GLOBAL INITIATIVE OF ACADEMIC NETWORKS

Ivan Slapničar

MODERN APPLICATIONS OF NUMERICAL LINEAR ALGEBRA METHODS

Module A - Short Julia Course

IIT INDORE, 2016

GLOBAL INITIATIVE OF ACADEMIC NETWORKS

Ivan Slapničar

MODERN APPLICATIONS OF NUMERICAL LINEAR ALGEBRA METHODS

Module A - Short Julia Course

https://github.com/ivanslapnicar/GIAN-Applied-NLA-Course

Cover photo: Stata Center at MIT, home of Julia Group

IIT INDORE, 2016

Contents

1 Installing and Running Julia

This notebook describes the installation process for various components of the software.

1.0.1 Competences

The reader will be able to install `Julia` and all its components, to run `Julia` and start `IJulia`.

1.0.2 Suggested reading

Nice introductory text is at http://quant-econ.net/jl/learning_julia.html.

1.1 Installing Python and Jupyter

To install and use `IJulia` you need to install Python and Jupyter:

- download and install Anaconda and follow the instructions - this installs Python (*be sure to choose version 3.5*) and most popular Python packages, including `IPython`.

Alternatively, you can follow the instructions on the Jupyter Installation page.

1.2 Installing Julia

To install Julia download andextract prebuilt binary for your operating system - see Downloads. If you have sufficient expertise, you can download the Julia source and compiling it yourself - see Source Download and Compilation. You can also install the current Nigtly Build, but are adviced against it.
After instalation, you can start Julia in terminal mode by clicking its icon.

1.3 Installing and running `IJulia`

Do the following: * start Julia in terminal mode * at the `julia` prompt type

```
Pkg.add("IJulia")
Pkg.add("PyPlot")
using IJulia
notebook()
```

This opens IJulia window in your browser. (*The first two commands need to be executed only the first time!*). Semi-colon is the shell escape symbol, so, for example `; ls` gives directory listing.
Later, you can also start `IJulia` by executing command

```
jupyter notebook
```

in the command prompt.

1.4 Remarks

In Linux, packges are installed in the directory `$HOME/.julia/v0.4/`.
In Windows (10), you will have Julia icon which starts Julia command window. Julia is installed in the directory (`AppData` is a hidden directory):

```
\Users\your_user_name\AppData\Local\Julia-0.4.5
```

The packages are installed in a directory `\Users\your_user_name\.julia\v0.4` In Julia, current path and directory listing are obtained by Julia commands `pwd()` and `readdir()`, respectively.
Prior to executing `notebook()` command, you can use shell commands in `Julia` prompt to change directory, something like

```
; cd ../../my_julia_directory‘
```

```
In [ ]:
```

2 Lightning Round - Basic Features and Commands

In this notebook, we go through basic constructs and commands.

2.1 Competences

The user should know to start `Julia` in various modes (command line prompt, `IJulia`), how to exit, learn some features and be able to write simple programs.

2.2 Credits

This notebook is based on the slides accompanying the Lightning Round video by Alan Edelman, all part of the Julia Tutorial.

2.3 Julia resources

Julia resources are accessible through the Julia home page.
Please check `packages`, `docs` and `juliacon` (*here you will also find links to videos from previous conferences*).

2.4 Execution

To execute cell use `Shift + Enter` or press `Play` (**Run cell**).
To run all cells in the notebook go to `Cell -> Run All`

2.5 Markdown cells

Possibility to write comments / code / formulas in `Markdown` cells, makes Jupyter notebooks ideal for teaching and research. Text is written using *Julia Markdown*, which is *GitHub Markdown* with additional understanding of basic `LaTeX`.
Mastering (GitHub) Markdown is a 3-minute read, another short and very good manual is at http://daringfireball.net/projects/markdown/.
Some particulars of Julia Markdown are described in Documentation section of Julia Manual, yet another 3-minute read.

2.6 `nbconvert`

It is extremely easy to convert notebooks to slides, LaTeX, or PDF. For details see the documentation.

2.6.1 Slides

Clicking `View -> Cell Toolbar -> Slideshow` opens the `Slide Type` menu for each cell.
The slideshow is made with the command
`jupyter nbconvert --to slides notebook.ipynb`
The slideshow is displayed in browser with the command
`jupyter nbconvert --to slides --post serve notebook.ipynb`

2.6.2 LaTeX

`jupyter nbconvert --to latex notebook.ipynb`

2.6.3 PDF

`jupyter nbconvert --to PDF notebook.ipynb`
N.B. For the above conversions Pandoc needs to be installed.

2.7 Which version of `Julia` is running?

```
In [1]: versioninfo()
```

```
Julia Version 0.4.5
Commit 2ac304d (2016-03-18 00:58 UTC)
Platform Info:
  System: Linux (x86_64-unknown-linux-gnu)
  CPU: Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz
  WORD_SIZE: 64
  BLAS: libopenblas (USE64BITINT DYNAMIC_ARCH NO_AFFINITY Sandybridge)
  LAPACK: libopenblas64_
  LIBM: libopenlibm
  LLVM: libLLVM-3.3
```

2.8 Quitting

Exiting from `julia>` or **restarting kernel** in `IJulia`

```
In [2]: # exit()
```

2.9 Documentation

Documentation is well written and the starting point is http://docs.julialang.org/en/latest/
But, also remeber that `Julia` is **open source** and all routines are available on GitHub. You will learn how to make full use of this later in the course.

2.10 Punctuation review

- `[...]` are for indexing, array constructors and **Comprehensions**
- `(...)` are **required** for functions `quit()`, `tic()`, `toc()`, `help()`
- `{...}` are for arrays
- `#` is for comments

2.11 Basic indexing

```
In [3]: A=rand(5,5) # Matrix with random entries between 0 and 1
```

```
Out[3]: 5x5 Array{Float64,2}:
         0.589141   0.236581  0.784648  0.609539  0.277695
         0.434805   0.768152  0.981204  0.328945  0.542079
         0.674388   0.310239  0.444064  0.956564  0.379369
         0.59245    0.21946   0.685676  0.338136  0.0315785
         0.0814153  0.247983  0.186041  0.856269  0.192363
```

```
In [4]: A[1,1]

Out[4]: 0.5891406598959215

In [5]: rand(5,5)[1:2,3:4] # You can even do this

Out[5]: 2x2 Array{Float64,2}:
         0.610606  0.00812121
         0.39545   0.333569
```

2.11.1 Indexing is elegant

If you want to compute the lower right 2×2 block of A^{10}, in most languages you need to first compute $B = A^{10}$ and then index into B. In Julia, the command is simply

```
In [6]: (A^10)[4:5,4:5] # Parenthesses around A^10 are necessary

Out[6]: 2x2 Array{Float64,2}:
         1242.61   535.399
          914.326  393.953
```

2.11.2 Comprehensions - elegant array constructors

```
In [7]: [i for i=1:5]

Out[7]: 5-element Array{Int64,1}:
         1
         2
         3
         4
         5

In [8]: [trace(rand(n,n)) for n=1:5]

Out[8]: 5-element Array{Float64,1}:
         0.765719
         0.893854
         1.27039
         2.38561
         2.3785

In [9]: x=1:10

Out[9]: 1:10

In [10]: [ x[i]+x[i+1] for i=1:9 ]

Out[10]: 9-element Array{Any,1}:
           3
           5
           7
           9
```

```
         11
         13
         15
         17
         19

In [11]: z = [eye(n) for n=1:5]  # z is Array of Arrays

Out[11]: 5-element Array{Array{Float64,2},1}:
          1x1 Array{Float64,2}:
          1.0
          2x2 Array{Float64,2}:
          1.0  0.0
          0.0  1.0
          3x3 Array{Float64,2}:
          1.0  0.0  0.0
          0.0  1.0  0.0
          0.0  0.0  1.0
          4x4 Array{Float64,2}:
          1.0  0.0  0.0  0.0
          0.0  1.0  0.0  0.0
          0.0  0.0  1.0  0.0
          0.0  0.0  0.0  1.0
          5x5 Array{Float64,2}:
          1.0  0.0  0.0  0.0  0.0
          0.0  1.0  0.0  0.0  0.0
          0.0  0.0  1.0  0.0  0.0
          0.0  0.0  0.0  1.0  0.0
          0.0  0.0  0.0  0.0  1.0

In [12]: z[1] # First element is a 1x1 Array

Out[12]: 1x1 Array{Float64,2}:
          1.0

In [13]: z[4] # What is the fourth element?

Out[13]: 4x4 Array{Float64,2}:
          1.0  0.0  0.0  0.0
          0.0  1.0  0.0  0.0
          0.0  0.0  1.0  0.0
          0.0  0.0  0.0  1.0

In [14]: A=[ i+j for i=1:5, j=1:5 ] # Another example of a comprehension

Out[14]: 5x5 Array{Int64,2}:
          2  3  4  5   6
          3  4  5  6   7
          4  5  6  7   8
          5  6  7  8   9
          6  7  8  9  10
```

```
In [15]: B=[ i+j for i=1:5, j=1.0:5 ] # Notice the promotion

Out[15]: 5x5 Array{Float64,2}:
          2.0  3.0  4.0  5.0   6.0
          3.0  4.0  5.0  6.0   7.0
          4.0  5.0  6.0  7.0   8.0
          5.0  6.0  7.0  8.0   9.0
          6.0  7.0  8.0  9.0  10.0
```

2.12 Commands `ndims()` and `typeof()`

```
In [16]: ndims(ans)

Out[16]: 2

In [17]: ndims(z) # z is a one-dimensional array

Out[17]: 1

In [18]: typeof(z) # Array of Arrays

Out[18]: Array{Array{Float64,2},1}

In [19]: typeof(z[5]) # z[5] is a two-dimensional array

Out[19]: Array{Float64,2}

In [20]: typeof(A)

Out[20]: Array{Int64,2}
```

2.13 Vectors are 1-dimensional arrays

See Multi-dimensional arrays for more.

```
In [21]: v=rand(5,1) # This is 2-dimensional array

Out[21]: 5x1 Array{Float64,2}:
          0.32014
          0.552837
          0.0274691
          0.490916
          0.646462

In [22]: vv=vec(v) # This is an 1-dimensional array or vector

Out[22]: 5-element Array{Float64,1}:
          0.32014
          0.552837
          0.0274691
          0.490916
          0.646462
```

```
In [23]: v==vv  # Notice that they are different

Out[23]: false

In [24]: v-vv # Again a promotion

Out[24]: 5x1 Array{Float64,2}:
          0.0
          0.0
          0.0
          0.0
          0.0

In [25]: w=rand(5) # This is again a vector

Out[25]: 5-element Array{Float64,1}:
          0.343098
          0.922818
          0.650603
          0.0498243
          0.414088

In [26]: Mv=[v w] # First column is a 5 x 1 matrix, second column is a vector of length 5

Out[26]: 5x2 Array{Float64,2}:
          0.32014     0.343098
          0.552837    0.922818
          0.0274691   0.650603
          0.490916    0.0498243
          0.646462    0.414088

In [27]: x=Mv[:,1] # Matrix columns are extracted as vectors

Out[27]: 5-element Array{Float64,1}:
          0.32014
          0.552837
          0.0274691
          0.490916
          0.646462

In [28]: y=Mv[:,2]

Out[28]: 5-element Array{Float64,1}:
          0.343098
          0.922818
          0.650603
          0.0498243
          0.414088

In [29]: x==v # The types differ

Out[29]: false

In [30]: y==w

Out[30]: true
```

2.13.1 Sometimes brackets are needed

In [31]:
```
w=1.0:5
```

Out[31]:
```
1.0:1.0:5.0
```

In [32]:
```
A*w  # This returns an error
```

```
    LoadError: MethodError: `A_mul_B!` has no method matching A_mul_B!(::Array{Float64,1}
Closest candidates are:
  A_mul_B!(::Union{DenseArray{T,1},SubArray{T,1,A<:DenseArray{T,N},I<:Tuple{Vararg{Union
  A_mul_B!(::Union{AbstractArray{T,1},AbstractArray{T,2}}, !Matched::Tridiagonal{T}, ::U
  A_mul_B!(::Union{AbstractArray{T,1},AbstractArray{T,2}}, !Matched::Base.LinAlg.Abstrac
  ...
while loading In[32], in expression starting on line 1



     in * at linalg/matmul.jl:87
```

In [33]:
```
w=collect(1.0:5)
```

Out[33]:
```
5-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0
 5.0
```

In [34]:
```
A*w  # This returns a 1-dimensional array
```

Out[34]:
```
5-element Array{Float64,1}:
  70.0
  85.0
 100.0
 115.0
 130.0
```

In [35]:
```
A*v # This returns a 2-dimensional array - v is a 5 x 1 array
```

Out[35]:
```
5x1 Array{Float64,2}:
  8.74202
 10.7798
 12.8177
 14.8555
 16.8933
```

2.14 Discussion

Such behavior is due to the fact that `Julia` has vectors as a special type. __ Pros? Cons? __
What is matrix × vector?
What is the result of

$$C[i, j] = A[i, :] * B[:, j]$$

```
In [36]: B=[A[i,:]*A[:,j] for i=1:5, j=1:5]

Out[36]: 5x5 Array{Any,2}:
          [90]    [110]   [130]   [150]   [170]
          [110]   [135]   [160]   [185]   [210]
          [130]   [160]   [190]   [220]   [250]
          [150]   [185]   [220]   [255]   [290]
          [170]   [210]   [250]   [290]   [330]

In [37]: inv(B) # Why this this happen? How to resolve it?


        LoadError: MethodError: `one` has no method matching one(::Type{Any})
    while loading In[37], in expression starting on line 1




In [38]: B=[(A[i,:]*A[:,j])[1] for i=1:5, j=1:5] # (First) element of a vector is a number

Out[38]: 5x5 Array{Any,2}:
           90  110  130  150  170
          110  135  160  185  210
          130  160  190  220  250
          150  185  220  255  290
          170  210  250  290  330

In [39]: map(Int64,B) # Need to map it to `Int64`

Out[39]: 5x5 Array{Int64,2}:
           90  110  130  150  170
          110  135  160  185  210
          130  160  190  220  250
          150  185  220  255  290
          170  210  250  290  330
```

Or, we can use the dot product of two vectors - still need mapping of the comprehension to `Int64`

```
In [40]: B=[vec(A[i,:])⋅A[:,j] for i=1:5, j=1:5]

Out[40]: 5x5 Array{Any,2}:
           90  110  130  150  170
          110  135  160  185  210
          130  160  190  220  250
          150  185  220  255  290
          170  210  250  290  330
```

2.15 `ones()`, `eye()` and `zeros()`

Notice that the output type depends on the argument. This is a general Julia feature called `Multiple dispatch` and will be explained later in more detail.

```
In [41]: ones(3,5), ones(5), ones(rand(1:3,4,6))
         # The output type depends on the argument. Float64 is the default.
```

```
Out[41]: (
         3x5 Array{Float64,2}:
          1.0  1.0  1.0  1.0  1.0
          1.0  1.0  1.0  1.0  1.0
          1.0  1.0  1.0  1.0  1.0,

         [1.0,1.0,1.0,1.0,1.0],
         4x6 Array{Int64,2}:
          1  1  1  1  1  1
          1  1  1  1  1  1
          1  1  1  1  1  1
          1  1  1  1  1  1)
```

```
In [42]: rand(1:3,4,6)
```

```
Out[42]: 4x6 Array{Int64,2}:
          1  1  2  3  2  3
          2  1  2  3  1  3
          3  1  1  1  3  2
          1  3  1  1  3  1
```

```
In [43]: zeros(3,5), zeros(5), zeros(rand(1:3,4,6))
```

```
Out[43]: (
         3x5 Array{Float64,2}:
          0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0,

         [0.0,0.0,0.0,0.0,0.0],
         4x6 Array{Int64,2}:
          0  0  0  0  0  0
          0  0  0  0  0  0
          0  0  0  0  0  0
          0  0  0  0  0  0)
```

```
In [44]: eye(4), round(Int64,eye(4)), round(Int32,eye(4)), complex(eye(4))
         # type can also be set
```

```
Out[44]: (
         4x4 Array{Float64,2}:
          1.0  0.0  0.0  0.0
          0.0  1.0  0.0  0.0
          0.0  0.0  1.0  0.0
```

```
 0.0  0.0  0.0  1.0,

4x4 Array{Int64,2}:
 1  0  0  0
 0  1  0  0
 0  0  1  0
 0  0  0  1,

4x4 Array{Int32,2}:
 1  0  0  0
 0  1  0  0
 0  0  1  0
 0  0  0  1,

4x4 Array{Complex{Float64},2}:
 1.0+0.0im  0.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0+0.0im  1.0+0.0im  0.0+0.0im  0.0+0.0im
 0.0+0.0im  0.0+0.0im  1.0+0.0im  0.0+0.0im
 0.0+0.0im  0.0+0.0im  0.0+0.0im  1.0+0.0im)
```

2.16 Complex numbers

`i` is too valuable symbol for loops, so Julia uses `im` for the complex unit.

```
In [45]: im

Out[45]: im

In [46]: 2im

Out[46]: 0 + 2im

In [47]: typeof(ans)

Out[47]: Complex{Int64}

In [48]: typeof(2.0im)

Out[48]: Complex{Float64}

In [49]: complex(3,4) # Another way of defining complex numbers

Out[49]: 3 + 4im

In [50]: complex(3,4.0) # If one of the arguments if Float64, so is the entire number

Out[50]: 3.0 + 4.0im

In [51]: sqrt(-1) # This produces an error (like in any other language),
```

```
    LoadError: DomainError:
sqrt will only return a complex result if called with a complex argument. Try sqrt(compl
while loading In[51], in expression starting on line 1


     in sqrt at math.jl:146
```

```
In [52]: sqrt(complex(-1))  # and this is fine.

Out[52]: 0.0 + 1.0im
```

2.17 Ternary operator

Let us define our version of the sign function

```
In [53]: si(x) = (x>0) ? 1 : -1

Out[53]: si (generic function with 1 method)

In [54]: si(-13)

Out[54]: -1
```

This is equivalent to:

```
In [55]: function si(x)
             if x>0
                 return 1
             else
                 return -1
             end
         end

Out[55]: si (generic function with 1 method)

In [56]: si(pi-8), si(0), si(0.0)

Out[56]: (-1,-1,-1)
```

The expressions can be nested:

```
In [57]: si(x) = (x>0) ? 1 : ((x<0) ? -1: 0) # now si(0) is 0

Out[57]: si (generic function with 1 method)

In [58]: si(π-8), si(0) # '\pi Tab' produces π and means π

Out[58]: (-1,0)
```

2.18 Typing

Special mathematical (LaTeX) symbols can be used (like α, Ξ, π, $\oplus$, $\cdot$, etc.). The symbol in both, the notebook and command line version, is produced by writing LaTeX command followed by `Tab`

```
In [59]: Ξ = 8; Ψ  = 6; Γ = Ξ · Ψ
```

```
Out[59]: 48
```

```
In [60]: typeof(Γ)
```

```
Out[60]: Int64
```

2.19 Writing a program and running a file

Special feature of Julia is that the results of commands are not displayed, unless explicitely required.
To display results you can use commands `@show` or `println()` (or many others, see the Text I/O in the manual.)
Consider the file `deploy.jl` with the following code

```
n=int(ARGS[1])          # take one integer argument
println(rand(1:n,n,n))  # generate and print n x n matrix of random integers between 1 and 
@show b=3               # set b to 3 and show the result
c=4                     # set c to 4
```

Running the program in the shell gives

```
$ julia deploy.jl 5
[1 3 2 4 1
 5 3 1 1 4
 5 4 2 2 5
 3 1 2 3 4
 4 4 5 4 4]
b = 3 => 3
```

Notice that the result of the last command (c) is not displayed.

> You can, of course, also run the above command in the `Console` tab of `JuliaBox`.
> To do this, you first have to change the directory

```
cd Julia-Course/src
```

Similarly, the program can be converted to executable and run directly, without referencing `julia` in the command line. The refernece to `julia` must be added in the first line, as in the file `deploy1.jl`:

```
#!/usr/bin/julia
n=int(ARGS[1])
println(rand(1:n,n,n))
@show b=3
c=4
```

In the shell do:

```
$ chmod +x deploy1.jl
$ ./deploy1.jl 5
[4 5 3 2 5
 4 2 1 5 1
 3 2 4 5 1
 2 4 4 3 1
 3 4 5 3 3]
b = 3 => 3
```

Finally, to run the same program in `julia` shell or `IJulia`, the input has to be changed, as in the file `deploy2.jl`:

```
n=int(readline(STDIN))
println(rand(1:n,n,n))
@show b=3
c=4
```

Notice that now the result of the last line is displayed by default - in this case it is `4`, the values of `c`. The output of the random matrix and of `b` is forced.

```
In [61]: include("deploy2.jl")

STDIN> 5
[1 2 4 1 2
 2 2 2 5 4
 2 3 5 5 4
 1 3 3 4 3
 2 5 1 3 4]
b = 3 = 3

Out[61]: 4
```

2.20 Running external programs and unix pipe

2.20.1 `run()` - calling external program

```
In [62]: ?(run) # ?() is also a function - gives help

search: run trunc truncate itrunc round RoundUp RoundDown RoundToZero

Out[62]:

run(command)
```

Run a command object, constructed with backticks. Throws an error if anything goes wrong, including the process exiting with a non-zero status.

```
In [63]: ?run # parentheses can be ommited

search: run trunc truncate itrunc round RoundUp RoundDown RoundToZero
```

Out[63]:

```
run(command)
```

Run a command object, constructed with backticks. Throws an error if anything goes wrong, including the process exiting with a non-zero status.
Notice, that this is not a gret help, Julia has much better commands for this.

```
In [64]: run(`cal`) # This calls the unix Calendar program
```

```
May 2016
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

```
In [65]: run(pipeline(`cal`,`grep Sa`)) # The pipe is '|>' instead of usual '|'
```

```
Su Mo Tu We Th Fr Sa
```

2.20.2 ccall() - calling C program

```
In [66]: ?("ccall") # ccall is the only function which needs "" in ?() - but this is no muc
```

```
Base.Libdl.find_library
(intrinsic function #87)
Base.Libc.errno
```

```
In [67]: ccall(:ctime, Int, ()) # Simple version

Out[67]: 139849201859136

In [68]: bytestring(ccall(:ctime, Ptr{UInt8}, ())) # Human readable version

Out[68]: "Mon Jul 25 16:30:56 4433340\n"

In [69]: bytestring(ccall((:ctime,"libc"), Ptr{UInt8}, ()))
         # With specifying the library

Out[69]: "Mon Jul 25 18:01:36 4433340\n"

In [70]: ccall(:ctime, Ptr{UInt8}, ()) # Or with pointers

Out[70]: Ptr{UInt8} @0x00007f312dffba40
```

2.21 Task(), produce() and consume()

Julia has a control flow feature that allows computations to be suspended and resumed in a flexible manner (see tasks in the manual).

```
In [71]: function stepbystep()
             for n=1:3
                 produce(n^2)
             end
         end

Out[71]: stepbystep (generic function with 1 method)

In [72]: p=Task(stepbystep)

Out[72]: Task (runnable) @0x00007f2f2d777080

In [73]: consume(p)

Out[73]: 1

In [74]: consume(p)

Out[74]: 4

In [75]: consume(p)

Out[75]: 9

In [76]: consume(p) # Guess what comes next?

In [ ]:
```

3 Julia is Fast - `@time`, `@elapsed` and `@inbounds`

In this notebook, we demonstrate how fast `Julia` is, compared to other dynamically typed languages.

3.1 Prerequisites

Read the text Why Julia? (3 min)
Read Performance tips section of the `Julia` manual. (20 min)

3.2 Competences

The reader should understand effects of "just-in-time compiler" called LLVM on the speed of execution of programs. The reader should be able to write simple, but fast, programs containing loops.

3.3 Credits

Some examples are taken from The Julia Manual.

3.4 Scholarly example - summing integer halves

Consider the function `f` which sums halves of integers from `1` to `n`:
N.B. `Esc l` toggles the line numbers in the current cell.

```
In [1]: function f(n)
            s = 0
            for i = 1:n
                s += i/2
            end
            s
        end
```

```
Out[1]: f (generic function with 1 method)
```

In order for the fast execution, the function must first be compiled. Compilation is performed automatically, when the function is invoked for the first time. Therefore, the first call can be done with some trivial choice of parameters.
The timing can be done by two commands, `@time` and `@elapsed`:

```
In [2]: ?@time
```

```
Out[2]:
```

`@time`

A macro to execute an expression, printing the time it took to execute, the number of allocations, and the total number of bytes its execution caused to be allocated, before returning the value of the expression.

```
In [3]: ?@elapsed
```

```
Out[3]:
```

```
@elapsed
```

A macro to evaluate an expression, discarding the resulting value, instead returning the number of seconds it took to execute as a floating-point number.

```
In [4]: @time f(1)
```

```
0.007015 seconds (2.47 k allocations: 127.565 KB)
```

```
Out[4]: 0.5
```

```
In [5]: @elapsed f(1)  # This run is much faster, since the function is already compiled
```

```
Out[5]: 4.314e-6
```

Let us now run the big-size computation. Notice the unnaturally high byte allocation and the huge amount of time spent on garbage collection.

```
In [6]: @time f(1000000) # Notice the unnaturally high byte  allocation!
```

```
0.047478 seconds (2.00 M allocations: 30.518 MB, 16.67% gc time)
```

```
Out[6]: 2.5000025e11
```

```
In [7]: @elapsed f(1000000) # We shall be using @time from now on
```

```
Out[7]: 0.039124939
```

> Since your computer can execute several *Gigaflops* (floating-point operations per second), this is rather slow. This slowness is due to *type instability*: variable `s` is in the beginning assumed to be of type `Integer`, while at every other step, the result is a real number of type `Float64`. Permanent checking of types requires permanent memory allocation and deallocation (garbage collection). This is corrected by very simple means: just declare `s` as a real number, and the execution is more than 10 times faster with almost no memory allocation (and, consequently, no garbage collection).

```
In [8]: function f1(n)
            s = 0.0
            for i = 1:n
                s += i/2
            end
            s
        end
```

```
Out[8]: f1 (generic function with 1 method)
```

```
In [9]: @time f1(1)
```

```
0.005002 seconds (1.79 k allocations: 90.213 KB)

Out[9]: 0.5

In [10]: @time f1(1000000)

0.001592 seconds (5 allocations: 176 bytes)

Out[10]: 2.5000025e11
```

`@time` can alo be invoked as a function, but only on a function call, and not when the output is assigned, as well:

```
In [11]: @time(f1(1000000))

0.001322 seconds (5 allocations: 176 bytes)

Out[11]: 2.5000025e11

In [12]: @time s2=f1(1000000)

0.001781 seconds (6 allocations: 224 bytes)

Out[12]: 2.5000025e11

In [13]: @time(s2=f1(1000000))


        LoadError: unsupported or misplaced expression kw
    while loading In[13], in expression starting on line 155
```

3.5 Real-world example - exponential moving average

Exponential moving average is a fast *one pass* formula (each data point of the given data set A is accessed only once) often used in high-frequency on-line trading (see Online Algorithms in High-Frequency Trading for more details). **Notice that the output array X is declared in advance.**

Using `return` in the last line is here optional.

