Problem 1

When adding the number 1 to 10000 copies of the number 1×10^{-15}, the answer mathematically is $1 + 1 \times 10^{-11} = 1.00000000001$. However, in order to explore computer-represented arithmetics, we want to use this as an example of the summation accuracy with numbers of very different magnitudes.

(a)

Asumming we do not carry out the calculations in a dumb way, the best precision we can expect of our result will be the machine *double.eps*=2.2204×10^{-16}, i.e. 16 decimal places.

(b)

As shown in the results below, using *sum()* does not give the right answer (16 decimal places), but only 10 decimal places.

```
# (b)
x <- c(1, rep(1e-15, 10000))
sum(x)  # No, 10 decimal places

## [1] 1.0000000000999955673 8
```

(c)

When using a *for* loop to do the summation with 1 as the first value, we do not get the right answer, but only 11 decimal places. However, with 1 as the last value during the summation, we can get the 16-decimal-place right answer.

```
# (c)
x <- c(1, rep(1e-15, 10000))
sum = 0
for (i in 1:length(x)) sum = sum + x[i]
sum  # No, 11 decimal places

## [1] 1.00000000011102230246


sum = 0
for (i in 2:length(x)) sum = sum + x[i]
sum + x[1]  # YES, 16 decimal places

## [1] 1.00000000010000000827
```

(d)

Now, when considering 10000 copies of 1×10^{-16}, the results of repeating (b) and (c) are shown below. Function *sum()* gives about 11 decimal places, and the *for* loop sum with leading 1 gives underflown results. However, *for* loop sum with 1 as the last value still gives the 16-decimal-place right answer.

```
# (d) 1e-16 repeat b and c
x <- c(1, rep(1e-16, 10000))
sum(x)  # (b) No, 11 decimal places

## [1] 1.00000000000999644811
```

```
sum = 0
for (i in 1:length(x)) sum = sum + x[i]
sum  # (c) No, 0 decimal places, 1e-16 underflow double.eps

## [1] 1


sum = 0
for (i in 2:length(x)) sum = sum + x[i]
sum + x[1]  # (c) YES, 16 decimal places

## [1] 1.000000000001000088901
```

(e)

From the results in parts (b), (c) and (d), we can see that R's *sum()* function does not sum from left to right just using regular 16-decimal-place numbers. There is more work undergoing with the function. See detailed exploration in part (g) given below.

(f)

In order to only have 4 decimal places of accuracy in adding numbers of order of magnitude 10^{-16}, we need 10^{12} copies, which will take the storage space of $10^{12} \cdot 8[byte]/10^9$ =8000Gb.

(g)

The *sum()* function in R calls a C function *sum()* to do the summation.

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

```
void sum (int* x){
       long double s = 0.0;
       for (i = 0; i < n; i=i+1){
              s = s + x[i]
       }
       return s
}
```

In the C code, we can see that the "trick" is with the type definition of s variable. Using *long double* in C, the machine implements s with extended precision data type, i.e.
80-bit number with sign(1b) + exp(15b) + int(1b) + frac(63b). This allows *sum()* to give almost accurate results when just summing left to right and convert down to regular double precision (64 bits).

Problem 2

We will ignore pivoting for the purpose of this problem when working out the number of calculations involved in the LU and Cholesky decomposition.

(a)

For an $n \times n$ invertible matrix, the number of flops involved in the forward reduction step of the LU decomposition, which finds L and U excluding the calculations that change b to b^* is

```
LU Decomposition
L{n-1}...L2*L1*A = U //forward reduction deducted in class
L = L{n-1}^-1...L2^-1*L1^-1 //no additional calculations
b* = L{n-1}...L2*L1*b

FLOPS : rows *( col{i} + 2*col{i+1}^n ) //zero-out ops are not calculated here
L1   : (n-1)*(0 + 2*(n-1))
L2   : (n-2)*(0 + 2*(n-2))
...
L{n-1} : 1*(0 + 1)
```

