Multiple dispatch and dynamic typing.
Julia has a dynamic type system like python. A property of dynamic typing is the type checks which are performed at runtime, mostly - opposed to static typing which makes the type checks at compile time. In Julia it is possible to specify types but for the most part it is not necessary because of the dynamic type system. Julia makes decisions about what size to assign to the variables, and can change at run time.

Julia has abstract types - for example abstractfloat. 16, 32 and 64 bit floats are of type abstractfloat. Julia automatically assigns the memory size of variables – although it can be done manually by calling the parse function, or initialize the variable with the type and size; a = Float32(0).

Julia supports:
- Signed / unsigned integers of; 8, 16, 32, 64 and 128 bits.
- Floating points of 16, 32 and 64 bits.
- All the standard types like Boolean, char and so on are supported.

Julia supports multiple dispatch which is used to determine which function to call by the type of one or more of the parsed argument(s). This is needed when for example two functions with the same name are declared:

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

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

If you declare a variable and give it the type of a Int8 and call the function with that variable, 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 floats and ints.

Garbage collector
Julia uses a garbage collector to automatically free the memory when needed. There is no guarantee when the garbage collector will run, even if you force garbage collection with gc(). This will only tell the garbage collector that there is a request for garbage collection and then it will decide when to do it. It is also possible to disable and enable the garbage collector. However, the objects in Julia are represented as C pointers behind the screen and you have no control over this.

Meta-programming and Macros
In Julia you can implement meta-programming. Meta-programming is a way to write programs in programs and let them use program code as data. In Julia you can do this by making macros. A macro we are going to use a lot in this project is the @time macro. 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 is known from Lisp, a programming language, and have been used in early AI research.

Implementing code from other languages

Julia is a new language which will mean there are not many libraries written in Julia. Julia has an import feature for both Python and C libraries to avoid this disadvantage. To use Python in Julia you would have to add library PyCall with the "using" operation. Now you can use a macro to import Python libraries @pyimport. To use C libraries or coding you can simply run the function ccall(). Another great feature is to execute shell commands. You can execute shell commands using run(``). An example would be running run(`echo Hello World`) which would return the output "Hello World".

Conclusion

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.

Personal experience with Julia

Some of the design choices are really frustrating when adapting to Julia. An example would be the index starting at 1 - this could be debatable because of the statement that Julia is used in a lot in scientific computations, but as a programmer who is used to start at index 0, this will cause a lot of bugs. The developers of Julia made a weird design choice to the for-loop, when you want to loop downwards. In most programming language it will look something like

```
for i = n; i > 0; i--
```

But in Julia they switch the increment / decrement part with the exit condition

```
for i = n : -1 : 0
```

So in both examples the loops start at n, decrements down to 0 and stops. This kind of syntax change can be the cause of a lot of bugs, when programming in Julia. This is a simple thing that is easy to overcome, but a really big problem with Julia is the documentation. It feels far from complete.
The documentation doesn't explain much but instead gives some examples. The problem is that it sometimes misses essential things like what types you'll have to parse for a certain function. It might tell you that you have to parse these arguments but forget to tell you what they are. An example from the documentation is:

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

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

an indexable collection (for example 1:n or ['x','y','z']), or
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;
S defaults to Float64."

They explain what S is, but forgets about rng and dims.

Simple things like how to access data in an array at a specific index. This should be one of the first things in the Array section, but only gets shown after about 2 pages of text. If you have experience in other programming languages, you might know that you'll just have to type a[index] but this isn't so obvious for new programmers. It is clear that something has to be done to the documentation.