```
In [14]: function fexpma{T}( A::Vector{T}, alpha::T )
             # fast exponential moving average: X - moving average, A - data,
             # alpha - exponential forgetting parameter
             n = length(A)
             X = Array(T,n) # Declare X
             beta = one(T)-alpha
             X[1] = A[1]
             for k = 2:n
                 X[k] = beta*A[k] + alpha*X[k-1]
             end
             return X
         end
```

```
Out[14]: fexpma (generic function with 1 method)
```

```
In [15]: fexpma([1.0],0.5) # First run for compilation
```

```
Out[15]: 1-element Array{Float64,1}:
          1.0
```

We now generate some big-size data:

```
In [20]: # Big random slightly increasing sequence
         A=[rand() + 0.00001*k*rand() for k=1:20_000_000]
```

```
Out[20]: 20000000-element Array{Float64,1}:
            0.369455
            0.149719
            0.205221
            0.382511
            0.27614
            0.512635
            0.994414
            0.497099
            0.0377593
            0.70887
            0.262477
            0.789219
            0.817069
            ⋮
          186.359
           71.6288
            9.84393
          139.452
          106.447
          150.534
           57.7558
           32.4917
          183.647
          187.343
          131.351
          198.581
```

```
In [21]: @time X=fexpma(A,0.9)
```

```
0.236168 seconds (6 allocations: 152.588 MB)
```

```
Out[21]: 20000000-element Array{Float64,1}:
            0.369455
            0.347481
            0.333255
            0.338181
            0.331977
            0.350043
```

```
    0.41448
    0.422742
    0.384243
    0.416706
    0.401283
    0.440077
    0.477776
    ⋮
  100.28
   97.4149
   88.6578
   93.7372
   95.0082
  100.561
   96.2803
   89.9014
   99.276
  108.083
  110.41
  119.227
```

3.6 @inbounds

The `@inbounds` command eliminates array bounds checking within expressions. Be certain before doing this. If the subscripts are ever out of bounds, you may suffer crashes or silent corruption. The above program runs 40% faster.,

```
In [22]: function fexpma{T}( A::Vector{T}, alpha::T )
             # fast exponential moving average: X - moving average, A - data,
             # alpha - exponential forgetting parameter
             n = length(A)
             X = Array(T,n) # Declare X
             beta = one(T)-alpha
             X[1] = A[1]
             @inbounds for k = 2:n
                 X[k] = beta*A[k] + alpha*X[k-1]
             end
             return X
         end

Out[22]: fexpma (generic function with 1 method)

In [24]: @time X=fexpma(A,0.9)

0.137284 seconds (6 allocations: 152.588 MB)

Out[24]: 20000000-element Array{Float64,1}:
    0.369455
    0.347481
    0.333255
    0.338181
```

```
  0.331977
  0.350043
  0.41448
  0.422742
  0.384243
  0.416706
  0.401283
  0.440077
  0.477776
  ⋮
100.28
 97.4149
 88.6578
 93.7372
 95.0082
100.561
 96.2803
 89.9014
 99.276
108.083
110.41
119.227
```

Similar `Matlab` programs give the following timing for the two versions of the function, first *without* prior declaration of X and then *with* prior declaration. The *latter* version is several times faster, but still slow.

```
function X = fexpma( A,alpha )
% fast exponential moving average: X - moving average, A - data,
% alpha - exponential forgetting parameter
n=length(A);
X=zeros(n,1); % Allocate X in advance
beta=1-alpha;
X(1)=A(1);
for k=2:n
    X(k)=beta*A(k)+alpha*X(k-1);
end

>> tic, X=fexpma(A,0.9); toc
Elapsed time is 0.320976 seconds.
```

3.7 Plotting the moving average

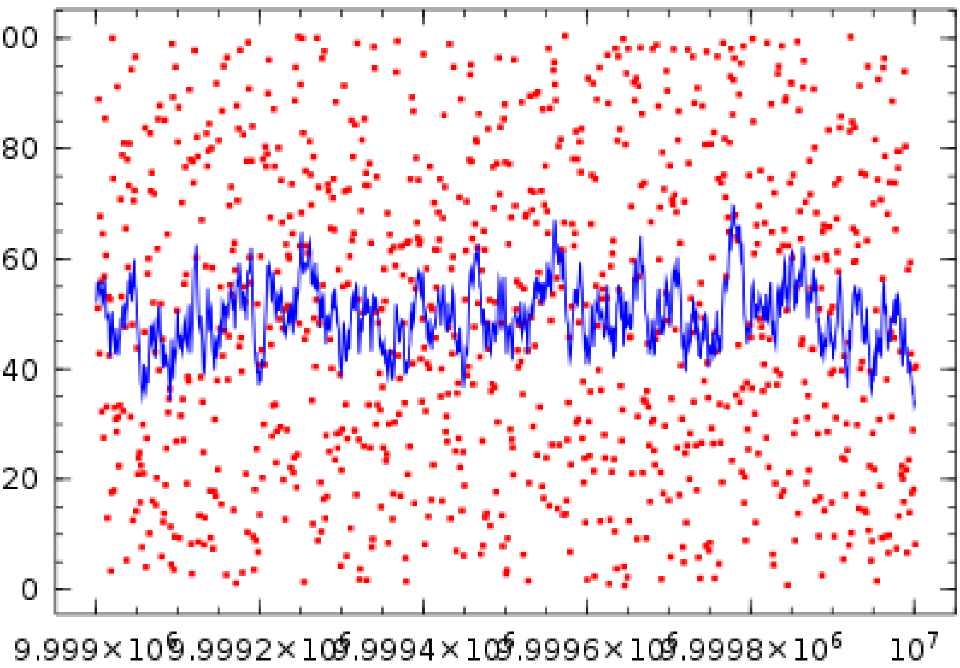
Let us plot the data A and its exponential moving average X. The dimension of the data is too large for meaningful direct plot. In `Julia` we can use `@manipulate` command to slide through the data. It takes a while to read packages `Winston` (for plotting) and `Interact`, but this is needed only for the first invocation.

```
In [26]: using Winston
         using Interact
```

```
In [27]: @manipulate for k=1:1000:20000000
             plot(collect(k:k+1000),A[k:k+1000],"r.",collect(k:k+1000),X[k:k+1000],"b")
         end
```

```
Interact.Slider{Int64}(Signal{Int64}(9999001, nactions=0),"k",9999001,1:1000:19999001,true)
```

Out[27]:



3.7.1 Remark

More details about optimizing your programs are given in the Profiling Notebook.

3.8 Pre-allocating output

The following example is from Pre-allocating outputs. The functions `loopinc()` and `loopinc_prealloc()` both compute $\sum_{i=2}^{10000001} i$, the second one being 10 times faster:

```
In [28]: function xinc(x)
             return [x, x+1, x+2]
         end

         function loopinc()
             y = 0
             for i = 1:10^7
                 ret = xinc(i)
                 y += ret[2]
             end
```

```
    y
end

function xinc!{T}(ret::AbstractVector{T}, x::T)
    ret[1] = x
    ret[2] = x+1
    ret[3] = x+2
    nothing
end

function loopinc_prealloc()
    ret = Array(Int, 3)
    y = 0
    for i = 1:10^7
        xinc!(ret, i)
        y += ret[2]
    end
    y
end
```

```
Out[28]: loopinc_prealloc (generic function with 1 method)
```

```
In [29]: @time loopinc()
```

```
0.960494 seconds (40.01 M allocations: 1.342 GB, 45.23% gc time)
```

```
Out[29]: 50000015000000
```

```
In [30]: @time loopinc_prealloc() # After the second run
```

```
0.037649 seconds (3.06 k allocations: 160.258 KB)
```

```
Out[30]: 50000015000000
```

3.9 Memory access

The following example is from Access arrays in memory order, along columns.

Multidimensional arrays in Julia are stored in column-major order, which means that arrays are stacked one column at a time. This convention for ordering arrays is common in many languages like Fortran, Matlab, and R (to name a few). The alternative to column-major ordering is row-major ordering, which is the convention adopted by C and Python (numpy) among other languages. The ordering can be verified using the `vec()` function or the syntax `[:]`:

```
In [31]: B = rand(0:9,4,3)
```

```
Out[31]: 4x3 Array{Int64,2}:
          9  8  7
          6  3  9
          6  1  1
          4  9  3
```

```
In [32]: B[:]

Out[32]: 12-element Array{Int64,1}:
          9
          6
          6
          4
          8
          3
          1
          9
          7
          9
          1
          3
```

```
In [33]: vec(B)

Out[33]: 12-element Array{Int64,1}:
          9
          6
          6
          4
          8
          3
          1
          9
          7
          9
          1
          3
```

The ordering of arrays can have significant performance effects when looping over arrays. Loops should be organized such that the subsequent accessed elements are close to each other in physical memory.

The following functions accept a `Vector` and and return a square `Array` with the rows or the columns filled with copies of the input vector, respectively.

```
In [34]: function copy_cols{T}(x::Vector{T})
             n = size(x, 1)
             out = Array(eltype(x), n, n)
             for i=1:n
                 out[:, i] = x
             end
             out
         end

         function copy_rows{T}(x::Vector{T})
             n = size(x, 1)
             out = Array(eltype(x), n, n)
             for i=1:n
                 out[i, :] = x
```

```
        end
        out
    end
```

Out[34]: copy_rows (generic function with 1 method)

In [35]:
```
copy_cols([1.0,2])
copy_rows([1.0,2])
```

Out[35]:
```
2x2 Array{Float64,2}:
 1.0  2.0
 1.0  2.0
```

In [36]:
```
x=rand(5000) # generate a random vector
```

Out[36]:
```
5000-element Array{Float64,1}:
 0.270683
 0.617161
 0.20085
 0.799526
 0.41825
 0.775518
 0.992601
 0.947305
 0.16775
 0.767546
 0.0377609
 0.313661
 0.934166
 ⋮
 0.955947
 0.413041
 0.470317
 0.805511
 0.224841
 0.789954
 0.100358
 0.594421
 0.864206
 0.873242
 0.162148
 0.702579
```

In [37]:
```
@time C=copy_cols(x)  # We generate a large matrix
```

0.467792 seconds (4.50 k allocations: 190.804 MB, 1.35% gc time)

Out[37]:
```
5000x5000 Array{Float64,2}:
 0.270683  0.270683  0.270683  …  0.270683  0.270683  0.270683
 0.617161  0.617161  0.617161     0.617161  0.617161  0.617161
 0.20085   0.20085   0.20085      0.20085   0.20085   0.20085
```

```
0.799526   0.799526   0.799526      0.799526   0.799526   0.799526
0.41825    0.41825    0.41825       0.41825    0.41825    0.41825
0.775518   0.775518   0.775518   …  0.775518   0.775518   0.775518
0.992601   0.992601   0.992601      0.992601   0.992601   0.992601
0.947305   0.947305   0.947305      0.947305   0.947305   0.947305
0.16775    0.16775    0.16775       0.16775    0.16775    0.16775
0.767546   0.767546   0.767546      0.767546   0.767546   0.767546
0.0377609  0.0377609  0.0377609  …  0.0377609  0.0377609  0.0377609
0.313661   0.313661   0.313661      0.313661   0.313661   0.313661
0.934166   0.934166   0.934166      0.934166   0.934166   0.934166
⋮                                ⋱
0.955947   0.955947   0.955947      0.955947   0.955947   0.955947
0.413041   0.413041   0.413041      0.413041   0.413041   0.413041
0.470317   0.470317   0.470317   …  0.470317   0.470317   0.470317
0.805511   0.805511   0.805511      0.805511   0.805511   0.805511
0.224841   0.224841   0.224841      0.224841   0.224841   0.224841
0.789954   0.789954   0.789954      0.789954   0.789954   0.789954
0.100358   0.100358   0.100358      0.100358   0.100358   0.100358
0.594421   0.594421   0.594421   …  0.594421   0.594421   0.594421
0.864206   0.864206   0.864206      0.864206   0.864206   0.864206
0.873242   0.873242   0.873242      0.873242   0.873242   0.873242
0.162148   0.162148   0.162148      0.162148   0.162148   0.162148
0.702579   0.702579   0.702579      0.702579   0.702579   0.702579
```

```
In [38]: @time D=copy_rows(x) # This is several times slower
```

```
0.346238 seconds (4.50 k allocations: 190.804 MB, 1.13% gc time)
```

```
Out[38]: 5000x5000 Array{Float64,2}:
 0.270683  0.617161  0.20085  0.799526  …  0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526  …  0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526  …  0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 ⋮                                      ⋱
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526  …  0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
 0.270683  0.617161  0.20085  0.799526  …  0.873242  0.162148  0.702579
```

```
0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
0.270683  0.617161  0.20085  0.799526     0.873242  0.162148  0.702579
```

3.9.1 Remark

There is also a built-in function `repmat()`:

```
In [39]: ?repmat
```

```
search: repmat
```

```
Out[39]:
```

`repmat(A, n, m)`

Construct a matrix by repeating the given matrix `n` times in dimension 1 and `m` times in dimension 2.

```
In [40]: @time C1=repmat(x,1,5000)
```

```
0.447391 seconds (60.16 k allocations: 193.389 MB, 18.40% gc time)
```

```
Out[40]: 5000x5000 Array{Float64,2}:
 0.270683   0.270683   0.270683   ...  0.270683   0.270683   0.270683
 0.617161   0.617161   0.617161        0.617161   0.617161   0.617161
 0.20085    0.20085    0.20085         0.20085    0.20085    0.20085
 0.799526   0.799526   0.799526        0.799526   0.799526   0.799526
 0.41825    0.41825    0.41825         0.41825    0.41825    0.41825
 0.775518   0.775518   0.775518   ...  0.775518   0.775518   0.775518
 0.992601   0.992601   0.992601        0.992601   0.992601   0.992601
 0.947305   0.947305   0.947305        0.947305   0.947305   0.947305
 0.16775    0.16775    0.16775         0.16775    0.16775    0.16775
 0.767546   0.767546   0.767546        0.767546   0.767546   0.767546
 0.0377609  0.0377609  0.0377609  ...  0.0377609  0.0377609  0.0377609
 0.313661   0.313661   0.313661        0.313661   0.313661   0.313661
 0.934166   0.934166   0.934166        0.934166   0.934166   0.934166
 ⋮                                ⋱
 0.955947   0.955947   0.955947        0.955947   0.955947   0.955947
 0.413041   0.413041   0.413041        0.413041   0.413041   0.413041
 0.470317   0.470317   0.470317   ...  0.470317   0.470317   0.470317
 0.805511   0.805511   0.805511        0.805511   0.805511   0.805511
 0.224841   0.224841   0.224841        0.224841   0.224841   0.224841
 0.789954   0.789954   0.789954        0.789954   0.789954   0.789954
 0.100358   0.100358   0.100358        0.100358   0.100358   0.100358
 0.594421   0.594421   0.594421   ...  0.594421   0.594421   0.594421
 0.864206   0.864206   0.864206        0.864206   0.864206   0.864206
 0.873242   0.873242   0.873242        0.873242   0.873242   0.873242
 0.162148   0.162148   0.162148        0.162148   0.162148   0.162148
 0.702579   0.702579   0.702579        0.702579   0.702579   0.702579
```

```
In [35]:
```

4 Julia is Open - `whos()`, `methods()`, `@which`, ...

`Julia` is an open-source project, source being entirely hosted on `github`: http://github.com/julialang

The code consists of (actual numbers may differ):

- 29K lines of `C/C++`
- 6K lines of `scheme`
- 68K lines of `julia`

Julia uses LLVM which itself has 680K lines of code. Therefore, `Julia` is very compact, compared to other languages, like LLVM's `C` compiler `clang` (513K lines of code) or `gcc` (3,530K lines). This makes it easy to read the actuall code and get full information, in spite the fact that some parts of the documentation are insufficient. `Julia`'s "navigating" system, consisting of commands `whos()`, `methods()` and `@which`, makes this even easier.

Further, the `Base` (core) of `Julia` is kept small, and the rest of the functionality is obtained through packages. Since packages are written in `Julia`, they are navigated on the same way.

In this notebook, we demonstrate how to get help and navigate the source code.

4.1 Prerequisites

Basic knowledge of programming in any language.

Read Methods section of the `Julia` manual. (5 min)

4.2 Competences

The reader should be able to read the code and be able to find and understand calling sequences and outputs of any function.

4.3 Credits

Some examples are taken from The Julia Manual.

4.4 Operators +, * and ·

Consider operators `+`, `*` and ·, the first two of them seem rather basic in any language. The · symbol is typed as LaTeX command `\cdot` + `Tab`.

`?+` gives some information, which is vary sparse. We would expect more details, and we also suspect that `+` can be used in more ways that just hose two.

`?*` explaind more instances where `*` can be used, but the text itself is vague and not sufficient.

`?·` appears to be what we expect fro the dot product off two vectors.

```
In [1]: ?+
```

```
search: + .+
```

```
Out[1]:
```

```
+(x, y...)
```

Addition operator. `x+y+z+...` calls this function with all arguments, i.e. `+(x, y, z, ...)`.

```
In [2]: ?*
```

```
search: * .*
```

```
Out[2]:
```

```
*(x, y...)
```

Multiplication operator. `x*y*z*...` calls this function with all arguments, i.e. `*(x, y, z, ...)`.

```
*(s, t)
```

Concatenate strings. The `*` operator is an alias to this function.

```
julia> "Hello " * "world"
"Hello world"
```

```
*(A, B)
```

Matrix multiplication

```
In [3]: ?·
```

```
search: ·
```

```
Out[3]:
```

```
dot(x, y)
·(x,y)
```

Compute the dot product. For complex vectors, the first vector is conjugated.

4.5 methods()

`Julia` functions have a feature called *multiple dispatch*, which means that the method depends on the name **AND** the input. Full range of existing methods for certain function name is given by the `methods()` command. > Running `methods(+)` sheds a completely differfent light on `+`. The great `IJulia` feature is that the links to the source code where the respective version of the function is defined, are readily provided.

```
In [4]: ?methods
```

search: methods methodswith method_exists Method MethodTable MethodError

Out[4]:

methods(f, [types])

Returns the method table for `f`. If `types` is specified, returns an array of methods whose types match.

4.5.1 The "+" operator

N.B. For convenience, Left click on the left area of the `Out[]` cell toggles scrolling. Double click collapses the output completely.

In [5]: methods(+)

Out[5]:
```
# 171 methods for generic function "+":
+(x::Bool) at bool.jl:33
+(x::Bool, y::Bool) at bool.jl:36
+(y::AbstractFloat, x::Bool) at bool.jl:46
+(x::Int64, y::Int64) at int.jl:8
+(x::Int8, y::Int8) at int.jl:16
+(x::UInt8, y::UInt8) at int.jl:16
+(x::Int16, y::Int16) at int.jl:16
+(x::UInt16, y::UInt16) at int.jl:16
+(x::Int32, y::Int32) at int.jl:16
+(x::UInt32, y::UInt32) at int.jl:16
+(x::UInt64, y::UInt64) at int.jl:16
+(x::Int128, y::Int128) at int.jl:16
+(x::UInt128, y::UInt128) at int.jl:16
+(x::Integer, y::Ptr{T}) at pointer.jl:77
+(x::Float32, y::Float32) at float.jl:207
+(x::Float64, y::Float64) at float.jl:208
+(z::Complex{T<:Real}, w::Complex{T<:Real}) at complex.jl:111
+(x::Bool, z::Complex{Bool}) at complex.jl:118
+(z::Complex{Bool}, x::Bool) at complex.jl:119
+(x::Bool, z::Complex{T<:Real}) at complex.jl:125
+(z::Complex{T<:Real}, x::Bool) at complex.jl:126
+(x::Real, z::Complex{Bool}) at complex.jl:132
+(z::Complex{Bool}, x::Real) at complex.jl:133
+(x::Real, z::Complex{T<:Real}) at complex.jl:144
+(z::Complex{T<:Real}, x::Real) at complex.jl:145
+(x::Rational{T<:Integer}, y::Rational{T<:Integer}) at rational.jl:179
+(x::Bool, A::AbstractArray{Bool,N}) at arraymath.jl:136
+(x::Integer, y::Char) at char.jl:43
+(a::Float16, b::Float16) at float16.jl:136
+(x::BigInt, y::BigInt) at gmp.jl:256
+(a::BigInt, b::BigInt, c::BigInt) at gmp.jl:279
+(a::BigInt, b::BigInt, c::BigInt, d::BigInt) at gmp.jl:285
+(a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt) at gmp.jl:292
```

```
+(x::BigInt, c::Union{UInt16,UInt32,UInt64,UInt8}) at gmp.jl:304
+(c::Union{UInt16,UInt32,UInt64,UInt8}, x::BigInt) at gmp.jl:308
+(x::BigInt, c::Union{Int16,Int32,Int64,Int8}) at gmp.jl:320
+(c::Union{Int16,Int32,Int64,Int8}, x::BigInt) at gmp.jl:321
+(x::BigFloat, y::BigFloat) at mpfr.jl:208
+(x::BigFloat, c::Union{UInt16,UInt32,UInt64,UInt8}) at mpfr.jl:215
+(c::Union{UInt16,UInt32,UInt64,UInt8}, x::BigFloat) at mpfr.jl:219
+(x::BigFloat, c::Union{Int16,Int32,Int64,Int8}) at mpfr.jl:223
+(c::Union{Int16,Int32,Int64,Int8}, x::BigFloat) at mpfr.jl:227
+(x::BigFloat, c::Union{Float16,Float32,Float64}) at mpfr.jl:231
+(c::Union{Float16,Float32,Float64}, x::BigFloat) at mpfr.jl:235
+(x::BigFloat, c::BigInt) at mpfr.jl:239
+(c::BigInt, x::BigFloat) at mpfr.jl:243
+(a::BigFloat, b::BigFloat, c::BigFloat) at mpfr.jl:379
+(a::BigFloat, b::BigFloat, c::BigFloat, d::BigFloat) at mpfr.jl:385
+(a::BigFloat, b::BigFloat, c::BigFloat, d::BigFloat, e::BigFloat) at mpfr.jl:392
+(x::Irrational{sym}, y::Irrational{sym}) at irrationals.jl:72
+(x::Number) at operators.jl:73
+{T<:Number}(x::T<:Number, y::T<:Number) at promotion.jl:211
+{T<:AbstractFloat}(x::Bool, y::T<:AbstractFloat) at bool.jl:43
+(x::Number, y::Number) at promotion.jl:167
+(r1::OrdinalRange{T,S}, r2::OrdinalRange{T,S}) at operators.jl:330
+{T<:AbstractFloat}(r1::FloatRange{T<:AbstractFloat}, r2::FloatRange{T<:AbstractFloa
+{T<:AbstractFloat}(r1::LinSpace{T<:AbstractFloat}, r2::LinSpace{T<:AbstractFloat})
+(r1::Union{FloatRange{T<:AbstractFloat},LinSpace{T<:AbstractFloat},OrdinalRange{T,S
+(x::Ptr{T}, y::Integer) at pointer.jl:75
+{S,T}(A::Range{S}, B::Range{T}) at arraymath.jl:69
+{S,T}(A::Range{S}, B::AbstractArray{T,N}) at arraymath.jl:87
+(A::BitArray{N}, B::BitArray{N}) at bitarray.jl:834
+{T}(B::BitArray{2}, J::UniformScaling{T}) at linalg/uniformscaling.jl:28
+(A::Array{T,2}, B::Diagonal{T}) at linalg/special.jl:122
+(A::Array{T,2}, B::Bidiagonal{T}) at linalg/special.jl:122
+(A::Array{T,2}, B::Tridiagonal{T}) at linalg/special.jl:122
+(A::Array{T,2}, B::SymTridiagonal{T}) at linalg/special.jl:131
+(A::Array{T,2}, B::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}) at lina
+(A::Array{T,N}, B::SparseMatrixCSC{Tv,Ti<:Integer}) at sparse/sparsematrix.jl:1019
+{P<:Union{Base.Dates.CompoundPeriod,Base.Dates.Period}}(x::Union{DenseArray{P<:Unio
+(A::AbstractArray{Bool,N}, x::Bool) at arraymath.jl:135
+(A::Union{DenseArray{Bool,N},SubArray{Bool,N,A<:DenseArray{T,N},I<:Tuple{Vararg{Uni
+(A::SymTridiagonal{T}, B::SymTridiagonal{T}) at linalg/tridiag.jl:84
+(A::Tridiagonal{T}, B::Tridiagonal{T}) at linalg/tridiag.jl:404
+(A::UpperTriangular{T,S<:AbstractArray{T,2}}, B::UpperTriangular{T,S<:AbstractArray
+(A::LowerTriangular{T,S<:AbstractArray{T,2}}, B::LowerTriangular{T,S<:AbstractArray
+(A::UpperTriangular{T,S<:AbstractArray{T,2}}, B::Base.LinAlg.UnitUpperTriangular{T,
+(A::LowerTriangular{T,S<:AbstractArray{T,2}}, B::Base.LinAlg.UnitLowerTriangular{T,
+(A::Base.LinAlg.UnitUpperTriangular{T,S<:AbstractArray{T,2}}, B::UpperTriangular{T,
+(A::Base.LinAlg.UnitLowerTriangular{T,S<:AbstractArray{T,2}}, B::LowerTriangular{T,
+(A::Base.LinAlg.UnitUpperTriangular{T,S<:AbstractArray{T,2}}, B::Base.LinAlg.UnitUp
+(A::Base.LinAlg.UnitLowerTriangular{T,S<:AbstractArray{T,2}}, B::Base.LinAlg.UnitLo
+(A::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}, B::Base.LinAlg.Abstrac
+(Da::Diagonal{T}, Db::Diagonal{T}) at linalg/diagonal.jl:86
```

