Course notes for High Performance Programming and Systems

Kenneth Skovhede, Troels Henriksen

November 4, 2020

0.1 Introduction

These notes are written to supplement the textbooks in the course *High Performance Programming and Systems*. Consider them terminally in-progress. These notes are not a textbook, do not cover the entire curriculum, and might not be comprehensible if isolated from the course and its other teaching activities.

Contents

Chapter 1

Compiled and Interpreted Languages

A computer can directly execute only machine code, consisting of raw numeric data. Machine code can be written by humans, but we usually use symbolic *assembly languages* to make it more approachable. However, even when using an assembly language, this form of programming is very tedious. This is because assembly languages are (almost) a transparent layer of syntax on top of the raw machine code, and the machine code has been designed to be efficient to *execute*, not to be a pleasant programming experience. Specifically, we are programming at a very low level of abstraction when we use assembly languages, and with no good ability to build new abstractions. In practice, almost all progamming is conducted in *high-level languages*.

1.1 Low-level and High-Level Languages

For the purpose of this chapter, a high-level programming language is a language that is designed not to directly represent the capabilities and details of some machine, but rather to *abstract* the mechanical details, in order to make programming simpler. However, we should note that "high-level" is a spectrum. In general, the meaning of the term "high-level programming language" depends on the speaker and the context (fig. 1.1). The pioneering computer scientist Alan Perlis said: *"A programming language is low-level when its programs require attention to the irrelevant"*. During the course you will gain familiarity with the programming language C, which *definitely* requires you to pay attention to things that are often considered irrelevant, which makes it low-level in Perlis's eyes. However, we will see that the control offered by C provides some capabilities, mostly the ability to *tune* our code for high performance, that are for some problems *not* irrelevant. The term *mid-level programming language* might be a good description of C, as it fills a niche between low-level assembly languages, and high-level languages such as Python and F#.

Generally speaking, low-level languages tend to be more *difficult* to program

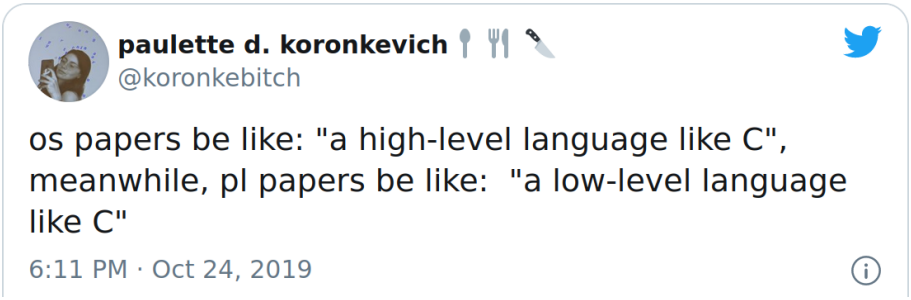


Figure 1.1: A remark on the clarity of terms in computer science.

in, while offering greater potential *performance* (i.e. they are faster). Higher-level languages are much easier to program in, but run slower and require more machine resources (e.g. memory). Given the speed of modern computers, this is a price we are often willing to pay—especially in the common case where the slowest part of our program is waiting for information from disk or network. Do not make the mistake of assuming that a program written in a low-level language is *always* faster than one written in a high-level language. Choice of algorithm is often more important than choice of language. Further, some high-level languages are designed specifically to execute very quickly. But there is no free lunch: these languages tend to make tradeoffs in other areas. There is no objective measure of where a language lies on the scale of "level-ness", so while a statement such as *"Python is more high-level than C"* is unlikely to raise any objections, it is usually pointless to try to rank very similar languages on this spectrum.

1.2 Compilers and Interpreters

As the computer natively understands only its machine code, other languages must be *translated* to machine code in order to run. For historical reasons, this is called *compilation*. We say that a compiler takes as input a file with a *source program*, and produces a file containing an executable *machine program* that can be run directly. This is a very simplified model, for the following reasons:

1. Strictly speaking, a compiler does not have to produce machine code. A compiler can also produce code in a different high level languag. For example, with the rise of browsers, it has become common to write compilers that produce Javascript code.

2. The machine program normally cannot be *directly* executed, as modern systems have many layers of abstraction on top of the processor. While the compiler does produce machine code, it is usually stored in a special file format that is understood by the *operating system*, which is responsible for making the machine code available to the processor.

