CySparse's users manual

Release 0.2.2

Nikolaj van Omme
Sylvain Arreckx
Dominique Orban

December 23, 2015

1 Introduction **3**
1.1 What is **CySparse**? . . . 3
1.2 Content . . . 3
1.3 **PySparse** legacy . . . 3
1.4 **PySparse** vs **CySparse** . . . 3
1.5 License . . . 4

2 CySparse installation **5**
2.1 **Python** installation mode . . . 5
2.2 **Cython** installation mode . . . 7

3 CySparse's basics **9**
3.1 Sparse Matrices and other matrix-like objects . . . 9
3.2 Storage schemes . . . 11
3.3 Types . . . 11
3.4 Common type attributes . . . 12
3.5 What to import . . . 12
3.6 Factory methods . . . 12

4 CySparse's types **15**
4.1 Compatibility with **NumPy** . . . 15
4.2 Available types . . . 15
4.3 Number's behavior in **CySparse** . . . 16

5 How to create a matrix? **19**
5.1 Sparse matrices all come from a `LLSparseMatrix` . . . 19
5.2 `LLSparseMatrix` matrices must be instantiated by a factory method . . . 19
5.3 Helpers . . . 20

6 Mutable matrices: LLSparseMatrix and LLSparseMatrixView **21**
6.1 The `LLSparseMatrix` class . . . 21
6.2 The `LLSparseMatrixView` class . . . 22

7 Imutable matrices: CSR-, CSC- and CSBSparseMatrix **25**
7.1 The `CSRSparseMatrix` class . . . 25
7.2 The `CSCSparseMatrix` class . . . 25
7.3 The `CSBSparseMatrix` class . . . 25

8 Sparse Matrix Proxies **27**
8.1 Available proxies . . . 27
8.2 Basic operations . . . 28

8.3 What if I need the full scale corresponding matrix? . . . 28

9 Multiplication with a `NumPy` vector **29**
9.1 Optimization . . . 29
9.2 `matvec()` . . . 30
9.3 `matvec_transp` . . . 30
9.4 `matvec_htransp` . . . 30
9.5 `matvec_conj` . . . 30
9.6 What about sparse vectors? . . . 30

10 IO formats **33**
10.1 Text formats . . . 33
10.2 Binary formats . . . 33

11 `CySparse` for `Cython` users **35**
11.1 How to compile with `Cython`? . . . 35
11.2 Creation of matrices . . . 35
11.3 Accessing matrix elements . . . 35

12 Indices and Tables **37**

Release 0.2

Date December 23, 2015

CHAPTER

ONE

INTRODUCTION

Welcome to **CySparse**'s users manual!

We tried to keep this manual as simple and short as possible. You can find much more detailed information in the developer's manual. To see **CySparse** in action, you can try the tutorials written in IPython Notebooks or run the examples in the `examples` directory.

1.1 What is CySparse?

CySparse is a fast sparse matrix library for **Python/Cython**.

1.2 Content

1.3 PySparse legacy

1.4 PySparse vs CySparse

Even if **CySparse** is (strongly) inspired from **PySparse**, there are notable differences. In short, **CySparse**:

- allows the use of matrices with **different types** of indices and elements at run time (see ...);
- is **faster** than **PySparse** (see ...);
- uses **matrix views** - a very light proxy object - that represent parts of a matrix without the need to copy elements (see...);
- has more **syntaxic sugar**, like `A * b, b * A, A.T * b` etc.
- has a **symmetric** version of **all** its matrix types.
- **doesn't use masks**.

Both libraries define similar but also different matrix classes:

Matrix type	**PySparse**	**CySparse**
Linked-List Format	`ll_mat`, `ll_mat_sym`, `PysparseMatrix`	`LLSparseMatrix`
Compressed Sparse Row Format	`csr_mat`	`CSRSparseMatrix`
Compressed Sparse Column Format	•	`CSCSparseMatrix`
Sparse Skyline Format	`sss_mat`	•
Compressed Sparse Row and Column Format	•	`CSBSparseMatrix`

1.5 License

CHAPTER

TWO

CYSPARSE INSTALLATION

There are basically [1], two modes to install **CySparse**:

- **Python** mode and
- **Cython** mode.

In **Python** mode, you install the library as a usual **Python** library. In **Cython** mode a little bit more work is involved as you also need to generate the source code from templated files.

