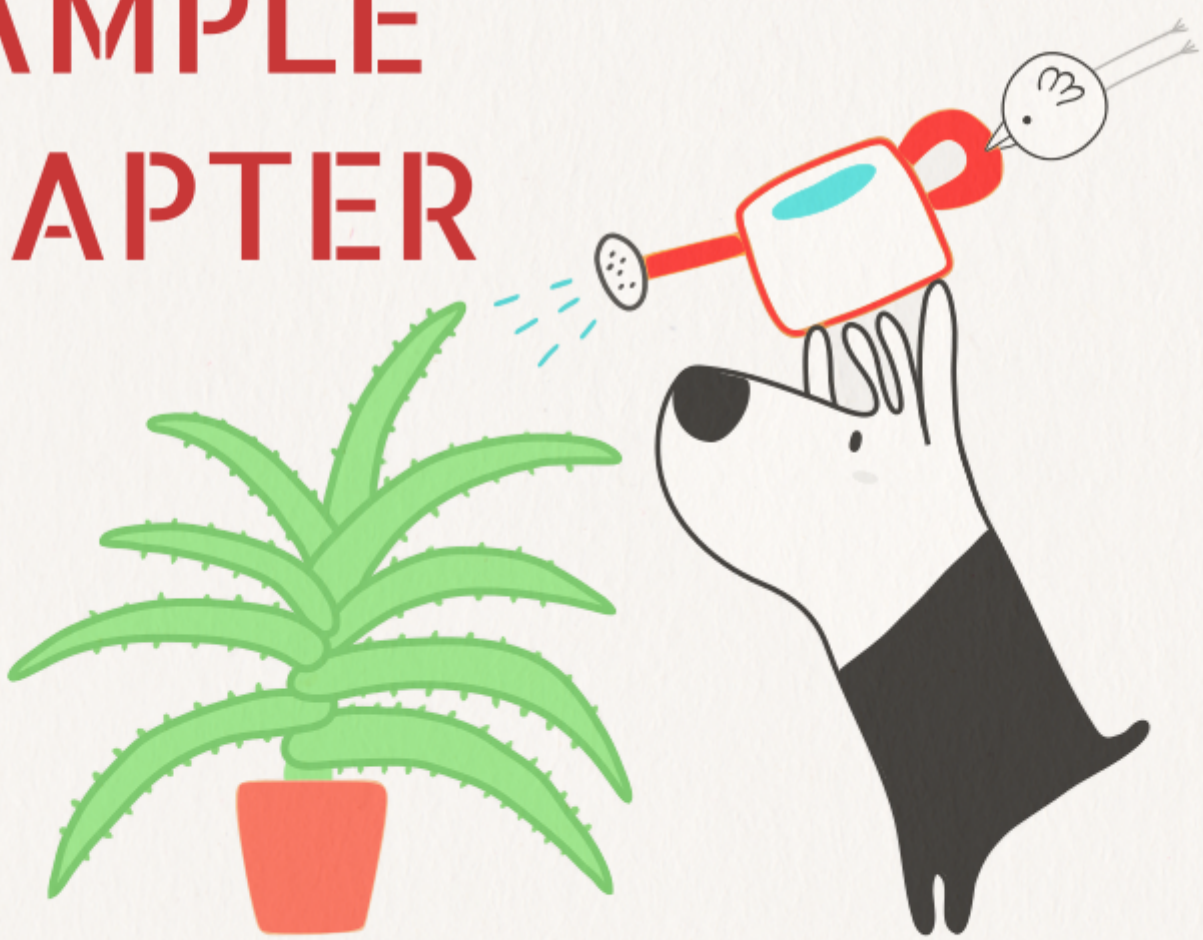
Numerical Linear Algebra for Programmers

Dragan Djuric

SAMPLE CHAPTER



DRAFT

An Interactive Tutorial with GPU, CUDA, OpenCL, MKL, Java, and Clojure

https://dragan.rocks

aiprobook.com

DRAGAN DJURIC

NUMERICAL LINEAR ALGEBRA FOR PROGRAMMERS SAMPLE CHAPTER [

DRAGAN ROCKS

PUBLISHED BY DRAGAN ROCKS

HTTP://AIPROBOOK.COM

First printing, 20 June 2019

Sample

Linear algebra refresher

If you are at least a bit like me, you had learned (some) linear algebra during your university days, had done well at those math courses playing the human calculator role, multiplying matrices with nothing but pen and paper, but then buried these experiences deep down in your brain during those many years at typical programming tasks (that are not famous for using much linear algebra).

