tutorial

September 7, 2017

1 Julia Basics

This is a *basic* introduction to an IJulia notebook.

```
In [1]: 1+1

Out[1]: 2

In [2]: x = 3

Out[2]: 3

In [3]: x

Out[3]: 3

In [4]: 3*x + x^3 + sin(x)

Out[4]: 36.141120008059865

In [5]: z = 3 + 4im

Out[5]: 3 + 4im

In [6]: z^3

Out[6]: -117 + 44im

In [7]: exp(z)

Out[7]: -13.128783081462158 - 15.200784463067954im

In [8]: sqrt(z)

Out[8]: 2.0 + 1.0im

In [9]: sqrt(-2+0im)

Out[9]: 0.0 + 1.4142135623730951im

In [10]: abs(Out[12])
```

```
        KeyError: key 12 not found


         in getindex(::Dict{Int64,Any}, ::Int64) at ./dict.jl:688
```

In [11]: abs(3+4im)

Out[11]: 5.0

In [12]: x = [1,2,3]

```
Out[12]: 3-element Array{Int64,1}:
 1
 2
 3
```

In [13]: y = [1 2 3]

```
Out[13]: 1×3 Array{Int64,2}:
 1  2  3
```

In [14]: y * x

```
Out[14]: 1-element Array{Int64,1}:
 14
```

In [15]: x * y

```
Out[15]: 3×3 Array{Int64,2}:
 1  2  3
 2  4  6
 3  6  9
```

In [16]: x + y

```
DimensionMismatch("dimensions must match")


 in promote_shape(::Tuple{Base.OneTo{Int64},Base.OneTo{Int64}}, ::Tuple{Base.OneTo{Int64}}) at .

 in promote_shape(::Tuple{Base.OneTo{Int64}}, ::Tuple{Base.OneTo{Int64},Base.OneTo{Int64}}) at .

 in promote_shape(::Array{Int64,1}, ::Array{Int64,2}) at ./operators.jl:397

 in _elementwise(::Base.#+, ::Type{Int64}, ::Array{Int64,1}, ::Array{Int64,2}) at ./arraymath.jl

 in +(::Array{Int64,1}, ::Array{Int64,2}) at ./arraymath.jl:49
```

In [17]: A = x .+ y

```
Out[17]: 3×3 Array{Int64,2}:
 2  3  4
 3  4  5
 4  5  6
```

In [18]: x .* x

```
Out[18]: 3-element Array{Int64,1}:
 1
 4
 9
```

```
In [19]: x = rand(20)

Out[19]: 20-element Array{Float64,1}:
          0.865113
          0.933176
          0.618455
          0.617167
          0.401063
          0.73383
          0.435374
          0.0494422
          0.937702
          0.26655
          0.369985
          0.361838
          0.498797
          0.173794
          0.948838
          0.108735
          0.262631
          0.0768341
          0.825819
          0.620872

In [20]: x[2]

Out[20]: 0.9331756932130784

In [21]: x[2:5]

Out[21]: 4-element Array{Float64,1}:
          0.933176
          0.618455
          0.617167
          0.401063

In [22]: x[7:2:end] # x[7], x[9], x[11], ... end of array

Out[22]: 7-element Array{Float64,1}:
          0.435374
          0.937702
          0.369985
          0.498797
          0.948838
          0.262631
          0.825819

In [23]: x[x .> 0.5]

Out[23]: 9-element Array{Float64,1}:
          0.865113
          0.933176
          0.618455
          0.617167
          0.73383
          0.937702
```

```
0.948838
0.825819
0.620872
```

Any function `f` can be applied elementwise to an array `x` by the syntax `f.(x)`.

You may be thinking that Matlab and Numpy do this without dots, but it turns out that the `.` in `f.(x)` enables extensive possibilities that aren't possible in other language. See also this blog post on the real power of this syntax.

e.g. compute $e^{\sin x_i}$ for each element x_i=`x[i]` of the array `x`:

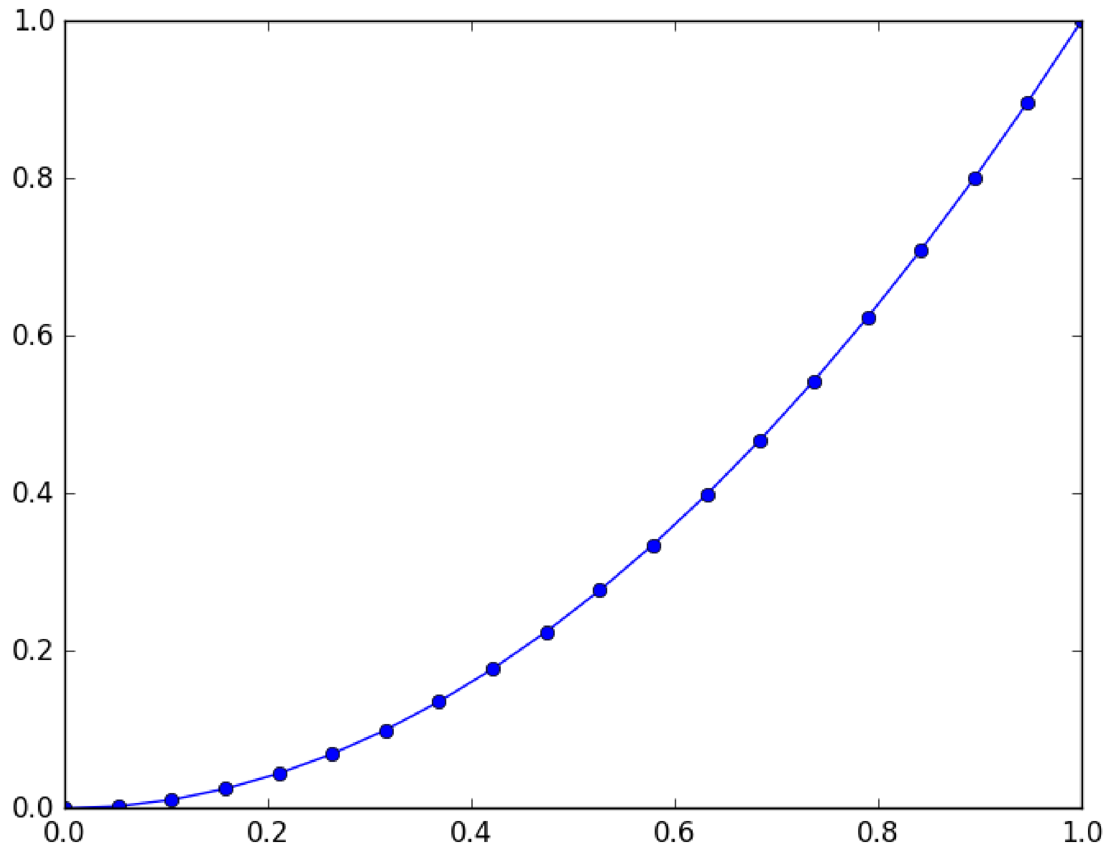
```
In [24]: exp.(sin.(x))
```

```
Out[24]: 20-element Array{Float64,1}:
          2.14078
          2.23338
          1.78564
          1.78377
          1.47757
          1.95369
          1.52463
          1.05066
          2.23938
          1.30135
          1.43563
          1.42475
          1.61344
          1.18877
          2.25407
          1.11463
          1.29644
          1.07978
          2.0857
          1.78916
```