3. The actual compiler contains many internal steps. Further, large programs are typically not compiled all at once, but rather in chunks. Typically, each *source file* is compiled to one *object file*, which are finally *linked* to form an executable program.

While compilers are a fascinating subject in their own right, we will discuss them only at a practical level. For a more in-depth treatment, you are encouraged to read a book such as Torben Mogensen's *Basics of Compiler Design*[1] for more information.

In contrast, an *interpreter* is a program that executes code directly, without first translating it. The interpreter can be a compiled program, or itself be interpreted. At the bottom level, we always have a CPU executing machine code, but there is no fundamental limit to how many layers of interpreters we can build on top. However, the most common case is that the interpreter is a machine code program, typically produced by a compiler. For example, Python is an interpreted language, and the `python` interpreter program used by most people, is written in C, and compiled to machine code.

Interpreters are generally easier to construct than compilers, especially for very dynamic languages, such as Python. The downside is that code that is interpreted generally runs much slower than machine code. This is called the *interpretive overhead*. When a C compiler encounters an integer expression `x + y`, then this can likely be translated to a single machine code instruction (possibly preceded by instructions to read `x` and `y` from memory. In contrast, whenever an interpreter encounters this expression, it has to analyse it and figure out what is supposed to happen (integer addition), and then dispatch to an implementation of that operation. This is usually at least an order of magnitude slower than actually doing the work. This means that interpreted languages are usually slower than compiled languages. Many programs spend most of their time waiting for user input, for a network request, or for data from the file system. Such programs are not greatly affected by interpretive overhead.

As an example of interpretive overhead, let us try writing programs for investigating the Collatz conjecture. The Collatz conjecture states that if we repeatedly apply the function

$$f(n) = \left\{ \begin{array}{ll} \frac{n}{2} & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{array} \right\}$$

to some initial number greater than 1, then we will eventually reach 1. To investigate this function, the Python program `collatz.py` in listing 1.1 takes an initial k from the user, then for every $1 \leq n < k$ prints out n followed by the number of iterations of the function it takes to reach 1.

[1] `http://hjemmesider.diku.dk/~torbenm/Basics/`

Listing 1.1: A Python program for investigating the Collatz conjecture.

```
import sys

def collatz(n):
    i = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        i = i + 1
    return i

k = int(sys.argv[1])
for n in range(1, k):
    print(n, collatz(n))
```

In a Unix shell we can time the program for $k = 100000$ as follows, where we explicitly ignore the output[2]:

```
$ time python3 ./collatz.py 100000 >/dev/null

real    0m1.368s
user    0m1.361s
sys     0m0.007s
```

The `real` measurement tells us that the program took a little more than $1.3s$ to run in the real world (we'll talk about the meaning of `user` and `sys` later in the course).

Now let us consider the same program, but written in C, which we call `collatz.c`, and is shown in listing 1.2.

C is a compiled language, so we have to compile `collatz.c`:

```
$ gcc collatz.c -o collatz
```

And then we can run it:

```
$ time ./collatz 100000 >/dev/null

real    0m0.032s
user    0m0.030s
sys     0m0.002s
```

[2]This is a very naive way of timing program—it's adequate for programs that run for a relatively long time, but later we will have to discuss better ways to measure performance. In particular, it ignores the overhead of starting up the Python interpreter, and it is sensitive to noise, because we only to a single run.

Listing 1.2: A C program for investigating the Collatz conjecture.

```c
#include <stdio.h>
#include <stdlib.h>

int collatz(int n) {
  int i = 0;
  while (n != 1) {
    if (n % 2 == 0) {
      n = n / 2;
    } else {
      n = 3 * n + 1;
    }
    i++;
  }
  return i;
}

int main(int argc, char** argv) {
  int k = atoi(argv[1]);
  for (int n = 1; n < k; n++) {
    printf("%d %d\n", n, collatz(n));
  }
}
```

Only $0.032s$! This means that our C program is

$$\frac{1.368}{0.032} = 42.75$$

times faster than the Python program. This is not unexpected. The ease of use of interpreted languages comes at a significant overhead.