The installation is done in a few simple steps:

1. Clone the repository;
2. Install the dependencies;
3. Tweak the configuration file `cysparse.cfg`;
4. Generate the source code if needed (i.e. in **Cython** mode);
5. Compile and install the library:

We detail these steps in the next sections for both installation modes.

2.1 Python installation mode

2.1.1 Clone the repository

2.1.2 Install the depencies

All **Python** dependencies are described in the `requirements.txt` files. You can easily install them all with:

```
pip install -r requirements.txt
```

or a similar command. Other dependencies need some manual installation. Read further.

CySparse

- **Cython**
- **Jinja2**
- argparse
- fortranformat

[1] Some special configurations might need a complete or partial **Cython** source generation.

- **SuiteSparse** (for the moment, it not possible to install **CySparse without SuiteSparse**)

Documentation

- **Sphinx**
- sphinx-bootstrap-theme

Unit testing

- **PySparse**

Performance testing

- **PySparse**
- benchmark.py (https://github.com/optimizers/benchmark.py)

2.1.3 Tweak the configuration file `cysparse.cfg`

[THIS IS WORK IN PROGRESS]

log file name **without** extension (by default, we use '.log') log_name = cysparse_generate_code # DEBUG/INFO/WARNING/ERROR/CRITICAL log_level = INFO console_log_level = WARNING file_log_level = WARNING

32bits/64bits # if left blank, we use INT64_t on 64 bits platforms and INT32_t on 32 bits platforms DEFAULT_INDEX_TYPE =

2.1.4 Generate the source code

Some parts of the library source code have to be generated **if** you use **Cython** or wish to generate the code from scratch. We use a script:

```
python generate_code.py -a
```

The switch `-a` stands for `--all` and generates the entire library. If you need help, try the `-h` switch.

2.1.5 Compile and install the library

The preferred way to install the library is to install it in its own *virtualenv*.

Wheter using a virtual environment or not, use the traditionnal:

```
python setup.py install
```

to compile and install the library.

2.2 Cython installation mode

2.2.1 Inconveniences

- Sometimes **Cython** can ask for a complete recompilation. Whenever this happens, it displays the following message when trying to import the library into **Python**:

```
ValueError: XXX has the wrong size, try recompiling
```

  where XXX is the first class that has the wrong size. The easiest way to deal with this is to recompile all the .pyx files again (you can force this by removing all the .c files) [2].

  See Robert Bradshaw's answer. See also enhancements distutils_preprocessing.

- **If** you modify the templated code, some dependencies might be missing in the (generated) `setup.py` file and require manual intervention, i.e. recompilation. The easiest way to go is to recompile everything from scratch [3]. First delete the generated files:

```
python generate_code.py -ac
```

  where `-ac` stands for `a`ll and `c`lean. This will delete **all** generated `.pxi`, `.pxd` and `.pyx` **Cython** files. Then delete the generated **C** files:

```
python clean.py
```

  This will delete **all C** `.c` files. You can then recompile the library from scratch.

[2] The problem is interdependencies between source files that are not catched at compile time. Whenever **Cython** can catch them at runtime, it throws this `ValueError`.

[3] Interdependencies between generated templates are **not** monitored. Instead of recompiling everything from scratch, you can also simply delete the corresponding **Cython** generated files. This will spare you some compilation time.

CHAPTER

THREE

CYSPARSE'S BASICS

This chapter presents the basics of **CySparse**. It covers a lot but you don't need to understand all the details. Instead, treat it as a gentle warm up for the rest of the manual.

3.1 Sparse Matrices and other matrix-like objects

The main objects of the **CySparse** library are *sparse matrices*: matrices that typically don't contain much elements that are different that `0`. Say otherwise, sparce matrices contain lots of `0`. To gain space and for efficiency reasons we don't store these zero elements explicitly [1]. Sparse matrices behave like usual matrices but behind the scene, their implementation use lots of trickery to use partial arrays that only store non zero elements. Several implementations exists with each their pros and cons. For instance, if you want to multiply a sparse matrix with a vector, you should use a `CSR` matrix, i.e. a *Compressed Sparse Row* format matrix. Don't worry, in **CySparse**, there are only **3** types of sparse matrices:

- Linked-list format (**LL**): a convenient format for creating and populating a sparse matrix.
- Compressed sparse row format (**CSR**): a format designed to speed up matrix-vector products.
- Compressed sparse column format (**CSC**): a format designed to speed up vector-matrix products.