```
+(A::Bidiagonal{T}, B::Bidiagonal{T}) at linalg/bidiag.jl:176
+(UL::UpperTriangular{T,S<:AbstractArray{T,2}}, J::UniformScaling{T<:Number}) at lin
+(UL::Base.LinAlg.UnitUpperTriangular{T,S<:AbstractArray{T,2}}, J::UniformScaling{T<
+(UL::LowerTriangular{T,S<:AbstractArray{T,2}}, J::UniformScaling{T<:Number}) at lin
+(UL::Base.LinAlg.UnitLowerTriangular{T,S<:AbstractArray{T,2}}, J::UniformScaling{T<
+(A::Diagonal{T}, B::Bidiagonal{T}) at linalg/special.jl:121
+(A::Bidiagonal{T}, B::Diagonal{T}) at linalg/special.jl:122
+(A::Diagonal{T}, B::Tridiagonal{T}) at linalg/special.jl:121
+(A::Tridiagonal{T}, B::Diagonal{T}) at linalg/special.jl:122
+(A::Diagonal{T}, B::Array{T,2}) at linalg/special.jl:121
+(A::Bidiagonal{T}, B::Tridiagonal{T}) at linalg/special.jl:121
+(A::Tridiagonal{T}, B::Bidiagonal{T}) at linalg/special.jl:122
+(A::Bidiagonal{T}, B::Array{T,2}) at linalg/special.jl:121
+(A::Tridiagonal{T}, B::Array{T,2}) at linalg/special.jl:121
+(A::SymTridiagonal{T}, B::Tridiagonal{T}) at linalg/special.jl:130
+(A::Tridiagonal{T}, B::SymTridiagonal{T}) at linalg/special.jl:131
+(A::SymTridiagonal{T}, B::Array{T,2}) at linalg/special.jl:130
+(A::Diagonal{T}, B::SymTridiagonal{T}) at linalg/special.jl:139
+(A::SymTridiagonal{T}, B::Diagonal{T}) at linalg/special.jl:140
+(A::Bidiagonal{T}, B::SymTridiagonal{T}) at linalg/special.jl:139
+(A::SymTridiagonal{T}, B::Bidiagonal{T}) at linalg/special.jl:140
+(A::Diagonal{T}, B::UpperTriangular{T,S<:AbstractArray{T,2}}) at linalg/special.jl:
+(A::UpperTriangular{T,S<:AbstractArray{T,2}}, B::Diagonal{T}) at linalg/special.jl:
+(A::Diagonal{T}, B::Base.LinAlg.UnitUpperTriangular{T,S<:AbstractArray{T,2}}) at li
+(A::Base.LinAlg.UnitUpperTriangular{T,S<:AbstractArray{T,2}}, B::Diagonal{T}) at li
+(A::Diagonal{T}, B::LowerTriangular{T,S<:AbstractArray{T,2}}) at linalg/special.jl:
+(A::LowerTriangular{T,S<:AbstractArray{T,2}}, B::Diagonal{T}) at linalg/special.jl:
+(A::Diagonal{T}, B::Base.LinAlg.UnitLowerTriangular{T,S<:AbstractArray{T,2}}) at li
+(A::Base.LinAlg.UnitLowerTriangular{T,S<:AbstractArray{T,2}}, B::Diagonal{T}) at li
+(A::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}, B::SymTridiagonal{T})
+(A::SymTridiagonal{T}, B::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}})
+(A::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}, B::Tridiagonal{T}) at
+(A::Tridiagonal{T}, B::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}) at
+(A::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}, B::Bidiagonal{T}) at l
+(A::Bidiagonal{T}, B::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}) at l
+(A::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}, B::Array{T,2}) at lina
+{Tv1,Ti1,Tv2,Ti2}(A_1::SparseMatrixCSC{Tv1,Ti1}, A_2::SparseMatrixCSC{Tv2,Ti2}) at s
+(A::SparseMatrixCSC{Tv,Ti<:Integer}, B::Array{T,N}) at sparse/sparsematrix.jl:1017
+(A::SparseMatrixCSC{Tv,Ti<:Integer}, J::UniformScaling{T<:Number}) at sparse/sparse
+{P<:Union{Base.Dates.CompoundPeriod,Base.Dates.Period}}(Y::Union{DenseArray{P<:Unio
+{P<:Union{Base.Dates.CompoundPeriod,Base.Dates.Period},Q<:Union{Base.Dates.Compound
+{T<:Base.Dates.TimeType,P<:Union{Base.Dates.CompoundPeriod,Base.Dates.Period}}(x::U
+{T<:Base.Dates.TimeType}(r::Range{T<:Base.Dates.TimeType}, x::Base.Dates.Period) at
+{T<:Number}(x::AbstractArray{T<:Number,N}) at abstractarraymath.jl:49
+{S,T}(A::AbstractArray{S,N}, B::Range{T}) at arraymath.jl:78
+{S,T}(A::AbstractArray{S,N}, B::AbstractArray{T,N}) at arraymath.jl:96
+(A::AbstractArray{T,N}, x::Number) at arraymath.jl:139
+(x::Number, A::AbstractArray{T,N}) at arraymath.jl:140
+(x::Char, y::Integer) at char.jl:42
+{N}(index1::CartesianIndex{N}, index2::CartesianIndex{N}) at multidimensional.jl:42
+(J1::UniformScaling{T<:Number}, J2::UniformScaling{T<:Number}) at linalg/uniformsca
```

```
+(J::UniformScaling{T<:Number}, B::BitArray{2}) at linalg/uniformscaling.jl:29
+(J::UniformScaling{T<:Number}, A::AbstractArray{T,2}) at linalg/uniformscaling.jl:3
+(J::UniformScaling{T<:Number}, x::Number) at linalg/uniformscaling.jl:31
+(x::Number, J::UniformScaling{T<:Number}) at linalg/uniformscaling.jl:32
+{TA,TJ}(A::AbstractArray{TA,2}, J::UniformScaling{TJ}) at linalg/uniformscaling.jl:
+{T}(a::Base.Pkg.Resolve.VersionWeights.HierarchicalValue{T}, b::Base.Pkg.Resolve.Ve
+(a::Base.Pkg.Resolve.VersionWeights.VWPreBuildItem, b::Base.Pkg.Resolve.VersionWei
+(a::Base.Pkg.Resolve.VersionWeights.VWPreBuild, b::Base.Pkg.Resolve.VersionWeights
+(a::Base.Pkg.Resolve.VersionWeights.VersionWeight, b::Base.Pkg.Resolve.VersionWeig
+(a::Base.Pkg.Resolve.MaxSum.FieldValues.FieldValue, b::Base.Pkg.Resolve.MaxSum.Fie
+{P<:Base.Dates.Period}(x::P<:Base.Dates.Period, y::P<:Base.Dates.Period) at dates/p
+(x::Base.Dates.Period, y::Base.Dates.Period) at dates/periods.jl:190
+(x::Base.Dates.CompoundPeriod, y::Base.Dates.Period) at dates/periods.jl:191
+(y::Base.Dates.Period, x::Base.Dates.CompoundPeriod) at dates/periods.jl:192
+(x::Base.Dates.CompoundPeriod, y::Base.Dates.CompoundPeriod) at dates/periods.jl:1
+(x::Base.Dates.CompoundPeriod, y::Base.Dates.TimeType) at dates/periods.jl:238
+(y::Base.Dates.Period, x::Base.Dates.TimeType) at dates/arithmetic.jl:66
+{T<:Base.Dates.TimeType}(x::Base.Dates.Period, r::Range{T<:Base.Dates.TimeType}) at
+(x::Union{Base.Dates.CompoundPeriod,Base.Dates.Period}) at dates/periods.jl:201
+{P<:Union{Base.Dates.CompoundPeriod,Base.Dates.Period}}(x::Union{Base.Dates.Compou
+(dt::DateTime, y::Base.Dates.Year) at dates/arithmetic.jl:13
+(dt::Date, y::Base.Dates.Year) at dates/arithmetic.jl:17
+(dt::DateTime, z::Base.Dates.Month) at dates/arithmetic.jl:37
+(dt::Date, z::Base.Dates.Month) at dates/arithmetic.jl:43
+(x::Date, y::Base.Dates.Week) at dates/arithmetic.jl:60
+(x::Date, y::Base.Dates.Day) at dates/arithmetic.jl:62
+(x::DateTime, y::Base.Dates.Period) at dates/arithmetic.jl:64
+(x::Base.Dates.TimeType) at dates/arithmetic.jl:8
+(a::Base.Dates.TimeType, b::Base.Dates.Period, c::Base.Dates.Period) at dates/peri
+(a::Base.Dates.TimeType, b::Base.Dates.Period, c::Base.Dates.Period, d::Base.Dates
+(x::Base.Dates.TimeType, y::Base.Dates.CompoundPeriod) at dates/periods.jl:233
+(x::Base.Dates.Instant) at dates/arithmetic.jl:4
+{T<:Base.Dates.TimeType}(x::AbstractArray{T<:Base.Dates.TimeType,N}, y::Union{Base.
+{T<:Base.Dates.TimeType}(y::Union{Base.Dates.CompoundPeriod,Base.Dates.Period}, x::
+{P<:Union{Base.Dates.CompoundPeriod,Base.Dates.Period}}(y::Base.Dates.TimeType, x::
+(a, b, c, xs...) at operators.jl:103
```

Following the first link, we get the following code snippet:

```
+(x::Bool) = int(x)
-(x::Bool) = -int(x)
+(x::Bool, y::Bool) = int(x) + int(y)
-(x::Bool, y::Bool) = int(x) - int(y)
*(x::Bool, y::Bool) = x & y
```

Therefore:

```
In [6]: +(true), +(false),-(true),-(false)
```

```
Out[6]: (1,0,-1,0)
```

```
In [7]: x, y = bitpack([0,1,0,1,0,1]), bitpack([0,1,1,1,1,0])
```

```
Out[7]: (Bool[false,true,false,true,false,true],Bool[false,true,true,true,true,false])
```

The above command is equivalent to

```
x = bitpack([0,1,0,1,0,1]); y = bitpack([0,1,1,1,1,0])
```

except that only the last result would be displayed

```
In [8]: +x,-(x)

Out[8]: (Bool[false,true,false,true,false,true],[0,-1,0,-1,0,-1])

In [9]: x+y, +(x,y)

Out[9]: ([0,2,1,2,1,1],[0,2,1,2,1,1])

In [10]: c1=x+y

Out[10]: 6-element Array{Int64,1}:
          0
          2
          1
          2
          1
          1

In [11]: c2=+(x,y)

Out[11]: 6-element Array{Int64,1}:
          0
          2
          1
          2
          1
          1
```

4.5.2 Manipulating dates

We see that one of the + methods is adding days to time:

```
+(x::Date,y::Base.Dates.Day) at dates/arithmetic.jl:60
```

Therefore, the 135-th day from today is:

```
In [12]: Dates.today()

Out[12]: 2016-05-24

In [13]: dd=Dates.today()+Dates.Day(135)

Out[13]: 2016-10-06

In [14]: typeof(dd)

Out[14]: Date
```

More information about the two types can be obtained by `methods(Dates.Date)` and `methods(Dates.Day)`, respectively.

4.5.3 Adding tridiagonal matrices

In the above output of `methods(+)`, we see that we can add tridiagonal matrices:

```
+(A::Tridiagonal{T}, B::Tridiagonal{T}) at linalg/tridiag.jl:404
```

Following the link, we see that the method separately adds lower, main and upper diagonals, denoted by `dl`, `d` and `du`, respectively:

```
404: +(A::Tridiagonal, B::Tridiagonal) = Tridiagonal(A.dl+B.dl, A.d+B.d, A.du+B.du)
```

Let us see how exactly is the type `Tridiagonal` defined:

```
In [15]: methods(Tridiagonal)
```

```
Out[15]: 6-element Array{Any,1}:
          call{T}(::Type{Tridiagonal{T}}, dl::Array{T,1}, d::Array{T,1}, du::Array{T,1}, du2
          call{T}(::Type{Tridiagonal{T}}, dl::Array{T,1}, d::Array{T,1}, du::Array{T,1}) at
          call{Tl,Td,Tu}(::Type{Tridiagonal{T}}, dl::Array{Tl,1}, d::Array{Td,1}, du::Array{
          call{T}(::Type{Tridiagonal{T}}, M::Bidiagonal{T}) at linalg/bidiag.jl:63
          call{T}(::Type{T}, arg) at essentials.jl:56
          call{T}(::Type{T}, args...) at essentials.jl:57
```

This output seems confusing, but from the second line we conclude that we can define three diagonals, lower, main and upper diagonal, denoted as above. We also know that that the lower and upper diagonals are of size $n-1$. Let us try it out:

```
In [16]: T1 = Tridiagonal(rand(6),rand(7),rand(6))
```

```
Out[16]: 7x7 Tridiagonal{Float64}:
          0.854281  0.112452  0.0       0.0       0.0       0.0        0.0
          0.438987  0.441594  0.310907  0.0       0.0       0.0        0.0
          0.0       0.858792  0.114737  0.506309  0.0       0.0        0.0
          0.0       0.0       0.472064  0.068842  0.207371  0.0        0.0
          0.0       0.0       0.0       0.51933   0.673037  0.0566967  0.0
          0.0       0.0       0.0       0.0       0.222867  0.839389   0.392377
          0.0       0.0       0.0       0.0       0.0       0.530459   0.254929
```

```
In [17]: T2 = Tridiagonal(rand(-5:5,6),randn(7),rand(-9:0,6))
```

```
Out[17]: 7x7 Tridiagonal{Float64}:
          -0.950533  0.0        0.0        0.0         0.0        0.0       0.0
           4.0       0.405133   0.0        0.0         0.0        0.0       0.0
           0.0       4.0        0.176014  -5.0         0.0        0.0       0.0
           0.0       0.0       -2.0       -0.0738621  -8.0        0.0       0.0
           0.0       0.0        0.0       -4.0         0.246255  -2.0       0.0
           0.0       0.0        0.0        0.0         3.0       -1.06209  -3.0
           0.0       0.0        0.0        0.0         0.0       -2.0       0.330312
```

```
In [18]: T3 = T1 + T2
```

```
Out[18]: 7x7 Tridiagonal{Float64}:
          -0.0962519  0.112452   0.0        ...   0.0         0.0         0.0
           4.43899    0.846727   0.310907        0.0         0.0         0.0
           0.0        4.85879    0.290752        0.0         0.0         0.0
           0.0        0.0       -1.52794       -7.79263     0.0         0.0
           0.0        0.0        0.0            0.919292   -1.9433      0.0
           0.0        0.0        0.0        ...   3.22287    -0.222705   -2.60762
           0.0        0.0        0.0            0.0        -1.46954     0.585241
```

This worked as expected, the result is again a `Tridiagonal`. We can access each diagonal by:

```
In [19]: println(T3.dl, T3.d, T3.du)

[4.438987365751102,4.858791645998533,-1.527935612238707,-3.480669544949091,3.22286677301413
```

4.5.4 `@which`

Let us take a closer look at what happens. The **`@which`** command gives the link to the part of the code which is actually invoked. The argument should be only function, without assignment, that is

```
@which T1=Tridiagonal(rand(6),rand(7),rand(6))
```

throws an error.

```
In [20]: @which Tridiagonal(rand(6),rand(7),rand(6))

Out[20]: call{T}(::Type{Tridiagonal{T}}, dl::Array{T,1}, d::Array{T,1}, du::Array{T,1}) at l
```

In the code, we see that there is a type definition in the `immutable` block:

```
## Tridiagonal matrices ##
immutable Tridiagonal{T} <: AbstractMatrix{T}
dl::Vector{T} # sub-diagonal
d::Vector{T} # diagonal
du::Vector{T} # sup-diagonal
du2::Vector{T} # supsup-diagonal for pivoting
end
```

The `Tridiagonal` type consists of **four** vectors. In our case, we actually called the function `Tridiagonal()` with **three** vector arguments. The function creates the type of the same name, setting the fourth reqired vector `du2` to `zeros(T,n-2)`.
The next function with the same name is invoked when the input vectors have different types, in which case the types arer promoted to a most general one, if possible.

```
In [21]: T4 = Tridiagonal([1,2,3], [2.0,3.0,pi,4.0],rand(3)+im*rand(3))

Out[21]: 4x4 Tridiagonal{Complex{Float64}}:
          2.0+0.0im  0.301008+0.141385im       0.0+0.0im               0.0+0.0im
          1.0+0.0im       3.0+0.0im       0.964517+0.917012im          0.0+0.0im
          0.0+0.0im       2.0+0.0im        3.14159+0.0im          0.587013+0.473182im
          0.0+0.0im       0.0+0.0im            3.0+0.0im               4.0+0.0im
```

4.5.5 `size()` and `full()`

For each matrix type we need to define the function which returns the size of a matrix, and the function which converts the matrix of a given type to a full matrix. These function are listed after the second `Tridiagonal()` function.

```
In [22]: size(T4)

Out[22]: (4,4)

In [23]: T4 = full(T4)

Out[23]: 4x4 Array{Complex{Float64},2}:
          2.0+0.0im  0.301008+0.141385im       0.0+0.0im             0.0+0.0im
          1.0+0.0im       3.0+0.0im       0.964517+0.917012im        0.0+0.0im
          0.0+0.0im       2.0+0.0im        3.14159+0.0im        0.587013+0.473182im
          0.0+0.0im       0.0+0.0im            3.0+0.0im             4.0+0.0im
```

4.5.6 `sizeof()`

Of course, using special types can leasd to much more efficient programs. For example, for `Tridiagonal` type, onlt four diagonals are stored, in comparison to storing full matrix when n^2 elements are stored. The storage used is obtained by the `sizeof()` function.

```
In [24]: T1

Out[24]: 7x7 Tridiagonal{Float64}:
          0.854281  0.112452  0.0       0.0       0.0       0.0        0.0
          0.438987  0.441594  0.310907  0.0       0.0       0.0        0.0
          0.0       0.858792  0.114737  0.506309  0.0       0.0        0.0
          0.0       0.0       0.472064  0.068842  0.207371  0.0        0.0
          0.0       0.0       0.0       0.51933   0.673037  0.0566967  0.0
          0.0       0.0       0.0       0.0       0.222867  0.839389   0.392377
          0.0       0.0       0.0       0.0       0.0       0.530459   0.254929

In [25]: T1f=full(T1)

Out[25]: 7x7 Array{Float64,2}:
          0.854281  0.112452  0.0       0.0       0.0       0.0        0.0
          0.438987  0.441594  0.310907  0.0       0.0       0.0        0.0
          0.0       0.858792  0.114737  0.506309  0.0       0.0        0.0
          0.0       0.0       0.472064  0.068842  0.207371  0.0        0.0
          0.0       0.0       0.0       0.51933   0.673037  0.0566967  0.0
          0.0       0.0       0.0       0.0       0.222867  0.839389   0.392377
          0.0       0.0       0.0       0.0       0.0       0.530459   0.254929

In [26]: sizeof(T1f)  #   392 =   7 * 7 * 8 bytes

Out[26]: 392

In [27]: sizeof(T1) # This is not yet implemented for Tridiagonal - only the storage requir

Out[27]: 32
```

4.5.7 immutable

The `immutable` command means that we can change individual elements of defined parts, but not the parts as a whole (an alternative is to use the `type` construtor). For example:

```
In [28]: @show T5 = Tridiagonal([1,2,3],[2,3,4,5],[-1,1,2])
         T5.d[2]=123
         @show T5
         T5.dl = [-1, -1 ,1]
```

```
T5 = Tridiagonal([1,2,3],[2,3,4,5],[-1,1,2]) = [2 -1 0 0
 1 3 1 0
 0 2 4 2
 0 0 3 5]
T5 = [2 -1 0 0
 1 123 1 0
 0 2 4 2
 0 0 3 5]
```

```
        LoadError: type Tridiagonal is immutable
    while loading In[28], in expression starting on line 4
```

4.5.8 methodswith()

This is the reverse of `methods()` - which methods exist for the given type. For example, what can we do with `Tridiagonal` matrices, or with `Dates.Day`:

```
In [29]: methodswith(Tridiagonal)
```

```
Out[29]: 77-element Array{Method,1}:
          *(A::Tridiagonal{T}, B::Number) at linalg/tridiag.jl:406
          *(A::Tridiagonal{T}, B::UpperTriangular{T,S<:AbstractArray{T,2}}) at linalg/triang
          *(A::Tridiagonal{T}, B::Base.LinAlg.UnitUpperTriangular{T,S<:AbstractArray{T,2}})
          *(A::Tridiagonal{T}, B::LowerTriangular{T,S<:AbstractArray{T,2}}) at linalg/triang
          *(A::Tridiagonal{T}, B::Base.LinAlg.UnitLowerTriangular{T,S<:AbstractArray{T,2}})
          *(B::Number, A::Tridiagonal{T}) at linalg/tridiag.jl:407
          +(A::Array{T,2}, B::Tridiagonal{T}) at linalg/special.jl:122
          +(A::Tridiagonal{T}, B::Tridiagonal{T}) at linalg/tridiag.jl:404
          +(A::Diagonal{T}, B::Tridiagonal{T}) at linalg/special.jl:121
          +(A::Tridiagonal{T}, B::Diagonal{T}) at linalg/special.jl:122
          +(A::Bidiagonal{T}, B::Tridiagonal{T}) at linalg/special.jl:121
          +(A::Tridiagonal{T}, B::Bidiagonal{T}) at linalg/special.jl:122
          +(A::Tridiagonal{T}, B::Array{T,2}) at linalg/special.jl:121
          +(A::SymTridiagonal{T}, B::Tridiagonal{T}) at linalg/special.jl:130
          +(A::Tridiagonal{T}, B::SymTridiagonal{T}) at linalg/special.jl:131
          +(A::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}, B::Tridiagonal{T}) a
          +(A::Tridiagonal{T}, B::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}) a
```

```
-(A::Array{T,2}, B::Tridiagonal{T}) at linalg/special.jl:122
-(A::Tridiagonal{T}, B::Tridiagonal{T}) at linalg/tridiag.jl:405
-(A::Diagonal{T}, B::Tridiagonal{T}) at linalg/special.jl:121
-(A::Tridiagonal{T}, B::Diagonal{T}) at linalg/special.jl:122
-(A::Bidiagonal{T}, B::Tridiagonal{T}) at linalg/special.jl:121
-(A::Tridiagonal{T}, B::Bidiagonal{T}) at linalg/special.jl:122
-(A::Tridiagonal{T}, B::Array{T,2}) at linalg/special.jl:121
-(A::SymTridiagonal{T}, B::Tridiagonal{T}) at linalg/special.jl:130
-(A::Tridiagonal{T}, B::SymTridiagonal{T}) at linalg/special.jl:131
-(A::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}, B::Tridiagonal{T}) a
-(A::Tridiagonal{T}, B::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}) a
/(A::Tridiagonal{T}, B::Number) at linalg/tridiag.jl:408
==(A::Tridiagonal{T}, B::Tridiagonal{T}) at linalg/tridiag.jl:410
==(A::Tridiagonal{T}, B::SymTridiagonal{T}) at linalg/tridiag.jl:411
==(A::SymTridiagonal{T}, B::Tridiagonal{T}) at linalg/tridiag.jl:412
A_mul_B!(C::Union{AbstractArray{T,1},AbstractArray{T,2}}, A::Tridiagonal{T}, B::Un
A_mul_B!(A::Tridiagonal{T}, B::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T
abs(M::Tridiagonal{T}) at linalg/tridiag.jl:320
ceil(M::Tridiagonal{T}) at linalg/tridiag.jl:320
ceil{T<:Integer}(::Type{T<:Integer}, M::Tridiagonal{T}) at linalg/tridiag.jl:325
conj(M::Tridiagonal{T}) at linalg/tridiag.jl:320
convert{T}(::Type{AbstractArray{T,2}}, M::Tridiagonal{T}) at linalg/tridiag.jl:418
convert{T}(::Type{Array{T,2}}, M::Tridiagonal{T}) at linalg/tridiag.jl:296
convert{T}(::Type{Array{T,2}}, M::Tridiagonal{T}) at linalg/tridiag.jl:306
convert{T}(::Type{Tridiagonal{T}}, M::Tridiagonal{T}) at linalg/tridiag.jl:417
convert{T}(::Type{SymTridiagonal{T}}, M::Tridiagonal{T}) at linalg/tridiag.jl:421
convert(::Type{Diagonal{T}}, A::Tridiagonal{T}) at linalg/special.jl:51
convert(::Type{Bidiagonal{T}}, A::Tridiagonal{T}) at linalg/special.jl:58
convert(::Type{SymTridiagonal{T}}, A::Tridiagonal{T}) at linalg/special.jl:65
copy(M::Tridiagonal{T}) at linalg/tridiag.jl:320
copy!(dest::Tridiagonal{T}, src::Tridiagonal{T}) at linalg/tridiag.jl:315
ctranspose(M::Tridiagonal{T}) at linalg/tridiag.jl:330
det(A::Tridiagonal{T}) at linalg/tridiag.jl:415
diag{T}(M::Tridiagonal{T}) at linalg/tridiag.jl:333
diag{T}(M::Tridiagonal{T}, n::Integer) at linalg/tridiag.jl:333
factorize(A::Tridiagonal{T}) at linalg/lu.jl:286
floor(M::Tridiagonal{T}) at linalg/tridiag.jl:320
floor{T<:Integer}(::Type{T<:Integer}, M::Tridiagonal{T}) at linalg/tridiag.jl:325
full{T}(M::Tridiagonal{T}) at linalg/tridiag.jl:294
getindex{T}(A::Tridiagonal{T}, i::Integer, j::Integer) at linalg/tridiag.jl:347
imag(M::Tridiagonal{T}) at linalg/tridiag.jl:320
inv(A::Tridiagonal{T}) at linalg/tridiag.jl:414
istril(M::Tridiagonal{T}) at linalg/tridiag.jl:364
istriu(M::Tridiagonal{T}) at linalg/tridiag.jl:363
lufact!{T}(A::Tridiagonal{T}) at linalg/lu.jl:222
lufact!{T}(A::Tridiagonal{T}, pivot::Union{Type{Val{false}},Type{Val{true}}}) at l
real(M::Tridiagonal{T}) at linalg/tridiag.jl:320
round(M::Tridiagonal{T}) at linalg/tridiag.jl:320
round{T<:Integer}(::Type{T<:Integer}, M::Tridiagonal{T}) at linalg/tridiag.jl:325
similar(M::Tridiagonal{T}, T, dims::Tuple{Vararg{Int64}}) at linalg/tridiag.jl:308
size(M::Tridiagonal{T}) at linalg/tridiag.jl:283
```

