First year project
Project 99: Who's Julia?

Group: 99a
Simon Lehmann Knudsen, simkn15
Sonni Hedelund Jensen, sonje15
Asbjørn Mansa Jensen, asjen15
FF501

May 29, 2016

1 Summary

Julia is a new programming language. Through tests and benchmarking it is found that * *

2 Preface

This report has been written in May 2016 for a first year project in a Bachelor Degree of Computer Science. We thank our supervisor Michal Kotrbčík for his help and time. In this project we have been doing benchmarks to find out if Julia is as good as the developers claims it to be.

Contents

3 Introduction

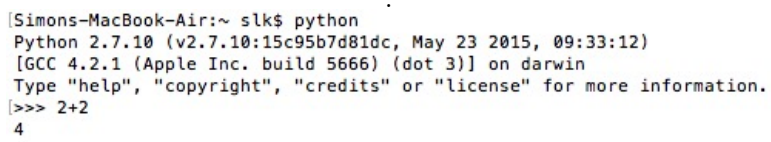
Java, C, C++, C# and Python are on top five of the most used programming languages. There are hundreds of languages, but not minding the hard competition new languages are still created with the thought of doing better. Julia is a new programming language which has been developed to contain other languages best features.

The purpose of this report is to find out if the programming language Julia has a chance in the marked and if it is worth changing to. The languages Java, C++, Python and Julia will be compared on syntax and time performance.

4 Julia

Julia is a programming language which has been under development since 2009. First released in 2012 and the newest stable version of Julia is version 0.4.5. The language has been created because the developers wanted a language with all the features they like from different languages. The developers also wanted the language to be open source, which means everybody can read how it is made and change the language. Julia shares many similarities with python. One of the idea was to make the language as simple, readable and easy to learn as possible. The language is made for high performance and scientific computations while still supporting general purpose programming. Programming in Julia can be done in the terminal like python with interactive mode:

(a) Julia



(b) Python

Figure 1: Julia in terminal

Writing longer programs in the terminal might not be the preferred method. The text editor Atom supports the Julia language. Atom has a package, **uber-juno**, which sets up Atom to act like an IDE, integrated development environment for Julia.

4.1 Syntax

Make examples of syntax between all the languages here.

```
1   Julia:     Python:     Java:           c++:
2   a = 10     a = 10      int a = 10;     int a = 10;
```

Figure 2: Declaring variables

```julia
for i = 1 : 10
    println(i)
end
```

(a) Julia

```python
for i in range(1, 11)
    print i
```

(b) Python

```java
for (int i = 1; i < 11; i++){
    System.out.println(i);
}
```

(c) Java

```cpp
for (unsigned int i = 1; i < 11; i++){
    std::cout << i << "\n";
}
```

(d) C++

Figure 3: For-loops: Incrementing

```julia
for i = 10 : -1 : 1
    println(i)
end
```

(a) Julia

```python
for i in range(10, 0, -1)
    print i
```

(b) Python

```java
for (int i = 10; i > 0; i--){
    System.out.println(i);
}
```

(c) Java

```cpp
for (unsigned int i = 10; i > 0; i--){
    std::cout << i << "\n";
}
```

(d) C++

Figure 4: For-loops: Decrementing

```
1  while i < 10
2      a -= 1
3  end
```

(a) Julia

```
1  while (i < 10):
2      a -= 1
```

(b) Python

```
1  while (i < 10){
2      i--;
3  }
```

(c) Java

```
1  while (a < 10){
2      i--;
3  }
```

(d) C++

Figure 5: while-loop

```
1 while i < 10
2 a -= 1
3 end
```

(a) Julia

```
1 while (i < 10):
2 a -= 1
```

(b) Python

```
1 while (i < 10){
2 i--;
3 }
```

(c) Java

```
1 while (a < 10){
2 i--;
3 }
```

(d) C++

Figure 6: while-loop

```julia
function hello(name)
    println("Hello $name")
end
```

