SOOM Extensions for NetEpi Analysis

Release 0.11

Dave Cole and Tim Churches

December 5, 2001

E-mail: djc@object-craft.com.au TCHUR@doh.health.nsw.gov.au

Abstract

The SOOM (Set Operations on Ordinal Mappings) Extensions package is a collection of Numeric Python optimisations which accelerate a number of performance critical SOOM functions.

See Also:

Numerical Python Web Site
(http://numpy.sourceforge.net/)
for information on Numpy

Contents

1 Installation

1.1 Prerequisites

- Python 1.5.2 or later.

- C compiler

  The SOOM package contains extension module code written in C. The extension module code is used by the Python wrapper module SOOM.py.

- Numeric Python

  The SOOM package uses the Numpy extension modules for Python.

  Numpy can be downloaded from http://numpy.sourceforge.net/.

1.2 Installing

The SOOM package the distutils package so all you need to do is type the following command as root:

```
python setup.py install
```

If you have problems with this step make sure that you contact the package author so that the installation process can be made more robust for other people.

1.3 Testing

The most simple way to test the SOOM package is via the interactive Python prompt.

```
>>> from soomfunc import *
>>> from Numeric import *
>>> intersect(array(range(10)) + 4, array(range(5)) * 2)
array([4, 6, 8])
```

2 soomfunc

The soomfunc module contains a collection of set operations which can be performed on Numpy arrays:

The soomfunc module contains the following:

__version__
A string which specifies the version of the soomfunc module.

unique(*a* [, *b*])
Remove all duplicate values from the sorted rank-1 array passed as the *a* argument returning a new rank-1 array of unique values. If the optional second argument *b* is supplied it will be used to store the result.

```
>>> from soomfunc import *
>>> from Numeric import *
>>> unique(array([1,2,3,3,4,4,5]))
array([1, 2, 3, 4, 5])
```

intersect(*a*, *b* [, ...])
Return the unique intersection of the sorted rank-1 arrays passed. All arrays must have the same typecode.

The function finds the smallest array then allocates an array of that length to hold the result. The result is primed by finding the intersection of the smallest array with the first array in the argument list (or the second if the first is the smallest). The result is then incrementally intersected in-place with the remainder of the arrays generating a new result.

With each intersection, the function compares the length of the two arrays. If one array is at least three times larger than the other, the `sparse_intersect()` method is used, otherwise the `dense_intersect()` method is used.

```
>>> from soomfunc import *
>>> from Numeric import *
>>> intersect(array([1,2,3]), array([2,3]), array([3,4]))
array([3])
```

`sparse_intersect`(*a*, *b* [, ...])

Return the intersection of the sorted rank-1 arrays passed. All arrays must have the same typecode. This generates the same result as the `intersect()` function.

When finding the intersection of two arrays it uses a binary search to locate matching values. The first point for the binary search is determined by dividing the number of elements remaining in the target array by the number of elements remaining in the source array.

This method can be thousands of times faster than the `dense_intersect()` function. For arrays of similar size, the `dense_intersect()` function will be more than twice as fast as this function.

You are encouraged to use the `intersect()` function as it will automatically use the best intersection function.

`dense_intersect`(*a*, *b* [, ...])

Return the intersection of the sorted rank-1 arrays passed. All arrays must have the same typecode. This generates the same result as the `intersect()` function.

When finding the intersection of two arrays it steps through both arrays one value at a time to locate matching values. The `sparse_intersect()` function can be thousands of times faster than this function for some inputs.

You are encouraged to use the `intersect()` function as it will automatically use the best intersection function.

`outersect`(*a*, *b* [, ...])

Return the unique symmetric difference of the sorted rank-1 arrays passed. All arrays must have the same typecode.

Steps through all arrays in lock-step and finds all values which do not occur in every array. This is the exact opposite of the `intersect()` function.

```
>>> from soomfunc import *
>>> from Numeric import *
>>> outersect(array([1,2,3]), array([2,3]), array([3,4]))
array([1, 2, 4])
```

`union`(*a*, *b* [, ...])

Return the unique union of the sorted rank-1 arrays passed. All arrays must have the same typecode.

