The Garden of Arrays

1. Introduction

Most modern programming languages (and many older languages) have a concept called an array (Python calls theses lists though this terminology is mixed with the concept of a linked list so should be avoided).

An *array* in programming stems originally from the mathematical idea of a vector or matrix, which is an "array" of numbers in a structured object with either a length (a vector) or length and breadth (a matrix). The concept is easily generalised in many ways.

At its root the idea is to have a collection of objects accessed through an index. In the simple vector case, the index is just a non-negative integer. The decision as to start the indices at 0 or 1 was never standardised either in math or in computer programming.

> *"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration." — Stan Kelly-Bootle*

A vector is indexed by single integer and a matrix is indexed by a pair of integers, and the idea extends to arbitrary dimensions (tensors).

However, the idea of an array extends both to the case where the objects in the array – the values – are no longer just numbers, and to the case where the array indexes are not integers. Both of these cases introduce nuances to the implementation of arrays in programming languages.

At the deepest level we can start to think of arrays more like a mapping from one set to another. The mapping is one-to-one or many-to-one. The issues and complexities of this vision mean it might be worth a few words of explanation. But this is not a primer on arrays in Julia. If you want that, there are many, *e.g.,* see:

- https://docs.julialang.org/en/v1/base/arrays/
- https://docs.julialang.org/en/v1/manual/arrays/
- https://en.wikibooks.org/wiki/Introducing_Julia/Arrays_and_tuples
- https://www.geeksforgeeks.org/arrays-in-julia/
- https://www.softcover.io/read/7b8eb7d0/juliabook/basics

This document is more in the line of a collection – a botanical garden if you like – describing the variations on arrays and why you should care, in the programming language Julia. The idea is to provide insights into the large number of array types in Julia in particular the motivation and use cases for each type, with some idea of how they are implemented where that is important to understand the use case.

2. Types of Array in Julia: Motivation, and Implementation

2.1 Core Types of Arrays

The Standard Integer-Indexed Array

Most modern programming languages (and many older languages) have a concept called an array (or sometimes a list though this terminology isn't mixed with the concept of a linked list so should be avoided). The standard array is a block of memory large enough to house a collection of objects, often some type of number.

Motivation: The standard array is such a common concept that you hardly ever see a justification for it anymore. The simplest justification comes from maths where vectors, matrices and tensors are used to hold structured sets of numbers for various applications: linear algebra, optimisation, fluid dynamics and so on. However, they are useful in many other ways. It is all too common in programming to see variables like x_1 and x_2, which are two instances of the same type. Holding these in an array allows all the advantages of having the two variables, but makes the "index" that chooses between them accessible to the program logic and thus provides flexibility about which would be accessed that otherwise would be hard-coded into the program. It also makes the program more extensible – it is easier to add $x[3]$ than x_3.

The key feature that distinguishes the standard array from some of the other arrays we will discuss later is *fast linear indexing*. An array is, at its heart, a block of memory that is $n \times m$ in size, where the array has n elements of a type that has size m bytes. Therefore we can obtain a pointer to the address of the ith element by taking

$$p_i = p_0 + i \times m, \tag{1}$$

where p_0 is a pointer to the start of the array. These pointers are usually hidden in modern languages, but it's what happens under the hood when you access $x[i]$.

Implementation in Julia: Julia's implementation of arrays is defined primarily by the type `Array`. Julia's implementation (like many modern implementations) is not quite as simple as that implied above. Arrays are implemented as a parameterised type, where the parameters are the type of the objects in the array and the dimension of the array. The types can be concrete or abstract, and these are (to me) very different beasts. If the type is concrete, then the size of the array is known at compile time, and hence Julia should be able to do all sorts of clever optimisations. If the type is abstract – in the most extreme case `Any` – then the objects in the array might not even be of the same type, so we lose the fundamental property of a standard array, *i.e.,* linear indexing.