```
size(M::Tridiagonal{T}, d::Integer) at linalg/tridiag.jl:285
sparse(T::Tridiagonal{T}) at sparse/sparsematrix.jl:396
transpose(M::Tridiagonal{T}) at linalg/tridiag.jl:329
tril!(M::Tridiagonal{T}) at linalg/tridiag.jl:367
tril!(M::Tridiagonal{T}, k::Integer) at linalg/tridiag.jl:367
triu!(M::Tridiagonal{T}) at linalg/tridiag.jl:384
triu!(M::Tridiagonal{T}, k::Integer) at linalg/tridiag.jl:384
trunc(M::Tridiagonal{T}) at linalg/tridiag.jl:320
trunc{T<:Integer}(::Type{T<:Integer}, M::Tridiagonal{T}) at linalg/tridiag.jl:325
```

In [30]:
```
methodswith(Dates.Day)
```

Out[30]:
```
13-element Array{Method,1}:
 +(x::Date, y::Base.Dates.Day) at dates/arithmetic.jl:62
 -(x::Date, y::Base.Dates.Day) at dates/arithmetic.jl:63
 call(::Type{DateTime}, y::Base.Dates.Year, m::Base.Dates.Month, d::Base.Dates.Day)
 call(::Type{DateTime}, y::Base.Dates.Year, m::Base.Dates.Month, d::Base.Dates.Day,
 call(::Type{DateTime}, y::Base.Dates.Year, m::Base.Dates.Month, d::Base.Dates.Day,
 call(::Type{DateTime}, y::Base.Dates.Year, m::Base.Dates.Month, d::Base.Dates.Day,
 call(::Type{DateTime}, y::Base.Dates.Year, m::Base.Dates.Month, d::Base.Dates.Day,
 call(::Type{Date}, y::Base.Dates.Year, m::Base.Dates.Month, d::Base.Dates.Day) at
 convert(::Type{Base.Dates.Week}, x::Base.Dates.Day) at dates/periods.jl:277
 convert(::Type{Base.Dates.Hour}, x::Base.Dates.Day) at dates/periods.jl:270
 convert(::Type{Base.Dates.Minute}, x::Base.Dates.Day) at dates/periods.jl:270
 convert(::Type{Base.Dates.Second}, x::Base.Dates.Day) at dates/periods.jl:270
 convert(::Type{Base.Dates.Millisecond}, x::Base.Dates.Day) at dates/periods.jl:27
```

4.5.9 The "*" operator

In [31]:
```
methods(*)
```

Out[31]:
```
# 138 methods for generic function "*":
*(x::Bool, y::Bool) at bool.jl:38
*{T<:Unsigned}(x::Bool, y::T<:Unsigned) at bool.jl:53
*(x::Bool, z::Complex{Bool}) at complex.jl:122
*(x::Bool, z::Complex{T<:Real}) at complex.jl:129
*{T<:Number}(x::Bool, y::T<:Number) at bool.jl:49
*(x::Float32, y::Float32) at float.jl:211
*(x::Float64, y::Float64) at float.jl:212
*(z::Complex{T<:Real}, w::Complex{T<:Real}) at complex.jl:113
*(z::Complex{Bool}, x::Bool) at complex.jl:123
*(z::Complex{T<:Real}, x::Bool) at complex.jl:130
*(x::Real, z::Complex{Bool}) at complex.jl:140
*(z::Complex{Bool}, x::Real) at complex.jl:141
*(x::Real, z::Complex{T<:Real}) at complex.jl:152
*(z::Complex{T<:Real}, x::Real) at complex.jl:153
*(x::Rational{T<:Integer}, y::Rational{T<:Integer}) at rational.jl:186
*(a::Float16, b::Float16) at float16.jl:136
*{N}(a::Integer, index::CartesianIndex{N}) at multidimensional.jl:50
*(x::BigInt, y::BigInt) at gmp.jl:256
*(a::BigInt, b::BigInt, c::BigInt) at gmp.jl:279
*(a::BigInt, b::BigInt, c::BigInt, d::BigInt) at gmp.jl:285
```

```
*(a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt) at gmp.jl:292
*(x::BigInt, c::Union{UInt16,UInt32,UInt64,UInt8}) at gmp.jl:326
*(c::Union{UInt16,UInt32,UInt64,UInt8}, x::BigInt) at gmp.jl:330
*(x::BigInt, c::Union{Int16,Int32,Int64,Int8}) at gmp.jl:332
*(c::Union{Int16,Int32,Int64,Int8}, x::BigInt) at gmp.jl:336
*(x::BigFloat, y::BigFloat) at mpfr.jl:208
*(x::BigFloat, c::Union{UInt16,UInt32,UInt64,UInt8}) at mpfr.jl:215
*(c::Union{UInt16,UInt32,UInt64,UInt8}, x::BigFloat) at mpfr.jl:219
*(x::BigFloat, c::Union{Int16,Int32,Int64,Int8}) at mpfr.jl:223
*(c::Union{Int16,Int32,Int64,Int8}, x::BigFloat) at mpfr.jl:227
*(x::BigFloat, c::Union{Float16,Float32,Float64}) at mpfr.jl:231
*(c::Union{Float16,Float32,Float64}, x::BigFloat) at mpfr.jl:235
*(x::BigFloat, c::BigInt) at mpfr.jl:239
*(c::BigInt, x::BigFloat) at mpfr.jl:243
*(a::BigFloat, b::BigFloat, c::BigFloat) at mpfr.jl:379
*(a::BigFloat, b::BigFloat, c::BigFloat, d::BigFloat) at mpfr.jl:385
*(a::BigFloat, b::BigFloat, c::BigFloat, d::BigFloat, e::BigFloat) at mpfr.jl:392
*{T<:Number}(x::T<:Number, D::Diagonal{T}) at linalg/diagonal.jl:89
*(x::Irrational{sym}, y::Irrational{sym}) at irrationals.jl:72
*(y::Real, x::Base.Dates.Period) at dates/periods.jl:55
*(x::Number) at operators.jl:74
*(y::Number, x::Bool) at bool.jl:55
*(x::Int8, y::Int8) at int.jl:19
*(x::UInt8, y::UInt8) at int.jl:19
*(x::Int16, y::Int16) at int.jl:19
*(x::UInt16, y::UInt16) at int.jl:19
*(x::Int32, y::Int32) at int.jl:19
*(x::UInt32, y::UInt32) at int.jl:19
*(x::Int64, y::Int64) at int.jl:19
*(x::UInt64, y::UInt64) at int.jl:19
*(x::Int128, y::Int128) at int.jl:456
*(x::UInt128, y::UInt128) at int.jl:457
*{T<:Number}(x::T<:Number, y::T<:Number) at promotion.jl:212
*(x::Number, y::Number) at promotion.jl:168
*{T<:Union{Complex{Float32},Complex{Float64},Float32,Float64},S}(A::Union{DenseArra
*(A::SymTridiagonal{T}, B::Number) at linalg/tridiag.jl:86
*(A::Tridiagonal{T}, B::Number) at linalg/tridiag.jl:406
*(A::UpperTriangular{T,S<:AbstractArray{T,2}}, x::Number) at linalg/triangular.jl:4
*(A::Base.LinAlg.UnitUpperTriangular{T,S<:AbstractArray{T,2}}, x::Number) at linalg
*(A::LowerTriangular{T,S<:AbstractArray{T,2}}, x::Number) at linalg/triangular.jl:4
*(A::Base.LinAlg.UnitLowerTriangular{T,S<:AbstractArray{T,2}}, x::Number) at linalg
*(A::Tridiagonal{T}, B::UpperTriangular{T,S<:AbstractArray{T,2}}) at linalg/triangu
*(A::Tridiagonal{T}, B::Base.LinAlg.UnitUpperTriangular{T,S<:AbstractArray{T,2}}) a
*(A::Tridiagonal{T}, B::LowerTriangular{T,S<:AbstractArray{T,2}}) at linalg/triangu
*(A::Tridiagonal{T}, B::Base.LinAlg.UnitLowerTriangular{T,S<:AbstractArray{T,2}}) a
*(A::Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}}, B::Base.LinAlg.Abstra
*{TA,TB}(A::Base.LinAlg.AbstractTriangular{TA,S<:AbstractArray{T,2}}, B::Union{Dens
*{TA,TB}(A::Union{DenseArray{TA,1},DenseArray{TA,2},SubArray{TA,1,A<:DenseArray{T,N
*{TA,Tb}(A::Union{Base.LinAlg.QRCompactWYQ{TA,M<:AbstractArray{T,2}},Base.LinAlg.QR
*{TA,TB}(A::Union{Base.LinAlg.QRCompactWYQ{TA,M<:AbstractArray{T,2}},Base.LinAlg.QR
*{TA,TQ,N}(A::Union{DenseArray{TA,N},SubArray{TA,N,A<:DenseArray{T,N},I<:Tuple{Vara
```

```
*(A::Union{Hermitian{T,S},Symmetric{T,S}}, B::Union{Hermitian{T,S},Symmetric{T,S}})
*(A::Union{DenseArray{T,2},SubArray{T,2,A<:DenseArray{T,N},I<:Tuple{Vararg{Union{Co
*{T<:Number}(D::Diagonal{T}, x::T<:Number) at linalg/diagonal.jl:90
*(Da::Diagonal{T}, Db::Diagonal{T}) at linalg/diagonal.jl:92
*(D::Diagonal{T}, V::Array{T,1}) at linalg/diagonal.jl:93
*(A::Array{T,2}, D::Diagonal{T}) at linalg/diagonal.jl:94
*(D::Diagonal{T}, A::Array{T,2}) at linalg/diagonal.jl:95
*(A::Bidiagonal{T}, B::Number) at linalg/bidiag.jl:192
*(A::Union{Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}},Bidiagonal{T},Di
*{T}(A::Bidiagonal{T}, B::AbstractArray{T,1}) at linalg/bidiag.jl:202
*(B::BitArray{2}, J::UniformScaling{T<:Number}) at linalg/uniformscaling.jl:122
*{T,S}(s::Base.LinAlg.SVDOperator{T,S}, v::Array{T,1}) at linalg/arnoldi.jl:261
*(S::SparseMatrixCSC{Tv,Ti<:Integer}, J::UniformScaling{T<:Number}) at sparse/linal
*{Tv,Ti}(A::SparseMatrixCSC{Tv,Ti}, B::SparseMatrixCSC{Tv,Ti}) at sparse/linalg.jl:
*{TvA,TiA,TvB,TiB}(A::SparseMatrixCSC{TvA,TiA}, B::SparseMatrixCSC{TvB,TiB}) at spa
*{TX,TvA,TiA}(X::Union{DenseArray{TX,2},SubArray{TX,2,A<:DenseArray{T,N},I<:Tuple{V
*(A::Base.SparseMatrix.CHOLMOD.Sparse{Tv<:Union{Complex{Float64},Float64}}, B::Base
*(A::Base.SparseMatrix.CHOLMOD.Sparse{Tv<:Union{Complex{Float64},Float64}}, B::Base
*(A::Base.SparseMatrix.CHOLMOD.Sparse{Tv<:Union{Complex{Float64},Float64}}, B::Unic
*{Ti}(A::Symmetric{Float64,SparseMatrixCSC{Float64,Ti}}, B::SparseMatrixCSC{Float64
*{Ti}(A::Hermitian{Complex{Float64},SparseMatrixCSC{Complex{Float64},Ti}}, B::Spars
*{T<:Number}(x::AbstractArray{T<:Number,2}) at abstractarraymath.jl:50
*(B::Number, A::SymTridiagonal{T}) at linalg/tridiag.jl:87
*(B::Number, A::Tridiagonal{T}) at linalg/tridiag.jl:407
*(x::Number, A::UpperTriangular{T,S<:AbstractArray{T,2}}) at linalg/triangular.jl:4
*(x::Number, A::Base.LinAlg.UnitUpperTriangular{T,S<:AbstractArray{T,2}}) at linalg
*(x::Number, A::LowerTriangular{T,S<:AbstractArray{T,2}}) at linalg/triangular.jl:4
*(x::Number, A::Base.LinAlg.UnitLowerTriangular{T,S<:AbstractArray{T,2}}) at linalg
*(B::Number, A::Bidiagonal{T}) at linalg/bidiag.jl:193
*(A::Number, B::AbstractArray{T,N}) at abstractarraymath.jl:54
*(A::AbstractArray{T,N}, B::Number) at abstractarraymath.jl:55
*(s1::AbstractString, ss::AbstractString...) at strings/basic.jl:50
*(this::Base.Grisu.Float, other::Base.Grisu.Float) at grisu/float.jl:138
*(index::CartesianIndex{N}, a::Integer) at multidimensional.jl:54
*{T,S}(A::AbstractArray{T,2}, B::Union{DenseArray{S,2},SubArray{S,2,A<:DenseArray{T
*{T,S}(A::AbstractArray{T,2}, x::AbstractArray{S,1}) at linalg/matmul.jl:86
*(A::AbstractArray{T,1}, B::AbstractArray{T,2}) at linalg/matmul.jl:89
*(J1::UniformScaling{T<:Number}, J2::UniformScaling{T<:Number}) at linalg/uniformsc
*(J::UniformScaling{T<:Number}, B::BitArray{2}) at linalg/uniformscaling.jl:123
*(A::AbstractArray{T,2}, J::UniformScaling{T<:Number}) at linalg/uniformscaling.jl:
*{Tv,Ti}(J::UniformScaling{T<:Number}, S::SparseMatrixCSC{Tv,Ti}) at sparse/linalg.
*(J::UniformScaling{T<:Number}, A::Union{AbstractArray{T,1},AbstractArray{T,2}}) at
*(x::Number, J::UniformScaling{T<:Number}) at linalg/uniformscaling.jl:127
*(J::UniformScaling{T<:Number}, x::Number) at linalg/uniformscaling.jl:128
*{T,S}(R::Base.LinAlg.AbstractRotation{T}, A::Union{AbstractArray{S,1},AbstractArra
*{T}(G1::Base.LinAlg.Givens{T}, G2::Base.LinAlg.Givens{T}) at linalg/givens.jl:307
*(p::Base.DFT.ScaledPlan{T,P,N}, x::AbstractArray{T,N}) at dft.jl:262
*{T,K,N}(p::Base.DFT.FFTW.cFFTWPlan{T,K,false,N}, x::Union{DenseArray{T,N},SubArray
*{T,K}(p::Base.DFT.FFTW.cFFTWPlan{T,K,true,N}, x::Union{DenseArray{T,N},SubArray{T,
*{N}(p::Base.DFT.FFTW.rFFTWPlan{Float32,-1,false,N}, x::Union{DenseArray{Float32,N}
*{N}(p::Base.DFT.FFTW.rFFTWPlan{Complex{Float32},1,false,N}, x::Union{DenseArray{Co
```

```
*{N}(p::Base.DFT.FFTW.rFFTWPlan{Float64,-1,false,N}, x::Union{DenseArray{Float64,N}
*{N}(p::Base.DFT.FFTW.rFFTWPlan{Complex{Float64},1,false,N}, x::Union{DenseArray{Co
*{T,K,N}(p::Base.DFT.FFTW.r2rFFTWPlan{T,K,false,N}, x::Union{DenseArray{T,N},SubArr
*{T,K}(p::Base.DFT.FFTW.r2rFFTWPlan{T,K,true,N}, x::Union{DenseArray{T,N},SubArray{
*{T}(p::Base.DFT.FFTW.DCTPlan{T,5,false}, x::Union{DenseArray{T,N},SubArray{T,N,A<:
*{T}(p::Base.DFT.FFTW.DCTPlan{T,4,false}, x::Union{DenseArray{T,N},SubArray{T,N,A<:
*{T,K}(p::Base.DFT.FFTW.DCTPlan{T,K,true}, x::Union{DenseArray{T,N},SubArray{T,N,A<
*{T}(p::Base.DFT.Plan{T}, x::AbstractArray{T,N}) at dft.jl:221
*(α::Number, p::Base.DFT.Plan{T}) at dft.jl:264
*(p::Base.DFT.Plan{T}, α::Number) at dft.jl:265
*(I::UniformScaling{T<:Number}, p::Base.DFT.ScaledPlan{T,P,N}) at dft.jl:266
*(p::Base.DFT.ScaledPlan{T,P,N}, I::UniformScaling{T<:Number}) at dft.jl:267
*(I::UniformScaling{T<:Number}, p::Base.DFT.Plan{T}) at dft.jl:268
*(p::Base.DFT.Plan{T}, I::UniformScaling{T<:Number}) at dft.jl:269
*{P<:Base.Dates.Period}(x::P<:Base.Dates.Period, y::Real) at dates/periods.jl:54
*(a, b, c, xs...) at operators.jl:103
```

We can multiply various types of numbers and matrices. Notice, however, that there is no multiplication specifically defined for `Tridiagonal` matrices. This would not make much sense, since the product of two tridiagonal matrices is a pentadiagonal matrix, the product of three tridiagonal matrices is septadiagonal matrix, ...

Therefore, two tridiagonal matrices are first converted to full matrices, and then multiplied, as is seen in the source code.

```
In [32]: T1*T2

Out[32]: 7x7 Array{Float64,2}:
          -0.362214  0.0455581   0.0         0.0        0.0       0.0        0.0
           1.3491    1.42253     0.0547241  -1.55454    0.0       0.0        0.0
           3.43517   0.806875   -0.992422   -0.611084  -4.05047   0.0        0.0
           0.0       1.88826    -0.054594   -3.19489   -0.49967  -0.414742   0.0
           0.0       0.0        -1.03866    -2.73051   -3.81881  -1.40629   -0.17009
           0.0       0.0         0.0        -0.891467   2.57305  -2.122     -2.38856
           0.0       0.0         0.0         0.0        1.59138  -1.07325   -1.50717

In [33]: @which T1*T2

Out[33]: *(A::Union{Base.LinAlg.AbstractTriangular{T,S<:AbstractArray{T,2}},Bidiagonal{T},Di

In [34]: T1*(T2*T1) # Currently, T1*T2*T1, which is equal to (T1*T2)*T1 does not work, it w

Out[34]: 7x7 Array{Float64,2}:
          -0.289433  -0.0206136   0.0141643  ...   0.0        0.0        0.0
           1.77699    0.826888   -0.285286        -0.322366   0.0        0.0
           3.28881   -0.109681   -0.151476        -2.85284   -0.229648   0.0
           0.828921   0.786958   -0.927386        -1.09126   -0.37646   -0.162735
           0.0       -0.891993   -1.40815         -3.44985   -1.48717   -0.595158
           0.0        0.0        -0.42083    ...   1.07397   -2.90233   -1.44154
           0.0        0.0         0.0              0.831863  -1.61014   -0.805341
```

4.5.10 The "·" operator

```
In [35]: methods(·)

Out[35]: # 5 methods for generic function "dot":
         dot(x::Number, y::Number) at linalg/generic.jl:291
         dot{T<:Union{Float32,Float64},TI<:Integer}(x::Array{T<:Union{Float32,Float64},1}, r
         dot{T<:Union{Complex{Float32},Complex{Float64}},TI<:Integer}(x::Array{T<:Union{Comp
         dot(x::BitArray{1}, y::BitArray{1}) at linalg/bitarray.jl:5
         dot(x::AbstractArray{T,1}, y::AbstractArray{T,1}) at linalg/generic.jl:292
```

By inspecting the source, we see that the `scalar` or the `dot` product of two vectors (1-dimensional arrays) is computed via `BLAS` function `dot` for real arguments, and the function `dotc` for complex arguments.

```
In [36]: x = rand(1:5,5); y  = rand(-5:0,5); a = x·y
         z = rand(5); b = x·z; c = z·x
         w = rand(5) + im*rand(5); d = x·w; e = z·w; f = w·z
         @show x, y, z, w
         @show a, b, c, d, e, f

(x,y,z,w) = ([3,2,2,3,2],[-4,-2,-3,-1,-4],[0.1744172168075051,0.3584729401627924,0.0206247
(a,b,c,d,e,f) = (-33,4.736809173915439,4.736809173915439,5.153596610669984 + 5.3794720968758


Out[36]: (-33,4.736809173915439,4.736809173915439,5.153596610669984 + 5.379472096875815im,1
```

4.6 whos()

The command `whos()` reveals the content of the specified package or module. It can be invoked either with the package name, or with the package name and a regular expression.

```
In [37]: whos(Dates)

                         Apr       8 bytes  Int64
                       April       8 bytes  Int64
                         Aug       8 bytes  Int64
                      August       8 bytes  Int64
                        Date     112 bytes  DataType
                  DateFormat     136 bytes  DataType
                  DatePeriod      92 bytes  DataType
                    DateTime     112 bytes  DataType
                       Dates     305 KB     Module
                         Day     112 bytes  DataType
                         Dec       8 bytes  Int64
                    December       8 bytes  Int64
                         Feb       8 bytes  Int64
                    February       8 bytes  Int64
                         Fri       8 bytes  Int64
                      Friday       8 bytes  Int64
                        Hour     112 bytes  DataType
               ISODateFormat     265 bytes  Base.Dates.DateFormat
```

```
   ISODateTimeFormat   461 bytes  Base.Dates.DateFormat
                 Jan     8 bytes  Int64
             January     8 bytes  Int64
                 Jul     8 bytes  Int64
                July     8 bytes  Int64
                 Jun     8 bytes  Int64
                June     8 bytes  Int64
                 Mar     8 bytes  Int64
               March     8 bytes  Int64
                 May     8 bytes  Int64
         Millisecond   112 bytes  DataType
              Minute   112 bytes  DataType
                 Mon     8 bytes  Int64
              Monday     8 bytes  Int64
               Month   112 bytes  DataType
                 Nov     8 bytes  Int64
            November     8 bytes  Int64
                 Oct     8 bytes  Int64
             October     8 bytes  Int64
              Period    92 bytes  DataType
       RFC1123Format   462 bytes  Base.Dates.DateFormat
                 Sat     8 bytes  Int64
            Saturday     8 bytes  Int64
              Second   112 bytes  DataType
                 Sep     8 bytes  Int64
           September     8 bytes  Int64
                 Sun     8 bytes  Int64
              Sunday     8 bytes  Int64
                 Thu     8 bytes  Int64
            Thursday     8 bytes  Int64
          TimePeriod    92 bytes  DataType
            TimeType    92 bytes  DataType
            TimeZone    92 bytes  DataType
                 Tue     8 bytes  Int64
             Tuesday     8 bytes  Int64
                 UTC    92 bytes  DataType
                 Wed     8 bytes  Int64
           Wednesday     8 bytes  Int64
                Week   112 bytes  DataType
                Year   112 bytes  DataType
              adjust  2689 bytes  Function
     datetime2julian  4149 bytes  Function
       datetime2rata  4114 bytes  Function
       datetime2unix  4141 bytes  Function
                 day  5002 bytes  Function
             dayabbr  6411 bytes  Function
             dayname  6411 bytes  Function
          dayofmonth  4106 bytes  Function
        dayofquarter  4166 bytes  Function
           dayofweek  5080 bytes  Function
    dayofweekofmonth  4237 bytes  Function
```

```
        dayofyear   4962 bytes  Function
      daysinmonth   5626 bytes  Function
       daysinyear   4501 bytes  Function
daysofweekinmonth   4732 bytes  Function
  firstdayofmonth   4570 bytes  Function
firstdayofquarter   4954 bytes  Function
   firstdayofweek   4572 bytes  Function
   firstdayofyear   4572 bytes  Function
             hour   4587 bytes  Function
       isleapyear   5463 bytes  Function
  julian2datetime   4242 bytes  Function
   lastdayofmonth   4926 bytes  Function
 lastdayofquarter   5266 bytes  Function
    lastdayofweek   4568 bytes  Function
    lastdayofyear   4940 bytes  Function
      millisecond   4121 bytes  Function
           minute   4601 bytes  Function
            month   5002 bytes  Function
        monthabbr   6433 bytes  Function
         monthday   5128 bytes  Function
        monthname   6433 bytes  Function
              now   1977 bytes  Function
    quarterofyear   4232 bytes  Function
    rata2datetime   4163 bytes  Function
            recur   3614 bytes  Function
           second   4601 bytes  Function
            today   1127 bytes  Function
          tofirst   1664 bytes  Function
           tolast   1662 bytes  Function
           tonext   3484 bytes  Function
           toprev   3492 bytes  Function
    unix2datetime   4234 bytes  Function
             week   4934 bytes  Function
             year   4998 bytes  Function
        yearmonth   5053 bytes  Function
     yearmonthday   7824 bytes  Function
```

```
In [38]: whos(LinAlg)
```

```
                /     30 KB     Function
  ARPACKException    144 bytes  DataType
       A_ldiv_B!     44 KB     Function
       A_ldiv_Bc    487 bytes  Function
       A_ldiv_Bt    486 bytes  Function
        A_mul_B!     61 KB     Function
        A_mul_Bc   6943 bytes  Function
       A_mul_Bc!     22 KB     Function
        A_mul_Bt   2442 bytes  Function
       A_mul_Bt!   2979 bytes  Function
       A_rdiv_Bc   2981 bytes  Function
       A_rdiv_Bt   2536 bytes  Function
```

Ac_ldiv_B	5890 bytes	Function
Ac_ldiv_Bc	1920 bytes	Function
Ac_mul_B	8932 bytes	Function
Ac_mul_B!	23 KB	Function
Ac_mul_Bc	1685 bytes	Function
Ac_mul_Bc!	1478 bytes	Function
Ac_rdiv_B	487 bytes	Function
Ac_rdiv_Bc	504 bytes	Function
At_ldiv_B	3723 bytes	Function
At_ldiv_Bt	1924 bytes	Function
At_mul_B	6213 bytes	Function
At_mul_B!	6919 bytes	Function
At_mul_Bt	1747 bytes	Function
At_mul_Bt!	1478 bytes	Function
At_rdiv_B	486 bytes	Function
At_rdiv_Bt	502 bytes	Function
BLAS	217 KB	Module
Bidiagonal	192 bytes	DataType
BunchKaufman	308 bytes	DataType
Cholesky	284 bytes	DataType
CholeskyPivoted	332 bytes	DataType
Diagonal	168 bytes	DataType
Eigen	428 bytes	DataType
Factorization	148 bytes	DataType
GeneralizedEigen	428 bytes	DataType
GeneralizedSVD	356 bytes	DataType
GeneralizedSchur	444 bytes	DataType
Hermitian	284 bytes	DataType
Hessenberg	284 bytes	DataType
I	8 bytes	UniformScaling{Int64}
LAPACK	933 KB	Module
LAPACKException	112 bytes	DataType
LDLt	272 bytes	DataType
LU	296 bytes	DataType
LinAlg	2432 KB	Module
LowerTriangular	272 bytes	DataType
PosDefException	112 bytes	DataType
QR	284 bytes	DataType
QRPivoted	296 bytes	DataType
RankDeficientException	112 bytes	DataType
SVD	272 bytes	DataType
Schur	408 bytes	DataType
SingularException	112 bytes	DataType
SymTridiagonal	180 bytes	DataType
Symmetric	284 bytes	DataType
Tridiagonal	204 bytes	DataType
UniformScaling	168 bytes	DataType
UpperTriangular	272 bytes	DataType
\	32 KB	Function
axpy!	9528 bytes	Function
bkfact	3114 bytes	Function

bkfact!	3912 bytes	Function
chol	2800 bytes	Function
cholfact	10 KB	Function
cholfact!	8896 bytes	Function
cond	10 KB	Function
condskeel	3776 bytes	Function
copy!	84 KB	Function
cross	746 bytes	Function
ctranspose	16 KB	Function
det	11 KB	Function
diag	11 KB	Function
diagind	2124 bytes	Function
diagm	4677 bytes	Function
diff	4890 bytes	Function
dot	9428 bytes	Function
eig	3228 bytes	Function
eigfact	8359 bytes	Function
eigfact!	12 KB	Function
eigmax	2921 bytes	Function
eigmin	2885 bytes	Function
eigs	11 KB	Function
eigvals	10 KB	Function
eigvals!	10 KB	Function
eigvecs	10 KB	Function
expm	3540 bytes	Function
eye	3151 bytes	Function
factorize	9903 bytes	Function
givens	2677 bytes	Function
gradient	4401 bytes	Function
hessfact	1142 bytes	Function
hessfact!	587 bytes	Function
isdiag	921 bytes	Function
ishermitian	6548 bytes	Function
isposdef	3391 bytes	Function
isposdef!	1231 bytes	Function
issym	5789 bytes	Function
istril	6683 bytes	Function
istriu	6587 bytes	Function
kron	13 KB	Function
ldltfact	6498 bytes	Function
ldltfact!	1997 bytes	Function
linreg	1192 bytes	Function
logabsdet	1996 bytes	Function
logdet	5829 bytes	Function
logm	19 KB	Function
lu	1801 bytes	Function
lufact	4642 bytes	Function
lufact!	6816 bytes	Function
lyap	2381 bytes	Function
norm	7438 bytes	Function
nullspace	1966 bytes	Function

```
   ordschur   2546 bytes  Function
  ordschur!   2951 bytes  Function
  peakflops   2766 bytes  Function
       pinv   6546 bytes  Function
         qr   3478 bytes  Function
     qrfact   2894 bytes  Function
    qrfact!   3472 bytes  Function
       rank   2282 bytes  Function
      scale   3011 bytes  Function
     scale!     16 KB     Function
      schur   1302 bytes  Function
  schurfact   2355 bytes  Function
 schurfact!   1195 bytes  Function
      sqrtm     10 KB     Function
        svd   6474 bytes  Function
    svdfact   7512 bytes  Function
   svdfact!   4968 bytes  Function
       svds   5708 bytes  Function
    svdvals   4953 bytes  Function
   svdvals!   3505 bytes  Function
  sylvester   2222 bytes  Function
      trace   3314 bytes  Function
  transpose     16 KB     Function
       tril     10 KB     Function
      tril!     15 KB     Function
       triu     10 KB     Function
      triu!     15 KB     Function
     vecdot     19 KB     Function
    vecnorm   3235 bytes  Function
```

```
In [39]: # Now with a regular expression - we are looking for 'eigenvalue' related stuff.
         whos(Base, Regex("eig"))
```

```
        eig   3228 bytes  Function
    eigfact   8359 bytes  Function
   eigfact!     12 KB     Function
     eigmax   2921 bytes  Function
     eigmin   2885 bytes  Function
       eigs     11 KB     Function
    eigvals     10 KB     Function
   eigvals!     10 KB     Function
    eigvecs     10 KB     Function
```

Funally, let us list all we have in `Julia`'s `Base` module. **It is a long list!** Notice that `Dates` and `LinAlg` are modules themselves.

