Computational Efficiency: An adaptation of the Forward Regression and PC algorithms from R to C++

Marios Dimitriadis

Dissertation submitted to the department of Computer Science of the University of Crete, in partial fulfillment of the requirements for the BSc degree.

Principal Advisor: Ioannis Tsamardinos
Secondary Supervisor: Michail Tsagris

June 2017

University of Crete

Acknowledgments: Special Account for Research Funds of the University of Crete, Department of Computer Science.

1. Introduction

The R programming language gives mainly statistically inclined programmers the ability to powerfully use the abstraction of mathematics and combine it with the convenience computing power can bring. Its history and community, along with its open source nature has allowed it to become the main choice for statistical analysis and data mining, and it is generally accepted that it can be successfully compared with other statistical tools (Burns, P., 2007).

As any widely used tool, R is also an object of criticism, and quite often the produced speed for certain areas/tasks of it is the main criticism received. There have been several attempts at patching or/and fixing the areas in which R is lacking in terms of speed: Packages are being offered publicly and predominantly on the Comprehensive R Archive Network (CRAN*), implementations of R with speed as a focus have and are being implemented (Talbot, J., DeVito, Z. and Hanrahan, P., 2012) (Neal, R.M.), guidelines, tips and a defined way of programming have been enforced, and in our case, the support of other programming languages through R has been sought.

This leads us to the programming language C++, machine code, and the performance driven discipline that they offer. C++ was designed as a general purpose programming language and with performance and efficiency as a goal. The now-well known initiative that brought the easiness with which the communication between R and C++ became possible, was first created by Dominick Samperi in 2006, and later adopted by Dirk Eddelbuettel in 2008, in the form of a package called Rcpp. The package, among others, aims at providing an interface for the communication between the two programming languages (Eddelbuettel, D., François, R., Allaire, J., Chambers, J., Bates, D. and Ushey, K., 2011).

The focus of this thesis, performance, was applied on two main functions: fs_reg, and pc_skel, along with their dependencies. The algorithm of fs_reg through forward selection determines variables in generalized linear models, which are based on the Bayesian Information Criterion (BIC), as well as a tolerance value provided by the user. The algorithm of pc_skel is broken into two stages: The first stage pre-computes the p-values, test statistics and degrees of freedom. The second stage is a materialization of the PC

* https://cran.r-project.org/

Algorithm as proposed by Spirtes et al (Spirtes, Peter, Clark N. Glymour, and Richard Scheines).

2. R to C++ and Back

The analysis of the programming language R, from a performance perspective, is a subject that can be divided under several subcategories, though many would argue on their number. For simplicity, we can categorize parts of R based on how the tokens of the language are handled, as well as, from the where and how they are retrieved. Looking at R's implementation, which is in C, Fortran, and in R itself, we can, for the purpose of this thesis, define two categories: Internal for tokens of the language that were written in R itself, and External for tokens of the language that were written in C or Fortran. (The terms "Internal" and "External" used here are unrelated to, and shouldn't be confused with, R's terms .Internal and .External respectively.)

The communication with any External tokens is materialized through interfaces that yield different or no overhead, and the evaluation of a function's tokens' arguments can differentiate based on the type of the function (Team, R.C., 2000).

On average, External tokens surpass Internal tokens by far in terms of performance, and therefore Internal tokens were the main target of the code that was to be replaced and be rewritten in C++, and through which performance improvements became visible. R's function "apply" is a notable example.

Aside from targeting areas of R that can on average be outperformed by their (output wise) equivalent version in C++, there's a more catholic set of variables that set the difference in terms of performance between the two languages. The main variable is the environment and the way in which R and C++ executables are run: R is a programming language for which, to the greatest extent, an interpreter is used, while C++ is a programming language for which a compiler is used. Interpreters allow for a smoother implementation and immediate execution, during which the instructions are often not directly (natively) executed by the host machine. Compilers allow optimizations during the compilation stage, and the produced file can be natively executed on the host machine after the compilation. The two latter very specific differences noted between the two methods signify the disadvantage of interpreters in terms of efficiency. The difference in speed could be tremendous between the two, or unnoticeable, though it is widely accepted that a difference of at least an order of magnitude will be observed. It should be noted here that this advantage comes at a cost, and the development speed and learning curve are usually the main concerns of users.

2.1 A Brief Look at the Efficiency of Programming Languages

Benchmarking the efficiency of a programming language can be considered to be dependent on many factors, nevertheless, below is a result*** of identical implementations of an algorithm in various programming languages, and an attempt was made to keep the implementations free from library, operating system calls, and advanced language feature differences. The implementations were run on a 300MHz Pentium processor. The operating system used was Debian GNU/Linux.

Programming Language	Computation Time in Seconds
FORTRAN, g77 V2.95.4	2.73
Ada 95, gnat V3.13p	2.73
C, hand optimized gcc V2.95.4	2.73
Java, gcj V3.0	3.03
D, gcc V4.0.3+	3.43
C, gcc V2.95.4	3.61
R*	3.69
Lisp**	4.69
Java, jikes V1.15 (bytecompiled)	8.23
FORTH,** Gforth 0.6.1	27.26
Perl V5.6.1 (nativecompiled)	367.23
Python V2.1.2 (interpreted)	505.50
Perl V5.6.1 (bytecompiled)	515.04
Ruby (interpreted)	1074.52
R V1.5.1 (interpreted)	5662.64

* translated to lisp using R2cl v0.1 and compiled with cmucl

** CMU Common Lisp V3.0.8 18c+, build 3030

*** The source can be found on: http://dan.corlan.net/bench.html

2.2 R's Internals

The communication between R and mainly C becomes possible through definitions of functions in C, and their invocation in R. The basic method provided is a C Application Programming Interface (API). C is responsible for receiving, recognizing and handling R objects that are stored in a common type called SEXP, or S-expression. An S-expression, is the type that R will expect to receive from C.