$LU(A_{nn}) = \sum_{i=1}^{n-1} 2(n-i)^2 = 2n^3/3 - n^2 + n/3$

Comparing this to the number of caculations involved in the Cholesky decomposition, we have

```
Chol Decomposition
A = UT*U //Positive definite (p.d.) matrix
b* = UT^-1*b

FLOPS  : a{ii} SQRT + col{i+1}^n DIV + (i-1)*col{i}^n MUL
row 1  : 1 + (n-1) + 0
row 2  : 1 + (n-2) + 1*(1 + n-2)
row 3  : 1 + (n-3) + 2*(1 + n-3)
...
row n-1 : 1 + 1 + (n-2)*(1 + 1)
row n  : 1 + 0 + (n-1)*(1 + 0)
```

$Chol(A_{nn}) = \sum_{i=1}^{n} i(1+n-i) = n^3/6 + n^2/2 + n/3$

Now, we consider the additional computation flops involved in finding b^*

$LU(b^*) = \sum_{i=1}^{n} 2(i-1) = n^2 - n$
$Chol(b^*) = \sum_{i=1}^{n} i = n^2/2 + n/2$

(b)

The additional flops involved in the backward elimination step that finds x based on $Ux = b^*$ is
$LU(x) = Chol(x) = \sum_{i=1}^{n} i = n^2/2 + n/2$

(c)

The flops involved in part (a) and (b) when b is acutally a matrix B with p columns are simply p times the regular vector-based flops, i.e.
$LU(B^*) = p * LU(b^*) = pn^2 - pn$
$Chol(B^*) = p * Chol(b^*) = pn^2/2 + pn/2$

$LU(X) = p * LU(x) = pn^2/2 + pn/2$
$Chol(X) = p * Chol(x) = pn^2/2 + pn/2$

(d)

Another way of doing this is to explicitly find the inverse, A^{-1}, and then multiply A^{-1} by the matrix B. The number of steps involved in using LU to find $V = A^{-1} s.t. LUV = I$ is

```
Matrix inversion and multiplication
LUV = I // A = LU
(1) find L,U; see results from (a) with LU(A_nn)
(2) V = U^-1*(L^-1*I) = U^-1*(-L with diag(1))
```

$INV(A_{nn}) = LU(A_{nn}) + LU(X)_{p=n} = 7n^3/6 - n^2 + n/3$
$INV(B^*) = 0$ as B is unchanged, no B^* flops involved

(e)

When we have $A^{-1} = V$ from part (d), the flops involved in calculating VB is
$INV(X) = pn^2$

(f)

Finally we can compare the total flops involved in finding $A^{-1}B$ based on (c), (d) and (e).
$LU_O = 2n^3/3 + (3p/2 - 1)n^2 + (1/3 - p/2)n \sim O[(2n/3 + 3p/2)n^2]$
$INV_O = 7n^3/6 + (p-1)n^2 + n/3 \sim O[(7n/6 + p)n^2]$
$Chol_O = n^3/6 + (p + 1/2)n^2 + (1/3 + p)n \sim O[(n/6 + p)n^2]$

The comparison of which is better DOES depend on how big p is. Roughly, when $p >> n$, INV will take fewer flops than LU, while $Chol$ is always better than the other two methods.

(g)