```
In [40]: # whos(Base)
```

```
In [ ]:
```

5 Working with Packages

Starting Julia loads Julia kernel and `Base` module. The `Base` (core) is kept small and all other functionality is accessible through packages which need to be individually included by the user. Currently there are **900+** registered packages listed at Julia Package Listing.
In this notebook, we demonstrate how to use packages.

5.1 Prerequisites

Read sections Packages and Package Development of the Julia manual (15 min).

5.2 Competences

The reader should be able to install and use registered and unregistered packages and create own packages.

5.3 Credits

Some examples are taken from The Julia Manual.

5.4 `Pkg.status()`

```
In [1]: ?Pkg.status()
```

```
Out[1]:
```

```
status()
```

Prints out a summary of what packages are installed and what version and state they're in.

```
In [2]: Pkg.status() # This is slow due to communication with GitHub
```

```
23 required packages:
 - ApproxFun                     0.1.0
 - Arrowhead                     0.0.1+             master
 - AudioIO                       0.1.1
 - DataFrames                    0.6.10
 - DoubleDouble                  0.1.0+             master
 - FastGaussQuadrature           0.0.3
 - Gadfly                        0.4.2
 - GitHub                        2.0.3
 - IJulia                        1.1.8
 - ImageMagick                   0.1.2
 - ImageView                     0.1.19
 - Images                        0.5.2
 - ImplicitEquations             0.1.0
 - Interact                      0.3.0
```

```
 - ODE                            0.2.1+              master
 - Polynomials                    0.0.5+              ef0d044b
 - PyPlot                         2.1.1
 - Roots                          0.1.25
 - SpecialMatrices                0.1.3+              master
 - SymPy                          0.2.35
 - TestImages                     0.1.0
 - WAV                            0.6.3
 - Winston                        0.11.13
69 additional packages:
 - ArrayViews                     0.6.4
 - BinDeps                        0.3.20
 - BufferedStreams                0.0.2
 - CRlibm                         0.2.1
 - Cairo                          0.2.31
 - Calculus                       0.1.14
 - Codecs                         0.1.5
 - ColorTypes                     0.2.0
 - ColorVectorSpace               0.1.1
 - Colors                         0.6.2
 - Compat                         0.7.8
 - Compose                        0.4.2
 - Conda                          0.1.8
 - Contour                        0.0.8
 - DataArrays                     0.2.20
 - DataStructures                 0.4.2
 - Dates                          0.4.4
 - Distances                      0.3.0
 - Distributions                  0.8.9
 - Docile                         0.5.23
 - DualNumbers                    0.2.1
 - FactCheck                      0.4.2
 - FileIO                         0.0.3
 - FixedPointNumbers              0.1.1
 - ForwardDiff                    0.1.4
 - GZip                           0.2.18
 - Graphics                       0.1.3
 - Grid                           0.4.0
 - Hexagons                       0.0.4
 - HttpCommon                     0.2.4
 - HttpParser                     0.1.1
 - HttpServer                     0.1.5
 - ImmutableArrays                0.0.11
 - IniFile                        0.2.5
 - Iterators                      0.1.9
 - JSON                           0.5.0
 - KernelDensity                  0.1.2
 - LaTeXStrings                   0.1.6
 - Libz                           0.0.2
 - Loess                          0.0.6
 - MPSolve                        0.0.0-              master (unregistered)
```

```
 - MacroTools                    0.2.1
 - MatrixDepot                   0.5.2
 - MbedTLS                       0.2.0
 - Measures                      0.0.2
 - NaNMath                       0.1.1
 - Nettle                        0.2.1
 - Optim                         0.4.4
 - PDMats                        0.3.6
 - Plots                         0.5.1
 - PyCall                        1.2.0
 - Reactive                      0.3.0
 - Reexport                      0.0.3
 - Requests                      0.3.4
 - Requires                      0.2.2
 - SHA                           0.1.2
 - SIUnits                       0.0.6
 - Showoff                       0.0.6
 - SortingAlgorithms             0.0.6
 - StatsBase                     0.7.4
 - StatsFuns                     0.2.0
 - TexExtensions                 0.0.3
 - Tk                            0.3.7
 - URIParser                     0.1.2
 - ValidatedNumerics             0.2.0
 - WoodburyMatrices              0.1.5
 - ZMQ                           0.3.1
 - ZipFile                       0.2.6
 - Zlib                          0.1.12
```

5.5 `Pkg.add()`

This command adds registered package from Julia Package Listing. Adding the package downloads the package source code (and all other required packages) to your `.julia/v0.4/` directory. GitHub repository names of registered Julia packages always end with the extension `.jl`, which is ommited in `Pkg.add()` command. The example below installs the package from the GitHub repository https://github.com/JuliaLang/Graphs.jl.
N.B. There are other registered packages dealing with graphs, please check them out.

```
In [3]: ?Pkg.add
```

```
Out[3]:
```

```
add(pkg, vers...)
```

Add a requirement entry for `pkg` to `Pkg.dir("REQUIRE")` and call `Pkg.resolve()`. If `vers` are given, they must be `VersionNumber` objects and they specify acceptable version intervals for `pkg`.

```
In [4]: Pkg.add("Graphs")
```

```
INFO: Updating cache of Graphs...
INFO: Installing Graphs v0.6.0
```

```
INFO: Package database updated
INFO: METADATA is out-of-date | you may not have the latest version of Graphs
INFO: Use ‘Pkg.update()‘ to get the latest versions of your packages
```

```
In [5]: a=readdir("/Users/Ivan/.julia/v0.4") # This is Julia’s default display
```

```
    LoadError: SystemError: unable to read directory /Users/Ivan/.julia/v0.4: No such f
while loading In[5], in expression starting on line 1



     in readdir at ./file.jl:241
```

```
In [6]: println(a)
```

```
Union{ASCIIString,UTF8String}[".cache",".trash","AMVW","Arrowhead","BinDeps","Cairo","Colors
```

5.6 Contents of a package

We now have directory `/Users/Ivan/.julia/v0.4/Graphs`. Let us examine its content (this can also be done directly from the GitHub repository https://github.com/JuliaLang/Graphs.jl).

5.6.1 Files

Each package has the following three files:

- `REQUIRE`
  - may contain the version of Julia needed for the package to run
  - must contain all other registered packages that the present package is using (these packages are installed automatically, if not present) and
  - may contain the version of those packages.
- `README.md` is the Markdown file, which contains the descritption of the package as displayed on the repository's home page.
- `LICENSE.md` contains the licensing information.

The file `travis.yml`, if present, defines how is the package tested on Travis-CI after every posted change (via `git push` command). Details on using Travis-CI for Julia projects are at https://docs.travis-ci.com/user/languages/julia. Since testing is done on machines other than yours, with operating systems other than yours, and using Julia version which may differ from yours, this is a great way to correct bugs, and also a way to give users examples of how to run your code.

5.6.2 Directories

The `src/` directory contains the actual code of your package.

It must contan the file named as the package itself, `src/Graphs.jl` in this case, which containd the following:

- `module` line starts the description of the main module, which has the same name as the package,
- `using` line(s) lists other registered packages used by the package. These packages are also listed in the `REQUIRE` file.
- `import` line lists the other modules and their components which are modified in this module
- `export` line lists all component which will be accessible directly in the main namespace. The components which are not exported, can still be used but the full name (including mogule name) must be used
- `include()` commands include the source files
- `end` concludes the description of the module.

If Travis-CI is used, the `test/` directory contains the file `runtests.jl` which is exaceuted during the testing, and, eventually, other files that this file is calling.

The `doc/` is optional and is used to store documentation.

The `deps/` directory is optional and is used to store dependencies if the package is using software written in other languages. There are many examples which can be checked out.

5.7 `using` and `import`

Package needs to be added only once, prior to the first use. We are now ready to use the package.

We have two methods to do so, which differ in their treatment of the namespace: * `using` adds all methods, constructors etc. from the package into the main namespace, so they can be called directly, like the function `simple_graph(4)` below. * `import` enables us to use all the methods, constructors, etc. from the package, but they are not included in the namespace, so they must be called together with the package name, `Graphs.simple_graph(4)`.

N.B. `import` can also be used on a particular function(s), as we shall explain later.

```
In [6]: using Graphs
```

```
INFO: Recompiling stale cache file /home/slap/.julia/lib/v0.4/DataStructures.ji for module I
```

```
In [7]: whos(Graphs)
```

```
@graph_implements    363 bytes  Function
               @graph_requires    361 bytes  Function
       AbstractDijkstraVisitor     92 bytes  DataType
 AbstractEdgePropertyInspector    148 bytes  DataType
                 AbstractGraph    188 bytes  DataType
          AbstractGraphVisitor     92 bytes  DataType
            AbstractMASVisitor     92 bytes  DataType
           AbstractPrimVisitor     92 bytes  DataType
                 AdjacencyList     80 bytes  TypeConstructor
                 AttributeDict    200 bytes  DataType
AttributeEdgePropertyInspector    168 bytes  DataType
              BellmanFordStates   232 bytes  DataType
                  BreadthFirst     92 bytes  DataType
 ConstantEdgePropertyInspector    168 bytes  DataType
                    DepthFirst     92 bytes  DataType
                DijkstraStates    348 bytes  DataType
```

Edge	192 bytes	DataType
EdgeList	120 bytes	TypeConstructor
ExEdge	204 bytes	DataType
ExVertex	136 bytes	DataType
GenericAdjacencyList	284 bytes	DataType
GenericEdgeList	312 bytes	DataType
GenericGraph	388 bytes	DataType
GenericIncidenceList	324 bytes	DataType
Graph	120 bytes	TypeConstructor
Graphs	366 KB	Module
IncidenceList	120 bytes	TypeConstructor
KeyVertex	180 bytes	DataType
LogGraphVisitor	168 bytes	DataType
MaximumAdjacency	92 bytes	DataType
NegativeCycleError	92 bytes	DataType
PrimStates	336 bytes	DataType
SimpleAdjacencyList	172 bytes	DataType
SimpleGraph	212 bytes	DataType
SimpleIncidenceList	180 bytes	DataType
TrivialGraphVisitor	92 bytes	DataType
VectorEdgePropertyInspector	168 bytes	DataType
WeightedEdge	220 bytes	DataType
add_edge!	7394 bytes	Function
add_vertex!	5916 bytes	Function
adjacency_matrix	1013 bytes	Function
adjacency_matrix_sparse	1027 bytes	Function
adjlist	3136 bytes	Function
attributes	950 bytes	Function
bellman_ford_shortest_paths	1635 bytes	Function
bellman_ford_shortest_paths!	4672 bytes	Function
close_vertex!	6920 bytes	Function
collect_edges	2002 bytes	Function
collect_weighted_edges	3590 bytes	Function
connected_components	2084 bytes	Function
create_bellman_ford_states	1067 bytes	Function
create_dijkstra_states	1430 bytes	Function
create_prim_states	1293 bytes	Function
dijkstra_shortest_paths	6348 bytes	Function
dijkstra_shortest_paths!	4450 bytes	Function
dijkstra_shortest_paths_withlog	1496 bytes	Function
discover_vertex!	6497 bytes	Function
distance_matrix	1232 bytes	Function
edge_index	2351 bytes	Function
edge_property	1745 bytes	Function
edge_property_requirement	668 bytes	Function
edge_type	578 bytes	Function
edgelist	1887 bytes	Function
edges	894 bytes	Function
enumerate_indices	2949 bytes	Function
enumerate_paths	2347 bytes	Function
erdos_renyi_graph	3449 bytes	Function

examine_edge!	2797 bytes	Function
examine_neighbor!	4629 bytes	Function
floyd_warshall	605 bytes	Function
floyd_warshall!	5496 bytes	Function
gdistances	1646 bytes	Function
gdistances!	2353 bytes	Function
graph	1823 bytes	Function
has_negative_edge_cycle	1792 bytes	Function
implements_adjacency_list	1298 bytes	Function
implements_adjacency_matrix	474 bytes	Function
implements_bidirectional_adjacency_list	862 bytes	Function
implements_bidirectional_incidence_list	862 bytes	Function
implements_edge_list	1250 bytes	Function
implements_edge_map	1638 bytes	Function
implements_incidence_list	910 bytes	Function
implements_vertex_list	2026 bytes	Function
implements_vertex_map	2026 bytes	Function
in_degree	568 bytes	Function
in_edges	596 bytes	Function
in_neighbors	586 bytes	Function
inclist	6460 bytes	Function
is_directed	1726 bytes	Function
kruskal_minimum_spantree	3128 bytes	Function
kruskal_select	2841 bytes	Function
laplacian_matrix	8870 bytes	Function
laplacian_matrix_sparse	13 KB	Function
make_edge	1179 bytes	Function
make_vertex	1044 bytes	Function
maximal_cliques	8446 bytes	Function
maximum_adjacency_visit	5962 bytes	Function
min_cut	3582 bytes	Function
moebius_kantor_graph	813 bytes	Function
num_edges	1716 bytes	Function
num_vertices	1738 bytes	Function
open_vertex!	967 bytes	Function
out_degree	1541 bytes	Function
out_edges	1092 bytes	Function
out_neighbors	1603 bytes	Function
plot	901 bytes	Function
prim_minimum_spantree	2278 bytes	Function
prim_minimum_spantree!	2922 bytes	Function
prim_minimum_spantree_withlog	1140 bytes	Function
revedge	1111 bytes	Function
shortest_path	3780 bytes	Function
simple_adjlist	3080 bytes	Function
simple_bull_graph	547 bytes	Function
simple_chvatal_graph	813 bytes	Function
simple_complete_graph	1587 bytes	Function
simple_cubical_graph	693 bytes	Function
simple_desargues_graph	873 bytes	Function
simple_diamond_graph	547 bytes	Function

simple_dodecahedral_graph	873 bytes	Function
simple_edgelist	1747 bytes	Function
simple_frucht_graph	753 bytes	Function
simple_graph	1575 bytes	Function
simple_heawood_graph	783 bytes	Function
simple_house_graph	633 bytes	Function
simple_house_x_graph	704 bytes	Function
simple_icosahedral_graph	873 bytes	Function
simple_inclist	1594 bytes	Function
simple_krackhardt_kite_graph	753 bytes	Function
simple_octahedral_graph	693 bytes	Function
simple_pappus_graph	843 bytes	Function
simple_path_graph	1583 bytes	Function
simple_petersen_graph	723 bytes	Function
simple_sedgewick_maze_graph	673 bytes	Function
simple_star_graph	1583 bytes	Function
simple_tetrahedral_graph	557 bytes	Function
simple_truncated_cube_graph	933 bytes	Function
simple_truncated_tetrahedron_graph	753 bytes	Function
simple_tutte_graph	1263 bytes	Function
simple_wheel_graph	1585 bytes	Function
source	1826 bytes	Function
sparse2adjacencylist	2404 bytes	Function
strongly_connected_components	2267 bytes	Function
target	1826 bytes	Function
test_cyclic_by_dfs	1839 bytes	Function
to_dot	9040 bytes	Function
topological_sort_by_dfs	1746 bytes	Function
traverse_graph	8316 bytes	Function
traverse_graph_withlog	1232 bytes	Function
vertex_index	3616 bytes	Function
vertex_type	578 bytes	Function
vertices	1714 bytes	Function
visited_vertices	969 bytes	Function
watts_strogatz_graph	2588 bytes	Function
weight_matrix	641 bytes	Function
weight_matrix_sparse	648 bytes	Function

5.7.1 Example

Let us construct the famous graph of the Seven Bridges of Königsberg, plot it, and compute the number of *different* walks which cross 3 bridges between the north side and the center island. Can you enumerate the walks?

For the `plot(g)` to work, GraphViz must be installed.

In Windows, this in not enough, and we must use the package IJuliaPortrayals (with line 263 changed from `gv_process.exitcode == 0` to `gv_process.exitcode != 0` - a bug!).

N.B. `IJuliaPortrayals` can be used to include various media, see the demo of the package.

```
In [8]: g=simple_graph(4,is_directed=false)
```

```
Out[8]: Undirected Graph (4 vertices, 0 edges)
```

```
In [9]: add_edge!(g,1,2)
        add_edge!(g,1,2)
        add_edge!(g,1,3)
        add_edge!(g,1,3)
        add_edge!(g,1,4)
        add_edge!(g,2,4)
        add_edge!(g,3,4)
        g

Out[9]: Undirected Graph (4 vertices, 7 edges)

In [10]: Pkg.add("IJuliaPortrayals")
         using IJuliaPortrayals

INFO: Cloning cache of IJuliaPortrayals from git://github.com/jbn/IJuliaPortrayals.jl.git
INFO: Installing IJuliaPortrayals v0.0.4
INFO: Building Nettle
INFO: Recompiling stale cache file /home/slap/.julia/lib/v0.4/BinDeps.ji for module BinDeps
INFO: Recompiling stale cache file /home/slap/.julia/lib/v0.4/SHA.ji for module SHA.
INFO: Building ZMQ
INFO: Building IJulia
INFO: Recompiling stale cache file /home/slap/.julia/lib/v0.4/Conda.ji for module Conda.
INFO: Found Jupyter version 3.2.0: ipython
Writing IJulia kernelspec to /home/slap/.julia/v0.4/IJulia/deps/julia-0.4/kernel.json ...
Installing julia kernelspec julia-0.4
INFO: Package database updated
INFO: METADATA is out-of-date | you may not have the latest version of IJuliaPortrayals
INFO: Use `Pkg.update()` to get the latest versions of your packages

In [11]: GraphViz(to_dot(g),"neato", "svg")

Out[11]: IJuliaPortrayals.GraphViz("graph graphname {\n1\n2\n3\n4\n1 -- 2\n1 -- 2\n1 -- 3\n1

In [12]: a=adjacency_matrix(g) #This is not what we want

Out[12]: 4x4 Array{Bool,2}:
          false   true   true   true
           true  false  false   true
           true  false  false   true
           true   true   true  false

In [13]: edges(g) # Lets look at the edges

Out[13]: 7-element Array{Graphs.Edge{Int64},1}:
          edge [1]: 1 -- 2
          edge [2]: 1 -- 2
          edge [3]: 1 -- 3
          edge [4]: 1 -- 3
          edge [5]: 1 -- 4
          edge [6]: 2 -- 4
          edge [7]: 3 -- 4
```

```
In [14]: weights=[2,2,2,2,1,1,1]  # We shall emulate our adjacency matrix with the weight m
         a=weight_matrix(g,weights)

Out[14]: 4x4 Array{Int64,2}:
          0  2  2  1
          2  0  0  1
          2  0  0  1
          1  1  1  0

In [15]: no_of_walks=(a^3)[1,2]

Out[15]: 22
```

5.8 Pkg.checkout()

The contents of a registered package obtained by the command `Pkg.add("Package_name")` is fixed at the time of registration.

The package owner may further develop the package, but those changes are not registered (until the registration of a new version).

If you want to use the latest available version, the command `Pkg.cehckout("Package_name")` downloads the latest master.

5.9 Pkg.clone()

Adds unregistered packages or repositories. Here the full GitHub address needs to be supplied. As an example, we shall use the package LinearAlgebra.jl.

N.B. In Julia, the linear algebra routines are incorporated as wrappers of various LAPACK. This package contains several routines written directly in Julia.

By inspecting the file `src/LinearAlgebra.jl`, we see that nothing is exported so all methods need to be fully specified. We also see that the SVD related stuff may be in the file `src/svd.jl`. There we see that the sub-module `SVDModule` is defined, but with nothing eported, and we must specify the full command `LinearAlgebra.SVDModule.svdvals!()`.

We shall compute the singular values of the bidiagonal unity Jordan form with the standard Julia function `svdvals()` and the function from the package.

```
In [16]: Pkg.clone("https://github.com/andreasnoack/LinearAlgebra.jl")

INFO: Cloning LinearAlgebra from https://github.com/andreasnoack/LinearAlgebra.jl
INFO: Computing changes...
INFO: No packages to install, update or remove

In [17]: using LinearAlgebra

In [18]: whos(LinearAlgebra) # Not much of an information

LinearAlgebra     316 KB     Module
                     numnegevals    2383 bytes   Function

In [19]: whos(LinearAlgebra.SVDModule) # Also no information

SVDModule      30 KB     Module
```

```
In [20]: methods(LinearAlgebra.SVDModule.svdvals!)

Out[20]: # 3 methods for generic function "svdvals!":
         svdvals!{T<:Real}(B::Bidiagonal{T<:Real}) at /home/slap/.julia/v0.4/LinearAlgebra/s
         svdvals!{T<:Real}(B::Bidiagonal{T<:Real}, tol) at /home/slap/.julia/v0.4/LinearAlge
         svdvals!(A::Union{DenseArray{T,2},SubArray{T,2,A<:DenseArray{T,N},I<:Tuple{Vararg{U

In [21]: methods(Bidiagonal) # We now know how to define bidiagonal matrix

Out[21]: 7-element Array{Any,1}:
          call{T}(::Type{Bidiagonal{T}}, dv::AbstractArray{T,1}, ev::AbstractArray{T,1}, isu
          call{T}(::Type{Bidiagonal{T}}, dv::AbstractArray{T,1}, ev::AbstractArray{T,1}) at
          call(::Type{Bidiagonal{T}}, dv::AbstractArray{T,1}, ev::AbstractArray{T,1}, uplo::
          call{Td,Te}(::Type{Bidiagonal{T}}, dv::AbstractArray{Td,1}, ev::AbstractArray{Te,1
          call(::Type{Bidiagonal{T}}, A::AbstractArray{T,2}, isupper::Bool) at linalg/bidiag
          call{T}(::Type{T}, arg) at essentials.jl:56
          call{T}(::Type{T}, args...) at essentials.jl:57

In [22]: n=70
         c=0.5
         J=Bidiagonal(c*ones(n),ones(n-1),true)

Out[22]: 70x70 Bidiagonal{Float64}:
          0.5  1.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.5  1.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.5  1.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.5  1.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.5  1.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.5  1.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.5  1.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.5     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          ⋮                        ⋮              ⋱              ⋮
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     1.0  0.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.5  1.0  0.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.5  1.0  0.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.5  1.0  0.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.5  1.0  0.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.5  1.0  0.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.5  1.0
          0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.5

In [24]: @time s=svdvals(J);
```

```
0.000317 seconds (19 allocations: 11.000 KB)
```

Julia uses convention that function names ending in `!` overwrite the input data. Thus, we first make a copy of `J`.

```
In [25]: J1=deepcopy(J);

In [26]: J0=Bidiagonal([1.0,1,1],[1,1],true)

Out[26]: 3x3 Bidiagonal{Float64}:
          1.0  1.0  0.0
          0.0  1.0  1.0
          0.0  0.0  1.0

In [27]: LinearAlgebra.SVDModule.svdvals!(J0)

Out[27]: 3-element Array{Float64,1}:
          1.80194
          1.24698
          0.445042

In [28]: @time s1=LinearAlgebra.SVDModule.svdvals!(J1);

0.000629 seconds (17 allocations: 2.031 KB)

In [29]: typeof(s1), s1

Out[29]: (Array{Float64,1},[1.49967,1.49867,1.49701,1.49468,1.4917,1.48805,1.48375,1.47879,1

In [30]: s-s1[1] # Tiny singular value is inaccurate and it should not be

Out[30]: 70-element Array{Float64,1}:
           4.44089e-16
          -0.000997306
          -0.00265876
          -0.00498325
          -0.00796927
          -0.0116148
          -0.0159175
          -0.0208745
          -0.0264825
          -0.0327378
          -0.0396361
          -0.0471727
          -0.0553428
           ⋮
          -0.889627
          -0.906882
          -0.923074
          -0.938069
          -0.951735
          -0.963934
```

```
-0.974534
-0.983409
-0.990443
-0.995542
-0.998632
-1.49967
```

```
In [32]: s[70], s1[70]
```

```
Out[32]: (6.352747104407252e-22,6.352747104407255e-22)
```

5.10 Pkg.rm()

This command removes (deletes) added or cloned packages and all required packages not in use otherwise.

```
In [23]: Pkg.rm("Graphs")
```

```
INFO: Removing Graphs (unregistered)
```

```
In [2]: Pkg.rm("LinearAlgebra")
```

```
INFO: Removing LinearAlgebra (unregistered)
```

```
In [24]: a=readdir("/Users/Ivan/.julia/v0.4");
         println(a)
```

```
Union{ASCIIString,UTF8String}[".cache",".trash","AMVW","Arrowhead","BinDeps","Cairo","Colors
```

5.11 Creating packages

You need to use GitHub:

1. Go to GitHub and `Sign up` and `Sign in`.
2. Set up Git at your computer.

N.B. Check out GitHub Guides.
One way to start developing packages is

1. Create new repository at GitHub.
2. Clone the created package to your computer with

   ```
   git clone https://github.com/your_user_name/your_repository_name.jl
   ```

3. Start writing your code as described in Contents of the Package.
4. Check what you have changed

   ```
   git commit
   ```

5. Add changes to be commited with

```
git add file1 file2 ...
```

6. Commit the changes (you need to supply the message)

```
git commit
```

7. Push the changes to your GitHub repository

```
git push
```

N.B. There are various other possibilities and shorthands (see the Guides). For example, steps 4., 5. and 6. can be shortened with

```
git commit -am "your message"
```

Also, if you work on your package from two computers, you may need to synchronize your repository: assume that you pushed the changes that you made on computer A to GitHub, and that you want to continue to work on your repository from computer B. Then, you obviously need to synchronize computer B with the latest version from GitHub. This is done with the following commands issued on computer B:

```
git fetch origin
git reset --hard origin/master
git clean -f -d
```

5.12 Be social

You can fork other people's repositories, and use them and change them as your own. You can make pull requests to incorporate those changes to those repositories.

You can easily make different branches of your repository, and test different options.

GitHub enables you to share your work with others, so even small, undocumented packages can be very useful.

