Strassen algorithm implementation in R

http://statistic-on-air.blogspot.com
todoslogos@gmail.com

October 18, 2010

1 Illustration of method

The Strassen algorithm is an algorithm used for matrix multiplication. It is faster than the standard matrix multiplication algorithm, but would be slower than the fastest known algorithm (Coppersmith-Winograd algorithm) for extremely large matrices.

Let $\mathbf{A}, \mathbf{B}$ two square matrix, $\in R^{2^n \times 2^n}$, with $n = 2, 3, \ldots$. We want to calculate the matrix $\mathbf{C}$, defined by $\mathbf{C} = \mathbf{AB}$.

First, we divide the two matrix $\mathbf{A}, \mathbf{B}$, into equally size block-matrices of dimensions $2^{n-1} \times 2^{n-1}$:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} \tag{1}$$

Now define new matrices:

$$\mathbf{M}_1 = (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22})$$

$$\mathbf{M}_2 = (\mathbf{A}_{21} + \mathbf{A}_{22})\mathbf{B}_{11}$$

$$\mathbf{M}_3 = \mathbf{A}_{11}(\mathbf{B}_{12} - \mathbf{B}_{22})$$

$$\mathbf{M}_4 = \mathbf{A}_{22}(\mathbf{B}_{21} - \mathbf{B}_{11})$$

$$\mathbf{M}_5 = (\mathbf{A}_{11} + \mathbf{A}_{12})\mathbf{B}_{22}$$

$$\mathbf{M}_6 = (\mathbf{A}_{21} - \mathbf{A}_{11})(\mathbf{B}_{11} + \mathbf{B}_{12})$$

$$\mathbf{M}_7 = (\mathbf{A}_{12} - \mathbf{A}_{22})(\mathbf{B}_{21} + \mathbf{B}_{22})$$

The block-matrices of the product matrix $\mathbf{C}$ are:

$$\mathbf{C}_{11} = \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7$$

$\mathbf{C}_{12} = \mathbf{M}_3 + \mathbf{M}_5$

$\mathbf{C}_{21} = \mathbf{M}_2 + \mathbf{M}_4$

$\mathbf{C}_{22} = \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6$

So the matrix $\mathbf{C}$ is:

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{bmatrix} \tag{2}$$

1.1 Solved example

We have these two matrices:

$$\mathbf{A} = \begin{bmatrix} 7 & 31 & 13 & 106 \\ 24 & 19 & 51 & 68 \\ 139 & 127 & 121 & 117 \\ 13 & 105 & 53 & 59 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 22 & 111 & 93 & 181 \\ 155 & 42 & 120 & 17 \\ 171 & 115 & 26 & 26 \\ 167 & 203 & 6 & 31 \end{bmatrix}$$

Split the matrices:

$$\mathbf{A} = \left[\begin{array}{cc|cc} 7 & 31 & 13 & 106 \\ 24 & 19 & 51 & 68 \\ \hline 139 & 127 & 121 & 117 \\ 13 & 105 & 53 & 59 \end{array}\right] \quad \mathbf{B} = \left[\begin{array}{cc|cc} 22 & 111 & 93 & 181 \\ 155 & 42 & 120 & 17 \\ \hline 171 & 115 & 26 & 26 \\ 167 & 203 & 6 & 31 \end{array}\right]$$

Now we have, for example, $\mathbf{A}_{11} = \begin{bmatrix} 7 & 31 \\ 24 & 19 \end{bmatrix}$, and $\mathbf{B}_{21} = \begin{bmatrix} 171 & 115 \\ 167 & 203 \end{bmatrix}$. Now calculate the matrices $\mathbf{M}_{1:7}$:

$$\mathbf{M}_1 = \begin{bmatrix} 29972 & 16254 \\ 28340 & 16243 \end{bmatrix}$$

$$\mathbf{M}_2 = \begin{bmatrix} 43540 & 26872 \\ 39108 & 14214 \end{bmatrix}$$

$$\mathbf{M}_3 = \begin{bmatrix} 4003 & 3774 \\ 651 & 3454 \end{bmatrix}$$

$$\mathbf{M}_4 = \begin{bmatrix} 19433 & 8605 \\ 19321 & 9711 \end{bmatrix}$$

$$\mathbf{M}_5 = \begin{bmatrix} 1342 & 2472 \\ 4767 & 4647 \end{bmatrix}$$

$$\mathbf{M}_6 = \begin{bmatrix} 41580 & 22385 \\ 44208 & 1862 \end{bmatrix}$$

$$\mathbf{M}_7 = \begin{bmatrix} -23179 & 1163 \\ -17802 & 1824 \end{bmatrix}$$

Now calculate:

$$\mathbf{C}_{11} = \begin{bmatrix} 24884 & 23550 \\ 25092 & 23131 \end{bmatrix}$$
$$\mathbf{C}_{12} = \begin{bmatrix} 5345 & 6246 \\ 5418 & 8101 \end{bmatrix}$$
$$\mathbf{C}_{21} = \begin{bmatrix} 62973 & 35477 \\ 58429 & 23925 \end{bmatrix}$$
$$\mathbf{C}_{22} = \begin{bmatrix} 32015 & 15541 \\ 34091 & 7345 \end{bmatrix}$$

The final result is:

$$\mathbf{C} = \begin{bmatrix} 24884 & 23550 & 62973 & 35477 \\ 25092 & 23131 & 58429 & 23925 \\ 5345 & 6246 & 32015 & 15541 \\ 5418 & 8101 & 34091 & 7345 \end{bmatrix}$$

1.2 Solution with R

We saw that there are 4 steps to be solved:

1. Split matrices
2. Calculate $\mathbf{M}_{1:7}$
3. Calculate block-matrices of the product $\mathbf{C}$
4. Recompose the matrix $\mathbf{C}$

```
A ← matrix(c
    (7,31,13,106,24,19,51,68,139,127,121,117,13,105,53,59),
    byrow=T, nrow=4)
B ← matrix(c
    (22,111,93,181,155,42,120,17,171,115,26,26,167,203,6,31),
     byrow=T, nrow=4)

     # Step -1-
A11 ← A[1:2,1:2]
A12 ← A[1:2,3:4]
A21 ← A[3:4,1:2]
A22 ← A[3:4,3:4]

B11 ← B[1:2,1:2]
B12 ← B[1:2,3:4]
B21 ← B[3:4,1:2]
B22 ← B[3:4,3:4]
```

```
    # Step -2-
M1 ← (A11+A22) %*% (B11+B22)
M2 ← (A21+A22) %*% B11
M3 ← A11 %*% (B12-B22)
M4 ← A22 %*% (B21-B11)
M5 ← (A11+A12) %*% B22
M6 ← (A21-A11) %*% (B11+B12)
M7 ← (A12-A22) %*% (B21+B22)

    # Step -3-
C11 ← M1+M4-M5+M7
C12 ← M3+M5
C21 ← M2+M4
C22 ← M1-M2+M3+M6

    # Step -4-
C ← rbind(cbind(C11,C12), cbind(C21,C22))
```

Now verify the result, comparing the matrix C obtained with Strassen algorithm, with that calculated with the standard function of R:

```
C
      [,1]  [,2]  [,3]  [,4]
[1,] 24884 25092  5345  5418
[2,] 23550 23131  6246  8101
[3,] 62973 58429 32015 34091
[4,] 35477 23925 15541  7345

A%*%B
      [,1]  [,2]  [,3]  [,4]
[1,] 24884 25092  5345  5418
[2,] 23550 23131  6246  8101
[3,] 62973 58429 32015 34091
[4,] 35477 23925 15541  7345

all(C == A%*%B)
[1] TRUE
```

1.3 Strassen algorithm for rectangular matrix

If $\mathbf{A}$ and $\mathbf{B}$ are two rectangular matrix (respectively $m \times n$ and $n \times p$), to use the Strassen algorithm we need to transform them into square matrices of size $2^k \times 2^k$.