(a) Julia

```python
def hello(name):
    print "Hello", name
```

(b) Python

```java
public static void hello(String name){
    System.out.println("Hello " + name);
}
```

(c) Java

```cpp
void hello(std::string name)
{
    std::cout << "Hello " << name << "\n"
        ;
}
```

(d) C++

Figure 7: Declaring functions

```julia
array = fill(0, 100)
```

(a) Julia

```python
array = [0 for col in range(100)]
```

(b) Python

```java
int[] array = new int[100];
Arrays.fill(array, 0);
```

(c) Java

```cpp
std::array<int,100> array;
myarray.fill(0);
```

(d) C++

Figure 8: Declaring functions

4.2 Features

4.2.1 Multiple dispatch and dynamic typing

Dynamic typing means that type checks are mostly performed at runtime - opposed to static typing which makes the type checks at compile time. Dynamic typing allows the programmer to skip the type declaration and let the dynamic type system handle it. In Julia it is possible to specify types but for the most part this is not necessary. Julia also makes decisions about how much memory to allocate for variables and this will not change even if a big value which exceeds the memory of the variable are assigned to it. The memory size can be deceased and this will be addressed later under the section Garbage Collector. There is a range of memory sizes to chose from when initializing a variable, Julia will by default assign 32 bits to an integer or a float if no more memory is needed at initialization, but it is possible to manually chose one of the following.

· Signed / unsigned integers of; 8, 16, 32, 64 and 128 bits.

· Floating points of 16, 32 and 64 bits.

· Boolean – 8 bits

· Char – 32 bits

To initialize a variable with a give memory size:

```
x = Int8(10)
```

But if the variable is later assigned to a new value. Julia will automatically allocate 32 bits for the variable and more if needed:

```
x = Int8(10)
typeof(x) #Int8
x = 4
typeof(x) #Int32
```

If no certain memory size is needed but only a specific type then abstract types will come in handy. An abstract type is just some form of generalization of a concrete type. For example 16, 32 and 64 bit floats are of type AbstractFloat. This is especially useful when the programmer do not know how much memory a certain variable may need. Too little memory will result in a bug and too much memory is a waste.

Multiple dispatch is used to determine which function to call by the type of one or more of the parsed argument(s). This is useful when, e.g. two functions with the same name are declared:

```
function a(arg1::Int8)
    println("Int")
end

function a(arg1::Float16)
    println("Float")
end
```

If a variable is declared with a type of Int8 and parsed to the function a(...), then it will print "Int", and if the variable is declared as Float16 it will print "Float". It is also possible to use abstract types here, so arg1::Float16 and arg1::Int8 could be changed to arg1::AbstractFloat and arg1::AbstractInt – this will make the function 'a' accept any kinds of floats and ints.

4.2.2 User defined types

User defined types, also known as composite types gives the option to define new types in Julia. A composite type in Julia may look something like the following:

```
1 type person
2     age::Int16
3     name
4 end
```

If a variable in a composite type is not specified with any type, the default is ::Any which accepts any kinds of types. To initialize a composite type you can treat it like an object. Person = person(21, "Carl"), and the values can be changed with "Person.age = 25". The developers of Julia states that user defined types are as fast and compact as built in types.

4.2.3 Garbage Collector

Julia uses a garbage collector to automatically free the memory when needed. There is no guarantee when the garbage collector will run, but is it possible to force garbage collection with a function call: **gc()**. One thing to keep in mind is that once a name is defined in Julia, it will always be present till termination. The garbage collector do not free the memory of unreachable object but rather reallocates memory for objects when memory size has changed. So the only way the garbage collector can free memory is when an object has been reduced in size, for example:

```
1 A = rand(float32, 10000, 10000)
2 A = 0
3 Gc()
```

This code will generate a 10000x10000 matrix filled with random 32 bit floats which consumes a bit of memory. **A** will be set to 0 but will keep the memory size of the 10000x10000 matrix until garbage collection is done either manually as in this example or somewhere in the future when it is done automatically.

