PyLinear

Release 0.92

Andreas Klöckner

July 9, 2006

Division of Applied Mathematiccs, Brown University
Campus Box F, Providence, RI 02912, USA

Abstract

PyLinear provides a comprehensive and extensible environment for linear algebra in Python.

It is comprehensive in that it offers sparse and dense matrices of real and complex types, along with a full set of operations on them.

It is extensible in that it can trivially be programmed in Python, but it may just as trivially be extended in C++. If `mex` programming in Matlab always sounded too cumbersome, PyLinear provides a refreshing new (and easy) perspective. This allows a new type of hybrid numerical system, where non-performance-critical parts may remain in the high-level language, while inner loops may be easily moved to C++.

There is no shortage of Python packages that provide matrices and operations on them. Numerical Python, numarray and the new NumPy (formerly SciPy Core) are good examples. However, few of these packages focus exclusively on Linear Algebra, and even fewer provide crucial tools for numerical analysis, such as sparse matrices. PyLinear fills this niche.

CONTENTS

1 Introduction **1**

2 Installation **3**
2.1 Checking prerequisites . . . 3
2.2 Installing Boost Python . . . 4
2.3 Installing the Boost UBlas Bindings . . . 5
2.4 Installing PyLinear . . . 5

3 The `Array` types **7**
3.1 Types and Flavors . . . 7
3.2 Creating `Array`s . . . 8
3.2.1 Pickle support . . . 11
3.3 Accessing `Array` Properties . . . 11
3.4 Accessing `Array` Data . . . 12
3.4.1 Indexing . . . 12
Row-wise Access . . . 13
Strides . . . 14
Broadcasting . . . 14
3.4.2 Picking `Array`s apart . . . 15
Picking apart complex `Array`s . . . 15
Getting Noncontiguous Parts of an `Array` . . . 16
3.5 Basic `Array` Math . . . 16
3.5.1 Generalities on Binary Operations . . . 17
Type promotion . . . 17
Broadcasting . . . 17
3.5.2 Multiplicative (and related) Operators . . . 18
Related operators . . . 19
3.6 Elementwise Functions . . . 23
3.6.1 Linear Algebra . . . 25
3.7 Flavor-specific Functionality . . . 25
3.8 Matrix algebra . . . 26
3.9 Simple computational routines . . . 26

4 Matrix-Free Methods **29**
4.1 The `Operator` concept . . . 29
4.2 `Operator`s Form an Algebra . . . 29
4.3 Types of `Operator`s . . . 30
4.4 Implementing your own `Operator`s . . . 32

5 Numerics with PyLinear **33**
5.1 Querying available functionality . . . 33
5.2 Matrix computations . . . 34
5.3 Convenient helpers . . . 34

6 Extending PyLinear **35**
6.1 Implementing custom operations . . . 35
6.2 Implementing custom `Operator`s . . . 35

7 Differences to NumPy and numarray **37**

A Acknowledgements **39**

Index **41**

CHAPTER

ONE

Introduction

This chapter introduces the PyLinear Python extension and outlines the rest of the document.

PyLinear is a set of extensions to the Python programming language which allows Python programmers to efficiently manipulate matrices and vectors, the primary objects of linear algebra. It allows real and complex arithmetic, currently only in double precision. Dense as well as two types of sparse matrices are supplied, and a large variety of numerical algorithms, from eigensolvers, to singular value decomposition, direct sparse solvers to sparse eigensolvers are also furnished as part of an ever-growing standard library.

PyLinear's programming interface is similar to that of Numerical Python and numarray to ease porting, but differs in a few key aspects. The most notable such aspect is matrix multiplication. While the term `A*B` in Numeric means element-by-element multiplication, PyLinear changes this to mean conventional matrix-matrix and matrix-vector multiplication, to match customary uses in scientific computing, and following the example of languages such as Matlab[1]. Chapter 7 is dedicated to highlighting the differences between PyLinear and NumPy and its desecendents.

In very simple terms, PyLinear is a mapping of the operators supplied by Boost.UBlas into Python using the Boost.Python library. This has two implications that balance each other. First, PyLinear is no speed demon. It does have the right asymptotic complexity guarantees (i.e. operations that ought to be linear-time in fact are). That's the bad news. The good news is that since PyLinear is essentially a scripting language for Boost.UBlas, it is sheepishly easy to convert a slow inner loop from Python into C++, without losing much abstraction: The matrix and vector types as well as most operations are available in C++ with only slightly more difficult syntax than in Python. But if that is so, why would you want to use Python in the first place? Because it's high-level, safe and does not require the sometimes lengthy compile times of C++. And you need to convert *only* that slow inner loop! Since you, too, can use Boost.Python to bind that inner loop to Python (and still use PyLinear's facilities), there's no real need to move the whole system into C++. That way, Python can be the convenient and safe prototyping language for large computation systems written in C++.

[1] This has one important gotcha. Vector-vector multiplication (also known as the dot product) is *not* associative, i.e. if a, b and c are vectors, then typically $(a \cdot b) \cdot c \neq a \cdot (b \cdot c)$. (Note that the type of the parenthesized expression is scalar.) PyLinear will not reject code such as `a*b*c`, but its meaning is inherently undefined. Matlab gets around this limitation by introducing column and row vectors.

CHAPTER

TWO

Installation

This chapter helps you install PyLinear onto your computer.

2.1 Checking prerequisites

The first step in installing PyLinear is to make sure that you have the right software installed on your computer. You will need the following:

- Python is of course the most important prerequisite. Version 2.3 or newer will work.

  You can find Python at `http://www.python.org`.

- The Boost libraries. Versions 1.33 and up work fine. Section 2.2 will help you with the installation of this prerequisite.

  You can find Boost at `http://www.boost.org`.

- The Boost Numeric library bindings. Section 2.3 will help you with this.

  You can find the Boost Numeric bindings at `http://news.tiker.net/software/boost-bindings`.

- A good enough C++ compiler. GCC version 3.3 and up work fine.

  GCC versions 4.0 and better will compile PyLinear using much less memory and generally much faster than the older 3.x series. Using them is highly recommended. Note however that GCC 4.1 has a bug that affects Boost.UBlas. (see `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=28016`)

  GCC can be found at `http://gcc.gnu.org`.

Optionally, you may install the following libraries to augment PyLinear's functionality:

- The *Basic Linear Algebra Subprograms, better known as the BLAS. This will not enable any new functionality by itself, but is a prerequisite for many of the following libraries.*

  You can find the original Fortran BLAS at `http://netlib.org/blas`. A good, generic implementation is ATLAS at `http://math-atlas.sf.net`.

- *The* Linear Algebra Package, better known as *LAPACK. This will enable a few extra operations on dense matrices, such as finding eigenvalues or the singular value decomposition.*

  You can find LAPACK at `http://netlib.org/lapack`. ATLAS now includes a LAPACK implementation.

- UMFPACK, which requires BLAS, provides an efficient direct solver for linear systems involving sparse matrices.

  As of late, most major Linux distributions package UMFPACK as part of a bigger package called UFsparse. While this is (in my opinion) not such a great idea, it's a fact that PyLinear has to live with.

Newer versions of PyLinear therefore default to using UFsparse, but will still run with just UMFPACK (and COLAMD) installed.

Unfortunately, versions of the Boost Numeric Bindings prior to release 2006-04-30 (from the PyLinear's author's web site) defaulted to using an include file 'umfpack/umfpack.h', which is not possible within UFsparse, where UMFPACK headers are typically installed under '/usr/include/ufsparse/umfpack.h'. The above-mentioned release fixes this, but also requires you to mention the full path to 'umfpack.h' in 'siteconf.py'.

BIG FAT WARNING: If you get the Boost Numeric Bindings from CVS, you will have to make this change yourself.

You can find UMFPACK at `http://www.cise.ufl.edu/research/sparse/`.

- *ARPACK, which requires both BLAS and LAPACK, allows the solving of sparse eigenvalue problems.*

  If you are planning on using ARPACK, please also read about the this bug *that might require you to patch ARPACK in order to use PyLinear with ARPACK. (If you don't use it, you may get invalid results or inexplicable crashes. Do yourself the favor.)*

  You can find ARPACK at `http://www.caam.rice.edu/software/ARPACK/`.

- DASKR, a well-known package for solving Differential-Algebraic Equations, or DAEs for short. DAEs are a generalization of Ordinary Differential Equations, or ODEs for short.

  This functionality is available as part of the toybox (see Chapter 5) for what the toybox is), but it's there if you need it.

  For ease of compilation, the DASKR source is packaged with PyLinear. To enable its use, simply go to the subdirectory 'fortran/daskr', type **./build.sh**, watch for its successful completion and uncomment the default options relating to DASKR in 'siteconf.py'.

  You can find DASKR at `http://netlib.org/ode`.

PyLinear allows you to query at runtime which of these packages are available, see Section 5.1.

2.2 Installing Boost Python

Installing the Boost libraries in a way that is suitable for PyLinear is, unfortunately, a non-straightforward process, at least if you are doing it for the first time. This section describes that process.