Now, you want to do some machine learning, or deep learning, or simply some random number crunching, and intuition takes you a long way, but not far enough: as soon as you hit more serious tasks, beyond the cats and dogs deep learning tutorials, there are things such as eigenvalues, or LU factorization, or whatever-the-hell-that-was. You can follow the math formulas, especially when someone else have already implemented them in software, but not everything is clear, and sooner or later **you** are the one that needs to make working software out of that cryptic formula involving matrices.

In other words, you are not completely illiterate when it comes to maths, but you can not really work out this proof or that out of the blue; your math-fu is way below your programming-fu. Fortunately, you are a good programmer, and do not need much hand holding when it comes to programming. You just need some dots connecting what you read in the math textbooks and the functions in a powerful linear algebra library.

This part briefly skims through a good engineering textbook on linear algebra, making notes that help you relate that material to Clojure code. I like the following book: Linear Algebra With Applications, Alternate Edition by Gareth Williams[1]. The reasons I chose this book are:

[1] We use the 7th edition as a reference.

- It is oriented towards applications.
- It is a nice hardcover, that can be purchased cheaply second-hand.
- The alternate edition starts with 100 pages of matrix applications before diving into more abstract theory; good for engineers!

Any decent linear algebra textbook will cover the same topics, but not necessarily in the same order, or with the same grouping. This part covers chapter starting with chapter 4 from the book that I recommended. A typical (non-alternate) textbook will have this chapter as a starting chapter.

Vector spaces

The vector space R^n

To work with vectors in a vector space R^n, we use Neandertal vectors. The simplest way to construct the vector is to call a constructor function, and specify the dimension n as its argument. Sometimes the general `vctr` function is appropriate, while sometimes it is easier to work with specialized functions such as `dv` and `sv` (for native vectors on the CPU), `clv` (OpenCL vectors on GPU or CPU), or `cuv` (CUDA GPU vectors).

At first, we use native, double floating-point precision vectors on the CPU (`dv`). Later on, when we get comfortable with the vector based thinking, we introduce GPU vectors.

To get access to these constructors, we require the namespace that defines them. We also require the core namespace, where most functions that operate on vectors and matrices are located.

```
(require '[uncomplicate.neanderthal
           [native :refer :all]
           [core :refer :all]])
```

For example, `v1` is a vector in R^4, while `v2` is a vector in R^{22}.

```
(def v1 (dv -1 2 5.2 0))
(def v2 (dv (range 22)))
```

If we print these vectors out in the REPL, we can find more details about them, including their *components* (elements, entries):

```
v1
```

```
=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[  -1.00    2.00    5.20    0.00 ]
```

```
v2
```

```
=>
#RealBlockVector[double, n:22, offset: 0, stride:1]
[   0.00    1.00    2.00         20.00   21.00 ]
```

Addition and Scalar Multiplication

Math books define vector spaces using surprisingly few (at least to us, programmers) things: *addition* and *scalar multiplication*.

Addition in mathematics is simply an operation, $+$, but in programming, we are doing numerical computations, so we are also concerned with the implementation details that math textbooks do not discuss. One way to add two vectors would be the function `xpy`[2]:

[2] `xpy` mnemonic: x plus y.

```
(xpy v1 v2)

=>
clojure.lang.ExceptionInfo
   Dragan says: You cannot add incompatible or ill-fitting structures.
```

This does not work, since we cannot add two vectors from different vector spaces (R^4 and R^{22}), and when we evaluate this in the REPL, we get an exception.

The vectors have to be compatible, both in the mathematical sense being in the same vector space, and in an implementation specific way, in which there is no sense to add single-precision CUDA vector to a double-precision OpenCL vector.

Let's try with `v3`, which is in R^4:

```
(def v3 (dv -2 -3 1 0))

(xpy v1 v3)

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[  -3.00   -1.00    6.20    0.00 ]
```

This function is pure; `v1` and `v2` have not changed, while the result is a new vector instance (which in this case has not been kept,but went to the garbage).

