The `affi` package

Tamás K Papp[*]

May 19, 2008

Abstract

This is a tutorial for the `affi` (affine indexing) package, that provides convenience functions for traversing slices of arrays, optionally with index permutations and other convenient transformations. A driver clause for `iterate` is provided, along with `map-subarray`, a simple yet powerful function for mapping arrays into each other using affine indexes.

1 Introduction

Even though this is not widely known, Common Lisp is an amazing language for numerical computation. The language constructs themselves are very powerful, and efficient compilers exist that allow the user to run his code very quickly even without tedious optimization, while allowing for the possibility of the latter when the need arises. Also, its interactive nature makes the language ideal for debugging glitches in the algorithm or exploring problems quickly.

However, Common Lisp lacks a few features that users of languages like Matlab, R or even Fortran are used to. One of these is the possibily of "slicing" arrays: for example, treating a column of a matrix as a vector, a rectangular submatrix of a matrix as another matrix when passing to another function, assigning rectangular sections of arrays to each other and performing operations on them with ease, with appropriate syntactic sugar and usually corresponding fast code.

This package is an attempt to fill this gap and implement a similar — but not identical — feature in Common Lisp called affine indexes. Affine indexes and appropriate helper functions allow CL users to copy, modify, transpose rectangular slices of arrays with ease. Instead of implementing syntactic sugar like `3:5` for elements from 3 to 5, I decided that a functional approach would be better. Thus instead of doing the tasks with specialized functions, affine indexes allow the user to generate a *walker*, which is essentially a function that yields successive indexes for the flat vector that holds the elements of an array. Section 2 presents affine indexes and walkers, and Section 3 explains some predefined operations on affine indexes (including subranges). Section 4 discusses useful functions constructed with affine indexes.

2 Affine indexes and walkers

An array of rank n can be conceptualized as a function that maps a set of integer subscripts $(s_1, \ldots, s_n)$, which are constrained by the dimensions of the array, to an element. It is useful to

[*]E-mail: tpapp@princeton.edu

imagine that the elements in the array are stored in a flat vector with a single index i, and think of this function as a composite of two other functions: one $(s_1, \dots, s_n) \mapsto i$ which maps the subscripts to the index in the flat vector, and another which retrieves the element of the given index i from this vector. In Common Lisp, `aref` does these two things in a single step, but like above, its operation can be broken up into two stages, implemented as the functions `array-row-major-index` and `row-major-aref`.

Affine indexes are nothing more than a class of integer functions that are more general than those used for row-major arrays. Consider a positive integer n (which we will call the *rank*), and a *domain*

$$D(d_0, \dots, d_{n-1}) = \big([0, 1, \dots, d_0) \cap \mathbb{N}\big) \times \cdots \times \big([0, 1, \dots, d_{n-1}) \cap \mathbb{N}\big)$$

where d_i is the *size* of dimension i. An affine index is a mapping $a : D \to \mathbb{Z}$, defined as

$$a(s_0, \dots, s_{n-1}) = C + c_0 s_0 + \cdots + c_{n-1} s_{n-1}$$

where $C \in \mathbb{Z}$ is called the *constant*, and $c_i \in \mathbb{Z}, i = 0, \dots, n-1$ are the *coefficients*.

An affine index is characterized by its domain, constant and coefficient. It is easy to see that affine indexes are quite general and can be used to implement row-major (or column-major, for that matter) indexing: for example, the $0 + 12s_0 + 4s_1 + 1s_2$ would index an array with dimensions `#(2 3 4)` (with the appropriate domain). Moreover, affine indexes allow us to index rectangular sub-arrays, or even reverse elements or permute indexes (transposing a matrix is a special case of the latter).

