A Short and Incomplete Introduction to Julia

Part 4: Plotting, arrays, algebra, exceptions

Riccardo Murri <riccardo.murri@uzh.ch>
S3IT: Services and Support for Science IT,
University of Zurich

September 12, 2019

Plotting

Plots is the most-used plotting library in the Julia community.

The most common options are summarized in this cheatsheet by Samuel S. Watson.

The official documentation is available at:
http://docs.juliaplots.org/latest/learning/

Preparing for plotting

Readying `Plots` for use is a two-step process:

1. Import the `Plots` module:

```
julia>   using Plots
```

2. Select a visualization backend:
   - GR, a fast rendering backend:

```
julia>   gr()   # default backend
Plots.GRBackend()
```

   - PlotlyJS, interactive visualizations (in the browser or GUI) based on Plotly.js:

```
julia>   plotlyjs()   # better for notebooks
Plots.PlotlyJSBackend()
```

   - (Other backends are available, see Plots' backends manual page.)

Making plots

The `plot()` function is all you need to create plots:

- Provide *x*-axis and *y*-axis coordinates of points;
- Specify how you want these drawn on the canvas.

```
julia>   plot(xs, ys, typ=:hline)
```

Note that:

- `xs` must be *sorted*!
- `xs` and `ys` must have the same length.

Making plots

The `plot()` function is all you need to create plots:

- Provide x-axis and y-axis coordinates of points;
- Specify how you want these drawn on the canvas.

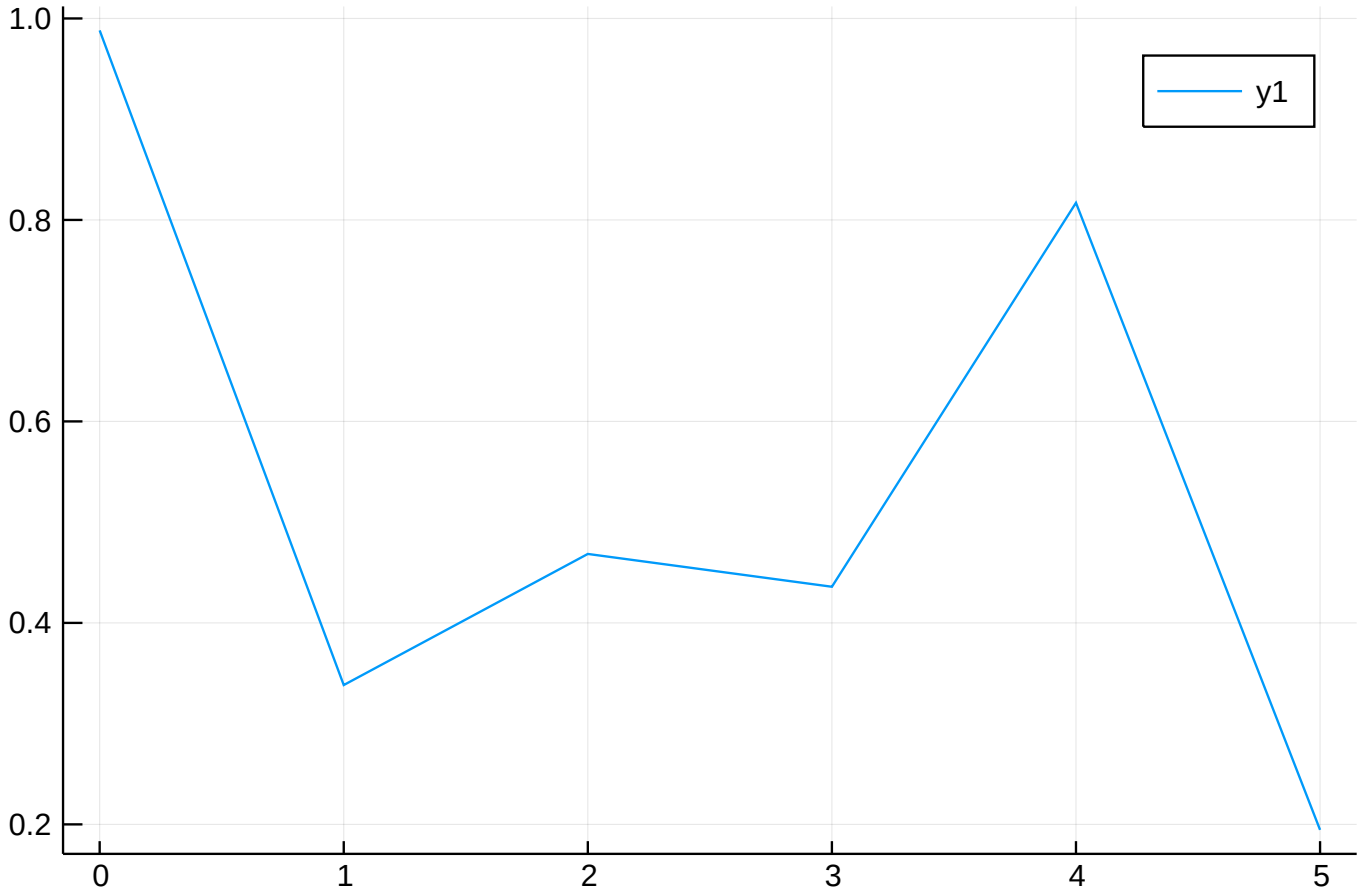
```
julia>   plot(xs, ys, typ=:line)
```