Linked-list (LL) format matrices are well suited to **create** and/or **modify** a matrix: you can easily add or delete elements for such matrices. Compressed sparse row (CSR) and Compressed sparse column (CSC) formats are more suited for optimized vector multiplication and are difficult to modify. In **CySparse** we even don't allow you to modify CSR or CSC matrices!

Matrices are the big thing in **CySparse** but these objects are still heavy and sometimes you don't really need to create them. Enter the matrix *proxies* and matrix *views*. These object behave like matrices but are **not** matrices. A small example will clarify this. Imagine you have a matrix A and you need to multiply its transposed with a vector. One way would be to create the transposed matrix and multiply it by this vector. Actually, you **don't** need to create the transpose of a matrix for that! **CySparse** let you play with a proxy transposed matrix, a fake matrix if you wish:

```
A = ...  # this is a sparse matrix
v = ...  # this is a vector

w = A.T * v
```

To compute the vector `w`, **CySparse** doesn't need to actually create a transposed matrix. The `T` attribute is used for the transposed matrix of `A`. No computations are done. When the time comes to do some computations, like multiplying this transposed `A.T` with a vector `v`, **CySparse** behind the scene computes this product directly from the matrix `A` and the vector `v` **without** creating the transposed matrix `A.T`. `A.T` behaves like a matrix but is not a matrix. If you want to modify the transposed, which is **not** allowed, like this:

[1] In some cases thought, it **is** worth keeping **some** zeros explicitly. This can be done in **CySparse** through the use of the `store_zero` parameter. Read further.

```
A.T[1,4] = 6
```

it throws an exception:

```
TypeError: 'cysparse.sparse.sparse_proxies.t_mat.TransposedSparseMatrix'
object does not support item assignment
```

The type of the `A.T` object seems complicated and it is. The reason is again efficiency. However, except when dealing with such error messages, the types of the objects don't matter for the users as **CySparse** takes care of such intricacies. For instance, the product seen above of the transposed with a vector looks very much like it is mathematically written:

$$A^t * v$$

There is only one more type of objects that behaves like a matrix but isn't: views. We tried to assign a new value to an element of the `A.T` proxy matrix and this is not allowed. It can happen that sometimes, you need to have access to a sub-matrix and even change some elements of this sub-matrix. Enter the **views** that as their name implies allow to have a *view* over a matrix:

```
A = ... # this is a sparse matrix
B = A[3:5, 2:7]
```

`B` is a *view* and corresponds to the sub-matrix of `A` made by its rows 4 to 5 (**Python** starts couting elements from `0`) and its columns 3 to 6. You probably recognized the *slice* notation and it is exactly that.

```
C = A[:, ::2]
```

`C` corresponds to the sub-matrix of `A` made by all its rows and every two columns. Actually, views can be much more complicated than that:

```
D = A[[1, 1, 1, 2 , 0], [3, 2 , 1, 0]]
```

Here we use two lists and `D` corresponds to a matrix that is no longer a sub-matrix of `A`: it is constructed by taking 3 times the second row, then adding the third one and then the first one combined with the first four columns in reversed order. We could also write:

```
D = A[[1, 1, 1, 2 , 0], 0:4:-1]
```

What about modifying an element of a view?

```
B[1, 4] = 42
```

is perfectly fine. That is, because `B` is a 2×5 matrix. We could have stored the value `42` in the view but that would complicate things a little bit too much and we would loose all efficiency. Instead, when you modify a view, you modify the original matrix! By assigning `42` to element $(1, 4)$ of `B` you are in fact assigning `42` to element $(4, 6)$ of the original matrix `A`!

Views only exist for **LL** format matrices but because you read carefully, you already knew that [2].

That's it. You have already seen **all** the main objects of the library! Of course, we need to talk a little bit more about the details and the common uses.

You can read more on:

- CSC and CSR matrices: *Imutable matrices: CSR-, CSC- and CSBSparseMatrix*;
- LL matrices and views: *Mutable matrices: LLSparseMatrix and LLSparseMatrixView*;
- proxy matrices: *Sparse Matrix Proxies*.

