Julia Programming Language

Laukik Limbukar[#1], Tushar Kamble[*2], Sameer Herkal[#3]

Computer Department, Mumbai University
Terna Engineering College, Mumbai, India

[1]laukik2424@gmail.com

[2]tshrtnmk8@gmail.com

1.Abstract-Julia is a dynamic programming language designed for numerical computing, statistical purposes, graphical representation of data , scientific computing, etc .It is a multi-paradigm language : Object Oriented, Procedural , Multi-staged , functional ,meta. It has an efficient compiler which gives good performance which is in line with statically typed languages such as C . It can use C functions directly and needs the Pycall package to use the Python language. It includes power-shell like capabilities and its user data types are as swift as its basic data types .It also provides Multi-dispatch for programming. It is MIT licensed which is free and open.It has functions and capabilities similar to the R programming language, Python but offers computing speed in line with fast programming languages like C , Fortran . It also has many graphical tools for plotting, graphical representation of statistics. Julia Developer community is actively contributing to the development of external packages .It uses indention similar to python. Julia's base library is written in Julia and also includes C and Fortran libraries for mathematical computations. It also offers many functions similar to R language. Its chief developers are Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and other contributors.

2. Introduction:

Julia is a swift dynamic programming language designed for the purpose of distributed, parallelism computing. It is designed for the data scientists to provide a fast and dynamic . Programming language to solve numerical and computation problems. R and Python offer the Capabilities and ease of use but become increasingly slow while handling large data sets.
Julia offers the capabilities and the ease of use while matching computation speed of statically compiled languages such as C.

b. Julia shows that machine performance and human ease of doing computation can go hand in hand.A type system with parametric types in a fully dynamic programming language and multiple dispatch as its core programming paradigm. It allows concurrent, parallel and distributed computing, and direct calling of C and Fortran libraries without glue code.
Also includes efficient libraries for floating-point calculations, linear algebra, random number generation, fast Fourier transforms and regular expression matching.The most notable aspect of Julia's implementation is its speed, which is often within a factor of two relative to fully optimized C code. Scientific computing requires high speed for programming..for high performancemodern language design and compiler techniques make it possible to mostly eliminate the performance trade-off and provide a single environment productive enough for prototyping .So Julia language came into existence.Development of Julia began in 2009 and an open-source version was publicized in February 2012.Julia, the 0.5.x line, is on a monthly release schedule where bugs are fixed and some new features from 0.6-dev are backported.

3. Keywords:

Julia, Dynamic, Multi-paradigm, Packages, Computation, Data, Fast,Plotting.

4.Julia data types

Data elements come in different shapes and sizes, which are called **types**.

Type hierarchy

In Julia types are organized in an hierarchical way, and this hierarchy has a tree structure. At the tree's root, we have a special type called `Any`, and all other types are connected to it directly or indirectly. Informally, we can say that the type `Any` has children. Its children are called `Any`'s **subtypes**. Alternatively, we say that a child's **supertype** is `Any`.

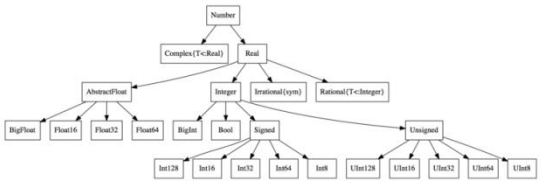

Fig 1 Data Type Hierarchy

Concrete and abstract types

Each object in Julia (informally, this mean everything you can put into a variable in Julia) has a type. But not all types can have a respective object (instances of that type). The only ones that can have instances are called **concrete types**. These types cannot have any subtypes. The types that can have subtypes (e.g. `Any`, `Number`) are called **abstract types**. Therefore we cannot have a object of type `Number`, since it's an abstract type. In other words, only the leaves of the type tree are concrete types and can be instantiated.

#this function gets a number and returns the same number plus one
Function plus_one(n::number)
Return n+1
end

In this example, the function expects a variable `n`. The type of `n` must be subtype of `Number` (directly or indirectly) as indicated with the :: syntax (but don't worry about the syntax yet). What does this mean? No matter if `n`'s type is `Int` (Integer number) or `Float64` (floating-point number), the function `plus_one()` will work correctly. Furthermore, `plus_one()` will not work with any types that are not subtypes of `Number`

We can divide concrete types into two categories: primitive (or basic), and complex (or composite). Primitive types are the building blocks, usually hardcoded into Julia's heart, whereas composite types group many other types to represent higher-level data structures.