[3] `scal`, scalar multiplication

Scalar multiplication is done using the pure function `scal`.[3] It accepts a scalar and a vector, and returns the scaled vector, while leaving the original as it was before:

```
(scal 2.5 v1)

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[  -2.50    5.00   13.00    0.00 ]

v1
```

```
=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[  -1.00    2.00    5.20    0.00 ]
```

Pure functions are nice, but in the real world of numerical computations, we are constrained with time and space: we need our vectors to fit into available memory, and the results to be available today. With vectors in R^4, computers will achieve that no matter how bad and unoptimized our code is. For the real world applications though, we'll deal with billions of elements and demanding operations. For those cases, we'll want to do two things: minimize memory copying, and fuse multiple operations into one.

When it comes to vector addition and scalar multiplication, that means that we can fuse `scal` and `xpy` into `axpy`[3] and avoid memory copying by using destructive functions such as `scal!` and `axpy!`[4].

[4] Note the `!` suffix.

```
(axpy! 2.5 v1 v3)

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[  -4.50    2.00   14.00    0.00 ]
```

Note that `v3` has changed as a result of this operation, and it now contains the result, written over its previous contents:

```
v3

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[  -4.50    2.00   14.00    0.00 ]
```

Special Vectors

Zero vector can be constructed by calling vector constructor with an integer argument that specifies dimension:[5]

[5] zero-vector

```
(dv 7)

=>
#RealBlockVector[double, n:7, offset: 0, stride:1]
[   0.00    0.00    0.00          0.00    0.00 ]
```

When we already have a vector, and need a zero vector in the same vector space, having the same dimension, we can call the `zero` function:

```
(zero v2)
```

```
=>
#RealBlockVector[double, n:7, offset: 0, stride:1]
[   0.00    0.00    0.00          0.00    0.00 ]
```

Negative of a vector is computed simply by scaling with `-1`:

```
(scal -1 v1)

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[   1.00   -2.00   -5.20   -0.00 ]
```

How to do *vector subtraction*? As we mentioned earlier, there are only two independent operations in R^n, vector addition and scalar multiplication. Vector subtraction is simply an addition with the negative vector. We can do this with one fused operation, be it `axpy` or `axpby`:

```
(axpby! 1 v1 -1 v3)

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[   3.50    0.00   -8.80    0.00 ]
```

Linear Combinations of Vectors

To compute a linear combination such as $a\mathbf{u} + b\mathbf{v} + c\mathbf{w}$, we can use multiple `axpy` calls or even let one `axpy` call sort this out (and waste less memory by not copying intermediate results)!

```
(let [u (dv 2 5 -3)
      v (dv -4 1 9)
      w (dv 4 0 2)]
  (axpy 2 u -3 v 1 w))

=>
#RealBlockVector[double, n:3, offset: 0, stride:1]
[  20.00    7.00  -31.00 ]
```

Column Vectors

Both *row* vectors and *column* vectors are represented in the same way in Clojure: with Neanderthal dense vectors. Orientation does not matter, and the vector will simply fit into an operation that needs the vector to be horizontal or vertical in mathematical sense.

Dot product, norm, angle, and distance

Dot product of two compatible vectors, is a scalar sum of products of the respective entries.

$$\mathbf{u} \cdot \mathbf{v} = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n \tag{1}$$

It can be computed using the function with a predictably boring name, `dot`[6]:

[6] dot product

```
(let [u (dv 1 -2 4)
      v (dv 3 0 2)]
  (dot u v))
```

```
=> 11.0
```

Norm (aka length, or magnitude, or L2 norm) most often refers to the Euclidean distance between two vectors, although other norms can be used:

$$\|\mathbf{u}\| = \sqrt{u_1^2 + u_2^2 + \cdots + u_n^2} \tag{2}$$

In Clojure, we use the `nrm2`[7] function to compute this norm:

[7] `nrm2`: L2 norm, length, magnitude

```
(nrm2 (dv 1 3 5))
```

```
=> 5.916079783099616
```

A *unit vector* is a vector whose norm is 1. Given a vector `v`, we can construct a unit vector `u` in the same direction.