Note that:

- `xs` must be *sorted*!
- `xs` and `ys` must have the same length.

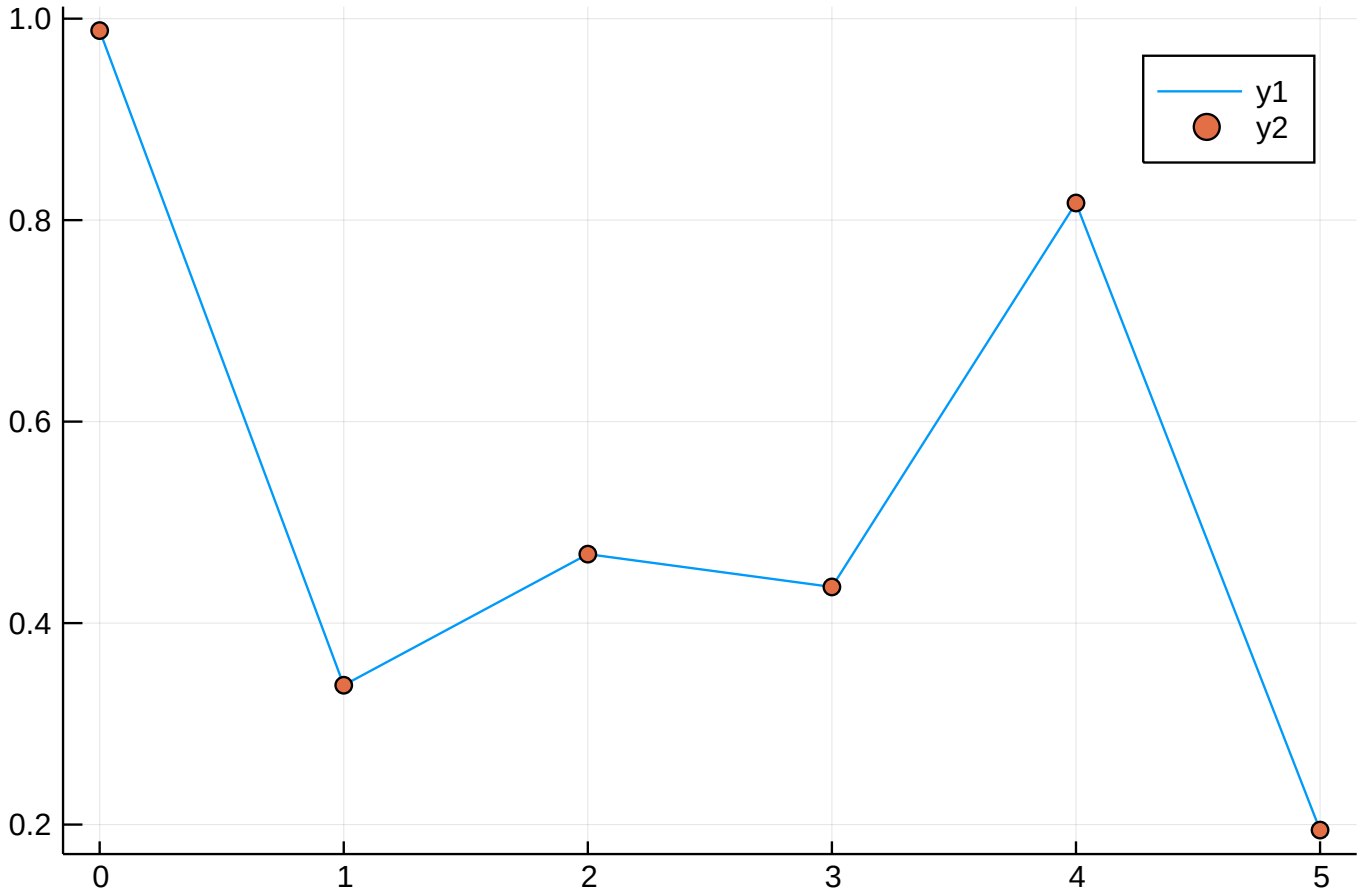
Starting plots

```
julia>    xs = 0:5;
julia>    ys = rand(length(xs));
julia>    plot(xs, ys)
```



