Michel Jan Marinus Bieleveld

outorado no Laboratório de Automação Agrícola da Escola Politécnica da USP.

O título de minha tese é: ".?.", orientado pelo prof. Antonio Mauro Saraiva.

Meus Exercícios

Linque para a página com os meus exercícios resolvidos;

- Aula 1 - Introdução ao R
- Aula 2 - Funções matemáticas
- Aula 3 - Leitura e manipulação de dados
- Aula 4 - Análise exploratória
- Aula 5 - Criação e edição de gráficos
- Aula 6 - Testes de significância
- Aula 7 - Regressão linear simples
- Aula 8 - Regressão linear múltipla
- Aula 9 - Construção de funções simples

Debugging

A small debugging tutorial with debug()

Final work

I have created a package and have considered the commentaries of Leo and Diago. Therefore I have let go the requirement of improving the speed of the matrix multiplication. Still I have learned considerable amount of working with packages, compiling ccode on windows for R and documenting the functions. I agree that the C++ version has perhaps not to do so much with R itself and have therefore added a performance measurement function completely in R. Without further ado here are the links to the package and documentation. In addition since the supplied binary will only work on Windows 64bit with R version 2.15 I have added a screenshot that shows the installation, loading, help file and running a build in example. {{:bie5782:01_curso_atual:alunos:trabalho_final:michel.bieleveld:screenshot.png?direct&300 }} bie5782_1.0.zip bie5782-manual.pdf ==== Installation and loading of the package ==== Download the package and set your R working directory to the same directory as where you saved the package. Because we will not be using a repository we will have to supply an additional argument to the install.packages command as follows;

```
> install.packages(repos=NULL,"bie5782_1.0.zip")
Installing package(s) into 'C:/Program Files/R/R-2.15.0/library'
(as 'lib' is unspecified)
```

```
package ‘bie5782’ successfully unpacked and MD5 sums checked
```

The library can from now on be loaded like any other library with;

```
> library("bie5782")
Loading required package: Rcpp
```

==== Help and build in examples==== The package and function documentation is provided as a normal help page as for any other R function. Furthermore, the example as described in the documentation can be run as for any other R function. The following help pages and examples are defined;

```
?bie5782
?mmm
?performanceMultiplicationTest
example(mmm)
example(performanceMultiplicationTest)
```

The documentation looks as follows for the mmm function;

Michel's Matrix Multiplication with C++

Description

Performs matrix multiplication through C++ interface and code.

Usage

```
mmm(a,b)
```

Arguments

a an n x m matrix
b an m x p matrix

Examples

```
mmm.nrows = 5
mmm.ncols = 5

mmm.a <- matrix(nrow=mmm.nrows, ncol=mmm.ncols,runif(mmm.nrows*mmm.ncols))
mmm.b <- matrix(nrow=mmm.nrows, ncol=mmm.ncols,runif(mmm.nrows*mmm.ncols))

(mmm.a %*% mmm.b)

(mmm(mmm.a,mmm.b))
```

The documentation looks as follows for the mm function;

Performance graph

Description

Plots a performance graph of a function compared to the basic matrix multiplication function.

Usage

```
performanceMultiplicationTest(func,nsize,nrep)
```

Arguments

`func`	a function that accepts two matrices as arguments, func(x,y)
`nsize`	the range of matrix sizes under test, defaults to seq(10, 1000, 100)
`nrep`	the number of repetition for each measurement, defaults to 5

Examples

```
performanceMultiplicationTest(mmm,nsize=seq(10, 1000, 100),nrep=5)
```

==== Measured results ==== Luckily enough for everyone that is using R my function performs worse than the default matrix multiplication function already present in R. To measure this performance I have written a function that measure the amount of time that it takes to do such a multiplication. The function is named performanceMultiplicationTest and accept as arguments the function under test, the size of the nxn matrix that need to be tested and amount of repetitions for each measurement. The last to make sure we have accurate measurements and that there is not on rogue measurement due to increased CPU usage by other applications. As a result the function produced the following graphs which clearly shows the performance of my C function in red and the default matrix multiplication function in blue.