4.2.4 Built-in package manager

Julia comes with a built in package manager which keeps track of which packages that needs to be included in the users program to run, so is it not necessary to state which packages that need to be included. For example when the user calls the sort function, **sort()**, no package include statement or **math.sort()** is needed – this is done automatically by the package manager.

4.2.5 Lightweight green threading

A thread is lightweight when it shares address space with other threads opposed to heavyweight threads which has its own address space. If threads share the same address space the communication between them is faster and much simpler. The communication between heavyweight threads have to go through pipes or sockets. Green threads are threads that are scheduled not by the operating system but rather by the runtime library or virtual machine. Green threads does not have to relay on the operating system and can be controlled much better – the threads are in user space and not kernel space. As of Julia 0.4.5 multithreading is not available but in the unstable version 0.5.0 an experimental support of multithreading is available but will be addressed no further.

4.2.6 Meta-programming and Macros

Meta-programming is a way to write programs in programs and let them use program code as data. In Julia meta-programming can be done by defining macros. For example, the @time macro was used extensively in the project. The @time is in the standard Julia library. The @time takes a function as an argument and sets a timer in the top of the code from the argument and stops the timer in the bottom and prints the time passed and memory allocated. Meta-programming and macros are known from Lisp, a programming language, and have been used in early AI research.

4.2.7 Implementing code from other languages

Since Julia is a newer language there are not many written libraries yet. Julia has an import feature for both Python and C libraries to avoid this disadvantage. Import the library PyCall with the "using" operation to use python and import Python libraries via a macro @pyimport. To use C libraries or

coding, simply run the function ccall(). The programming language C is a well known and used language. Many systems are based on C, which makes C an advantage for hardware programming. Another feature is to execute shell commands. Execute shell commands using run(“). An example would be running run(‘echo Hello World‘) which would return the output "Hello World".

4.2.8 Extensible conversions and promotions for numeric and other types.

4.2.9 Designed for parallelism and distributed computation

Julia claims to be designed for parallelism and distributed computation. Parallel computing is dividing a problem into smaller problems and solve the smaller problems at the same time. Distributed computing is solving a problem using a network of computers working together to get to the solution. Julia have been trying to make this easier to achieve with different macros.

4.2.10 Automatic generation of specialized code for different argument types

4.2.11 Supports Unicode

The Unicode support globalize the language. Unicode is an encoding system for characters. Unicode supports a large range of characters and is designed to make it easier to read and write non-latin characters.

4.2.12 Devectorized code is fast

Vectorization of code is, for example, when a for loop is rewritten to sequential operations. Figure 9 shows an example of a non-vectorized for loop and the vectorized version. Vectorized code is usually faster. Julia claims that there is no reason to vectorize as the performance stays the same.

```
1       a = 1, 2, 3, 4
2       b = 2, 3, 4, 5
3       c is empty
4       for i = 1 to 4
5       begin
6           c[i] = a[i] + b[i]
7       end
```

(a) Devectorized code

```
1       a = [1, 2, 3, 4]
2       b = [2, 3, 4, 5]
3       c is empty
4       c[1] = a[1] + b[1]
5       c[2] = a[2] + b[2]
6       c[3] = a[3] + b[3]
7       c[4] = a[4] + b[4]
```

(b) Vectorized code

Figure 9: Vectorization

5 Project Euler

Project Euler is a website with a huge database of mathematical and programming challenges. In this report Project Euler has been used to find challenges for the testing of the different programming languages.

6 Theory

6.1 Benchmarking

Benchmarking is measuring the quality of a product. In this report we will be measuring the time it takes for a programming language to execute an algorithm.

6.2 Profiling

Profiling is a way to analyse code. For example the analysis could be to measuring the memory usage, time and function calls. The analysis now

shows which part of the code took the longest time or used up all memory. Profiling is offen used for optimization.

