework
- Homework 0
- Resources

Binary representation of a signed integer

- *Signed magnitude*
  - Consider a one-byte integer: $12 = (00001100)_2$ and $-12 = (10001100)_2$.
  - Only 7 working digits and the red digit is *signed digit*.
  - Ranging from $-127 = -(2^7 - 1)$ to $127 = 2^7 - 1$.
  - How about 0?

Binary representation of a signed integer

- *Signed magnitude*
  - Consider a one-byte integer: $12 = (00001100)_2$ and $-12 = (10001100)_2$.
  - Only 7 working digits and the red digit is *signed digit*.
  - Ranging from $-127 = -(2^7 - 1)$ to $127 = 2^7 - 1$.
  - How about 0?
- *Two's complement rule*
  - The left-most bit is allowed to represent -2^{m-1} in an m bit storage allocation
  - It represents all the negative integers between -2^{m-1} and -1

$$-a = -2^{m-1} + \sum_{k=0}^{m-2} b_k 2^{m-2-k}, \qquad a = (1b_0 \dots b_{m-2})_2$$

  - $-12 = -128 + 64 + 32 + 16 + 4 = (11110100)_2$
  - Ranging from -2^{m-1} to $2^{m-1} - 1$

Floating point representation

- Computer cannot precisely store an irrational number. Why?
- There is a limit to the *precision* (the number of significant digits) that can be achieved.

Floating point representation

- Computer cannot precisely store an irrational number. Why?
- There is a limit to the *precision* (the number of significant digits) that can be achieved.
- A real number x is represented in base 10

$$x = \pm \sum_{j=0}^{\infty} a_j 10^{p-j}, \qquad x \approx \pm a_0.a_1 \ldots a_m 10^p.$$

where $a_j \in \{0, 1, \ldots, 9\}$.

Floating point representation

- Computer cannot precisely store an irrational number. Why?
- There is a limit to the *precision* (the number of significant digits) that can be achieved.
- A real number x is represented in base 10

$$x = \pm \sum_{j=0}^{\infty} a_j 10^{p-j}, \qquad x \approx \pm a_0.a_1 \ldots a_m 10^p.$$

where $a_j \in \{0, 1, \ldots, 9\}$.

- A real number x is represented in base 2 in

$$x = \pm \sum_{j=0}^{\infty} b_j 2^{p-j}, \qquad x \approx \underbrace{\pm}_{\text{Sign}} \underbrace{b_0.b_1 \ldots b_m}_{\text{Significand}} \underbrace{2^p}_{\text{Exponent}} .$$

where $b_j \in \{0, 1\}$ for $j = 1, \ldots, m$ and $b_0 = 1$.

Floating point representation – Single precision

- `float` type needs ______ bytes of storage
- A total of ______ bits of storage including
  - ______ for sign
  - ______ for exponent
  - ______ for significand

Floating point representation – Single precision

- `float` type needs 4 bytes of storage
- A total of 32 bits of storage including
  - 1 for sign
  - 8 for exponent
  - 23 for significand
- IEEE 754 Standard

$$x \approx (-1)^{d_1}\left(1+\sum_{j=1}^{23} d_{j+9}2^{-j}\right)\times 2^{\left(\sum_{j=1}^{8} d_{j+1}2^{8-j}-127\right)}$$

where $d_j \in \{0,1\}$

$$x \approx (\underbrace{d_1}_{\text{Sign}}\ \underbrace{d_2\ldots d_9}_{\text{Exponent}}\ \underbrace{d_{10}\ldots d_{32}}_{\text{Significand}})_2$$

Single precision – Example

Convert -4.5 into the binary representation in single precision

$$-4.5 = -(2^2 + 2^{-1}) = (-1)^{(1)} \times (1 + 2^{(-3)}) \times 2^2$$

Single precision – Example

Convert -4.5 into the binary representation in single precision

$$-4.5 = -(2^2 + 2^{-1}) = (-1)^{(1)} \times (1 + 2^{(-3)}) \times 2^2$$

- Sign: $d_1 = 1$
- Exponent: $(127 + 2) = (10000001)_2 = (d_2 \dots d_9)_2$
- Significand: $2^{-3} = (00100000000000000000000)_2 = (d_{10} \dots d_{32})_2$
- $-4.5 = (11000000100100000000000000000000)_2$