Functions

The help describes the following two functions; mmm() in C++ and performanceMultiplicationTest() in R.

```
performanceMultiplicationTest <- function(func,nsize=seq(10, 1000,
100),nrep=5)
{
  # initialize vectors to empty vectors
  time.base = c()
  time.func = c()
  time.base.mean = c()
  time.func.mean = c()
```

```
  # repeat measurements for each size in steps given by argument nsize
  for (i in 1:length(nsize)){
    # create two random nxn matrices where n is the size for this iteration
    a <- matrix(nrow=nsize[i], ncol=nsize[i],runif(nsize[i]^2))
    b <- matrix(nrow=nsize[i], ncol=nsize[i],runif(nsize[i]^2))

    # clear the mean time for this matrix size
    temp.base.mean = c()
    temp.func.mean = c()

    # repeat the measurements for nrep times for this matrix size
    for (j in 1:nrep){
      # do a measurement against the default matrix multiplier
      t1 = system.time(a%*%b)[1]
      # do a measurement with a new function
      t2 = system.time(func(a,b))[1]

      # add the measurement to a vector of all measurements
      time.base = c(time.base,t1)
      # add the measurement to a vector of only the measurement for this
matrix size
      temp.base.mean = c(temp.base.mean,t1)

      # add the measurement to a vector of all measurements
      time.func = c(time.func,t2)
      # add the measurement to a vector of only the measurement for this
matrix size
      temp.func.mean = c(temp.func.mean,t2)
    }
    # now average the results for this matrix size so we can plot a line of
avg. measurement
    time.base.mean = c(time.base.mean,mean(temp.base.mean))
    time.func.mean = c(time.func.mean,mean(temp.func.mean))
  }

  # create a plot where the ceiling is the max of the measured values
  plot(rep(nsize,each=nrep),time.base,col="blue",
       ylim=c(0,ceiling(max(time.base,time.func))),
       main="Performance graph",
       xlab="Matrix size",
       ylab="Time (s)",
       ann=FALSE,
       axes=FALSE)
  # create an x-axis where the labels are set to the matrix size in nxn
notation
  axis(1, at=rep(nsize,each=nrep),
lab=paste(rep(nsize,each=nrep),"x",rep(nsize,each=nrep)),pos=0,las=1)
  # create an y-axis where the labels are set to the measure time in seconds
  axis(2, at=0:ceiling(max(time.base,time.func)),pos=0)
  # plot the axis lines themselves
```

```
  abline(v=0, h=0)

  # plot the average measurement time measured for the default function
  lines(nsize,time.base.mean,type="l",col="blue")
  # plot the points as measured for the new function
  points(rep(nsize,each=nrep),time.func,col="red",pch=22)
  # plot the line with the average measurements as measured for the new
function
  lines(nsize,time.func.mean,type="l",col="red")
  # add a legend at the top left corner
  legend(1, max(time.base,time.func), c("base","function"), cex=0.8,
         col=c("blue","red"), pch=21:22, lty=1:2);
  # add further information to the graph
  title("Performance", xlab="Matrix size",ylab="Time (s)")
  TRUE
}
```

```
#include <R.h>
#include <Rdefines.h>

extern "C" {

    SEXP rcpp_matrix_multiplication_f(SEXP a, SEXP b){

        int arows,acols,bcols;
        double *xa,*xb,*rans,temp;
        SEXP ans;

        PROTECT(a = AS_NUMERIC(a));
        PROTECT(b = AS_NUMERIC(b));

        arows = INTEGER(GET_DIM(a))[0];
        acols = INTEGER(GET_DIM(a))[1];
        bcols = INTEGER(GET_DIM(b))[1];

        PROTECT(ans = allocMatrix(REALSXP, arows, bcols));

        xa   = NUMERIC_POINTER(a);
        xb   = NUMERIC_POINTER(b);
        rans = NUMERIC_POINTER(ans);

        for ( int i = 0; i < arows; i++ )
        {
            for ( int j = 0; j < bcols; j++ )
            {
                temp = 0;
                for ( int k = 0; k < acols; k++ )
                {
                    temp = temp + (xa[i+k*bcols] * xb[k+acols*j]);
                }
                rans[i+j*bcols] = temp;
```

```
            }
        }
        UNPROTECT(3) ;
        return ans;
    }
}
```

Proposal final work

Proposal A:

The speedup of matrix computation

The proposal is to create a package with a function to possibly speed up simple matrix multiplications by utilizing the rcpp package[1].

[1] http://dirk.eddelbuettel.com/code/rcpp.html

Proposta B:

The speedup of matrix computation in combination with a benchmarking function

Same as A but in addition a function will be written that calls the cpp function of A with different matrix sizes and compares the execution time with the standard library version.

Comentários da proposta (Leo)

Apparently you are very familiar with such matrix analysis, but be careful not to make things too hard for you. Perhaps it is better to stick with a function, and not with a whole package.

I missed more information about the input, possible arguments (such as what kind of matrix operation the user wishes to perform), and outputs (of course the output is not just a faster analysis, but the resut of the matrix operation in the apropriate format).

Comentários da proposta (Diogo)

This is a nice thing to learn, especially if you mess with complicated (or slow) analysis. But improving on the basic matrix operations can be very challenging. They already are implemented in C or FORTRAN. If you plan to use threading inside C++, even getting a linear n core n speed up on matrix multiplication is rather challenging (I know it took me a while...). If you're good with pointers and subtle allocation problems, go ahead! But this will be much more a C++ project than an R project.