idss_selfstudy_numerical_i_ch_1

January 14, 2021

Introduction to Data Science and Systems
Self-study: Arrays, numpy and vectorisaiton
Chapter 1: Why use arrays?

#####
University of Glasgow - material prepared by John H. Williamson*; adapted to IDSS by BSJ.

$*arg\ min$

---

1 Why use arrays?

1.0.1 Images, sounds, videos

Numerical arrays - i.e. data types storing numbers in a structured manner - sound boring. But they are arguably the most "fun" data structure. Images, sounds, videos are all most easily worked with as arrays of numbers. * An *image* is a 2D array of brightness values; * a *sound* is a 1D array of sound pressure levels; * a *video* is a 3D array of brightness values (x, y, and time).

And the manipulations that we might want to apply to these kinds of data (e.g. brightening a video, mixing together two sounds, cropping a video, rotating a 3D model) are very straightforward to express in terms of array operations. This leads to compact, elegant code that can be astonishingly efficient.

1.0.2 Scientific data

Scientific data (e.g. from physics experiments, weather models, even models of how people choose search terms on Google) can often be most conveniently represented as numerical arrays. The kind of operations we want to do to scientific data (e.g. find the weather most similar today in the historical record) are easily expressed as array operations.

1.0.3 3D graphics

3D computer graphics, as you would encounter in a game or VR, usually involves manipulating **geometry**. Geometry is typically specified as simple geometric shapes, like triangles. These shapes are made up of points – **vertices** – typically with an $[x, y, z]$ location. Operations like moving, rotating, scaling of objects are operations on big arrays of these vertices:

```
vec3 player = [
        [x,y,z]
        [x,y,z]
        [x,y,z]
        [x,y,z]
        ...
    ]
```

Being able to manipulate positions in space efficiently and cleanly is an important tool in making computer graphics programming work.

1.0.4 Abstraction and elegance

By representing data as a numerical array, we can extend the operations we apply to single numbers (like integers or floating points) to entire arrays of numbers. For example, if we have an array of 100 3D positions `pos`, it would be very nice if we could scale all of the points by a factor of 2 (double in size) and move the whole array 100 units right like this:

```
pos = pos * 2 + [100,0,0]
```

This very clearly expresses the operation to be performed. Applying an identical operation to many elements of an array simultaneously is a very useful operation. Code which can express this type of operation without explicit loops is a easier to read and write. Consider the alternative:

```
new_pos = []
for x,y,z in pos:
    new_pos.append((2*x+100, 2*y+0, 2*z+0))
pos = new_pos
```

1.0.5 Mathematical power

There is a rich set of mathematical abstractions that work on spaces defined over array-valued elements. For example, **linear algebra** provides tools to work with 1D arrays (*vectors*) and 2D arrays (*matrices*) and can be used to solve many difficult problems. Having these types represented as basic types in a programming language makes working with linear algebraic problems vastly easier.

1.0.6 Efficiency

Numerical arrays are both **compact** (they store data in a very memory efficient way) and **computationally efficient** (it is possible to write code that manipulates arrays extremely quickly).

[Image credit: NOAA, Public domain]

For big, number-focused problems like: * weather simulation * image processing * speech recognition * machine learning

arrays are the best way we have of solving these problems.

In big, text-heavy problems with irregular structure, databases are a more natural structure to store and work with data.

Deep learning Some of you may have seen recent advances in *machine learning* involving "deep learning". This has had some major impact in the last five years:

- Redefined state of the art in speech recognition.
- Cutting edge speech synthesis: wavenet
- State of the art image recognition: inceptionnet
- Auto captioning images: image-from-text
- Recognising sentiment from words: word2vec
- Synthesizing images: stackgan

The key to deep learning is to be able to represent data as arrays of numbers and to do **all** computations as array operations. That is we perform operations that act on all elements of an array simultaneously.

1.0.7 Vectorisation: one operation, many data

The practice of writing code which acts on arrays of values simultaneously is called **vectorised computation**. It is a special case of **parallel** computing, where we restrict ourselves to numerical operations on fixed size arrays. Modern CPUs have numerous **vectorised** instructions to perform the same operation on many numbers at once (e.g. MMX, SSE, SSE2, SSE3 on x86, NEON on ARM, etc.). This is called **Single Instruction Multiple Data**.

GPUs The major importance of vectorised computation is that **graphics processor units** (GPUs) are by far the most powerful computational units in any modern computer or phone; they are essentially supercomputers on a card. They can perform calculation much more quickly the central processing unit (CPU).

GPUs are array processors **But** they are effectively big groups of very simple processors, which are able to deal very well with data in numerical arrays, but are very slow when working with other data structures. Anything that can be written as an operation on numerical arrays can be done at lightning speed on a GPU.

In fact, GPUs are basically devices that can do computations on numerical arrays, **and that's it**. To write (efficient) GPU code, you need to write code in terms of numerical arrays.

Spreadsheet-like computation Array types are much like entire *spreadsheets in a single variable*, which you can perform standard spreadsheet operations on, like:

- tallying up columns
- selecting values which have a certain range
- plotting charts
- joining together several sheets

The abstraction of array types makes it easy to do what are complex operations with a standard spreadsheet. And they work on data beyond just 2D tables.

---

2 ndarray

2.1 Our basic datatype

There are several basic data types you will have encountered in your Computing Science/Math/Programming degree so far, including:

- lists (sequence types)
- strings (character sequences)
- dictionaries/hash tables (maps)
- classes/structures (record types)
- trees

Many languages focus on a central data type, for example: (very roughly – this is *not* definitive!)

- Lisp, Scheme -> **lists**
- Java, C++, Smalltalk, Objective-C -> **classes**
- Assembly -> **integers**
- C -> **pointers, structs**
- Lua -> **tables**
- Javascript -> **objects**
- Perl -> **strings**
- Haskell -> **mixed immutable (lists, ADTs, etc.)**
- Python, Ruby -> **mixed**
- APL, J, K, GLSL, HLSL -> **arrays**

2.2 ndarrays

The fundamental data type for this course is the **multidimensional numerical array**. This is a very powerful data type, although simple in structure, there are great many operations that can be done elegantly with an array.

We will call these arrays **ndarrays** (for *n-dimensional arrays*) or sometimes **tensors** (in reference to the mathematical object which generalises vectors and matrices to higher orders), which some people use.

...