$$\mathbf{u} = \frac{1}{\|\mathbf{v}\|}\mathbf{v} \tag{3}$$

In Clojure code, we can do this as the following code shows.

```
(let [v (dv 1 3 5)]
  (scal (/ (nrm2 v)) v))
```

```
=>
#RealBlockVector[double, n:3, offset: 0, stride:1]
[   0.17    0.51    0.85 ]
```

This process is called *normalizing* the vector.[8]

[8] normalizing the vector

Angle Between Vectors

Any linear algebra book will teach how to compute the cosine of an angle between two vectors.

$$cos\theta = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\|\|\mathbf{v}\|} \tag{4}$$

We can program this easily using the `dot` and `nrm2` functions that we have already introduced.

```
(let [u (dv 1 0 0)
      v (dv 1 0 1)]
  (/ (dot u v) (nrm2 u) (nrm2 v)))
```

```
=> 0.7071067811865475
```

We can compute θ out of this cosine, but sometimes we do not even need to do that. For example, two vectors are *orthogonal*[9] if the angle between them is 90°. Since we know that $cos\frac{\pi}{4} = 0$, we can simply test whether the dot product is `0`.

[9] orthogonal vectors

```
(dot (dv 2 -3 1) (dv 1 2 4))
```

```
=> 0.0
```

These two vectors are orthogonal.

We can determine *distance between points*[10] in a vector space by calculating the norm of the difference between their direction vectors.

[10] distance between points

$$d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\| \tag{5}$$

In Clojure, that is as simple to do as computing the norm of the difference of two vectors.

```
(let [x (dv 4 0 -3 5)
      y (dv 1 -2 3 0)]
  (nrm2 (axpy -1 y x)))
```

```
=> 8.602325267042627
```

General vector spaces

It is important to note that real vectors are not the only "kind" of vector spaces there is. Anything that has operations of addition and scalar multiplication that have certain properties[11] is a vector space. Some well known vector spaces are real vectors (R^n), *matrices* (M^{mn}), complex vectors (C^n), and, of course, functions. However, when we do numerical computing, we usually work with real or complex vectors and matrices, so in the following discussion we refer only to the parts of the textbook that deal with those. For the abstract theoretical parts, we typically do not use programming anyway, but our trusty pen and paper.

[11] See math textbooks for details.

We have already learned how to use vectors. With matrices, it's similar, just with more options. The first difference is that we can use

different optimized implementations of matrices, to exploit knowledge of their structure. Therefore, there are *dense* matrices (GE)[12], *dense triangular* matrices (TR), *dense symmetrical* matrices (SY), various banded storages for symmetric and triangular matrices, sparse matrices (Sp), etc.

Let's construct a matrix and compute its norm:

```
(def m (dge 2 3 (range 6)))
m

=>
#RealGEMatrix[double, mxn:2x3, layout:column, offset:0]
┌       ↓       ↓       ↓       ┐
→       0.00    2.00    4.00
→       1.00    3.00    5.00
└                               ┘

(nrm2 m)

=> 7.416198487095662
```

[13] Frobenius norm

Euclidean norm applied to matrices is known as the Frobenius norm.[13]

Most of the functions that can be applied to vectors, can also be applied to matrices. Furthermore, there are many more linear algebra functions that work exclusively with matrices.

Subspaces and linear combinations

These two topics, *Subspaces* and *Linear Combinations* are theoretical. We skip them, since we cannot do anything in code related to this.

Linear dependence and independence

Typical math textbook covers lots of theory related to linear dependence and independence of vectors.

When it comes to computation, we can note that two vectors are linearly dependent if they are on the same line, implying that the angle between them is 0, meaning the cosine is 1. If cosine is not 1, they are independent. There is a catch, though: floating-point computations are not absolutely precise. For very large vectors, sometime there will be rounding error, so we might get something like 0.99999999 or 1.000001.

Comparing floating-point numbers for equality is a tricky business. Keep that in mind, and know that even there Neanderthal can help us, since it offers functions that take some imprecision margin into account.

See the Neanderthal API docs and look up functions such as `f=` in the `math` namespace.

