Introduction to the Julia language

Marc Fuentes - SED Bordeaux

Outline

1. motivations

Outline

1. motivations
2. Julia as a numerical language

Outline

1. motivations
2. Julia as a numerical language
3. types and methods

Outline

1. motivations
2. Julia as a numerical language
3. types and methods
4. about performance

Why yet another Matlab -like language?

- *scripting languages* such as Matlab , **Scipy** , **Octave** etc... are efficient to prototype algorithms (more than C++/Fortran)

Why yet another Matlab -like language?

- *scripting languages* such as Matlab , **Scipy** , **Octave** etc... are efficient to prototype algorithms (more than C++/Fortran)
- Matlab is not free neither open source. It still remains the *lingua franca* in numerical algorithms.

Why yet another Matlab -like language?

- *scripting languages* such as Matlab , **Scipy** , **Octave** etc... are efficient to prototype algorithms (more than C++/Fortran)
- Matlab is not free neither open source. It still remains the *lingua franca* in numerical algorithms.
- **Octave** is very slow (but highly compatible with Matlab)

Why yet another Matlab -like language?

- *scripting languages* such as Matlab , **Scipy** , **Octave** etc... are efficient to prototype algorithms (more than C++/Fortran)
- Matlab is not free neither open source. It still remains the *lingua franca* in numerical algorithms.
- **Octave** is very slow (but highly compatible with Matlab)
- **Python** along **Numpy** and **Scipy** is a beautiful language, but a little bit slow and do not support any parallelism as a built-in feature.

Why yet another Matlab -like language?

- *scripting languages* such as Matlab , **Scipy** , **Octave** etc... are efficient to prototype algorithms (more than C++/Fortran)
- Matlab is not free neither open source. It still remains the *lingua franca* in numerical algorithms.
- **Octave** is very slow (but highly compatible with Matlab)
- **Python** along **Numpy** and **Scipy** is a beautiful language, but a little bit slow and do not support any parallelism as a built-in feature.
- **Pypy** which is a nice JIT compiler of **Python** , but does not support neither **Numpy** neither **Scipy**

Why yet another Matlab -like language?

- *scripting languages* such as Matlab , **Scipy** , **Octave** etc... are efficient to prototype algorithms (more than C++/Fortran)
- Matlab is not free neither open source. It still remains the *lingua franca* in numerical algorithms.
- **Octave** is very slow (but highly compatible with Matlab)
- **Python** along **Numpy** and **Scipy** is a beautiful language, but a little bit slow and do not support any parallelism as a built-in feature.
- **Pypy** which is a nice JIT compiler of **Python** , but does not support neither **Numpy** neither **Scipy**
- **R** is well suited for statistics, but suffer from old-syntax troubles

Why yet another Matlab -like language?

- *scripting languages* such as Matlab , **Scipy** , **Octave** etc... are efficient to prototype algorithms (more than C++/Fortran)
- Matlab is not free neither open source. It still remains the *lingua franca* in numerical algorithms.
- **Octave** is very slow (but highly compatible with Matlab)
- **Python** along **Numpy** and **Scipy** is a beautiful language, but a little bit slow and do not support any parallelism as a built-in feature.
- **Pypy** which is a nice JIT compiler of **Python** , but does not support neither **Numpy** neither **Scipy**
- **R** is well suited for statistics, but suffer from old-syntax troubles

Why yet another Matlab -like language?

- *scripting languages* such as Matlab , **Scipy** , **Octave** etc... are efficient to prototype algorithms (more than C++/Fortran)
- Matlab is not free neither open source. It still remains the *lingua franca* in numerical algorithms.
- **Octave** is very slow (but highly compatible with Matlab)
- **Python** along **Numpy** and **Scipy** is a beautiful language, but a little bit slow and do not support any parallelism as a built-in feature.
- **Pypy** which is a nice JIT compiler of **Python** , but does not support neither **Numpy** neither **Scipy**
- **R** is well suited for statistics, but suffer from old-syntax troubles

$\longrightarrow$ Why do not try a new language for numerical computation?

