Numerical Linear Algebra for Programmers

Dragan Djuric

SAMPLE CHAPTER

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

DRAGAN DJURIC

NUMERICAL LINEAR ALGEBRA FOR PROGRAMMERS [SAMPLE 1.0.0]

DRAGAN ROCKS

Please check other books from the *Interactive Programming for Artificial Intelligence* book series at `https://aiprobook.com`

This book is available at `https://aiprobook.com/numerical-linear-algebra-for-programmers`. Subscribe to the Patreon campaign, at https://patreon.com/linear_algebra, and get access to all available drafts of further versions, and a number of nice perks.

All proceeds go towards funding the author's work on the Uncomplicate software libraries: Please check out `https://uncomplicate.org` and `https://github.com/uncomplicate`.

PUBLISHED BY DRAGAN ROCKS

HTTPS://AIPROBOOK.COM

Current version, 9 November 2021

This book would not have been possible without your support. Thank you!

Stuart Halloway, Bobby Calderwood, Erich Oliphant, Nolan, Kieran Owens, Carlton Schuyler, Jacob Rahme, (anon), Philip Cooper, Forrest Galloway, Alex Lykostratis, Tobi Lehman, Eric Ihli, Chuck Cassel, Alex Gian, Alan Thompson, ka yu Lai, Paul Boeschoten, Gurvesh Sanghera, Dmitri, Mario Trost, Jacob Maine, Brian Abbott, michiakig, Stepan Parunashvili, Richard Wofford, Andrew Dean, Pixel Pie, Anders Murphy, Frank Elliott, Stephen Cagle, Chris Curtis, David Hoyt, Dwayne, Al King, Timo Kauranen, and many others (list continues at the back of the book).

I would also like to thank Clojurists Together and Cognitect for supporting my work on the Uncomplicate libraries with generous grants.

Contents

Getting started

Introduction

Writing the first sentence is often the hardest part. Not this time. Likewise, kick-starting the understanding of an unfamiliar topic might be the steepest point in any programming journey. In hindsight, I'd love that I had learned Lisp and functional programming earlier, because now it seems so elegant and logical. But, something had to happen to nudge me to notice what was, from this perspective, the right tool for me. Clojure helped me leave the old ways behind, and embrace what was new.

Programming with linear algebra is somewhat similar in this regard. You might be hearing about it, you probably even know the basics from your math education, but you simply can't see how to use it in daily programming. You might even have tried some software libraries, but haven't progressed beyond using them as fancy arrays. This book will help you take that first step right away, and - if you take it - be your companion in a path that leads quite far away.

This text is imperfect, but useful. However, it's only the first version and, like any good software, is going to be developed continuously. I see a stream of improvements going far into the future, with milestone versions 2, 3, etc. Rather than the first edition, it's version one-point-zero.

What?

Exactly what the title "Numerical Linear Algebra for Programmers: An Interactive Tutorial with GPU, CUDA, OpenCL, MKL, Java, and Clojure" (Figure 1) says.

This book is not a math book on linear algebra or numerical mathematics; there is no shortage of such books. This books concentrates on how to learn enough linear algebra to apply it in everyday programming practice, mainly in machine learning and high performance computing. This means that this book is written *for programmers*, rather than mathematicians.

Every single line of code in this book is written in Clojure[1], a modern dialect of Lisp that compiles to Java bytecode and runs on Java Virtual Machine (JVM). I assume that you're already proficient in Clo-

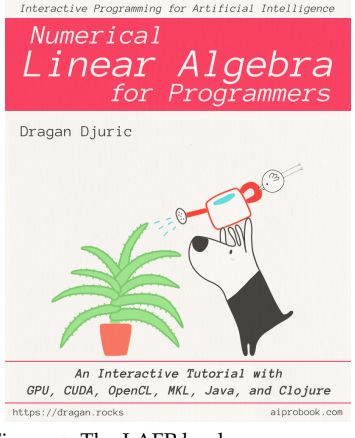

Figure 1: The LAFP book

[1] https://clojure.org

jure, but if you are not, the basics are fairly easy to learn[2].

Since our programs will run on a broad choice of exotic hardware, more notably GPUs, we'll have to work with software platforms such as OpenCL, or Nvidia's CUDA, in addition to the Java platform. These platforms are notoriously tricky to learn and use. Fortunately, I managed to hide them under the hood. You'll only need to learn to control them through Clojure, in a similar way you control the JVM. The only programming language we'll use is Clojure, and, yet, we get the full speed and power of these platforms.

[2] There is a free online book "Clojure for the Brave and True" at `https://brave-clojure.com`, and if you prefer a deeper text, `https://clojure.org/books` is a good place to browse.

How?

The title suggests that this is *An Interactive Tutorial*. Rather than throwing a pile of in-depth theory and specification at you, it assumes that you have a proper math book with you, and shows you how each math concept translates to code. Then, it shows you how these concepts are implemented and applied in high-performance settings. Finally, we look at how to implement several popular algorithms related to data analysis, statistics, and machine learning. The first three parts of the book should be read in order, since each chapter builds on what has been learned earlier. The chapters in the fourth part are more self-contained.

Thanks to Clojure and its REPL[3], this tutorial is *interactive*: as soon as I show you a line of code, you can evaluate it, and see the result *immediately*, even when the program is far from complete. There are no restarts, debug statements, `println`, nor carpets of boilerplate code.

[3] Short from Read Evaluate Print Loop.

You start the Clojure's JVM *once*, and work on the pieces of the program *while it is running all the time*, including the code that runs on *GPU*. If you're already familiar with what I'm talking about, great; but if that's not the case, I strongly advise that you keep your mind open and look around for demonstrations of Clojure's REPL[3] oriented programming. I recommend using Emacs + CIDER, but there are other tools that you might prefer. I don't care, as long as you are using the REPL-oriented process, rather than emulating interactive Python console[4].

[4] You might expect that Clojure REPL workflow is something like you've seen in Python, Ruby, and JavaScript ecosystems, but it's *not*; don't skip this.

Why?

Simply, linear algebra can help solve many programming problems with great elegance and performance. However, programmers, outside of a few specialized areas, side step around it. Although they learned some linear algebra in school, they usually didn't go far with understanding how to apply it, and many had forgotten what they had learn.

When?

Wherever you are in your Clojure journey, you can start right now! In that regard, the book is self-sufficient, when paired with a traditional linear algebra math textbook.[5] We start from very basic things without assuming much about your math background, other than high school math.

[5] I'll give my recommendation in Part 2.

The Interactive Programming for Artificial Intelligence series

Fine - you might think - you'll show me many nice examples, but can I use this to create bigger, serious stuff? Do I intend to show elegant, pure problems, and leave just before anyone asks how to create something of real substance? Good question, which I answer with more books!

Deep Learning for Programmers

"Deep Learning for Programmers: An Interactive Tutorial with CUDA, OpenCL, DNNL, Java, and Clojure" (Figure 2) implements a full, super-fast, deep learning library from scratch!

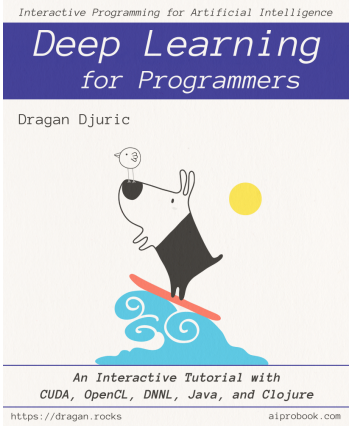

Figure 2: The DLFP book

In short, you're going to learn:

- the principles behind deep learning (and machine learning)
- only the necessary math and theory
- how to translate ML theory to code, step-by-step
- how to apply these techniques to implement neural networks from scratch
- how to implement other algorithms using vectors and matrices
- how to encapsulate these with elegant interfaces
- how to integrate with high-performance vendor libraries
- the nuts and bolts of a tensor-based deep learning library
- how to write simple, yet fast, number crunching software
- generally applicable high-performance programming techniques

Rather than being constrained to deep learning only, I hope that it opens a real-world portal to *high-performance scientific computing*.

You can check the contents of the DLFP book, available at `https://aiprobook.com/deep-learning-for-programmers/` and see whether it could be a good fit for you (there's a free sample).