6.3 Time measurements

There are three types of time measurements. Wall, user and system time. Wall time is the time it takes in real time and wall time is affected if other program are running. System time is the cpu time used inside kernel. The time used inside kernel is for memory allocation, file reading, file writing and other thing going on outside the cpu and memory. User time is the time used calculating and comparing etc. In this report the sum of system and user time is used, because the system and user time are not affected by heavy loads on the machine and only shows the actual time taken for the Computer to solve the problem.

7 Materials and methods

7.1 Benchmarking

For the benchmarking we used: Lenovo W530: Intel Core i5-3320M 2.60GHz x 4, 8GB RAM, Ubuntu 16.04 64-bit, Windows 10 64-bit, Windows XP 32-bit MacBook: MacBook: Julia 0.4.5 Python 2.7.11+ C++11 Java 1.8.0_92 All time measurements are measured using the shell 'time' command which returns the three types of time taken for the algoritm to run. The problems are scaled in three different sizes and run in each language several times. A tool created for this project gather the data and calculates the average and median running time.

The benchmarking will be about the running time of each individual problem. When looking at running time the most common terms are wall time (real time) and CPU time. Wall time is the time it took the program to terminate, from start to finish. CPU time is how much time the program used on the CPU. The time difference between wall time and CPU time comes from the fact that the program being benchmarked might get interrupted by the operating system, because other tasks needs to be handled. These tasks could be other programs, or something that the OS needs to get done. The processing time of those tasks will be included in the wall time but excluded in the CPU time. To get the running times in Unix based system, the time

command is available from the terminal:

```
1 time <COMMAND>
```

Where <COMMAND> can be any terminal command. The time command will return three different times.

```
real    0m0.627s
user    0m0.462s
sys     0m0.147s
```

Figure 10: Output of time

.

· Real: This is the wall time mentioned above.

· User: User time is how much CPU time is spent in user mode.

· Sys: System time is how much CPU time is spent in kernel mode.

The difference between user- and kernel mode is the following:
In most memory protected system there are some locked operations for security reasons. E.g. allocation of memory or accessing hardware requires kernel mode and cannot be done in user mode. Processes like malloc and reading / writing to files will somewhere in the process sent a request to the kernel to do certain things and this is counted as system time. This does not necessarily mean that all time is spent in kernel mode when reading a file, since the only time that is spent in kernel mode is accessing the requested data in memory the rest of the operation is counted towards user time.

So to get much times is spent on the CPU, User time + system is a good estimation. To get an even better indication of the real CPU time, multiple test are done and the average is calculated. The average running time is considered since there can be fluctuations in run times for every test.

Another way to measure time spent on the CPU is to use the built in libraries for example System.currentTimeMillis() in Java. This is especially useful for profiling and figuring out which part of a program is time consuming. Benchmarking will be done by comparing CPU time of the same problem in different languages. The problems will be scaled to have smaller

and larger inputs. However, in some situations a large input can cause memory problems, which will be addressed in the specific problem. The memory problems is unfortunately making an upper bound for how large data is possible to pass for certain problems.

The algorithm used for a given problem is the same used in all compared languages. Extended tests will be included for which the algorithm is optimized for a given language.

8 Problems

8.1 Projecteuler 11

In the 20x20 grid below, four numbers along a diagonal line have been marked in red. The product of these numbers is 26 x 63 x 78 x 14 = 1788696. What is the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the 20x20 grid?

```
4851 3238 1738 4414 2453 3781
1058 2209 2485 2302 1601 4652
4791 3553 706 2520 4236 3842
1210 4707 2305 497 4336 3310
2611 3945 2743 1781 4386 3920
2476 1321 1870 471 1313 3326
```

The program runs from command-line/terminal with command line arguments in order to be able to test on different inputs and amount of adjacent numbers multiplied. To run the Julia version from the terminal:
Julia euler11.jl 100 4.
This would make the program run with a matrix of size 100 and calculating product of 4 adjacent numbers. A matrix generator was created, which can be found in appendix, to produce the matrix data. The matrix generator requires one argument, which is the size of the matrix. So if 500 is passed to the matrix generator, a file named 500mat.txt will be created containing 500x500 digits ranging from 100-9999.