[2] Remember that the LL format matrices are the only ones you can modify in **CySparse**!

3.2 Storage schemes

[TO BE REWRITTEN]

3.2.1 Common strorage attributes

For efficiency reasons, **CySparse** can use different storage schemes/methods to store matrices. For instance, symmetric matrices can be stored with only half of their elements.

store_symmetric

Symmetric matrices can be stored by only storing the **lower** triangular part and the diagonal of a matrix. To create a symmetric matrix, add the arguement `store_symmetric=True` to the call of one of the factory methods. The attribute `store_symmetric` returns if this storage method is used or not. Thus, if `store_symmetric` is `True`, you know that you deal with a symmetric matrix **and** that roughly only half of its elements are stored. If `store_symmetric` is `False`, it simply means that this storage scheme is not used. The matrix itself migth be symmetric or not.

store_zero

By default non specified (implicit) elements are zero (`0`, `0.0` or `0+0j`). **CySparse** allow the user to store explicitely zeros. To explicitely store zeros, declare `store_zero=True` as an argument in any factory method:

```
A = LLSparseMatrix(store_zero=True, ...)
```

The matrix `A` will store any zero explicitely as will any matrix created from it. You can access the value of this attribute:

```
A.store_zero
```

returns `True` for our example. This attribute is read-only and cannot be changed. If you want to temporarily exclude zeros in some operations, you can use the `NonZeros` context manager:

```
with NonZeros(A):
    # use some method to add entries to A but disregard zeros entries
    ...
```

This context manager temporarily set the `store_zero` attribute to `False` before restoring its inital value.

By default, `store_zero` is set to `False`.

is_mutable

`is_mutable` returns if the matrix can be modified or not. Note that for the moment, **only** an `LLSparseMatrix` matrix can be modified.

3.3 Types

Each library has been written with a main goal in mind. We tried to optimize **CySparse** for speed. Its code is therefor highly optimized and we use *typed* **C** variables internally. All elements inside a matrix have the same well-defined types. Even indices are typed! For efficiency reasons, we often don't allow XXX

You will not be surprised that in **CySparse**, we **strongly** discourage the mixing of types. Actually, most operations require to deal with objets that have **exactly** the same types for their indices **and** their elements. As a rule of thumb, try not to mix objects with different types. We call this our *Golden Rule*.

Important: The Golden Rule:

Never mix matrices with different types!

If you follow this rule, you will not run into troubles in **CySparse**.

NumPy and **SciPy** let you mix matrices with different types. You can even declare your own type for the elements of a matrix! This flexibility has a cost as you can see in our benchmarks. **CySparse** is not as flexible but is some order of magnitude more efficient.

[TO BE REWRITTEN]

3.4 Common type attributes

3.4.1 `dtype` and `itype`

Each matrix (matrix-like) object has an internal index *type* and stores *typed* elements. Both types (enums) can be retrieved. `dtype` returns the type of the elements of the matrix and `itype` returns its index type.

See section *Available types* about the available types.

3.5 What to import

3.6 Factory methods

[TO BE REWRITTEN]

3.6.1 Common content attributes

`nrow` and `ncol`

`nrow` and `ncol` give respectively the number of rows and columns. You also can grab both at the same time with the `shape` attribute:

```
A = ...
A.shape == A.nrow, A.ncol  # is True
```

You can use `nrow` and `ncol` as arguments to construct a new matrix. Whenever the number of rows is equal to the number of columns, i.e. when the matrix is square, you can instead use the argument `size=...` in most factory methods.

`nnz`

The `nnz` attribute returns the number of "non zeros" stored in the matrix. Notice that `0` could be stored if `store_zero` is set to `True` and if so, it will be counted in the number of "non zero" elements. Whenever the

symmetric storage scheme is used (`store_symmetric` is `True`), `nnz` only returns the number of "non zero" elements stored in the lower triangular part and the diagonal of the matrix, i.e. `nnz` returns exactly how many elements are stored internally.

Warning: `nnz` returns **exactly** the number of elements stored internally.

When using views, this attribute is **costly** to retrieve as it is systematically recomputed each time and we don't make any assomption on the views (views can represent matrices with rows and columns in any order and duplicated rows and columns any number of times). The number returned is the number of "non zero" elements stored in the equivalent matrix using the **same** storage scheme than viewed matrix.