You'll probably see the following primitive types:

- the basic integer and float types (signed and unsigned): Int8, Uint8, Int16, Uint16, Int32, Uint32, Int64, Uint64, Int128, Uint128, Floa t16, Float32, and Float64
- more advanced numeric types: BigFloat, BigInt
- Boolean and character types: Bool and Char
- Text string types: String

A simple example of a composite type is Rational, used to represent fractions. It is composed of two pieces, a numerator and a denominator, both integers (of type Int).

Type	Signed?	Number of bits	Smallest value	Largest valu
Int8	√	8	-2^7	2^7-1
UInt8		8	0	2^8-1
Int16	√	16	-2^15	2^15-1
UInt16		16	0	2^16-1
Int32	√	32	-2^31	2^31-1
UInt32		32	0	2^32-1
Int64	√	64	-2^63	2^63-1
UInt64		64	0	2^64-1
Int128	√	128	-2^127	2^127-1
UInt128		128	0	2^128-1
Bool	N/A	8	false (0)	true (1)

Fig 2 Types of data

Type	Precision	Number of bits
Float16	half	16
Float32	single	32
Float64	double	64

Fig 3 Types of Float

Creating types

In Julia, it's very easy for the programmer to create new types, benefiting from the same performance and language-wise integration that the native types (those made by Julia's creators) have.

Abstract Types

Suppose we want to create an abstract type. To do this, we use Julia's keyword `abstract` followed by the name of the type you want to create:

abstract MyAbstractType

Concrete Types

You can create new concrete types. To do this, use the `type` keyword, which has the same syntax as declaring the supertype. Also, the new type may contain multiple fields, where the object stores values. As an example, let's define a concrete type that is a subtype of `MyAbstractType`:

```
type MyType<:MyAbstractType
foo
bar::Int
end
```

We just created a type called `MyType`, a subtype of `MyAbstractType`, with two fields: `foo` that can be of any type, and `bar`, that is of type `Int`.

By default, Julia creates a **constructor**, a function that returns an object of that type. The function has the same name of the type, and each argument of the function correspond to each field. In this example, we can create a new object by typing:

```
Julia> x = mytype(“Hello World!”,10)
myType(“Hello World!”,10)
```

5.Applications :

Introduction to the equilibrium green’s functions: Condensed matter examples with numerical implementations :

The Green’s function method has applications in several fields in Physics, from classical differential

equations to quantum many-body problems. In the quantum context, Green's functions are correlation functions, from which it is possible to extract information from the system under study, such as the density of states, relaxation times and response functions. Despite its power and versatility, it is known as a laborious and sometimes cumbersome method. Here we introduce the equilibrium Green's functions and the equation-of-motion technique, exemplifying the method in discrete lattices of non-interacting electrons. We start with simple models, such as the two-site molecule, the infinite and semi-infinite one-dimensional chains, and the two-dimensional ladder. Numerical implementations are developed via the recursive Green's function, implemented in Julia, an open-source, efficient and easy-to-learn scientific language.

6.WHERE THE LANGUAGE CAN BE USED :

Current and future platforms

While Julia uses JIT (MCJIT from LLVM) – Julia generates native machine code, directly, the first time a function is run (not a byte code that is run on a VM, as with e.g. Java/JVM or Java/Dalvik in Android).

Current support is for 32- and 64-bit (all except for ancient pre-Pentium 4-era, to optimize for newer) x86 processors (and with download of executables or source code also available for other architectures). "Experimental and early support for ARM, AARCH64, and POWER (little-endian) is available too." Including support for Raspberry Pi 1 and later (e.g. "requires at least armv6").

Support for GNU Hurd is being worked on.

Julia version 0.6 is planned for 2016 and 1.0 for 2017 and some features are discussed for 2+ that is also planned, e.g. "multiple inheritance for abstract types".

Julia2C source-to-source compiler

A Julia2C source-to-source compiler from Intel Labs is available. This source-to-source compiler is a fork of Julia, that implements the same Julia language syntax, which emits C code (for compatibility with more CPUs) instead of native machine code, for functions or whole programs. The compiler is also meant to allow analyzing code at a higher level than C.