The Interactive Programming Book

This is the book that teaches the programming tools and the software development process. Although it is not a machine learning book, it teaches fundamental skills necessary for effective application of what the other books teach. It also complements the books on Clojure, which teach you the language itself, but not how to hold the pencil and how to press it to the paper. Figure 3 shows the cover page that I have already designed, but I'm yet to write it. Please check my blog `https://dragan.rocks` for news and updates. Once the first drafts are ready, they'll be available at `https://aiprobook.com`.

Other books!

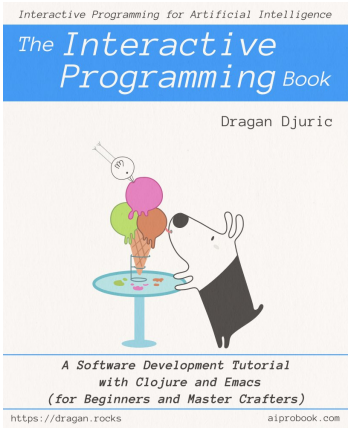

Figure 3: The Interactive Programming Book (TIPB)

And that is not all! Two books (DLFP and LAFP) are already here, at version 1.0, one (TIPB) is in my mind, about to be started, but I plan to write at least a few more. One book will teach GPU programming that will cover CUDA and OpenCL. In DLFP and LAFP we only use GPU, there we will learn how to write custom GPU programs. I expect that to be fairly thin, 150-200 pages.

Yet another will be a from-basics-to-GPU tutorial about probability and probabilistic data analysis. That's a tricky subject, so it will take some time, and may grow to be even thicker than DLFP. I hope that I'll be able to interest enough readers and subscribers so these two books can see the light of day, too.

So, that's 5 books in total planned so far. I'm looking forward to see Clojure with such a strong covering of these tricky topics!

Beyond that, it's too early to say, but who knows...

Let's go

So, check out the Appendix if you need to set up the libraries that we use in this book, take a look at the TOC to get a feel of what we're about to cover, and enjoy reading the first chapter without further delay.

Please do not hesitate to share any thoughts publicly online (and help spreading the word). I'm interested in both what's good and what should be improved. I'm learning from you, too!

Hello world

Motivation

As common with programming books, we'll start with a Hello World example right away. I won't dwell much on the theory, here, or later in the book – there are plenty of books that do that. I'll try to guide you through the applications, so you can discover yourself whatever can make your programming life easier. Here's the namespace definition that we'll use in this chapter. Let's go!

```
(ns dragan.rocks.lafp.part-1.hello-world
  (:require [uncomplicate.commons.core :refer [with-release]]
            [uncomplicate.fluokitten.core :refer [foldmap]]
            [uncomplicate.clojurecl.core :as opencl]
            [uncomplicate.clojurecuda.core :as cuda]
            [uncomplicate.neanderthal
             [core :refer [dot copy asum copy! row mv mm rk axpy entry!
                           subvector trans mm! zero]]
             [vect-math :refer [mul]]
             [native :refer [dv dge fge]]
             [cuda :refer [cuv cuge with-default-engine]]
             [opencl :as cl :refer [clv]]
             [random :refer [rand-uniform!]]]
            [criterium.core :refer :all]))
```

Just an ordinary domain model

Imagine this simplified[6] code for inventory modeling.

```
(def products {:banana {:price 1.3 :id :banana}
               :mango {:price 2.0 :id :mango}
               :pineapple {:price 1.9 :id :pineapple}
               :pears {:price 1.8 :id :pears}})
```

This (imaginary) application exists to track sales. Our customer puts the desired products into a cart, we calculate the total price,

[6] When I say "simplified" i mean *really* simplified. We're using floats for prices (bad), we store the data in global state, the architecture is far from even a simple web application. But the model is familiar enough to a typical software developer.

and, later, perform the delivery. Each cart only stores the products' identifiers and quantities (in unspecified units[7]).

[7] Yes, it's super-simplified.

```
(def cart1 {:banana 10
            :pineapple 7
            :pears 3})

(def cart2 {:pineapple 3
            :mango 9})
```

Having defined product and cart data, we write a function that, given the products "database" and a cart, calculates the total price of the products in the cart. The `cart-price` function reduces all `[product quantity]` pairs in the `cart`, by retrieving the appropriate product map in the `product-db`, and taking the value associated with its `:price` key. It multiplies that price with the quantity, and accumulates it in `total`.

```
(defn cart-price [product-db cart]
  (reduce (fn [total [product quantity]]
            (+ total (* (:price (product-db product)) quantity)))
          0
          cart))
```

Let's call this function with the available carts, and see it in action.

```
(cart-price products cart1)

=> 31.699999999999996

(cart-price products cart2)

=> 23.7
```

We hopefully have more than one order. Our code can easily process sequences of carts, and compute the total revenue.

```
(reduce + (map (partial cart-price products) [cart1 cart2]))

=> 55.39999999999999
```

It's all good; but what does it have to do with linear algebra?

A more general algorithm

In the previous implementation, we entangled the specifics of data storage and the algorithm that computes the total price. In this simple model, it's not much of a problem, but if the data model is more complex, and the algorithm not as simple as the straightforward map/reduce, this quickly leads to (at least) two problems:

- code becomes too complicated
- program performance degrades quickly

Let's first tackle the code complexity by extracting the computation logic from the domain into the abstract mathematical notion of vectors and operations on these vectors.[8] In this particular example, vectors help us encapsulate a bunch of numbers as one atomic unit.

[8] Part 2 of the book is dedicated to the relevant theory, and helps you connect the theoretical foundation to the code you need to invent and develop.

```
(def product-prices [1.3 2.0 1.9 1.8])
(def cart-vec-1 [10 0 7 3])
(def cart-vec-2 [0 9 3 0])
```

We recognize that the logic we've already developed for computing the total price matches a simple and well known mathematical operation, known as the *dot product*, a scalar product of two vectors.[9]

[9] More explanation comes right after this example.

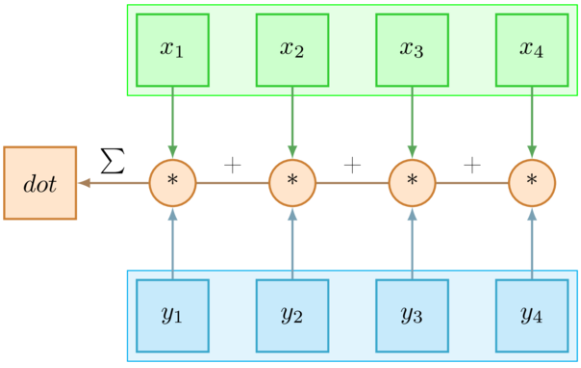

Figure 4: An informal block diagram showing how dot product is computed.

```
(defn dot-product-vec [xs ys]
  (reduce + (map * xs ys)))
```

Given two vectors, `[1 2 3]` and `[4 5 6]`, the dot product computes one number, a *scalar*, that represent a *scalar* product of these two vectors.[10] Right now, we don't even care about theoretical details of the dot product; we recognize that it technically computes the same thing that we need in our domain, and it seems useful.

[10] There are other ways to multiply vectors, which return non-scalar structures.

```
(dot-product-vec [1 2 3] [4 5 6])

=> 32
```

We can see that, when applied to the vectors holding product prices and quantities, it returns the correct results that we've already seen.

```
(dot-product-vec product-prices cart-vec-1)

=> 31.699999999999996
```

```
(dot-product-vec product-prices cart-vec-2)
```

```
=> 23.7
```

Getting the total price requires another map/reduce, but we will quickly see that this, too, can be generalized.

```
(reduce + (map (partial dot-product-vec product-prices)
               [cart-vec-1 cart-vec-2]))
```

```
=> 55.39999999999999
```

A library of linear algebra operations

When we abstract away the specifics of the domain, we end up with a number of general operations that can be reused over and over, and combined into more complex, but still general, operations. Countless such operations have been studied and theoretically developed by various branches of mathematics and related applied disciplines for a long time. What's more, many have been implemented and optimized for popular hardware and software ecosystems, so our main task is to learn how to apply that vast resource to the specific domain problems.