Single precision – Example

Convert -4.5 into the binary representation in single precision

$$-4.5 = -(2^2 + 2^{-1}) = (-1)^{(1)} \times (1 + 2^{(-3)}) \times 2^2$$

- Sign: $d_1 = 1$
- Exponent: $(127 + 2) = (10000001)_2 = (d_2 \dots d_9)_2$
- Significand: $2^{-3} = (00100000000000000000000)_2 = (d_{10} \dots d_{32})_2$
- $-4.5 = (11000000100100000000000000000000)_2$

More examples:

$$\begin{aligned} 2015.915 &\approx (01000100111110111111110101001000)_2 \\ &= 2015.9150390625 \\ 2015.91509 &\approx \; ? \end{aligned}$$

Floating point representation – Double precision

- `double` type needs ______ bytes of storage
- A total of ______ bits of storage including
  - ______ for sign
  - ______ for exponent
  - ______ for significand

Floating point representation – Double precision

- `double` type needs 8 bytes of storage
- A total of 64 bits of storage including
  - 1 for sign
  - 11 for exponent
  - 52 for significand
- IEEE 754 Standard

$$x = (-1)^{d_1}\left(1 + \sum_{j=1}^{52} d_{j+12} 2^{-j}\right) \times 2^{\left(\sum_{j=1}^{11} d_{j+1} 2^{11-j} - 1023\right)}$$

where $d_j \in \{0, 1\}$

$$x = (\underbrace{d_1}_{\text{Sign}}\underbrace{d_2 \ldots d_{12}}_{\text{Exponent}}\underbrace{d_{13} \ldots d_{64}}_{\text{Significand}})_2$$

Double precision – Example

Convert 2015.915 into the binary representation in double precision

Double precision – Example

Convert 2015.915 into the binary representation in double precision

- Sign: $d_1 = 0$
- Exponent: $(1023 + 10) = (10000001001)_2 = (d_2 d_3 \dots d_{12})_2$
- Significand:

$$
\begin{aligned}
& (2015.915/2^{10} - 1) \\
\approx \quad & (1111011111111010100011110101110000101000111101011100)_2 \\
= \quad & (d_{13} \dots d_{64})_2
\end{aligned}
$$

Double precision – Example

Convert 2015.915 into the binary representation in double precision

- Sign: $d_1 = 0$
- Exponent: $(1023 + 10) = (10000001001)_2 = (d_2 d_3 \ldots d_{12})_2$
- Significand:

$$
\begin{aligned}
& (2015.915/2^{10} - 1) \\
\approx\ & (1111011111111010100011110101110000101000111101011100)_2 \\
=\ & (d_{13} \ldots d_{64})_2
\end{aligned}
$$

$$
\begin{aligned}
2015.915 \approx\ & (0100000010011 \ldots 1000111101011100)_2 \\
=\ & 2015.9149999999999636202119290 8
\end{aligned}
$$

Relative errors – Upper Bound

Suppose that x is a positive real number that can be written as

$$x = \sum_{j=0}^{\infty} b_j 2^{p-j}$$

with $b_0 = 1$. Then *round-to-closest* x by the approximation

$$\tilde{x} = 2^p \left[\sum_{j=0}^{m} b_j 2^{-j} + \tilde{b}_{m+1} 2^{-(m+1)} \right]$$

更低位更接近

进位

for some integer m, where $\tilde{b}_{m+1} = 2$ if $b_{m+1} = 1$ and $\tilde{b}_{m+1} = 0$. It can be shown that

$$|x - \tilde{x}| \le 2^{p-m-1}$$

若~bm+1=1则必定不是最接近的

Thus,

$$\mathrm{Rel}(\tilde{x}) = \frac{|x - \tilde{x}|}{|x|} \le 2^{-(m+1)}$$

which are referred as the *machine epsilon*.