1.1 Plotting

```
In [27]: using PyPlot # thin wrapper around Python Matplotlib
```

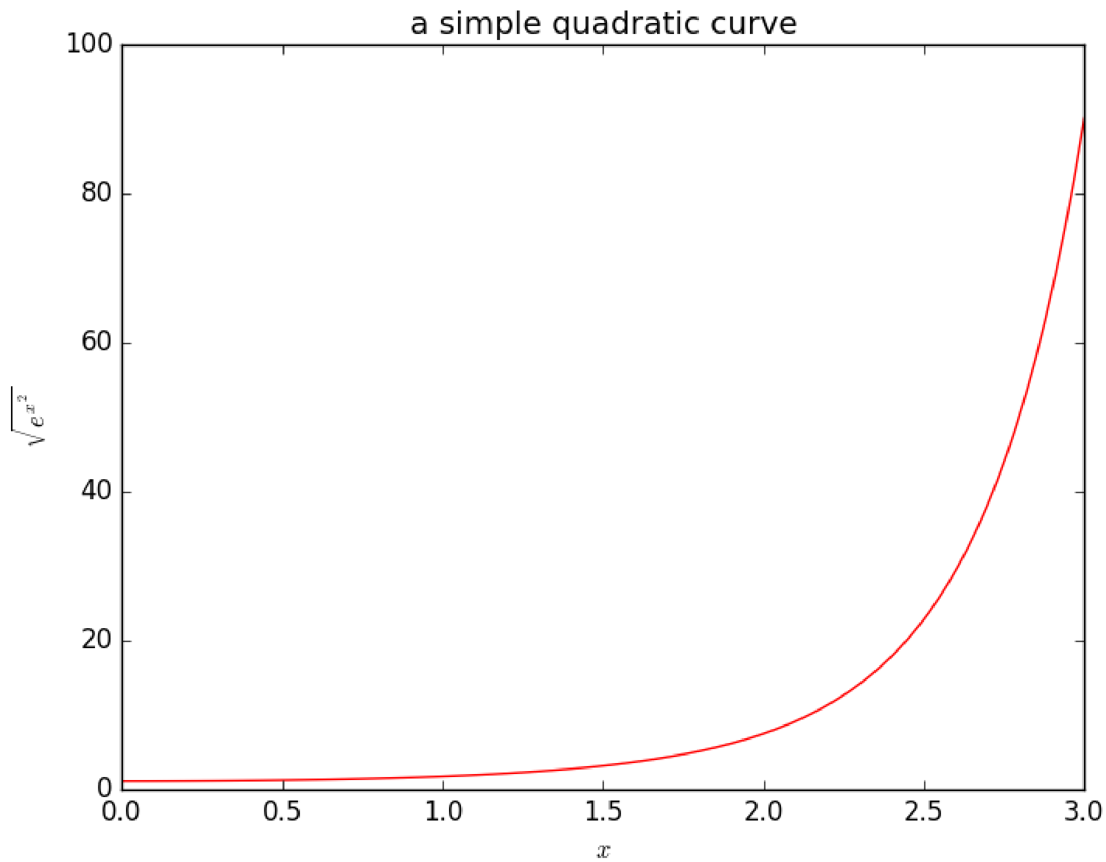
```
In [29]: x = linspace(0,1,20)
         plot(x, x.*x, "bo-")
```




```
Out[29]: 1-element Array{Any,1}:
          PyObject <matplotlib.lines.Line2D object at 0x322f17250>
```

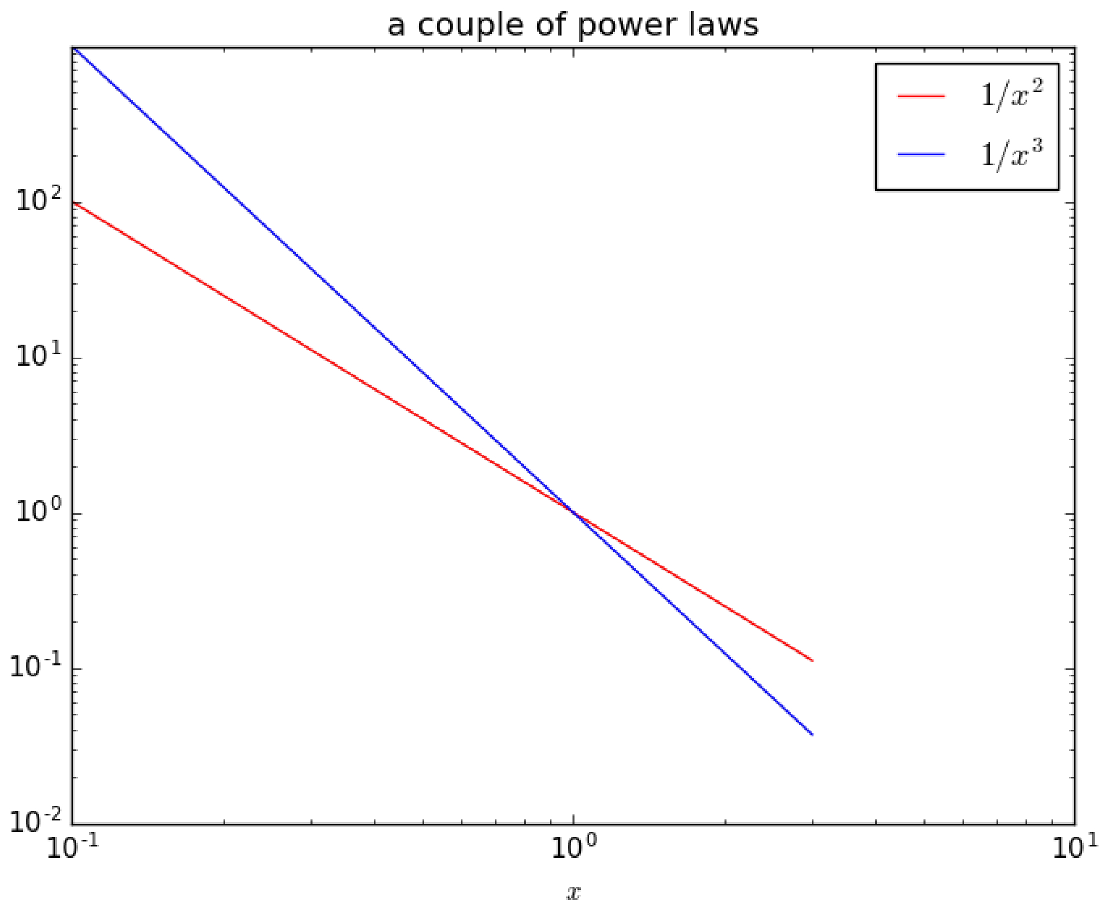
Let's plot a more complicated function, $\sqrt{e^{x^2}} = e^{x^2/2}$:

```
In [30]: x = linspace(0, 3, 100)
         plot(x, sqrt(exp(x.*x)), "r-")
         title("a simple quadratic curve")
         xlabel(L"x")
         ylabel(L"\sqrt{e^{x^2}}")
```




```
Out[30]: PyObject <matplotlib.text.Text object at 0x322f615d0>
```

```
In [31]: x = linspace(0.1, 3, 100)
         loglog(x, x.^(-2), "r-")
         loglog(x, x.^(-3), "b-")
         title("a couple of power laws")
         xlabel(L"x")
         legend([L"1/x^2", L"1/x^3"])
```




```
Out[31]: PyObject <matplotlib.legend.Legend object at 0x323328610>
```