`is_symmetric`

[TODO in the code!!!]

Returns if the matrix is symmetric or not. While matrices using the symmetric storage (`store_symmetric == True`) are symmetric by definition and `is_symmetric` returns immediatly `True`, this attribute is costly to compute in general.

3.6.2 Common string attributes

Some attributes are stored as `C` struct internally and can thus not be accessed from **`Python`**. We do however provide some strings for the most important ones.

`base_type_str` and `full_type_str`

Each matrix or matrix-like object has its own type and type name defined as strings. For instance:

```
A = NewLLSparseMatrix(size=10, dtype=COMPLEX64_T, itype=INT32_T)
print A.base_type_str
print A.full_type_str
```

returns

```
LLSparseMatrix
LLSparseMatrix [INT32_t, COMPLEX64_t]
```

The type `LLSparseMatrix` is common among `LL` sparse format matrices while the `full_type_str` gives the specific details of the index and element types.

CHAPTER

FOUR

CYSPARSE'S TYPES

Internally, **CySparse** uses typed variables. As a **Python** user, you don't have directly access to these types but you can ask for a specific type to be used through the use of simple (enum) arguments. As a **Cython** user however, you do have access to these types and you can (and should) use them in your **Cython** programs. For every type, there is a corresponding (enum) argument. Types are written with `_t` as suffix: `INT32_t`, `COMPLEX256_t`. The corresponding (enum) arguments are simply the names of the types with `_T` as suffix. Thus the (enum) argument corresponding to type `FLOAT64_t` is `FLOAT64_T` (notice the capital `T`).

4.1 Compatibility with NumPy

In short, **CySparse**'s types are 100% compatible with the **NumPy** corresponding types [1]. There are some differences thought between the way both libraries treat their types and numbers. Read on.

Warning: The behavior of similar types in **NumPy** and **CySparse** can be different!

4.2 Available types

CySparse has some types that can be used for indices and some types that can be used for elements.

4.2.1 Indices: integer numbers

Two index types exist: signed 32 and signed 64 bits integers. Internally, they are named `INT32_t` and `ÌNT64_t`.

4.2.2 Elements: integers, real and complex numbers

CySparse has three different families of element types:

- integers: simple and double precision signed integers (`INT32_t` and `INT64_t`).
- real numbers: simple, double and quadruple precision real numbers (`FLOAT32_t`, `FLOAT64_t` and `FLOAT128_t`).
- complex numbers: simple and double precision complex numbers (`COMPLEX64_t`, `COMPLEX128_t`, `COMPLEX256_t`).

[1] Behind the hood, both libraries use **C99** types (whenever **NumPy** is compiled with a **C99** compliant compiler). **CySparse** doesn't offer as many different types as **NumPy** though«.

Quadruple precision has some limited support in **C99** standard and **Cython** and thus in **CySparse**. The same can be said about complex number in general although the simple and double precision are quite well integrated. This is worth a warning:

Warning: Quadruple precision and complex numbers have some limitations.

4.3 Number's behavior in CySparse

The three families of element types behave somewhat differently.

4.3.1 Overflow on assignment

Numbers do overflow [2]. When assigning an integer number a value too big for its type, an `OverflowError` is raised.

```
B = ll_mat.NewLLSparseMatrix(size=2, dtype=types.INT32_T)

B[1, 1] = 2**31
```

raises an `OverflowError` with the following message:

```
Traceback (most recent call last):
  File "new_types.py", line 102, in <module>
    B[1, 1] = 2**31
  File "cysparse/sparse/ll_mat_matrices/ll_mat_INT32_t_INT32_t.pyx", line 538, in cysparse.sparse.ll_
OverflowError: value too large to convert to int
```

When assigning a real or complex number a value that is too big for its type, it is assigned *inf*, i.e. $+\infty$ **without** any warning:

```
B = ll_mat.NewLLSparseMatrix(size=2, dtype=types.FLOAT32_T)

B[0, 0] = 232
B[0, 1] = 1.3
B[1, 1] = 2**310

B.print_to(sys.stdout)
```

prints:

```
LLSparseMatrix [INT32_t, FLOAT32_t] (G, NZ, [2, 2])
232.000000  1.300000
  0.000000       inf
```

4.3.2 Overflow on operation

We follow the **C99** standard and let an overflow during an operation pass silently (and give strange results) without a warning.