Relative errors – Example

- For single precision (float), $m = 23$, the relative error is

$$2^{-23-1} \approx 0.6 \times 10^{-7}$$

  There are 7 valid digits for single precision arithmetic

- For double precision (double), $m = 52$, the relative error is

$$2^{-52-1} \approx 0.1 \times 10^{-15}$$

  There are 15 valid digits for double precision arithmetic

A summary

Type	Size in bits	Format	Value range: Approximate	Value range: Exact
character	8	signed (one's complement)		**-127** to **127**[note 1]
		signed (two's complement)		**-128** to **127**
		unsigned		**0** to **255**
integral	16	signed (one's complement)	**± 3.27 · 10^4**	**-32767** to **32767**
		signed (two's complement)		**-32768** to **32767**
		unsigned	**0** to **6.55 · 10^4**	**0** to **65535**
	32	signed (one's complement)	**± 2.14 · 10^9**	**-2,147,483,647** to **2,147,483,647**
		signed (two's complement)		**-2,147,483,648** to **2,147,483,647**
		unsigned	**0** to **4.29 · 10^9**	**0** to **4,294,967,295**
	64	signed (one's complement)	**± 9.22 · 10^{18}**	**-9,223,372,036,854,775,807** to **9,223,372,036,854,775,807**
		signed (two's complement)		**-9,223,372,036,854,775,808** to **9,223,372,036,854,775,807**
		unsigned	**0** to **1.84 · 10^{19}**	**0** to **18,446,744,073,709,551,615**
floating point	32	IEEE-754	**± 3.4 · $10^{\pm 38}$** (~7 digits)	• min subnormal: **± 1.401,298,4 · 10^{-47}** • min normal: **± 1.175,494,3 · 10^{-38}** • max: **± 3.402,823,4 · 10^{38}**
	64	IEEE-754	**± 1.7 · $10^{\pm 308}$** (~15 digits)	• min subnormal: **± 4.940,656,458,412 · 10^{-324}** • min normal: **± 2.225,073,858,507,201,4 · 10^{-308}** • max: **± 1.797,693,134,862,315,7 · 10^{308}**

Errors on basic arithmetic operations

Let x, y be the true values and $\tilde{x}, \tilde{y}$ be the approximate values (machine representation). Recall that

$$\mathrm{Rel}(\tilde{x}) = \frac{x - \tilde{x}}{x}, \qquad \mathrm{Rel}(\tilde{y}) = \frac{y - \tilde{y}}{y},$$

Then

$$\begin{aligned}
\mathrm{Rel}(\tilde{x} \times \tilde{y}) &= \mathrm{Rel}(\tilde{x}) + \mathrm{Rel}(\tilde{y}) - \mathrm{Rel}(\tilde{x}\tilde{y}) \\
&\approx \mathrm{Rel}(\tilde{x}) + \mathrm{Rel}(\tilde{y}) \\
\mathrm{Rel}(\tilde{x}/\tilde{y}) &= \frac{\mathrm{Rel}(\tilde{x}) - \mathrm{Rel}(\tilde{y})}{1 - \mathrm{Rel}(\tilde{y})} \\
&\approx \mathrm{Rel}(\tilde{x}) - \mathrm{Rel}(\tilde{y}) \\
\mathrm{Rel}(\tilde{x} + \tilde{y}) &= \mathrm{Rel}(\tilde{x}) \frac{x}{x+y} + \mathrm{Rel}(\tilde{y}) \frac{y}{x+y} \\
\mathrm{Rel}(\tilde{x} - \tilde{y}) &= \mathrm{Rel}(\tilde{x}) \frac{x}{x+y} - \mathrm{Rel}(\tilde{y}) \frac{y}{x+y}
\end{aligned}$$

Relative errors on summation

Let $\tilde{x}_1, \ldots, \tilde{x}_n$ be floating-point approximations of numbers $x_1, \ldots, x_n$. Then the relative errors of $\widetilde{S}_k = \sum_{j=1}^{k} \tilde{x}_j$, for $k = 2, \ldots, n$, is given by

$$\begin{aligned}\mathrm{Rel}(\widetilde{S}_k) &= \mathrm{Rel}(\widetilde{S}_{k-1} + \tilde{x}_k) \\ &= \frac{1}{S_k}\left[\mathrm{Rel}(\widetilde{S}_{k-1})S_{k-1} + \mathrm{Rel}(\tilde{x}_k)x_k\right]\end{aligned}$$

This suggests that

- The large error will accumulate as multiples of the previous term that are entered into the summation
- The best strategy: addition should proceed with values being summed from the small magnitude to the large magnitude

Effects of order on addition: Example

Compute (direct order)

$$\sum_{j=1}^{n} \frac{1}{j^2} = \frac{1}{1^2} + \frac{1}{2^2} + \ldots + \frac{1}{n^2}$$

This is mathematically equivalent to (reverse order)

$$\sum_{j=1}^{n} \frac{1}{(n-j+1)^2} = \frac{1}{n^2} + \frac{1}{(n-1)^2} + \ldots + \frac{1}{1^2}$$