Under the surface of every C function, in order for C to recognize the parameters received, automatic conversions are being performed between C and R, and the responsibility of handling R's garbage collector is fully transferred to C after its invocation (Team, R.C., 2000). Due to the processing necessary after every exchange between (the) two languages, an overhead is being accumulated. It will be noticed that the functions that were rewritten in C++ didn't only target the External parts of R for this very reason.

3. Rcpp and RcppArmadillo

The communication between two programming languages is rarely, by default, an easy task. In some cases, like in R's, an API is provided in order to ease the communication procedure, but as described in the previous chapter, it still appears to include enough foreign details to alienate people from performing it. Taking into account that R is a programming language that has as audience users from different backgrounds (and not mainly of a Computer Science background), the prior mentioned effect we can assume is more prevalent.

Rcpp, as it was briefly mentioned in the Introduction, is a library that focuses on patching up, and hiding from the end user the low level details that need to be handled by default when using the API provided by R, in order to communicate with C, or with C++ in the case of Rcpp. Such low level details include the lack of need to handle R's garbage collector, and an automated interfacing and matching of R's data types to C++'s (through custom classes) and back (Eddelbuettel, D., François, R., Allaire, J., Chambers, J., Bates, D. and Ushey, K., 2011). The ability or the jump from C to C++, from a programming language perspective, gives users theoretically a more high level view, and this is through Rcpp itself and the inherent differences between the two languages independently.

Rcpp acts as a bridge between R and C++. It allows C++ to be called directly from R, but as a library it doesn't provide a means to cover the fundamental differences between the two languages, and more specifically, the ease of use R provides and the functions (of mathematical and statistical nature, among others) that are available as part of the core language in R. Hence, a rewrite of an R codebase in C++ is not only time consuming but in certain cases also performance-wise inadvisable.

3.1 Tests on Matrix Multiplication

An example that demonstrates the above is the matrix multiplication operation. The tests were run on an Intel i3 4005U 1.7GHz processor. Figures 1 and 2 illustrate the box plots of the tests. The operating system used was Fedora 25 GNU/Linux.

3.1.1 Input Size 100 x 100

Input:

```
> nvalues <- 3; nsamples <- nrows <- 100; nvars <- ncols <- 100
> ds <- matrix(rnorm(nsamples * nvars), nrow = nrows,
            ncol = ncols)
> ds2 <- matrix(rnorm(nsamples * nvars), nrow = nrows,
            ncol = ncols)
```

Execution:

```
> microbenchmark(ds %*% ds2, cust_mult(ds, ds2), times = 100)
```

Results (in microseconds):

```
expr        min         mean        median      max
R           169.522     177.3612    172.975     214.647
C++         33026.755   33318.3950  33163.742   34362.900
```

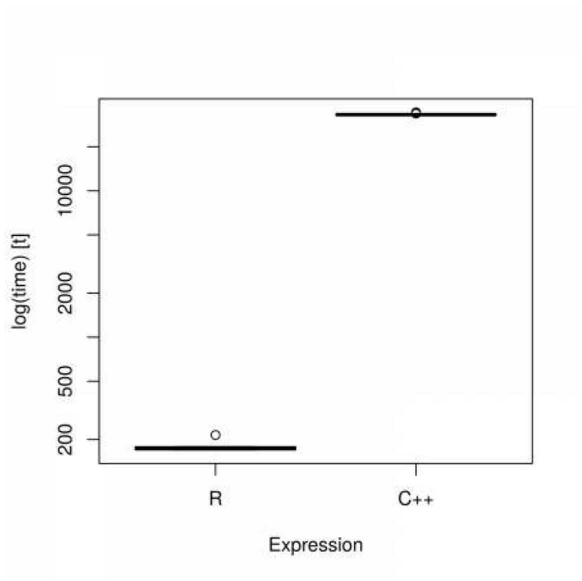

Figure 1: Log of execution time between R's default matrix multiplication and a custom matrix multiplication in C++.
Input size: 100x100

3.1.2 Input Size 1000 x 1000

Input:

```
> nvalues <- 9; nsamples <- nrows <- 1000; nvars <- ncols <- 1000
> ds <- matrix(rnorm(nsamples * nvars), nrow = nrows,
            ncol = ncols)
> ds2 <- matrix(rnorm(nsamples * nvars), nrow = nrows,
            ncol = ncols)
```

Execution:

```
> microbenchmark(ds %*% ds2, cust_mult(ds, ds2), times = 100)
```

Results (in milliseconds):

```
expr        min         mean        median      max
R           92.424      99.953      95.852      137.737
C++         37677.233   39050.561   38791.995   40720.065
```

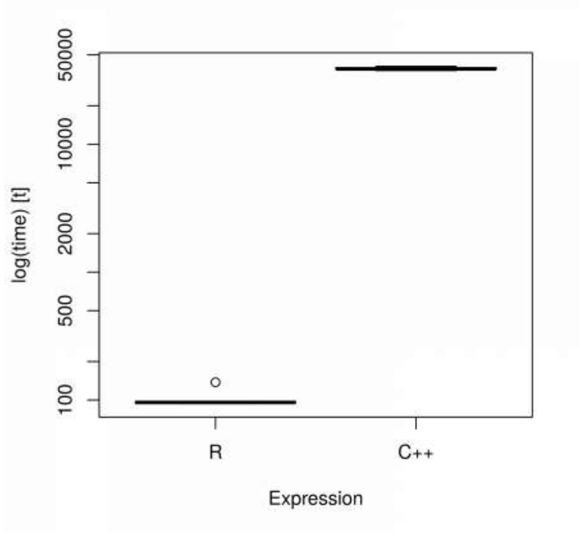

Figure 2: Log of execution time between R's default matrix multiplication and a custom matrix multiplication in C++.
Input size: 1000x1000