Intel's Parallel Accelerator.jl can be thought of as a partial Julia to C++ compiler, but the objective is parallel speedup (can by "100x over plain Julia", for the older 0.4 version, and could in cases also speed up serial code manyfold for that version), not compiling the full language to C++ (it's only an implementation detail, that may be dropped later). It needs not compile all syntax, as the rest is handled by Julia.

8.Experimental multi-threading support for the Julia programming language :

Julia is a young programming language that is designed for technical computing. Although Julia is dynamically typed it is very fast and usually yields C speed by utilizing a just-in-time compiler. Still, Julia has a simple syntax that is similar to Matlab, which is widely known as an easy-to-use programming environment. While Julia is very versatile and provides asynchronous programming facilities in the form of tasks (coroutines) as well as distributed multi-process parallelism, one missing feature is shared memory multi-threading. In this paper we present our experiment on introducing multi-threading support in the Julia programming environment. While our implementation has some restrictions that have to be taken into account when using threads, the results are promising yielding almost full speedup for perfectly parallelizable tasks.

7.Parallel prefix polymorphism permits parallelization, presentation :

Polymorphism in programming languages enables code reuse. Here, we show that polymorphism has broad applicability far beyond computations for technical computing: parallelism in distributed computing, presentation of visualizations of runtime data flow, and proofs for formal verification of correctness. The ability to reuse a single codebase for all these purposes provides new ways to understand and verify parallel programs.

Despite all this, the language can find its niche as an open-source alternative to MATLAB because its syntax might be appealing to MATLAB users. It is doubt it can seriously challenge Python as the de-facto standard for numerical computing.

Julia, a general purpose programming language is made specifically for scientific computing. It is a flexible dynamic language with performance comparable to traditional statically-typed languages. Julia tries to provide a single environment

productive enough for prototyping and efficient for industry grade applications. You will find it's model very intuitive and in align with what you want from a programming language for machine learning.

It's statistical and machine learning packages are evolving at a good pace and provides an amazing library for Deep learning too. It also supports parallel computing and High performance computing out of the box with an implementation of Message passing which is comparatively easier to program. It will definitely replace MATLAB/OCTAVE and maybe R, and will complement Python.

Julia is good, and we can do pretty much everything with it. It is becoming better and better with every release. It is not appropriate to say that it is the "only" future of programming, but it definitely is there somewhere and it definitely is worth learning.

9.Features of Julia:

- It is a multi-paradigm language multiple dispatch, procedural-based upon procedural calls, meta-program can modify itself, multistaged-compilation is divided into various phases.
- It feels like R and python but provides the good performance such as C languages.
- Multiple dispatch: selection of a function implementation on the basis of the types of each argument of the function.
- It is licensed by the MIT under their free and open source.
- It offers good performance, in line with the statically typed languages such as C.
- It automatically generates efficient, specialized code for different argument types
- To use Python functions use the PyCall package
- User defined data types are in line with the built-in data types in terms of execution. Example.

  In[18]: # Type Parameter Example (Parameter T)
  # Define a Symmetric Arrow Matrix Type with elements of type T
  type SymArrow{T}
  dv::Vector{T} # diagonal
  ev::Vector{T} # 1st row[2:n]
  end
  # Create your first Symmetric Arrow Matrix
  S = SymArrow([1,2,3,4,5],[6,7,8,9])
  Out[18]:
  SymArrow{Int64}([1,2,3,4,5],[6,7,8,9])

- It is crafted for the distributed, cloud computing and parallelism computing.
- It provides built-in package manager.
- It offers various functions for drawing graphs, plotting curves, etc.
- It used indention instead of parenthesis.

10.Julia for the beginners:

We can run Julia in the following ways:

- Using built-in Julia command line.