Now, we are going to empirically test the flops results in R for calculating $A^{-1}B$ using an arbitrary matrix $A = Z^T Z$ with $n \in (100, 3000), p \in (1, 100, 3000)$.

```
## (g) empirical comparison: calculate X = A^{-1}*B
options(digits = 4)
require(rbenchmark)

## Loading required package:  rbenchmark

## (1) use LU:
LU <- function(A, B) {
    solve(A, B)
}
## (2) use Inverse
INV <- function(A, B) {
    V <- solve(A)
    V %*% B
}
## (3) use Cholesky
Chol <- function(A, B) {
    U <- chol(A)
    backsolve(U, backsolve(U, B, transpose = TRUE))
}
```

```
n <- c(100, 3000)  # really slow when n=3000, be careful
p <- c(1, 100, 3000)
for (i in 1:2) {
    for (j in 1:3) {
        nn <- n[i]
        pp <- p[j]
        LU_O <- 2 * nn^3/3 + (3 * pp/2 - 1) * nn^2 + (1/3 - pp/2) * nn
        INV_O <- 7 * nn^3/6 + (pp - 1) * nn^2 + nn/3
        Chol_O <- nn^3/6 + (pp + 1/2) * nn^2 + (1/3 + pp) * nn
        cat("n =", nn, "and p =", pp, ":\n")
        cat("LU O[f(n)]:  ", format(LU_O, width = 16, justify = "right"), "\n")
        cat("INV O[f(n)]: ", format(INV_O, width = 16, justify = "right"), "\n")
        cat("Chol O[f(n)]:", format(Chol_O, width = 16, justify = "right"),
            "\n")
        Z <- matrix(rnorm(nn^2), nn)
        A <- crossprod(Z)
        B <- matrix(rnorm(nn * pp), nn)
        cat("all.equal :", all.equal(LU(A, B), INV(A, B), Chol(A, B)), "\n")
        print(benchmark(LU(A, B), INV(A, B), Chol(A, B), replications = (2/i)^5)[1:6])
        cat("\n")
    }
}
```

```
## n = 100 and p = 1 :
## LU O[f(n)]:             671650
## INV O[f(n)]:           1166700
## Chol O[f(n)]:           181800
## all.equal : TRUE
##         test replications elapsed relative user.self sys.self
## 3 Chol(A, B)           32   0.024    1.000     0.028    0.020
## 2  INV(A, B)           32   0.069    2.875     0.080    0.060
## 1   LU(A, B)           32   0.025    1.042     0.020    0.032
##
## n = 100 and p = 100 :
## LU O[f(n)]:            2151700
## INV O[f(n)]:           2156700
## Chol O[f(n)]:          1181700
## all.equal : TRUE
##         test replications elapsed relative user.self sys.self
## 3 Chol(A, B)           32   0.072    1.636     0.084    0.060
## 2  INV(A, B)           32   0.083    1.886     0.092    0.072
## 1   LU(A, B)           32   0.044    1.000     0.056    0.032
##
## n = 100 and p = 3000 :
## LU O[f(n)]:           45506700
## INV O[f(n)]:          31156700
## Chol O[f(n)]:         30471700
## all.equal : Mean relative difference: 1.073e-15
##         test replications elapsed relative user.self sys.self
## 3 Chol(A, B)           32   1.747    3.617     2.052    1.424
## 2  INV(A, B)           32   0.483    1.000     0.804    0.160
## 1   LU(A, B)           32   0.809    1.675     1.164    0.436
##
```

```
## n = 3000 and p = 1 :
## LU O[f(n)]:            1.8e+10
## INV O[f(n)]:          3.15e+10
## Chol O[f(n)]:        4.514e+09
## all.equal : Mean relative difference: 6.882e-15
##         test replications elapsed relative user.self sys.self
## 3 Chol(A, B)            1   1.623    1.000     2.352    0.432
## 2  INV(A, B)            1   9.585    5.906    16.637    0.868
## 1   LU(A, B)            1   2.579    1.589     4.369    0.352
##
## n = 3000 and p = 100 :
## LU O[f(n)]:          1.934e+10
## INV O[f(n)]:         3.239e+10
## Chol O[f(n)]:        5.405e+09
## all.equal : Mean relative difference: 2.265e-15
##         test replications elapsed relative user.self sys.self
## 3 Chol(A, B)            1   1.910    1.000     2.892    0.472
## 2  INV(A, B)            1   9.696    5.076    16.969    0.832
## 1   LU(A, B)            1   2.848    1.491     4.833    0.368
##
## n = 3000 and p = 3000 :
## LU O[f(n)]:          5.849e+10
## INV O[f(n)]:         5.849e+10
## Chol O[f(n)]:        3.151e+10
## all.equal : Mean relative difference: 2.301e-15
##         test replications elapsed relative user.self sys.self
## 3 Chol(A, B)            1   8.198    1.000     14.42    0.752
## 2  INV(A, B)            1  15.171    1.851     27.68    1.020
## 1   LU(A, B)            1   8.657    1.056     15.91    0.636
```