Let be for example:

$$\mathbf{A} = \begin{bmatrix} 7 & 31 & 13 \\ 24 & 19 & 51 \\ 139 & 127 & 121 \\ 13 & 105 & 53 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 22 & 111 & 93 & 181 \\ 155 & 42 & 120 & 17 \\ 171 & 115 & 26 & 26 \end{bmatrix}$$

We transform that into:

$$\mathbf{A} = \begin{bmatrix} 7 & 31 & 13 & 0 \\ 24 & 19 & 51 & 0 \\ 139 & 127 & 121 & 0 \\ 13 & 105 & 53 & 0 \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} 22 & 111 & 93 & 181 \\ 155 & 42 & 120 & 17 \\ 171 & 115 & 26 & 26 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

and now we can procede with the 4 steps seen before.

These transformation is easily done in R :

```
A ← matrix(c(7,31,13,24,19,51,139,127,121,13,105,53), byrow=
    T, nrow=4)
B ← matrix(c(22,111,93,181,155,42,120,17,171,115,26,26),
    byrow=T, nrow=3)

A ← "[←"(matrix(0, 4, 4), 1:nrow(A), 1:ncol(A), value = A)
B ← "[←"(matrix(0, 4, 4), 1:nrow(B), 1:ncol(B), value = B)

A
      [,1] [,2] [,3] [,4]
[1,]     7   31   13    0
[2,]    24   19   51    0
[3,]   139  127  121    0
[4,]    13  105   53    0

B
      [,1] [,2] [,3] [,4]
[1,]    22  111   93  181
[2,]   155   42  120   17
[3,]   171  115   26   26
[4,]     0    0    0    0
```

Now repeat the same 4 steps:

```
    # Step -1-
A11 ← A[1:2,1:2]
A12 ← A[1:2,3:4]
A21 ← A[3:4,1:2]
A22 ← A[3:4,3:4]

B11 ← B[1:2,1:2]
B12 ← B[1:2,3:4]
B21 ← B[3:4,1:2]
B22 ← B[3:4,3:4]

    # Step -2-
M1 ← (A11+A22) %*% (B11+B22)
M2 ← (A21+A22) %*% B11
M3 ← A11 %*% (B12-B22)
M4 ← A22 %*% (B21-B11)
M5 ← (A11+A12) %*% B22
```

```
M6 ← (A21-A11) %*% (B11+B12)
M7 ← (A12-A22) %*% (B21+B22)

    # Step -3-
C11 ← M1+M4-M5+M7
C12 ← M3+M5
C21 ← M2+M4
C22 ← M1-M2+M3+M6

    # Step -4-
C ← rbind(cbind(C11,C12), cbind(C21,C22))
```

Verify the result:

```
C
      [,1]  [,2]  [,3]  [,4]
[1,]  7182  3574  4709  2132
[2,] 12194  9327  5838  5993
[3,] 43434 34678 31313 30464
[4,] 25624 11948 15187  5516

A%*%B
      [,1]  [,2]  [,3]  [,4]
[1,]  7182  3574  4709  2132
[2,] 12194  9327  5838  5993
[3,] 43434 34678 31313 30464
[4,] 25624 11948 15187  5516

all(C == A%*%B)
[1] TRUE
```

Consider now a second example. Lets suppose we have two matrices, respectively $m \times n$ and $n \times p$. In the previous example $m = p$, and so the matrix product was a square matrix. But if $m \neq p$, the matrix product will be rectangular itself. Consider for example:

$$\mathbf{A} = \begin{bmatrix} 1 & 6 \\ 2 & 7 \\ 3 & 8 \\ 4 & 9 \\ 5 & 10 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 1 & 3 & 5 & 7 & 9 & 11 & 13 & 15 & 17 & 19 \\ 2 & 4 & 6 & 8 & 10 & 12 & 14 & 16 & 18 & 20 \end{bmatrix}$$

Matrix $\mathbf{A}$ is 5×2, while matrix $\mathbf{B}$ is 2×10. So $\mathbf{C} = \mathbf{AB}$ will be 5×10.