1.2.1 Advantages of interpreters

People implement interpreters because they are easy to construct, especially for advanced or dynamic languages, and because they are easier to work with. For example, when we are compiling a program to machine code, then the compiler throws away a lot of information, which makes it difficult to relate the generated machine code with the code originally written by the compiler. This makes debugging harder, because the connection between what the machine physically *does*, and what the programmer *wrote*, is more complicated. In contrast, an interpreter more or less executes the program as written by the user, so when things go wrong, it is easier to explain where in the source code the problem occurs.

In practice, to help with debugging, good compilers can generate significant amounts of extra information in order to let special *debugger* programs connect the generated machine code with the original source code. However, this does tend to affect the quality of the generated code.

Another typical advantage of interpreters is that they are straightforwardly *portable*. When writing a compiler that generates machine code, we must explicitly write a code generator every CPU architecture we wish to target. An interpreter can be written once in a portable programming language (say, C), and then compiled to any architecture for which we have a C compiler (which is essentially all of them).

As a rule of thumb, very high-level languages tend to be interpreted, and low-level languages are almost always interpreted. In practice, things are not always so clear cut, and *any* language can in principle be compiled—it may just be very difficult for some languages.

1.2.2 Blurring the lines

Very few production languages are *pure* interpreters, in the sense that they do no processing of the source program before executing it. Even Python, which is our main example of an interpreted language, does in fact compile Python source code to Python *bytecode*, which is a kind of invented machine code that is then interpreted by the Python *virtual machine*, which is an interpreter written in C. We can in fact ask Python to show us the bytecode corresponding to a function:

```
>>> import dis
>>> def add(a,b,c):
...     return a + b + c
...
>>> dis.dis(add)
  2           0 LOAD_FAST                0 (a)
              2 LOAD_FAST                1 (b)
              4 BINARY_ADD
              6 LOAD_FAST                2 (c)
              8 BINARY_ADD
             10 RETURN_VALUE
```

This is not machine code for any processor that has ever been physically constructed, but rather an invented machine code that is interpreted by Python's bytecode interpreter. This is a common design because it is faster than interpreting raw Python source code, but it is still much slower than true machine code.

JIT Compilation

An even more advanced implementation technique is *just-in-time* (JIT) compilation, which is notably used for languages such as C#, F# and JavaScript.

Here, the source program is first compiled to some kind of intermediary bytecode, but this bytecode is then further compiled *at run-time* to actual machine code. The technique is called just-in-time compilation because the final compilation typically occurs on the user's own machine, immediately prior to the program running.

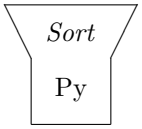
The main advantage of JIT compilation is that programs run much faster than when interpreting bytecode, because we ultimately do end up executing a machine code version of the program. Because JIT compilation takes place while the program is running, it is also able to measure the actual run-time behaviour of the program and tailor the code generation to the actual data encountered in use. This is useful for hightly dynamic languages, where traditional *ahead-of-time* (AOT) compilers have difficulty getting good results. In theory, a JIT compiler can always be *at least as good* as an AOT compiler, but in practice, AOT compilers tend to generate better code, as they can afford to spend more time on compilation. In practice, JIT compilers are only used to compute those parts of the program that are "hot" (where a lot of time is spent), and an interpreter is used for the rest. This tends to works well in practice, due to the maxim that 80% of the run-time is spent in 20% of the code. An AOT compiler will not know which 20% of the code is actually hot, and so much dedicate equal effort to every part, while a JIT compiler can measure the run-time behaviour of the program, and see where it is worth putting in extra effort.

The main downside of JIT compilation is that it is difficult to implement. It has been claimed that AOT compilers are 10× as difficult to write as interpreters, and JIT compilers are 10× as difficult to write as AOT compilers.

1.3 Tombstone diagrams

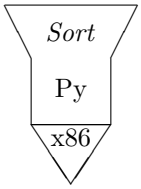
Interpreters and compilers allow us to consider programs as input and output of other programs. That is, they are *data*. *Tombstone diagrams* (sometimes called *T-diagrams*) are a visual notation that lets us describe how a program is translated between different languages (*compiled*), and when execution takes place (either through a software interpreter or a hardware processor). They are not a completely formal notation, nor can they express every kind of program transformation, but they are useful for gaining an appreciation of the big picture.