(h)

Now suppose we are going to have to do calculations with lots of new matrices, $B_j, j = 1, 2, ...$ in the future. We have flops comparison, as below,
$LU(X_{future}) = LU(B^*) + LU(x) = 3pn^2/2 - pn/2 \sim O[(3n/2 - 1/2)pn]$
$INV(X_{future}) = INV(X) = pn^2 \sim O[n \cdot pn]$
As shown above, there are advantages to pre-compute A^{-1} with large p.

Problem 3

To compute the generalized least squares estimator, $\hat{\beta} = (X^T\Sigma^{-1}X)^{-1}X^T\Sigma^{-1}Y$ for $Xn \times p, \Sigma n \times n; n \sim 1000, p \sim 100$, we will do this in an efficient way with the following pseudo-code:

```
// S = W * W'
W = transpose(cholesky(S));
// X' * S^-1 * X * beta = X' * S^-1 * Y
// X' * (W * W')^-1 * X * beta = X' * (W * W')^-1 * Y
// X' * W'^-1 * W^-1 * X * beta = X' * W'^-1 * W^-1 * Y
// X_s' * X_s * beta = X_s' * Y_s
X_s = W^(-1) * X;
Y_s = W^(-1) * Y;
// Reduced from GLS to OLS:
X_s.qr = qr(X_s);
// Q: orthogonal; R: upper triangular
Q = qr.Q(X_s.qr);
R = qr.R(X_s.qr);
// X_s' * X_s * beta = X_s' * Y_s
// R' * Q' * Q * R * beta = R' * Q' * Y
// R * beta = Q' * Y
beta = R^(-1) * transpose(Q) * Y
```

In R, my function *gls()* is developed to do this computation.

```
#### 3. GLS estimator
n <- 2000
p <- 200

Z <- matrix(abs(rnorm(n^2)), n)
S <- crossprod(Z)/max(Z)
X <- matrix(rnorm(n * p) * 10, n)
Y <- matrix(rnorm(n) * 100, n)

gls <- function(X, S, Y) {
    W <- t(chol(S))  ## S = W * W'
    X_s <- solve(W, X)  ## X* = W^(-1) * X
    Y_s <- solve(W, Y)
    X_s.qr <- qr(X_s)
    Q = qr.Q(X_s.qr)
    R = qr.R(X_s.qr)
    beta <- backsolve(R, crossprod(Q, Y))
    return(beta)  ## efficient way
}

beta_hat <- function(X, S, Y) {
    Xt <- t(X)
    Sinv <- solve(S)
    beta <- solve(Xt %*% Sinv %*% X) %*% Xt %*% Sinv %*% Y
    return(beta)  ## naive way
}

benchmark(gls(X, S, Y), beta_hat(X, S, Y), replications = 5)[1:6]
```

```
##                test replications elapsed relative user.self sys.self
## 2 beta_hat(X, S, Y)            5   17.32    1.179     30.34    2.032
## 1      gls(X, S, Y)            5   14.69    1.000     22.39    4.413
```

We can see that there is 2X speed gain with function *gls()*.