There is a bit of good news, though: If you are lucky enough to be using the Debian flavor of Linux or one of its derivatives, you may simply type `aptitude install libboost-python-dev` and ignore the rest of this section.

Otherwise, you must follow these steps:

- Download a Boost release.
- Download and install Boost.Jam, a build tool.
- Build Boost, such as by typing

```
bjam -sPYTHON_ROOT=/usr -sPYTHON_VERSION=2.4 \
  -sBUILD="release <runtime-link>dynamic <threading>multi"
```

  (You may have to adapt `PYTHON_ROOT` and `PYTHON_VERSION` depending on your system.)

- Check the directory

```
boost/bin/boost/libs/python/build/libboost_python.so...
  .../gcc/release/shared-linkable-true/threading-multi
```

  and find 'libboost_python*.so'. (Don't copy the dots–they are only there to make the line fit on this page.) Copy these files to somewhere on your dynamic linker path, for example:

  – '/usr/lib'
  – a directory on LD_LIBRARY_PATH
  – or something mentioned in '/etc/ld.so.conf'.

  You should also create a symbolic link called 'libboost_python.so' to the main '.so' file.

- Run **ldconfig**.

2.3 Installing the Boost UBlas Bindings

This part is, fortunately, very easy. Just go to `http://news.tiker.net/software/boost-bindings`, download the current snapshot and extract it somewhere, for example by typing

```
tar xvfz boost-bindings-NNNNNNNN.tar.gz
```

Then remember the path where you unpacked it for the next step.

If you get the Boost Numeric Bindings from CVS, please read the section on installing UMFPACK/UFsparse in Section 2.1.

2.4 Installing PyLinear

As a first step, copy the file 'siteconf-template.py' to 'siteconf.py' and open an editor on that file. You will see a bunch of variables that you may customize to adapt PyLinear to your system. First of all, there are a few variables that are named *HAVE_XXX*, such as *HAVE_BLAS*. They all default to `False`. If you have the corresponding library available, set that variable to `True`.

For each library that you have answered `True` above, you may need to state in which directories to find the header files (in *XXX_INCLUDE_DIRS*), the libraries (in *XXX_LIBRARY_DIRS*) and finally, if the libraries are named in some nonstandard fashion, you may also have to change the library names to link against (in *XXX_LIBRARIES*). The defaults work at least with Debian Linux.

These above instructions apply to all prerequisite libraries. Here are a few hints for specific libraries:

- For Boost, set *BOOST_INCLUDE_DIR* to the directory where the root of your boost tree. Typically, it ends in 'boost'. For *BOOST_LIBRARY_DIRS*, give the path where you put the `libboost_python*.so` files. Finally, you should usually leave BPL_LIBRARIES unchanged and make a symbolic link from 'libboost_python.so' to the actual (non-symlink) '.so' file.
- For the Boost bindings, just insert the path where you unpacked them–No further installation is required.
- Here's an extra trick for BLAS and LAPACK if you are using Debian: If you install lapack and blas, make sure to install the versions ending in "3" (i.e. `blas3-dev` and `atlas3-dev`), and also install "`atlas3-ARCH-dev`", where `ARCH` is your processor architecture. Debian will then automatically activate an accelerated BLAS for your computer.

Then, type

```
python setup.py build
```

and wait what happens. This will compile PyLinear, which, depending on your compiler, will take a little while. Once this step completes, type

```
su -c "python setup.py install"
```

As a final step, you may change into the ‘test/’ subdirectory and execute

```
python test_matrices.py
```

This will execute PyLinear’s unit test suite. All tests should run fine, outputting a long line of dots and ”OK” as the last line of output.

Congratulations! You have now successfully installed PyLinear.

CHAPTER

THREE

The `Array` types

This chapter decribes the basic array types provided by PyLinear, and the elementary operations available on them.

We're describing the module `pylinear.array`. For simplicity, it is assumed to be imported using

```
>>> from pylinear.array import *
```

in the examples.

3.1 Types and Flavors

First, let's fix a bit of terminology.

From this point onwards, we will use the term `Array` to mean any type of matrix or vector. The term `Matrix` will refer to matrices, both sparse and dense. The term `Vector` shall refer to only the dense vector type. Note that these do not exist as actual Python classes, but we will pretend that they do.

In PyLinear, two things determine an `Array` type: the *flavor* and the *data type* (or *dtype* for short).

The supported data types are

- double precision (i.e. 64-bit) real, and
- double precision (i.e. 2x64-bit) complex.

The supported flavors are

- dense vector,
- dense matrix,
- sparse build matrix, and
- sparse execute matrix.

There is only one vector flavor, but there are three different flavors of matrices with different performance and memory characteristics. *Dense* matrices store *m*-by-*n* elements in a two-dimensional grid of *m* rows and *n* columns. They are used for small matrices or those which have mostly non-zero elements. Contrast this with the sparse types, which are typically used for matrices consisting of mostly zero elements. *Sparse build* matrices store their elements a list of '`(i, j, a[i,j])`', to which new elements are simply appended, which is very fast. This list is typically unsorted, but may have to be sorted by *i* and *j* for multiplication, element access or element removal, which makes these operations

pretty slow. Consequently, this flavor is typically used for the assembly of large sparse matrices. It is then converted to the *sparse execute* flavor for fast matrix multiplication. This flavor uses a standard compressed column format for fast linear algebra operations.

Each of the flavors is represented by a symbolic constant:

Constant	Corresponding Flavor
`Vector`	The dense vector flavor.
`DenseMatrix`	The dense matrix flavor.
`SparseBuildMatrix`	The sparse build matrix flavor.
`SparseExecuteMatrix`	The sparse execute matrix flavor.

Likewise, each of the element types has its own symbolic constant.

Constant	Corresponding Element Type
`Float64`	The 64-bit real element type.
`Complex64`	The 2x64-bit complex element type.
`Float`	The machine-native C++ `double` element type. An alias for `Float64`.
`Complex`	The machine-native C++ `std::complex<double>` element type. An alias for `Complex64`.

Despite a good bit of internal dissimilarity, PyLinear's programming interface attempts to be mostly compatible with NumPy, which is the traditional (non-sparse) array package for Python. So suppose you have some NumPy code that you would like to run on PyLinear. That code likely has a line like `import numpy` somewhere near the top. Then you can try and say `import pylinear.array as numpy`, which should get you most of the way there.

3.2 Creating `Arrays`

The following functions in the module `pylinear.array` permit the creation of new `Array`s:

`array`(*sequence, dtype=None*)

There are many ways to create arrays. The most basic one is the use of the `array` function:

```
>>> a = array([1.2, 3.5, -1])
```

to make sure this worked, do:

```
>>> a
array([1.2, 3.5, -1.0])
```

The `array` function takes several arguments — the first one is a Python sequence object (such as a list or a tuple). The optional argument `type` specifies the element type of the matrix. If omitted, as in the example above, Python tries to find the best data type which can represent all the elements. `array` always creates dense matrices or vectors, depending on the *dimensionality* of the input data. (The dimension of the data is 1 for a list, 2 for a list of lists, and so on. 1-dimensional data will be converted to vectors, 2-dimensional data to matrices.)

Since the elements we gave our example were two floats and one integer, it chose `Float64` as the type of the resulting array. One can specify unequivocally the `type` of the elements—this is especially useful when, for example, one wants an array contains complex numbers even though all of its input elements are reals:

```
>>> array([1,2,3]) # reals are enough for 1, 2 and 3
array([1.0, 2.0, 3.0])
>>> array([1,2,3], dtype=Complex64) # not the default type
array([(1+0j), (2+0j), (3+0j)])
>>> array([1,2,3+0j]) # same effect
array([(1+0j), (2+0j), (3+0j)])
```

Note that in NumPy, `array` takes a few more arguments, such as `copy`, `savespace`, and `shape`. These are not supported.

`sparse`(*mapping, shape=None, dtype=None, flavor=SparseBuildMatrix*)

This function creates a (not necessarily sparse) `Matrix` of the given `shape`, `dtype`, and `flavor` based on a sparse representation of its entries. At present, it cannot create `Vector`s. The sparse representation consists of a dictionary of dictionaries, whose keys are the row indices for the outer dictionary, and the column indices for the inner one.

If the `shape` parameter is unspecified, the shape is specified by the largest row and column indices seen in examining the `mapping`. If the `dtype` is unspecified, `sparse` uses the same logic as `data` to determine it.

```
>>> sparse({0:{4:17, 3:1+2j},3:{2:15}})
sparse({0: {3: (1+2j), 4: (17+0j)},
        3: {2: (15+0j)}},
       shape=(4, 5), flavor=SparseBuildMatrix)
>>> sparse({0:{4:17, 3:1+2j},3:{2:15}}, flavor=SparseExecuteMatrix)
sparse({0: {3: (1+2j), 4: (17+0j)},
        3: {2: (15+0j)}},
       shape=(4, 5), flavor=SparseExecuteMatrix)
```

`asarray`(*seq, dtype, flavor=None*)

This function converts scalars, lists and tuples to an `Array` type, if possible. It passes `Array`s through, making copies only to convert types. In any other case a `TypeError` is raised.