To apply the Strassen algorithm we need to expand the two matrices to obtain square matrices, and so the result will be square matrix:

```
A ← matrix(c(1:10), nrow=5)
B ← matrix(c(1:20), nrow=2)

A ← "[←"(matrix(0, 4, 4), 1:nrow(A), 1:ncol(A), value = A)
B ← "[←"(matrix(0, 4, 4), 1:nrow(B), 1:ncol(B), value = B)
```

```

    # Step -1-
A11 ← A[1:2,1:2]
A12 ← A[1:2,3:4]
A21 ← A[3:4,1:2]
A22 ← A[3:4,3:4]

B11 ← B[1:2,1:2]
B12 ← B[1:2,3:4]
B21 ← B[3:4,1:2]
B22 ← B[3:4,3:4]

    # Step -2-
M1 ← (A11+A22) %*% (B11+B22)
M2 ← (A21+A22) %*% B11
M3 ← A11 %*% (B12-B22)
M4 ← A22 %*% (B21-B11)
M5 ← (A11+A12) %*% B22
M6 ← (A21-A11) %*% (B11+B12)
M7 ← (A12-A22) %*% (B21+B22)

    # Step -3-
C11 ← M1+M4-M5+M7
C12 ← M3+M5
C21 ← M2+M4
C22 ← M1-M2+M3+M6

    # Step -4-
C ← rbind(cbind(C11,C12), cbind(C21,C22))
```

From the product matrix we need to delete the zero-rows and the zero-columns:

```
        m ← dim(A)[1]+1
        p ← dim(B)[2]+1
        mC ← dim(C)[1]
        pC ← dim(C)[2]
        if(m<mC) { C ← C[-c(m:mC),] }
        if(p<pC) { C ← C[,-c(p:pC)] }

C
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   13   27   41   55   69   83   97  111  125   139
[2,]   16   34   52   70   88  106  124  142  160   178
[3,]   19   41   63   85  107  129  151  173  195   217
[4,]   22   48   74  100  126  152  178  204  230   256
[5,]   25   55   85  115  145  175  205  235  265   295

A%*%B
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]   13   27   41   55   69   83   97  111  125   139
```

```
[2,]    16    34    52    70    88   106   124   142   160    178
[3,]    19    41    63    85   107   129   151   173   195    217
[4,]    22    48    74   100   126   152   178   204   230    256
[5,]    25    55    85   115   145   175   205   235   265    295

all(C == A%*%B)
[1] TRUE
```

1.4 Function `strassen()`

If it is clear the mechanism by which Strassen's algorithm works, and the steps to perform it (in the case of square matrices and in the case of rectangular matrices), we can now write a function that automates the computations:

```
strassen ← function(A, B){

        div4 ← function(A, r){
                A ← list(A)
                A11 ← A[[1]][1:(r/2),1:(r/2)]
                A12 ← A[[1]][1:(r/2),(r/2+1):r]
                A21 ← A[[1]][(r/2+1):r,1:(r/2)]
                A22 ← A[[1]][(r/2+1):r,(r/2+1):r]
                A ← list(X11=A11, X12=A12, X21=A21, X22=A22)
                return(A)
        }

        n ← round(log(max(nrow(A), ncol(A), nrow(B), ncol(B)
            ), 2))
        if(n < log(max(nrow(A), ncol(A), nrow(B), ncol(B)),
            2)) { n = n+1 }
        A ← "[←"(matrix(0, 2^n, 2^n), 1:nrow(A), 1:ncol(A),
            value = A)
        B ← "[←"(matrix(0, 2^n, 2^n), 1:nrow(B), 1:ncol(B),
            value = B)

        A ← div4(A, dim(A)[1])
        B ← div4(B, dim(B)[1])
        M1 ← (A$X11+A$X22) %*% (B$X11+B$X22)
        M2 ← (A$X21+A$X22) %*% B$X11
        M3 ← A$X11 %*% (B$X12-B$X22)
        M4 ← A$X22 %*% (B$X21-B$X11)
        M5 ← (A$X11+A$X12) %*% B$X22
        M6 ← (A$X21-A$X11) %*% (B$X11+B$X12)
        M7 ← (A$X12-A$X22) %*% (B$X21+B$X22)

        C11 ← M1+M4-M5+M7
        C12 ← M3+M5
        C21 ← M2+M4
        C22 ← M1-M2+M3+M6
```

```
        C ← rbind(cbind(C11,C12), cbind(C21,C22))
        m ← dim(A)[1]+1
        p ← dim(B)[2]+1
        mC ← dim(C)[1]
        pC ← dim(C)[2]
        if(m<mC) { C ← C[-c(m:mC),] }
        if(p<pC) { C ← C[,-c(p:pC)] }
        return(C)
}
```