Linear algebra is particularly well supported in implementations. Whenever we need to process arrays of numbers, it is likely that at least some part of this processing, if not all of it, can be described through vector, matrix, or tensor operations.

Vectors

Instead of developing our own naive implementations, we should reuse the well-defined data structures and functions provided by Neanderthal.

Here we use vectors of double precision floating point numbers to represent products' prices and carts.

```
(def product-prices (dv [1.3 2.0 1.9 1.8]))
(def cart-vctr-1 (dv [10 0 7 3]))
(def cart-vctr-2 (dv [0 9 3 0]))
```

We use the general `dot` function in the same way as the matching function that we had implemented before.

```
(dot product-prices cart-vctr-1)
```

```
=> 31.7
```

```
(dot product-prices cart-vctr-2)
```

```
=> 23.7
```

Matrices

Once we start applying general operations, we can see new ways to improve our code, not so obvious at first.

Instead of maintaining sequences of vectors that represent carts, and coding custom functions to process these vectors, we can put that data into the rows of a matrix. All carts are now represented by one matrix, and each row of the matrix represents one cart.

```
(def carts (dge 2 4))

=>
#RealGEMatrix[double, mxn:2x4, layout:column, offset:0]
┌       ↓       ↓       ↓       ↓       ┐
→       0.00    0.00    0.00    0.00
→       0.00    0.00    0.00    0.00
└                                       ┘
```

We could have populated the matrix manually, but, since we already have the data loaded in appropriate vectors, we can copy it, showing how these structures are related.

```
(copy! cart-vctr-1 (row carts 0))

=>
#RealBlockVector[double, n:4, offset: 0, stride:2]
[  10.00    0.00    7.00    3.00 ]

(copy! cart-vctr-2 (row carts 1))

=>
#RealBlockVector[double, n:4, offset: 1, stride:2]
[   0.00    9.00    3.00    0.00 ]
```

The following step is the usual opportunity for a novice to slip. Should we now iterate the rows of our newly created matrix, calling dot products on each row? No! We should recognize that the equivalent operation already exists: matrix-vector multiplication, implemented by the `mv` function![11]

```
(mv carts product-prices)

=>
#RealBlockVector[double, n:2, offset: 0, stride:1]
[  31.70   23.70 ]

(asum (mv carts product-prices))

=> 55.4
```

[11] Most functions in this domain have short names that might sound cryptic until you get used to it. There is a method to their naming, though, and they are usually very descriptive mnemonics. For example, `mv` stands for Matrix-Vector multiplication. You'll guess that `mm` is Matrix-Matrix multiplication and so on. Like in mathematical formulas, this naming makes for code that can be viewed in a contained place that can be grasped in one view.

Not only that the `mv` operation is equivalent to multiple calls to `dot`, but it takes advantage of the structure of the matrix, and optimizes the computation to the available hardware. This achieves much better performance, which can compound to orders of magnitude in improvements.[12]

[12] These improvements materialize in more serious examples. Any implementation of a small toy problem works fast.

. . . and more

Let's introduce a bit more complication. Say that we want to support different discounts for each product, in the form of multipliers. That gets us the price reductions, that we should subtract from the price. An alternative way, shown in the following snippets is to subtract the discount coefficients from `1.0` to get the direct multiplier that gets us to the reduced price.

```
(def discounts (dv [0.07 0 0.33 0.25]))
(def ones (entry! (dv 4) 1))
```

We can subtract two vectors by the `axpy` function.[13]

[13] `axpy` stands for "scalar a times x plus y".

```
(axpy -1 discounts ones)

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[   0.93    1.00    0.67    0.75 ]
```

The `mul` function multiplies its vector, matrix, or tensor arguments element-wise, entry by entry.

```
(mul (axpy -1 discounts ones) product-prices)

=>
#RealBlockVector[double, n:4, offset: 0, stride:1]
[   1.21    2.00    1.27    1.35 ]
```

The following code seamlessly incorporates this new part of the algorithm into the implementation that we already have.

```
(asum (mv carts (mul (axpy -1 discounts ones) product-prices)))

=> 46.87
```

Why stop here? Suppose that we want to simulate the effects of multiple discount combinations on the total price. As earlier, we put all these hypothetical discount vectors into a matrix, in this case, three samples that we'd like to investigate.

```
(def discount-mat (dge 4 3 [0.07 0 0.33 0.25
                            0.05 0.30 0 0.1
                            0 0 0.20 0.40]))
```

```
=>
#RealGEMatrix[double, mxn:4x3, layout:column, offset:0]
┌       ↓       ↓       ↓       ┐
→       0.07    0.05    0.00
→       0.00    0.30    0.00
→       0.33    0.00    0.20
→       0.25    0.10    0.40
└                               ┘
```

We have to subtract these numbers from `1.0`. Instead of populating the matrix with `1.0`, we will demonstrate the outer product operation, implemented by the function `rk`. Given two vectors, it produces a matrix that holds all combinations of the product of the entries of the vectors.

```
(rk ones (subvector ones 0 3))
```

```
=>
#RealGEMatrix[double, mxn:4x3, layout:column, offset:0]
┌       ↓       ↓       ↓       ┐
→       1.00    1.00    1.00
→       1.00    1.00    1.00
→       1.00    1.00    1.00
→       1.00    1.00    1.00
└                               ┘
```

We can also utilize `rk` to "lift" product prices vector to a matrix whose shape matches the shape of the discount combinations matrix.

```
(def ones-3 (subvector ones 0 3))
(def discounted-prices (mul (axpy -1 discount-mat (rk ones ones-3))
                            (rk product-prices ones-3)))
```

The result is a matrix of hypothetical discount prices that we'd like to simulate.

```
=>
#RealGEMatrix[double, mxn:4x3, layout:column, offset:0]
┌       ↓       ↓       ↓       ┐
→       1.21    1.23    1.30
→       2.00    1.40    2.00
→       1.27    1.90    1.52
→       1.35    1.62    1.08
└                               ┘
```

Now, the most interesting part: how do we calculate the totals from this matrix and the matrix of carts we've produced earlier. (Not) surprisingly, just a *single* operation, matrix multiplication, completes this task!

```
(mm carts discounted-prices)

=>
#RealGEMatrix[double, mxn:2x3, layout:column, offset:0]
┌       ↓       ↓       ↓       ┐
→      25.05   30.51   26.88
→      21.82   18.30   22.56
└                               ┘
```

Now we only need to sum the columns up to get the three final totals. We won't do this column-by-column. Instead, we'll use the "mv with ones" approach we've already encountered. Note that we need to transpose the matrix to match the desired structure.

```
(trans (mm carts discounted-prices))

=>
#RealGEMatrix[double, mxn:3x2, layout:row, offset:0]
          ↓       ↓       ┐
   →      25.05   21.82
   →      30.51   18.30
   →      26.88   22.56
   └                       ┘
```

And, the final answer is…

```
(mv (trans (mm carts discounted-prices)) (subvector ones 0 2))

=>
#RealBlockVector[double, n:3, offset: 0, stride:1]
[  46.87   48.81   49.44 ]
```

Given three (or three million) possible discount combinations, we get a vector of the total revenue amounts. Of course, being a toy example, this code doesn't take into account that lower prices would (likely) induce more sales; let's not carry a Hello World example too far.

So, the first major benefit of using a library, such as Neanderthal, based on many decades of numerical computing research and development is that *we have access to a large treasure trove of useful, well thought, general functions for developing useful, general or customized, number processing algorithms*.

Performance improvements

Another major improvement is performance. Although toy examples may be implemented in any way you'd like, and they'd still work reasonably well, real-world data processing almost always involves either many data points, or many computation steps, or, often – both.

To show you that I'm not talking about a couple dozen percentages, but improvements of many orders of magnitude, I'll demonstrate a few simple core operations. Note that these operations are only the building blocks of your algorithm, so these improvements often compound.

We start with the simplest case: dot product over two mildly large vectors of 100k entries. First, we measure the running time of the Clojure vector-based `dot-product-vec`.[14]

[14] These measurements use an old CPU from 2013, i7-4790k.

```
(def x-vec (vec (range 100000)))
(def y-vec (vec (range 100000)))

(with-progress-reporting (quick-bench (dot-product-vec x-vec y-vec)))

Execution time mean : 15.843749 ms
```

15 milliseconds doesn't seem that much. After all, it's faster than a blink of an eye. Remember - you never need *one* dot product, but, as in our Hello World demo, many of them.