`empty`(*shape, dtype=Float, flavor=None*)

`empty` creates an `Array` of the given *shape*, *dtype* and *flavor* which is not initialized at all. With a little bit of luck, it'll contain all zeros, but in general it may contain whichever garbage was in the chunk of memory that the `Array` now occupies. This is typically faster than `zeros`, especially for dense `Array`s.

If you do decide that you need the `Array` empty, call `clear()` on it.

See the `shape` attribute in Section 3.3 for information on the *shape* parameter.

```
>>> empty((2,5), Float) # ATTENTION: random garbage
array([[-3.92715777576e-39,  6.3659873729e-314, 6.36598737438e-314,
            6.36598737438e-314, 6.36598737438e-314],
       [5.54176250076e+257, 2.80259807429e+262, 1.99389578315e-313,
            9.89838462118e+169, 3.60739284523e-313]])
```

`zeros`(*shape, dtype=Float, flavor=None*)

`zeros` creates an `Array` of the given *shape*, *dtype* and *flavor* which is filled with zeros. See the `shape` attribute in Section 3.3 for information on the *shape* parameter.

```
>>> zeros((2,5), Float)
array([[0.0, 0.0, 0.0, 0.0, 0.0],
       [0.0, 0.0, 0.0, 0.0, 0.0]])
```

`ones`(*shape, dtype=Float, flavor=None*)

`ones` creates an `Array` of the given *shape*, *dtype* and *flavor* which is filled with ones. See the `shape` attribute in Section 3.3 for information on the *shape* parameter.

```
>>> ones((3,7), Float)
array([[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
       [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
       [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]])
```

eye(*n, m=None, offset=0, dtype=Float, flavor=None*)

`eye` creates a `Matrix` of shape `(n,m)` and the given *dtype* and *flavor* which is filled with zeros and has ones on the `offset`th super-diagonal. `m` is assumed identical to `n` if it is `None`.

`offset` may also be negative, thus pointing to a sub-diagonal.

If `offset` is 0, you get an $n \times m$ identity matrix.

```
>>> eye(4, 3, offset=1, dtype=Complex)
array([[0j, (1+0j),     0j],
       [0j,     0j, (1+0j)],
       [0j,     0j,     0j],
       [0j,     0j,     0j]])
```

tri(*n, m=None, offset=0, dtype=Float, flavor=None*)

`tri` creates a `Matrix` of shape `(n,m)` and the given *dtype* and *flavor* which has ones on the `offset`th super-diagonal and below, and zeros elsewhere.

`offset` may also be negative, thus pointing to a sub-diagonal.

```
>>> tri(4, 3, offset=1, dtype=Complex)
array([[(1+0j), (1+0j),     0j],
       [(1+0j), (1+0j), (1+0j)],
       [(1+0j), (1+0j), (1+0j)],
       [(1+0j), (1+0j), (1+0j)]])
```

hstack(*tup*)

Take a sequence of arrays and stack them horizontally to make a single array. All arrays in the sequence must have the same shape along all but the second axis. `hstack` will rebuild arrays divided by `hsplit`.

```
>>> a = identity(3)[:,:2]
>>> a
array([[1.0, 0.0],
       [0.0, 1.0],
       [0.0, 0.0]])
>>> hstack((a,2*a,3*a))
array([[1.0, 0.0, 2.0, 0.0, 3.0, 0.0],
       [0.0, 1.0, 0.0, 2.0, 0.0, 3.0],
       [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]])
```

vstack(*tup*)

Take a sequence of arrays and stack them vertically to make a single array. All arrays in the sequence must have the same shape along all but the first axis. `vstack` will rebuild arrays divided by `vsplit`.

```
>>> a = identity(3)[:2]
>>> a
array([[1.0, 0.0, 0.0],
       [0.0, 1.0, 0.0]])
>>> vstack((a,2*a,3*a))
array([[1.0, 0.0, 0.0],
       [0.0, 1.0, 0.0],
       [2.0, 0.0, 0.0],
       [0.0, 2.0, 0.0],
       [3.0, 0.0, 0.0],
       [0.0, 3.0, 0.0]])
```

3.2.1 Pickle support

`Array`s also have efficient support for pickling. Pickling is a convenient way to store complicated data structures in a platform-independent byte stream. Unless you need human-readable output, pickling makes an excellent way of saving PyLinear arrays to disk.

As such, unpickling a previously pickled `Array` is another way to create one:

```
>>> x = array([[1,2,3],[4,5,6]])
>>> x
array([[1.0, 2.0, 3.0],
       [4.0, 5.0, 6.0]])
>>> import pickle
>>> string_rep = pickle.dumps(x)
>>> isinstance(string_rep, str)
True
>>> y = pickle.loads(string_rep)
>>> y
array([[1.0, 2.0, 3.0],
       [4.0, 5.0, 6.0]])
```

Pickling also works on arbitrarily larger data structures of which `Array`s are only a part.

3.3 Accessing `Array` Properties

Once you've created a few `Array`s, you might want to query them about their properties, such as their data type or shape.

`shape`

Reading the `shape` attribute gets the shape tuple, that is, a tuple of length equal to the array's rank specifying the dimensions of the matrix. For a vector, this is a singleton containing an integer, for a matrix, this is a pair containing the number of rows and columns, in this order.

Assigning a value to the `shape` attribute will destructively resize the array.

```
>>> x = array([[1,2,3],[4,5,6]])
>>> x.shape
(2, 3)
>>> x.T.shape
(3, 2)
>>> x.shape = (4,2)
>>> x # typically random garbage
array([[             1.0,                2.0],
       [             4.0,                5.0],
       [6.3659873729e-314, 8.70018274296e-313],
       [2.2424384485e-269, 1.91651066261e-313]])
```

`dtype`

Reading the `dtype` attribute gets the data type of the given matrix. Assigning a value to the `dtype` attribute is not supported.

```
>>> x = array([[1,2,3],[4,5,6]])
>>> x.dtype
pylinear.array.Float64
>>> x = array([[1+3j,2-4j,3],[4,5+1j,6]])
>>> x.dtype
pylinear.array.Complex64
```

`flavor`
: Reading the `flavor` attribute gets the flavor of the given matrix. Assigning a value to the `flavor` attribute is not supported.

3.4 Accessing `Array` Data

PyLinear provides a multitude of ways to access and manipulate the data contained in an array, the simplest of which may be just accessing the elements one-by-one or in chunks, as described in the next section.

3.4.1 Indexing

PyLinear supports indexing for reading and writing on `Array`s, in nearly the same way as you might be used to it from either Python sequences or Matlab matrices. This mode of access is quite power- and featureful, so let's go over the possibilities one by one.

`Matrix` objects are indexed by 2-tuples, whereas `Vector`s are indexed by single values. Like all indices in Python, PyLinear's indices are 0-based.

```
>>> a = array([[1,2,3],[4,5,6],[7,8,9]])
>>> a
array([[1.0, 2.0, 3.0],
       [4.0, 5.0, 6.0],
       [7.0, 8.0, 9.0]])
>>> a[0,2]
3.0
>>> a[0,1:]
array([2.0, 3.0])
>>> v = array([1,2,3])
>>> v[1]
2.0
>>> v[1:]
array([2.0, 3.0])
```

Negative indices count from the end of the respective dimension:

```
>>> a = array([[1,2,3],[4,5,6],[7,8,9]])
>>> a
array([[1.0, 2.0, 3.0],
       [4.0, 5.0, 6.0],
       [7.0, 8.0, 9.0]])
>>> a[-1,-2]
8.0
>>> a[-1,1:]
array([8.0, 9.0])
>>> v = array([1,2,3])
>>> v[-1]
3.0
>>> v[-1:]
array([3.0])
```

Writing data to specific places in `Array`s is just as simple:

```
>>> a = array([[1,2,3],[4,5,6],[7,8,9]])
>>> a[0,2] = 17
>>> a
array([[1.0, 2.0, 17.0],
       [4.0, 5.0,  6.0],
       [7.0, 8.0,  9.0]])
>>> a[0:2,1:] = array([[17,18],[19,20]])
>>> a
array([[1.0, 17.0, 18.0],
       [4.0, 19.0, 20.0],
       [7.0,  8.0,  9.0]])
```

Row-wise Access

Indexing a matrix with a single value returns entire rows as `Vector`s:

```
>>> a[0]
array([1.0, 17.0, 18.0])
>>> a[0].flavor
Vector
```

Note the subtlety: If you specify just a column or row, you get a `Vector`. If you specify a one-element slice, you get a `Matrix`:

```
>>> a[0:1]
array([[1.0, 17.0, 18.0]])
>>> a[0:1].flavor
Matrix
```

The same logic, of course, goes for column-wise access:

```
>>> a[:,2]
array([18.0, 20.0, 9.0])
>>> a[:,2].flavor
Vector
>>> a[:,2:3]
array([[18.0],
       [20.0],
       [ 9.0]])
>>> a[:,2:3].flavor
Matrix
```

Write access is less picky in that respect:

```
>>> a[0] = array([1.41,3.14,2.71]) # write as vector
>>> a
array([[1.41, 3.14, 2.71],
       [ 4.0, 19.0, 20.0],
       [ 7.0,  8.0,  9.0]])
>>> a[0] = array([[3.14,2.71,1.56]]) # write as matrix
>>> a
array([[3.14, 2.71, 1.56],
       [ 4.0, 19.0, 20.0],
       [ 7.0,  8.0,  9.0]])
```

Strides

Strides are supported, i.e. `a[3:9:2]` gives you the entries at indices 3, 5, and 7. Strides may also be negative:

```
>>> a
array([[3.14, 2.71, 1.56],
       [ 4.0, 19.0, 20.0],
       [ 7.0,  8.0,  9.0]])
>>> a[::-1]
array([[ 7.0,  8.0,  9.0],
       [ 4.0, 19.0, 20.0],
       [3.14, 2.71, 1.56]])
```

Unlike Python lists, `Array`s may not be resized using slice assignments. Like in the rest of Python, yet unlike NumPy, slices return copies, not views of the corresponding data.