Steps through all arrays in lock-step and finds all unique values across every array.

```
>>> from soomfunc import *
>>> from Numeric import *
>>> union(array([1,2,3,3]), array([2,2,3]), array([3,4]))
array([1, 2, 3, 4])
```

`difference`(*a*, *b* [, ...])

Return the result of subtracting the second and subsequent sorted rank-1 arrays from the first sorted rank-1 array. All arrays must have the same typecode.

```
>>> from soomfunc import *
>>> from Numeric import *
>>> difference(array([1,2,3,3]), array([2,2,3]), array([3,4]))
array([1])
```

valuepos(*a*, *b*)
Return an array of indexes into rank-1 array *a* where the values in rank-1 array *b* appear. Both arrays must have the same typecode.

```
>>> from soomfunc import *
>>> from Numeric import *
>>> valuepos(array([2,2,3,3,4,5,6]),array([3,5]))
array([2, 3, 5])
```

preload(*a* [, *num* = `-1`])
Step backwards over *num* entries of the array in the *a* argument forcing the pages into memory. If *num* is less than zero or greater than the length of the array, the entire array is scanned.

This function is used to optimise the use of arrays stored in memory mapped blobs.

```
>>> from soomfunc import *
>>> from Numeric import *
>>> preload(array([2,2,3,3,4,5,6]))
```

3 soomarray

The `soomarray` module contains a Python wrapper around the `blobstore` extension module.

The `soomarray` module contains the following:

exception `Error`
Exception raised when unexpected data is located in the BLOB store.

class `MmapArray`(*blob*)
Return a new instance of the `MmapArray` class using the `Blob` object specified in the *blob* argument.

class `ArrayDict`(*filename, mode* = `'r'`)
Return a new instance of the `ArrayDict` class.

The *filename* argument specifies the name of the file which will be used to store and retrieve memory arrays. The *mode* argument is can be `'r'` to access an existing file in read only mode, `'r+'` to access an existing file in read-write mode, or `'w+'` to create a new file in read-write mode.

3.1 ArrayDict Objects

These objects behave like a Python dictionary. Each item stored must be either a plain Numpy array, or a masked array from the MA package.

Internally the object manages a `BlobStore` object in which the BLOB at index zero contains a pickled dictionary which yields the index of the BLOB which contains the Numpy array. For masked arrays, the BLOB referenced by the dictionary contains the index of the mask array in the `other` BLOB member.

You must be careful when modifying the contents of the `ArrayDict` object because the internal `BlobStore` may need to grow the memory mapped file. Growing the file will invalidate all of the addresses contained in arrays retrieved from the file. The `MmapArray` objects are able to automatically handle this event, but any arrays derived from these will not. If you slice and dice `MmapArray` objects then modify their containing `ArrayDict`, you will get a segmentation fault.

```
>>> from Numeric import *
>>> from soomarray import ArrayDict
>>> ad = ArrayDict('file.dat', 'w+')
>>> ad['a'] = array(xrange(100))
>>> ad.keys()
['a']
>>> len(ad)
1
>>> ad['b'] = array(range(10))
>>> ad['b']
[0,1,2,3,4,5,6,7,8,9,]
>>> from soomfunc import *
>>> intersect(ad['a'], ad['b'])
[0,1,2,3,4,5,6,7,8,9,]
```

`ArrayDict` objects have the following interface:

`__del__``()`
: When the last reference to the object is dropped, the object will repickle the internal array dictionary to BLOB zero in the associated `BlobStore` object.

`__getitem__``(`*key*`)`
: Returns a `MmapArray` object indexed by the *key* argument. This is called to evaluate `ad[key]`. Raises `KeyError` if *key* is not a key in the dictionary.

`__setitem__``(`*key, a*`)`
: Called to implement assignment of *a* to `ad[key]`.

`__delitem__``(`*key*`)`
: Called to delete `ad[key]`. Raises `KeyError` if *key* is not a key in the dictionary.

`__len__``()`
: Returns the number of items stored in the dictionary. A masked array counts as one item.

`clear``()`
: Deletes all contents.

`get``(`*key* [, *default* = *None*]`)`
: Implements the dictionary `get()` method.