The algorithm first reads through the input file and creates a matrix. To calculate products in all directions needed in the problem, the algorithm

goes through the matrix a total of 3 times. A nested for loop is needed to go though a matrix, lines 1-2 at figure 11. The outer loop iterates through the rows, and the inner loop iterates through the columns. Figure 11 calculates the horizontal and vertical directions, firgure 12 diagonally from left to right(downwards) and figure 13 going diagonally from right to left(downwards). The algorithm starts in the most upper left cell, and iterates through all cells in the matrix. Lets have the first cell as (1,1), first number is row(**i**) and second number is column(**j**). Looking at figure 11, **matLength** is the size of the matrix, size = 100 would mean a matrix of size $100 \cdot 100$. **numProd** is the number of adjacent numbers multiplied together. For matrix with size 100, and multiplying four adjacent numbers, the algorithm would do the following in figure 11:

Starting at cell (1,1) to the end which is cell (100, 100 - 4).

Line 5-7: Loops over the adjacent cells and multiplies the numbers.

Line 8-10: Sets current product to max product, if current is larger than previously max product.

```
1  for i = 1 : matLength
2    for j = 1 : matLength - numProd
3      #right/left
4      prod = 1
5      for k = 0 : numProd - 1
6        prod *= mat[i, j + k]
7      end
8      if prod > maxProd
9        maxProd = prod
10     end
11     #up/down
12     prod = 1
13     for k = 0 : numProd - 1
14       prod *= mat[j + k, i]
15     end
16     if prod > maxProd
17       maxProd = prod
```

Figure 11: Horizontal and vertical

Figure 12 shows the loop that iterates over the matrix and calculating the product of the diagonal going downwards from left to right. Figure 13 shows the diagonal product going upwards from left to right. The loop starts in the first row, and starting column is the last, not the first like the previously loops.

```
#diagonal left->right
for i = 1 : matLength - numProd
  for j = 1 : matLength - numProd
    prod = 1
    for k = 0 : numProd - 1
      prod *= mat[i + k, j + k]
    end
    if prod > maxProd
      maxProd = prod
```

Figure 12: Diagonal downwards left to right

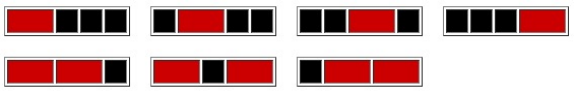
```
#diagonal right->left
for i = 1 : matLength - numProd
  for j = matLength : -1 : numProd
    prod = 1
    for k = 0 : numProd - 1
      prod *= mat[i + k, j - k]
    end
    if prod > maxProd
      maxProd = prod
```

Figure 13: Diagonal upwards left to right

8.2 Projecteuler 116

Description: A row of five black square tiles is to have a number of its tiles replaced with coloured oblong tiles from red(length two), green(length three), or blue(length four). If red tiles are chosen there are exactly seven ways. If green tiles are chosen there are three ways. And if blue tiles are chosen there are two ways. Figure 14 is a visualization of how the tiles can

be lain. Assuming that colours cannot be mixed there are $7 + 3 + 2 = 12$ ways of replacing the black tiles in a row measuring five units in length. How many different ways can the black tiles in a row measuring fifty units in length be replaced if colours cannot be mixed and at least one coloured tile must be used?

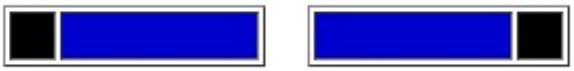


(a) Red tiles

.

(b) Green tiles

.



(c) Blue tiles

.