Broadcasting

When assigning to `Array`s slices and/or subscripts, the right hand side of the assignment may have lesser rank than the left hand side. In this case, the right hand side is *broadcast* across the missing rank. Observe:

```
>>> a = ones((5,5))
>>> b = arange(5)
>>> a[:,:] = b
>>> a
array([[0.0, 1.0, 2.0, 3.0, 4.0],
       [0.0, 1.0, 2.0, 3.0, 4.0],
       [0.0, 1.0, 2.0, 3.0, 4.0],
       [0.0, 1.0, 2.0, 3.0, 4.0],
       [0.0, 1.0, 2.0, 3.0, 4.0]])
>>> a[:,:] = 10
>>> a
array([[10.0, 10.0, 10.0, 10.0, 10.0],
       [10.0, 10.0, 10.0, 10.0, 10.0],
       [10.0, 10.0, 10.0, 10.0, 10.0],
       [10.0, 10.0, 10.0, 10.0, 10.0],
       [10.0, 10.0, 10.0, 10.0, 10.0]])
```

The same trick works for vectors as well.

3.4.2 Picking `Arrays` apart

The following operations return only parts of the data contained in a `Array` object:

Picking apart complex `Arrays`

`real`
: Reading this attribute obtains a copy of the real part of the matrix. For real matrices, the matrix is simply copied.

  In NumPy, this method does not return a copy, but a view.

```
>>> x = array([[1+3j,2-4j,3],[4,5+1j,6]])
>>> x
array([[(1+3j), (2-4j), (3+0j)],
       [(4+0j), (5+1j), (6+0j)]])
>>> x.real
array([[1.0, 2.0, 3.0],
       [4.0, 5.0, 6.0]])
```

`imaginary`
: Reading this attribute obtains a copy of the imaginary part of the matrix. For real matrices, a zero matrix of the same size is returned.

  In NumPy, this method does not return a copy, but a view.

```
>>> x = array([[1+3j,2-4j,3],[4,5+1j,6]])
>>> x
array([[(1+3j), (2-4j), (3+0j)],
       [(4+0j), (5+1j), (6+0j)]])
>>> x.imaginary
array([[3.0, -4.0, 0.0],
       [0.0,  1.0, 0.0]])
```

take(*matrix, indices, axis=0*)
Assembles an Array from the entries of the Array listed in *indices*, which must be simple numbers. *axis* specifies the axis along which the indices are taken.

Next, we will discuss a few functions that return just parts of Matrix objects.

hsplit(*ary, indices_or_sections*)
Split a single Matrix array into multiple sub-Matrix instances. The array is divided into groups of columns. If indices_or_sections is an integer, ary is divided into that many equally sized sub-arrays. If it is impossible to make the sub-arrays equally sized, the operation throws a ValueError exception.

```
>>> a = array([[1,2,3],[4,5,6]])
>>> a
array([[1.0, 2.0, 3.0],
       [4.0, 5.0, 6.0]])
>>> hsplit(a.T, 2)
[array([[1.0],
       [2.0],
       [3.0]]), array([[4.0],
       [5.0],
       [6.0]])]
>>> hsplit(a, [1])
[array([[1.0],
       [4.0]]), array([[2.0, 3.0],
       [5.0, 6.0]])]
>>> hstack(hsplit(a, [1])) == a
True
```

vsplit(*ary, indices_or_sections*)
Split a single Matrix array into multiple sub-Matrix instances. The array is divided into groups of rows. If indices_or_sections is an integer, ary is divided into that many equally sized sub-arrays. If it is impossible to make the sub-arrays equally sized, the operation throws a ValueError exception.

```
>>> a = array([[1,2,3],[4,5,6]])
>>> a
array([[1.0, 2.0, 3.0],
       [4.0, 5.0, 6.0]])
>>> vsplit(a, 2)
[array([[1.0, 2.0, 3.0]]), array([[4.0, 5.0, 6.0]])]
>>> vsplit(a, [1])
[array([[1.0, 2.0, 3.0]]), array([[4.0, 5.0, 6.0]])]
>>> vstack(vsplit(a, [1])) == a
True
```

diagonal(*matrix, offset=0*)
Returns the diagonal of *matrix* as a vector, or the *offset*th super- (for offset>0) or sub-diagonal (for offset<0).

```
 >>> a =
array([[1,2,3],[4,5,6]]) >>> a >>> diagonal(a) >>> diagonal(a,1)
```

3.5 Basic Array Math

In this section, we will discuss how to perform basic calculations with Array objects.

To begin with, `Array`s support the usual range of algebraic operators, such as `+`, `-`, `*` and `/`. For the additive operators, this is straightforward, as the standard elementwise meaning applies. The multiplicative operators, on the other hand, acquire different meanings depending on the types of their arguments, such as matrix or dot products. The details can be found in 3.5.2. Despite the default "complicated" meaning of multiplication, elementwise multiplication is also available, as is a host of other elementwise operations. See 3.6 for details.

PyLinear also supplies a large number of more advanced matrix procedures. These are described in Chapter 5.

3.5.1 Generalities on Binary Operations

`Array`s provide all the operators you would expect, like sums, differences, products and such. There are, however, a couple of fine points that are worth noting:

Type promotion

If binary operators or elementwise functions (see Section 3.6) are applied to arrays of non-matching flavor or dtype, the operands are promoted to a common type. (For the case of non-matching dimension, see Section **??** for broadcasting rules.)

If the only mismatch is in dtype, one argument array is cast upward in the type hierarchy (e.g. from integer to real, from real to complex) in order to match the other.

```
>>> a = array([[1,2],[0,1]])
>>> b = ones((2,2)) * 1j
>>> a.dtype
pylinear.array.Float64
>>> b.dtype
pylinear.array.Complex64
>>> (a+b).dtype
pylinear.array.Complex64
```

If there is also a mismatch in flavor, the result assumes the flavor of the first operand:

```
>>> a = ones((3,3))
>>> b = ones((3,3), flavor=SparseBuildMatrix)
>>> a+b
array([[2.0, 2.0, 2.0],
       [2.0, 2.0, 2.0],
       [2.0, 2.0, 2.0]])
>>> b+a
sparse({0: {0: 2.0, 1: 2.0, 2: 2.0},
        1: {0: 2.0, 1: 2.0, 2: 2.0},
        2: {0: 2.0, 1: 2.0, 2: 2.0}},
       shape=(3, 3), flavor=SparseBuildMatrix)
```

Broadcasting

The binary elementwise operators as well as all the binary elementwise functions (see Section 3.6) accept argument pairs where one argument has lesser rank than the other. In this case, the missing ranks are *broadcast* across the remainder of the matrix. If the lesser-rank argument is a scalar, this is easy to explain: It is treated like an array of the

right size filled with that scalar. If it is a vector, that vector is treated like a matrix filled with rows consisting of the given vector.

```
>>> a = ones((4,4))
>>> b = arange(4)
>>> a
array([[1.0, 1.0, 1.0, 1.0],
       [1.0, 1.0, 1.0, 1.0],
       [1.0, 1.0, 1.0, 1.0],
       [1.0, 1.0, 1.0, 1.0]])
>>> b
array([0.0, 1.0, 2.0, 3.0])
>>> b+5
array([5.0, 6.0, 7.0, 8.0])
>>> a+b
array([[1.0, 2.0, 3.0, 4.0],
       [1.0, 2.0, 3.0, 4.0],
       [1.0, 2.0, 3.0, 4.0],
       [1.0, 2.0, 3.0, 4.0]])
>>> a+b+5
array([[6.0, 7.0, 8.0, 9.0],
       [6.0, 7.0, 8.0, 9.0],
       [6.0, 7.0, 8.0, 9.0],
       [6.0, 7.0, 8.0, 9.0]])
```

3.5.2 Multiplicative (and related) Operators

This section explains the value of the expression`a*b`, where at least one of `a` and `b` is an `Array`. Since multiplication is probably the most significant element of linear algebra, there's quite a bit to know here. Also, we will touch upon other related notions such as outer, inner and Kronecker products as well as inversion and exponentiation.