```
In [ ]:
```

6 Profiling

Julia has several means for inspection of the program execution:

- viewing execution time and overall memory allocation,
- viewing lower level code,
- tracking function calls, and
- tracking memory allocation.

6.1 Prerequisites

Read sections Profiling and Reflection and introspection of the Julia manual (20 min).

6.2 Competences

The reader should be able to determine frequency of executed commands and memory allocation a function.

6.3 Execution time and overall memory allocation

Let us compute $\alpha * a \cdot b$ for scalar α and (fairly long) vectors a and b:

```
In [1]: a=rand(1000000)
        b=rand(1000000)
        α=rand()

Out[1]: 0.7576316974050816

In [2]: @time α*a⋅b

0.041210 seconds (15.39 k allocations: 8.420 MB)

Out[2]: 189402.18130885967

In [3]: @time α*(a⋅b)

0.003342 seconds (37 allocations: 2.141 KB)

Out[3]: 189402.18130885987
```

We see that the second evaluation is much faster, due to adequate memory allocation - the reason is that there is no operator precedence between * and ·. Also,

Loops in Julia are very fast:

```
In [4]: @time s=0.0; for i=1:1000000; s+=α*a[i]*b[i]; end; s

0.000015 seconds (5 allocations: 208 bytes)

Out[4]: 189402.18130886118
```

6.4 Lower level code

The function `code_llvm()` and `code_native()` return the LLVM intermediate representation of a function and the compiled machine code, respectively. (There are other useful functions described in the Manual.)
They can also be called as macros, which is the form that we shall use.
Observe the differences in the small examples below:

```
In [5]: ?code_llvm

search: code_llvm @code_llvm

Out[5]:

..  code_llvm(f, types)

Prints the LLVM bitcodes generated for running the method matching the given generic functio

All metadata and dbg.* calls are removed from the printed bitcode. Use code_llvm_raw for the

In [6]: ?code_native

search: code_native @code_native

Out[6]:

code_native(f, types)
```

Prints the native assembly instructions generated for running the method matching the given generic function and type signature to `STDOUT`.

```
In [7]: @code_llvm +(1,2)

define i64 @"julia_+_21787"(i64, i64) {
top:
  %2 = add i64 %1, %0
  ret i64 %2
}

In [8]: @code_llvm +(1.0,2)

define double @"julia_+_21793"(double, i64) {
top:
  %2 = sitofp i64 %1 to double
  %3 = fadd double %2, %0
  ret double %3
}

In [9]: @code_native +(1,2)
```

```
.text
Filename: int.jl
Source line: 8
        pushq        %rbp
        movq        %rsp, %rbp
Source line: 8
        addq        %rsi, %rdi
        movq        %rdi, %rax
        popq        %rbp
        ret
```

```
In [10]: @code_native +(1.0,2)
```

```
.text
Filename: promotion.jl
Source line: 167
        pushq        %rbp
        movq        %rsp, %rbp
Source line: 167
        cvtsi2sdq        %rdi, %xmm1
        addsd        %xmm0, %xmm1
        movaps        %xmm1, %xmm0
        popq        %rbp
        ret
```

6.5 Tracking function calls

We shall demonstrate the process on simple problem of polynomial evaluation. We shall use the registered package Polynomials.jl for polynomial manipulations, and two own functions for polynomial evalutaion: * Horner scheme * evaluation with remembering powers.

```
In [11]: # Pkg.add("Polynomials")
         using Polynomials
```

```
In [12]: whos(Polynomials)
```

```
/     31 KB     Function
                   Pade    232 bytes  DataType
                   Poly    180 bytes  DataType
            Polynomials     97 KB     Module
                 coeffs    502 bytes  Function
                 degree    503 bytes  Function
                padeval    622 bytes  Function
                   poly   3898 bytes  Function
                polyder   4720 bytes  Function
                polyfit   3487 bytes  Function
                polyint   4327 bytes  Function
                polyval   2030 bytes  Function
                  roots   4109 bytes  Function
```

```
In [13]: methods(polyval) # polyval() is just Horner's scheme
```

```
Out[13]: # 2 methods for generic function "polyval":
         polyval(p::Polynomials.Poly{T<:Number}, v::AbstractArray{T,1}) at /home/slap/.julia
         polyval{T,S}(p::Polynomials.Poly{T}, x::S) at /home/slap/.julia/v0.4/Polynomials/sr

function polyval{T,S}(p::Poly{T}, x::S)
    R = promote_type(T,S)
    lenp = length(p)
    if lenp == 0
        return zero(R) * x
    else
        y = convert(R, p[end]) + 0*x
        for i = (endof(p)-1):-1:0
            y = p[i] + x*y
        end
        return y
    end
end

In [14]: p=Poly([1.0,2,3,4]) # Polynomial with given coefficients

Out[14]: Poly(1.0 + 2.0x + 3.0x^2 + 4.0x^3)

In [15]: q=poly([1.0,2,3,4]) # Polynomial with given zeros

Out[15]: Poly(24.0 - 50.0x + 35.0x^2 - 10.0x^3 + x^4)

In [16]: p(pi), polyval(p,pi), q(pi), polyval(q,pi)

Out[16]: (160.91710523164693,160.91710523164693,-0.29715441035788004,-0.29715441035788004)

In [17]: function mypolyval(p::Poly,x::Number)
             s=p[0]
             t=one(x)
             for i=1:length(p)-1
                 t*=x
                 s+=p[i]*t
             end
             s
         end

         function myhorner(p::Poly,x::Number)
             s=p[end]
             for i=length(p)-2:-1:0
                 s=s*x+p[i]
             end
             s
         end

Out[17]: myhorner (generic function with 1 method)

In [18]: mypolyval(p,map(Float64,pi)), myhorner(p,map(Float64,pi))
```

```
Out[18]: (160.91710523164693,160.91710523164693)
```

Let us perform some timings:

```
In [19]: n=1000001
         pbig=Poly(rand(n))
         x=0.12345;

In [22]: @time pbig(x)

0.007569 seconds (5 allocations: 176 bytes)

Out[22]: 1.0721728131555393

In [23]: @time polyval(pbig,x)

0.008773 seconds (5 allocations: 176 bytes)

Out[23]: 1.0721728131555393

In [24]: @time myhorner(pbig,x)

0.008803 seconds (5 allocations: 176 bytes)

Out[24]: 1.0721728131555393

In [25]: @time mypolyval(pbig,x) # This is two times faster! Why?

0.003256 seconds (5 allocations: 176 bytes)

Out[25]: 1.0721728131555393

In [26]: @code_native mypolyval(pbig,x)

.text
Filename: In[17]
Source line: 2
        pushq         %rbp
        movq         %rsp, %rbp
Source line: 2
        movq         (%rdi), %rax
        xorps         %xmm1, %xmm1
        cmpq         $0, 8(%rax)
        jle         L28
        movq         (%rax), %rax
        movsd         (%rax), %xmm1
Source line: 4
L28:        movq         (%rdi), %rax
        movq         8(%rax), %rcx
        xorl         %r9d, %r9d
        decq         %rcx
        movl         $0, %eax
```

```
        cmovnsq         %rcx, %rax
        testq           %rax, %rax
        je              L173
Source line: 6
        movq            (%rdi), %rdi
        movabsq         $139736514726144, %rdx  # imm = 0x7F16F1528900
Source line: 4
        testq           %rcx, %rcx
Source line: 6
        cmovnsq         %rcx, %r9
        movsd           (%rdx), %xmm2
        movq            8(%rdi), %r8
        movl            $1, %esi
        movq            $-1, %rdx
        movl            $2, %ecx
L104:           xorps           %xmm3, %xmm3
        cmpq            %rcx, %r8
        jl              L143
        cmpq            %r8, %rsi
        jae             L181
        leaq            (,%rdx,8), %r10
        movq            (%rdi), %rax
        subq            %r10, %rax
        movsd           (%rax), %xmm3
Source line: 4
L143:           incq            %rsi
Source line: 5
        mulsd           %xmm0, %xmm2
Source line: 6
        mulsd           %xmm2, %xmm3
        addsd           %xmm3, %xmm1
        incq            %rcx
        decq            %rdx
        decq            %r9
        jne             L104
Source line: 8
L173:           movaps          %xmm1, %xmm0
        movq            %rbp, %rsp
        popq            %rbp
        ret
Source line: 6
L181:           movq            %rsp, %rax
        leaq            -16(%rax), %rsi
        movq            %rsi, %rsp
        movq            %rcx, -16(%rax)
        movabsq         $jl_bounds_error_ints, %rax
        movl            $1, %edx
        callq           *%rax
```

```
In [27]: @code_native myhorner(pbig,x) # Code is the same for pbig
```

```
.text
Filename: In[17]
Source line: 12
        pushq       %rbp
        movq        %rsp, %rbp
        pushq       %r15
        pushq       %r14
        pushq       %rbx
        subq        $24, %rsp
        movq        %rdi, %r14
Source line: 12
        movq        (%r14), %rdi
        movq        8(%rdi), %rax
        movq        %rax, %rcx
        addq        $-1, %rcx
        jae         L219
        movsd       %xmm0, -40(%rbp)
        movq        (%rdi), %rax
        movsd       (%rax,%rcx,8), %xmm0
Source line: 13
        movsd       %xmm0, -32(%rbp)
        movq        (%r14), %rax
        movq        8(%rax), %r15
        leaq        -2(%r15), %rbx
        movabsq     $steprange_last, %rax
        movq        %rbx, %rdi
        movq        $-1, %rsi
        xorl        %edx, %edx
        callq       *%rax
        cmpq        %rax, %rbx
        jl          L203
        leaq        -1(%r15), %rcx
        cmpq        %rax, %rcx
        movsd       -40(%rbp), %xmm1
        je          L203
        shlq        $3, %r15
        movl        $16, %edx
        subq        %r15, %rdx
Source line: 14
        movq        (%r14), %rdi
        movq        8(%rdi), %r8
L135:   xorps       %xmm0, %xmm0
        cmpq        %rcx, %r8
        jl          L166
        cmpq        %r8, %rbx
        jae         L250
        movq        (%rdi), %rsi
        subq        %rdx, %rsi
        movsd       (%rsi), %xmm0
L166:   movsd       -32(%rbp), %xmm2
        mulsd       %xmm1, %xmm2
```

```
        addsd        %xmm0, %xmm2
        movsd        %xmm2, -32(%rbp)
        addq        $8, %rdx
Source line: 13
        decq        %rbx
Source line: 14
        decq        %rcx
        cmpq        %rcx, %rax
        jne        L135
Source line: 16
L203:        movsd        -32(%rbp), %xmm0
        leaq        -24(%rbp), %rsp
        popq        %rbx
        popq        %r14
        popq        %r15
        popq        %rbp
        ret
Source line: 12
L219:        movq        %rsp, %rcx
        leaq        -16(%rcx), %rsi
        movq        %rsi, %rsp
        movq        %rax, -16(%rcx)
        movabsq        $jl_bounds_error_ints, %rax
        movl        $1, %edx
        callq        *%rax
Source line: 14
L250:        movq        %rsp, %rax
        leaq        -16(%rax), %rsi
        movq        %rsi, %rsp
        movq        %rcx, -16(%rax)
        movabsq        $jl_bounds_error_ints, %rax
        movl        $1, %edx
        callq        *%rax
```

It is difficult to see where the difference in speed comes from. Let us track function calls.

6.5.1 `@profile`

```
In [28]: ?@profile
```

```
Out[28]:
```

`@profile`

`@profile <expression>` runs your expression while taking periodic backtraces. These are appended to an internal buffer of backtraces.

```
In [29]: Profile.clear()
```

```
In [30]: @profile (for i = 1:100; mypolyval(pbig,x); end)
```

```
In [31]: Profile.print()
```

```
223 task.jl; anonymous; line: 447
 223 .../IJulia/src/IJulia.jl; eventloop; line: 142
  223 ...rc/execute_request.jl; execute_request_0x535c5df2; line: 182
   223 loading.jl; include_string; line: 282
    223 In[30]; anonymous; line: 1
     36  In[17]; mypolyval; line: 5
     186 In[17]; mypolyval; line: 6
```

In [32]:
```
Profile.clear()
@profile (for i = 1:100; myhorner(pbig,x); end)
Profile.print()
```

```
453 task.jl; anonymous; line: 447
 453 .../IJulia/src/IJulia.jl; eventloop; line: 142
  453 ...rc/execute_request.jl; execute_request_0x535c5df2; line: 182
   453 loading.jl; include_string; line: 282
    453 In[32]; anonymous; line: 2
     453 In[17]; myhorner; line: 14
```

By inspecting the output, we see that the main load is the execution of computational lines inside the loops. This still does not explain the difference in speed.

The above profiles also includes IJulia calls. It profiling is done in terminal mode, IJulia calla will not be present. This can be done by the following commands:

```
include("myfunctions.jl") # Contains the function definitions
Profile.clear()
@profile (for i = 1:100000; mypolyval(pbig,x); end)
Profile.print()

Profile.clear()
@profile (for i = 1:100000; myhorner(pbig,x); end)
Profile.print()
```

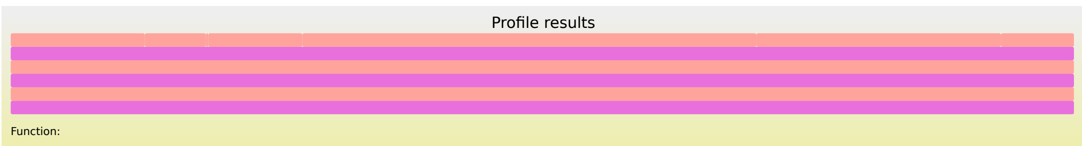
Output can also be viewed using the registered package ProfileView.jl:

In [33]:
```
# Pkg.add("ProfileView")
```

In [35]:
```
using ProfileView
```

In [36]:
```
ProfileView.view()
```

Out[36]:



6.6 Tracking memory allocation

Memory allocation analysis must be performed in terminal mode. The entire code must be stored in a single file, for example `myfile.jl`.
The command

```
julia --track-alocation=user myfile.jl
```

generates the file `myfile.jl.mem` with memory allocation displayed for each line of code.
We see that the memory allocation is as expected, and there is still no explanation for the difference in execution time.

In [ ]:

7 Plotting

Julia has several high quality registered plotting packages:

- Winston.jl - simple but efficient 2D plots
- Gadfly.jl - versatile 2D package with nice output
- PyPlot.jl - Julia interface to Python's Matplotlib - 2D, 3D, implicit, ...
- Plots.jl - wrapper for several backends.

7.1 Prerequisites

Browse the manuals (20 min):

- Winston Documentation
- Gadfly
- The PyPlot module for Julia and Matplotlib
- Intro to Plots in Julia

7.2 Competences

The reader should be able to use some of the features of the above packages.

7.2.1 Remark

Plotting packages are rather complex and depend on additional software, so it is advised to execute corresponding `Pkg.add()` commands in terminal mode.
Also, plotting packges frequently use same (obvious) names for plot functions. When using more than one package in a Julia session, the functions need to be called by specifying the package, as well.
We shall ilustrate the packages on several numerical examples, which also give the flavor of Julia.

7.3 Winston

We compute and plot: * the natural cubic spline, as defined in W. Cheney and D. Kincaid, Numerical Mathematics and Computing, pp. 266-267, and * the standard interpolating polynomial.
We shall use the registered package SpecialMatrices.jl.

```
In [2]: # Pkg.add("SpecialMatrices")
        using Winston
        using SpecialMatrices
        using Polynomials
```

```
In [3]: # Number of intervals
        n=5
        # n+1 points
```

```
t=[1.0,2,4,5,8,9]
y=[-1.0,1,4,0,2,6]
# Computation
h=t[2:end]-t[1:end-1]
b=(y[2:end]-y[1:end-1])./h
v=6*(b[2:end]-b[1:end-1])
H=SymTridiagonal(2*(h[1:end-1]+h[2:end]),h[2:end-1])
```

```
Out[3]: 4x4 SymTridiagonal{Float64}:
 6.0  2.0  0.0  0.0
 2.0  6.0  1.0  0.0
 0.0  1.0  8.0  3.0
 0.0  0.0  3.0  8.0
```

```
In [4]: z=H\v
        z=[0;z;0]
```

```
Out[4]: 6-element Array{Float64,1}:
  0.0
  1.74766
 -6.74299
  3.96262
  1.01402
  0.0
```

```
In [5]: # Define the splines
        B=b-(z[2:end]-z[1:end-1]).*h/6
        S=Array(Any,n)
        S=[x-> y[i]-z[i]*h[i]^2/6+B[i]*(x-t[i])+z[i]*(t[i+1]-x)^3/
            (6*h[i])+z[i+1]*(x-t[i])^3/(6*h[i]) for i=1:n]
```

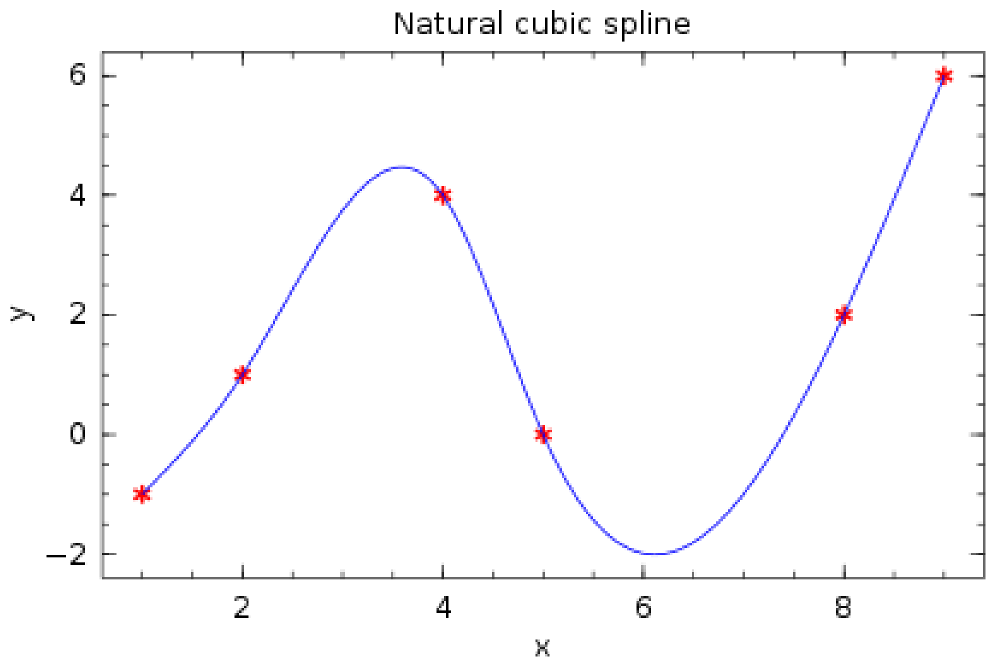
```
Out[5]: 5-element Array{Function,1}:
 (anonymous function)
 (anonymous function)
 (anonymous function)
 (anonymous function)
 (anonymous function)
```

```
In [6]: # Define the points to plot
        lsize=200
        x=linspace(t[1],t[end],lsize)
        zSpline=Array(Float64,lsize)
        for i=1:lsize
            for k=1:n
                if x[i]<=t[k+1]
                    zSpline[i]=S[k](x[i])
                    break
                end
            end
        end
```

```
In [7]: # Plot
        Winston.plot(t,y,"r*",x,zSpline,"b")
```

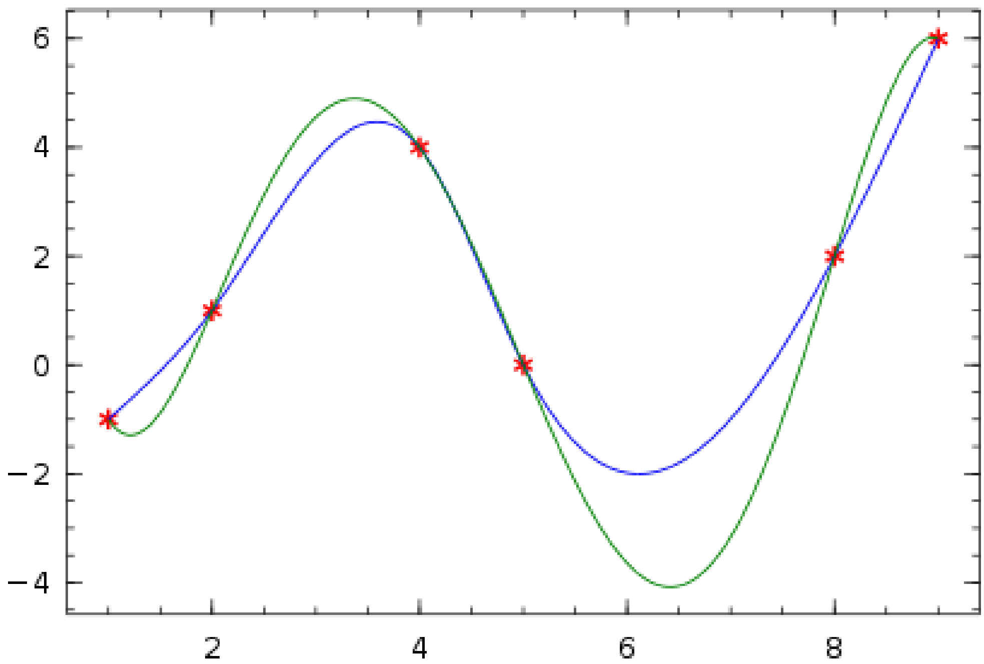
```
Winston.title("Natural cubic spline")
xlabel("x")
ylabel("y")
```

Out[7]:




```
In [8]: # Standard interpolating polynomial
        A=Vandermonde(t)
        p=Poly(full(A)\y)
        yPoly=p(x)
        Winston.plot(t,y,"r*",x,zSpline,"b",x,yPoly,"g")
```

Out[8]:



7.4 Gadfly

We shall illustrate Gadfly with two examples: * function and its derivative, and * exact solution of an initial value problem v.s. the solution computed with our implementation of the Euler's method.

N.B. Gadfly plots can be nicely zoomed in or out. Variety of ODE solvers can be found in the package ODE.jl

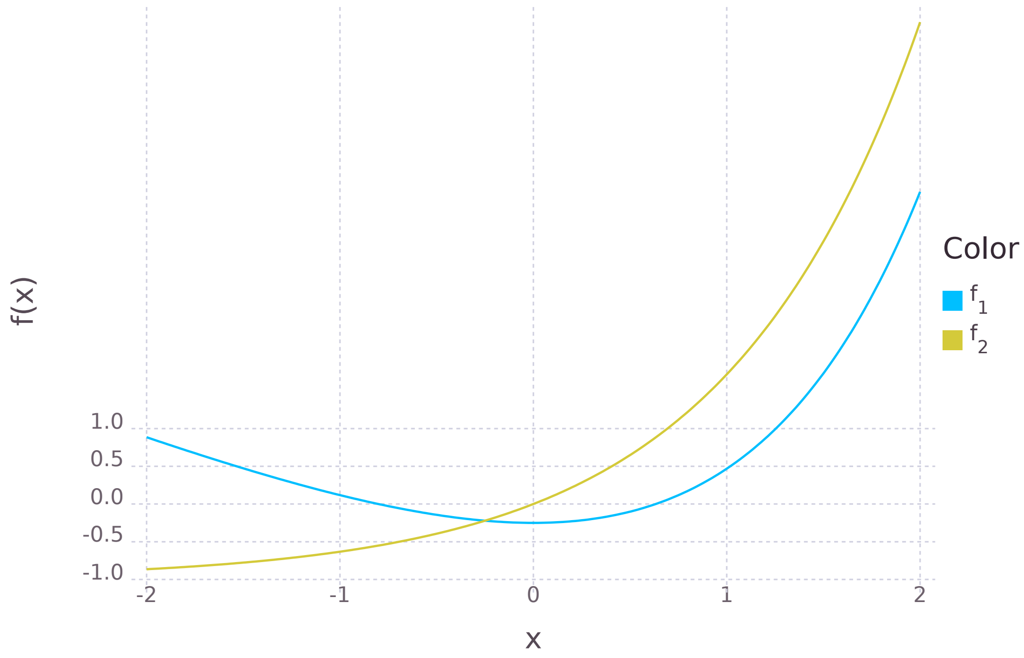
7.4.1 Function and its derivative

Derivative of a function can be: * approximated by finite differences using the package Calculus.jl, * approximated by [automatic differentiation (https://en.wikipedia.org/wiki/Automatic_differentiation) using the package ForwardDiff.jl which is fast, more accurate, and is our method of choice (see the Documentation), and * computed symbolicaly using the package SymPy.jl.