As the most fundamental concepts, we have programs, which are written in some language. Suppose we have a sorting program written in Python, which we draw as follows:

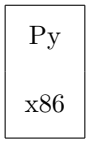


This is an incomplete diagram, since it contains programs we have not described how to execute. A machine that executes some language, say x86 machine code is illustrated as a downward-pointing triangle:

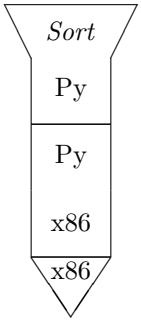
We can say that the Python program is executed on this machine, by stacking them:



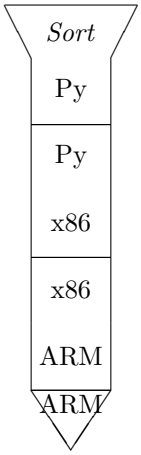
But this diagram is *wrong* — we are saying that a program written in Python is running on a machine that executes only x86. When putting together a tombstone diagram, we must ensure that the languages of the components match. While on paper, we can just assume a Python machine, this is not very realistic. Instead, we use an interpreter for Python, written in x86, written like this:



We can then stack the Python program on top of the interpreter, which we then stack on top of the machine:

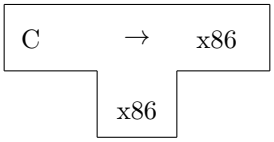


But maybe we are actually running on an ARM machine (as can be found in most phones), but still only have a Python interpreter in x86. As long as we have an x86 interpreter written in ARM machine code, this is no problem:

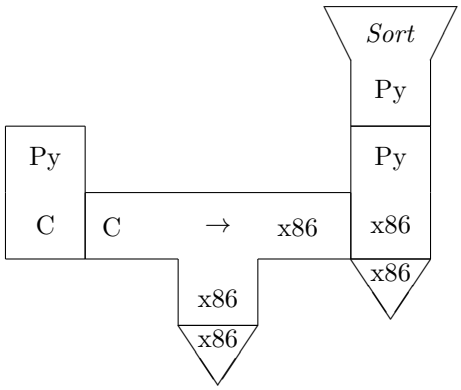


There is no limit to how high we can stack interpreters. All that matters is that at the end, we have either a machine that can run the implementation language of the bottommost interpreter. Of course, in practice, each level of interpretation adds overhead, so while tombstone diagrams show what is *possible*, they do not necessarily show what is a good idea. Tall interpreter stacks mostly occur in retrocomputing or data archaeology, where we are simulating otherwise dead hardware.

The diagrams above are a bit misleading, because the Python interpreter is not actually written in machine code—it is written in C, which is then translated by a compiler. With a tombstone diagram, a compiler from C to x86, where the compiler is itself also written in x86, is illustrated as follows:

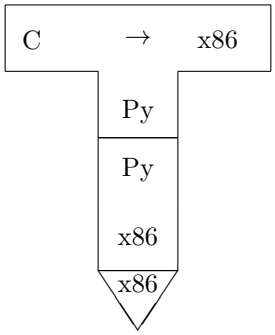


We can now put together a full diagram showing how the Python interpreter is translated from C to x86, and then used to run a Python program:



For a diagram to be valid, every program, interpreter, or compiler, must either be stacked on top of an interpreter or machine, or must be to the left of a compiler, as with the Python interpreter above.

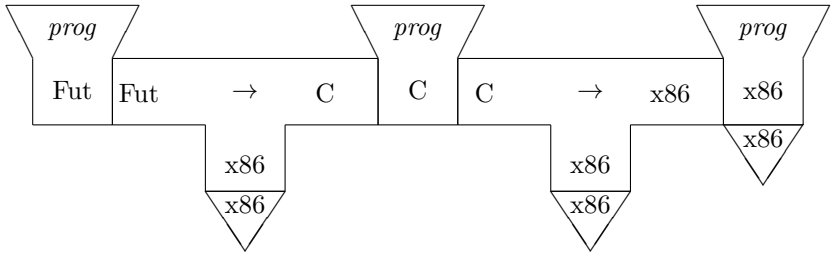
Compilers are also just programs, and must either be executed directly by an appropriate machine, or interpreted. For example, the following diagram shows how to run a C compiler in Python, on top of a Python interpreter in x86 machine code:



How the Python interpreter has been obtained, whether written by hand or compiled from another language, is not visible in the diagram.