Problem 4

Here, we want to explore whether integer calculations in R are faster than floating point calculations.

```
#### 4. compare integer and floating point calculations
require(rbenchmark)
options(digits = 4)
### comparison: Integer vs. Double
xi <- as.integer(rnorm(1e+06))  # 1e6 samples
xf <- rnorm(1e+06)
benchmark(xi[100:10000], xf[100:10000], replications = 100)[1:6]  #subsetting

##            test replications elapsed relative user.self sys.self
## 2 xf[100:10000]          100   0.016        1     0.016    0.000
## 1 xi[100:10000]          100   0.016        1     0.012    0.004

benchmark(sum(xi), sum(xf), replications = 100)[1:6]  #arithmetic: sum

##      test replications elapsed relative user.self sys.self
## 2 sum(xf)          100   0.352     1.37     0.352        0
## 1 sum(xi)          100   0.257     1.00     0.256        0

benchmark(crossprod(xi), crossprod(xf), replications = 100)[1:6]  #arithmetic: mul

##            test replications elapsed relative user.self sys.self
## 2 crossprod(xf)          100   0.727    1.000     0.728     0.00
## 1 crossprod(xi)          100   1.789    2.461     1.244     0.54
```

As shown above in the results, this claim is not necessary true. It seems that integer arithmetics are slower while subsetting vectors is almost the same as compared to floating points.