Before I discuss functions in this package, two general remarks are in order. First, all functions in the package are properly documented, so you can read their documentation string for a complete description — examples in this tutorial are just meant to whet your appetite and make you aware of features, and are not meant to be extensive descriptions. Second, the function names are deliberately kept short, because I intend that they are used with their namespace prefix and not imported to the current namespace using `use-package` — hence in your code, you would write `affi:drop` instead of `drop`. Examples below are run from the `affi` namespace for simplicity, but generally you should use another one (`cl-user:`, your own package, etc).

You can create an affine index conforming to a list if dimensions characterizing a row-major array or the array itself using `make-affi`:

```
AFFI> (make-affi '(2 3 4))
#<AFFI domain #(2 3 4), const 0, coeff #(12 4 1) {B0C4D61}>
AFFI> (make-affi (make-array '(5 6 7)))
#<AFFI domain #(5 6 7), const 0, coeff #(42 7 1) {B3B8501}>
```

Or should you find yourself in need of interfacing with some vile language that uses column-major indexing, call `make-affi-cm` on a list of indexes:

```
AFFI> (make-affi-cm '(2 3 4))
#<AFFI domain #(2 3 4), const 0, coeff #(1 2 6) {B7340F1}>
```

How can you use affine indexes? If you just want to calculate the index of a single element, you can use

```
AFFI> (defparameter *a* (make-affi '(2 3 4)))
AFFI> (calculate-index *a* #(0 1 2))
6
```

But when you are traversing an array, it is better to use a *walker*. A walker is simply a function that traverses the subscripts in a particular (lexicographic, with rightmost subscripts changing the fastest) order, and returns the index corresponding to each:

```
AFFI> (defparameter *w1* (make-walker (make-affi '(2 2))))
AFFI> (funcall *w1*)
0
AFFI> (funcall *w1*)
1
AFFI> (funcall *w1*)
2
AFFI> (funcall *w1*)
3
AFFI> (funcall *w1*)
NIL
```

On each call, it returns the *current* index, and *then* increments its internal counter. Once it runs out of elements, it will just return `nil`. `make-walker` actually returns two functions, and the second one allows you to query the index without incrementing it.

Of course, a row-major walker is not a particularly interesting one, as it justs lists consecutive integers.[1] Using the convenience function `test-walker`,[2] this is how a column-major walker would traverse the indexes:

```
AFFI> (test-walker (make-walker (make-affi-cm '(2 3))))
0 2 4 1 3 5
```

Finally, there are a few convenience functions: `size` returns the number of integers indexed, `range` returns the smallest and largest integers indexed, and `rank` returns the rank.

When using two walkers simultaneously, it is a good idea to check if they are *conformable*. You can do that using `check-conformability`, which recognizes three types of conformability (in order of decreasing strictness):

- `:strict` requires that the two domains are exactly the same
- `:dropped` checks if they are the same when we drop dimensions of size 1
- `:size` just checks the size of the two ranges.

`dropped` is usually a good compromise between catching errors and convenience, it still allows you to treat a 6×1 matrix and a 6-element vector as conformable (`:strict` would not), but would catch obvious errors like trying to walk the former matrix alogn with a 7-element vector.

For convenience, there is an `iterate` driver for affine indexes:[3]

[1]This is of course recognized, and walkers are optimized to recognize contiguous blocks of integers. [FIXME: not yet, still working on it]

[2]This functions displays the indexes returned by a walker until it encounters `nil`. You can also call it directly on an affine index.

[3]I prefer to put a `:` before keywords other than the first one when using iterate, you can just write `(for i in-affi (make-affi '(2 3)))` if you prefer.

```
(iter
  (for i :in-affi (make-affi '(2 3)))
  (collecting i) ;; evaluates to (0 1 2 3 4 5)
```

`affi` instances are *not* meant to be modified directly: all utility functions below leave the original argument intact and create a new instance. You can safely assume that an `affi` instance is immutable, and the slots are not exported from the package. Nevertheless, they can be accessed `get-const`, `get-coeff` and `get-domain`, the latter two makes copies of the internal vectors.