3.1.3 Matrix Multiplication Code in R and C++

Code in R:
```
ds %*% ds2
```

Code in C++:
```
Rcpp::NumericMatrix cust_mult(Rcpp::NumericMatrix lh,
            Rcpp::NumericMatrix rh) {
      int nrows = lh.nrow(); int ncols;
      lh.ncol() > rh.ncol() ?
            ncols = rh.ncol() : ncols = lh.ncol();
      Rcpp::NumericMatrix res_mat(nrows, ncols);
      for (int i = 0; i < lh.nrow(); i++) {
            for (int j = 0; j < rh.ncol(); j++) {
                  for (int k = 0; k < lh.ncol(); k++) {
                        res_mat(i, j) += lh(i, k) * rh(k, j);
                  }
            }
      }
      return res_mat;
}
```

Note: This by no means refers to an inability of C++ to cover the same ground in the same computing speed as R when it comes to this very function, but rather of a need of a more specialized approach to designing it from both a mathematical and development perspective. However, it does show that using the default and intuitive way to replicate the usage of certain operations and functions would lead to a significant degradation in performance.

3.2 RcppArmadillo's Solution

RcppArmadillo is a package that attempts to cover the usability gap between R and C++. The library in its original form is called Armadillo, was created by Conrad Sanderson, and was initially written as a C++ linear algebra library, while aiming for a balance between ease of use and speed (Sanderson, C., 2010). It targets fields from Machine Learning to Bioinformatics, and more generally, Statistics.

In order to make the types in Rcpp and Armadillo compatible, as well as making Armadillo compatible with R, RcppArmadillo was created. This compatibility and the maintenance of the library has been brought forth by Romain Francois, Dirk Eddelbuettel and Doug Bates while using Rcpp and building on top of the Armadillo library (Eddelbuettel, D. and Sanderson, C., 2014).

RcppArmadillo offers a wealth of linear algebra and statistical operations and functions, and in most cases those additions offer a performance that is close to the performance provided by R's operations and functions. RcppArmadillo was mainly used in this thesis for operations upon matrices, including, but not limited to, multiplication and calculation.

3.3 Tests on Matrix Multiplication and the Solve Operation

Below are the test details demonstrating the differences between R's matrix multiplication, the custom matrix multiplication, and Armadillo's matrix multiplication, as well as, of the solve function between R's and RcppArmadillo's. Figures 3 and 4 illustrate the box plots of the matrix multiplication tests. Figures 5 and 6 illustrate the box plots of the solve operation tests. The tests were run on an Intel i3 4005U 1.7GHz processor. The operating system used was Fedora 25 GNU/Linux.

3.3.1 Matrix Multiplication with Input Size 100 x 100

Input:
```
> nvalues <- 3; nsamples <- nrows <- 100; nvars <- ncols <- 100
> ds <- matrix(rnorm(nsamples * nvars), nrow = nrows,
          ncol = ncols)
> ds2 <- matrix(rnorm(nsamples * nvars), nrow = nrows,
          ncol = ncols)
```

Execution:

```
> microbenchmark(ds %*% ds2, cust_mult(ds, ds2),
            arma_mult(ds, ds2), times = 100)
```

Results (in microseconds):

```
expr        min         mean        median      max
R           155.680     172.6007    165.198     247.527
C++         33434.058   33709.749   33736.896   33899.151
C++/Arma    172.654     189.006     188.384     213.666
```

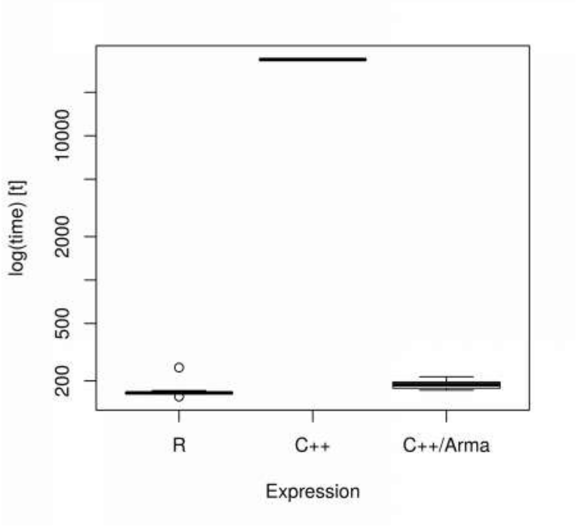

Figure 3: Log of execution time between R's default matrix multiplication, a custom matrix multiplication in C++, and Armadillo's default matrix multiplication.
Input size: 100x100

3.3.2 Matrix Multiplication with Input Size 1000 x 1000

Input:

```
> nvalues <- 9; nsamples <- nrows <- 1000; nvars <- ncols <- 1000
> ds <- matrix(rnorm(nsamples * nvars), nrow = nrows,
            ncol = ncols)
> ds2 <- matrix(rnorm(nsamples * nvars), nrow = nrows,
            ncol = ncols)
```

Execution:

```
> microbenchmark(ds %*% ds2, cust_mult(ds, ds2),
            arma_mult(ds, ds2), times = 100)
```

Results (in milliseconds):

```
expr        min         mean        median      max
R           94.017      98.433      97.931      105.780
C++         37273.166   38147.909   37892.192   39254.230
C++/Arma    102.824     108.757     108.254     120.131
```

