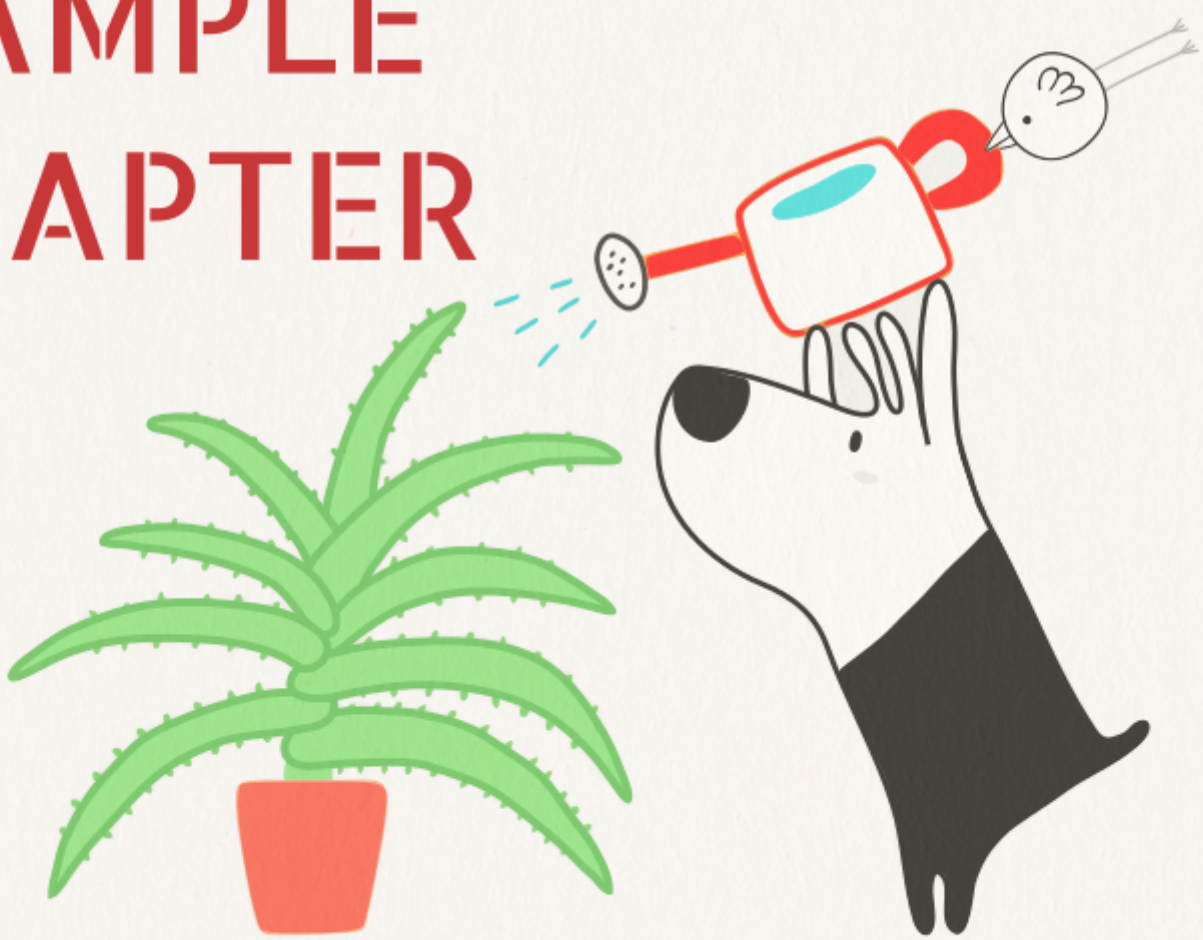
Numerical Linear Algebra for Programmers

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.