1.2 Linear algebra

```
In [32]: A = [1 2 3 4; 5 6 7 8; 9 10 11 12]
```

```
Out[32]: 3×4 Array{Int64,2}:
          1   2   3   4
          5   6   7   8
          9  10  11  12
```

```
In [33]: A = [1 2 3 4
              5 6 7 8
              9 10 11 12]
```

```
Out[33]: 3×4 Array{Int64,2}:
          1   2   3   4
          5   6   7   8
          9  10  11  12
```

```
In [34]: A = rand(4,4)
```

```
Out[34]: 4×4 Array{Float64,2}:
 0.354297  0.796286  0.817323  0.0345978
 0.690757  0.545312  0.10516   0.960872
 0.534802  0.541574  0.910064  0.547655
 0.246295  0.804683  0.217789  0.227035

In [35]: A = randn(1000,1000)

Out[35]: 1000×1000 Array{Float64,2}:
 -1.65081   -0.240849   0.303505   …  -1.7985      1.58347     2.17258
  1.1676    -1.14529   -0.79521        0.757942    0.136036   -0.202522
 -0.42534   -0.636201  -0.271092       0.636029    0.432697    0.982919
  0.548365   0.813392   0.180881       0.892248   -2.66757     0.623155
  0.358272  -1.32407   -1.62438       -0.583573    1.39367     1.84467
 -0.522505   0.635723  -1.83343    …   0.373494    0.880843    0.197541
 -1.36032    0.328761   0.966728       0.214797   -0.519872    0.786407
 -1.96225   -1.18058    0.0220994      1.03323     1.73717    -1.02543
 -2.3271     2.59105   -0.105579      -0.740164   -0.588027   -1.22532
  0.455888  -1.17448   -1.23745        1.26299    -0.996511   -0.0343119
 -1.37476   -0.695493  -0.349228   …   0.623711   -0.118538   -2.10232
  0.385374   0.500938   0.328916       1.69133     0.0510919  -1.33315
  0.367484  -0.88171   -0.0158286     -0.0158428   0.0727947   0.330221
  ⋮                                ⋱
  2.07491    0.301412  -2.88188       -2.13433    -1.46011    -1.33492
  0.445576   0.200213   0.867564       2.16332    -0.607085   -0.719657
  1.15989    1.38095    0.0995334  …   0.656682   -0.286683    1.1383
  1.49364    1.61671    1.74414        2.3082     -0.608303   -0.857497
  1.00611   -0.49429   -0.999569      -1.72488     0.0250074   1.70796
 -0.653577  -0.867363  -1.2563        -1.27231     0.629835    0.731937
  0.381445   0.974918  -0.0255441     -0.824598   -1.34282     0.00359473
  0.695848   0.0786676  1.72538    …  -0.0131363  -0.859186   -0.483053
 -0.775784  -1.15708   -0.238277       0.658991   -0.40027     1.3225
  0.733769   0.673854  -1.31093        0.176824   -1.0881     -1.46115
 -0.240597   0.142289   0.265261       0.365702   -1.2539      0.0856285
 -1.05008   -0.707914   0.447666       0.743693    0.61162     1.02591

In [36]: b = randn(1000)

Out[36]: 1000-element Array{Float64,1}:
  0.704118
 -0.096427
 -0.733399
 -0.343855
  1.94833
 -0.712985
  0.37097
  0.438342
 -0.486407
  0.772552
  0.370966
 -1.60214
  0.0380739
  ⋮
 -1.5625
```

```
  0.67892
 -0.305153
  0.872263
  1.37475
 -0.0145622
  1.55491
 -2.12784
 -1.57598
  0.625179
  1.07015
 -0.840333
```