A language for numerical computation

- Julia 's syntax is very similar to langages as Matlab , **Python** or **Scilab** , so switching to Julia is fast

A language for numerical computation

- Julia 's syntax is very similar to langages as Matlab , **Python** or **Scilab** , so switching to Julia is fast
- do not require *vectorized* code to run fast(JIT compiler)

A language for numerical computation

- Julia 's syntax is very similar to langages as Matlab , **Python** or **Scilab** , so switching to Julia is fast
- do not require *vectorized* code to run fast(JIT compiler)
- it uses references (also for function arguments)

A language for numerical computation

- Julia 's syntax is very similar to langages as Matlab , **Python** or **Scilab** , so switching to Julia is fast
- do not require *vectorized* code to run fast(JIT compiler)
- it uses references (also for function arguments)
- indices start to 1 and finish to `end`

A language for numerical computation

- Julia 's syntax is very similar to langages as Matlab , **Python** or **Scilab** , so switching to Julia is fast
- do not require *vectorized* code to run fast(JIT compiler)
- it uses references (also for function arguments)
- indices start to 1 and finish to `end`
- use brackets `[,]` for indexing

A language for numerical computation

- Julia 's syntax is very similar to langages as Matlab , **Python** or **Scilab** , so switching to Julia is fast
- do not require *vectorized* code to run fast(JIT compiler)
- it uses references (also for function arguments)
- indices start to 1 and finish to `end`
- use brackets `[,]` for indexing
- it supports broadcasting

A language for numerical computation

- Julia 's syntax is very similar to langages as Matlab , **Python** or **Scilab** , so switching to Julia is fast
- do not require *vectorized* code to run fast(JIT compiler)
- it uses references (also for function arguments)
- indices start to 1 and finish to `end`
- use brackets `[,]` for indexing
- it supports broadcasting
- support 1D arrays

A language for numerical computation

- Julia 's syntax is very similar to langages as Matlab , **Python** or **Scilab** , so switching to Julia is fast
- do not require *vectorized* code to run fast(JIT compiler)
- it uses references (also for function arguments)
- indices start to 1 and finish to `end`
- use brackets `[,]` for indexing
- it supports broadcasting
- support 1D arrays

A language for numerical computation

- Julia 's syntax is very similar to langages as Matlab , **Python** or **Scilab** , so switching to Julia is fast
- do not require *vectorized* code to run fast(JIT compiler)
- it uses references (also for function arguments)
- indices start to 1 and finish to `end`
- use brackets `[,]` for indexing
- it supports broadcasting
- support 1D arrays

$\longrightarrow$ let us have a look to some examples

Functional aspects

- support anonymous functions : `(x -> x*x)(2)`

Functional aspects

- support anonymous functions :   `(x -> x*x)(2)`
- support `map`, `reduce`, `filter` functions

Functional aspects

- support anonymous functions :   `(x -> x*x)(2)`
- support `map`, `reduce`, `filter` functions
- functions support variadic arguments (using tuples)

Functional aspects

- support anonymous functions : `(x -> x*x)(2)`
- support `map`, `reduce`, `filter` functions
- functions support variadic arguments (using tuples)
- comprehension lists

Functional aspects

- support anonymous functions : `(x -> x*x)(2)`
- support `map`, `reduce`, `filter` functions
- functions support variadic arguments (using tuples)
- comprehension lists
- functions are not supposed to modify their arguments, otherwise they follow the `!` convention like `sort!`

Parallelism

Parallelism

- Julia has a built-in support for a distributed memory parallelism

Parallelism

- Julia has a built-in support for a distributed memory parallelism
- one-sided message passing routines

Parallelism

- Julia has a built-in support for a distributed memory parallelism
- one-sided message passing routines
  - `remotecall` to launch a computation on a given process

Parallelism

- Julia has a built-in support for a distributed memory parallelism
- one-sided message passing routines
  - `remotecall` to launch a computation on a given process
  - `fetch` to retrieve informations

Parallelism

- Julia has a built-in support for a distributed memory parallelism
- one-sided message passing routines
  - `remotecall` to launch a computation on a given process
  - `fetch` to retrieve informations
- high level routines

Parallelism

- Julia has a built-in support for a distributed memory parallelism
- one-sided message passing routines
  - `remotecall` to launch a computation on a given process
  - `fetch` to retrieve informations
- high level routines
  - `@parallel` reduction for lightweight iterations

Parallelism

- Julia has a built-in support for a distributed memory parallelism
- one-sided message passing routines
  - `remotecall` to launch a computation on a given process
  - `fetch` to retrieve informations
- high level routines
  - `@parallel` reduction for lightweight iterations
  - `pmap` for heavy iterations