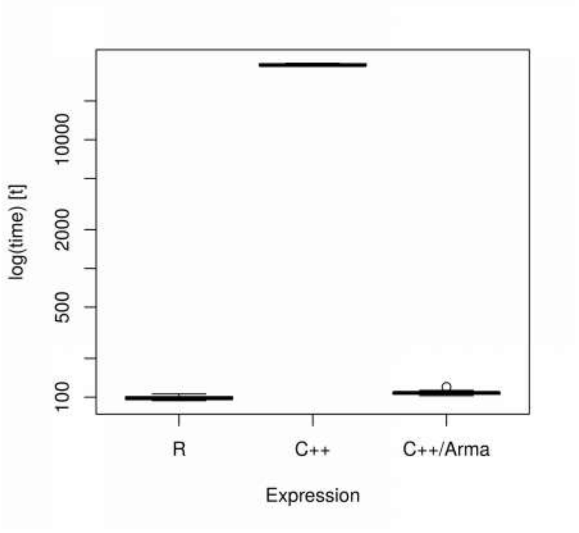

Figure 4: Log of execution time between R's default matrix multiplication, a custom matrix multiplication in C++, and Armadillo's default matrix multiplication.
Input size: 1000x1000

3.3.3 Matrix Multiplication Code in R and C++

Code in R:

```
ds %*% ds2
```

Code in C++ (custom matrix multiplication):

```
Rcpp::NumericMatrix cust_mult(Rcpp::NumericMatrix lh,
            Rcpp::NumericMatrix rh) {
      int nrows = lh.nrow(); int ncols;
      lh.ncol() > rh.ncol() ?
            ncols = rh.ncol() : ncols = lh.ncol();
      Rcpp::NumericMatrix res_mat(nrows, ncols);
      for (int i = 0; i < lh.nrow(); i++) {
            for (int j = 0; j < rh.ncol(); j++) {
                  for (int k = 0; k < lh.ncol(); k++) {
                        res_mat(i, j) += lh(i, k) * rh(k, j);
                  }
            }
      }
      return res_mat;
}
```

Code in C++ (RcppArmadillo):

```
arma::mat arma_mult(arma::mat lh, arma::mat rh) {
      return lh * rh;
}
```

3.3.4 Solve Operation with Input Size 100 x 100

Input:

```
> nvalues <- 3; nsamples <- nrows <- 100; nvars <- ncols <- 100
> ds <- matrix(rnorm(nsamples * nvars), nrow = nrows,
            ncol = ncols)
> ds2 <- matrix(rnorm(nsamples * nvars), nrow = nrows,
            ncol = ncols)
```

Execution:

```
> microbenchmark(solve(ds, ds2), arma_solve(ds, ds2))
```

Results (in microseconds):

```
expr        min         mean        median      max
R           593.394     676.553     651.606     2009.247
C++/Arma    9727.705    10293.140   10273.171   11245.191
```

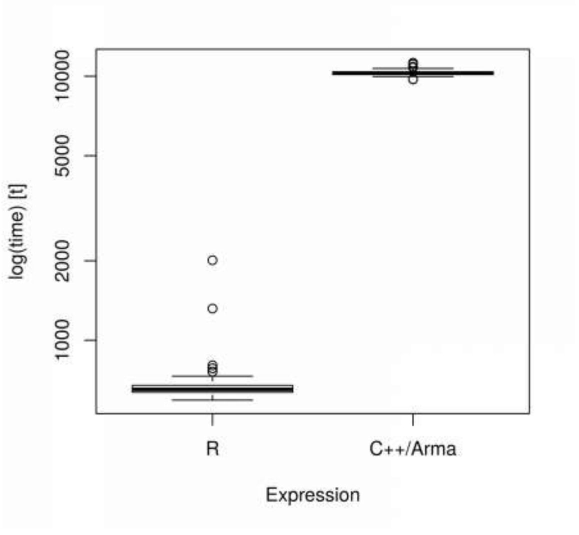

Figure 5: Log of execution time between R's default solve function and Armadillo's default solve function.
Input size: 100x100

3.3.5 Solve Operation with Input Size 1000 x 1000

Input:

```
> nvalues <- 9; nsamples <- nrows <- 1000; nvars <- ncols <- 1000
> ds <- matrix(rnorm(nsamples * nvars), nrow = nrows,
            ncol = ncols)
> ds2 <- matrix(rnorm(nsamples * nvars), nrow = nrows,
            ncol = ncols)
```

Execution:

```
> microbenchmark(solve(ds, ds2), arma_solve(ds, ds2), times = 100)
```

Results (in milliseconds):

```
expr        min         mean        median      max
R           173.711     182.589     178.656     211.563
C++/Arma    11127.102   11388.493   11281.386   11972.58
```

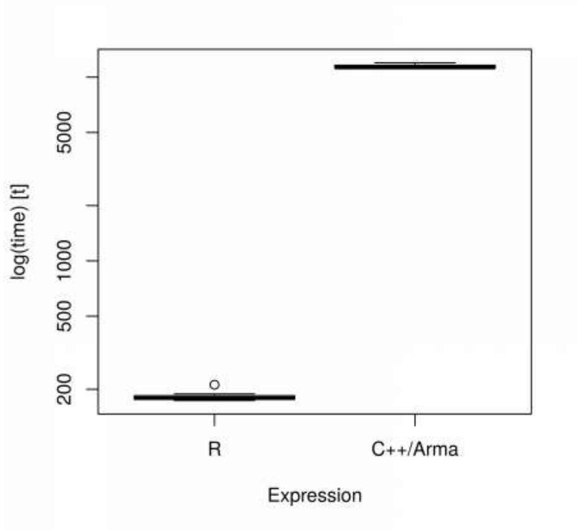

Figure 6: Log of execution time between R's default solve function and Armadillo's default solve function.
Input size: 1000x1000

3.3.6 Solve Operation Code in R and C++

Code in R:

```
solve(ds, ds2)
```

Code in C++ (RcppArmadillo):

```
arma::mat solve_mat(arma::mat lh, arma::mat rh) {
      return arma::solve(lh, rh);
}
```

4. Forward Regression

Forward Regression was the first algorithm that was implemented* in C++ for this thesis. The algorithm performs variable selection in regression models through forward selection. Forward selection begins with no variables in the model, and during each step of the algorithm testing of variables is performed using the chosen model. The variable whose addition gives the most statistically significant improvement to the overall result is chosen. The algorithm ends when no such variables can be found.