Linear transformations

Now that we got ourselves acquainted with matrix transformations, the next obvious step is to generalize our knowledge with linear transformations. This post covers a little more theoretical ground, and it will refer the reader to the linear algebra textbook more often than the last chapter. There will be less code, since the illustrating computations are similar to what we have already used in previous chapters. That is a good thing; it indicates that we have covered the most important things, and just need more understanding of math theory and applications, and a finer grasp of the details of the performance of numerical computations.

This chapter covers the second part of chapter 6 of the alternate edition of the textbook.

Defining linear transformations

Recall from the chapter on vector spaces that a vector space has two operations: addition and scalar multiplication.

Consider the following matrix transformations:

$$T(\mathbf{u}+\mathbf{v}) = A(\mathbf{u}+\mathbf{v}) = A\mathbf{u} + A\mathbf{v} = T(\mathbf{u}) + T(\mathbf{v}) \tag{1}$$

$$T(c\mathbf{u}) = A(c\mathbf{u}) = cA\mathbf{u} = cT(\mathbf{u}) \tag{2}$$

Having this in mind, the following textbook definition seems obvious.

Let $\mathbf{u}$ and $\mathbf{v}$ be vectors in R^n and c be a scalar. A transformation $T : R^n \rightarrow R^n$ is a *linear transformation*[2] if $T(\mathbf{u}+\mathbf{v}) = T(\mathbf{u}) + T(\mathbf{v})$ and $T(c\mathbf{u}) = cT(\mathbf{u})$.

These properties tell us that all linear transformations preserve addition and scalar multiplication. Every matrix transformation is linear, but translations and affine transformations are not.[3]

We can refer to a transformation where domain and codomain are the same, such as $R^n \rightarrow R^n$, as an *operator*.[4]

[2] linear transformation

[3] We will see later that there is a workaround.

[4] operator

Following the textbook, in the previous post we used ad hoc ways of arriving at matrices that describe transformations such as dilations, rotations, and reflections. Now, we will learn a formal method.

The following example[5] tries to find a matrix that describes the following transformation:

[5] Example 3 from page 259 of the textbook.

$$T\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} 2x + y \\ 3y \end{bmatrix}$$

Here is the strategy for our programming code. First we find the effect of T on the standard basis of the domain R^n, and then form a matrix whose columns are those images. Easy!

```
(defn t! [v]
  (v 0 (+ (* 2 (v 0)) (v 1)))
  (v 1 (* 3 (v 1))))

(def a (dge 2 2 [1 0 0 1]))

(t! (col a 0))
(t! (col a 1))
a

=>
#RealGEMatrix[double, mxn:2x2, layout:column, offset:0]
┌       ↓       ↓       ┐
→       2.00    1.00
→       0.00    3.00
└                       ┘
```

Please excuse me for introducing the imperative code in function `t!`, but we've found the matrix that represents the linear transformation. Please keep in mind that vectors and matrices are functions that can retrieve or update values at specific index, but that these functions are usually boxed. In tight loops, we would like to use `entry` and `entry!` from the `real` namespace, or some other operation optimized for performance.

Let T be a linear transformation on R^n, $\{\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}\}$ the *standard basis*[6] and $\mathbf{u}$ arbitrary vector in R^n.

[6] Review the chapter on vector spaces.

$$\mathbf{e_1} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \mathbf{e_2} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \ldots, \mathbf{e_n} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}, \text{ and } \mathbf{u} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}.$$

We can express $\mathbf{u}$ as a linear combination of $\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$: $\mathbf{u} = c_1\mathbf{e_1} + c_2\mathbf{e_2} + \cdots + c_n\mathbf{e_n}$. If we fill that in $T(\mathbf{u})$, and apply the properties of linear transformations, we find that $A = [T(\mathbf{e_1}) \cdots T(\mathbf{e_n})]$. A is called the *standard matrix*[7] of T.

[7] standard matrix

The previous definition of linear transformations in R^n can be expanded for any vector space, not only R^n.

Homogeneous coordinates

Translation and affine transformation[8] are not linear transformations and cannot be composed, yet they are very useful operations. Programmers in computer graphics understand the value of linear algebra and utilize it well. We can see an example of how the problem of translation and affine transformation composability have been overcome in that field, and keep that in mind when we develop solutions that use more than 3 dimensions.