Parallelism

- Julia has a built-in support for a distributed memory parallelism
- one-sided message passing routines
  - `remotecall` to launch a computation on a given process
  - `fetch` to retrieve informations
- high level routines
  - `@parallel` reduction for lightweight iterations
  - `pmap` for heavy iterations
- support for distributed arrays in the standard library

external libraries calls

external libraries calls

- sometimes you need to call a C/Fortran code

external libraries calls

- sometimes you need to call a C/Fortran code
- "no boilerplate" philosophy : do not require Mexfiles, Swig or other wrapping system

external libraries calls

- sometimes you need to call a C/Fortran code
- "no boilerplate" philosophy : do not require Mexfiles, Swig or other wrapping system
- the code must be in a **shared** library

external libraries calls

- sometimes you need to call a C/Fortran code
- "no boilerplate" philosophy : do not require Mexfiles, Swig or other wrapping system
- the code must be in a **shared** library
- the syntax is the following

external libraries calls

- sometimes you need to call a C/Fortran code
- "no boilerplate" philosophy : do not require Mexfiles, Swig or other wrapping system
- the code must be in a **shared** library
- the syntax is the following

external libraries calls

- sometimes you need to call a C/Fortran code
- "no boilerplate" philosophy : do not require Mexfiles, Swig or other wrapping system
- the code must be in a **shared** library
- the syntax is the following

```
ccall(:function, "lib"), return_type, (type_1,...,type_n), arg
```

Types

Types

- There is a graph type in Julia reflecting the hierarchy of types

```
None <: Int64 <: Number <: Real <: Any
```

Types

- There is a graph type in Julia reflecting the hierarchy of types

```
None <: Int64 <: Number <: Real <: Any
```

- support both abstract and concrete types

Types

- There is a graph type in Julia reflecting the hierarchy of types

```
None <: Int64 <: Number <: Real <: Any
```

- support both abstract and concrete types
- user can annotate the code with operator `::` "is an instance of"

Types

- There is a graph type in Julia reflecting the hierarchy of types

```
None <: Int64 <: Number <: Real <: Any
```

- support both abstract and concrete types
- user can annotate the code with operator `::` "is an instance of"
- Julia supports

Types

- There is a graph type in Julia reflecting the hierarchy of types

```
None <: Int64 <: Number <: Real <: Any
```

- support both abstract and concrete types
- user can annotate the code with operator `::` "is an instance of"
- Julia supports
  - composite types

Types

- There is a graph type in Julia reflecting the hierarchy of types

```
None <: Int64 <: Number <: Real <: Any
```

- support both abstract and concrete types
- user can annotate the code with operator `::` "is an instance of"
- Julia supports
  - composite types
  - union types

Types

- There is a graph type in Julia reflecting the hierarchy of types

  `None <: Int64 <: Number <: Real <: Any`

- support both abstract and concrete types
- user can annotate the code with operator `::` "is an instance of"
- Julia supports
  - composite types
  - union types
  - tuple types

Types

- There is a graph type in Julia reflecting the hierarchy of types

```
None <: Int64 <: Number <: Real <: Any
```

- support both abstract and concrete types
- user can annotate the code with operator `::` "is an instance of"
- Julia supports
  - composite types
  - union types
  - tuple types
  - parametric types

Types

- There is a graph type in Julia reflecting the hierarchy of types

```
None <: Int64 <: Number <: Real <: Any
```

- support both abstract and concrete types
- user can annotate the code with operator `::` "is an instance of"
- Julia supports
  - composite types
  - union types
  - tuple types
  - parametric types
  - singleton types

Types

- There is a graph type in Julia reflecting the hierarchy of types

  `None <: Int64 <: Number <: Real <: Any`

- support both abstract and concrete types
- user can annotate the code with operator `::` "is an instance of"
- Julia supports
  - composite types
  - union types
  - tuple types
  - parametric types
  - singleton types
  - type aliases

Multiple dispatch

Multiple dispatch

- main idea : define piecewisely methods or functions depending on their arguments types

Multiple dispatch

- main idea : define piecewisely methods or functions depending on their arguments types
- let us define `f`

```
f(x::Float64,y::Float64) = 2x + y
f(x::Int,y::Int) = 2x + y
f(2.,3.) # returns 7.0
f(2,3) # returns 7.0
f(2, 3.) # throw an ERROR: no method f(Int64,Float64)
```

Multiple dispatch

- main idea : define piecewisely methods or functions depending on their arguments types
- let us define f

```
f(x::Float64,y::Float64) = 2x + y
f(x::Int,y::Int) = 2x + y
f(2.,3.) # returns 7.0
f(2,3) # returns 7.0
f(2, 3.) # throw an ERROR: no method f(Int64,Float64)
```

but if we define g

```
g(x::Number, y::Number) = 2x + y
g(2.0, 3) # now returns 7.0
```