Overlaying plots, I

```
julia>    xs = 0:5;
julia>    ys = rand(length(xs));
julia>    plot(xs, ys)
julia>    plot!(xs, ys, typ=:scatter)
```



Overlaying plots, II

The `plot()` function creates a *new* plot;
the `plot!()` variant *adds* to an existing one.

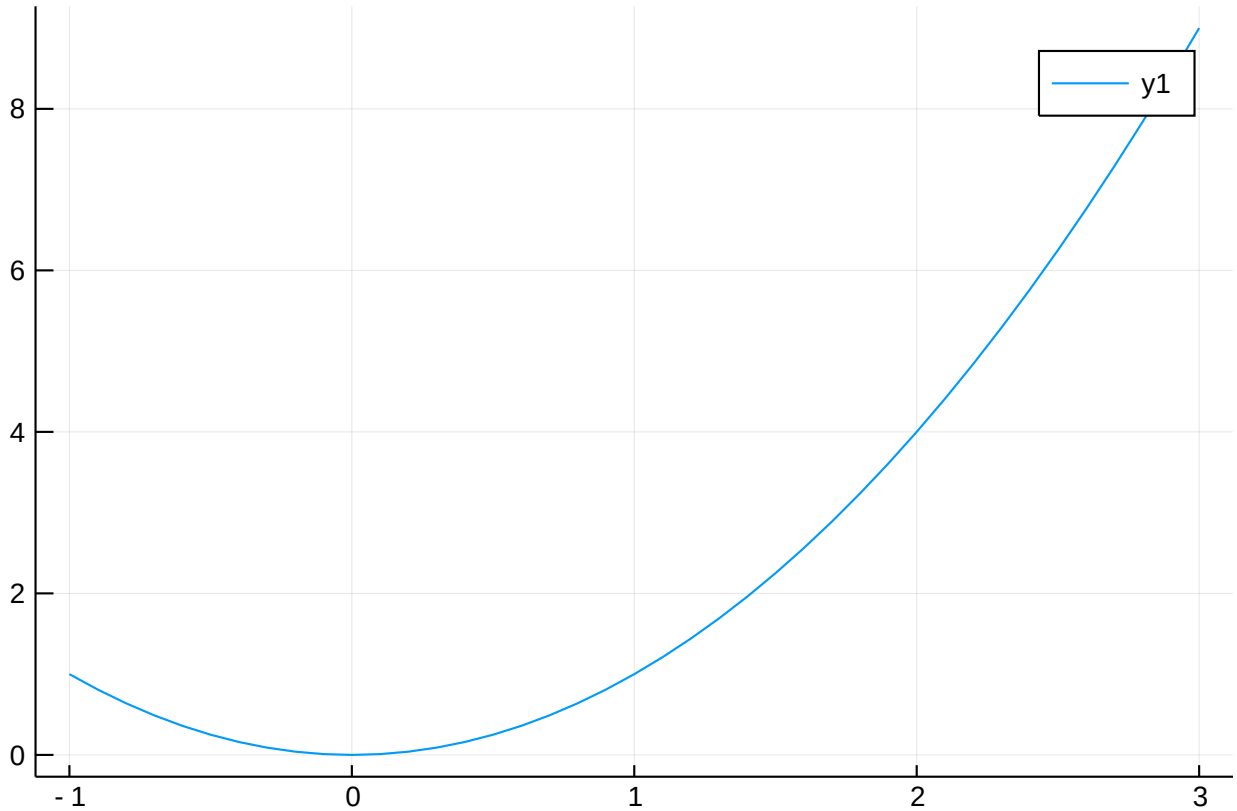
You can also save a plot object to modify it later:

```
p = plot(xs, ys)
# ...
plot!(p, ...)
```