The signature for the function is the following:

In R:

```
fs.reg(y, ds, sig, tol, type)
```

In C++:

```
Rcpp::NumericMatrix fs_reg(Rcpp::IntegerVector y,
      Rcpp::NumericMatrix ds, const double sig, const double tol,
      const std::string type);
```

4.1 An Explanation of the Parameters and Usage

The return value of this algorithm is a matrix which depicts any correlation found through forward regression among the variables in the matrix (ds) that was passed to it as an argument. The correlation itself, and whether one is found is dependent on four parameters: y, sig, tol, and type. The first parameter, y, is a vector that holds the objects towards which the correlation is targeted at. The second parameter, sig, stands for statistical significance and it's used as a base for the calculations and their acceptance. The third parameter, tol, stands for tolerance, and holds the value that will weigh the importance in terms of statistics, and will be directly compared with the calculated Bayesian Information Criterion (BIC). The fourth parameter, type, defines the regression model to be used which can be, at the time of writing, either "logistic" or "poisson".

* Available on https://cran.r-project.org/web/packages/Rfast/index.html

4.2 Tests on Forward Regression

Below are the test details that show the performance gain. Figures 7 and 8 illustrate the box plots indicating execution with type "logistic". Figures 9 and 10 illustrate the box plots indicating execution with type "poisson". The tests were run on an Intel i3 4005U 1.7GHz processor. The operating system used was Fedora 25 GNU/Linux.

4.2.1 Logistic Regression with Input size 100 x 100

Input:

```
> ds <- matrix(rnorm(100 * 100), ncol = 100)
> y1 <- rbinom(100, 1, 0.6)
> sig <- 0.05
> tol <- 2
```

Execution:

```
> microbenchmark(fs.reg(y1, ds, threshold = sig, tol = 2,
            test = "testIndLogistic"),
            fs_reg(y1, ds, sig, tol, "logistic"),
            times = 100)
```

Results (in milliseconds):

```
expr        min         mean        median      max
R           251.102     260.543     255.128     305.472
C++         10.453      10.608      10.584      10.908
```

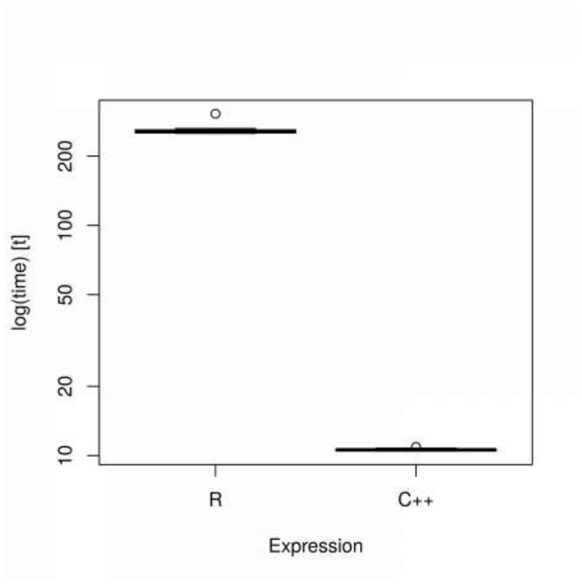

Figure 7: Log of execution time between R's forward regression function and C++'s forward regression function. The type used was "logistic". Input size: 100x100

4.2.2 Logistic Regression with Input Size 1000 x 1000

Input:

```
> ds <- matrix(rnorm(1000 * 1000), ncol = 1000)
> y1 <- rbinom(1000, 1, 0.6)
> sig <- 0.05
> tol <- 2
```

Execution:

```
> microbenchmark(fs.reg(y1, ds, threshold = sig, tol = 2,
            test = "testIndLogistic"),
            fs_reg(y1, ds, sig, tol, "logistic"),
            times = 100)
```

Results (in seconds):

```
expr        min         mean        median      max
R           19.680      19.913      19.913      20.193
C++         1.478       1.489       1.484       1.511
```

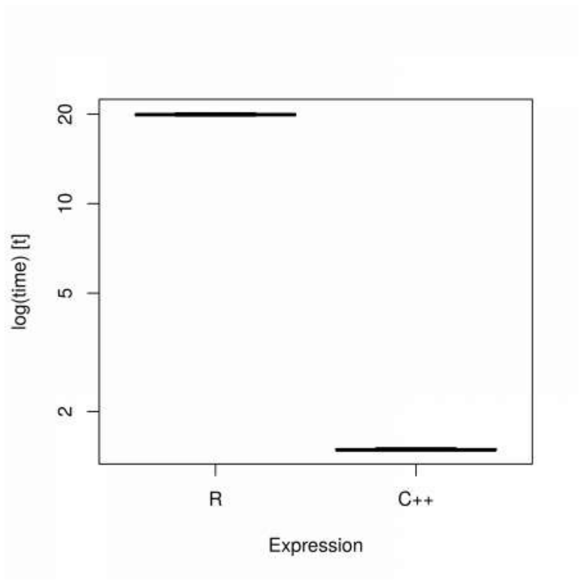

Figure 8: Log of execution time between R's forward regression function and C++'s forward regression function. The type used was "logistic". Input size: 1000x1000

4.2.3 Poisson Regression with Input Size 100 x 100

Input:
```
> ds <- matrix(rnorm(100 * 100), ncol = 100)
> y2 <- rpois(100, 1)
> sig <- 0.05
> tol <- 2
```

Execution:
```
> microbenchmark(fs.reg(y2, ds, threshold = sig, tol = 2,
            test = "testIndPois"),
            fs_reg(y2, ds, sig, tol, "poisson"),
            times = 100)
```