Still staying in Clojure, we can improve this by manually looping, and casting numbers from the Java's `Number` into primitives. This alone makes the code more complicated, but we reduced the running time 7-fold.

```
(defn dot-product-loop [xs ys]
  (loop [res 0.0
         x (first xs) xs (next xs)
         y (first ys) ys (next ys)]
    (if (and x y)
      (recur  (+ res (* (double x) (double y)))
              (first xs) (next xs)
              (first ys) (next ys))
      res)))
(with-progress-reporting (quick-bench (dot-product-loop x-vec y-vec)))

Execution time mean : 2.234458 ms
```

Similar speedup can be achieved using Fluokitten, Neanderthal's cousin library. Fluokitten is implemented in pure Clojure, so it cannot go beyond Clojure speed, but it can reduce the complexity a lot.

```
(with-progress-reporting (quick-bench (foldmap + 0.0 * x-vec y-vec)))

Execution time mean : 4.054223 ms
```

The next step is to delve into primitive Java arrays. The code is somewhat low-level, but the speed is much better since the data is arranged in cache-friendly consecutive memory locations (most of the time). The calculations are done with primitives, too.

```
(defn dot-product-arr ^double [^doubles xs ^doubles ys]
  (let [n (alength xs)]
    (loop [i 0 res 0.0]
      (if (< i n)
        (recur (inc i) (+ res (* (aget xs i) (aget ys i))))
        res))))
(def x-arr (double-array (range 100000)))
(def y-arr (double-array (range 100000)))
(with-progress-reporting (quick-bench (dot-product-arr x-arr y-arr)))

Execution time mean : 79.123590 µs
```

This is great improvement! 200 times faster than the first snippet! Even C code that one would write would not be faster, and it is likely to even be slower, if you use only vanilla C.

How does Neanderthal fare?

```
(def x (dv (range 100000)))
(def y (copy x))
(with-progress-reporting (quick-bench (dot x y)))

Execution time mean : 8.574476 µs
```

This code is executed on 4 cores, and is an order of magnitude faster. Of course, if executed on 1 core it would be only twice as fast. But, remember that `dot` is a very simple algorithm; there's not much to squeeze the difference from.

This is all the speedup that we can achieve on the CPU. Numerical code should be a great match for a powerful GPU, right? The following computation runs on Nvidia GTX-1080Ti.

```
(cuda/with-default
  (with-default-engine
    (with-release [gpu-x (cuv (range 100000))
                   gpu-y (cuv (range 100000))]
      (with-progress-reporting (quick-bench (dot gpu-x gpu-y))))))

Execution time mean : 22.697660 µs
```

This is surprisingly slow! It is faster than pure Java, but not even as fast as an old Intel processor! Is something wrong with Nvidia? Let's try AMD's Vega 64 with an open source OpenCL engine.

```
(opencl/with-default
  (cl/with-default-engine
    (with-release [gpu-x (clv (range 100000))
                   gpu-y (copy gpu-x)]
      (with-progress-reporting (quick-bench (dot gpu-x gpu-y))))))

Execution time mean : 74.070430 µs
```

It is roughly as fast as the Java implementation. Where's the catch? The catch is in the simplicity of the `dot` operation. GPU is a sort of a nuclear aircraft carrier of computing. It does not pay to move it for catching a bucket of shrimps. This is finely demonstrated with matrix multiplication, which, although it can logically be implemented with three simple nested loops, has quadratic complexity, and requires repeated read/write pattern that stresses the cache.

The following implementation is adapted from Rosetta code.[15]

[15] Adapted from Rosetta Code (realize seqs with `doall`).

```
(defn matrix-mult-vec [a b]
  (let [transpose (fn [s]
                    (doall (apply map vector s)))
        nested-for (fn [f x y]
                     (doall (map (fn [a]
                                   (doall (map (fn [b]
                                                 (f a b))
                                               y)))
                                 x)))]
    (nested-for (fn [x y] (reduce + (map * x y)))
                a
                (transpose b))))
```

Here's a demonstration of its functionality. Multiplying a matrix by itself, we get another matrix of appropriate dimensions.

```
(def small-mat [[1 2 3 4]
                [5 6 7 8]
                [9 19 11 12]
                [13 14 15 16]])
(matrix-mult-vec small-mat small-mat)

=>
((90 127 110 120)
```

```
 (202 291 254 280)
 (359 509 461 512)
 (426 619 542 600))
```

This result is not interesting now. We want to see performance on a modestly sized matrix first; let's say 256×256.

```
(def ma-vec (vec (take 256 (cycle [(take 256 (cycle [1 2 3 4]))]))))
(def mb-vec (vec (take 256 (cycle [(take 256 (cycle [4 3 2 1]))]))))
(with-progress-reporting (quick-bench (matrix-mult-vec ma-vec mb-vec)))
```

```
Execution time mean : 2.304788 sec
```

That's more than 2 *seconds*!

We compare this with the equivalent matrix in Neanderthal. For demonstration, we'll also use single precision floats instead of double precision floats, which, typically, increases the speed by 2×.[16]

[16] If you wonder why we didn't use single precision in Clojure's implementation, it's because Clojure only supports double precision primitive numbers in functions.

```
(def a (rand-uniform! (fge 256 256)))
(def b (rand-uniform! (fge 256 256)))
(def c (fge 256 256))
(with-progress-reporting (quick-bench (mm a b)))
```

```
Execution time mean : 129.143314 µs
```

You see it well: 130 *micro* seconds! That's *17692 times* faster!

That's not all. In most algorithm implementations, it is recommended to use the destructive versions of the internal computations. By avoiding unnecessary memory allocations we considerably speed the code up.

```
(with-progress-reporting (quick-bench (mm! 1 a b 0 c)))
```

```
Execution time mean : 105.646650 µs
```

This is 21904 times faster than what we started with! Although still simple, this code should be able to show the advantage of GPU's.

```
(cuda/with-default
  (with-default-engine
    (with-release [gpu-a (rand-uniform! (cuge 256 256))
                   gpu-b (rand-uniform! (cuge 256 256))
                   gpu-c (zero gpu-a)]

      (with-progress-reporting
        (quick-bench
         (do (mm! 1.0 gpu-a gpu-b 0.0 gpu-c)
             ;;GPU calls are asynchronous, measure with finish!
             (cuda/synchronize!)))))))
```

```
Execution time mean : 14.806265 µs
```

At last, GPU is at least order of magnitude faster than CPU, which is the least we'd expect for the trouble of dealing with it, even considering that Neanderthal will automate almost all of that.

I should remind you that 256×256 is not particularly impressive size for a matrix. As a rule of thumb, the bigger the size, the more advantage there is for the GPU. 8192×8192? No problem for Neanderthal, and this is something that you'll see in real use cases.

```
(cuda/with-default
  (with-default-engine
    (with-release [cpu-a (rand-uniform! (fge 8192 8192))
                   cpu-b (rand-uniform! (fge 8192 8192))
                   cpu-c (zero cpu-a)
                   gpu-a (rand-uniform! (cuge 8192 8192))
                   gpu-b (rand-uniform! (cuge 8192 8192))
                   gpu-c (zero gpu-a)]
      (time (mm! 1.0 cpu-a cpu-b 0.0 cpu-c))

      (with-progress-reporting
        (quick-bench
         (do (mm! 1.0 gpu-a gpu-b 0.0 gpu-c)
             ;;GPU calls are asynchronous, measure with finish!
             (cuda/synchronize!)))))))
```

```
CPU: "Elapsed time: 2864.339836 msecs"
GPU: "Execution time mean : 112.954961 ms"
```

One monster against the other; they're both unbelievably fast, but now one is 25 times faster. I recommend that you try this with any Clojure/Java library to just appreciate what we got. I'd do that, but I'm afraid that I'd have to delay the book, waiting for such computation to complete.

