Outline

1 motivations

2 Julia as a numerical language

3 types and methods

4 about performance
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 \texttt{Matlab} -like language?

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

→ Why do not try a new language for numerical computation?
A language for numerical computation

- Julia’s syntax is very similar to languages as matlab, Python or Scilab, so switching to Julia is fast.
A language for numerical computation

- Julia’s syntax is very similar to languages 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 languages 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
Julia’s syntax is very similar to languages as Matlab, Python or Scilab, so switching to Julia is fast. It does not require vectorized code to run fast (JIT compiler). It uses references (also for function arguments), indices start to 1 and finish to end, and uses brackets [ , ] for indexing.

A language for numerical computation
A language for numerical computation

- Julia’s syntax is very similar to languages 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 languages 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 languages 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 languages 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

→ let us have a look to some examples
Functional aspects

- support anonymous functions: \((x \rightarrow x \times x)(2)\)
Functional aspects

- support anonymous functions: \((x \rightarrow x \cdot x)(2)\)
- support map, reduce, filter functions
Functional aspects

- support anonymous functions: \((x \rightarrow x\ast x)(2)\)
- support map, reduce, filter functions
- functions support variadic arguments (using tuples)
Functional aspects

- support anonymous functions: \((x \rightarrow x \times x)(2)\)
- support map, reduce, filter functions
- functions support variadic arguments (using tuples)
- comprehension lists
Functional aspects

- support anonymous functions: \((x \rightarrow x \times x)(2)\)
- 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

- Julia has a built-in support for distributed memory parallelism.
- One-sided message passing routines.
- `remotecall` to launch a computation on a given process.
- `fetch` to retrieve information.
- High level routines: `@parallel` for lightweight iterations, `pmap` for heavy iterations.
- Support for distributed arrays in the standard library.
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

sometimes you need to call a C/Fortran code

"no boilerplate" philosophy: do not require Mexfiles, Swig

the code must be in a shared library

the syntax is the following:

ccall (: function , "lib"), return_type , (type_1 ,... , type_n), args)
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 \texttt{mexfiles}, Swig or other wrapping system
- the code must be in a \texttt{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

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

```
call (: function, "lib"), return_type, (type_1, ..., type_n), args
```
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

```
call(:function, "lib"), return_type, (type_1, ..., type_n), args)
```
There is a graph type in Julia reflecting the hierarchy of types

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

Julia supports 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
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:
  
  \[
  \text{None <: Int64 <: Number <: Real <: Any}
  \]

- 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
  
  $\text{None} <: \text{Int64} <: \text{Number} <: \text{Real} <: \text{Any}$

- support both abstract and concrete types

- user can annotate the code with operator $:\text{::}$ ”is an instance of”

- Julia supports
Types

- There is a graph type in Julia reflecting the hierarchy of types:
  
  \[
  \text{None <: Int64 <: Number <: Real <: Any}
  \]

- Support both abstract and concrete types.

- User can annotate the code with operator :: "is an instance of".

Julia supports:

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

\[
\text{None <: Int64 <: Number <: Real <: Any}
\]

- 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
  
  \[
  \text{None} <: \text{Int64} <: \text{Number} <: \text{Real} <: \text{Any}
  \]

- 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:
  
  \[
  \text{None} <: \text{Int64} <: \text{Number} <: \text{Real} <: \text{Any}
  \]

- 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

Main idea: define piecewisely methods or functions depending on their arguments types.

Let us define:

\[ f(x:: \text{Float64}, y:: \text{Float64}) = 2x + y \]
\[ f(x:: \text{Int}, y:: \text{Int}) = 2x + y \]
\[ f(2., 3.) \quad \text{returns} \quad 7.0 \]
\[ f(2, 3) \quad \text{returns} \quad 7.0 \]
\[ f(2, 3.) \quad \text{throw an ERROR: no method f(Int64, Float64)} \]

But if we define:

\[ g(x:: \text{Number}, y:: \text{Number}) = 2x + y \]
\[ g(2.0, 3) \quad \text{now returns} \quad 7.0 \]

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

Supports parametric methods:

\[ \text{myappend} \{T\}(v:: \text{Vector} \{T\}, x::T) = [v..., x] \]
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 $f$

\[
\begin{align*}
f(\text{x::} \text{Float64} , \text{y::} \text{Float64}) &= 2x + y \\
f(\text{x::} \text{Int} , \text{y::} \text{Int}) &= 2x + y \\
f(2. , 3.) &\text{ # returns } 7.0 \\
f(2, 3) &\text{ # returns } 7.0 \\
f(2, 3. ) &\text{ # throw an ERROR: no method } f(\text{Int64} , \text{Float64})
\end{align*}
\]
Multiple dispatch

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

\[
\begin{align*}
  f(x::\text{Float64}, y::\text{Float64}) &= 2x + y \\
  f(x::\text{Int}, y::\text{Int}) &= 2x + y \\
  f(2, 3.) &\text{ # returns 7.0} \\
  f(2, 3) &\text{ # returns 7.0} \\
  f(2, 3.) &\text{ # throw an ERROR: no method } f(\text{Int64}, \text{Float64})
\end{align*}
\]

but if we define g

\[
\begin{align*}
  g(x::\text{Number}, y::\text{Number}) &= 2x + y \\
  g(2.0, 3) &\text{ # now returns 7.0}
\end{align*}
\]
Multiple dispatch

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

  \[
  f(x::\text{Float64}, y::\text{Float64}) = 2x + y \\
  f(x::\text{Int}, y::\text{Int}) = 2x + y \\
  f(2.,3.) \# \text{returns 7.0} \\
  f(2,3) \# \text{returns 7.0} \\
  f(2, 3.) \# \text{throw an ERROR: no method f(Int64, Float64)}
  \]

  but if we define $g$

  \[
  g(x::\text{Number}, y::\text{Number}) = 2x + y \\
  g(2.0, 3) \# \text{now returns 7.0}
  \]

- 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 $f$

  $$
  f(x::\text{Float64}, y::\text{Float64}) = 2x + y \\
  f(x::\text{Int}, y::\text{Int}) = 2x + y \\
  f(2., 3.) \# \text{ returns 7.0} \\
  f(2, 3) \# \text{ returns 7.0} \\
  f(2, 3.) \# \text{ throw an ERROR: no method } f(\text{Int64}, \text{Float64})
  $$

  but if we define $g$

  $$
  g(x::\text{Number}, y::\text{Number}) = 2x + y \\
  g(2.0, 3) \# \text{ now returns 7.0}
  $$

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

  $$
  \text{myappend}\{\text{T}\}(v::\text{Vector}\{\text{T}\}, x::\text{T}) = [v..., x]
  $$
Performance

- to prove that Julia is a 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),

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

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

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

- **cons:**

- **some non presented points:**
  - meta-programming aspects: macros
  - reflection
  - packaging system based on Git

more info at [http://julialang.org/](http://julialang.org/)
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),

more info at [http://julialang.org/](http://julialang.org/)
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

more info at http://julialang.org/
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:**

more info at http://julialang.org/
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/