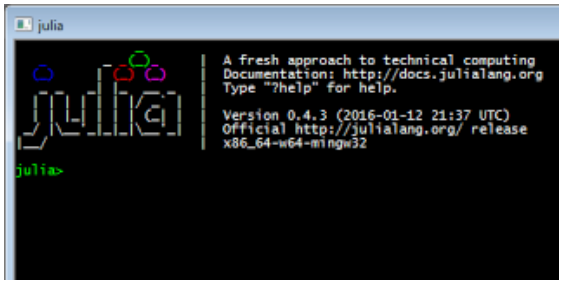

Fig 4 Julia Command line

- Using Juno-IDE

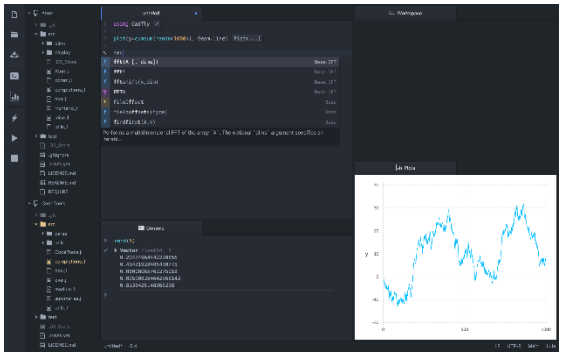

Fig 5 Juno IDE

- Using installation free Julia website which is powered by Jupyter notebooks.

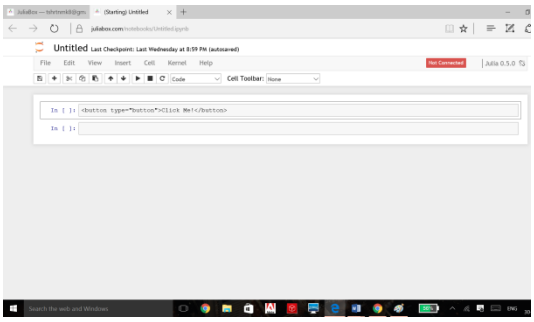

Fig 6 Juliabox

11.Functions in Julia:

In Julia functions are objects which returns a value after mapping a tuple .Functions can be affected by the global state of the program. In Julia a function can be written in two ways:

- function f(u,c)
  u+c
  end

- f(u,c)=u+c

A function is called using tradition parenthesis syntax

```
julia> f(2,3)
5
```

Return keyword in Julia
Return keyword returns the evaluated value of the last expression in the body.

```
function g(u,c)
 return u* c
 y + u
end
```

Operations are functions:
In Julia operators perform the role of functions.

T=+;
T(1,9,0)
10

Example:

```
map([ B, C]) do p

 if x < 0 && iseven(p)
 return 0
 elseif p == 0
 return 1
  else
 return p
  end
end
```

Plotting and graphs in Julia:
Julia use external languages for plotting. Following are the packages available in Julia:

```
Pkg.add("PyPlot")

using PyPlot

u = linspace(0,2*pi,1080); c = sin(3*x +
4*cos(2*x))

plot(u, c, color="blue", linewidth=4.0, linestyle="--")
```

- Gadfly: Gadfly uses a Wickham-Wilkinson style of graphics in Julia.

Example:

```
Pkg.add("Gadfly")

using Gadfly

draw(SVG("output.svg", 5inch, 8inch), plot([sin,
cos], 0, 60))
```

Example-using Plotly

trace1 = [

```
  "x" => [1, 2, 3, 4],
  "y" => [0, 2, 3, 5],
  "fill" => "tozeroy",
  "type" => "scatter"
]
trace2 = [
  "x" => [1, 2, 3, 4],
  "y" => [3, 5, 1, 7],
  "fill" => "tonexty",
  "type" => "scatter"
]
data = [trace1, trace2]
response = Plotly.plot(data, ["filename" =>
"basic-area", "fileopt" => "overwrite"])
plot_url = res
ponse["url"]
```

OUTPUT:

Fig 6

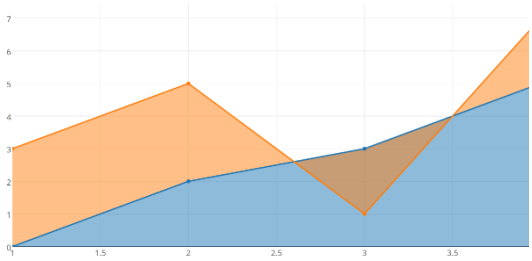

Fig 7 Graph plotted using Julia

```
julia> Pkg.add("Plots")
julia> Pkg.add("PyPlot")
julia> Pkg.add("GR")
julia> Pkg.add("UnicodePlots")
julia> Pkg.add("PlotlyJS")
```