[8] See the chapter on matrix transformations.

It turns out that *homogeneous coordinates*[9] can describe points in a plane in a way that matrix multiplication can be used for translations. In homogeneous coordinates, a third component of 1 is added to each coordinate, and the transformations are defined by the following matrices:

[9] homogenous coordinates

- point $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$[10]
- rotation $A = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$[11]
- reflection $B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$[12]
- dilation/contraction $C = \begin{bmatrix} r & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & 1 \end{bmatrix}, r > 0$[13]
- translation $E = \begin{bmatrix} 1 & 0 & h \\ 0 & 1 & k \\ 0 & 0 & 1 \end{bmatrix}$[14]

[10] point

[11] rotation

[12] reflection

[13] dilation/contraction

[14] translation

If we, for example, wanted to do a dilation followed by a translation and then a rotation, we would use the AEC matrix.

The recommended math textbook offers a practical example[15]. There is a triangle, and we would like to rotate it, but not about the origin, but about a point $P(h, k)$. Thus, the solution is to translate P to origin, rotate, and then translate P back. As a concrete

[15] Example 7, page 264 of the textbook.

example, we implement the rotation through an angle $\pi/2$ about the point $P(5,4)$.

```
(defn t1 [p]
  (dge 3 3 [1 0 0 0 1 0 (- (p 0)) (- (p 1)) 1]))

(defn r [theta]
  (let [c (cos theta)
        s (sin theta)]
    (dge 3 3 [c s 0 (- s) c 0 0 0 1])))

(defn t2 [p]
  (dge 3 3 [1 0 0 0 1 0 (p 0) (p 1) 1]))

(defn t2rt1 [p theta]
  (mm (t2 p) (r theta) (t1 p)))

(t2rt1 (dv 5 4) (/ pi 2))

=>
#RealGEMatrix[double, mxn:3x3, layout:column, offset:0]
┌       ↓       ↓       ↓       ┐
→       0.00   -1.00    9.00
→       1.00    0.00   -1.00
→       0.00    0.00    1.00
└                               ┘
```

Kernel, range, and the rank/nullity theorem

Besides *domain* and *codomain*, linear transformations are associated with another two important vector spaces: *kernel*[16] and *range*[17].

In the transformation $T : U \to V$:

- *kernel*, $ker(T)$, is the set of vectors in U that are mapped into the zero vector of V.
- *range* is the set of vectors in V that are the images of vectors in U.

This section does not contain anything new we could demonstrate with code, but it is important to grasp the theory. Therefore, we mention the important stuff, and know when to find it in the math textbooks if the need arises.

The following example[18] demonstrates how to find the kernel, by solving a system of equations $T(\mathbf{x}) = A\mathbf{x} = \mathbf{0}$. The first thing that we could do is to try to use the function for solving linear systems of equations `sv!`[19]. That does not help us much; since our system

[16] kernel
[17] range

[18] Example 2 on page 273 of the textbook.

[19] `sv!` mnemonic is *solve* (a system of equation).

is homogeneous, `sv!` can only give us the trivial solution $\mathbf{x} = \mathbf{0}$. Fortunately, in this example, A is a square matrix, so we can exploit the equality $A\mathbf{x} = \lambda\mathbf{x}$ between eigenvectors $\mathbf{x}$ and eigenvalue(s) $\lambda = 0$, if such λ exists.[20]

[20] Since that is a textbook example, we should be surprised if it didn't.

```
(def a1 (dge 3 3 [1 0 1 2 -1 1 3 1 4]))
(def eigenvectors (dge 3 3))
(def lambdas (dge 3 2))
(ev! a1 lambdas nil eigenvectors)
```

The eigenvalues have only the real part, while the imaginary part is zero.

```
=>
#RealGEMatrix[double, mxn:3x2, layout:column, offset:0]
┌        ↓        ↓        ┐
→        5.00     0.00
→       -0.00     0.00
→       -1.00     0.00
└                           ┘
```