Let's solve $Ax = b$:

```
In [37]: x = inv(A) * b
```

```
Out[37]: 1000-element Array{Float64,1}:
  0.582043
  1.18294
 -0.045209
 -0.671191
  1.51479
 -2.11266
 -0.849299
  1.68512
  0.101655
  0.228883
 -0.619881
 -0.755786
 -4.20192
  ⋮
 -1.57702
  3.67938
 -1.59842
  2.11529
 -2.26215
 -0.348302
 -2.48251
  2.41252
  2.1558
 -2.08519
 -1.97143
 -1.54951
```

```
In [38]: x = A \ b
```

```
Out[38]: 1000-element Array{Float64,1}:
  0.582043
  1.18294
 -0.045209
 -0.671191
  1.51479
 -2.11266
 -0.849299
  1.68512
```

```
  0.101655
  0.228883
 -0.619881
 -0.755786
 -4.20192
  ⋮
 -1.57702
  3.67938
 -1.59842
  2.11529
 -2.26215
 -0.348302
 -2.48251
  2.41252
  2.1558
 -2.08519
 -1.97143
 -1.54951
```

```
In [39]: λ = eigvals(A)
```

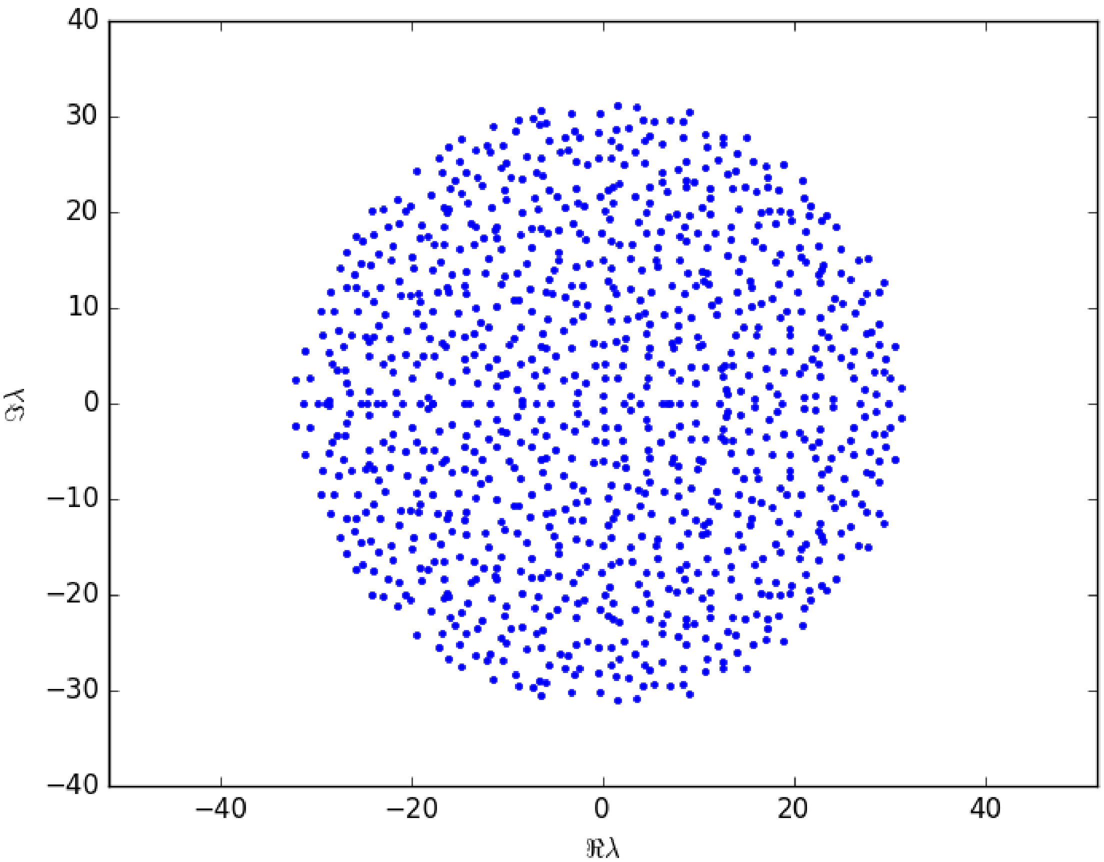
```
Out[39]: 1000-element Array{Complex{Float64},1}:
   23.8373+20.5315im
   23.8373-20.5315im
   16.9749+26.496im
   16.9749-26.496im
   16.5081+26.5372im
   16.5081-26.5372im
   31.0478+5.66555im
   31.0478-5.66555im
   29.4471+11.4433im
   29.4471-11.4433im
   28.0514+14.2995im
   28.0514-14.2995im
    30.288+6.98167im
          ⋮
 -0.353185-3.43256im
    3.2686+1.52675im
    3.2686-1.52675im
   3.89872+0.0im
   1.85608+2.95923im
   1.85608-2.95923im
 -0.532069+2.10994im
 -0.532069-2.10994im
    1.4669+1.1996im
    1.4669-1.1996im
   -1.4334+0.0im
  0.659698+0.0im
```