We can also use diagrams to show compilation pipelines that chain multiple compilers. For example, programs written in the Futhark[3] programming language are typically compiled first to C, and then uses a C compiler to generate machine code, which we can then finally run:

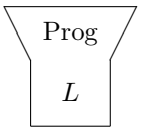
[3] `https://futhark-lang.org`



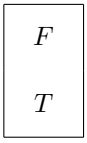
Many compilers have multiple internal steps—for example, a C compiler does not usually generate machine code directly, but rather generates symbolic assembly code, which an *assembler* then translates to binary machine code. Typically tombstone diagrams do not include such details, but we can include them if we wish, such as with the Futhark compiler above.

Tombstone diagrams can get awkward in complex cases (sometimes there will be no room!), but they can be a useful illustration of complex setups of compilers and interpreters. Also, if we loosen the definition of "machine" to include "operating systems", then we can use these diagrams to show how we can emulate Windows or DOS programs on a GNU/Linux system.

Tombstone diagrams hide many details that we normally consider important. For example, a JIT compiler is simply considered an interpreter in a tombstone diagram, since that is how it appears to the outside. Also, tombstone diagrams cannot easily express programs written in multiple languages, like the example shown in section 1.4. Always be aware that tombstone diagrams are a very high-level visualisation. In practice, such diagrams are mostly used for describing *bootstrapping* processes, by which we make compilers available on new machines. The tombstone diagram components are summarised in fig. 1.2.

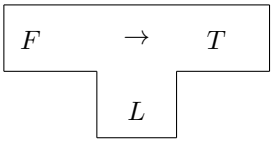


(a) A program written in language L.



(b) An interpreter for F, written in T.

(c) A machine that runs L programs.



(d) A compiler from F to T, written in L.

Figure 1.2: A summary of tombstone diagram building blocks.

1.4 Combining Python and C

As discussed above, interpreted languages are typically substantially slower than compiled languages, especially for languages with high *computational intensity*. By this term, we mean how much of the execution time is spent directly executing program code, and how much is spent waiting for data (e.g. user input or network data). For programs with low computational intensity, an interpreted language like Python is an excellent choice, as the interpretive overhead has little impact. However, Python is also very widely used for computationally heavy programs, such as data analysis. Do we just accept that these programs are much slower than a corresponding program written in C? Not exactly. Instead, we use high-performance languages, such as C, to write *computational kernels* in the form of C functions. These C functions can then be called from Python using a so-called *foreign function interface* (FFI).

As a simple example, let us consider the `collatz.c` program from listing 1.2. Instead of compiling the C program to an executable, we compile it to a so-called *shared library*, which allows it to be loaded by Python[4]:

```
$ gcc collatz.c -fPIC -shared -o libcollatz.so
```

We can now write a Python program that uses the `ctypes` library to access the compiled in the `libcollatz.so` library, and call the `collatz` function we wrote in C:

Listing 1.3: A Python program that uses a C implementation of `collatz`.

```
import ctypes
import sys

c_lib = ctypes.CDLL('./libcollatz.so')

k = int(sys.argv[1])
for n in range(1, k):
    print(n, c_lib.collatz(n))
```

Let's time it as before:

```
$ time python3 ./collatz-ffi.py 100000 >/dev/null

real    0m0.165s
user    0m0.163s
sys     0m0.003s
```

The pure Python program ran in $1.3s$, the pure C in $0.032s$, and this mixture in $0.165s$ - significantly faster than Python, but slower than C by itself. The difference is mostly down to the implicit work required to converte Python

[4]Don't worry about the details of the command line options here—the technical details are less important than the overall concept.

values to C values when calling `c_lib.collatz`. The overhead is particularly acute for this program, because each call to `collatz` does relatively little work.

While this example is very simple, the basic idea is *fundamental* to Python's current status as perhaps the most popular language for working data scientists and students. Ubiquitous libraries such as NumPy and SciPy have their computational core written in high-performance C or Fortran, which is then exposed in a user-friendly way through Python functions and objects. While a program that uses NumPy is certainly much slower than a tightly optimised C program, it is *much* faster than a pure Python program would be, and *far* easier to write than a corresponding C program.