A number of functions are available to modify a plot's **attributes**: `title!`, `xlims!`, `xlabel!`, `xticks!`, `ylims!`, `ylabel!`, etc.

Plotting functions, I

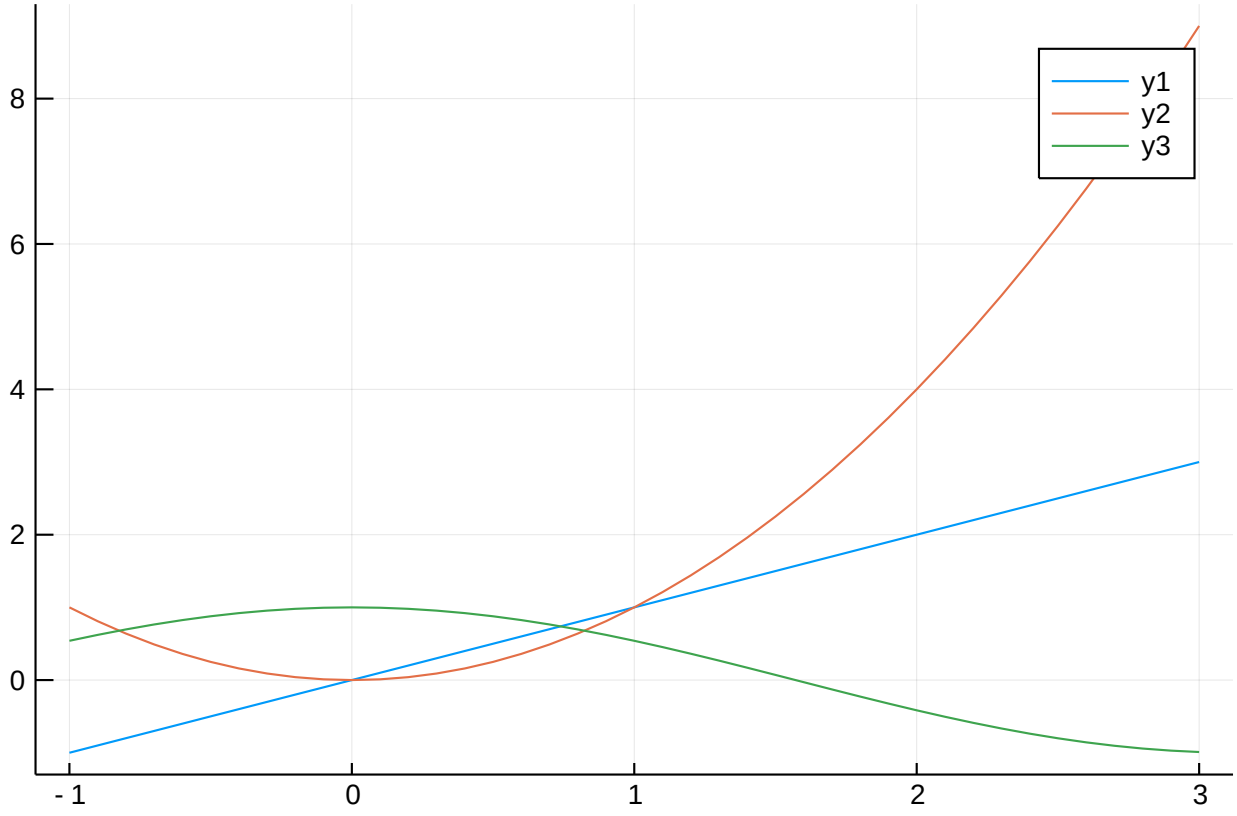
```
julia>    xs = -1:0.1:3; # from -1 to 3 in steps of 0.1
julia>    plot(xs, x->x^2 )
```