```
In [9]: # Pkg.add("ForwardDiff")
        using ForwardDiff
        using Gadfly
```

```
In [10]: f(x)=exp(x)-x-5.0/4
         Gadfly.plot([f,derivative(f)],-2.0,2.0,Guide.yticks(ticks=[-1.0,-0.5,0.0,0.5,1.0]))
```

Out[10]:



7.4.2 Solution of an initial value problem

The exact solution of the initial value problem

$$y' = x + y, \quad y(0) = 1,$$

is

$$y = 2e^x - x - 1.$$

```
In [11]: # Euler's method
         function myEuler{T,T1}(f::Function,y0::T,x::T1)
             h=x[2]-x[1]
             y=Array(T,length(x))
             y[1]=y0
             for i=2:length(x)
                 y[i]=y[i-1]+h*f(x[i-1],y[i-1])
             end
             y
         end

Out[11]: myEuler (generic function with 1 method)

In [12]: # n subintervals on the interval [0,1]
         n=30
         x=linspace(0,1,n+1)
         f1(x,y)=x+y
         y=myEuler(f1,1.0,x)
```

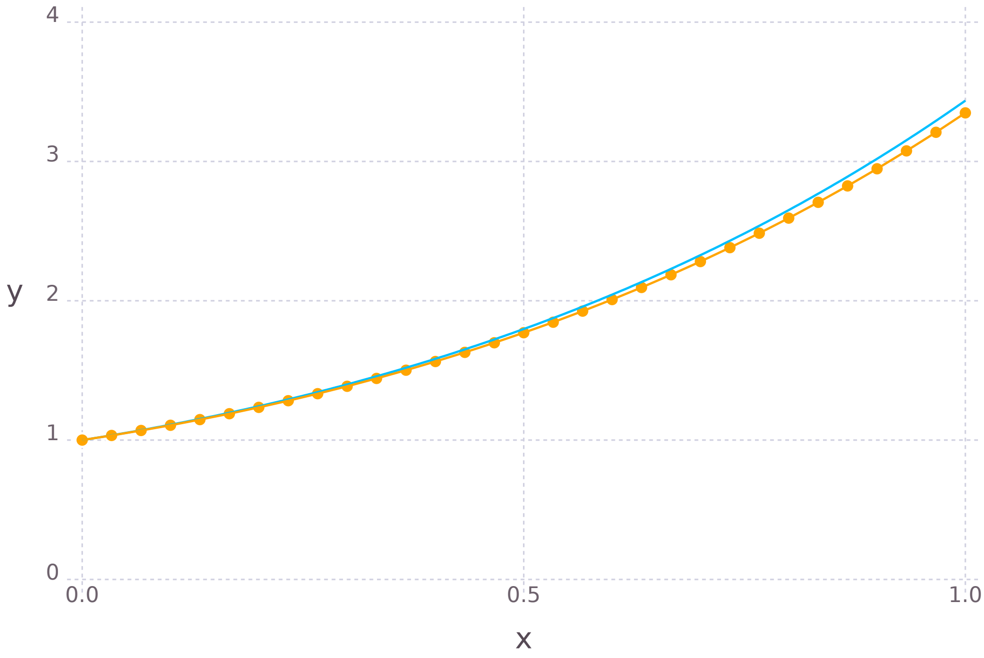
Out[12]: 31-element Array{Float64,1}:

```
 1.0
 1.03333
 1.06889
 1.10674
 1.14697
 1.18964
 1.23485
 1.28268
 1.33321
 1.38654
 1.44276
 1.50197
 1.56425
 ⋮
 2.09572
 2.18669
 2.2818
 2.38119
 2.48501
 2.5934
 2.70651
 2.82451
 2.94755
 3.0758
 3.20943
 3.34864
```

In [13]:

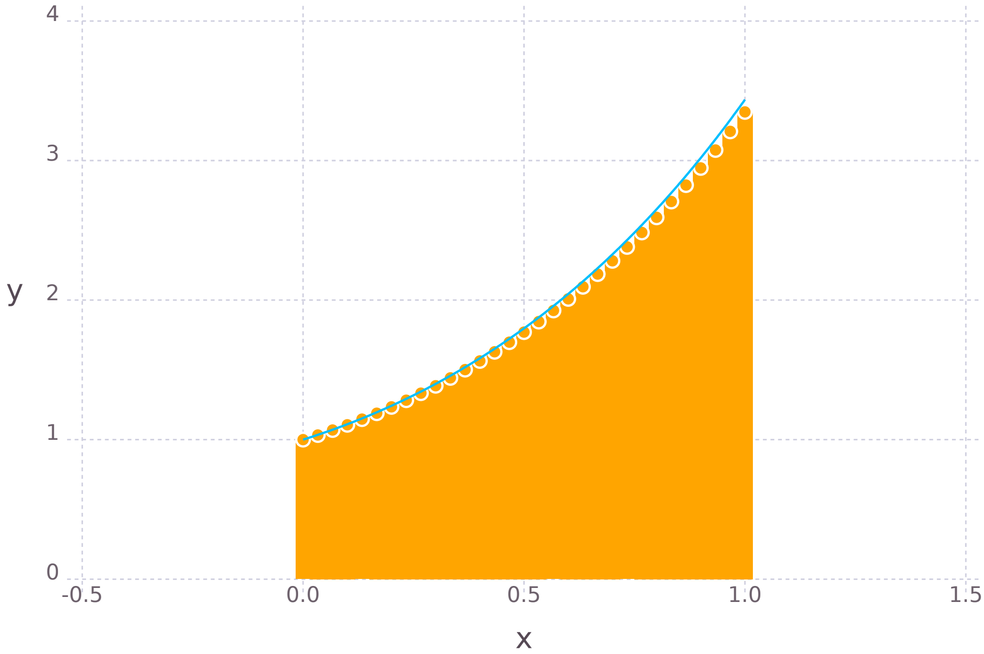
```
# We can plot functions and data sets (points) using layers
solution(x)=2*exp(x)-x-1
Gadfly.plot(layer(solution,0,1),
layer(x=x,y=y,Geom.point, Geom.line, Theme(default_color=colorant"orange")))
```

Out[13]:



In [14]:
```
# Or, with a different geometry
Gadfly.plot(layer(solution,0,1),
layer(x=x,y=y,Geom.point, Geom.bar, Theme(default_color=colorant"orange")))
```

Out[14]:



7.5 PyPlot

We shall illustrate PyPlot with two examples: * 3D and contour plots to graphically solve small system of non-linear equations, and * implicit plot of the solution of Lotka-Volterra equations in the phase-space.

7.5.1 System of non-linear equations

The solutions of the system

$$\begin{aligned} 2(x_1+x_2)^2-(x_1-x_2)^2 &= 8 \\ 5x_1^2+(x_2-3)^2 &= 9 \end{aligned}$$

are

$$\begin{aligned} &S_1=(-1.183467003241957, 1.5868371427229244), \\ &S_2=(1,1). \end{aligned}$$

Let us plot the surfaces:

```
In [15]: using PyPlot
```

```
WARNING: using PyPlot.xlabel in module Main conflicts with an existing identifier.
WARNING: using PyPlot.ylabel in module Main conflicts with an existing identifier.
```

```
In [16]: # Define the system
         x=Vector{Float64}
         f(x)=[2(x[1]+x[2])^2+(x[1]-x[2])^2-8,5*x[1]^2+(x[2]-3)^2-9]
```

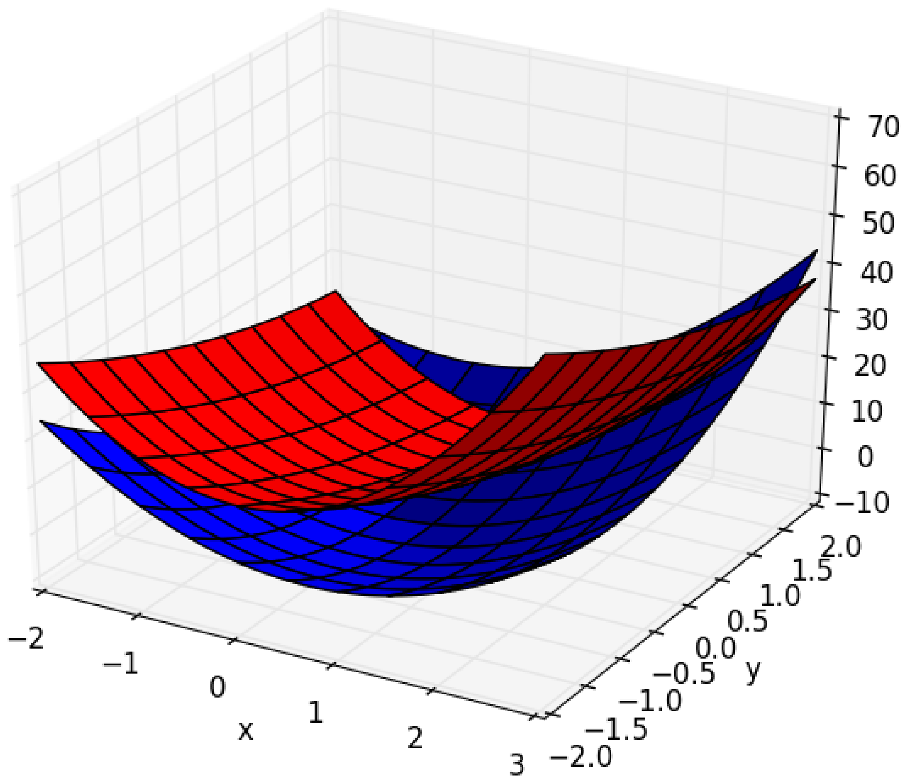
```
Out[16]: f (generic function with 1 method)
```

```
In [17]: # Prepare the meshgrid manually
         gridsize=101
         X=linspace(-2,3,gridsize)
         Y=linspace(-2,2,gridsize)
         gridX= map(Float64,[x for x in X, y in Y])
         gridY= map(Float64,[y for x in X, y in Y])
         # gridX,gridX=meshgrid(X,Y)
         Z1=[f([gridX[i,j],gridY[i,j]])[1] for i=1:gridsize, j=1:gridsize]
         Z2=[f([gridX[i,j],gridY[i,j]])[2] for i=1:gridsize, j=1:gridsize]
```

```
Out[17]: 101x101 Array{Any,2}:
          36.0      35.6016  35.2064  34.8144  ...  12.2544  12.1664  12.0816  12.0
          35.0125   34.6141  34.2189  33.8269      11.2669  11.1789  11.0941  11.0125
          34.05     33.6516  33.2564  32.8644      10.3044  10.2164  10.1316  10.05
          33.1125   32.7141  32.3189  31.9269       9.3669   9.2789   9.1941   9.1125
          32.2      31.8016  31.4064  31.0144       8.4544   8.3664   8.2816   8.2
```

31.3125	30.9141	30.5189	30.1269	...	7.5669	7.4789	7.3941	7.3125
30.45	30.0516	29.6564	29.2644		6.7044	6.6164	6.5316	6.45
29.6125	29.2141	28.8189	28.4269		5.8669	5.7789	5.6941	5.6125
28.8	28.4016	28.0064	27.6144		5.0544	4.9664	4.8816	4.8
28.0125	27.6141	27.2189	26.8269		4.2669	4.1789	4.0941	4.0125
27.25	26.8516	26.4564	26.0644	...	3.5044	3.4164	3.3316	3.25
26.5125	26.1141	25.7189	25.3269		2.7669	2.6789	2.5941	2.5125
25.8	25.4016	25.0064	24.6144		2.0544	1.9664	1.8816	1.8
⋮				⋱				⋮
46.0125	45.6141	45.2189	44.8269		22.2669	22.1789	22.0941	22.0125
47.25	46.8516	46.4564	46.0644	...	23.5044	23.4164	23.3316	23.25
48.5125	48.1141	47.7189	47.3269		24.7669	24.6789	24.5941	24.5125
49.8	49.4016	49.0064	48.6144		26.0544	25.9664	25.8816	25.8
51.1125	50.7141	50.3189	49.9269		27.3669	27.2789	27.1941	27.1125
52.45	52.0516	51.6564	51.2644		28.7044	28.6164	28.5316	28.45
53.8125	53.4141	53.0189	52.6269	...	30.0669	29.9789	29.8941	29.8125
55.2	54.8016	54.4064	54.0144		31.4544	31.3664	31.2816	31.2
56.6125	56.2141	55.8189	55.4269		32.8669	32.7789	32.6941	32.6125
58.05	57.6516	57.2564	56.8644		34.3044	34.2164	34.1316	34.05
59.5125	59.1141	58.7189	58.3269		35.7669	35.6789	35.5941	35.5125
61.0	60.6016	60.2064	59.8144	...	37.2544	37.1664	37.0816	37.0

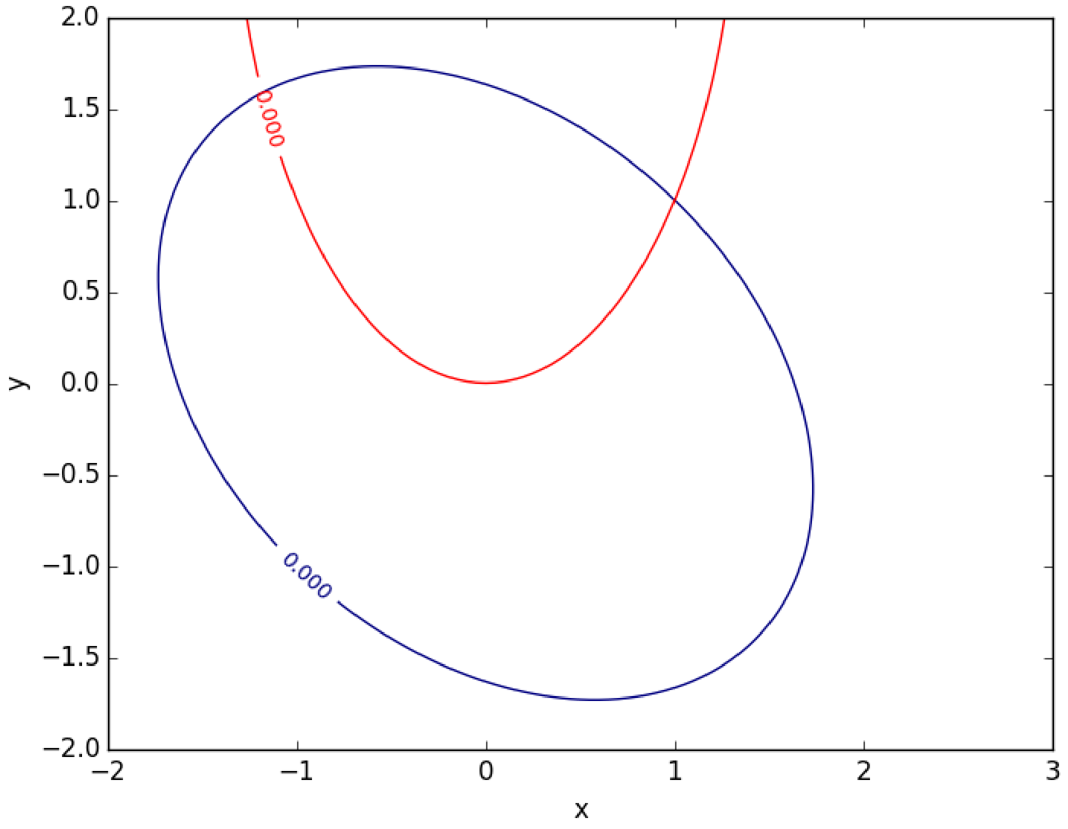
```
In [18]: # Plot
         PyPlot.surf(gridX,gridY,Z1)
         PyPlot.surf(gridX,gridY,Z2,color="red")
         PyPlot.xlabel("x")
         PyPlot.ylabel("y")
```



```
Out[18]: PyObject <matplotlib.text.Text object at 0x7fb7184fe0b8>
```

Let us plot the contours at $z = 0$:

```
In [19]: C1=contour(gridX,gridY,Z1,levels=[0])
         C2=contour(gridX,gridY,Z2,levels=[0],colors="red")
         clabel(C1,inline=1, fontsize=10)
         clabel(C2,inline=1, fontsize=10)
         PyPlot.xlabel("x")
         PyPlot.ylabel("y")
```




```
Out[19]: PyObject <matplotlib.text.Text object at 0x7fb717a6b438>
```

7.5.2 Plotting implicit functions

The phase-space solution of the Lotka-Volterra system of equations in dimensionless variables in scaled time has the form

$$yx^{\sigma} = Ce^{y}e^{x\sigma}.$$

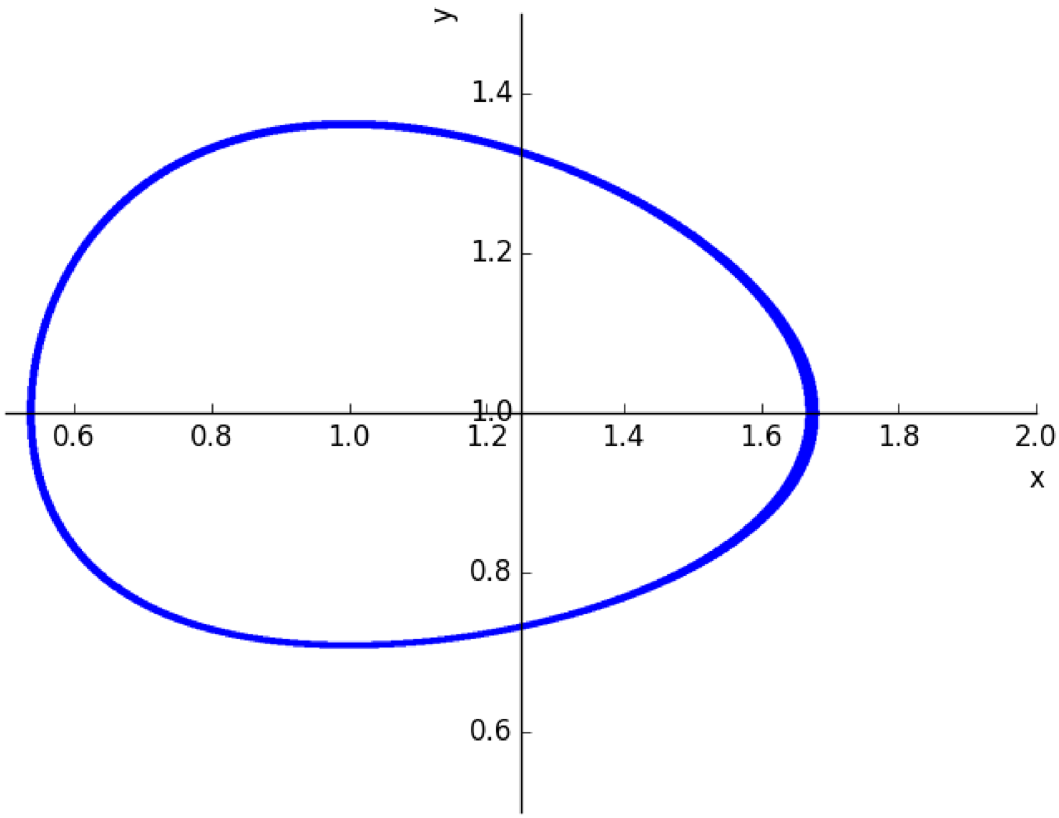
For implicit plots, we also need the package `SymPy.jl`. Plotting takes a little longer.

```
In [21]: using SymPy

In [22]: # Define the parameters
         C=0.25
         σ=1/3
         # Define symbolic variables
         @vars x,y

Out[22]: (x,y)

In [23]: SymPy.plot_implicit(Eq(y*x^σ,C*exp(y+σ*x)),(x,0.5,2),(y,0.5,1.5))
```




```
Out[23]: PyObject <sympy.plotting.plot.Plot object at 0x7fb71791df60>

In [ ]:
```

8 Tutorial 1 - Examples in Julia

8.1 Assignment 1

Using the package `Polynomials.jl`, write the function which implements Graeffe's method (see also here) for computing roots of polynomials with only real roots with simple moduli.

In the function, use Julia's `BigFloat` numbers to overcome the main disadvantage of the method. What is the number of significant decimal digits, and the largest and the smallest number?

Test the function on the Wilkinson's polynomial $\omega(x)$, and the Chebyshev polynomial $T_{50}(x)$ (the latter needs to be transformed in order to apply the method).

Compare your solutions with the exact solutions.

8.2 Assignment 2

Write the function which computes simple LU factorization (without pivoting) where the matrix is overwritten by the factors.

Make sure that the function also works with block-matrices.

Compare the speed on standard matrices and block-matrices with the built-in LU factorization (which also uses block algorithm AND pivoting). Check the accuracy.

8.3 Assignment 3

Use the function `eigvals()` to compute the eigenvalues of k random matrices (with uniform and normal distribution of elements) of order n.

Plot the results using the macro `@manipulate` from the package `Interact.jl`. Use `Winston.jl` for plotting.

Are the eigenvalues random? Can you describe their behaviour? Can random matrices be used to test numerical algorithms?

```
In [ ]:
```

9 Solutions 1 - Examples in Julia

9.1 Assignment 1

The function `eps()` return the smallest real number larger than 1.0. It can be called for each of the `AbstractFloat` types.
Functions `realmin()` and `realmax()` return the largest and the smallest positive numbers representable in the given type.

```
In [1]: ?eps

search: eps RepString @elapsed indexpids expanduser escape_string peakflops

Out[1]:

eps(::DateTime) -> Millisecond
eps(::Date) -> Day
```

Returns `Millisecond(1)` for `DateTime` values and `Day(1)` for `Date` values.

`eps(x)`

The distance between `x` and the next larger representable floating-point value of the same `DataType` as `x`.

`eps(T)`

The distance between 1.0 and the next larger representable floating-point value of `DataType T`. Only floating-point types are sensible arguments.

`eps()`

The distance between 1.0 and the next larger representable floating-point value of `Float64`.

```
In [2]: ?realmax

search: realmax realmin readdlm ReadOnlyMemoryError

Out[2]:

realmax(T)
```

The highest finite value representable by the given floating-point DataType `T`.

```
In [3]: subtypes(AbstractFloat)

Out[3]: 4-element Array{Any,1}:
         BigFloat
         Float16
         Float32
         Float64
```

```
In [4]: # Default values are for Float66
        eps(), realmax(), realmin()
```

```
Out[4]: (2.220446049250313e-16,1.7976931348623157e308,2.2250738585072014e-308)
```

```
In [5]: T=Float32
        eps(T), realmax(T), realmin(Float32)
```

```
Out[5]: (1.1920929f-7,3.4028235f38,1.1754944f-38)
```

```
In [6]: T=BigFloat
        eps(T), realmax(T), realmin(T), map(Int64,round(log10(1/eps(T))*log(10)/log(2)))
```

```
Out[6]: (1.727233711018888925077270372560079914223200072887256277004740694033718360632485e-7
```

We see that BigFloat has approximately 77 significant decimal digits (actually 256 bits) and very large exponents. This makes the format ideal for Greaffe's method.
Precision of BigFloat can be increased, but exponents do not change.

```
In [7]: get_bigfloat_precision()
```

```
Out[7]: 256
```

```
In [8]: set_bigfloat_precision(512)
        eps(T), realmax(T)
```

```
Out[8]: (1.491668146240041348658193063092586767475294306920081378854303666641255677014023660
```

```
In [9]: set_bigfloat_precision(256)
```

```
Out[9]: 256
```

Here is the function for Graeffe's method. We also define small test polynomial with all real simple zeros.

```
In [10]: using Polynomials
         p=poly([1,2,3,4])
```

```
Out[10]: Poly(24 - 50x + 35x^2 - 10x^3 + x^4)
```

```
In [11]: roots(p)
```

```
Out[11]: 4-element Array{Float64,1}:
          1.0
          2.0
          3.0
          4.0
```

```
In [12]: function Graeffe{T}(p::Poly{T},steps::Int64)
             # map the polynomial to BigFloat
             pbig=Poly(map(BigFloat,coeffs(p)))
             px=Poly([zero(BigFloat),one(BigFloat)])
```

```
    n=degree(p)
    σ=map(BigFloat,2^steps)
    for k=1:steps
        peven=Poly(coeffs(pbig)[1:2:end])
        podd=Poly(coeffs(pbig)[2:2:end])
        pbig=peven^2-podd^2*px
    end
    # @show p[end]
    y=Array(BigFloat,n)
    # Normalize if p is not monic
    y[1]=-pbig[end-1]/pbig[end]
    for k=2:n
        y[k]=-pbig[end-k]/pbig[end-(k-1)]
    end
    # Extract the roots
    for k=1:n
        y[k]=exp(log(y[k])/σ)
    end
    # Return root in Float64
    map(Float64,y)
end
```

```
Out[12]: Graeffe (generic function with 1 method)
```

```
In [13]: Graeffe(p,8)
```

```
Out[13]: 4-element Array{Float64,1}:
          4.0
          3.0
          2.0
          1.0
```

Now the Wilkinson's polynomial:

```
In [14]: ω=poly(collect(one(BigFloat):20))
```

```
Out[14]: Poly(2.43290200817664000000000000000000000000000000000000000000000000000000000
```

```
In [15]: Graeffe(ω,8)
```

```
Out[15]: 20-element Array{Float64,1}:
          20.0
          19.0
          18.0
          17.0
          16.0
          15.0
          14.0
          13.0
          12.0
          11.0
          10.0
```

```
9.0
8.0
7.0
6.0
5.0
4.0
3.0
2.0
1.0
```

We need to generate the Chebyshev polynomial $T_{50}(x)$ using the three term recurence.