Chapter 2

Data Layout

One of the things that makes C a difficult programming language is that it does not provide many built-in data types, and provides poor support for making it convenient to work with new data types. In particular, C has notoriously poor support for multi-dimensional arrays. Given that multi-dimensional arrays are perhaps the single most important data structure for scientific computing, this is not good. In this chapter we will look at how we *encode* mathematical objects such as matrices (two-dimensional arrays) with the tools that C makes available to us. One key point is that there are often multiple ways to represent the same object, with different pros and cons, depending on the situation.

2.1 Arrays in C

At the surface level, C does support arrays. We can declare a $n \times m$ array as

```
double A[n][m];
```

and then use the fairly straightforward A[i][j] syntax to read a given element. However, C's arrays are a second-class language construct in many ways:

- They decay to points in many situations.
- They cannot be passed to a function without "losing" their size.
- They cannot be returned from a function at all.

In practice, we tend to only use language-level arrays in very simple cases, where the sizes are statically known, and they are not passed to or from funtions. For general-purpose usage, we instead build our own representation of multi-dimensional arrays, using C's support for *pointers* and *dynamic allocation*. Since actual machine memory is essentially a single-dimensional array, working with multi-dimensional arrays in C really just requires us to answer one central question:

How do we map a multi-dimensional index to a single-dimensional index?

Or to put it another way, representing a d-dimensional array in C requires us to define a bijective[1] *index function*

$$I : \mathbb{N}^d \to \mathbb{N} \tag{2.1}$$

The index function maps from our (mathematical, conceptual) multi-dimensional space to the one-dimensional memory space offered by an actual computer. This is sometimes also called *unranking*, although this is strictly speaking a more general term from combinatorics.

As an example, suppose we wish to represent the following 3×4 matrix in memory:

$$\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{pmatrix} \tag{2.2}$$

We can do this in any baroque way we wish, but the two most common representations are:

Row-major order, where elements of each *row* are contiguous in memory:

11	12	13	14	21	22	23	24	31	32	33	34

with index function

$$(i, j) \mapsto i \times 4 + j$$

Column-major order, where elements of each *column* are contiguous in memory:

11	21	31	12	22	32	13	23	33	14	24	34

with index function

$$(i, j) \mapsto j \times 3 + i$$

The index functions are generalised on fig. 2.1. Note that the two representations contain the exact same values, so they encode the same mathematical object, but in different ways. The intuition for the row-major index function is that we first skip i rows ahead to get to the row of interest, then move j columns into the row.

Row-major order is used by default in most programming languages and libraries, but not universally so—the scientific language Fortran is famously column-major. The NumPy library for Python uses row-major by default (called `C` in Numpy), but one can explicitly ask for arrays in column-major order (called `F`), which is sometimes needed when exchanging data with systems that expect a different representation.

[1] A bijective function is a function between two sets that maps each element of each set to a distinct element of the other set.

$$(i, j) \mapsto i \times m + j \qquad (2.3)$$

(a) Row-major indexing.

$$(i, j) \mapsto j \times n + i \qquad (2.4)$$

(b) Column-major indexing.

Figure 2.1: Index functions for $n \times m$ arrays represented in row-major and column-major order. For an example of why computer scientists tend to prefer 0-indexing, try rewriting the above to work with 1-index arrays instead.

2.1.1 Implementation in C

Let's look at how to implement this in C. Let's say we wish to represent the matrix from eq. (2.2) in row-major order. Then we would write the following (assuming `n=3`, `m=4`):

```
int *A = malloc(n*m*sizeof(int));
A[0] = 11;
A[1] = 12;
...
A[11] = 34;
```

Note that even though we *conceptually* wish to represent a two-dimensional array, the actual C type is technically a single-dimensional array with 12 elements. If we when wish to index at position (i, j) we then use the expression A[i*4+j].

Similarly, if we wished to use column-major order, we would program as follows:

```
int *A = malloc(n*m*sizeof(int));
A[0] = 11;
A[1] = 21;
...
A[11] = 34;
```

To C there is no difference—and there is no indication in the types what we intended. This makes it very easy to make mistakes.

Note also how it is on *us* to keep track of the sizes of the array—C is no help. Don't make the mistake of thinking that sizeof(A) will tell you how big this array is—while C will produce a number for you, it willindicate the size of a *pointer* (probably 8 on your machine).