Let's see whether any of the eigenvectors is our solution.

```
eigenvectors
```

```
=>
#RealGEMatrix[double, mxn:3x3, layout:column, offset:0]
┌        ↓        ↓        ↓        ┐
→       -0.64    -0.96     0.71
→       -0.13     0.19    -0.71
→       -0.76     0.19     0.00
└                                    ┘
```

The second eigenvalue is 0, so the second column $(-0.96, 0.19, 0.19)$ is the normalized basis of one-dimensional subspace of R^n that is the kernel we were looking for.

```
(seq (col eigen 1))
```

```
=> (-0.9622504486493764 0.1924500897298754 0.1924500897298751)
```

For a linear transformation T,
$dim(ker(T)) + dim(range(T)) = dim(domain(T))$.
Also, $dim(range(T)) = rank(A)$.

The kernel of T is also called the *null space*[21]. $dim(ker(T))$ is called *nullity*[22] and $dim(range(T))$ is the *rank*[23] of the transformation. The previous equality is referred to as the *rank/nullity* theorem.

[21] null space
[22] nullity
[23] rank

Systems of Linear Equations

As we've already seen in the previous section, there is a close connection between linear transformations and systems of linear equations. System $A\mathbf{x} = \mathbf{y}$ can be written as $T(\mathbf{x}) = \mathbf{y}$.

Note that the solution of the *homogeneous system* $A\mathbf{x} = \mathbf{0}$ is the *kernel* of the transformation, as we've already shown.

There is an interesting relation between an element of the *kernel* $\mathbf{z}$, particular solution $\mathbf{x_1}$ of the *nonhomogeneous system* $A\mathbf{x} = \mathbf{y}$, and every other solution $\mathbf{x_1}$ of that system: $\mathbf{x} = \mathbf{z} + \mathbf{x_1}$!

We can illustrate this with an example. [24] We use the `sv!` function to solve this non-homogeneous system.

[24] Example 2 on page 286 in the textbook.

$$\begin{bmatrix} 1 & 1 & 3 \\ 0 & -1 & 1 \\ 1 & 1 & 4 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 11 \\ -2 \\ 9 \end{bmatrix}$$

```
(def x1
  (let [a (dge 3 3 [1 0 1 1 -1 1 3 1 4])
        b (dge 3 1 [11 -2 9])]
    (col (sv! a b) 0)))
```

```
=>
#RealBlockVector[double, n:3, offset: 0, stride:1]
[  17.00   -0.00   -2.00 ]
```

Now, knowing the kernel from the previous section, and having this particular solution, we can construct all solutions to this system:

```
(let [z (col eigen 1)]
  (defn y [r]
    (axpy r z x1)))


(y 1.33)
```

```
=>
#RealBlockVector[double, n:3, offset: 0, stride:1]
[  15.72    0.26   -1.74 ]
```

As an exercise, check whether those vectors are really the solutions.

The `sv!` solves more than only one system at a time. We can give it as many systems as we like, and solve them in one sweep, as in this example.

$$\begin{bmatrix} 1 & 1 & 3 \\ 0 & -1 & 1 \\ 1 & 1 & 4 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 11 & 3 & 4 & -1 \\ -2 & -2 & 5 & 0 \\ 9 & 1 & -6 & 1 \end{bmatrix}$$

```
(let [a (dge 3 3 [1 1 3
                  0 -1 1
                  1 1 4]
             {:layout :row})
      b (dge 3 4 [11 3 4 -1
                  -2 -2 5 0
                  9 1 -6 1]
             {:layout :row})]
  (sv! a b))

=>
#RealGEMatrix[double, mxn:3x4, layout:column, offset:0]
┌       ↓       ↓       ↓       ↓       ┐
→      17.00    9.00   49.00   -9.00
→      -0.00   -0.00  -15.00    2.00
→      -2.00   -2.00  -10.00    2.00
└                                       ┘
```

We did not kill seven in one blow, like the Brave Little Tailor, only four. But we could have, easily, even more than seven!