Figure 14: Projecteuler: 116

8.3 Quicksort

Quicksort applies the divide-and-conquer paradigm to sort numbers. Divide-and-conquer has three steps. **Divide** the problem into a number of subproblems that are smaller instances of the same problem. **Conquer** the subproblems by solving them recursively. If the subproblems sizes are small enough, however, just solve the subproblems in a straightforward manner. **Combine** the solutions to the subproblems into the solution for the original problem [1, chapter 4, page 65]. Description of quicksort: [1, chapter 7, page 170-171]

> **Divide**: Partition (rearrange) the array A[p..r] into two (possible empty) subarrays A[p..q-1] and A[q+1..r] such that each element of A[p–q1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1..r]. Compute the index q as part of this partitioning procedure.
>
> **Conquer**: Sort the two subarrays A[p..q-1] and A[q+1..r] by recursive calls to quicksort.

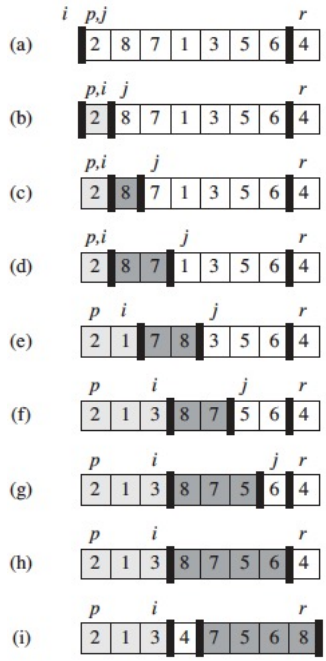


Figure 15: Quicksort

Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p..r] is now sorted.

Figure 15 shows a visualization of quicksort.

9 Results

9.1 Benchmarking

10 Personal experience with Julia

10.1 Sonni

Some of the design choices, in Julia, can be really frustrating for someone who is adapting to Julia. The indexes starts at 1 - this could be debatable because of the statement that Julia is used in a lot of scientific computations and many programming languages made for scientific computations starts at index 1. But as a programmer who is used to indexes starting at 0, this can cause a lot of unnecessary bugs. The developers of Julia also made a weird

design choice with the for-loop syntax. A incrementing for loop looks like:

```
Julia:
for i = 1 : n
    Some code ...
end

General languages:
for i = 1; i < n; i++
    Some code ...
```

This is straightforward like many other languages, but take a look at the decrementing for-loop:

```
Julia:
for i = n : -1 : 0
    Some code ...
end

General languages:
for i = n; i > 0; i--
    Some code
```

In Julia the exit condition and incrementation / decrementation are swapped and there is no consistency. The decrementing for-loops both start at n, and stops at 0. This kind of syntax can again cause bugs, and is not that easy to get eyes on.

The documentation of Julia is far from being great, which is a huge bump on the road to learn Julia. It feels far from complete. The documentation do not explain much but instead gives some examples. Essential information are often lacking, e.g. what types is needed to parse for a given function. It might point out that these arguments are needed, but lacking to describe those arguments. An example from the documentation is the function **rand()** which is used to generate random numbers:

```
rand( [ rng ] [ ,S ] [ , dims... ] )

    Pick a random element or array of random elements from the set
        of values specified by S; S can be:
```

```
4     - an indexable collection (for example 1:n or ['x','y','z']), or
5     - a type: the set of values to pick from is then equivalent to
          typemin(S):typemax(S) for integers (this is not applicable to
           BigInt), and to [0,1) for floating point numbers;
6     S defaults to Float64.
```

The argument **S** is explained, but does not mention anything about **rng** or **dims**.

Antoher example is how to access data within an array at a specific index. This should be one of the first things in the Array section, but is first described after about two pages of text. A person with experience in programming might know that to access an element in an array with an index is this simple line of code:

```
1 a[index]
```

But this isn't so obvious for new programmers. It is clear that something has to be done with the documentation.

Another downside to Julia is its consistency. The developers has tried to remove some of the burden from the programmer with dynamic typing, package manager system etc., but with memory allocation and garbage collection it does not make much sense. The garbage collector does not free memory from unreachable object. Sizes of variables cannot be changed after declaration, and is not automatically increased if needed. If the programmer assign for example 8 bit memory to a variable and later change the value of that variable, then julia automatically assigns 32 bit or more if needed to that variable even if the 8 bits were enough.