The y-axis coordinates can be a *function* to compute the values from the x's . . .

Plotting functions, II

```
julia>    xs = -1:0.1:3;
julia>    plot(xs, [x->x, x->x^2, x->cos(x)])
```



. . . or a collection of functions!

Exercise 4.A: Plot the graph of function $f(x) = 1/x$. Can you make the line 2 pixels thick? Can you make the plot line red?

Exercise 4.B: Plot the functions $sin(x)$ and $2 \cdot cos(x) - 1$ on the interval $[-\pi, +\pi]$.
How can you assign different attributes (e.g., line color) to the two lines? For instance, color the *sin* line red and the *cos* line blue.

Exercise 4.C: Set `xs=-8:0.01:+8`; use `map` to collect the values of functions `sin(x)` and `cos(x)` into vectors `ys` and `zs`, respectively.
Now evaluate `plot(xs, ys, zs)` — what do you see? Does it work the same if you use the `sin` and `cos` functions instead of arrays `ys` and `zs`?

Ranges, I

The notation $a:\delta:b$ denotes the collection of numbers a, $a+\delta$, $a+2\delta$, $\ldots$, b (inclusive).

When $\delta = 1$, this can be abbreviated $a:b$.

Note that b will be adjusted (towards 0) if $b-a$ is not an integral multiple of δ:

```
julia>   0.5:0.25:2.3
0.5:0.25:2.25
```

The `collect()` function can be used to turn a range into a vector:

```
julia>   collect(0.5:0.25:2.3)
8-element ArrayFloat64,1:
 0.5
 0.75
 1.0
 ...
```

Ranges, II

The `range()` function allows a bit more flexibility:

```
julia>   range(1, stop=5) # same as 1:5
1:5
julia>   range(1, stop=5, step=0.2) # same as 1:0.25:2.
1.0:0.2:5.0
```

`range()` allows specifying `stop=` and `step=` parameters in any order:

```
julia>   range(0.5, step=0.25, stop=2.3)
0.5:0.25:2.25
```

In addition, `range` allows automatically computing the step given a number of subdivisions:

```
julia>   range(-1.0, stop=1.0, length=120)
-1.0:0.016806722689075631:1.0
```

Elementwise operations and Broadcasting

Elementwise operations, I

For every binary operation like `*`, there is a corresponding "dot" operation `.*` that is automatically defined to perform `*` element-by-element on arrays:

```
julia>    v = [1,2,3];
julia>    v .* v   # multiply elementwise
3-element Array{Int64,1}:
 1
 4
 9
```

Reference: https://docs.julialang.org/en/v1/manual/mathematical-operations/#man-dot-operators-1

Elementwise operations, II

There is a corresponding "dot" notation for functions; note that the dot '.' goes *after* the function name:

```
julia>    xs = [0,1,2,3,5];
julia>    ys = cos.(xs)
5-element Array{Float64,1}:
  1.0
  0.5403023058681398
 -0.4161468365471424
 -0.9899924966004454
  0.28366218546322625
```

There are no spaces between the function name (e.g., `cos`) and the dot character.