So, linear dependence is computed in the following way.

```
(let [x (dv 2 -1 3)
      y (dv 4 -2 6)]
  (/ (dot x y) (nrm2 x) (nrm2 y)))

=> 1.0
```

These two vectors are linearly dependent indeed.

But, what if we have more than two vectors, and are tasked with ckecking whether their set is dependent or not?

Consider example 1 from the page 170 of the textbook I recommended. There is a set of vectors $\{x = (1,2,0), y = (0,1,-1), z = (1,1,2)\}$, and we need to find whether there are scalars $c_1, c_2,$ and c_3, such that $c_1\mathbf{x} + c_2\mathbf{y} + c_3\mathbf{z} = 0$.

This leads to a system of linear equations that, when solved, could have an unique solution, in which case the vectors are linearly independent, or to more than one solution, in which case the vectors are linearly dependent.

Luckily for us, Neanderhal offers functions for solving systems of linear equations. Require the `linalg` namespace to reach for them.

```
(require '[uncomplicate.neanderthal.linalg :refer :all])
```

Now, we can set up and solve the system that will find the answer.

```
(sv! (dge 3 3 [1 2 0
               0 1 -1
               1 1 2])
     (dge 3 1))

=>
#RealGEMatrix[double, mxn:3x1, layout:column, offset:0]
┌       ↓       ┐
→       0.00
→      -0.00
→      -0.00
└               ┘
```

Just like in the textbook, the solution is: $c_1 = 0, c_2 = 0, c_3 = 0$. If we tried with obviously dependent vectors, we would have got an exception complaining that the solution could not be computed.

Basis and dimension

Here are two more theoretical topics. They do not offer great material for code demonstration.

Note that the dimension of the space of a vector could be inquired with the `(dim x)` function. The dimension of the space of the matrix is $m \times n$, so `(* (mrows a) (ncols a))`. All the spaces we deal with here have *finite dimension*, while we leave *infinite dimension* vectors to pen and paper adventures.

Rank

Rank of a matrix is useful since it relates matrices to vectors, and can tell us some useful properties of a matrix at hand. See the textbook for the definition and use. What is the question right now is how do we compute it? It is not easy at all, without a linear algebra library. When we inquire about the rank, we expect a natural number as a result.

We can get rank by doing singular value decomposition by calling the function `svd`[14], and inspecting one of its results, namely the sorted vector of singular values `s`. The number of singular values in `:sigma` that are greater than 0 is the rank of the matrix. As before, take into the account that floating point numbers should be compared with equality carefully.

[14] `svd`: singular value decomposition

Here is an example[15] done in Clojure.

[15] Example 3 on page 185 in the textbook.

```
(let [a (dge 4 4 [1 0 0 0
                  2 0 0 0
                  0 1 0 0
                  0 0 1 0])]
  (svd a))

=>
#uncomplicate.neanderthal.internal.common.SVDecomposition{
:sigma #RealDiagonalMatrix[double, type:gd mxn:4x4, offset:0]
┌                                          ┐
↘       2.24    1.00    1.00    0.00
└                                          ┘
, :u nil, :vt nil, :master true}
```

After inspecting the vector `:sigma` for non-zero values (visually, or by writing a small function that checks its elements) we are a little better informed than before since we now know that $rank(A) = 3$.

This seems so cumbersome. Why isn't there a function `rank` in Neanderthal that does this for us? Well, the problem is that we rarely

need to compute only the rank. Often we also need those other results from `svd!` or similar functions. If `rank` was so easy to access, it would have promoted wasteful computations of other results that go with it. If we really need `rank` it is easy to write that function as part of a personal toolbox. What is important is that the cake is available right now, so we can easily add our own cherries on top of it.

Orthonormal vectors and projections

This is yet another theoretical section. From the computational point of view, what could be interesting, is computing projection of a vector onto another vector or a subspace.

$$proj_u v = \frac{v \cdot u}{u \cdot u} u \tag{6}$$

It boils down to the ability to compute the dot product and then scale the vector u.

```
(let [v (dv 6 7)
      u (dv 1 4)]
  (scal (/ (dot u v) (dot u u)) u))
```

```
=>
#RealBlockVector[double, n:2, offset: 0, stride:1]
[   2.00    8.00 ]
```

This is similar for subspaces, so that might be the obligatory trivial exercise that is being left to the reader.