You may be wondering how to type λ. It turns out you can just type `\lambda` (the LaTeX code for λ) and then hit *tab*. You can type much more complicated variable names this way. (This is all thanks to Julia's support for something called Unicode.)

```
In [50]: x̂₂′ = 7 # x\hat<tab>\_2<tab>\prime<tab>
```

```
Out[50]: 7

In [40]: A = randn(1000,1000)
         λ = eigvals(A)
         plot(real(λ), imag(λ), "b.")
         axis("equal")
         xlabel(L"\Re \lambda")
         ylabel(L"\Im \lambda")
         savefig("foo.pdf")
```




```
In [41]: A = rand(5,5)

Out[41]: 5×5 Array{Float64,2}:
          0.654851   0.870059  0.0121391  0.28378    0.281063
          0.185478   0.835295  0.921174   0.211865   0.792001
          0.402456   0.174917  0.385094   0.104552   0.449979
          0.0853418  0.462815  0.404998   0.0762736  0.566363
          0.501122   0.769946  0.985798   0.533097   0.331967

In [42]: λ, X = eig(A)

Out[42]: (Complex{Float64}[2.33956+0.0im,0.202439+0.471556im,0.202439-0.471556im,-0.504527+0.0im,0.0435
         Complex{Float64}[0.439548+0.0im 0.731467+0.0im ... -0.0449559+0.0im -0.283944+0.0im; 0.564317+
```

In [43]: λ

Out[43]: 5-element Array{Complex{Float64},1}:
```
   2.33956+0.0im
  0.202439+0.471556im
  0.202439-0.471556im
 -0.504527+0.0im
 0.0435738+0.0im
```

In [44]: X

Out[44]: 5×5 Array{Complex{Float64},2}:
```
 0.439548+0.0im    0.731467+0.0im       ...  -0.0449559+0.0im  -0.283944+0.0im
 0.564317+0.0im   -0.317749+0.340296im        0.184355+0.0im  -0.177871+0.0im
  0.28513+0.0im  -0.0230972-0.413087im        0.349483+0.0im  -0.268076+0.0im
  0.32097+0.0im   -0.134243+0.232145im         0.41431+0.0im   0.840298+0.0im
 0.551383+0.0im  -0.0572381-0.042745im       -0.818658+0.0im   0.331318+0.0im
```

In [45]: λ, X

Out[45]:
```
(Complex{Float64}[2.33956+0.0im,0.202439+0.471556im,0.202439-0.471556im,-0.504527+0.0im,0.04357
Complex{Float64}[0.439548+0.0im 0.731467+0.0im ... -0.0449559+0.0im -0.283944+0.0im; 0.564317+0
```