Now we'll try to solve the three example seen before, using the function `strassen(A,B)`:

```
    # Example -1-
A ← matrix(c
    (7,31,13,106,24,19,51,68,139,127,121,117,13,105,53,59),
    byrow=T, nrow=4)
B ← matrix(c
    (22,111,93,181,155,42,120,17,171,115,26,26,167,203,6,31),
     byrow=T, nrow=4)

strassen(A,B)
      [,1]  [,2]  [,3]  [,4]
[1,] 24884 25092  5345  5418
[2,] 23550 23131  6246  8101
[3,] 62973 58429 32015 34091
[4,] 35477 23925 15541  7345

A%*%B
      [,1]  [,2]  [,3]  [,4]
[1,] 24884 25092  5345  5418
[2,] 23550 23131  6246  8101
[3,] 62973 58429 32015 34091
[4,] 35477 23925 15541  7345

    # Example -2-
A ← matrix(c(7,31,13,24,19,51,139,127,121,13,105,53), byrow=
    T, nrow=4)
B ← matrix(c(22,111,93,181,155,42,120,17,171,115,26,26),
    byrow=T, nrow=3)

strassen(A,B)
      [,1]  [,2]  [,3]  [,4]
[1,]  7182  3574  4709  2132
[2,] 12194  9327  5838  5993
[3,] 43434 34678 31313 30464
[4,] 25624 11948 15187  5516

A%*%B
      [,1]  [,2]  [,3]  [,4]
```

```
[1,]  7182  3574  4709  2132
[2,] 12194  9327  5838  5993
[3,] 43434 34678 31313 30464
[4,] 25624 11948 15187  5516

    # Example -3-
A <- matrix(c(1:10), nrow=5)
B <- matrix(c(1:20), nrow=2)

all( strassen(A,B) == A%*%B )
[1] TRUE
```

2 Computation time

Now comes the important part. All the simulations that follow were performed on a Core i3-530, 2.93gHz, 64bit.

In the introduction, it was specified that the Strassen algorithm is faster than the classical matrix multiplication. We want to verify this claim; first write a function that performs matrix multiplication with the standard method, and call it `matmult(A,B)`:

```
matmult <- function(A, B){
        p <- dim(A)[1]
        q <- dim(B)[2]
        c <- matrix(nrow=p, ncol=q)
        for(i in 1:p){
                for(j in 1:q){
                        c[i, j] <- sum(A[i,] * B[,j])
                }
        }
        return(c)
}
```

Now generate two square matrices 32×32 (I hate decimal number, so I'll work with integer, just for my convenience):

```
A <- matrix(abs(trunc(rnorm(32*32)*100)), 32,32)
B <- matrix(abs(trunc(rnorm(32*32)*100)), 32,32)
```

Now compare how long it takes to run the product with the functions `matmult()` and `strassen()`:

```
system.time(matmult(A,B))
   user  system elapsed
   0.02    0.00    0.01
system.time(strassen(A,B))
   user  system elapsed
      0       0       0

all( matmult(A,B) == strassen(A,B) )
```

```
[1] TRUE
```