At the end of this introduction, I invite you to think about what even a 100 or 1000 times improvement in speed mean. Of course, we rarely care whether the complete computation is 1 microsecond or 1 millisecond – both complete before we even notice that they started. But, for the building blocks, that are part of a much larger program, that means that, instead of 1000 minutes we get results in 1 minute. And, 1000 minutes is... almost 17 hours.

Our algorithm can be even more complex. Let's say it takes 2 hours. If a less optimized algorithm is just 100 times slower (which is likely) it would take 200 hours - more than a whole week!

Vectors, matrices, and Neanderthal API

Neanderthal's goal is to provide fast and simple tools for computationally demanding problems in Clojure. That alone can mean anything and everything, so I'll narrow it by being more tech-specific. Neanderthal primarily concentrates around arrays, vectors, and matrices. It aims at providing efficient structures that represent these concepts on modern hardware, and efficient operations to transform them. Presently, that usually means using raw primitive arrays, and integrating with raw native libraries provided by hardware vendors.

We could use Clojure's pure structures to represent vectors, matrices, and tensors, and we could implement practically any operation in pure Clojure, on the JVM. The resulting functions would probably be nice and elegant. The trouble is that they would not be practical at all, and would have awfull performance beyond toy programs. The real world is ugly. The only way to get acceptable performance is to closely tailor such data structures and algorithms to specific hardware. It's complex and expensive, so the best solutions are usually provided by hardware vendors themselves.

Since we have to use these dirty and complex native programs, could we at least aim to hide their complexity behind a nice Clojure API? Absolutely! This chapter gives a high level overview of how Neanderthal does it.

Creation

To compute anything, we first have to provide data. Neanderthal works with various kinds of vectors and matrices[17]. The next part of the book covers the proper mathematical theory that will help you understand what vectors and matrices *are*. In this part, we'll look at them in a crude mechanistic manner and say that vectors are a sort of one dimensional arrays, while matrices can be two dimensional arrays, if we squint[18].

Since Neanderthal's function are polymorphic, the type of a object implicitly controls which underlying engine, ie. implementation, will perform the computation. The most important characteristic of an object, besides its logical type (one of available vector or matrix types) is

[17] If you're interested in n-dimensional arrays, aka tensors, please see the Deep Diamond library and the Deep Learning for Programmers book.

[18] I do that because it's how most programmers would view vectors and matrices at first, but it's a wrong and constraining way to think about these concepts in general.

the memory where it resides. The default, and the most accessible, is the main memory that the CPU uses transparently, without much effort by the programmer. Additionally, we might want to store our objects and run computations on the GPUs and other accelerators attached to our computer. This requires more knob twiddling, most of which Neanderthal does under the hood for us, that we have to be aware of.

Core constructor functions

The `uncomplicate.neanderthal.core` package contains technology-agnostic constructor functions. The name of the function determines the kind of structure to be created: vector, general matrix, symmetric matrix, triangular matrix, packed matrix, etc. These functions do, of course, create concrete, technology-specific objects, but that is not explicitly configured. The underlying technology is defined by the `fact` argument, standing for *factory provider*, which polymorphically delegates the call to the proper technology specific implementation. In the remaining of that object's life, the polymorphic calls to actual operations will be transparently directed to the proper engine.

Take the simplest structure, dense vector, and its constructor, `vctr`, for example.

We can try to create a dense vector with nothing more than a few numbers.

```
(vctr 1 2 3)
```

```
=> No implementation of method: :factory of protocol: #'uncomplicate.neanderthal.internal.api/FactoryProvi
```

Unfortunately, this information is not enough. A number, plain Integer in this case, does not implement the `FactoryProvider` protocol. We could have configured the default engine, which could decide from the information we have that we would like to create a primitive vector of `int` in the main memory, but that assumption is often incorrect; in these types of applications, we usually use floating point. OK, then, why the default is not a floating point? It could be, but then we would still miss the information about precision (`double`, `float`, `half`, etc.), and the memory space (main memory, or memory attached to a device). Such default would also make it difficult to distinguish the desired role for the first argument: factory provider, or initial content. It would make the API more confusing than necessary, as in the following examples.

```
(vctr [1 2 3])
```

```
=> Wrong number of args (1) passed to: uncomplicate.neanderthal.core/vctr
```

```
(vctr [1 2 3] [1 2 3])
```

```
=> No implementation of method: :factory of protocol: #'uncomplicate.neanderthal.internal.api/FactoryProvi
found for class: clojure.lang.PersistentVector
```

So, the simple rule is that the first argument for a constructor function should be an object that implements the `FactoryProvider` protocol. The remaining arguments can provide object dimensions, and/or initial contents. Creating a vector of single-precision floats with dimension 3, $(1, 2, 3)$, in main memory, is straightforward.

```
(vctr native-float [1 2 3])
```

```
=>
#RealBlockVector[float, n:3, offset: 0, stride:1]
[   1.00    2.00    3.00 ]
```

Neanderthal provides factories for typical primitive numeric types that Java and Clojure support, fully, partially, or "kind of".[19]

[19] Naturally, to the level that the JVM and the backing low-level library allow, and the context in which these structures are typically used.

```
(vctr native-double [1 2 3])
```

```
=>
#RealBlockVector[double, n:3, offset: 0, stride:1]
[   1.00    2.00    3.00 ]
```

Operations on single and double precision floating point numbers are typically supported in all underlying performance libraries, while the support for other floating point types, such as half-precision floats, depends on the hardware. Integers are usually used for indexing, and aren't available for most computation operations.

```
(vctr native-int [1 2 3])
```

```
=> #IntegerBlockVector[int, n:3, offset: 0, stride:1](1 2 3)
```

If you wondered about why the contents of this structure is printed differently than in the past few examples, it's because printing floating point numbers is tricky with different scales and precision, while integers are straightforward, and we can simply print out their Clojure literal representation. For floats, Neanderthal supports the scientific notation, and prints only two significant decimals, since the printout serves primarily to get the feel of the overall data in the REPL anyway, not to inspect each of potentially millions of data points.

```
(def fx (vctr native-float [10000000000000 2 3]))
```

```
=>
#RealBlockVector[float, n:3, offset: 0, stride:1]
[1.00E+132.0     3.0     ]
```

The factory provider doesn't have to be the actual factory; it can be any of the Neanderthal structures. We would typically want to write code that would work on any platform (CPU, GPU, etc.), and would not provide a concrete factory, such as `native-float` upfront, but would probably receive it as an argument. However, passing that argument around is not always the most practical way to get a reference to a factory. What if we can already determine the factory from another argument? Almost all functions will work on some vectors and matrices that they receive as arguments, so we could use these as factory providers.

The `fx` vector that we have just created can be used as a sapling for creating other objects supported by the same backend technology.

```
(def x (vctr fx [1 2 3 4]))
```

```
=>
#RealBlockVector[float, n:4, offset: 0, stride:1]
[   1.00    2.00    3.00    4.00 ]
```

Constructor functions can recognize a wide range of structures that as data source. Clojure sequences, arrays, other vectors or matrices, and even varargs are supported.

```
(vctr x 1 2 3 4 5 6)
```

```
=>
#RealBlockVector[float, n:6, offset: 0, stride:1]
[   1.00    2.00    3.00          5.00    6.00 ]
```

We don't have to provide source data each time we create a vector; providing the dimension is enough to create a zero vector.

```
(vctr x 3)
```

```
=>
#RealBlockVector[float, n:3, offset: 0, stride:1]
[   0.00    0.00    0.00 ]
```

Constructors that create more complex structures typically require that we provided dimensions.

```
(ge x 2 3)
```

```
=>
#RealGEMatrix[float, mxn:2x3, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ┐
   →       0.00    0.00    0.00
   →       0.00    0.00    0.00
   └                               ┘
```

Of course, we are able to load the structure from a data source at the time of creation.

```
(ge x 2 3 [1 2 3 4 5 6 7 8])
```

```
=>
#RealGEMatrix[float, mxn:2x3, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ┐
   →       1.00    3.00    5.00
   →       2.00    4.00    6.00
   └                               ┘
```

Although Neanderthal tries to figure out as much information from the context, sometimes the context doesn't provide enough information. In the following example, we try to create a dense matrix from a vector, but we can't extract any meaningful dimension other than the trivial $n \times 1$, so Neanderthal complains that it's probably not a very useful thing to do.

```
(ge x x)
```

```
=>
Execution error (ExceptionInfo) at uncomplicate.commons.utils/dragan-says-ex (utils.clj:105).
Dragan says: This is not a valid source for matrices.
```

On the other hand, we can create a matrix from a matrix.

```
(ge x a)
```

```
=>
#RealGEMatrix[float, mxn:2x3, layout:column, offset:0]
┌       ↓       ↓       ↓       ┐
→       1.00    3.00    5.00
→       2.00    4.00    6.00
└                               ┘
```

Native objects (CPU)

Sometimes you don't want to write functions that support all backends. Perhaps you know that the specific functionality works only on specific hardware, such as the CPU, or you're exploring the problem in the REPL, and you want the code to run without any configuration.