`has_key``(`*key*`)`
: Returns TRUE if *key* is a valid key in the dictionary.

`keys``()`
: Returns a list containing all of the keys in the dictionary.

`values``()`
: Returns a list containing all of the arrays in the dictionary.

`items``(`*key*`)`
: Returns a list of tuples containing all `(key, array)` pairs in the dictionary.

3.2 MmapArray Objects

Implements a Numpy array which has memory mapped data within a `BlobStore`. These objects act exactly like the `UserArray` object from the Numpy package (they should since their code was pasted from the Numpy source).

Internally they wrap a `Blob` object which was retrieved from the parent `BlobStore` object. Each time the array is used the object calls the `Blob` `as_array()` method to refresh the address of the array data.

Apart from the standard Numpy array methods the `MmapArray` class implements the following:

`append``(`*a*`)`
: This method appends the array in the *a* argument to the BLOB. The `BlobStore` container will resize as

required. This is a handy way to build up an array which is too large to fit in memory.

```
>>> from Numeric import *
>>> from soomarray import ArrayDict
>>> ad = ArrayDict('file.dat', 'w+')
>>> ad['a'] = array(xrange(100))
>>> a = ad['a']
>>> a.append(xrange(100))
>>> len(a)
200
```

4 blobstore

The `blobstore` extension module contains a fairly thin wrapper on top of the memory mapped BLOB storage which is implemented in the Storage API.

The `blobstore` module contains the following:

`__version__`
A string which specifies the version of the `blobstore` module.

`open`(*filename* [, *mode* = `'r'`])
Open the file named in the *filename* argument using the specified *mode*. Returns a `BlobStore` object.

```
>>> import blobstore
>>> bs = blobstore.open('file.dat')
```

4.1 BlobStore Objects

These objects are used to manage a sequence of BLOB items. The object behaves much like a Python list. Each element in the list is a `Blob` object.

`BlobStore` objects have the following interface:

`__len__`()
Returns the number of blobs in the blob store.

`__getitem__`(*i*)
Uses the blob store sequence to retrieve the blob descriptor indexed by *i* and returns it as a new instance of the `Blob` class. The blob data is accessed via the `Blob` object.

`append`()
Reserves an index for a new blob in the blob store. The reserved index is returned. Note that no data will be allocated in the blob store until data is saved to that blob index.

`get`(*i*)
Retrieves the raw blob descriptor indexed by *i* and returns it as a new instance of the `Blob` class. This by-passes the blob store sequence and indexes directly into the blob table. This allows you to see the blobs which are used to store the blob sequence and table as well as free blobs. The `Blob` objects obtained this way cannot be used to load or save data.

`free`(*i*)
Free the storage associated with the blob in the sequence at index *i*.

`usage`()
Returns a tuple which contains two numbers; the amount of data storage allocated in the blob store, and the amount of space free or wasted in the blob store.

`header`()
Returns a tuple containing the following values;

table_size	Number of blob table entries allocated.
table_len	Number of blob table entries in use.
table_index	Table index of the blob which contains the blob table.
table_loc	File offset of the start of the blob table.
seq_size	Number of blob sequence entries allocated.
seq_len	Length of the blob sequence.
seq_index	Table index of the blob which contains the blob sequence.

To understand these numbers you should refer to the documentation of the Storage API.

4.2 Blob Objects

Blobs retrieved from a `BlobStore` object are returned as `Blob` objects. The `Blob` objects provide access to the data associated with a blob in a blob store. The `Blob` object retains a reference to the containing `BlobStore` and they also store their index within the blob store.

`Blob` objects have the following interface:

`len`
This read-only member contains the length of the blob data.

`size`
This read-only member contains the allocated size of the blob. The blob `size` will always be at least as large as the blob `len`. Blob allocations are aligned to the size of an integer.

When a blob is resized or freed it can leave some free space in the blob store. When allocating a new blob, the smallest free area which can contain the new data will be used in preference to growing the file.

`loc`
This read-only member contains the offset within the file of the blob data.

`status`
This read-only member contains the status of the blob. The values and their meaning are:

0	The blob is free space.
1	The blob contains the blob table.
2	The blob contains the blob sequence.
3	The blob contains user data.