One-to-One and Inverse Transformations

This section covers only the theory, so we just summarize the results. The details can be explored in any math textbook.

A transformation is one-to-one[25] if each element in the range is the image of exactly one element in the domain.

A transformation T is invertible[26] if there is a transformation S such that $S(T(\mathbf{u})) = \mathbf{u}$ and $T(S(\mathbf{u})) = \mathbf{u}$ for every $\mathbf{u}$.

The following statements are equivalent:

- T is invertible.
- T is nonsingular ($det(a) \neq 0$).
- T is one-to-one.
- $ker(T) = \mathbf{0}$.
- $ran(T) = R^n$.
- T^{-1} is linear.
- T^{-1} is defined by A^{-1}.

[25] one-to-one transformation

[26] invertible transformation

Coordinate Vectors

Let U be a vector space, $B = \{\mathbf{u_1}, \mathbf{u_2}, \cdots, \mathbf{u_n}\}$ basis, $\mathbf{u}$ a vector in U, and $a_1, a_2, \cdots, a_n$ scalars such that $\mathbf{u} = a_1\mathbf{u_1} + a_2\mathbf{u_2} + \cdots + a_n\mathbf{u_n}$.

The column vector $\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$ is called the *coordinate vector of* $\mathbf{u}$ releative to B,[27] and $a_1, a_2, \cdots, a_n$ are called the *coordinates of* $\mathbf{u}$.[28]

[27] coordinate vector

[28] coordinates

Column vectors are more convenient to work with when developing the theory. In the program code, no distinction is made. The same vector can be used either as a column or as a row vector.

The standard bases are the most convenient bases to work with. To express a vector in coordinates relative to another base, we need to solve a system of linear equations. One special case is *orthonormal basis*[29]. In that case, the coordinate vector is $\mathbf{v}_B = \begin{bmatrix} \mathbf{v} \cdot u_1 \\ v \cdot u_2 \\ \vdots \\ v \cdot u_n \end{bmatrix}$.

[29] orthonormal basis

Change of Basis

In general, if B and B' are bases of U, and $\mathbf{u}$ is a vector in U having coordinate vectors $\mathbf{u_B}$ and $\mathbf{u_{B'}}$ relative to B and B', then $\mathbf{u_{B'}} = P\mathbf{u_B}$, where P is a *transition matrix* from B to B'. $P = [(\mathbf{u_1})_{B'} \cdots (\mathbf{u_n})_{B'}]$.

The change from a nonstandard basis to the standard basis is easy to do. The columns of P are the columns of the first basis, which is illustrated with the following example.[30]

[30] Example 4 on page 294 in the textbook.

```
(let [b (dge 2 2 [1 3
                  2 -1]
             {:layout :row})
      b' (dge 2 2 [1 0
                   0 1]
              {:layout :row})
      p (copy b)]
  (mv p (dv 3 4)))

=>
#RealBlockVector[double, n:2, offset: 0, stride:1]
[  15.00    2.00 ]
```

A more interesting case is when neither basis is standard. In that case, we can use the standard basis as an intermediate basis. Let B and B' be bases, S the standard basis, and P and P' transition matrices

to the standard basis. Then, transition from B to B' can be accomplished by transition matrix $P'^{-1}P$, and from B' to B with $P^{-1}P'$.

Here is an illustrative example.[31]

[31] Example 5, page 295 in the textbook.

```
(let [p (dge 2 2 [1 3
                  2 -1]
             {:layout :row})
      p' (dge 2 2 [3 5
                   1 2]
              {:layout :row})
      p-1 (tri (trf p))
      p'-1 (tri (trf p'))]
  (mv (mm p'-1 p) (dv 2 1)))

=>
#RealBlockVector[double, n:2, offset: 0, stride:1]
[  -5.00    4.00 ]
```

This wraps up the usual basics covered by linear algebra textbooks.