Since all operations in the `core` namespace are polymorphic, it's enough to provide platform-specific constructors, which, for the MKL engine that runs on the CPU and uses the main memory, can be found in the `uncomplicate.neanderthal.native` namespace. For each constructor from the `core` namespace, there is a function in the `native` namespace, prefixed with the type of entries: `d` for double, `f` for float, `l` for long, `i` for int, `s` for short, `b` for byte.

For example, the `dge` function creates a double dense (general) matrix in main memory, and all subsequent calls are going to be computed by the native CPU engine, which is backed by MKL.

```
(dge 2 3)
```

```
=>
#RealGEMatrix[double, mxn:2x3, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ┐
   →       0.00    0.00    0.00
   →       0.00    0.00    0.00
   └                               ┘
```

OpenCL (CPU & GPU) objects

GPU backends are a bit more complex, because we need to supply the context and execution queue that will be used in computations. This diminishes the usefulness of removing the factory parameter, which handles the context and queue internally. This is solved by handling the factory as a dynamically bound `*opencl-factory*`.

Instead of type-specific functions, such as `dge`, or `fge`, the `uncomplicate.neanderthal.opencl` namespace provides functions prefixed by `cl`, such as `clv` (OpenCL vector) and `clge` (OpenCL general matrix). The following example creates an OpenCL matrix in the default computation context on the default OpenCL device (which is a GPU on my machine, but might be a CPU on your system).

```
(clojurecl/set-default-1!)
(cl/set-engine!)
```

In this example, we set the root binding to the default factory. Until we release these bindings, we can interactively experiment with OpenCL code, even if it's on GPU, which would typically require much more boilerplate.

```
(def gpu-a (clge 2 3 (range 7)))
```

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

Don't forget to release the context at the end. It releases all resources related to that context, including the backends. Please note that this is unrelated to the actual objects in the JVM, which might still exist. This releases the resources on the device.

```
(clojurecl/release-context!)
```

CUDA (Nvidia GPU) objects

Nvidia's CUDA works in a similar way: we provide the context and engine as a dynamic binding, create CUDA-specific vectors and matrices, and use normal Neanderthal functions transparently to the technology. In the following example, we use the `with-*` macros (also available for OpenCL) that take care of the lifecycle of all relevant objects automatically.[20]

[20] Constructors from the `core` namespace are the recommended way in most code, though.

```
(clojurecuda/with-default
  (with-default-engine
    (with-release [gpu-a (cuge 2 3 (range 7))]
      (asum gpu-a))))
```

```
=> 15.0
```

Moving data around

If we'd like to do useful things with vectors or matrices, by calling appropriate operations that transform data, we need to have some initial data to begin with. Since the object is usually full of zeroes at the time of creation, the next task is usually to move some numbers to that object.

Copying

The most efficient way to move data between two Neanderthal objects is copying, which is the preferred operation when these objects have the same structure, reside in the same memory space, on the same device. In that case, we should always use the `copy!` function, or its pure equivalet `copy` when we want to copy to a new object.

```
(def x-copy (copy x))
```

```
=>
#RealBlockVector[float, n:4, offset: 0, stride:1]
[   1.00    2.00    3.00    4.00 ]
```

Suppose we now change `x-copy`, and would like to move the final numbers back to `x`. The source and the destination already exist, and we use the destructive `copy!` function.

```
(scal! 2.0 x-copy)
(copy! x-copy x)
x
```

```
=>
#RealBlockVector[float, n:4, offset: 0, stride:1]
[   2.00    4.00    6.00    8.00 ]
```

Please note that the JVM is rather slow at allocating native buffers on the heap. We should strive to re-use objects and prefer destructive functions, suffixed by "!", whenever possible. This is especially important to keep in mind when we deal with temporary local objects that store intermediate results that are only used in the local scope of a few functions that implement the particular algorithm.

The `copy!` function requires matching objects. For example, it will refuse to copy a vector of insufficient length.

```
(def x-small (vctr x [100 200]))
(copy! x-small x)
```

```
=>
Execution error (ExceptionInfo)
Dragan says: You cannot copy data of incompatible or ill-fitting structures.
```

Neanderthal does support this use case. Copy to/from a matching subvector, which refers to the appropriate chunk of memory of the original object.

```
(copy! x-small (subvector x 0 2))
x
```

```
=>
#RealBlockVector[float, n:4, offset: 0, stride:1]
[ 100.00  200.00    6.00    8.00 ]
```

The arguments to copying functions do not have to be dense. Both `copy` and `copy!` support sparse vector and matrix structures, regardless of whether they originate from an explicit constructor, or refer to a smaller part of the contents of a larger object. For example, a row or a column of a matrix may have a stride, and we can copy it to a densely packed vector.

Transferring

A more general and versatile way of moving data around is the `transfer!` method. It does not require that the source and destination data structures match, that they reside in the same memory space, that the entries are of the same type, nor that there is enough entries in the source to fill the destination. It will simply try to fit the data that you provide into the destination, if it can find a viable path. It supports plain Clojure data structures, and can be extended to any source and destination combination that you want to support yourself!

```
(transfer! [-4 -5] x)

=>
#RealBlockVector[float, n:4, offset: 0, stride:1]
[  -4.00   -5.00    6.00    8.00 ]

(seq (transfer! x (double-array 8)))

=> (-4.0 -5.0 6.0 8.0 0.0 0.0 0.0 0.0)
```

Transferring a strided row of a double matrix from the main memory to a symmetric matrix of single precision floats on the GPU would be an example of a more complex operation. It works out of the box, but we should check whether it chooses the fastest path underneath.

```
(clojurecuda/with-default
  (with-default-engine
    (with-release [b-cuda (cusy 3)]
      (println b-cuda))))

=>
#CUUploMatrix[float, type:sy, mxn:3x3, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ┐
   →       0.00    *       *
   →       0.00    0.00    *
   →       0.00    0.00    0.00
   └                               ┘
```

If `transfer!` is so useful, why bother with the nagging `copy!` at all? Its advantages become disadvantages when we want to be sure that our code works as fast as possible, with minimal resources.

When we copy a float matrix to a matching float matrix, we don't use any addittional workspace, and the operation is executed by one call to a highly optimized native function on the device where the memory is physically located. That operation is usually optimal.

Copying is still quite fast, and does not require intermediate steps when the data is compatible, but there is no single native operation that we need. In these cases, Neanderthal transparently calls a more fine-grained operation multiple times to copy the appropriate parts of the structures one by one.

Copying only succeeds if a fast and lean operation is possible to do what we require. Otherwise, we get an exception, and we have to decide what we'll do instead, which might be to request a transfer.

The `transfer!` method will delegate to `copy!` when appropriate, but if that is not possible, it will silently find a workaround, which might include multiple intermediate steps and storage objects. Additionally, the operation may be implemented with Clojure functions, and not optimized native operations.

The bottom line is that we should prefer to explicitly use `copy!`, and only use `transfer!` at the boundaries, when there is no explicit path that we can think of. Please do not underestimate the value of copy's refusal to perform your request; often it is a plea to make our idea less complicated, which is achievable more often that we think at first!

Lifecycle

Clojure and Java programmers usually don't have to explicitly care about the lifecycle of the objects they use. We create our sequences and hash maps liberally, our functions do that even more in the course of their normal operation, Java garbage collector takes care of cleaning them up when appropriate, and we are happy. We also become careless. We often forget that the resources that we use (files, threads, etc.) are not automatically cleaned up. It's the same with memory outside the total control of the JVM, especially the memory on the GPU.