Results (in milliseconds):
```
expr        min         mean        median      max
R           301.531     306.615     305.708     315.627
C++         8.570       8.716       8.707       8.860
```

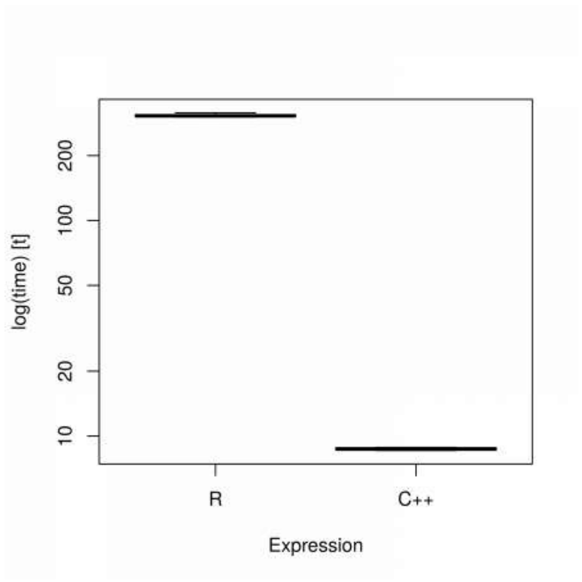

Figure 9: Log of execution time between R's forward regression function and C++'s forward regression function. The type used was "poisson". Input size: 100x100

4.2.4 Poisson Regression with Input Size 1000 x 1000

Input:

```
> ds <- matrix(rnorm(1000 * 1000), ncol = 1000)
> y2 <- rpois(1000, 1)
> sig <- 0.05
> tol <- 2
```

Execution:

```
> microbenchmark(fs.reg(y2, ds, threshold = sig, tol = 2,
            test = "testIndPois"),
            fs_reg(y2, ds, sig, tol, "poisson"),
            times = 100)
```

Results (in milliseconds):

```
expr        min         mean        median      max
R           7403.640    7540.728    7529.355    7684.168
C++         461.142     471.792     467.857     496.031
```

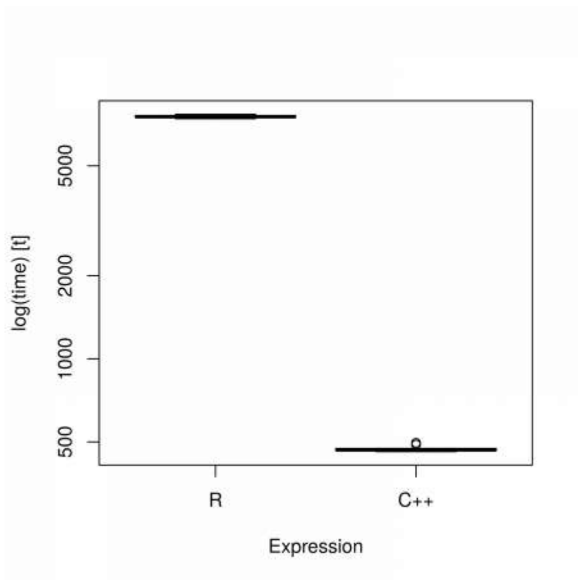

Figure 10: Log of execution time between R's forward regression function and C++'s forward regression function. The type used was "poisson". Input size: 1000x1000

5. PC Algorithm

PC Algorithm was the second algorithm that was implemented* in C++ for this thesis. The algorithm begins by forming a complete undirected graph based on the dataset provided, and during each step after the initial formation of the graph edges are dynamically being removed. The dynamism is dependent upon the step that is at-that time being processed: During the first step, edges with zero order conditional independence relations are removed, during the second step, edges with first order independence relations are removed, etc. The algorithm ends upon encountering a pair that has edges fewer than the steps performed up to that point.

The signature for the function is the following:

In R:
pc.skel(ds, method, sig, r);

In C++:
```
Rcpp::List pc_skel(arma::mat ds, const std::string
        method, const double sig, const in r);
```

* Available on https://cran.r-project.org/web/packages/Rfast/index.html

5.1 An Explanation of the Parameters and Usage

The return value of this algorithm is a list that depicts the most statistically associated variables of the matrix (ds) that was passed to it as an argument, as per Spirtes et al. (2000). The statistical association itself, and whether one is found is dependent upon three parameters: method, sig, and r. The first parameter, method, is a string that defines the correlation method to be used: "pearson", "spearman", or "cat". The first two methods are correlation coefficients referring to continuous data. In specific, "spearman" should be used when outliers are present in the data. The third option stands for categorical data in which case the G-square test of independence is used. The second parameter, sig, stands for statistical significance and it's used as a base for the calculations and their acceptance. The third parameter, r, defines the number of permutations.

5.2 Tests on PC Algorithm

Below are the test details that show the performance gain. Figures 11 and 12 illustrate the box plots indicating execution with method "pearson". Figures 13 and 14 illustrate the box plots indicating execution with method "spearman". Figures 15 and 16 illustrate the box plots indicating execution with method "cat". The tests were run on an Intel i3 4005U 1.7GHz processor. The operating system used was Fedora 25 GNU/Linux.