The function `strassen()` is faster; now try on bigger matrices:

```
A <- matrix(abs(trunc(rnorm(64*64)*100)), 64,64)
B <- matrix(abs(trunc(rnorm(64*64)*100)), 64,64)

system.time(matmult(A,B))
   user  system elapsed
   0.05    0.00    0.05

system.time(strassen(A,B))
   user  system elapsed
      0       0       0

all( matmult(A,B) == strassen(A,B) )
[1] TRUE

# And with rectangular matrices:
A <- matrix(abs(trunc(rnorm(120*80)*100)), 120,80)
B <- matrix(abs(trunc(rnorm(80*110)*100)), 80,110)

system.time(matmult(A,B))
   user  system elapsed
   0.14    0.00    0.14

system.time(strassen(A,B))
   user  system elapsed
      0       0       0

all( matmult(A,B) == strassen(A,B) )
[1] TRUE
```

What can we say about the internal operator `%*%`? I don't know what is the algorithm used by R for matrix multiplication; however we can use it as a reference, to check the speed of the function `strassen()`. For example for matrices of 128×128, we have:

```
A <- matrix(abs(trunc(rnorm(128*128)*100)), 128,128)
B <- matrix(abs(trunc(rnorm(128*128)*100)), 128,128)

system.time(strassen(A,B))
   user  system elapsed
   0.01    0.00    0.02

system.time(A%*%B)
   user  system elapsed
      0       0       0

all( strassen(A,B) == A%*%B )
[1] TRUE
```

It appears that the command `%*%` is faster. But try to multiply bigger matrices:

```
A ← matrix(abs(trunc(rnorm(1024*1024)*100)), 1024,1024)
B ← matrix(abs(trunc(rnorm(1024*1024)*100)), 1024,1024)

system.time(strassen(A,B))
   user  system elapsed
   1.27    0.01    1.28

system.time(A%*%B)
   user  system elapsed
   1.50    0.00    1.52

all( strassen(A,B) == A%*%B )
[1] TRUE
```

Bingo! For matrices of size near to $2^{10} \times 2^{10}$, the `strassen()` function is faster.

What happens for matrices $2^{11} \times 2^{11}$?

```
A ← matrix(abs(trunc(rnorm(2048*2048)*100)), 2048,2048)
B ← matrix(abs(trunc(rnorm(2048*2048)*100)), 2048,2048)

system.time(strassen(A,B))
   user  system elapsed
  11.64    0.14   11.81

system.time(A%*%B)
   user  system elapsed
  11.93    0.01   11.98

all( strassen(A,B) == A%*%B )
[1] TRUE
```

There was still a gain, but not significant. This is because when the matrices $\mathbf{M}_{1:7}$ are calculated, it is used the R internal algorithm, and not the Strassen's algorithm. Then we can obtain a further improvement, writing the function `strassen2()`:

```
strassen2 ← function(A, B){

        div4 ← function(A, r){
                A ← list(A)
                A11 ← A[[1]][1:(r/2),1:(r/2)]
                A12 ← A[[1]][1:(r/2),(r/2+1):r]
                A21 ← A[[1]][(r/2+1):r,1:(r/2)]
                A22 ← A[[1]][(r/2+1):r,(r/2+1):r]
                A ← list(X11=A11, X12=A12, X21=A21, X22=A22)
                return(A)
        }