The memory that Neanderthal uses is a precious resource. Direct buffer allocation is expensive and slow. Don't do it. I mean, of course you have to do it, but only when it's necessary. *Prefer re-using existing objects whenever possible*. I admit, that is not the most elegant solution, but it's critical to be pragmatic in this case.

When you finish working with an object, don't litter. The garbage collector can't clean the objects that control the GPU memory, and even if the object is in the main memory, GC doesn't know how large the memory buffer is, so it might not it realease as soon as you hope. If there is an area where we should be overzealous, this is the one.

Releaseable

There are two parts of every Neanderthal vector or matrix: normal Clojure (Java) objects that we use directly, and the raw memory buffer that they control. The Clojure part is managed by the GC, so we don't have to worry[21] about its lifecycle; the raw memory is outside the JVM's control, and we can't rely on GC.

[21] At least not too much, most of the time.

If we don't do anything about releasing these outside resources, the GC will eventually release the Clojure managing object, while the raw buffer will stay allocated. If we're just experimenting in the REPL, we might not even notice that leak. In a long running process in production, such approach would lead to disaster.

The approach to releasing these outside resources is similar to how we treat files in Java: although the `File` object is managed by the GC, we still have to explicitly call the `close` method. Uncomplicate libraries provide the `Viewable` protocol, its explicit `release` method, and

the `with-release` and `let-release` macros that help with automation.

It's time for a simple demonstration in the REPL. Here's a tiny vector.

```
(def x10 (dv 1 2 3))
```

```
=>
#RealBlockVector[double, n:3, offset: 0, stride:1]
[   1.00    2.00    3.00 ]
```

We call an operation, the data is in its place, and we get our answer.

```
(asum x10)
```

```
=> 6.0
```

Then, we decide that we don't need the data any more, and we remember that we should take care of the underlying direct buffer, so we release the vector.

```
(release x10)
x10
```

Note how, although the Clojure managing object still exist, the contents randomly changed. The third entry is there, but the first two are suspiciously set to zero.

```
=>
#RealBlockVector[double, n:3, offset: 0, stride:1]
[   0.00    0.00    3.00 ]
```

This is an implementation detail of the CPU backend; the contents of small vectors could have even remained unchanged, for a time, until the OS finds a better use for it. But it might change right away, with, from the context of our application, pure garbage of numbers without any meaning.

Fortunately, most of the time we don't have to call `release` explicitly! If we need the vector only in some well-defined scope, then we can use `with-release`, just like in the following snippet.

```
(with-release [x20 (dv 1 2 3)]
  (asum x20))
```

```
=> 6.0
```

An interesting implementation detail: the JVM *will* de-allocate the raw byte buffer even if we don't release it! The problem is that it does not know how large the buffer is, so it treats every vector,

even a gigabyte size one, as a small object, whose de-allocation is not urgent. On the CPU, thus, `release` is a huge help.

On the GPU, however, explicit `release` is necessary, since the allocated memory will be reserved until we de-allocate it explicitly, or its context has been de-allocated. Consequently, the memory leaks are more severe.

```
(clojurecuda/with-default
  (with-default-engine
    (let [x10-cuda (cuv 1 2 3)]
      (asum x10-cuda)
      (release x10-cuda)
      (asum x10-cuda))))
```

```
=> Execution error (NullPointerException) at jcuda.jcublas.JCublas2/cublasSasumNative (JCublas2.java:-2).
Parameter 'x' is null for cublasSasum
```

Note that we got a NPE. That might look severe, but it's actually good. This is due to the protection provided by ClojureCUDA. CUDA code in C++ or Java that would try to use a previously destroyed object would lead to a crash of the process! In Clojure, we only get a reminder that, although the controlling object is still in scope, the controlled memory is not available any more. Our REPL session is uninterrupted!

In most situations we don't need to explicitly release objects; the `with-release` and `let-release` are preferred in most situations. `with-release` is a `let` like binding that releases the bound objects at the end.

```
(clojurecuda/with-default
  (with-default-engine
    (with-release [x20-cuda (cuv 1 2 3)]
      (asum x20-cuda))))
```

```
=> 6.0
```

The lifecycle management in a well-defined scope is fairly simple; the problems arise when we have to keep an object around for a longer time. The first challenge is to make sure that the underlying buffer of a partially created object is released when an exception breaks the normal flow. This is the task of `let-release`, which releases its bindings only if an exception occurs in its scope.

```
(defn produce-a-vector [source]
  (let-release [result (vctr source 2)]
    (entry! result 0 (iamax source))
```

```
    (entry! result 1 (amax source))
    result))
(def x30 (dv (range 10)))
(produce-a-vector x30)

=>
#RealBlockVector[double, n:2, offset: 0, stride:1]
[   9.00    9.00 ]

(release *1)
```

Typically, you'll use `with-release` for workspace objects that should only exist in a limited scope, and `let-release` when you create new objects that should be returned from a scope as the final result of some computation. The explicit `release` function is mostly used in lower-level engines; for example, see how `release` is used in Deep Diamond's and Neanderthal's engines.

Views

There are two main use cases for viewing a structure in another light. Often we would like to treat a vector as a matrix, or a matrix as a vector. Both structures will work with the same data, and both will see the changes immediately. Another case is when we have to forward some objects further, but we can't trust that the client code will know whether to release it or not. Then, we can forward the view to the client - releasing a view is a no-op - and manage the lifecycle of the master object in the main flow.

This is the main object.

```
(def a40 (dge 2 3 (range 10)))

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

The structure returned by the `view` function looks and works in exactly the same way.

```
(def a40-view (view a40))

=>
#RealGEMatrix[double, mxn:2x3, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ┐
```

```
   →       0.00    2.00    4.00
   →       1.00    3.00    5.00
   └                             ┘
```

Changing the view changes the main object.

```
(scal! 100.0 a40-view)
a40-view

=>
#RealGEMatrix[double, mxn:2x3, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ┐
   →       0.00  200.00  400.00
   →     100.00  300.00  500.00
   └                             ┘

a40

=>
#RealGEMatrix[double, mxn:2x3, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ┐
   →       0.00  200.00  400.00
   →     100.00  300.00  500.00
   └                             ┘
```

The same principle holds for sub-parts. In this example, we are changing the view of the first row of the matrix.

```
(scal! 0.01 (view (row a40-view 1)))
a40

=>
#RealGEMatrix[double, mxn:2x3, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ┐
   →       0.00  200.00  400.00
   →       1.00    3.00    5.00
   └                             ┘
```

We demonstrate the effects of releasing the view and master objects on the GPU, since, as we mentioned earlier, the data on the CPU might not be overwritten for a while even after being released.

```
(clojurecl/set-default-1!)
(cl/set-engine!)

(def gpu-a40 (clge 2 3 (range 7)))
gpu-a40
```

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

We release a view on the GPU, but `gpu-a40` is unaffected.

```
(release (view gpu-a40))
gpu-a40

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

However, when we release `gpu-a40`, the data becomes unavailable, even though the managing object is still in scope.

```
(release gpu-a40)
gpu-a40

=> #CLGEMatrix[float, mxn:2x3, layout:column, offset:0]
```

If we tried to call a computation on a released structure, Neanderthal will complain.

```
(asum gpu-a40)

=>
Execution error (NullPointerException) at org.jocl.blast.CLBlast/CLBlastSasumNative (CLBlast.java:-2).
Parameter 'x_buffer' is null for CLBlastSasum
```

When we need to look at the data from another angle, we might want to switch between vector and matrix view, including the various special matrix formats that we will discuss in a minute. Each of these structures has a `view-<suffix>` function. For example, we can view a matrix as if it was a vector with the help of the `view-vctr` function.

```
(view-vctr a40)

=>
#RealBlockVector[double, n:6, offset: 0, stride:1]
[   0.00    1.00    2.00          4.00    5.00 ]
```

This works on the GPU, too.

```
(view-vctr gpu-a40)
```

```
=>
#CLBlockVector[float, n:6, offset:0 stride:1]
[   0.00    1.00    2.00          4.00    5.00 ]
```

These views are capable of producing further views. In the following example, ve look at a matrix as a vector, and then we look at that vector view as a matrix.

```
(view-ge (view-vctr gpu-a40))
```

```
=>
#CLGEMatrix[float, mxn:6x1, layout:column, offset:0]
   ┌       ↓       ┐
   →       0.00
   →       1.00
   →       ⋱
   →       4.00
   →       5.00
   └               ┘
```

Note that the shape of these two matrices is not the same, as some information about the structure was lost in the process. All `view-*` functions support additional dimension parameters when it makes sense, so we could specify these details.