If the one operand in the expression `a*b` is a scalar (it doesn't matter which), the result will be the elementwise product of the array with that scalar.

```
>>> a = array([[1,2,3],[4,5,6]])
>>> a*2
array([[2.0,  4.0,  6.0],
       [8.0, 10.0, 12.0]])
>>> 2j*a
array([[2j,  4j,  6j],
       [8j, 10j, 12j]])
```

If both operands are `Vector`s, `a*b` computes the inner product of both vectors. Note that in the complex case no complex conjugates are taken. If you require them, use the expression `a*b.H`. *WARNING:* This notation is convenient, but slightly dangerous, mathematically. The inner product, if simply written as a "dot product", is *not* associative, meaning that for vectors a, b and c, typically $(a \cdot b) \cdot c \neq a \cdot (b \cdot c)$. PyLinear has no way of rejecting unparenthesized expressions such as `a*b*c`, but their meaning is uncertain since the order of evaluation is not explicitly specified.

```
>>> a = array([1,2,3])
>>> b = array([4,5,6])
>>> c = array([7,8,9])
>>> (a*b)*c
array([224.0, 256.0, 288.0])
>>> a*(b*c)
array([122.0, 244.0, 366.0])
>>> a*b*c # RANDOM RESULT!
array([224.0, 256.0, 288.0])
```

If `a` is a `Vector` and `b` is a `Matrix`, `a*b` will result in $b^T a$, using the conventional matrix-vector product.

If `a` is a `Matrix` and `b` is a `Vector`, `a*b` will result in ab, using the conventional matrix-vector product.

```
>>> a = array([[1,2,3],[4,5,6],[7,8,9]])
>>> b = array([1,3,5])
>>> a*b
array([22.0, 49.0, 76.0])
>>> b*a
array([48.0, 57.0, 66.0])
>>> a.T*b # less efficient!
array([48.0, 57.0, 66.0])
```

If both `a` and `b` are `Matrix` types, `a*b` will result in ab, using the conventional matrix-matrix product.

```
>>> a = array([[1,2,3],[4,5,6]])
>>> b = array([[1,2,1,2],[3,4,3,4],[5,6,5,6]])
>>> a*b
array([[22.0, 28.0, 22.0, 28.0],
       [49.0, 64.0, 49.0, 64.0]])
```

All these explanations also apply to the inplace multiplication operator `*=`.

All multiplication operators obey type promotion rules as laid out in Section **??**.

Related operators

The following operators are not invoked as `a*b`, but are still related to multiplication:

- `matrix**n`

  Computes the *n*th power of *matrix*. *n* must be integer, but may be negative. Only for dense matrices.

```
>>> a = array([[1,2,3],[3,2,1],[1,3,2]])
>>> a
array([[1.0, 2.0, 3.0],
       [3.0, 2.0, 1.0],
       [1.0, 3.0, 2.0]])
>>> a**2
array([[10.0, 15.0, 11.0],
       [10.0, 13.0, 13.0],
       [12.0, 14.0, 10.0]])
>>> a**3
array([[66.0, 83.0, 67.0],
       [62.0, 85.0, 69.0],
       [64.0, 82.0, 70.0]])
>>> a**3 * (1/a)
array([[10.0, 15.0, 11.0],
       [10.0, 13.0, 13.0],
       [12.0, 14.0, 10.0]])
```

- `scalar/matrix`

  Computes the *scalar* multiple of the inverse of *matrix*. Only for dense matrices.

  Do not use code like `1/a*b` to solve the linear system $Ax = b$; besides being slow, this tends to yield imprecise results. Instead, use the `<<solve>>` pseudo-operator.

  Use of this operator will fail unless the module `pylinear.operation` is available.

```
>>> a = array([[1,2,3],[3,2,1],[1,3,2]])
>>> a
array([[1.0, 2.0, 3.0],
       [3.0, 2.0, 1.0],
       [1.0, 3.0, 2.0]])
>>> 1/a
array([[0.0833333333333,   0.416666666667, -0.333333333333],
       [-0.416666666667, -0.0833333333333,  0.666666666667],
       [ 0.583333333333, -0.0833333333333, -0.333333333333]])
>>> 1/a * a
array([[              1.0, 8.32667268469e-17, 4.16333634234e-17],
       [-4.16333634234e-17,               1.0, -1.2490009027e-16],
       [ 5.55111512313e-17,               0.0,               1.0]])
```

  Observe that the results are likely useless if the matrix is singular:

```
>>> a = array([[1,2,3],[4,5,6],[7,8,9]])
>>> a
array([[1.0, 2.0, 3.0],
       [4.0, 5.0, 6.0],
       [7.0, 8.0, 9.0]])
>>> 1/a
array([[ 3.15221190597e+15, -6.30442381193e+15,  3.15221190597e+15],
       [-6.30442381193e+15,  1.26088476239e+16, -6.30442381193e+15],
       [ 3.15221190597e+15, -6.30442381193e+15,  3.15221190597e+15]])
>>> 1/a * a
array([[ 3.0,  3.0,  3.0],
       [-1.0, -2.0, -3.0],
       [-2.5, -2.0, -1.5]])
>>> from pylinear.computation import determinant
>>> determinant(a)
-9.51712667008e-16
```

- `matrix <<solve>> vector`

  Returns the solution of the linear system of equations `matrix*x=vector`. Available for dense and sparse execute matrix types of *matrix*.

  Use of this operator will fail unless the module `pylinear.computation` is available.

  Since this is not built using actual Python syntax, but rather cheaply composed of a special-purpose `solve` object with left and right shift operators, some care needs to be exercised regarding operator precedence. When in doubt, just use parentheses.

```
>>> a = array([[1,2,3],[3,2,1],[1,3,2]])
>>> b = array([9,1,1])
>>> v = a <<solve>> b
>>> v
array([0.833333333333, -3.16666666667, 4.83333333333])
>>> a * v
array([9.0, 1.0, 1.0])
```

  Note that you need to qualify `<<solve>>` with the module name if you do not import `pylinear.array` using '`from ... import *`':

```
>>> import pylinear.array as num
>>> a = num.array([[1,2,3],[3,2,1],[1,3,2]])
>>> b = num.array([9,1,1])
>>> v = a <<num.solve>> b
>>> v
array([0.833333333333, -3.16666666667, 4.83333333333])
>>> a * v
array([9.0, 1.0, 1.0])
```

  Observe that the results are likely useless if the matrix is singular:

```
>>> a = array([[1,2,3],[4,5,6],[7,8,9]])
>>> b = array([9,1,1])
>>> v = a <<solve>> b
>>> v
array([2.52176952477e+16, -5.04353904954e+16, 2.52176952477e+16])
>>> a * v
array([0.0, 0.0, -32.0])
>>> from pylinear.computation import determinant
>>> determinant(a)
-9.51712667008e-16
```

- `vector1 <<outer>> vector2`

  Computes the outer product of `vector1` and `vector2`, whose result is the matrix $v_1 \cdot v_2^T$.

```
>>> v = array([1,2,3])
>>> w = array([3,2,1])
>>> v <<outer>> w
array([[3.0, 2.0, 1.0],
       [6.0, 4.0, 2.0],
       [9.0, 6.0, 3.0]])
>>> w <<outer>> v
array([[3.0, 6.0, 9.0],
       [2.0, 4.0, 6.0],
       [1.0, 2.0, 3.0]])
```

  Please see the section on `<<solve>>` above for important considerations on operator precedence and module qualification that also apply here.

- `vector1 <<cross>> vector2`

  Computes the cross product of `vector1` and `vector2`. Both `vector1` and `vector2` must be of dimension 2 or 3. For dimension 2, the z component of the corresponding 3-dimensional cross product is returned as a scalar.

```
>>> v = array([1,2,3])
>>> w = array([3,2,1])
>>> v <<cross>> w
array([-4.0, 8.0, -4.0])
>>> w <<cross>> v
array([4.0, -8.0, 4.0])
>>> (v <<cross>> w) * v
0.0
```

  Please see the section on `<<solve>>` above for important considerations on operator precedence and module qualification that also apply here.

- `matrix1 <<kron>> matrix2`

  Computes the Kronecker product (sometimes called the tensor product) of `matrix1` and `matrix2`:

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}$$

```
>>> v = array([[1,4],[4,8]])
>>> w = array([[1,2],[2,1]])
>>> v <<kron>> w
array([[1.0, 2.0,  4.0,  8.0],
       [2.0, 1.0,  8.0,  4.0],
       [4.0, 8.0,  8.0, 16.0],
       [8.0, 4.0, 16.0,  8.0]])
```

Please see the section on `<<solve>>` above for important considerations on operator precedence and module qualification that also apply here.

XXX Is outerproduct documented?

3.6 Elementwise Functions

PyLinear sports a few so-called *Elementwise Functions*, some of which are *unary*, while others are *binary*. (In NumPy, this kind of function is called a ufunc, or "Universal Function".) Elementwise functions generally apply a certain functionality to each element in an array. For example, the `sin` elementwise function computes the sine of each of the given array's entries, and returns the processed matrix, which will be of the same size, flavor, and dtype. Binary elementwise functions receive two `Array`s of equal size as arguments, apply a binary function (such as, for example, addition or multiplication) to each pair of entries of the two `Array`s, pairing the entries at the same location in each `Array`, and return an `Array` with the results. Binary elementwise functions obey type promotion laws as laid out in section **??**.