10.2 Simon

10.3 Asbjørn

I found the syntax a bit confusing at first. I am not used to Python and I missed code structure and forced typing, in Julia and Python it looks like the code is hovering. After some time I got used to it and I found it very easy to learn. It makes it a little hard to find answers with the documentation and community being a bit thinner than with other languages. I think the 1-indexing is a bit complicated because I had never thought any language

would use this, but I found that a lot of languages actually use this, they just are not that popular.

11 Discussion

Pros and Cons with good arguments

12 Conclusion

Is Julia a worthy competitor? Is it worth switching to Julia? Would we recommend Julia to anyone?

It can be difficult to make any conclusions by the benchmarking but it may give an idea of how Julia performs, compared to Java, Python and C++. One thing to keep in mind is that as of writing this report, Julia is in version 0.4.5 - it has yet to reach version 1.0.0 while the other languages are much older than Julia. This could mean that Julia might get even faster in the future.

13 Perspective

Articles to perspectivate:
http://www.evanmiller.org/why-im-betting-on-julia.html
High-Performance JIT Compiler chart at http://julialang.org/
http://www.nowozin.net/sebastian/blog/the-julia-language-for-scientific-computing.html
http://radar.oreilly.com/2013/10/julias-role-in-data-science.html
* Used too much times searching for articles.. *

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein *Introduction to Algorithms*, Cambridge, Massachusetts, third edition, 2009.

[2] http://www.tiobe.com/tiobe_index

[3] http://julialang.org/blog/2012/02/why-we-created-julia

[4] http://docs.julialang.org/en/latest/manual/parallel-computing/

[5] http://unicode.org/standard/WhatIsUnicode.html

[6] http://www.cs.cornell.edu/courses/cs1112/2013fa/Exams/exam2/vectorizedCode.pdf

[7] https://projecteuler.net/

[8] http://dictionary.cambridge.org/dictionary/english/benchmark

[9] http://dictionary.cambridge.org/dictionary/english/benchmark

14 Appendix

14.1 Process analysis

14.1.1 Projectmanagement, organizing workflow and organizing process of learning

The first thing we agreed doing was getting to know Julia for the research. We used a lot of time solving problems in Julia and trying to figure out how to get a view of what was to be done and how to organize it. We mostly communicated through facebook and created a repository on github to share project files. We gathered our solutions to Project Euler in the repository so that we do comparison. We mostly run into walls of information on how to benchmark right and we did not know where to start. Everytime we thought we were about to get to understand benchmarking our supervisor taught us something new. We created a board on Trello. We made a to-do list of cards so we knew what to be done and started handing out tasks. We started communicating more with our supervisor via email as we started writing the report. The group was really good at getting the tasks, we agreed on doing, done. We should have been using more tools for organizing and we did not set any deadlines for anything. Scrum is a tool which would have helped us a lot, but it is not that easy to learn. We should have met and worked together, several times a week, discussing our problems and goals.

14.1.2 Process of learning

We learn by both direct and indirect learning process. We learned Julia indirectly by solving Project Euler. We pretty much read about every topic which is direct learning. We cannot think of another way to learn programming than reading, trying it out, reading some more and get a result. Project Euler was a good support for learning, because when you are trying to learn something new, it is much easier, when you have a purpose for it. Programming is not as fun when you does not have a problem to solve.

14.1.3 Cooperation, difficulties in group and with supervisor

We actually had a good idea about who was good at what and we split the tasks between us with this in mind. The communication could have been better. It seemed that we worked on different times of the day which made

communication difficult. Our supervisor did his best and we think he did a good job.

14.1.4 Conclusion

Trello is a very good tool for organizing and we should have been using it from the beginning. Looking beyond the fact that some data was lost using Git, the tool is perfect for team programming. We have payed the price not making any deadlines. Deadlines seems useless when you got the project in your head, but when you loose track you wish you set some. We should have met and worked synced our progress. Overall we have done very well keeping the things we have done wrong in mind.

14.2 Copy of poster in A4