Julia for Scientific Programming

Seminar 3

Fatemeh Mohammadi

Dept. of Numerical Analysis
Lund University

Sep 4, 2015

Linear algebra

- Vectors
- Matrices
- Indexing and Slices
- Stacking and Concatenating
- Broadcasting
- Built-in functions

Vectors

In Julia, arrays are the most important data type which represent vectors or matrices in linear algebra.

```
julia> a=[1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> a=["foo","bar",1]
3-element Array{Any,1}:
  "foo"
  "bar"
 1
```

The 1 in Array{Int64,1} and Array{Any,1} indicates that the array is one dimensional.

```
julia> size(a)
(3,)

julia> ndims(a)
1
```

NOTE: The common terms "row vector" and "column vector" don't make sense in Julia. A vector in Julia is flat and hence neither row nor column. An n vector, an $n \times 1$ and a $1 \times n$ are three different objects.

```
julia> b=[-1,1,2.]
3-element Array{Float64,1}:
 -1.0
  1.0
  2.0
julia> b=[-1 1 2.]
1x3 Array{Float64,2}:
 -1.0  1.0  2.0
```

- Basic Operations

```
julia> 2a          # or 2*a
3-element Array{Int64,1}:
 2
 4
 6
julia> dot(a,a)   # or a'*a
1-element Array{Float64,1}:
14
julia> norm(a)
3.741657386773941
```

NOTE: Operations $.+, .-, .*$ are all elementwise.

- Creating vectors

  Some handy methods to quickly create vectors:

  zeros(n) creates a vector of size n filled with zeros.

  ones(n) is the same filled with ones

  rand(n) creates a vector with uniformly distributed random numbers between 0 and 1

  NOTE: We can create an empty array using Array():

```
julia> Array(Float64,3)
3-element Array{Float64,1}:
 7.94707e-316
 2.60867e-321
 0.0
```

What happens?

```
julia> a=[1,2,3];b=[-1,1,2.];

MATLAB>> a/b
ans =
   1.1667

Python>>> a/b
array([-1. ,  2. ,  1.5])
```

What happens?

```
julia> a=[1,2,3];b=[-1,1,2.];

MATLAB>> a/b
ans =
   1.1667

Python>>> a/b
array([-1. ,  2. ,  1.5])

julia> a/b
 3x3 Array{Float64,2}:
 -0.166667   0.166667   0.333333
 -0.333333   0.333333   0.666667
 -0.5        0.5        1.0
```

Answer:

```
julia> a=[1,2,3];b=[-1,1,2.];

MATLAB>> a/b     #a*pinv(b)
ans =
   1.1667

Python>>> a/b    # Elementwise division
array([-1. ,  2. ,  1.5])

julia> a/b    #a'*b(bb')^{-1}
 3x3 Array{Float64,2}:
 -0.166667   0.166667   0.333333
 -0.333333   0.333333   0.666667
 -0.5        0.5        1.0
```

Matrices

```
julia> A=[1. 0.;0. 1.]
2x2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0

julia> eye(2)
2x2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0

julia> ndims(A)
2

julia> size(A)
(2,2)
```

- Creating matrices

  Some convenient methods to create matrices are:

  eye(n) is the identity matrix of size n

  zeros(n, m) creates a matrix of size $n \times m$ filled with zeros.

  ones(n, m) fills an $n \times m$ matrix with ones

  rand(n, m) creates a matrix of size $n \times m$ with uniformly distributed random numbers between 0 and 1

- Creating matrices

  Some convenient methods to create matrices are:

  eye(n) is the identity matrix of size n

  zeros(n, m) creates a matrix of size $n \times m$ filled with zeros.

  ones(n, m) fills an $n \times m$ matrix with ones

  rand(n, m) creates a matrix of size $n \times m$ with uniformly distributed random numbers between 0 and 1

  NOTE: We can create an empty array using Array():

```
julia> Array(Float64,2,2)
2x2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
```

ndexing and Slices

```
julia> a[1]
1
```

```
julia> a[end]
3
```

```
julia> a[1:2]
2-element Array{Int64,1}:
 1
 2
```

```
julia> A[1,1]
1.0
```

```
julia> A[2,end]
1.0
```

```
julia> A[1:2,1]
2-element Array{Float64,1}:
 1.0
 0.0
```

Matrix coefficients are accessed by two indices

Stacking and Concatenating

```
julia> c="Hel"
"Hel"
julia> d="lo"
"lo"
julia> *(c,d)
"Hello"

julia> [a ;a[1,:]]
3x2 Array{Int64,2}:
 1  2
 3  4
 1  2

julia> [a a[:,1]]
2x3 Array{Int64,2}:
 1  2  1
 3  4  3
```

Broadcasting

It denotes the ability to guess a common, compatible shape between two arrays.

```
julia> a=[1,2,3,4]

julia> a+1
4-element Array{Int64,1}:
 2
 3
 4
 5

julia> a=[1. 0;0 1.]

julia> a+1
2x2 Array{Float64,2}:
 2.0  1.0
 1.0  2.0
```

Built-in Functions

Linear algebra functions in Julia are largely implemented by calling functions from LAPACK.

- dot(a,b): Compute the dot product
- cross(a,b): Compute the cross product of two 3-vectors.

```
julia> A=[1 2;0 1.];B=[-1 0;0 1];

julia> *(A,B)
2x2 Array{Float64,2}:
 -1.0  2.0
  0.0  1.0
```

- If A is a matrix and b is a vector, we can solve the linear equation

$$Ax = b$$

  using "\" which has the syntax $x = A\backslash b$.
- eig(A): Compute eigenvalues and eigenvectors of A
- rank(A): Compute the rank of matrix A
- etc