5.2.1 Pearson Method with Input Size 1000 x 50

Input:
```
> ds <- matrix(runif(1000 * 50, 1, 100), nrow = 1000)
```

Execution:
```
> microbenchmark(pc.skel(ds, method = "pearson",
            alpha = 0.05, R = 1),
            pc_skel(ds, "pearson", 0.05, 1), times = 100)
```

Results (in milliseconds):
```
expr        min         mean        median      max
R           263.038     271.481     268.333     299.122
C++         7.231       7.843       7.414       12.097
```

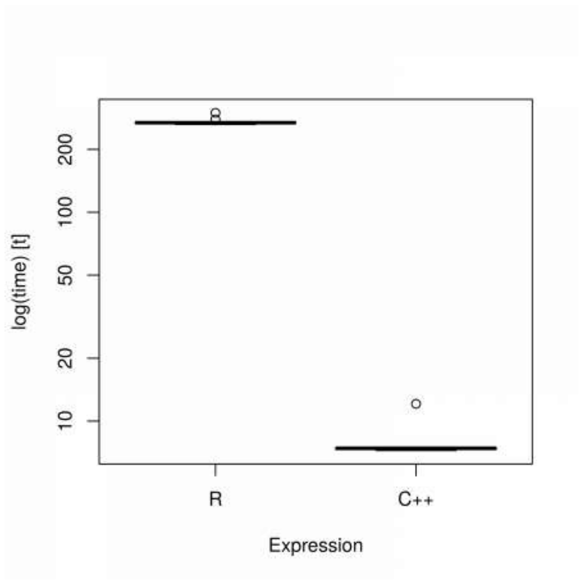

Figure 11: Log of execution time between R's PC algorithm's function and C++'s PC algorithm's function. The method used was "pearson".
Input size: 1000x50

5.2.2 Pearson Method with Input Size 1000 x 100

Input:
```
> ds <- matrix(runif(1000 * 100, 1, 100), nrow = 1000)
```

Execution:
```
> microbenchmark(pc.skel(ds, method = "pearson",
            alpha = 0.05, R = 1),
            pc_skel(ds, "pearson", 0.05, 1), times = 100)
```

Results (in milliseconds):
```
expr        min         mean        median      max
R           3898.839    3980.430    3950.838    4286.617
C++         111.913     116.038     114.173     125.0161
```

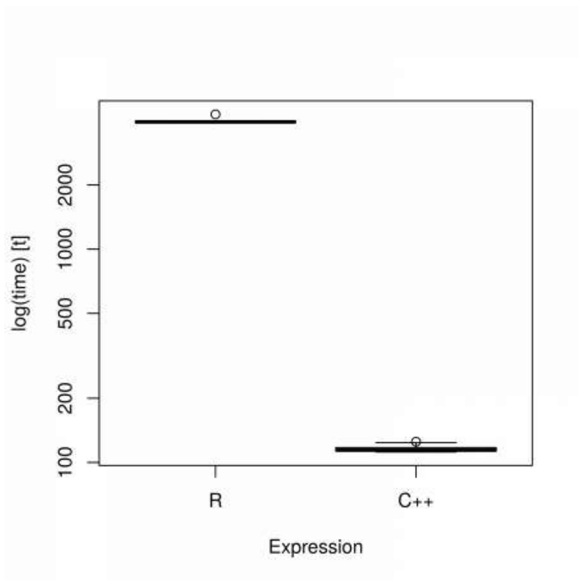

Figure 12: Log of execution time between R's PC algorithm's function and C++'s PC algorithm's function. The method used was "pearson".
Input size: 1000x100

5.2.3 Spearman Method with Input Size 1000 x 50

Input:
```
> ds <- matrix(runif(1000 * 50, 1, 100), nrow = 1000)
```

Execution:
```
> microbenchmark(pc.skel(ds, method = "spearman",
            alpha = 0.05, R = 1),
            pc_skel(ds, "spearman", 0.05, 1), times = 100)
```

Results (in milliseconds):
```
expr        min         mean        median      max
R           145.296     149.698     151.178     153.827
C++         43.469      45.459      45.712      48.276
```

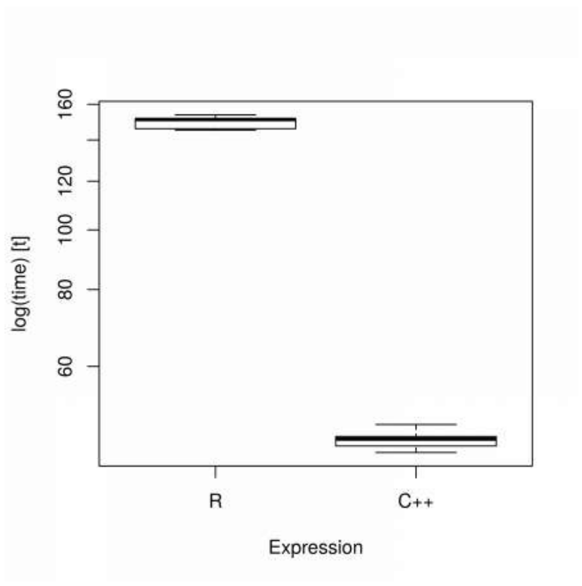

Figure 13: Log of execution time between R's PC algorithm's function and C++'s PC algorithm's function. The method used was "spearman".
Input size: 1000x50

5.2.4 Spearman Method with Input Size 1000 x 100

Input:
```
> ds <- matrix(runif(1000 * 100, 1, 100), nrow = 1000)
```

Execution:
```
> microbenchmark(pc.skel(ds, method = "spearman",
            alpha = 0.05, R = 1),
            pc_skel(ds, "spearman", 0.05, 1), times = 100)
```

Results (in milliseconds):
```
expr        min         mean        median      max
R           1705.038    1747.954    1720.952    1910.737
C++         127.147     129.7823    129.649     133.988
```