Julia uses *column-major order* (see https://en.wikipedia.org/wiki/Row-_and_column-major_order, and https://docs.julialang.org/en/v1/manual/performance-tips/#man-performance-column-major) for matrices of 2 or more dimensions. This means that consecutive elements from the same column are stored in adjacent memory. That can result in performance improvements due to caching if the elements are accessed in order down columns and then across rows rather than the other way around. C and Numpy use row-major, and Fortran, R and Matlab use column-major ordering.

Julia does one extra trick. Intuitively an array uses the "right" amount of memory $n \times m$, but a common operation that programmers want to perform is to grow an array. In Julia we can add elements to the end of an array with `push!` and append an entire extra chunk with `append!`. In order to avoid frequent re-allocations of memory, Julia seems to use a type of *dynamic array* which pre-allocates extra memory when an array is created and reallocates it in chunks.

See: https://docs.julialang.org/en/v1/manual/arrays/#Implementation and https://github.com/JuliaLang/julia/issues/28588 and https://www.r-bloggers.com/2014/03/dynamic-arrays-in-r-python-and-julia/ for more details.

Notes: `Vector` and `Matrix` are aliases for the 1D and 2D `Array` types and `AbstractVecOrMat{T} = Union{AbstractVector{T}, AbstractMatrix{T}}`, that is, it is an alias of either a vector or matrix.

Julia has a great deal of syntactic sugar for working naturally with arrays. Array literals are specified with square brackets:

```
julia> x = [1, 2, 3]  # a (column) vector
3-element Vector{Int64}:
 1
 2
 3
```

```
julia> y = j[1; 2; 3]  # a (column) vector
3-element Vector{Int64}:
 1
 2
 3

julia> A = [1 2 3; 4 5 6] # 2D array
2×3 Matrix{Int64}:
 1  2  3
 4  5  6
```

We can also create complex arrays using a *comprehension*, which is specified, for instance as

```
julia> x = [ exp(i) for i=1:3]
3-element Vector{Float64}:
  2.718281828459045
  7.38905609893065
 20.085536923187668
```

And there is much more, but as we stated, this is not a tutorial on using arrays.

Types Directly Related to Standard Arrays

- `AbstractArray` is the abstract supertype intended to hold all of these arrays.
- `DenseArray` is an abstract subtype of `AbstractArray` holding the column-major ordered arrays.
- `SubArray` is a way to share a chunk of another array (rather than copy it) and is created by a `view`.
- `StridedArray` is a subtype of `AbstractArray` whose entries are stored with fixed "strides" between them.
- Images, for instance as created by `JuliaImages` are just 2D arrays (or 3D if they are colour): https://juliaimages.org/v0.21/quickstart/.
- `Diagonal` matrices are a special type of matrix with zeros everywhere except the diagonal entries. In essence, these are 2D matrices, but can be stored as a vector of diagonal entries. The `LinearAlgebra` package also introduces and number of other special matrices: `Symmetric`, `Hermitian`, `UpperTriangular` and so on: https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#Special-matrices.

Bit Arrays

Motivation: A `BitArray` (sometimes called a bit map or bit string) is a special case of the standard array intended to store Boolean values efficiently. The problem it addresses is that computer architectures store memory in chunks, sometimes called *words*. Sometimes machines cannot address memory at a smaller scale than a word (or at least a byte), so often data takes a minimum of one word (or byte) per variable. Modern computers have quite large words, so storing 1 bit per word (or even per byte) would be very inefficient. A bit array allows one to access the bits by storing them in groups and accessing the bits through bitwise operations.

Something similar is probably going on under the hood for arrays of smaller sized integers on some computer architectures, etc.

Implementation: Some wastage will happen with these if the number of bits to be stored is not a multiple of the word size.

https://en.wikipedia.org/wiki/Bit_array

Tuples

Motivation: A Tuple is essentially an *immutable* array. That means that once created, a tuple cannot be changed. In Julia a literal tuple is created with round brackets, *e.g.,* `t=(2,3,3)`. Julia uses them primary as a type to pass input and (multiple) output arguments to and from functions.