We won't use this OpenCL engine for the remaining of this chapter, so we can clean everything up by releasing the context itself.

```
(clojurecl/release-context!)
```

Specific matrix structures

Most of the time, our needs will be best served by general dense matrices (GE). In some cases, though, we know, by employing theoretical knowledge, that the matrix we're working with, has a special structure for which there are special, optimized algorithms. In most of these cases, GE matrices might be the simplest, and might work well enough, but we have the option to fit the solution to the problem even better, with a bit extra specificity and involvement, if we wish so.

Most of the operations are polymorphic; Neanderthal automatically executes the right implementation, but the specialization has to be specified somewhere. Most often, it's at the time of construction or view taking.

For example, by calling the `tr` function, we create a triangular matrix, which assumes zero on one side of the diagonal. Since triangular matrices are quadratic, `tr` appropriately asks for just one dimension.

```
(tr x 3 (range 1 7))

=>
#RealUploMatrix[float, type:tr, mxn:3x3, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ┐
   →        1.00    *       *
   →        2.00    4.00    *
   →        3.00    5.00    6.00
   └                               ┘
```

TR matrices use the same logically rectangular area as GE matrices. They just ignore the positions that don't belong to them, printed out as "∗". If we want to save space, we might use triangular packed matrices, created by `tp`.

```
(tp x 3 (range 6))

=>
#RealPackedMatrix[float, type:tp, mxn:3x3, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ┐
   →        1.00    .       .
   →        2.00    4.00    .
   →        3.00    5.00    6.00
   └                               ┘
```

Note the "." instead "∗" in the printout. It indicates that this entry logically exists, and is 0, but physically doesn't use any memory. The difference between TR and TP matrices can be easily understood if we look at their vector views.

```
(seq (view-vctr (tr x 3 (range 1 7))))

=>
(1.0 2.0 3.0 0.0 4.0 5.0 0.0 0.0 6.0)

(seq (view-vctr (tp x 3 (range 1 7))))

=>
(1.0 2.0 3.0 4.0 5.0 6.0)
```

Please note that, while packed matrices save half of the storage space, the algorithms for general matrices are more developed and performant.

Symmetric matrices have a similar triangular structure, but instead of assuming zeres for entries it does not control, it assumes the "mirror image" of entries opposite the diagonal.

```
(sy x 3 (range 1 6))
```

```
=>
#RealUploMatrix[float, type:sy, mxn:3x3, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ┐
   →       1.00    *       *
   →       2.00    4.00    *
   →       3.00    5.00    0.00
   └                               ┘
```

Packed symmetric matrices are available, too.

```
(sp x 3 (range 1 7))

=>
#RealPackedMatrix[float, type:sp, mxn:3x3, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ┐
   →       1.00    .       .
   →       2.00    4.00    .
   →       3.00    5.00    6.00
   └                               ┘
```

There's more specific structures, which we might touch later in the book, that you can find in the `core` namespace, such as banded matrices (GB), their triangular (TB) and symmetric (SB) variants, various flavors of diagonal matrices, and other structures that are supported by BLAS and similar standards.

```
(sb x 8 1 (range 1 100))

=>
#RealBandedMatrix[float, type:sb, mxn:8x8, layout:column, offset:0]
   ┌       ↓       ↓       ↓       ↓       ↓       ↓
   ↘       1.00    3.00    5.00    7.00    9.00
   ↘       2.00    4.00    6.00    8.00   10.00
   └                                                 ┘
```

Note the arrows in the printout. They remind you that these entries, which are printed out vertically, are logically diagonally spaced, since banded matrices only have non-zero entries in a short band around the diagonal (in this case, the main diagonal and one sub-diagonal).

View switching is pretty versatile, where it makes sense.

```
(view-vctr (view-tr (view-ge (sy x 3 (range 1 6)))))

=>
#RealBlockVector[float, n:9, offset: 0, stride:1]
[   1.00    2.00    3.00          0.00    0.00 ]
```

Polymorphic acceleration

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[22]. The reasons I chose this book are:

[22] 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`[23]:

[23] `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).

[24] `scal`, scalar multiplication

Scalar multiplication is done using the pure function `scal`.[24] 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`[24] and avoid memory copying by using destructive functions such as `scal!` and `axpy!`[25].

[25] 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:[26]

[26] 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`[27]:

[27] 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`[28] function to compute this norm:

[28] `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.[29]

[29] 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*[30] 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`.

[30] orthogonal vectors

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

```
=> 0.0
```

These two vectors are orthogonal.

We can determine *distance between points*[31] in a vector space by calculating the norm of the difference between their direction vectors.

[31] 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[32] 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.

[32] 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)[33], *dense triangular* matrices (TR), *dense symmetrical* matrices (SY), various banded storages for symmetric and triangular matrices, sparse matrices (Sp), etc.

[33] dense matrix (GE)

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
```

Euclidean norm applied to matrices is known as the Frobenius norm.[34]

[34] Frobenius norm

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`[35], 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.

[35] `svd`: singular value decomposition

Here is an example[36] done in Clojure.

[36] 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.

Eigenvalues and eigenvectors

Matrix transformations

Linear transformations

High performance matrix computations

Use matrices efficiently

Linear systems and factorization

Singular value decomposition (SVD)

Orthogonalization and least squares

In practice

GPU computing crash course

Generating random matrices

Broadcasting

Mean, variance, and correlation

Principal component analysis (PCA)

Appendix

Setting up the environment and the JVM

Many more early supporters helped me create this book. Thank you!

David Pham, Mandimby RAVELOARINJAKA, Davide Del Vecchio, Joshua M, Scott MacFarlane, Estevo U. C. Castro, Rokas Ramanauskas, Saransh Mohapatra, James, Amar, Scott Jappinen, Don Jackson, Brandon Adams, Kimmo Koskinen, Salvatore Uras, Jeffrey Cummings, Benjamin Rood, Alexander Yakushev, Evan Niessen-Derry, Dan Dorman, Robert Postill, Felipe Gerard, Ted McFadden, James Tolton, drew verlee, Tracy Shields, Emmanuel Oga, Jon Anthony, Boris, Jason Gilbertson, Pedro Gomes, Thomas Wengerek, Joseph Bore, Leon Talbot, k-sunako, Robert Crim, Nick Jones, Christoph Ulrich, Chris Bilson, Vijay Edwin, Zack Teo, Hans Royer, James Good, Tim Howes, Erkki Keränen, Arthur Ulfeldt, Alfred Thompson, Mike Ananev, Alex Doroshenko, Alex Murphy, Trevor Prater, Tomáš Drenčák, Brandon, Matt Burbidge, Matt Pendergraft, Yiu Ming Huynh, Paul Snively, Donald Bleyl, Ken Rawlings, Martin Jung, Otis Clark, Pavithra Solai Jawahar, Nicholas Stares, Jonathan Rioux, Dennis Orsini, Amresh Venugopal, Stephan S-E, Jason Mulvenna, Rohit Thadani, Valerii Praid, Pitti, Peter Makumbi, MR A BIN SALMAN, Pete Windle, John Sullivan, Eric Fode, Matthew Chadwick, Jaihindh Reddy, Darrick Wiebe, Bor Hodošček, Christopher Hugh Graham, Sam Symons, Ben Wiedemann, infiniteone, Dan Pomohaci, Samuel Curran, Ariel Ferdman, Szabolcs, Jonathan Smith, indraniel, Martin, Rangel Spasov, Jason Waack, Brandon Olivier, Machiel Keizer-Groeneveld, Amilcar Blake, Nick Burkhart, Mooreisenough, Dmitri, Jay Zisch, Keith Montgomery, Tara Lorenz, Jon Irving, Adam Morgan, Tory S. Anderson.