The following unary elementwise functions exist:

`conjugate`(*array*)
: Returns the complex-conjugate of the given `Array`. Simply copies real matrices.

`cos`(*array*)
: Returns the elementwise cosine of the given `Array`.

`cosh`(*array*)
: Returns the elementwise hyperbolic cosine of the given `Array`.

`exp`(*array*)
: Returns the elementwise natural exponential of the given `Array`.

  WARNING: This is not matrix exponentiation.

`log`(*array*)
: Returns the elementwise natural logarithm of the given `Array`.

`log10`(*array*)
: Returns the elementwise base-10 logarithm of the given `Array`.

`sin`(*array*)
: Returns the elementwise sine of the given `Array`.

`sinh`(*array*)
: Returns the elementwise hyperbolic sine of the given `Array`.

`sqrt`(*array*)
: Returns the elementwise square root of the given `Array`.

`tan`(*array*)
: Returns the elementwise tangent of the given `Array`.

tanh(*array*)
Returns the elementwise hyperbolic tangent of the given `Array`.

floor(*array*)
Returns the elementwise floor of the given `Array`.

ceil(*array*)
Returns the elementwise ceiling of the given `Array`.

argument(*array*)
Returns the elementwise complex argument of the given `Array`. Resulting matrix consists of values of zero and π for real matrices.

absolute(*array*)
Returns the elementwise absolute value of the given `Array`.

The following binary elementwise functions exist:

add(*op1*, *op2*)
Returns the elementwise sum of the given `Array`s. Obeys broadcasting (see Section **??**) and type promotion (see Section **??**) laws.

Equivalent to the `+` operator.

subtract(*op1*, *op2*)
Returns the elementwise difference of the given `Array`s. Obeys broadcasting (see Section **??**) and type promotion (see Section **??**) laws.

Equivalent to the `-` operator.

multiply(*op1*, *op2*)
Returns the elementwise product of the given `Array`s. Obeys broadcasting (see Section **??**) and type promotion (see Section **??**) laws.

NOT equivalent to the `*` operator, except in the scalar case.

divide(*op1*, *op2*)
Returns the elementwise quotient of the given `Array`s. Obeys broadcasting (see Section **??**) and type promotion (see Section **??**) laws.

NOT equivalent to the `/` operator, except in the scalar case.

power(*op1*, *op2*)
Returns the elementwise power `op1[i]**op2[i]` of the given `Array`s. Obeys broadcasting (see Section **??**) and type promotion (see Section **??**) laws.

NOT equivalent to the `**` operator, except in the scalar case.

maximum(*op1*, *op2*)
Returns the elementwise maximum of the given `Array`s. Obeys broadcasting (see Section **??**) and type promotion (see Section **??**) laws.

For complex matrices, the maximum is found based on the real part.

minimum(*op1*, *op2*)
Returns the elementwise minimum of the given `Array`s. Obeys broadcasting (see Section **??**) and type promotion (see Section **??**) laws.

For complex matrices, the minimum is found based on the real part.

Additional elementwise function methods, such as `reduce`, as they are found in NumPy, are not (yet) supported in PyLinear.

3.6.1 Linear Algebra

The following operations are related to the theory of linear algebra.

`T`

This object property a real-transpose copy of the matrix.

Does not exist in NumPy.

```
>>> x = array([[1,2,3],[4,5,6]])
>>> x
array([[1.0, 2.0, 3.0],
       [4.0, 5.0, 6.0]])
>>> x.T
array([[1.0, 4.0],
       [2.0, 5.0],
       [3.0, 6.0]])
```

`H`

This object property returns a conjugate-transpose copy of the matrix. Identical to `T` for real matrices.

Does not exist in NumPy.

```
>>> x = array([[1+3j,2-4j,3],[4,5+1j,6]])
>>> x
array([[(1+3j), (2-4j), (3+0j)],
       [(4+0j), (5+1j), (6+0j)]])
>>> x.H
array([[(1-3j), (4-0j)],
       [(2+4j), (5-1j)],
       [(3-0j), (6-0j)]])
```

`trace`(*matrix, offset=0*)

Returns the sum of the *offset*th diagonal. See `diagonal` for details of the meaning of *offset*.

```
>>> a = array([[1,2,3],[4,5,6]])
>>> a
array([[1.0, 2.0, 3.0],
       [4.0, 5.0, 6.0]])
>>> trace(a)
6.0
>>> trace(a,1)
8.0
```

3.7 Flavor-specific Functionality

The following methods tie into the particulars of the sparse matrices' memory layouts.

`sort`()

The list of $(i, j, a_{i,j})$ stored by a `SparseBuildMatrix` can become unsorted, depending on the order of insertions into the matrix. This is rectified by the `sort` method. During normal usage, you don't have to worry about sorting your matrices, since this action is triggered automatically whenever it is necessary.

`set_element`(*i, j, entry*)

Sets the entry in the *i*th row and *j*th column to the value *entry*.

Why would you want this if you could easily say `A[i,j] = entry`? This method is guaranteed to be an O(1) operation if it is available, whereas the alternative notation will always work, but may be exceedingly

slow. (Consider the case of a `SparseExecuteMatrix`, which might have to perform an $O(n^2)$ move to accomodate a new element.)

Available on all but `SparseExecuteMatrix` objects.

`set_element_past_end`(*i, j, entry*)
Sets the entry in the *i*th row and *j*th column to the value *entry*. If used, the user guarantees that for all $k > i$, $A_{k,l} = 0$ for all l and that $A_{i,l} = 0$ for $l >= j$.

Why would you want this if you could easily say `A[i,j] = entry`? This method is guaranteed to be an O(1) operation if it is available, whereas the alternative notation will always work, but may be exceedingly slow. (Consider the case of a `SparseBuildMatrix`, which might have to be resorted.)

`add_element`(*i, j, number*)
Adds *number* to the entry in the *i*th row and *j*th column.

Why would you want this if you could easily say '`A[i,j] += entry`'? This method is guaranteed to be an O(1) operation if it is available, whereas the alternative notation will always work, but may be exceedingly slow. (Consider the case of a `SparseExecuteMatrix`, which might have to perform an $O(n^2)$ move to accomodate a new element.)

Available on all but `SparseExecuteMatrix` objects.

`complete_index1_data`()
This is a rather internal method, but it is explained here nonetheless.

The `SparseExecuteMatrix` class uses a list to indicate the column starts in a linear field of numbers. This list of column starts, in its original state, is usually incomplete, i.e. does not cover all rows, which allows $O(1)$ insertion at the end of the number field. This method makes sure that the column start list is complete. This is required by certain third-party sparse matrix libraries that directly read the structure of your sparse matrices. Within PyLinear, UMFPACK is one such example. Its wrappers call this method automatically, however, so that you don't have to worry about this here. But if you are binding to other sparse matrix libraries, this call might come in useful.

3.8 Matrix algebra

3.9 Simple computational routines

The following methods are available on PyLinear's `Array` types:

`sum`()
The `sum` method returns the sum of all non-zero array elements. (Saying "non-zero" sounds stupid, but it actually means that for sparse arrays, only non-zero elements are considered, and thus represents a guarantee with respect to asymptotic complexity of the operation.) Returns the sum of all elements in the array.

```
>>> x = array([[1,2,3],[4,5,6]])
>>> x.sum()
21.0
>>> v = array([1,2,3])
>>> v.sum()
6.0
```

`__iter__`()
For `Matrix` types, this method returns an iterator whose consecutive values are the rows of the matrix. Note that this method always returns a *dense* `Vector`, so it can be slow to use for sparse matrices.

For `Vector`s, this method returns an iterator whose consecutive values are the entries of the vector.

This method is implicitly called in `for` loops:

```
>>> x = array([[1,2,3],[4,5,6]])
>>> for row in x:
...   print row
...
array([1.0, 2.0, 3.0])
array([4.0, 5.0, 6.0])
>>> x = array([1,2,3])
>>> for entry in x:
...   print entry
...
1.0
2.0
3.0
```

indices()

This method works like the `keys`() method on a dictionary: It returns an iterator whose values are all indices of the `Array` for which the corresponding value is potentially non-zero. (That is, for dense matrices, it returns each element's index, while for sparse matrices, only non-zero elements are enumerated.)

```
>>> x = sparse({0:{4:17, 3:1+2j},3:{2:15}})
>>> for index in x.indices():
...   print index
(0, 3)
(0, 4)
(3, 2)
```

add_scattered(*row_indices*, *column_indices*, *little_matrix*)

Modifies the called matrix in-place by adding a the entries of a *little_matrix* to the already present entries, where the affected rows and columns are given by *row_indices*, *column_indices*.

```
>>> a = zeros((10,10), Float, SparseBuildMatrix)
>>> b = array([[1,2],[3,4]])
>>> a.add_scattered([4,8], [1,3], b)
>>> a
sparse({4: {1: 1.0, 3: 2.0},
        8: {1: 3.0, 3: 4.0}},
       shape=(10, 10), flavor=SparseBuildMatrix)
```

This operation is common in finite element codes.

copy()

Returns an identical copy of the matrix.

```
>>> a = array([1,2,3])
>>> b = a.copy()
>>> b[0] = 15
>>> a
array([1.0, 2.0, 3.0])
>>> b
array([15.0, 2.0, 3.0])
```

solve_upper(*vector*)

If the matrix is an non-singular upper triangular matrix, then a vector `result` is returned that satisfies `matrix*result=vector`, i.e. this routine solves the linear system given by the matrix.