The dot notation can be applied to user-defined and built-in functions alike.

Reference: https://docs.julialang.org/en/v1/manual/mathematical-operations/#man-dot-operators-1

Elementwise operations, III

A string of elementwise operations has a performance advantage: the Julia compiler can "fuse" multiple operations and make a single loop over the involved arrays, avoiding the creation of intermediate temporary storage places.

An expression like:

```
ys = 2 .* cos.(xs) .- 1
```

is automatically unrolled into:

```
temp = similar(xs)
for i in 1:length(xs)
  temp[i] = 2*cos(xs[i]) - 1
end
ys = temp
```

Reference:
https://docs.julialang.org/en/v1/manual/functions/#man-vectorized-1

Elementwise operations, IV

When combining multiple "dot" operations in a row, it is possible to use the `@.` macro to save typing and add readability.

This expression:

```
ys = 2 .* cos.(xs) .- 1
```

can be rewritten as:

```
ys = @. 2 * cos(xs) - 1
```

Reference: https://docs.julialang.org/en/v1/manual/mathematical-operations/#man-dot-operators-1

Elementwise operations, V

The "dotted assignment" operator `.=` implements **in-place update** of an array: the array being assigned to is destructively replaced by values of the expression on the right-hand side.

For example, expression:

```
ys .= @. 2cos(xs)-1
```

is internally rewritten as:

```
for i in 1:length(xs)
  ys[i] = 2*cos(xs[i]) - 1
end
```

The macro `@allocated` allows checking how much memory is used for computing an expression:

```
julia>   @allocated ys1 = @. 2cos(xs)-1
8000560
julia>   @allocated ys2 .= @. 2cos(x)-1
480
```

Exercise 4.D*: Define three functions for computing the dot-product* of vectors:

- `dotprod1`: Use a `for`-loop and accumulate results;
- `dotprod2`: Use `.*` and the built-in function `sum()`;
- `dotprod3`: Use function `dot` from the `LinearAlgebra` module.

Collect execution times (with the `@elapsed` macro) on a random vector of size 10^6; how do they fare?

Now collect allocation size (with the `@allocated` macro) on the same random vector; how does this compare to the running times?

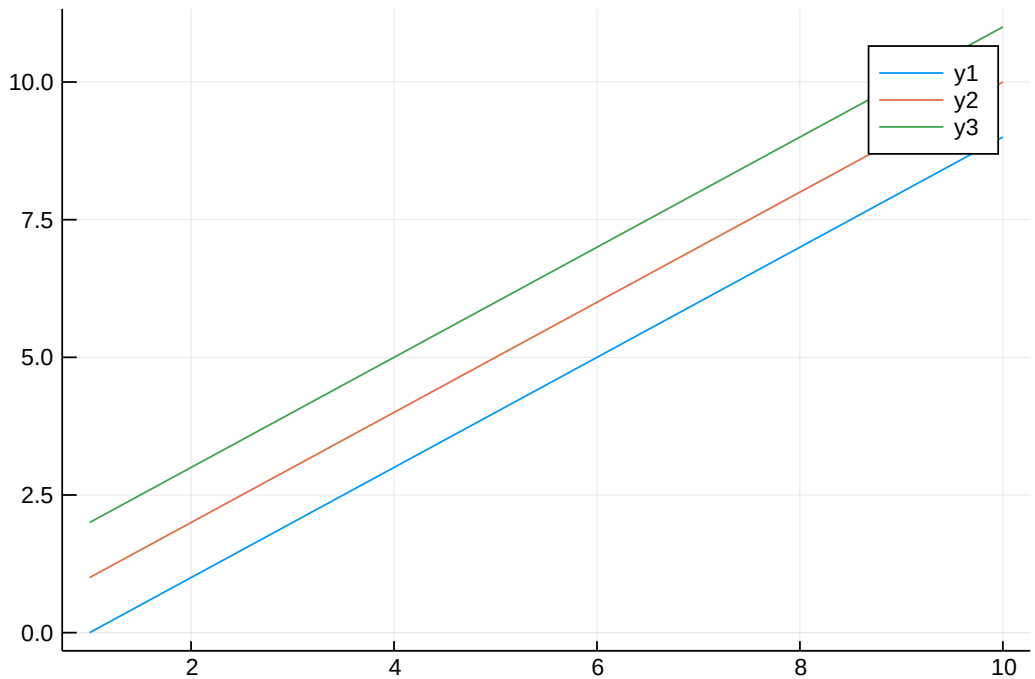
(To make things more interesting, you may want to collect times and allocation sizes on vectors of varying length and plot that data.)