```
In [16]: n=50
         T0=Poly([BigInt(1)])
         T1=Poly([0,1])
         Tx=Poly([0,1])
         for i=3:n+1
             T=2*Tx*T1-T0
             T0=T1
             T1=T
         end
```

```
In [17]: T1
```

```
Out[17]: Poly(-1 + 1250x^2 - 260000x^4 + 21528000x^6 - 947232000x^8 + 25638412800x^10 - 466
```

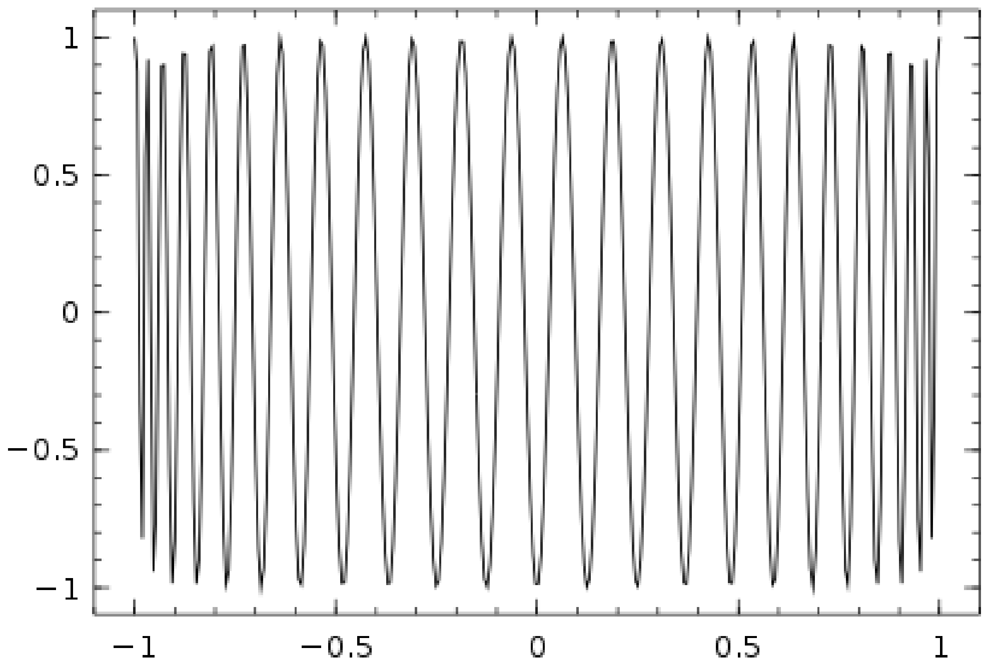
```
In [19]: using Winston
```

```
In [21]: x=linspace(-1,1,300)
```

```
Out[21]: linspace(-1.0,1.0,300)
```

```
In [22]: plot(x,T1(x))
```

```
Out[22]:
```

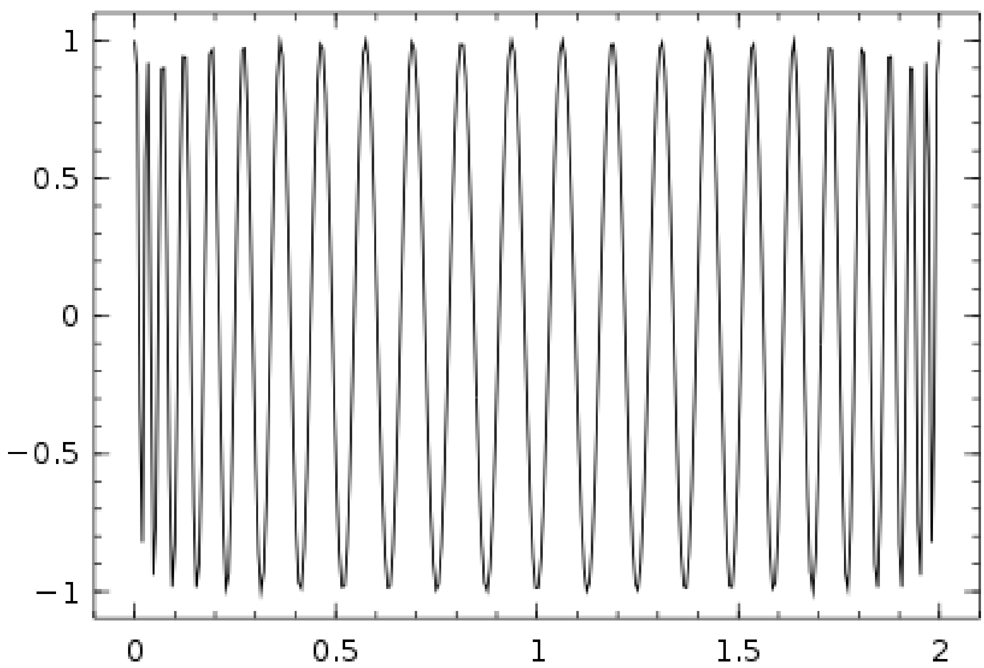


In order to use Graeffe's method, we need to shift T to the right by one, so that all roots also have simple moduli, that is we compute $T(1-x)$:

```
In [23]: Ts=T1(Poly([BigFloat(1),-1]));
```

```
In [24]: xs=linspace(0,2,300)
         plot(xs, Ts(xs))
```

```
Out[24]:
```



In [25]:
```
# Computed roots, 16 steps are fine
y=Graeffe(Ts,12)-1
```

Out[25]:
```
50-element Array{Float64,1}:
  0.999507
  0.995562
  0.987688
  0.975917
  0.960294
  0.940881
  0.917755
  0.891007
  0.860742
  0.827081
  0.790155
  0.750111
  0.707107
  ⋮
 -0.750111
 -0.790155
 -0.827081
 -0.860742
 -0.891007
 -0.917755
 -0.940881
 -0.960294
 -0.975917
 -0.987688
 -0.995562
 -0.999507
```

In [26]:
```
# Exact roots
z=map(Float64,[cos((2*k-1)*pi/(2*n)) for k=1:n])
```

Out[26]:
```
50-element Array{Float64,1}:
  0.999507
  0.995562
  0.987688
  0.975917
  0.960294
  0.940881
  0.917755
  0.891007
  0.860742
  0.827081
  0.790155
  0.750111
  0.707107
```

```
        ⋮
 -0.750111
 -0.790155
 -0.827081
 -0.860742
 -0.891007
 -0.917755
 -0.940881
 -0.960294
 -0.975917
 -0.987688
 -0.995562
 -0.999507
```

```
In [27]: # Relative error
         maximum(abs(z-y)./z)
```

```
Out[27]: 1.5019142646242862e-7
```

9.2 Assignment 2

The key is that / works for block matrices, too. A is overwritten and must therefore be copied at the beggining of the function, so that the original matrix is not overwritten.

```
In [28]: function mylu{T}(A1::Array{T}) # Strang, page 100
             A=copy(A1)
             n,m=size(A)
             for k=1:n-1
                 for rho=k+1:n
                     A[rho,k]=A[rho,k]/A[k,k]
                     for l=k+1:n
                         A[rho,l]=A[rho,l]-A[rho,k]*A[k,l]
                     end
                 end
             end
             # We return L and U
             L=tril(A,-1)
             U=triu(A)
             # This is the only difference for the block case
             for i=1:maximum(size(L))
                 L[i,i]=one(L[1,1])
             end
             L,U
         end
```

```
Out[28]: mylu (generic function with 1 method)
```

```
In [29]: A=rand(5,5)
```

```
Out[29]: 5x5 Array{Float64,2}:
          0.403232  0.0356822  0.0464912  0.604242  0.532244
```

```
 0.358695  0.271174   0.0499363  0.107327  0.925905
 0.054824  0.733145   0.633516   0.428514  0.362232
 0.345642  0.862507   0.585965   0.402968  0.227538
 0.802874  0.948046   0.283802   0.904392  0.389828
```

In [30]:
```
mylu(A)
```

Out[30]:
```
(
5x5 Array{Float64,2}:
 1.0       0.0      0.0        0.0     0.0
 0.889549  1.0      0.0        0.0     0.0
 0.135961  3.04174  1.0        0.0     0.0
 0.857179  3.47454  0.858933   1.0     0.0
 1.9911    3.66281  0.265858  -20.0664  1.0,

5x5 Array{Float64,2}:
 0.403232  0.0356822  0.0464912    0.604242    0.532244
 0.0       0.239433   0.00858012  -0.430175    0.452448
 0.0       0.0        0.601097     1.65484    -1.08636
 0.0       0.0        0.0         -0.041711   -0.867629
 0.0       0.0        0.0          0.0       -19.4485   )
```

In [31]:
```
L,U=mylu(A)
```

Out[31]:
```
(
5x5 Array{Float64,2}:
 1.0       0.0      0.0        0.0     0.0
 0.889549  1.0      0.0        0.0     0.0
 0.135961  3.04174  1.0        0.0     0.0
 0.857179  3.47454  0.858933   1.0     0.0
 1.9911    3.66281  0.265858  -20.0664  1.0,

5x5 Array{Float64,2}:
 0.403232  0.0356822  0.0464912    0.604242    0.532244
 0.0       0.239433   0.00858012  -0.430175    0.452448
 0.0       0.0        0.601097     1.65484    -1.08636
 0.0       0.0        0.0         -0.041711   -0.867629
 0.0       0.0        0.0          0.0       -19.4485   )
```

In [32]:
```
L*U-A
```

Out[32]:
```
5x5 Array{Float64,2}:
  0.0           0.0  0.0   0.0           0.0
  0.0           0.0  0.0   0.0           0.0
 -6.93889e-18   0.0  0.0   0.0           0.0
  0.0           0.0  0.0  -1.11022e-16   1.11022e-16
  0.0           0.0  0.0   0.0          -2.22045e-16
```

We now try block-matrices. First, a small example:

In [33]:
```
# Try k,l=32,16 i k,l=64,8
k,l=2,4
Ab=[rand(k,k) for i=1:l, j=1:l]
```

```
Out[33]: 4x4 Array{Any,2}:
          2x2 Array{Float64,2}:
          0.859241  0.290347
          0.546656  0.251575    ...  2x2 Array{Float64,2}:
          0.569337  0.706363
          0.489503  0.583619
          2x2 Array{Float64,2}:
          0.134563   0.665494
          0.0123687  0.471731     2x2 Array{Float64,2}:
          0.344356  0.75955
          0.947989  0.589276
          2x2 Array{Float64,2}:
          0.552753  0.598627
          0.8736    0.797129       2x2 Array{Float64,2}:
          0.110275  0.730796
          0.312197  0.601599
          2x2 Array{Float64,2}:
          0.999222  0.612258
          0.32229   0.273818       2x2 Array{Float64,2}:
          0.67545   0.747955
          0.357495  0.778605
```

```
In [34]: Ab[1,1]
```

```
Out[34]: 2x2 Array{Float64,2}:
          0.859241  0.290347
          0.546656  0.251575
```

```
In [35]: L,U=mylu(Ab)
```

```
Out[35]: (
         4x4 Array{Any,2}:
          2x2 Array{Float64,2}:
          1.0  0.0
          0.0  1.0                       ...  2x2 Array{Float64,2}:
          0.0  0.0
          0.0  0.0
          2x2 Array{Float64,2}:
          -5.74377  9.27429
          -4.435    6.99361     2x2 Array{Float64,2}:
          0.0  0.0
          0.0  0.0
          2x2 Array{Float64,2}:
          -3.27598  6.16038
          -3.75985  7.50786     2x2 Array{Float64,2}:
          0.0  0.0
          0.0  0.0
          2x2 Array{Float64,2}:
          -1.45038  4.10761
          -1.19428  2.46676     2x2 Array{Float64,2}:
          1.0  0.0
          0.0  1.0,
```

```
4x4 Array{Any,2}:
 2x2 Array{Float64,2}:
 0.859241  0.290347
 0.546656  0.251575  ...  2x2 Array{Float64,2}:
 0.569337  0.706363
 0.489503  0.583619
 2x2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0                        2x2 Array{Float64,2}:
 -0.925294   -0.595918
  0.0496047  -0.359614
 2x2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0                        2x2 Array{Float64,2}:
 -1.02537   -0.273877
  0.953836  -0.517351
 2x2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0                        2x2 Array{Float64,2}:
  9.7431   3.08869
 -8.6654  -3.96928        )
```

```
In [36]: L*U-Ab
```

```
Out[36]: 4x4 Array{Any,2}:
 2x2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0                                      ...  2x2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
 2x2 Array{Float64,2}:
 -1.33227e-15  -2.22045e-16
 -2.22045e-16   2.22045e-16     2x2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
 2x2 Array{Float64,2}:
 -4.44089e-16  -1.11022e-16
 -2.22045e-16   0.0             2x2 Array{Float64,2}:
 0.0  0.0
 0.0  1.11022e-16
 2x2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0                                        2x2 Array{Float64,2}:
 4.44089e-16  -2.22045e-16
 4.44089e-16   2.22045e-16
```

```
In [37]: norm(ans) # This is not defined
```

```
    LoadError: MethodError: `zero` has no method matching zero(::Type{Any})
while loading In[37], in expression starting on line 1
```

We need a convenience function to unblock the block-matrix:

```
In [38]: unblock(A) = mapreduce(identity, hcat, [mapreduce(identity, vcat, A[:,i]) for i = 1
```

```
Out[38]: unblock (generic function with 1 method)
```

```
In [39]: unblock(Ab)
```

```
Out[39]: 8x8 Array{Float64,2}:
          0.859241   0.290347  0.882112   0.0589339   …  0.561973  0.569337  0.706363
          0.546656   0.251575  0.0792403  0.00382017     0.511518  0.489503  0.583619
          0.134563   0.665494  0.26147    0.899233       0.267063  0.344356  0.75955
          0.0123687  0.471731  0.0504757  0.912636       0.148593  0.947989  0.589276
          0.552753   0.598627  0.164754   0.676265       0.559291  0.110275  0.730796
          0.8736     0.797129  0.664858   0.175908    …  0.747539  0.312197  0.601599
          0.999222   0.612258  0.222019   0.838063       0.314485  0.67545   0.747955
          0.32229    0.273818  0.933838   0.32583        0.830421  0.357495  0.778605
```

```
In [40]: norm(unblock(L*U-Ab))
```

```
Out[40]: 1.6583733836878267e-15
```

We now compute timings an errors for bigger example:

```
In [41]: # This is 512x512 matrix consisting of 16x16 blocks of dimension 32x32
         k,l=32,16
         Ab=[rand(k,k) for i=1:l, j=1:l]
         # Unblocked version
         A=unblock(Ab);
```

```
In [42]: ?lu
```

```
search: lu lufact lufact! flush flush_cstdio ClusterManager mylu values include
```

```
Out[42]:
```

```
lu(A) -> L, U, p
```

Compute the LU factorization of `A`, such that `A[p,:] = L*U`.

```
In [43]: # Built-in LAPACK function with pivoting
         @time L,U,p=lu(A);
```

```
0.114271 seconds (79.83 k allocations: 9.639 MB)
```

```
In [44]: norm(L*U-A[p,:])
```

```
Out[44]: 3.4769000543978225e-14
```

```
In [45]: # mylu() unblocked
         @time L,U=mylu(A);
```

```
0.285764 seconds (13 allocations: 6.000 MB, 1.38% gc time)
```

```
In [46]: norm(L*U-A)
```

```
Out[46]: 6.854977602946937e-12
```

```
In [47]: # mylu() on a block-matrix - much faster, but NO pivoting
         @time L,U=mylu(Ab);
```

```
0.088273 seconds (7.04 k allocations: 26.606 MB, 3.29% gc time)
```

```
In [48]: norm(unblock(L*U-Ab))
```

```
Out[48]: 1.682521213463684e-11
```

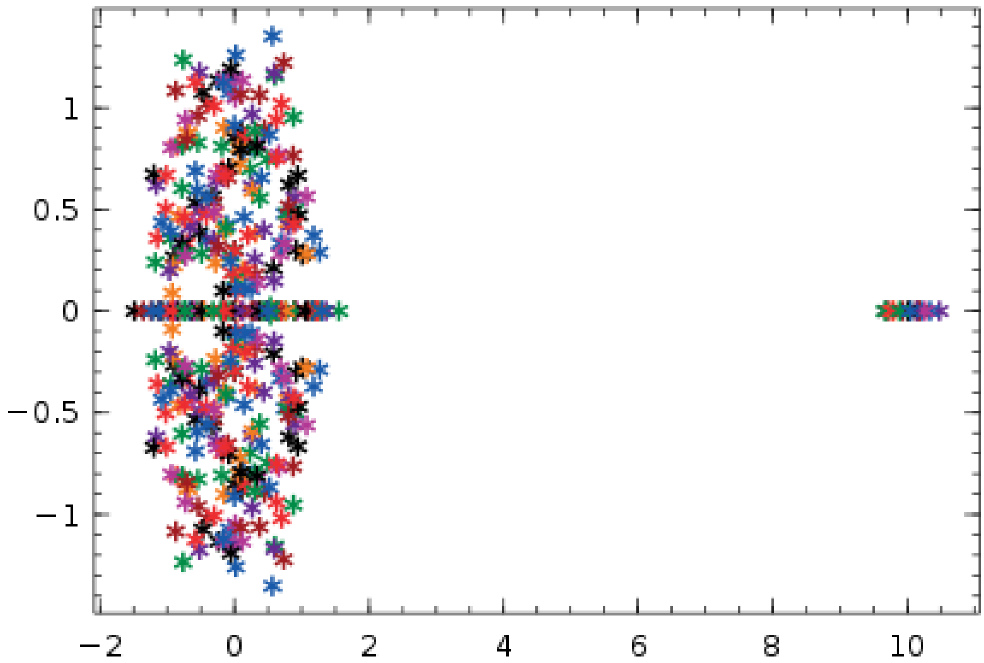
9.3 Assignment 3

```
In [49]: k=20
         n=20
         E=Array(Any,n,k)
         # Unsymmetrix random uniform distribution
         for i=1:k
             A=rand(n,n)
             E[:,i]=eigvals(A)
         end
         # We need this since plot cannot handle ‘Any‘
         E=map(eltype(E[1,1]),E)
```

```
Out[49]: 20x20 Array{Complex{Float64},2}:
       10.1512+0.0im            10.151+0.0im        ...     10.0494+0.0im
     -1.48639+0.0im            1.17471+0.0im              -1.21833+0.0im
     -1.12961+0.0im           0.626151+0.940903im        -0.942371+0.392003im
    -0.899554+0.266722im      0.626151-0.940903im        -0.942371-0.392003im
    -0.899554-0.266722im     -0.857192+0.81126im         -0.162806+1.12041im
    -0.575746+0.53258im      -0.857192-0.81126im    ...   -0.162806-1.12041im
    -0.575746-0.53258im       -1.14461+0.358316im        -0.578274+0.690318im
  -0.00294929+0.8518im        -1.14461-0.358316im        -0.578274-0.690318im
  -0.00294929-0.8518im       0.0969548+0.88284im           1.17872+0.371568im
    -0.324318+0.562959im     0.0969548-0.88284im           1.17872-0.371568im
    -0.324318-0.562959im      -1.11309+0.0im        ...     1.30492+0.0im
   0.00379823+0.299854im     -0.710935+0.330363im         0.523667+0.866791im
   0.00379823-0.299854im     -0.710935-0.330363im         0.523667-0.866791im
     0.815602+0.620837im     -0.176539+0.697334im         0.142096+0.462282im
     0.815602-0.620837im     -0.176539-0.697334im         0.142096-0.462282im
      1.00644+0.275937im      0.589194+0.0im        ...  -0.0597885+0.242614im
      1.00644-0.275937im      0.298893+0.0im            -0.0597885-0.242614im
     0.578307+0.20953im     -0.0448251+0.0im              0.498165+0.0im
     0.578307-0.20953im      0.0239464+0.092426im         0.171089+0.113072im
      1.19624+0.0im          0.0239464-0.092426im         0.171089-0.113072im
```

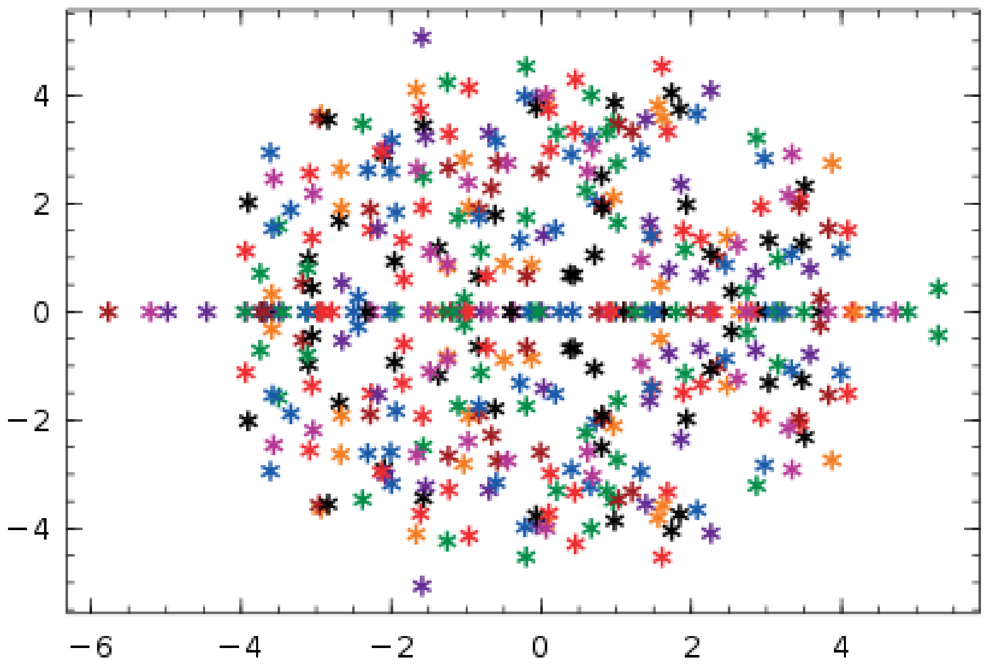
In [50]: Winston.plot(E,"*")

Out[50]:



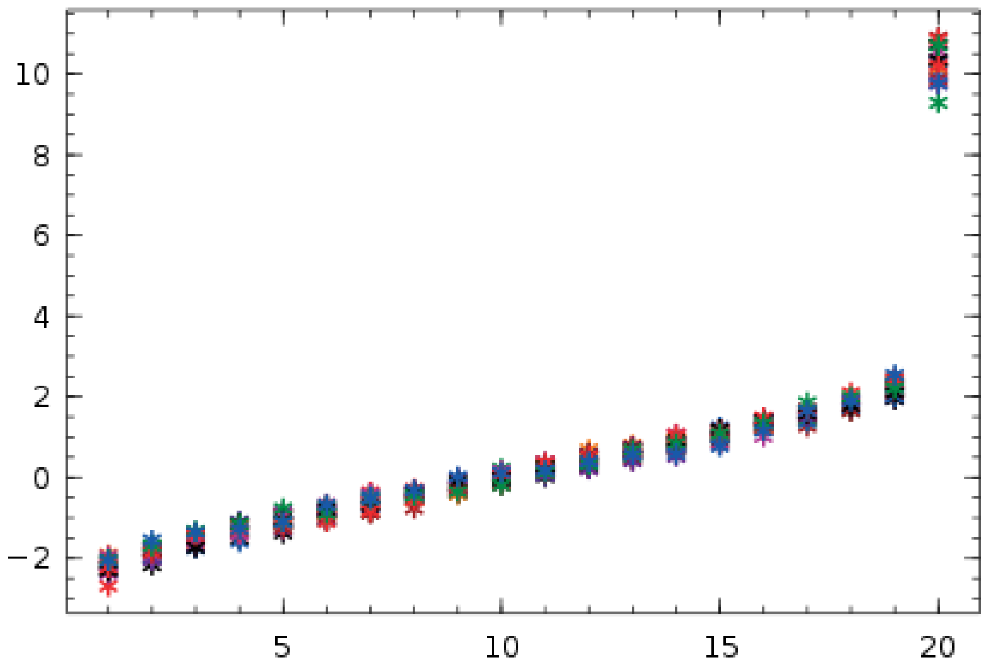
In [51]:
```
# Unsymmetric random normal distribution
E=Array(Any,n,k)
for i=1:k
    A=randn(n,n)
    E[:,i]=eigvals(A)
end
# We need this for plot to work
E=map(eltype(E[1,1]),E)
Winston.plot(E,"*")
```

Out[51]:



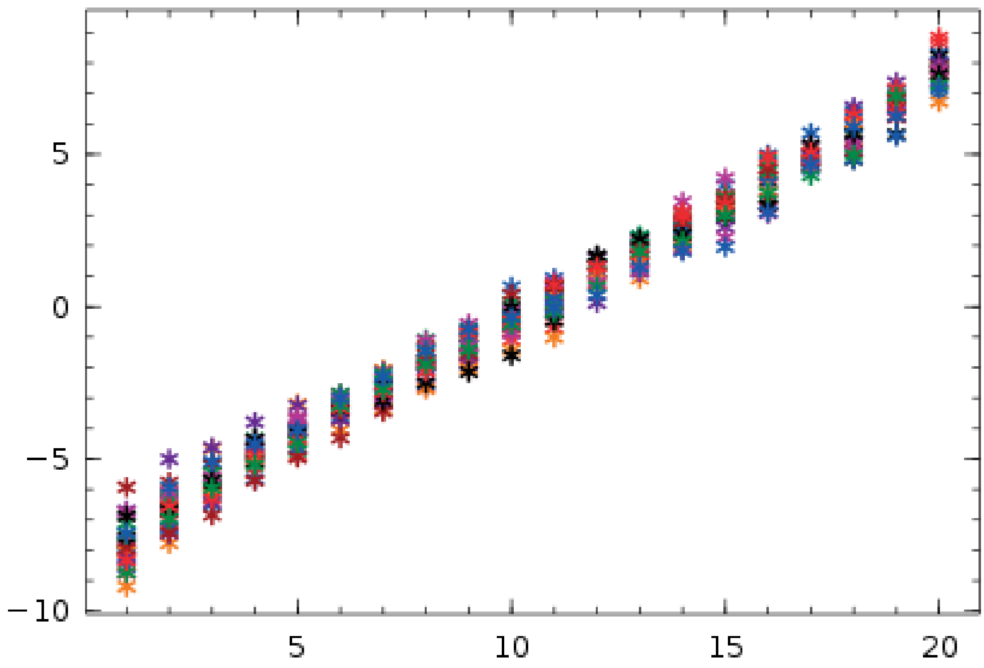
In [52]:
```
# Symmetric random uniform distribution
E=Array(Any,n,k)
for i=1:k
    A=rand(n,n)
    A=triu(A)+triu(A,1)'
    E[:,i]=eigvals(A)
end
# We need this for plot to work
E=map(eltype(E[1,1]),E)
Winston.plot(E,"*")
```

Out[52]:



In [53]:
```
# Symmetric random normal distribution
E=Array(Any,n,k)
for i=1:k
    A=randn(n,n)
    A=triu(A)+triu(A,1)'
    E[:,i]=eigvals(A)
end
# We need this for plot to work
E=map(eltype(E[1,1]),E)
Winston.plot(E,"*")
```

Out[53]:

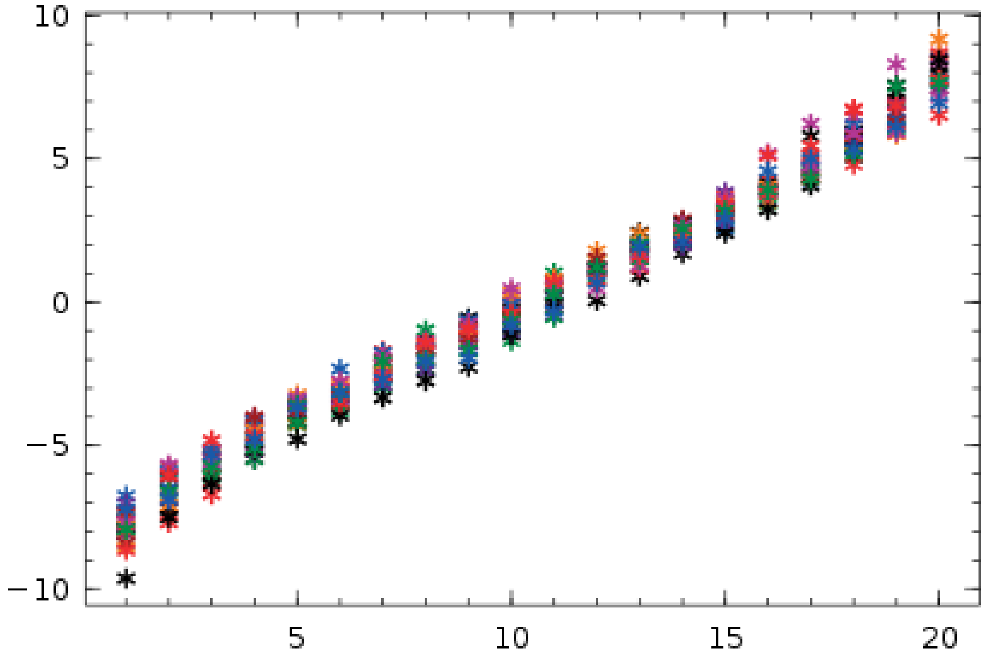



```
In [54]: # Now the interactive part
         using Interact
         @manipulate for k=10:30, n=10:30
             E=Array(Any,n,k)
             for i=1:k
                 A=randn(n,n)
                 A=triu(A)+triu(A,1)'
                 E[:,i]=eigvals(A)
             end
             # We need this for plot to work
             E=map(eltype(E[1,1]),E)
             Winston.plot(E,"*")
         end
```

```
Interact.Slider{Int64}(Signal{Int64}(20, nactions=0),"k",20,10:30,true)
```

```
Interact.Slider{Int64}(Signal{Int64}(20, nactions=0),"n",20,10:30,true)
```

Out[54]:



Mathematics is about spotting patterns (Alan Edelman)

```
In [ ]:
```