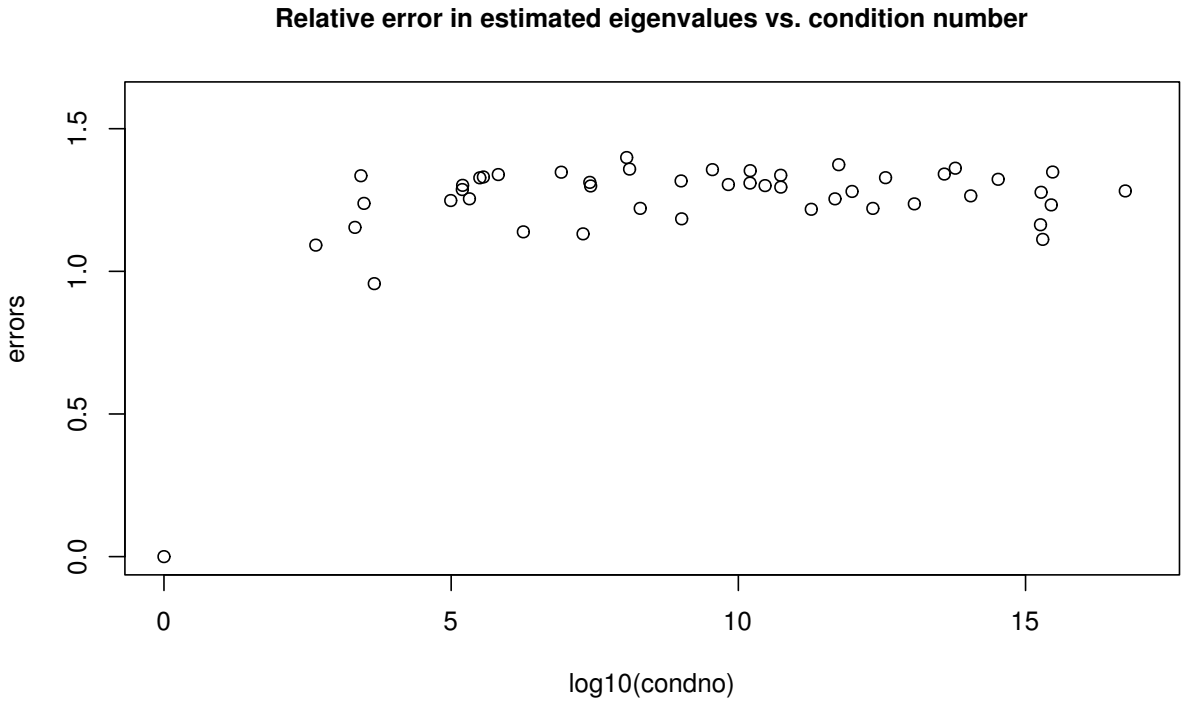
Problem 5

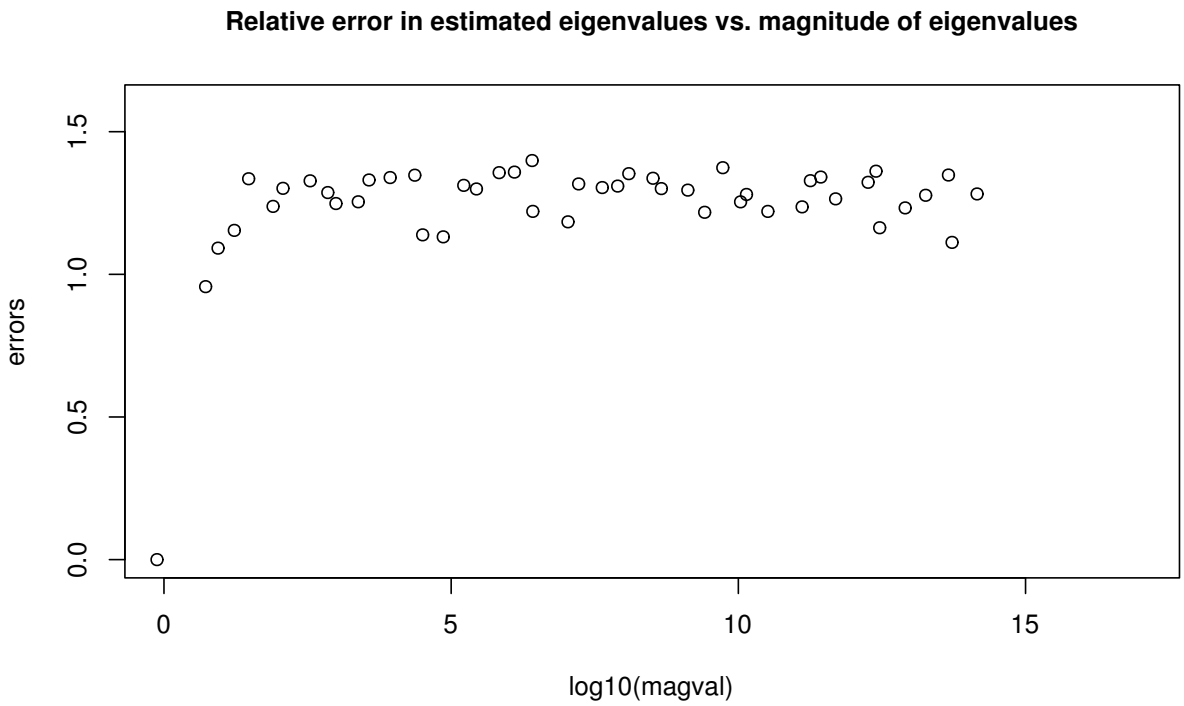
We want to empirically explore the condition number of the eigen-decomposition.

```
#### 5. Condition number of eigen decomposition
norm2 <- function(x) sqrt(sum(x^2)) # aux function calc norm2 of vecs
### Create a set of eigenvectors
n <- 100
Z <- matrix(rnorm(n^2), n)
A <- crossprod(Z)
GAMMA <- eigen(A)$vec #eigen vectors created from a random p.d. matrix
### Explore positive eigenvalues of different magnitudes
t <- .Machine$double.digits #computer presicion bits
errors <- rep(as.numeric(NA), t) #relative error
magval <- rep(as.numeric(NA), t) #eigen value magnitude
condno <- rep(as.numeric(NA), t) #condition number
for (i in 1:t){
    ## get a random set of indices to enlarge with condition number
    ids <- sample(1:n, n/4)
    if (i == 1){ # i=1, with all eigen values equal
        lambda <- rep(mean(abs(rnorm(n))), n)
    }
    else{
        lambda <- abs(rnorm(n))
    }
    ## generate eigen values with certain condition number
    lambda[ids] <- lambda[ids]*(2^(i-1))
    magval[i] <- max(lambda)
    condno[i] <- magval[i]/min(lambda)

    LAMBDA <- diag(lambda)
    testA <- GAMMA %*% LAMBDA %*% t(GAMMA) # create p.d. matrix
    errors[i] <- norm2(eigen(testA)$val - lambda)/norm2(lambda)
    if (length(which(eigen(testA)$val < 0)) > 0){ # not p.d.
        cat("Failed at condition number", condno[i],
            "with relative error of", errors[i], "\n")
        par(mfrow=c(2,1),cex= 1, cex.main= 1)
        plot(log10(condno), errors, xlim=c(0, 17), ylim=c(0, 1.6),
             main="Relative error in estimated eigenvalues vs. condition number")
        plot(log10(magval), errors, xlim=c(0, 17), ylim=c(0, 1.6),
             main="Relative error in estimated eigenvalues vs. magnitude of eigenvalues")
        break
    }
}

## Failed at condition number 5.454e+16 with relative error of 1.282
```