Multiple dispatch

- main idea : define piecewisely methods or functions depending on their arguments types
- let us define `f`

```
f(x::Float64,y::Float64) = 2x + y
f(x::Int,y::Int) = 2x + y
f(2.,3.) # returns 7.0
f(2,3) # returns 7.0
f(2, 3.) # throw an ERROR: no method f(Int64,Float64)
```

  but if we define g

```
g(x::Number, y::Number) = 2x + y
g(2.0, 3) # now returns 7.0
```

- no automatic or magic conversions : for operators arguments are promoted to a common type (user-definable) and use the specific implementation

Multiple dispatch

- main idea : define piecewisely methods or functions depending on their arguments types
- let us define f

  ```
  f(x::Float64,y::Float64) = 2x + y
  f(x::Int,y::Int) = 2x + y
  f(2.,3.) # returns 7.0
  f(2,3) # returns 7.0
  f(2, 3.) # throw an ERROR: no method f(Int64,Float64)
  ```

  but if we define g

  ```
  g(x::Number, y::Number) = 2x + y
  g(2.0, 3) # now returns 7.0
  ```

- no automatic or magic conversions : for operators arguments are promoted to a common type (user-definable) and use the specific implementation
- supports parametric methods

  ```
  myappend{T}(v::Vector{T}, x::T) = [v..., x]
  ```

Performance

- to prove that Julia is fast language, we did some tests

Performance

- to prove that Julia is fast language, we did some tests
- benchmarks sources taken from the Julia site and modified

Performance

- to prove that Julia is fast language, we did some tests
- benchmarks sources taken from the Julia site and modified
- all times are in milliseconds

Performance

- to prove that Julia is fast language, we did some tests
- benchmarks sources taken from the Julia site and modified
- all times are in milliseconds
- on a Z800 (8 threads, 24G of memory),

Performance

- to prove that Julia is fast language, we did some tests
- benchmarks sources taken from the Julia site and modified
- all times are in milliseconds
- on a Z800 (8 threads, 24G of memory),

Performance

- to prove that Julia is fast language, we did some tests
- benchmarks sources taken from the Julia site and modified
- all times are in milliseconds
- on a Z800 (8 threads, 24G of memory),

appli	fib	mandel	quicksort	pisum	randstat	randmul
Matlab	191	22	28	57	97	69
Octave	924	310	1138	21159	484	109
Python	4	7	14	1107	253	101
Pypy	8	3(faux)	13	44	xxx	xxx
Julia	0.09	0.28	0.57	45	34	49
Fortran	0.08	$\leqslant 10^{-6}$	0.62	44	16	275(16)

Conclusion

- pros :

Conclusion

- pros :
  - a true language, with lots of powerful features,

Conclusion

- pros :
    - a true language, with lots of powerful features,
    - Julia is fast!

Conclusion

- pros :
  - a true language, with lots of powerful features,
  - Julia is fast!
- cons :

Conclusion

- pros :
  - a true language, with lots of powerful features,
  - Julia is fast!
- cons :
  - poor graphics support (only 2D with additional package),

Conclusion

- pros :
  - a true language, with lots of powerful features,
  - Julia is fast!
- cons :
  - poor graphics support (only 2D with additional package),
  - no support for shared-memory parallelism

Conclusion

- pros :
  - a true language, with lots of powerful features,
  - Julia is fast!
- cons :
  - poor graphics support (only 2D with additional package),
  - no support for shared-memory parallelism
  - small community

Conclusion

- pros :
  - a true language, with lots of powerful features,
  - Julia is fast!
- cons :
  - poor graphics support (only 2D with additional package),
  - no support for shared-memory parallelism
  - small community
- some non presented points :

Conclusion

- pros :
  - a true language, with lots of powerful features,
  - Julia is fast!
- cons :
  - poor graphics support (only 2D with additional package),
  - no support for shared-memory parallelism
  - small community
- some non presented points :
  - meta-programming aspects : macros

Conclusion

- pros :
  - a true language, with lots of powerful features,
  - Julia is fast!
- cons :
  - poor graphics support (only 2D with additional package),
  - no support for shared-memory parallelism
  - small community
- some non presented points :
  - meta-programming aspects : macros
  - reflection

Conclusion

- pros :
  - a true language, with lots of powerful features,
  - Julia is fast!
- cons :
  - poor graphics support (only 2D with additional package),
  - no support for shared-memory parallelism
  - small community
- some non presented points :
  - meta-programming aspects : macros
  - reflection
  - packaging system based on Git

Conclusion

- pros :
  - a true language, with lots of powerful features,
  - Julia is fast!
- cons :
  - poor graphics support (only 2D with additional package),
  - no support for shared-memory parallelism
  - small community
- some non presented points :
  - meta-programming aspects : macros
  - reflection
  - packaging system based on Git
- more info at `http://julialang.org/`