Some C programmers like defining functions to help them generate the flat indexes when indexing arrays:

```
int idx2_rowmajor(int n, int m, int i, int j) {
  return i * m + j;
}

int idx2_colmajor(int n, int m, int i, int j) {
  return j * n + i;
}
```

Note how row-major indexing does not use the `n` parameter, and column-major indexing does not use `m`.

However, these functions do not on their own fully prevent us from making mistakes. Consider indexing the `A` array from before with the expression

A[idx2_rowmajor(n, m, 2, 5)].

Here we are trying to access index $(2, 5)$ in a 3×4 array—which is conceptually an out-of-bounds access. However, by the index function, this translates to the flat index $2 \times 3 + 5 = 11$, which is in-bounds for the 12-element array we use for our representation in C. This means that handy tools like `valgrind` will not even be able to detect our mistake—from C's point of view, we're doing nothing wrong! This like this make scientific computing in C a risky endeavour. We can protect ourselves by using helper functions like those above, and augment them with `assert` statements that check for problems:

```
int idx2_rowmajor(int n, int m, int i, int j) {
  assert(i >= 0 && i < n);
  assert(j >= 0 && j < m);
  return i * m + j;
}
```

We can still make mistakes, but at least now they will be noisy, rather than silently reading (or corrupting!) unintended data.

2.1.2 Size passing

With the previously discussed representation, a multidimensional array (e.g. a matrix) is just a pointer to the data, along with metadata about its size. The C language does not help us keep this metadata in sync with reality. When passing one of these arrays to a function, we must manually pass along the sizes, and we must get them right without much help from the compiler. For example, consider a function that sums each row of a (row-major) $n \times m$ array, saving the results to an n-element output array:

Listing 2.1: Summing the rows of a matrix.

```
void sumrows(int n, int m,
             const double *matrix, double *vector) {
  for (int i = 0; i < n; i++) {
    double sum = 0;
    for (int j = 0; j < m; j++) {
      sum += matrix[i*m+j];
    }
    vector[i] = sum;
  }
}
```

C gives us the raw building blocks of efficient computation, but we must put together the pieces ourselves. We protect ourselves by carefully documenting the data layout expected of the various functions. For the `sumrows` function above, we would document that `matrix` is expected to be a row-major array of size $n \times m$.

2.1.3 Slicing

In high-level languages like Python, we can use notation such as `A[i:j]` to extract a *slice* (a contiguous subsequence) of an array. No such syntactical niceties are available in C, but by using our knowledge of how arrays are physically laid out in memory, we can obtain similar effect in many cases.

Suppose `V` is a vector of `n` elements, and we wish to obtain a slice of the elements from index `i` to `j` (the latter exclusive). In Python, we would merely write `V[i:j]`. In C, we compute the size of the slice as

```
int m = j - i;
```

and then compute a pointer to the start of the slice:

```
double *slice = &V[i];
```

Now we can simply treat `slice` as an `m`-element array, which uses the same underlying storage as `V`—just as in Python.

Similarly, if `A` represents a matrix of size `n` by `m` in row-major order, then we can produce a vector representing the `i`th row as follows:

```
double *row = &A[i*m];
```

The restriction is that such slicing can only produce elements that are *contiguous* in memory. For example, we cannot easily extract a column of a row-major array, because the elements of a column are not contiguous in memory. If we wish to extract a column, then we have to allocate space and copy element-by-element, in a loop[2].

2.1.4 Even higher dimensions

The examples so far have focused on the two-dimensional case. However, the notion of row-major and column-major order generalises fine to higher dimensions. For a row-major array of shape $n_0 \times \cdots \times n_d$, the index function where p is a d-dimensional index point is

$$p \mapsto \sum_{0 \leq i < d} p_i \times \prod_{i < j < d} n_j \tag{2.5}$$

where p_i gets the ith coordinate of p, and the product of an empty series is 1.

[2]There are more sophisticated array representations that use *strides* to allow array elements that are not contiguous in memory—NumPy uses these, but their representation are outside the scope of our course.

We can also have more complex cases, such as a three-dimensional array where the two-dimensional "rows" are stored consecutively, but are individually column-major. Such constructions can make sense, but are beyond the scope of this course (and are a nightmare to implement).