*In the linear algebraic sense.

Broadcasting, I

"Dotted" operators allow mixing
scalars and arrays of different sizes:

```
julia>    xs = 1:10;
julia>    ys = collect(1:10); # range->vector
julia>    plot(xs, [ys .- 1, ys, ys .+ 1])
```



Broadcasting, II

When arrays of different sizes are mixed, Julia:

- replicates array values along missing axes,
- expands singleton dimensions in array arguments

to match the other array:

```
julia>   v = [1,2,3]  # column vector
3-element Array{Int64,1}:
 1
 2
 3
julia>   m = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9
julia>   m .+ v  # v[i] added to row i
3×3 Array{Int64,2}:
  2   3   4
  6   7   8
 10  11  12
```

Broadcasting, III

When arrays of different sizes are mixed, Julia:

- replicates array values along missing axes,
- expands singleton dimensions in array arguments

to match the other array:

```
julia>    v'   # transpose -> row vector
1×3 LinearAlgebra.AdjointInt64,ArrayInt64,1:
 1  2  3
julia>    m  .*  v'   # v[i] muliplies column i
3×3 ArrayInt64,2:
 1   4   9
 4  10  18
 7  16  27
```

Exercise 4.E: Write a function `zscore(xs)` which, given a array `xs` of numbers, returns a similar array where every element has been normalized ($x \mapsto (x - \mu)/\sigma$) so that the result has mean 0 and std deviation 1.

Indexing

Array indexing

When getting or setting array elements `m[i, j, k, ...]` each of the indices can be:

- a single scalar value — selects only items whose corresponding coordinate is exactly that value;
- a range $a:b$ — selects items whose coordinate falls in the range;
- a strided range $a:s:b$ — selects items whose coordinate is one of a, $a+s$, $\ldots$, b;
- a boolean 1D array — selects those items whose coordinate has the value `true` in the array.

Array indexing

Array slices can be used for getting or setting:

```
julia>   using Statistics
julia>   v = randn(100);
julia>   q = quantile(v, 0.90);
julia>   i = (v .> q)  # need dotted comparison
100-element BitArray{1}:
 0
 0
 0
 1
...
julia>   w = v[i]  # elements > q
10-element Array{Float64,1}:
 1.5486428522593445
 1.459431316814836
...
```

(This could be shortened as `w = v[v .> q]`)

Array indexing

Array slices can be used for getting or setting:

```
julia>    m = zeros(3, 3);
julia>    m[ :, 2 ] = ones(3);
julia>    m
3×3 Array{Float64,2}:
 0.0   1.0   0.0
 0.0   1.0   0.0
 0.0   1.0   0.0
```

```
julia>    m = zeros(3, 3);
julia>    m[ 1, : ] = randn(3);
julia>    m
3×3 Array{Float64,2}:
 0.5467 -0.0869 -1.2040
 0.0     0.0     0.0
 0.0     0.0     0.0
```

Array indexing

When assigning from a scalar value, or an array that does not match the shape of the one being assigned to, the dotted '.=' *must* be used:

```
julia>   checkers = [isodd(i+j)
                     for i in 1:3, j in 1:3];
julia>   m = zeros(Int, (3,3));
julia>   m[checkers] .= 1;
julia>   m
3×3 Array{Int64,2}:
 0  1  0
 1  0  1
 0  1  0
```

Exercise 4.F: Write a function `cutoff!(a, q)` that given an array `a` and a percentage `q`, modifies `a` by setting all entries that are larger than the `q`-th percentile to exactly that value. (You need to import package `Statistics` to access the `quantile()` function.)