Implementation: I really don't know much about this ... yet.

Note there is a related type called a `NamedTuple`. I haven't ever needed these, so I am not really aware of their utility except maybe for keyword arguments for functions.

Sparse Arrays

Motivation: It is common in real applications that only a small subset of array indices correspond to non-zero elements, *i.e.,* the array is *sparse*. If an array is sparse there are much more efficient means to store and work with it than storing a solid block that would mostly be zeros. Julia has support for these through a standard module `SparseArrays`.

Implementation: The matrices are stored in Compressed Sparse Column (CSC) form. Essentially this is a set of triples giving the row and column address and the values of non-zero entries of the matrix in a struct

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <:
AbstractSparseMatrixCSC{Tv,Ti}
    m::Int                  # Number of rows
    n::Int                  # Number of columns
    colptr::Vector{Ti}      # Column j is in colptr[j]:(colptr[j+1]-1)
    rowval::Vector{Ti}      # Row indices of stored values
    nzval::Vector{Tv}       # Stored values, typically nonzeros
end
```

As with standard matrices the indices are integers, but the values can come from almost any type. There are additional functions to convert to sparse form: `sparse` and `dropzeros` or directly create sparse matrices `spzeros` and `sprand`. Functions such as `nnz` return the number of non-zero elements. Note that this formulation is intrinsically linked to a 2D matrix array (and there is another for vectors) so this is not as generic as the formulation of dense arrays. Once a sparse matrix is defined, however, you can often operate on it as if it were a dense matrix.

Types Directly Related to Sparse Arrays

There are many types, often through contributed packages, that relate to specific cases for sparse arrays. See https://juliapackages.com/c/sparse-matrices for a listing.

Strings

Motivation: A string is a list of characters. They are used to store text for many, many purposes.

Implementation: A string is *very roughly* a 1D array of characters. Actually a string in Julia is an array of byte-codes, that in combination can form Unicode characters. And note that despite this crude view, the supertype of `AbstractString` is `Any` not an array type. Essentially this is because a Unicode character is not a nice fixed-length block of data. For instance, elements of type `Char` are 4 bytes, but a string incorporating such a character can be smaller:

```
julia> x = 'a'
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
julia> sizeof(x)
4
```

```
julia> x = "a"
julia> sizeof(x)
1
julia> x = 'α'
'α': Unicode U+03B1 (category Ll: Letter, lowercase)
julia> sizeof(x)
4
julia> x = "α"
julia> sizeof(x)
2
```

The single quotes define a character (4 bytes) and the double quotes define a string whose length depends on the length of the Unicode representation of the characters in the string.

This can lead to some strange errors, for instance:

```
julia> x = "α is alpha"
julia> x[1]
'α': Unicode U+03B1 (category Ll: Letter, lowercase)

julia> x[2]
ERROR: StringIndexError: invalid index [2], valid nearby indices
[1]=>'α', [3]=>' '
```

That problem is caused because α is taking up two bytes, so $x[2]$ is not a valid character (it is the second byte of the 1st character). There are ways to avoid such errors but they go beyond the scope of this document.

Strings are immutable, so you can't change them. It might look like you can, but in essence you are replacing the old string with a new one.

https://docs.julialang.org/en/v1/manual/strings/

Types Directly Related to Strings

There are many types associated with strings.

- `SubString` is exactly what it says a substring often returned by matching operations.
- `Regex` is a regular expression pattern, but they are often specified by using a string literal preceeded by a 'r', *e.g.*, `r"matchthis"`. The regular expression itself might be stored in a non-array format (parsed into a state machine) but is input via a string.
- Byte array literals: again these are outside my ken but seem to be a way to get back to a simpler array format for strings of standard ASCII characters.
- Version numbers, *e.g.*, `v"0.2"`, like regular expressions, are specified using strings, but the syntax holds extra meaning.
- Raw string literals are a way to create a string without any *interpolation*. They create a string, but do so without the secret sauce of standard string creation, *e.g.*, we can deal with quote marks, backslashes and $ signs without any fuss.