Example: Compute series

```
//Example 3.3: series.cpp
#include <cstdlib>
#include <iostream>
#include <iomanip>

using namespace std;

int main(int argc, char* argv[]){
    float sumL = 0.0, sumU = 0.0;
    int n = atoi(argv[1]);
    for (int j=1; j<=n; j++){
        sumL = sumL + 1/(float)(j*j);
        sumU = sumU + 1/(float)((n+1-j)*(n+1-j));
    }
    cout << setprecision(8) << "Direct and reverse sums are "
         << sumL << " and " << sumU << endl;
    return 0;
}
```

Useful header files

```
//Example 3.3: series.cpp
#include <cstdlib>
#include <iostream>
#include <iomanip>
 ...
}
```

- `cstdlib`: a collection of useful C++ functions including `atoi` in the example
- `iomanip`: output manipulators for formatting the printed output including `setprecision` in the example.

Basic syntax: Command line input arguments

```
//Example 3.3: series.cpp
...
int main(int argc, char* argv[]){
    float sumL = 0.0, sumU = 0.0;
    int n = atoi(argv[1]);
  ...
    return 0;
}
```

- `argc`: argument count
- `argv`: argument vector
- `char* argv[]`: an array of character strings (actually, a pointer to memory that holds arrays of `char` variables) with each string being a non-white space component of the white-space delimited text that was entered on the shell command line.
- The first array element strings `argv[0]` contains the name of the executable while `argc` is the number of strings held in `argv`.

Basic syntax: Convert string to integer

```
//Example 3.3: series.cpp
#include <cstdlib>
...
int main(int argc, char* argv[]){
    float sumL = 0.0, sumU = 0.0;
    int n = atoi(argv[1]);
  ...
    return 0;
}
```

- `atoi`: It is made available with the `cstdlib` header
- It transforms a string of character values into an integer variable
- The argument to `atoi` in this case is assumed to represent a sequence of digits that my be preceded by a sign
- If the string cannot be interpreted as a number, `atoi`, returns `0`.
- The function `atof` performs the same type of operation except that the transformation is to a floating-point representation

Basic syntax: Typecasting

```
//Example 3.3: series.cpp
...
int main(int argc, char* argv[]){
    float sumL = 0.0, sumU = 0.0;
    int n = atoi(argv[1]);
    for (int j=1; j<=n; j++){
        sumL = sumL + 1/(float)(j*j);
        sumU = sumU + 1/(float)((n+1-j)*(n+1-j));
    }
...
    return 0;
}
```

- Directly evaluate $1/(j*j)$ for every j exceeds 1. A value `0` will be returned
- `(float) j*j` computes the integer product and converts the outcome to type `float`.

Basic syntax: Set precision

```
//Example 3.3: series.cpp
...
#include <iomanip>
...
int main(int argc, char* argv[]){
    float sumL = 0.0, sumU = 0.0;
...
    cout << setprecision(8) << "Direct and reverse sums are "
         << sumL << " and " << sumU << endl;
    return 0;
...
}
```

- `setprecision(n)` to have `n` decimal numbers written to standard output
- Sets the decimal precision to be used to format floating-point values on output operations
- This manipulator is declared in the header `iomanip`

Output of the program

```
$ ./series 2015
Direct and reverse sums are 1.6444356 and 1.6444379

$ ./series 12015
Direct and reverse sums are 1.6447253 and 1.6448509
```

- When $n = 2015$, the two ways have the same answer up to 6 decimal digits
- When $n = 12015$, the difference appears in the 5th digits.
- Which one is closer to the true value?
- The computation can be redone in double precision: i.e. every `float` designation in series.cpp is replaced with `double` and it saves as dseries.cpp.

Compute the sum in double precision

```
//Example 3.4: dseries.cpp
#include <cstdlib>
#include <iostream>
#include <iomanip>

using namespace std;

int main(int argc, char* argv[]){
    double sumL = 0.0, sumU = 0.0;
    int n = atoi(argv[1]);
    for (int j=1; j<=n; j++){
        sumL = sumL + 1/(double)(j*j);
        sumU = sumU + 1/(double)((n+1-j)*(n+1-j));
    }
    cout << setprecision(8) << "Direct and reverse sums are "
         << sumL << " and " << sumU << endl;
    return 0;
}
```

Output of the program

```
$ ./series 2015
Direct and reverse sums are 1.6444356 and 1.6444379

$ ./series 12015
Direct and reverse sums are 1.6447253 and 1.6448509

$ ./dseries 2015
Direct and reverse sums are 1.6444379 and 1.6444379

$ ./dseries 12015
Direct and reverse sums are 1.6448508 and 1.6448508
```

