A Few useful Python/Numpy and Matlab primitives

September 13, 2022

Python and Matlab are easy languages to learn, but if one is not careful they can both be extremely slow – up to 400 times slower than C. In order to get them to perform well, you need to use *array programming* primitives/building blocks: operations that work on whole vectors or sub-matrices at a time, when possible. When this is done with a little thought, you end up with code that is nearly as fast as `C` or `Fortran`, but much simpler and more readable.

Slices

Slices are used to address regions of arrays. For example, in C or Fortran, one would write a forward substition with two for loops:

Forward Substitution in `C/C++`:

```
for(int k=0;k<n;k++){
  y[k] = b[k];
  for(int j=0;j<k;j++)
    y[k] -= L[j,k]*y[j];
}
```

In Python and Matlab, this would be very slow. Instead, the inner loop can be replaced by a fast dot-product imported from Numpy:

Forward Substitution in `Python`

```
for k in range(n):
  y[k] = b[k] - dot(L[0:k,k],y[0:k])
```

The notation "`a:b`" is called a *slice*, and is how you pick out sub-arrays. `L[0:k,k]` means "the slice of `L` from `L[0,`k`]` to `L[`k`-1,`k`]`", i.e., we take the dot-product between the first k elements of the k^{th} column of `L`, and the first k elements of `y`.

If we didn't happen to have the hyper-optimized `dot`-function, we could also have written it nearly as efficiently implemented using Numpy's sum function:

```
for k in range(n):
  y[k] = b[k] - sum(L[0:k,k]*y[0:k])
```

The multiplication operation on two arrays with matching shapes multiplies element-wise and returns a new array of the same shape.

Outer product

The outer product is often very useful. It produces all the products as follows:

$$\texttt{outer}(\mathbf{a},\mathbf{b}) = \begin{bmatrix} a_1b_1 & \cdots & a_mb_1 \\ \vdots & \ddots & \vdots \\ a_1b_n & \cdots & a_mb_n \end{bmatrix} \tag{1}$$

For example, in the Householder method, a reflection can be applied to *all columns of a matrix* in one go, without a loop, by using Numpy's outer product:

```
def apply_reflection(v,A):  # v:          n x 1
    c = -2*dot(v,A)         # c=v^T A:    1 x m
    A += outer(v,c)         # outer(v,c): n x m
```

Then, the `for`-loop

```
for j in range(k,M):
    apply_reflection(v,R[k:N,j])
```

simply becomes

```
  apply_reflection(v,R[k:N,k:M])
```

and the entire Householder transform can be written with only the outer `for`-loop, which will run significantly faster, and is easier to read as well.

Broadcasting with newaxis

One is not always lucky enough that there already exists an optimized vector function that does exactly what's needed. Sometimes, we have to build it ourselves.

A second important building block is *newaxis*: this tells Python/Matlab to introduce a virtual axis, along which the elements are implicitly copied. Technically, it adds a dummy axis with a stride of 0, i.e., increasing the index along this axis doesn't change the memory position that it points to. This may all seem a little abstract, but hopefully this example illustrates it:

Suppose we wanted to program matrix multiplication[1] with array operations: Given $\mathbf{A} : m \times p$, $\mathbf{B} : p \times n$ we want to calculate $\mathbf{AB} : m \times n$.

$$(AB)_{ij} = \sum_{k=1}^{p} A_{ik}B_{kj} = \sum_{k=1}^{p} A_{ik}B^T_{jk} = \sum_{k=1}^{p} A_{ijk}B^T_{ijk} \tag{2}$$

In the final equation, we've added a *newaxis* to both $\mathbf{A}$ and $\mathbf{B}$, indicated with red: The index j from B^T_{jk} is added to A_{ik} and i to B^T_{jk} so that they are indexed identically as A_{ijk} and B^T_{ijk}. Translated into python, this looks like

```
  def mmul(A,B): return sum(A[:,newaxis,:]*B.T[newaxis,:,:],axis=2)
```

Similarly, if we had not had the outer product available, reflecting all the columns of a matrix about the same vector `v` could look like:

```
def apply_reflection(v,:
    c   = -2*dot(v,A)             # c:   1 x m
    A += v[:,newaxis]*c[newaxis,:] # A:   n x m
```

corresponding to

$$\begin{aligned} c_j &= -2\mathbf{v}^T\mathbf{A}_j \\ A_{ij} &+= v_i c_j = v_{ij} v_{ij} \end{aligned} \tag{3}$$

[1]As it happens, Python and Matlab of course have matrix-matrix multiplication, which both simply call the hyper-optimized linear algebra library BLAS.