```
julia> println(raw"\\ $x \\\"")
\\ $x \"
```

The Associative Array or Dictionary

Motivation: As noted above, the most general idea of an array is a mapping from one set A to another B. The associative array or dictionary implements this idea in a very general way. The mapping is one-to-one or many-to-one (but not one-to-many). The objects in the sets being mapped can be almost anything, though if you want use dictionaries for newly created types you may need to create some associated methods, in particular `hash` and `isequal` functions. In the context of associative arrays the indexes are often called *keys*, and the objects to which they are mapped called *values*.

Implementation: There are several common implementations of associative arrays (hash tables and trees are common). Julia's documentation isn't very clear about the exact implementation but it seems it uses a hash table, which is a hash function that maps the objects in the index set A to integers, and thence to the indices of a integer-indexed array that contains the members of set B (this is slightly simplistic as it doesn't consider how collisions are handled or how to efficiently dimension the hash table).

Types Directly Related to Dictionaries

There are quite a few types of a similar nature to a dictionary:

- A `Set` is an un-ordered set of objects. In some sense we can think of this as an associative array without any values.
- `Base.ImmutableDict` is exactly what it sounds like. However, the key difference is that because it is immutable it can be implemented in a more efficient manner as a linked list.
- Standard dictionaries have no ordering, either by key or value. A `SortedDict` in the `DataStructures.jl` package adds an ordering. It is generally slower that a standard dictionary because it has to maintain this ordering, and it seems to use a tree structure to do so. The package also includes `SortedSet`.

Ranges

Ranges are not strictly an array type. However, they can seem to be when one uses them.

A range might be defined in the form `x = 1: 2 : 10`, which means the sequence of numbers starting at 1, increasing with step size, until we reach 10. In Matlab, this code would create an array `[1,3,5,7,9]`. Julia uses a *lazy* approach. The values are calculated as needed. This avoids storing a long array which might only be used in chunks. It can still be accessed as if it were an array, *e.g.,* `x[2]` would be 3. However to convert it to a true array the function `collect` should be used. Incidentally `collect` will aim to do this on any collection or iterable.

2.2 Contributed Array Types

The previous section covers (primarily) types that are part of every Julia installation. There are also many, many array-like objects that are part of contributed packages. The list below is incomplete and probably outdated at the moment of publication, but aims to cover the most commonly useful contributed array types. More complete listings can be found at

- https://juliapackages.com/u/juliaarrays
- https://juliaobserver.com/categories/Matrix%20Theory
- https://github.com/JuliaArrays
- https://juliapackages.com/c/sparse-matrices

The order of presentation of these is somewhat random at present, but is intended to present types before they are used in other types.

Indirect and Pooled Arrays

https://github.com/JuliaArrays/IndirectArrays.jl

https://github.com/JuliaData/PooledArrays.jl

Motivation: In many situations an array stores values that come from a small set, $i.e.$, the array is a many-to-one mapping. Sometimes the objects in that set are also individually large and of variable size (*e.g.,* long strings). Thus a standard array that stored these directly would require a fairly large amount of memory to store objects that are often repeated (see DRY as a principle for computer programming). An indirect or pooled array is a much more efficient way to store such data. It uses a "pool" that contains a simple vector of each unique values and the main (index) array contains integers that index into the pool.

Implementation: The essential idea of an indirect array is that it creates a *value* or *pool* vector, which is a lookup table, and an *index* array such that

```
A[i,j] = value[ index[i,j] ]
```

The core of the indirect array package is a structure:

```
struct
IndirectArray{T,N,A<:AbstractArray{<:Integer,N},V<:AbstractVector{T}}
<: AbstractArray{T,N}
    index::A
    values::V
    ... # defines the constructor
end
```

The pooled array and indirect array packages solve essentially the same problem. It is not obvious which of indirect or pooled arrays is "best" but pooled arrays seems to be being more actively developed in recent years.

Categorical Arrays

https://categoricalarrays.juliadata.org/stable/

https://github.com/JuliaData/CategoricalArrays.jl

Motivation: Many arrays store numbers. However, we might wish to store *categorical* data, which is data where the traditional structures of numbers is missing. Categorical data can be *ordinal*, $i.e.$, they have an order (as with numbers, but lacking other structure such as operations like addition) or *nominal* where the categories don't have an ordering, *e.g.,* $\{fish, dog\}$. A standard Julia array can store any such type, but if there is a small finite set of such, then the same issues as for `IndirectArrays` and `PooledArrays` apply, *i.e.,* it can be highly inefficient to store the values directly.

Implementation: The `CategoricalArrays` package is similar to an indirect or pooled array, but has a more flexible interface It allows more easily for the array to support `Missing` values, and for an ordering to be specified.

Static Arrays

https://github.com/JuliaArrays/StaticArrays.jl

https://juliapackages.com/p/staticarrays

Motivation: As noted, the standard array in Julia is a *dynamic* array. Such arrays have advantages when the array size will vary during run time, but if the size of the array will be fixed a performance improvement can be gained by letting Julia know. The package `StaticArrays` allows one to do this. The speed ups using this package can be very large (estimated to be $5.9\times$ for matrix multiplications, for example). However, the advice in the package is that this speedup applies for smaller arrays, and that for arrays with more than 100 elements you should use the standard Julia arrays.

Implementation: There are a variety of approaches used in this package including mutable and immutable types. The consistent feature is the sizes of the arrays are fixed. It's all about letting the compiler (the LLVM) know enough to trigger various optimisations.

Mapped Arrays

https://github.com/JuliaArrays/MappedArrays.jl

Motivation: We often want to map the values in an array through some function, *e.g.,* `M[i] = f(A[i])`. Julia provides a syntax to do so: `M = map(f, A)` where f is a function or we can use the dot syntax `M = f.(A)`. However, if the matrix is large, and we only need the values of some elements (though we may not know which at compile time), then it makes sense to defer the calculation until the value is needed. However, it would be convenient to have an object M that works like a matrix to all effects. This type of *lazy* evaluation is a valuable strategy in computation.

Implementation: The intention is to create a lazy (in-place, element-wise) evaluation for cases such as

```
julia> A = [1 2 3; 4 5 6]
julia> f = x -> x^2
julia> M = map(f,A)
2×3 Matrix{Int64}:
  1   4   9
 16  25  36
```

However, we want to defer the calculation until needed. We do so as follows:

```
julia> using MappedArrays
julia> A = rand(100,100)
julia> f = x -> x^2
julia> M = mappedarray(f, A)
julia> sizeof(A), sizeof(M)
(80000, 8)
```

You can't set values in such an array because it doesn't really exist, *i.e.,* the numbers aren't stored in a block of memory. They are calculated as needed.

The package has some other facilities (you can include inverse functions and multiple source arrays).

Lazy Arrays

https://juliapackages.com/p/lazyarrays

Motivation: We have seen a simple case of lazy evaluation in mapped arrays. The `LazyArrays` package generalises this by allowing lazy versions of concatenation and multiplication.

Implementation: This is a somewhat technical subject, and one might be best referred to the documentation of the package.

Offset Arrays

https://github.com/JuliaArrays/OffsetArrays.jl

Motivation: a perennial question in the design of programming languages is "Should the array indices start at 0 or 1?" Mathematics does not have a consistent convention. Sometimes mathematicians start at 0 and sometimes 1. And there are good reasons for both. Starting at 0 means the index is directly related to the memory location of the item, whereas 1 is often more intuitive (1 is the first entry, to me at least). Julia starts indexing at 1, but many other languages start at 0, including the ever popular C. The `OffsetArrays` package allows you to choose 0 or 1, or some other starting index. To be honest, it seems to be a fairly optional package, but in some applications it may mean that you can make the program code match up to mathematical formulae more closely.

Implementation: The type seems to be a wrapper around standard `AbstractArray` that records the offsets and modifies the indexing functions appropriately.

Infinite Arrays

https://github.com/JuliaArrays/InfiniteArrays.jl

Motivation: Infinite arrays are useful objects in some mathematical contexts. In some cases we can make calculations with such arrays, or know their properties. The `InfiniteArrays.jl` package provides some methods to work with these.

Implementation: It may seem impossible to hold an infinite array in finite memory. It is done using Lazy Arrays (see above). However, the documentation of the package makes it sound like more is planned than implemented as yet.

Struct Arrays

https://juliapackages.com/p/structarrays

Motivation: It is common to store a set of composite objects – structs – in an array. For instance, we may want to store an array of complex or rational numbers. This is easy enough to do, but then one can only access sub-elements of the structure by indexing into the array, and selecting the element. The `StructArray` package creates a natural interface to index the sub-elements of the arrays as arrays themselves.

Implementation: The package provides quite a few features but the most obviously useful (to me at least) are things like the creation and indexing. For instance

```
julia> s = StructArray([ 1+3im, 2-1im ])
julia> s[1]    # as one would expect for a normal array
1 + 3im
julia> s.im[:] # access an array of the imaginary components
2-element Vector{Int64}:
  3
 -1
```

One can also create struct arrays without the intervening matrices (much like a comprehension):

```
julia> StructArray(log(j+2.0*im) for j in 1:3)
3-element StructArray(::Vector{Float64}, ::Vector{Float64}) with eltype
ComplexF64:
 0.8047189562170501 + 1.1071487177940904im
 1.0397207708399179 + 0.7853981633974483im
 1.2824746787307684 + 0.5880026035475675im
```

There are many other advanced computation features that make this package more than just a convenience package.

Axis and Named Arrays

https://juliapackages.com/p/axisarrays

https://github.com/davidavdav/NamedArrays.jl

Motivation: A 2D array is indexed by (row,col) as in standard mathematical notation. That is, `A[i,j]` refers to the ith row and jth column. Not all programming languages use this convention (some index first by column). Moreover, in some cases it might be that we don't want to call the indexes 'rows' or 'columns', and possibly we want to be able to use an arbitrary ordering of indexes. The `AxisArrays` and `NamedArrays` packages allow access through indexes that have a *name*.

Implementation: For example