Let $X_1, \ldots, X_n$ denote data of interest. We would like to compute the sample mean and sample variance

$$\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

$$S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})^2$$

We need to design an algorithm

An Informal Definition

- An algorithm is a sequence of well-defined computational steps
- that takes a set of values as input (Data)
- and produces a set of values as output (Result)

Key Features of Good Algorithms

- Correctness
  - ✓ Algorithms must produce correct outputs across all legitimate inputs
- Efficiency
  - ✓ Time efficiency : consume as small computational time as possible
  - ✓ Space efficiency : consume as small memory / storage as possible
- Simplicity
  - ✓ Concise to write down & easy to interpret

Algorithm 3.1: Two-pass algorithm

Data: $X_1, \ldots, X_n$
Result: S^2 and $\overline{X}$
$\overline{X} = 0$;
for $i = 1$ **to** n **do**
    $\overline{X} = \overline{X} + X_i$;
end
$\overline{X} = \overline{X}/n$;
$S^2 = 0$;
for $i = 1$ **to** n **do**
    $S^2 = S^2 + (X_i - \overline{X})^2$;
end
$S^2 = S^2/(n-1)$;

- First compute $\overline{X}$, and then compute S^2.
- Need to go through data twice

Data: $X_1, \dots, X_n$
Result: S^2 and $\overline{X}$
$\overline{X} = S^2 = 0$;
for $i = 1$ **to** n **do**
    $\overline{X} = \overline{X} + X_i$;
    $S^2 = S^2 + X_i^2$;
end
$S^2 = S^2 - \overline{X}^2/n$;
$\overline{X} = \overline{X}/n$;
$S^2 = S^2/(n-1)$;

- Based on $(n-1)S^2 = (\sum_{i=1}^n X_i^2) - (\sum_{i=1}^n X_i)^2/n$
- Only need to go through data once
- Might produce inaccuracies when compute the subtraction between $(\sum_{i=1}^n X_i^2)$ and $(\sum_{i=1}^n X_i)^2/n$

Algorithm 3.3: West algorithm

Data: $X_1, \ldots, X_n$
Result: S^2 and $\overline{X}$
$\overline{X} = X_1$;
$S^2 = 0$;
for $i = 2$ **to** n **do**
  $S^2 = S^2 + \frac{i-1}{i}(X_i - \overline{X})^2$;
  $\overline{X} = \overline{X} + (X_i - \overline{X})/i$;
end
$\overline{X} = \overline{X}/n$;
$S^2 = S^2/(n-1)$;

- Only need to go through data once
- Avoid to produce the inaccuracies in one pass algorithm

R storage

- The primitive data types in R
  - `character`: holds character strings rather than just a single character in C++
  - `double`: or `numeric` is the same as C++ double precision (eight-byte)
  - `integer`: is the same as (signed) int in C++ (four-byte)
  - `logical`: is the same as boolean type in C++
- The most basic data structure in `R` is an array comprised of one of the primitive data types that is referred to as an *atomic vector*.
- Using the `mode` or `storage.mode` to access the storage mode of a given object
- Machine specific details concerning storage, etc. are held in the `R` list variable `.Machine`

R storage details

```
> noquote(format(.Machine))
           double.eps       double.neg.eps          double.xmin
       2.220446049e-16      1.110223025e-16     2.225073859e-308
           double.xmax          double.base        double.digits
      1.797693135e+308                    2                   53
       double.rounding         double.guard    double.ulp.digits
                     5                    0                  -52
double.neg.ulp.digits      double.exponent       double.min.exp
                   -53                   11                -1022
        double.max.exp          integer.max          sizeof.long
                  1024           2147483647                    8
       sizeof.longlong   sizeof.longdouble       sizeof.pointer
                     8                   16                    8
```

- ?.Machine to see the meanings of each specification.
- `format`: Format an R object for pretty printing.
- `noquote`: suppresses the use of quotes in the printed output.

Summary

- Binary representation of a singed integer
- Floating point representation
  - Single precision
  - Double precision
- Relative errors
  - 7 valid digits for `float`
  - 15 valid digits for `double`
  - Relative errors on arithmetic
- Basic syntax
  - Command line input arguments `argc` and `argv`
  - Convert string to integer `atoi` or floating-point `atom` with `#inlcude<cstdlib>`
  - Typecasting
  - Set precision in the standard output with `#inlcude<iomanip>`
- Three algorithms for compute sample mean and variance
- R storage