Linear Algebra

Linear Algebra

The standard library `LinearAlgebra` provides:

- functions for calculations with vectors and matrices,
- matrix factorization (LU, DU, QR, etc.)
- specialized types for storing and computing with special matrices

Reference: https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/index.html

Linear Algebra

Matrix-vector and matrix-matrix products are built-in the Julia core:

```
julia>   v = [1,2,3];
julia>   m
3×3 Array{Int64,2}:
 0  1  0
 1  0  1
 0  1  0
julia>   m*v
3-element Array{Int64,1}:
 2
 4
 2
```

```
julia>   m*m
3×3 Array{Int64,2}:
 1  0  1
 0  2  0
 1  0  1
```

```
julia>   m^2 + m
3×3 Array{Int64,2}:
 1  1  1
 1  2  1
 1  1  1
```

Linear Algebra

Other functions must be imported from `LinearAlgebra`:

`dot(v,w)`, `cross(v,w)`
Dot- and cross-product of two vectors.
(Also available in operator form `v⋅ w` and `v × w`.)

`tr(m)`, `det(m)`, `inv(m)`
Trace, determinant, and inverse of matrix `m`

`eigvals(m)`, `eigvecs(m)`
Eigenvalues and Eigenvectors of matrix `m`

`factorize(m)`, `lu(m)`, `cholesky(m)`, etc.
Compute factorizations of matrix `m`. Function `factorize(m)` is the generic front-end that will compute a convenient factorization of `m` based on the actual "shape".

Solving linear systems, I

The linear problem $Ax = b$ is solved by Julia's operator '\':

```
julia>    a = randn((3,3));
julia>    v = [1,2,3];
julia>    x = a \ v
3-element Array{Float64,1}:
  1.3909476657572728
 -3.398174579781128
  0.6163698662831701

julia>    a*x == v   # verify
true
```

If matrix `a` is not square, Julia's '`a\v`' returns the (minimum norm) least squares solution.

Solving linear systems, II

Still, if a square matrix is not full-rank then an error is thrown:

```
julia>    m = [0 1 0; 1 0 1; 0 1 0];
julia>    v = [1,2,3];
julia>    m\v
ERROR: SingularException(3)
```

Errors and exceptions

Exceptions

"Exceptions" is the name given in to error conditions that cause a function to abort execution and signal the condition to the caller.

If the caller cannot handle it, the exception is propagated up the call stack until a handler is found or the program exits.

You can write code that intercepts some error conditions and reacts appropriately.

See also:
https://scls.gitbooks.io/ljthw/content/_chapters/11-ex8.html

Handling exceptions

```
try
  # code that might raise an exception
catch err
  # handle some exception
finally
  # performed on exit in any case
end
```

After a `catch` keyword, an optional name indicates a local variable that will hold the exception value — code can inspect that to determine if and how to handle the exception.

The optional **`finally`** clause is executed on exit from the **`try`** or **`catch`** block in *any* case.

Reference: https://docs.julialang.org/en/v1/base/base/#try

Handling exceptions, II

Julia's `try/catch` mechanism does not discriminate on the type of exception value thrown; it is completely up to the user to do that:

```
try
  x = m\v;
catch err
  if isa(err, SingularException)
    @warn "Linear system is singular!"
    x = nothing;
  else
    # some other error happened, cannot handle
    rethrow() # propagate exception
  end
end
```

The `rethrow` function, if called in a `catch` block, throws the current exception object, keeping the original throw location.

Throwing exceptions

You can throw exceptions from your own code with these functions:

`error(msg)`
Raise an ErrorException with the given message.

`throw(err)`
Throw a value as an exception. Note that Julia does not enforce a specific type for exceptions — any value will do!

`rethrow()`
Throw an exception, without changing the backtrace (i.e., keeping the origin location intact). If within a `catch` block, the exception value can be omitted and defaults to the exception being handled.

Reference: https://docs.julialang.org/en/v1/base/base/#Base.error