If the matrix is not regular upper triangular, then the result of this routine is still a vector, but of undefined meaning.

```
>>> a = array([[1,2],[0,3]])
>>> b = array([17,12])
>>> v = a.solve_upper(b)
>>> v
array([9.0, 4.0])
>>> a*v
array([17.0, 12.0])
```

solve_lower(*vector*)

If the matrix is an non-singular lower triangular matrix, then a vector `result` is returned that satisfies `matrix*result=vector`, i.e. this routine solves the linear system given by the matrix.

If the matrix is not regular upper triangular, then the result of this routine is still a vector, but of undefined meaning.

```
>>> a = array([[1,0],[3,4]])
>>> b = array([17,12])
>>> v = a.solve_lower(b)
>>> v
array([17.0, -9.75])
>>> a*v
array([17.0, 12.0])
```

CHAPTER

FOUR

Matrix-Free Methods

This chapter introduces the notion of an `Operator`, which is PyLinear's way of expressing matrix-free methods.

4.1 The `Operator` concept

Everything that has to do with `Operator` instances is contained in the module `pylinear.operator`. Let's import it:

```
>>> from pylinear.array import *
>>> import pylinear.operator as op
```

class `Operator`

An `Operator` is a (typically linear) mapping of one vector to another. A matrix is a particularly prominent example of this, but `Operator`s are mainly used to represent vector-to-vector mappings for which no matrix is explicitly stored (or too expensive to compute explicitly). [1]

Given its single purpose, an `Operator` has a pretty simple interface:

`shape`

Returns a tuple `(h,w)`, which, in analogy to a `Matrix`, specifies the sizes of the vectors received and returned by the `Operator`.

`typecode``()`

Returns the typecode (see 3.1) of the `Vector`s that this `Operator` operates on. This is also the typecode of the `Vector`s returned by the operations of this `Operator`. For technical reasons, the two always match.

`apply`(*before, after*)

This method operates on the `Vector` `before` and returns the result of the operation in `after`. `after` needs to be a properly-sized `Vector`. Its initial values typically do not matter, but may be used as starting guesses, for example by iterative solvers. Initializing after to all zeroes is always acceptable.

4.2 `Operators` Form an Algebra

On top of the simple interface of an `Operator`, PyLinear provides a layer of convenience functions that facilitate the creation of derived instances.

For an `Operator` `A`, saying `A(x)` with a properly sized and typed `Vector` `x` will return the result of applying `A` to `x`, by calling the `apply` method described above.

[1] Note, however, that for technical reasons `Matrix` classes are not automatically `Operator` instances–they need to be explicitly made into these, as we will see soon.

For two `Operator`s `A` and `B`, you may write `A+B` to obtain an `Operator` mapping that will perform the operation `A(x)+B(x)`. The operator `-` works in an analogous fashion.

For two `Operator`s `A` and `B`, you may write `A*B` to obtain an `Operator` mapping that will perform the composed operation `A(B(x))`. You may also say `a*B` or B*a with an `Operator` `B` and a scalar `a`, and will obtain an `Operator` that performs `a*B(x)`. A unary minus `-A` returns the negated operator.

4.3 Types of `Operator`s

Matrix-generated operators are the most obvious kind of `Operator`, but there are more–and they do not necessarily correspond to a stored matrix representation.

Each type of operator comes with a constructor class. For example, matrix operators are constructed by calling the method `make` on the object called *MatrixOperator* in the `operator` module. Consider this example:

```
>>> a = array([[1,2],[3,4]])
>>> a_op = op.MatrixOperator.make(a)
>>> v = array([5,6])
>>> a*v
array([17.0, 39.0])
>>> a_op(v)
array([17.0, 39.0])
>>> a_plus_a_op = a_op+a_op
>>> a_plus_a_op(v)
array([34.0, 78.0])
>>> four_a_op = 2*a_op + a_plus_a_op
>>> four_a_op(v)
array([68.0, 156.0])
```

class `MatrixOperator`
A `MatrixOperator` makes a matrix into an `Operator`.

`make`(*matrix*)
This static method takes a matrix argument and returns a matrix operator of the corresponding type.

It does not make a copy of the matrix, instead, it just keeps a reference to the given matrix around.

Now that you have seen one constructor class, you have basically seen them all, as they are basically structured in the same way. What is going to vary from here on down is

- the name of the constructor class and
- the arguments of the `make` call.

Note however that for technical reasons the instance returned by `make` is not an instance of class `MatrixOperator`; in fact, `MatrixOperator` is not even technically a class.

So, let's dive right in. The next best thing past applying a linear operator directly is applying its inverse. Here are some operators to achieve that. Of course you could always compute the inverse of the matrix that generates the operator. There are better ways, however. The simplest one is the `LUInverseOperator`:

class `LUInverseOperator`
A `LUInverseOperator` operates on vectors as the inverse of the dense matrix it is constructed for, by computing a LU decomposition.

`make`(*matrix*)
Returns an `Operator` whose effect on a vector is A^{-1}, if A is the given *matrix*.

This is a static method.

However, for sparse matrices, computing the plain LU decomposition is rarely feasible. More finesse is required to maintain the sparseness, and thus the tractability, of the operation. That kind of finesse is provided by UMFPACK, which is also wrapped in an `Operator` interface in PyLinear.

class `UMFPACKOperator`

A `UMFPACKOperator` operates on vectors as the inverse of a `SparseExecuteMatrix` given to it.

Unlike the `CGOperator` and the `BiCGSTABOperator`, it does not perform an iterative, but rather a direct method. Upon construction, it computes a sparse LU-like decomposition to make actual solving an efficient process.

`make`(*matrix*)

Returns an `Operator` whose effect on a vector is A^{-1}, if A is the given *matrix*, which has to be a `SparseExecuteMatrix` instance.

This is a static method.

Direct methods like the ones above are important tools, but for some types of matrices, one can do even better, by means of iterative methods, which, as an added benefit, do not require a matrix representation of the operation they are inverting:

class `CGOperator`

A `CGOperator` inverts an `Operator` given to it by means of the Hestenes/Stiefel Conjugate Gradient method.

It requires that the matrix representation of the given `Operator` be symmetric (or hermitian for complex matrices) and positive definite.

`make`(*matrix_op, max_it=None, tolerance=1e-12, precon_op=None*)

Returns a `CGOperator` that will iteratively approximate the inverse of the `Operator` *matrix_op*. *max_it* specifies a bound on the number of iterations taken to reach the goal of decreasing the residual $\sqrt{(A * x - b)^2}$ by a factor of *tolerance* (where, obviously, A is a matrix representation of *matrix_op*, b is the vector to which the `CGOperator` is applied, and x is the candidate result. If the given target precision is not reached in the given number of iterations, an exception is thrown.

precon_op, finally, is an approximation to A^{-1} that is applied once each iteration. As a preconditioner, it should be computationally inexpensive–e.g., if an application of A takes $O(n)$ computations, so should the preconditioner.

Notice that neither *matrix_op* nor *precon_op* are needed in matrix form–they are only passed in as `Operator` instances.

This is a static method.

class `BiCGSTABOperator`

A `BiCGSTABOperator` inverts an `Operator` given to it by means of the BiCGSTAB method.

It relieves the symmetry requirement of the CG method, but may break down for some matrices.

`make`(*matrix_op, max_it=None, tolerance=1e-12, precon_op=None*)

Returns a `BiCGSTABOperator` that will iteratively approximate the inverse of *matrix_op*. For the parameters, see `CGOperator.make`.

Notice that neither *matrix_op* nor *precon_op* are needed in matrix form–they are only passed in as `Operator` instances.

This is a static method.

One preconditioner that is usable with the above iterative methods is available as part of NumPy:

class `SSORPreconditioner`

A `SSORPreconditioner` computes an approximate inverse of the given matrix: Let L be the lower-left submatrix not including the diagonal, and D the diagonal part. Then, for a given parameter ω, the SSOR preconditioner is given by

$$(2-\omega)(D+\omega L)^{-H}(\omega D)(D+\omega L)^{-1},$$

which can be rather efficiently implemented.

ω is usually chosen to be one.

It requires the matrix to be symmetric (or hermitian in the complex case).

`make`(*matrix, omega=1*)

Returns an `Operator` whose effect on a vector is A^{-1}, if A is the given *matrix*, which has to be a `SparseExecuteMatrix` instance.

This is a static method.