```
julia> n = NamedArray([1 3; 2 4], ( ["a", "b"], ["c", "d"] ), ("Rows", "Cols"))
Rows \ Cols | c  d
────────────┼────
a           | 1  3
b           | 2  4
```

`AxisArrays` provides somewhat more detail (including where it is going), but (to me) seems to have a more complex interface. One feature that seems appealing is the ability to index by an *interval*, that is, to select a range of values by some set bounds.

Both seem to provide an interesting idea, but examples often seem to be reaching towards tables, which we will examine in more detail below.

Tables

https://github.com/JuliaData/Tables.jl

https://tables.juliadata.org/stable/

Motivation: A *Table* is a 2D array of data. It varies from a simple array in that it has:

- named columns that are all the same length,
- columns that (typically) have a consistent type within the column, but the types may vary between columns, and
- a common set of interfaces: *e.g.,* `select`, `filter`, `transform`, `map`, …

Tables are very, very common data structures for dealing with many types of data.

However, to me, the `Tables` package feels like the underlying toolkit, not the interface that I want to use (which is DataFrames, see below).

Implementation: this is a big topic, and beyond my ken. See for discussion:

- https://discourse.julialang.org/t/tables-jl-a-table-interface-for-everyone/14071

It is worth, however, noting the existence of the **`PrettyTables`** package, which is very useful for outputting the results in a manifold variety of nice formats including text, LaTeX and markdown: see https://ronisbr.github.io/PrettyTables.jl/dev/.

Data Frames

https://dataframes.juliadata.org/stable/

https://github.com/JuliaData/DataFrames.jl

https://en.wikibooks.org/wiki/Introducing_Julia/DataFrames

Motivation: I see the `Tables` package as an underlying mechanism to unify and make more efficient a range of other packages. The one that I use most frequently is the `DataFrames` package. The concept of a Data Frame comes (as far as I know) from R, where it is an integral part of the Tidyverse (specifically `dplyr`) and similar to some concepts in Python's PANDAS package. Data Frames are Tables that have a few additional features:

1. They allow (efficient) missing values;
2. They are linked to the idea (in statistical analysis) that each row corresponds to an observation and each column to a (co)variate (a measurement made for that observation).

Implementation: A Data Frame is a structure where there are

1. Column names (in a vector of `Symbol`s)
2. A vector of columns, each element is a column, which is a vector itself.

Each column needs to be the same length, so we can think of a Data Frame as a 2D array or elements, and index it as if it were, but this is not how it actually works. The links above go into much more detail about how to use Data Frames, so I will not do so here.

Julia has several addition packages for working with DataFrames specifically, *e.g.,* `Query.jl` and `DataFramesMeta.jl` and the package integrates well with others such as `CSV.jl` for data I/O, statistics packages, machine learning packages and some plotting packages. It uses packages like `CategoricalArrays` to efficiently store data when required.

Comparisons to PANDAS and dplyr can be founds here:

https://dataframes.juliadata.org/stable/man/comparisons/

Many Others

There are many other array types that have been created over time, *e.g.,*:

- BlockArrays https://juliapackages.com/p/blockarrays
- TiledArrays https://juliapackages.com/p/tilediteration

and many others. See:

- https://juliapackages.com/u/juliaarrays
- https://juliaobserver.com/categories/Matrix%20Theory
- https://github.com/JuliaArrays
- https://juliapackages.com/c/sparse-matrices

3. Issues and Discussion

Julia includes dictionaries, arrays, tuples, sets and some others under the heading of "collections." Within this you can subdivide these collections by whether they are

- indexable (by integers) or associative; and
- mutable or immutable.

Collections have a group of common interfaces (methods) such as `pop!`, `push!` and `append!`, though obvious not all are implemented for all collections.

https://scls.gitbooks.io/ljthw/content/_chapters/07-ex4.html

Other common methods:

- filter()
- find() (and findnext() and findfirst() and …)
- sort()
- all() and any()
- size()
- length()
- get()
- set!()
- keys(), values()
- isempty()
- …

Linear-indexed arrays can be operated on by the usual range of mathematical operators. Julia also includes the "dotted" version of operators to create element-wise versions of operations and to broadcast a function across all elements of an array. Much is said about this, so I won't say much more here except that muich of the Linear Algebra (see the `LinearAlgebra` package) is built on the LAPACK library, which is built on top of BLAS (the basic linear-algebra package), which are written in highly optimised FORTRAN 90 and used by very many modern programming languages to implement their linear algebra. One of the key goals, in trying to improve performance in alternative array implementations should always be to ensure you can still use these underlying routines, because they are likely to be many times more efficient than any crudely customised routines.

4. Summary

Julia has a vast "array" of array types useful for almost any purpose you can imagine. Keeping track of what is available and what use case each serves is not easy. That is the main purpose of this document.

The breadth and scope of the variations described here, along with the fact of Julia's enviable efficiency, are why we should think of languages like Matlab as wind-up tinker toy cars, R and Python as work-a-day family minivans, and Julia as the Batmobile. We won't be replacing the minivan any time soon, but I know which one I want to drive. BTW, in this crude analogy C is a Formula-1 car: it's ridiculously fast, but if you don't end up crashing on the first corner, it's just luck.