`other`
This integer member can be used by your code. The `ArrayDict` class uses it to link the mask and data parts of a masked array.

`type`
This integer member can be used by your code.

`as_array``()`
Returns a Numpy array object which uses the memory mapped blob data as the contents of the array. The array object is cached internally, so you can call this without penalty.

Every time you call this method the cached Numpy array will have its internal data pointer updated to protect against blob store resizing.

`as_str``()`
Returns a Python string which is constructed using the blob data. The string allocates its own copy of the data. The string is cached internally, so you can call this without penalty.

`save_array``(`*a*`)`
Save the array passed in the *a* argument in the blob associated with this object.

`append_array``(`*a*`)`
Append the array passed in the *a* argument to the blob associated with this object.

`save_str``(`*s*`)`

Save the string passed in the *s* argument in the blob associated with this object.

5 Storage API

The `blobstore` extension module is built on top of the Storage API. This API provides a blob storage via a memory mapped file.

The memory mapped file managed by the Storage API is logically arranged like the following figure:

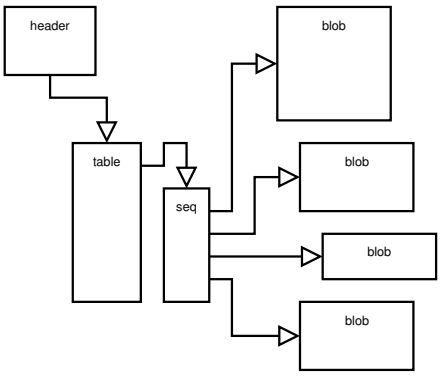

Figure 1: Memory Mapped File Layout

There is a simplicifcation in the above diagram. The blob containing the blob sequence (labelled “seq”) contains blob numbers which are then used as indexes back into the blob table to locate the actual blob data.

At offset zero in the file is the file header which is a `StoreHeader` structure from which everything else is located.

`StoreHeader`

This structure which is contained at offset zero in the memory mapped file. Its definition, found in ‘storage.h’, is:

```
typedef struct {
    off_t table_loc;            /* where the blob table is stored */
    int table_size;             /* allocated table size */
    int table_len;              /* entries used in the table */
    int table_index;            /* blob which contains the blob table */

    int seq_index;              /* blob which contains the array lookup */
    int seq_size;               /* allocated sequence size */
    int seq_len;                /* number of blobs in the sequence */
} StoreHeader;
```

The `table_loc` member contains the file offset of a special blob which stores the blob table. The blob table contains an array of `BlobDesc` structures. Each `BlobDesc` describes one blob in the file. All blob descriptors are kept together to minimise the amount of data which will be paged in while accessing the blob meta-data.

As the blob table grows it will move around in the file. The `table_size` member records the number of array elements allocated in the blob table while `table_len` records the length of the blob table. Once allocated a blob cannot be deleted, it can only be marked as free space.

The `table_index` member records the blob table index of the blob which contains the blob table.

The blob table entries are always arranged in strict ascending sequence of the offset of the data that they describe. This means that as blobs are resized they may be handled by different entries in the blob table.

To present external code with blob numbers which do not change after the initial blob allocation, a blob sequence is maintained. The blob sequence is a simple indirection table which translates external blob

index into an internal blob table index. This allows the table entry for a blob to change without affecting external code.

The `seq_index` member records the blob table index of the blob which describes the blob sequence.

`BlobDesc`

The blob table is constructed from an array of these structures. Its definition, found in 'storage.h', is:

```
typedef struct {
    off_t loc;                  /* location of the blob */
    size_t size;                /* size of blob */
    size_t len;                 /* length of blob */
    int status;                 /* status of blob */
    int type;                   /* user: type of blob */
    int other;                  /* user: index of related blob */
} BlobDesc;
```

The `loc` member contains the file offset of the start of the blob data. The `size` member contains the allocated size of the blob. The `loc` member of the next `BlobDesc` *is always equal to* `loc + size` of this `BlobDesc`. The `len` member contains size of the blob requested by the external code.