4.4 Implementing your own `Operators`

TO BE WRITTEN. Refer to the source in 'src/operator.py', class `LUInverseOperator` for an example. FIXME

For information on implementing your own operators in C++, please refer to Chapter 6.

CHAPTER

FIVE

Numerics with PyLinear

This chapter introduces the numerical algorithms available in PyLinear.

PyLinear features six different modules of numerical algorithms:

- `pylinear.operation` uses the previously introduced notion of an `Operator` and offers several implementations of the concept. It is also the main module of linear algebra computational routines in PyLinear. It offers a comprehensive set of linear algebra primitives, such as determinants, decompositions, linear solves, eigenvalue finding and the like. While the `Operator`-based functions have been described in Chapter 4, the conceptually simpler function-call interfaces are described in Section **??**.
- `pylinear.linear_algebra` is a compatibility module which aims for complete exchangeability with NumPy's `LinearAlgebra`. It offers a high-level subset of `pylinear.operation` with less exposed detail. See its documentation, which was made available as part of NumPy and numarray.
- `pylinear.mpi` will provide an interface between MPI and PyLinear, but is not yet written.
- `pylinear.toybox` serves as a staging area for the above modules and has an unspecified interface that may change at any time. Look in the source to find experimental algorithms that may solve your problems, but be warned that these may disappear or change at any time.

5.1 Querying available functionality

Some features in PyLinear depend on outside software (such as BLAS and LAPACK). Many of these software packages are optional, and may or may not have been available when PyLinear was compiled. Whether or not this was the case, PyLinear still promises to function, albeit with reduced functionality.

The following functions in the module `pylinear` enable you to query whether certain functionality is available:

`has_blas``()`
: Returns a `bool` indicating whether BLAS was available at compile time.

`has_lapack``()`
: Returns a `bool` indicating whether LAPACK was available at compile time.

`has_arpack``()`
: Returns a `bool` indicating whether ARPACK was available at compile time.

`has_umfpack``()`
: Returns a `bool` indicating whether UMFPACK was available at compile time.

`has_daskr``()`
: Returns a `bool` indicating whether DASKR was available at compile time.

If a given function depends on some external package, the relevant documentation section will state this. Section 2.1 provides a description of these packages.

5.2 Matrix computations

`solve_linear_system`(*matrix, rhs*)
Solves the linear system `matrix*solution=rhs`. Uses LU decomposition or UMFPACK, depending on availability and sparseness of *matrix*.

Returns the vector `solution`.

`solve_linear_system_cg`(*matrix, rhs*)
Solves the linear system `matrix*solution=rhs`, where *matrix* is symmetric and positive definite. Uses the Hestenes-Stiefel Conjugate Gradient method. See also `CGOperator`.

`cholesky`(*matrix*)
Returns the Cholesky decomposition L of *matrix*. If we let A be equal to *matrix*, then L satisfies $A = LL^H$.

`lu`(*matrix*)
Returns a tuple `(l, u, perm, sign)` that represents the LU decomposition of *matrix*.

Let A be equal to *matrix*, L equal to *l*, U equal to *u* and P equal to a permutation matrix with $P_{i,j} = 1$ iff `perm[i]=j`. Then the LU decomposition satisfies $LU = PA$.

See also `make_permutation_matrix`, which turns *perm* into a matrix like P.

`eigenvalues`(*matrix*)
Returns a sequence (of unspecified type) that contains all eigenvalues of *matrix*. Requires LAPACK.

`diagonalize`(*matrix*)
Returns a tuple `(vr, w)` of a matrix *vr* and a vector *w*.

Let A be equal to *matrix*, V equal to *vr*, and D equal to a *u* and P equal to a permutation matrix with $P_{i,j} = 1$ iff `perm[i]=j`. Then the LU decomposition satisfies $LU = PA$.

5.3 Convenient helpers

CHAPTER

SIX

Extending PyLinear

This chapter ...

FIXME

6.1 Implementing custom operations

6.2 Implementing custom `Operators`

CHAPTER

SEVEN

Differences to NumPy and numarray

This chapter outlines the differences between PyLinear and the packages Numerical Python and numarray.

Unlike NumPy, PyLinear does *not* allow more than two or less than one index dimension, i.e. it only suports objects with one and two indices, also known as vectors and matrices. In fact, even vectors and matrices are different types internally, while NumPy glosses over these differences and makes them all a single *array type*. This makes sense since NumPy's focus is on array-shaped data, such as images and measurements, while PyLinear's focus is on linear algebra.

- Ufunc methods are not supported.
- Slices copy, do not return views.

APPENDIX

A

Acknowledgements

PyLinear owes much to the heroes who fleshed out Numerical Python, numarray, numpy and their corresponding interfaces. In particular, some parts of this manual are shamelessly borrowed from numarray, as are a few docstrings.

PyLinear was written as part of a Diplom thesis at the Institut für Angewandte Mathematik at Universität Karlsruhe (TH), Germany. I am grateful that my advisor, Prof. Dr. Willy Dörfler, gave me the freedom to choose to write my own matrix package as part of my thesis. The package was significantly enhanced and released to the public during a paid research stay at his institute, whose support I gratefully acknowledge.

PyLinear has also benefitted from discussions I had with Roman Geus of the Paul Scherrer Institute in early stages of the project.

Last, but not least, PyLinear would not even exist if it weren't for Python and the Boost C++ libraries.

INDEX

`()`
    operator, 29
`*`
    operator, 30
`+`
    operator, 29
`__iter__()` (method), 26

`absolute()` (in module), 24
`add()` (in module), 24
`add_element()` (Matrix method), 26
`add_scattered()` (method), 27
`apply()` (Operator method), 29
`argument()` (in module), 24
array, 7
`array()` (in module), 8
`asarray()` (in module), 9

`BiCGSTABOperator` (class in), 31
broadcast, 14
broadcasting, 17
Build
    Matrix, Sparse, 7

`ceil()` (in module), 24
`CGOperator` (class in), 31
`cholesky()` (in module), 34
code
    type, 8
`complete_index1_data()` (SparseExecuteMatrix method), 26
`conjugate()` (in module), 23
`copy()` (method), 27
`cos()` (in module), 23
`cosh()` (in module), 23
cross
    product, 22

Data
    Type, 7
data
    type, 8
Dense
    Matrix, 7
    Vector, 7
`diagonal()` (in module), 16
`diagonalize()` (in module), 34
dimension, 8
`divide()` (in module), 24
dtype, 7, 8
`dtype` (Array attribute), 11

`eigenvalues()` (in module), 34
Elementwise
    Function, 23
`empty()` (in module), 9
environment variables
    LD_LIBRARY_PATH, 5
Execute
    Matrix, Sparse, 7
`exp()` (in module), 23
`eye()` (in module), 10

Flavor, 7
`flavor` (Array attribute), 12
`floor()` (in module), 24
Function
    Elementwise, 23
    Universal, 23

`H` (Matrix attribute), 25
`has_arpack()` (in module), 33
`has_blas()` (in module), 33
`has_daskr()` (in module), 33
`has_lapack()` (in module), 33
`has_umfpack()` (in module), 33
`hsplit()` (in module), 16
`hstack()` (in module), 10

`imaginary` (Array attribute), 15
`indices()` (method), 27

Kronecker
    product, 22

LD_LIBRARY_PATH, 5
log() (in module), 23
log10() (in module), 23
lu() (in module), 34
LUInverseOperator (class in), 30

make() (BiCGSTABOperator method), 31
make() (CGOperator method), 31
make() (LUInverseOperator method), 30
make() (MatrixOperator method), 30
make() (SSORPreconditioner method), 32
make() (UMFPACKOperator method), 31
Matrix
    Dense, 7
    Sparse Build, 7
    Sparse Execute, 7
matrix, 7
MatrixOperator (class in), 30
maximum() (in module), 24
minimum() (in module), 24
multiply() (in module), 24

ones() (in module), 9
Operator, 29
Operator (class in), 29
operator
    (), 29
    *, 30
    +, 29

power() (in module), 24
product
    cross, 22
    Kronecker, 22
    Tensor, 22

real (Array attribute), 15

set_element() (method), 25
set_element_past_end() (Matrix method), 26
shape (Array attribute), 11
shape (Operator attribute), 29
sin() (in module), 23
sinh() (in module), 23
solve_linear_system() (in module), 34
solve_linear_system_cg() (in module), 34
solve_lower() (method), 28
solve_upper() (method), 27
sort() (SparseBuildMatrix method), 25
Sparse
    Build Matrix, 7
    Execute Matrix, 7
sparse() (in module), 9
sqrt() (in module), 23
SSORPreconditioner (class in), 31
subtract() (in module), 24
sum() (method), 26

T (Matrix attribute), 25
take() (in module), 16
tan() (in module), 23
tanh() (in module), 24
Tensor
    product, 22
trace() (in module), 25
tri() (in module), 10
Type
    Data, 7
type
    code, 8
    data, 8
typecode, 8
typecode() (Operator method), 29

UMFPACKOperator (class in), 31
Universal
    Function, 23

Vector
    Dense, 7
vector, 7
vsplit() (in module), 16
vstack() (in module), 10

zeros() (in module), 9