```

```
        n <- round(log(max(nrow(A), ncol(A), nrow(B), ncol(B)), 2))
        if(n < log(max(nrow(A), ncol(A), nrow(B), ncol(B)), 2)) { n = n+1 }
        A <- "[<-"(matrix(0, 2^n, 2^n), 1:nrow(A), 1:ncol(A), value = A)
        B <- "[<-"(matrix(0, 2^n, 2^n), 1:nrow(B), 1:ncol(B), value = B)

        A <- div4(A, dim(A)[1])
        B <- div4(B, dim(B)[1])
        M1 <- strassen((A$X11+A$X22) , (B$X11+B$X22))
        M2 <- strassen((A$X21+A$X22) , B$X11)
        M3 <- strassen(A$X11 , (B$X12-B$X22))
        M4 <- strassen(A$X22 , (B$X21-B$X11))
        M5 <- strassen((A$X11+A$X12) , B$X22)
        M6 <- strassen((A$X21-A$X11) , (B$X11+B$X12))
        M7 <- strassen((A$X12-A$X22) , (B$X21+B$X22))

        C11 <- M1+M4-M5+M7
        C12 <- M3+M5
        C21 <- M2+M4
        C22 <- M1-M2+M3+M6

        C <- rbind(cbind(C11,C12), cbind(C21,C22))
        m <- dim(A)[1]+1
        p <- dim(B)[2]+1
        mC <- dim(C)[1]
        pC <- dim(C)[2]
        if(m<mC) { C <- C[-c(m:mC),] }
        if(p<pC) { C <- C[,-c(p:pC)] }
        return(C)
}
```

Now compare the computation timing:

```
system.time(strassen(A,B))
   user  system elapsed
  11.61    0.14   11.78

system.time(A%*%B)
   user  system elapsed
  11.83    0.02   11.87

system.time(strassen2(A,B))
   user  system elapsed
   9.53    0.28    9.81

all( strassen(A,B) == A%*%B )
[1] TRUE

```

```
all( strassen(A,B) == strassen2(A,B) )
[1] TRUE
```

As expected, with the function `strassen2()` there was a further speeding up the process, saving the 19.44% of the time, compared to the operator of R.

It's easy to write the function `strassen3()`, `strassen4()`, `strassen5()` and more, for bigger matrices.

3 Conclusions

The table below shows a simulation of the timing of the various functions on matrices of increasing size:

Matrix	%*%	strassen1()	strassen2()	strassen3()	strassen4()	Saving time
512×512	0.16	0.17	0.20	0.23	0.43	/
1024×1024	1.49	1.23	1.37	1.63	2.09	17.45%
2048×2048	11.90	11.87	9.04	9.84	11.04	24.03%
4096×4096	96.24	88.02	83.65	68.13	71.99	29.21%

What happens with rectangular matrices or square matrices of sizes different from $2^n \times 2^n$? The computation time is longer, because we need to transform the rectangular matrices into matrices of appropriate dimensions, more time to execute the algorithm on block-matrices of zeros, and more time to delete the zero-rows and the zero-columns from the product matrix. Consequently, the operator R is more functional. For example, consider the following case:

```
A ← matrix(abs(trunc(rnorm(2*1000)*100)), 2,1000)
B ← matrix(abs(trunc(rnorm(1000*4)*100)), 1000,4)

A%*%B
        [,1]    [,2]    [,3]    [,4]
[1,] 6113773 6046968 6255471 5994522
[2,] 6335823 6116985 6607189 6228294

system.time(strassen(A,B))
   user  system elapsed
   0.23    0.00    0.24

system.time(A%*%B)
   user  system elapsed
      0       0       0
```

The `strassen()` function is still faster for that rectangular matrices, with dimensions near to $2^n \times 2^n$:

```
A ← matrix(abs(trunc(rnorm(1000*1010)*100)), 1000,1010)
B ← matrix(abs(trunc(rnorm(1010*1000)*100)), 1010,1000)

system.time(strassen(A,B))
```

```
   user  system elapsed
   1.28    0.00    1.30

system.time(A%*%B)
   user  system elapsed
   1.42    0.00    1.44
```

In conclusion, if we need to multiply two matrices, both $2^n \times 2^n$, with $n \geq 10$, `strassenn()` function allows a faster calculation. The same goes for multiplication of rectangular matrices, whose dimensions are close to $2^n \times 2^n$, with $n \geq 10$.