As shown in the results above, the test matrix fails to be numerically positive definite with condition number around 10^{17} and estimated eigenvalue relative error around 1.2. The plots show that the error saturates pretty quickly at the smaller condition numbers, but only when the condition number overflows, the matrix will no longer be positive definite.

Problem 6

In order to see the representation of decimal numbers in base-2 computer bits, both R and C codes are written to see the rounding or "garbage digits" phenomenon. In the R code, we can see "rounding" with $x \sim 10^{-17}$, when x get multiplied by 2^{56} to be converted into integers.

```
#### 6. "Garbage" digits
options(digits = 22)
x <- c(0.12345678123456780000, 0.12345678123456781000,
       0.12345678123456782000, 0.12345678123456783000,
       0.12345678123456784000, 0.12345678123456785000,
       0.12345678123456786000, 0.12345678123456787000,
       0.12345678123456788000, 0.12345678123456789000)
x

##  [1] 0.1234567812345677972896 0.1234567812345678111674
##  [3] 0.1234567812345678250452 0.1234567812345678250452
##  [5] 0.1234567812345678389230 0.1234567812345678528008
##  [7] 0.1234567812345678666786 0.1234567812345678666786
##  [9] 0.1234567812345678805563 0.1234567812345678944341

x*2^56 # see longdouble.c (consistent, last bit is garbage)

##  [1] 8895998623429766 8895998623429767 8895998623429768 8895998623429768
##  [5] 8895998623429769 8895998623429770 8895998623429771 8895998623429771
##  [9] 8895998623429772 8895998623429773


y <- c(0.00000000123456781234567800000, 0.00000000123456781234567810000,
       0.00000000123456781234567820000, 0.00000000123456781234567830000,
       0.00000000123456781234567840000, 0.00000000123456781234567850000,
       0.00000000123456781234567860000, 0.00000000123456781234567870000,
       0.00000000123456781234567880000, 0.00000000123456781234567890000)
y

##  [1] 1.2345678123456780141600e-09 1.2345678123456780141600e-09
##  [3] 1.2345678123456782209560e-09 1.2345678123456782209560e-09
##  [5] 1.2345678123456784277510e-09 1.2345678123456784277510e-09
##  [7] 1.2345678123456786345460e-09 1.2345678123456786345460e-09
##  [9] 1.2345678123456788413410e-09 1.2345678123456788413410e-09

y*2^82 # see longdouble.c (minor inconsistency, last bit is also garbage)

##  [1] 5970003617639354 5970003617639354 5970003617639355 5970003617639355
##  [5] 5970003617639356 5970003617639356 5970003617639357 5970003617639357
##  [9] 5970003617639358 5970003617639358
```