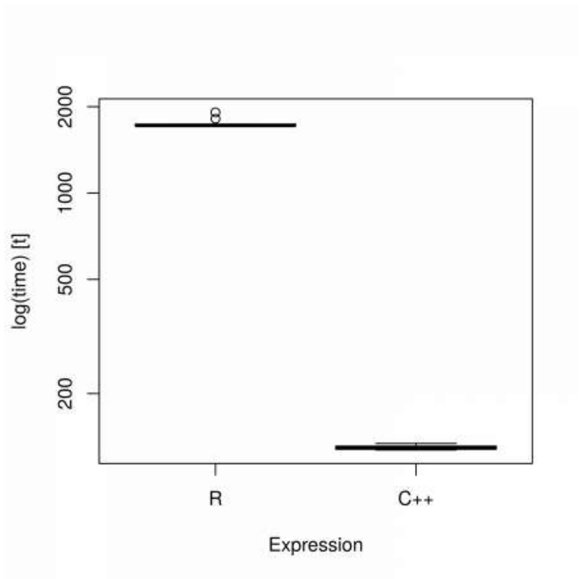

Figure 14: Log of execution time between R's PC algorithm's function and C++'s PC algorithm's function. The method used was "spearman".
Input size: 1000x100

5.2.5 Cat method with Input Size 1000 x 50

Input:

```
> nvalues <- 3
> nsamples <- nrows <- 1000
> nvars <- ncols <- 50
> ds <- matrix(sample(0:(nvalues - 1), replace = TRUE,
            nvars * nsamples), nrows, ncols)
```

Execution:

```
> microbenchmark(pc.skel(ds, method = "cat",
            alpha = 0.05, R = 1),
            pc_skel(ds, "cat", 0.05, 1), times = 100)
```

Results (in milliseconds):

```
expr        min         mean        median      max
R           69.074      71.126      71.417      71.887
C++         8.747       8.916       8.896       9.119
```

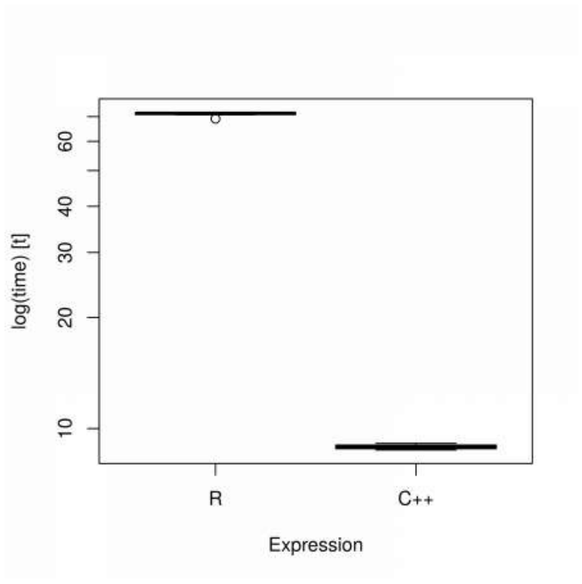

Figure 15: Log of execution time between R's PC algorithm's function and C++'s PC algorithm's function. The method used was "cat".
Input size: 1000x50

5.2.6 Cat Method with Input Size 1000 x 100

Input:
```
> nvalues <- 3
> nsamples <- nrows <- 1000
> nvars <- ncols <- 100
> ds <- matrix(sample(0:(nvalues - 1), replace = TRUE,
            nvars * nsamples), nrows, ncols)
```

Execution:
```
> microbenchmark(pc.skel(ds, method = "cat",
            alpha = 0.05, R = 1),
            pc_skel(ds, "cat", 0.05, 1), times = 100)
```

Results (in milliseconds):
```
expr        min         mean        median      max
R           360.736     415.825     411.731     503.019
C++         35.621      38.836      36.530      60.767
```

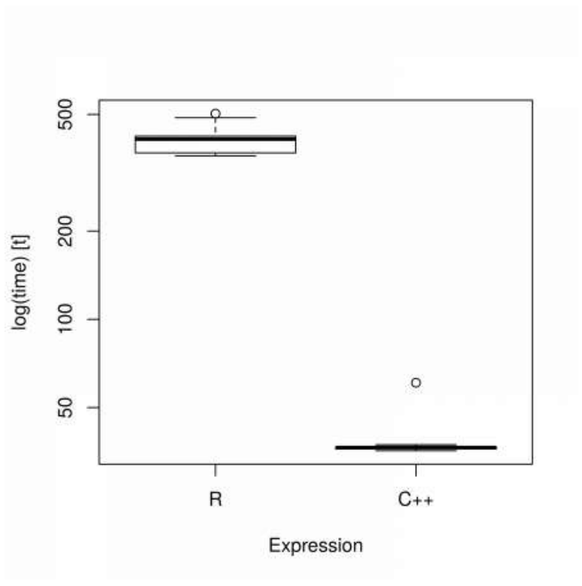

Figure 16: Log of execution time between R's PC algorithm's function and C++'s PC algorithm's function. The method used was "cat".

Input size: 1000x100

References

Burns, P., 2007. R relative to statistical packages: comment 1 on technical report number 1 (Version 1.0) strategically using general purpose statistics packages: A look at Stata, SAS and SPSS. *UCLA Technical Report Series*.

Talbot, J., DeVito, Z. and Hanrahan, P., 2012, September. Riposte: a trace-driven compiler and parallel VM for vector code in R. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques* (pp. 43-52). ACM.

Neal, R.M., Speed Improvements in pqR: Current Status and Future Plans. (https://www.r-project.org/conferences/DSC-2014/slides/Neal_talk_1.pdf)

Eddelbuettel, D., François, R., Allaire, J., Chambers, J., Bates, D. and Ushey, K., 2011. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, *40*(8), pp.1-18.

Spirtes, Peter, Clark N. Glymour, and Richard Scheines. *Causation, prediction, and search*. MIT press, 2000.

Team, R.C., 2000. R language definition. *Vienna, Austria: R foundation for statistical computing*.

Sanderson, C., 2010. *Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments* (p. 84). Technical report, NICTA.

Eddelbuettel, D. and Sanderson, C., 2014. RcppArmadillo: Accelerating R with high-performance C++ linear algebra. *Computational Statistics & Data Analysis*, *71*, pp.1054-1063.