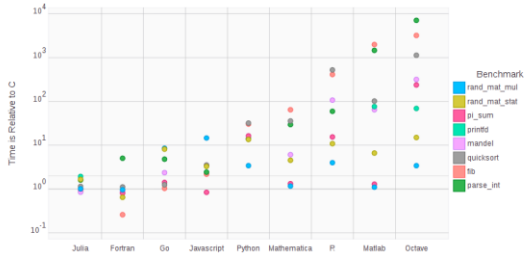

Fig 8 Plotting using Julia

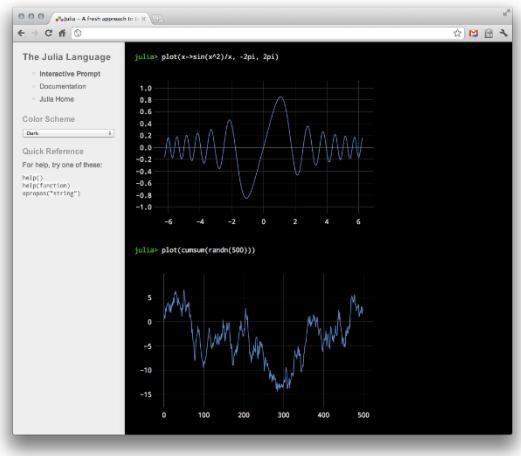

Fig 9 Waves dawn in Julia

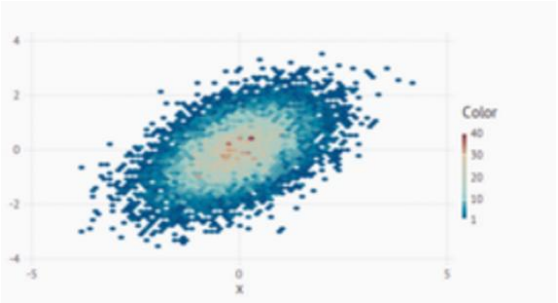

Fig 10 Plotting

Fig 11 Graphs Using plotting packages

12.Some Packages in Julia:

- ACME-Used for Analog and circuit modelling for Julia.
- ASCIIPlots:Used for generating for simple plots as ASCII art in Julia
- Bio:Bioinformatics and Computational Biology Infrastructure for Julia
- Blink:Web-based GUIs for Julia
- Blocks:A framework to represent chunks of entities and parallel methods on them.
- Bukdu:Bukdu is a web development framework for Julia.
- CPUTime:Julia module for CPU timing
- Cairo:Bindings to the Cairo graphics library.
- Calculus:Calculus functions in Julia
- PyCall:Used to call python functions.

13.Embedding Julia:

Julia's ccall keyword is used to call C-exported or Fortran shared library functions individually.

Julia has Unicode 9.0 support, with UTF-8 used for source code (and by default for strings) and e.g. optionally allowing common math symbols for many operators, such as ∈ for the in operator.

Julia has packages supporting markup languages such as HTML, (and also for HTTP), XML, JSON and BSON.

A second programming language ca also be used for embedding. In the following example C language calls Julia function:

```
#include <julia.h>

int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init(NULL);

    /* run Julia commands */
    jl_eval_string("print(sqrt(2.0))");

    /* strongly recommended: notify Julia that the
         program is about to terminate. this allows
         Julia time to cleanup pending write requests
         and run all finalizers
    */
    jl_atexit_hook(0);
    return 0;
}
```

15.Networking and streams

Real time application

Networking and Streams

Julia provides a rich interface to deal with streaming I/O objects such as terminals, pipes and TCP sockets.

This interface,presented in a synchronous manner to the programmer dParallel Computing

Most modern computers possess more than one CPU, and several computers can be combined together in a cluster. Harnessing the power of these multiple CPUs allows many computations to be completed more quickly. There are two major factors that influence performance: the speed of the CPUs themselves, and the speed of their access to memory. In a cluster, it's fairly obvious that a given CPU will have fastest access to the RAM within the same computer (node)

Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once.

15.CONCLUSION:

Julia is under continuous development (0.4-0.5) and Version 1.0 is at least 1 year away from this date. The core language is very reliable and stable but isn't backward compatible as you would expect. So, it is not completely ready for production systems yet. It has a large community of active users and contributors which frequently create new packages and functions.

References:

1. https://github.com/dcjones/Gadfly.jl/issues/110
2. http://lambda-the-ultimate.org/node/4452
3. https://en.wikibooks.org/wiki/Introducing_Julia/Plotting
4. http://junolab.org/
5. http://www.gregorioambrosio.com/2016/02/julia-new-direction-in-scientific.html
6. https://github.com/dcjones/Gadfly.jl
7. http://pkg.julialang.org/