3 Operations

Of course, affine indexes become useful when we consider transformations on them. The `affi` package has the following predefined transformations:

`permute`, which allows us to shuffle indexes around, for example, `(permute affi '(1 0))` is equivalent to transposing a matrix (actually, `transpose` is defined too), but works for affine indexes of any rank:

```
AFFI> (defparameter *affi* (make-affi '(2 3 4)))
AFFI> *affi*
#<AFFI domain #(2 3 4), const 0, coeff #(12 4 1) {BCEBFF9}>
AFFI> (permute *affi* '(1 2 0))
#<AFFI domain #(3 4 2), const 0, coeff #(4 1 12) {A903111}>
```

`drop`, which allows the elimination of degenerate dimensions (of size 1). It has an optional argument which is supposed to contain the list of dimensions that are considered for elimination:

```
AFFI> (drop (make-affi '(1 1 2 3)))
#<AFFI domain #(2 3), const 0, coeff #(3 1) {A9D5441}>
AFFI> (drop (make-affi '(1 1 2 3)) '(1))
#<AFFI domain #(1 2 3), const 0, coeff #(6 3 1) {AA04371}>
```

When this last argument is `t` (the default), all degenerate dimensions will be eliminated.

Finally, the most powerful operation is `subrange`, which constrains and/or transforms an affine index given a list of range specifications, one for each dimensions.

One way to specify a range is using a two-element list: for example, `(3 5)` denotes the subscripts from 3 to 5, inclusive. Negative numbers are counted from the dimension: if a subscript is constrained to be below 6, -3 will denote 3. You can mix positive and negative subscripts, as in `(3 -1)`. If the two subscripts are in decreasing order (after being converted to nonnegative integers), the indexes will be *reversed* along this dimension.

Also, there are some shortcuts: specifying a single integer `n` is equivalent to `(n n)`, selecting a single subscript. `:all` select all subscripts, and `:rev` selects them in reverse order. Some examples:

```
AFFI> (defparameter *affi* (make-affi '(4 5)))
AFFI> (test-walker *affi*)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
AFFI> (test-walker (subrange *affi* (:all :rev)))
4 3 2 1 0 9 8 7 6 5 14 13 12 11 10 19 18 17 16 15
AFFI> (test-walker (subrange *affi* '((1 -1) 2)))
7 12 17
```

```
AFFI> (test-walker (subrange *affi* '((-1 1) 2)))
17 12 7
```

4 map-subarray

The function

```
(defun map-subarray (source target
                     &key source-range target-range permutation
                     (drop-which t)
                     (conformability 'dropped) (key #'identity)
                     (target-element-type (array-element-type source)))
```

combines the above transformations and allows us to copy ranges from one array to another, optionally permuting the indexes.

If `target` is `nil`, a new array is created with element-type `target-element-type` and is returned by the function (in this case, you can't specify `target-range`, and degenerate dimensions will be dropped according to `drop-which`). Otherwise, conformability is checked between the two affine indexes (which are calculated internally using the given ranges and permutation) using `conformability`.[4] `key` is called on each element before copying and the result is copied in the target array.

Some examples:

```
AFFI> *m* ;; the original array
#2A((0 1 2 3) (4 5 6 7) (8 9 10 11))
AFFI> (map-subarray *m* nil :permutation '(1 0)) ;; transpose
#2A((0 4 8) (1 5 9) (2 6 10) (3 7 11))
AFFI> (map-subarray *m* nil :source-range '(all 1)) ;; 2nd column
#(1 5 9)
AFFI> (map-subarray *m* nil :source-range '(all 1) :drop-which nil)
#2A((1) (5) (9))
AFFI> (map-subarray *m* nil :source-range '((1 2) rev))
#2A((7 6 5 4) (11 10 9 8))
```

If you check the source code of `map-subarray`, you will notice that it is very straightforward. While the function is useful as it is, the tools in this package make writing similar functions easy.

[4]See `check-conformability` three kinds of conformability recognized by `affi`.