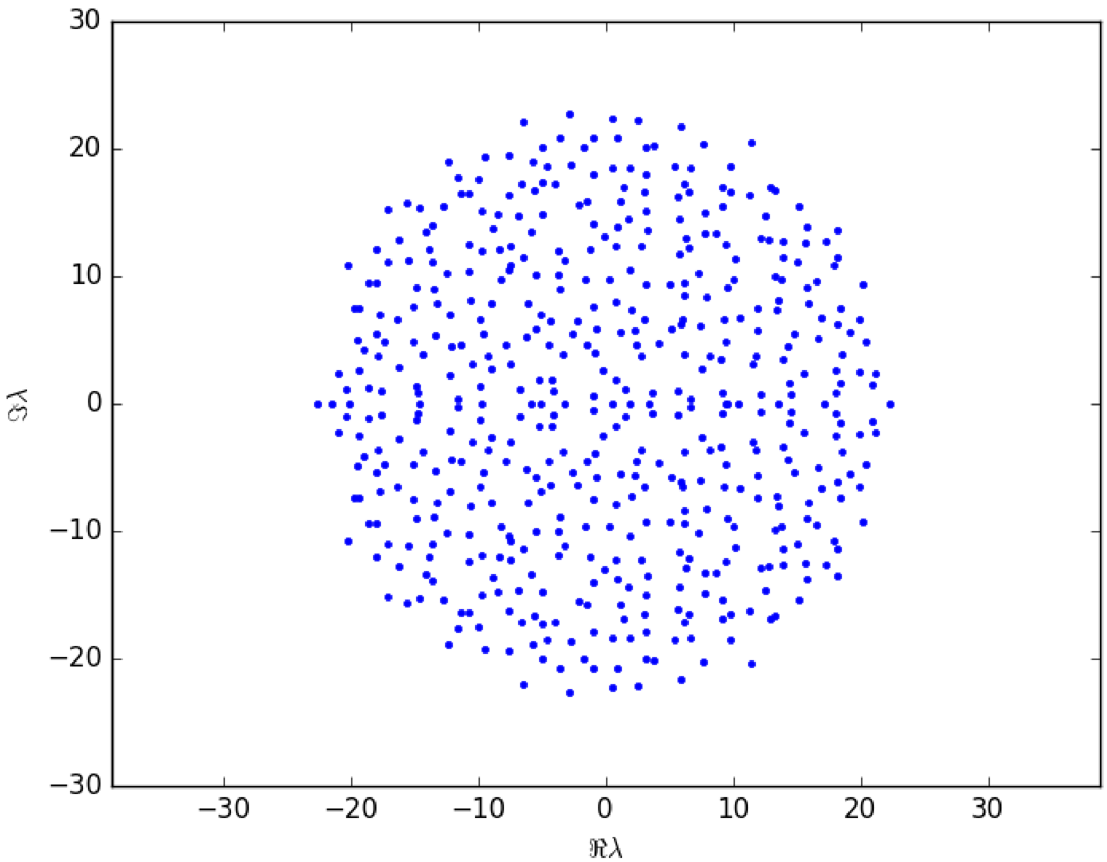
In [46]: using Interact

INFO: Precompiling module DataStructures.

In [47]:
```julia
f = figure()
@manipulate for n in 10:1000
    withfig(f) do
        A = randn(n,n)
        λ = eigvals(A)
        plot(real(λ), imag(λ), "b.")
        axis("equal")
        xlabel(L"\Re \lambda")
        ylabel(L"\Im \lambda")
    end
end
```

```
Interact.Options{:SelectionSlider,Int64}(Signal{Int64}(505, nactions=1),"n",505,"505",Interact.OptionDi
```

Out[47]:



1.3 Calling Python

Being a young language, Julia doesn't have as many mature libraries and packages as a language like Python. Fortunately, Julia code can *call Python libraries* directly, with the help of the PyCall package:

```
In [48]: using PyCall

In [49]: @pyimport scipy.special as special

In [50]: special.airy(3)

Out[50]: (0.006591139357460717,-0.011912976705951313,14.037328963730229,22.92221496638217)

In [51]: @pyimport scipy.optimize as opt

In [52]: opt.newton(cos, 1.4) - pi/2

Out[52]: 0.0

In [53]: opt.newton(x -> cos(x) - x, 1.4)

Out[53]: 0.7390851332151607
```