The `status` member records the status of this space controlled by this blob. The value will be one of:

BLOB_FREE	The blob is free space.
BLOB_TABLE	The blob contains the blob table.
BLOB_SEQUENCE	The blob contains the blob sequence.
BLOB_DATA	The blob contains user data.

The `type` and `other` members are not used by the Storage API. Applications are free to store whatever they wish in these members.

`MmapBlobStore`

This structure is allocated by the `store_open` function when the blob storage is opened. Its definition, found in 'storage.h', is:

```
typedef struct {
    int mode;                   /* file open mode */
    int fd;                     /* file descriptor */
    int prot;                   /* mmap prot */
    StoreHeader *header;        /* address of start of store */
    size_t size;                /* size of file */
    int cycle;                  /* increment when file remapped */
} MmapBlobStore;
```

`MmapBlobStore * store_open(`*char *filename, char *mode*`)`

Opens the file specified in the *filename* argument using the mode specified in the *mode* argument.

`'r'`	Open an existing file in read only mode.
`'r+'`	Open an existing file in read-write mode.
`'w+'`	Create a new file in read-write mode.

If successful the function allocates and returns a `MmapBlobStore` structure which is passed as the first argument to all other Storage API functions. If the operation fails for any reason the function will set a Python exception and will return `NULL`.

`int store_close(`*MmapBlobStore *sm*`)`

Closes the blob store and frees the `MmapBlobStore` structure. On failure the function will set a Python exception and will return `-1`. On success the function returns `0`.

`int store_get_header(`*MmapBlobStore *sm, StoreHeader *header*`)`

Copies the contents of the header at the start of the memory mapped file into the buffer supplied as the *header* argument.

The function returns `0` on success and `-1` on failure.

`void` **`store_usage`**(*MmapBlobStore *sm, size_t *used, size_t *unused*)
Traverses the blob table and returns the amount of space in use in the *usage* argument and the amount of space which is either free or wasted in the *unused* argument.

`size_t` **`store_compress`**(*MmapBlobStore *sm*)
Compresses the blob store to remove any space contained in free blobs or at the end of existing blobs. Note this function is not current implemented.

Returns `0` on success and code-1 on failure.

`int` **`store_cycle`**(*MmapBlobStore *sm*)
Every time the file is remapped due to size changes, the `cycle` member of the `MmapBlobStore` is incremented. This function returns the current value of the `cycle` member.

`int` **`store_num_blobs`**(*MmapBlobStore *sm*)
Returns the number of blobs in the blob sequence.

`BlobDesc *` **`store_get_blobdesc`**(*MmapBlobStore *sm, int index, int from_seq*)
Returns a pointer to the blob table entry which describes the blob specified by the *index* argument. If the *from_seq* argument is non-zero then the *index* argument will be used to look up the blob sequence to obtain the index into the blob table. When *from_seq* is zero the index is used directly to index the blob table.

If the *index* argument is outside the bounds of the blob sequence or table the function will set a Python `IndexError` exception and will return `NULL`.

`void *` **`store_blob_address`**(*MmapBlobStore *sm, BlobDesc *desc*)
Returns the address of the start of the data for the blob descriptor passed in *desc*. The `store_blob_blobdesc()` returns values suitable for use as the *desc* argument.

The function preforms no checking on the arguments.

`int` **`store_append`**(*MmapBlobStore *sm*)
Allocates a new entry in the blob sequence and returns the index. Note that no blob data is allocated until `store_blob_resize()` is called to allocate some space for the blob.

If the operation fails for any reason the function will set a Python exception and will return `-1`.

`int` **`store_blob_size`**(*MmapBlobStore *sm, int index*)
Return the length of the blob identified by the blob sequence index in the *index* argument.

`int` **`store_blob_resize`**(*MmapBlobStore *sm, int index, size_t data_len*)
Resize the blob identified by the blob sequence index in the *index* argument to be the new size specified in the *data_len* argument.

If you are growing the blob, the address of the blob will almost certainly change after calling this function. You must call `store_blob_blobdesc()` and `store_blob_address()` to obtain a valid address for the blob.

`int` **`store_blob_free`**(*MmapBlobStore *sm, int index*)
Free the blob identified by the blob sequence index in the *index* argument. The space will be reused.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

O

P

S

T

U

V