We can see that there really is "rounding" with these numbers when represented using base-2 computer numbers. The C code uses *long double* type to represent floating points in 80 bits, which gives us the references when verifying the R code results.

```c
#include <stdio.h>
#include <math.h>
int main (void)
{
        long double a[] = {0.12345678123456780000,0.12345678123456781000,0.12345678123456782000,
                           0.12345678123456783000,0.12345678123456784000,0.12345678123456785000,
                           0.12345678123456786000,0.12345678123456787000,0.12345678123456788000,
                           0.12345678123456789000};
        long double b[] = {0.000000001234567812345678000,0.000000001234567812345678100,
                           0.000000001234567812345678200,0.000000001234567812345678300,
                           0.000000001234567812345678400,0.000000001234567812345678500,
                           0.000000001234567812345678600,0.000000001234567812345678700,
                           0.000000001234567812345678800,0.000000001234567812345678900};
        long double ax2[10];
        long double by2[10];
        long double x2 = powl(2, 56);
        long double y2 = powl(2, 82);
        int i=0;
        printf("x2 = %Lf\n", x2);
        for (i=0; i<10; i++){
          ax2[i] = a[i]*x2;
          printf("long_d a = %.60Lf\n", a[i]);
          printf("integera = %Lf\n", ax2[i]);
        }
        printf("y2 = %Lf\n", y2);
        for (i=0; i<10; i++){
          by2[i] = b[i]*y2;
          printf("long_d b = %.90Lf\n", b[i]);
          printf("integerb = %Lf\n", by2[i]);
        }
        return 0;
}
```

```
x2 = 72057594037927936.000000
long_d a = 0.123456781234567797289614077271835412830114364624023437500000
integera = 8895998623429766.000000
long_d a = 0.123456781234567811167401885086292168125510215759277343750000
integera = 8895998623429767.000000
long_d a = 0.123456781234567825045189692900748923420906066894531250000000
integera = 8895998623429768.000000
long_d a = 0.123456781234567825045189692900748923420906066894531250000000
integera = 8895998623429768.000000
long_d a = 0.123456781234567838922977500715205678716301918029785156250000
integera = 8895998623429769.000000
long_d a = 0.123456781234567852800765308529662434011697769165039062500000
integera = 8895998623429770.000000
long_d a = 0.123456781234567866678553116344119189307093620300292968750000
integera = 8895998623429771.000000
long_d a = 0.123456781234567866678553116344119189307093620300292968750000
integera = 8895998623429771.000000
long_d a = 0.123456781234567880556340924158575944602489471435546875000000
integera = 8895998623429772.000000
long_d a = 0.123456781234567894434128731973032699897885322570800781250000
integera = 8895998623429773.000000
```

```
y2 = 4835703278458516698824704.000000
long_d b = \
    0.000000001234567812345678014160409589395663021438309669974842108786106109619140625000000000
integerb = 5970003617639354.000000
long_d b = \
    0.000000001234567812345678014160409589395663021438309669974842108786106109619140625000000000
integerb = 5970003617639354.000000
long_d b = \
    0.000000001234567812345678220955562727652581739290482687465555500239133834838867187500000000
integerb = 5970003617639355.000000
long_d b = \
    0.000000001234567812345678220955562727652581739290482687465555500239133834838867187500000000
integerb = 5970003617639355.000000
long_d b = \
    0.000000001234567812345678427750715865909500457142655704956268891692161560058593750000000000
integerb = 5970003617639356.000000
long_d b = \
    0.000000001234567812345678427750715865909500457142655704956268891692161560058593750000000000
integerb = 5970003617639356.000000
long_d b = \
    0.000000001234567812345678634545869004166419174994828722446982283145189285278320312500000000
integerb = 5970003617639357.000000
long_d b = \
    0.000000001234567812345678634545869004166419174994828722446982283145189285278320312500000000
integerb = 5970003617639357.000000
long_d b = \
    0.000000001234567812345678841341022142423337892847001739937695674598217010498046875000000000
integerb = 5970003617639358.000000
long_d b = \
    0.000000001234567812345678841341022142423337892847001739937695674598217010498046875000000000
integerb = 5970003617639358.000000
```

We can see that C code results verifies our conclusion that the last bit is "garbage bit" here.