Let's define a matrix with huge numbers:

[2] Overflow is compiler-dependent (and compilers are often system-dependent).

```
B = ll_mat.NewLLSparseMatrix(size=2, dtype=types.INT32_T)

B[0, 0] = 232
B[0, 1] = 1.3
B[1, 1] = 2**31 -1

B.print_to(sys.stdout)
```

This prints:

```
LLSparseMatrix [INT32_t, INT32_t] (G, NZ, [2, 2])
232          1
  0 2147483647
```

Multiplying `B` by itself returns a strange result:

```
C = B * B
C.print_to(sys.stdout)
```

prints:

```
53824 -2147483417
    0           1
```

4.3.3 `nan` and `inf`

Like **NumPy**, **CySparse** defines `nan` and `inf`. These are compatible with their **NumPy** counterparts and can be used interchangebly. They are used internally as the *C99* standard recommends.

```
import cysparse.types.cysparse_types as types

B = ll_mat.NewLLSparseMatrix(size=2, dtype=types.FLOAT64_T)

B[0, 0] = 232
B[0, 1] = 1.3
B[1, 1] = types.inf

B.print_to(sys.stdout)
```

This prints:

```
LLSparseMatrix [INT32_t, FLOAT64_t] (G, NZ, [2, 2])
232.000000  1.300000
  0.000000       inf
```

If we multiply `B` by itself, we obtain:

```
LLSparseMatrix [INT32_t, FLOAT64_t] (G, NZ, [2, 2])
53824.000000       inf
    0.000000       inf
```

as expected.

4.3.4 Types compatibilities (implicit castings)

Whenever an integer is assigned a real number, **CySparse** assigns the integer part, i.e. takes its `floor()` part.

```
B = ll_mat.NewLLSparseMatrix(size=2, dtype=types.INT64_T)

B[0, 0] = 232
B[0, 1] = 1.3
B[1, 1] = -0.89
```

This matrix is in fact:

```
LLSparseMatrix [INT32_t, INT64_t] (G, NZ, [2, 2])
232  1
  0  0
```

Complex number are compatible between them if they don't overflow but otherwise are **not** compatible nor with the integer neither with the real numbers.

[TO BE DONE: write an example with complex numbers]

To create a matrix, we use *factory methods* [3]: functions that return an object corresponding to their arguments. Different arguments make them return different kind of objects (matrices).

[3] The term *factory method* is coined by the Design Pattern community. The *method* in itself can be a function, method, class, ...

CHAPTER

FIVE

HOW TO CREATE A MATRIX?

Before you can use any type of sparse matrix, you **must** first instantiate an `LLSparseMatrix`. This matrix is well suited for construction but is not very optimized for most matrix operations. Once you have an `LLSparseMatrix`, you can create a specialized sparse matrix from it.

5.1 Sparse matrices all come from a `LLSparseMatrix`

The `LLSparseMatrix` matrix type is the only one that is *mutable*. You can add and/or delete elements, rows, columns, sub-matrices at will. Once you have constructed your matrix, it is time to transform it into an appropriate matrix format that is optimized for your needs. This transformation is not done in place and a copy is made. Here is an example:

```
A = ...  # A is a LLSparseMatrix
# add some elements
for i in range(n):
    for j in range(m):
        A[i, j] = ...

# once the matrix is constructed, transform it into suitable matrix format
# here to CSC
C = A.to_csc()
```

5.2 `LLSparseMatrix` matrices must be instantiated by a factory method

Matrices **must** be instantiated by one of the factory methods. For instance, to create a `LLSparseMatrix` (see ll_mat), use the following code:

```
from cysparse.sparse.ll_mat import MakeLLSparseMatrix

A =  MakeLLSparseMatrix(nrow=4, ncol=3)
```

`MakeLLSparseMatrix()` is really a function, not a class. This not very Pythonesque approach is made necessary because **`Cython`** doesn't allow the use of pure C variables as arguments in the constructors of classes [1].

If you don't use a factory method:

[1] This not exactly true. **`Cython`** allows to pass some pure C variables that can be *easily* mapped to **`Python`** arguments. The idea is that the same arguments are passed to `__cinit__()` **and** `__init__()` methods.

```
A = LLSparseMatrix()
```

you'll get the following error:

```
AssertionError: Matrix must be instantiated with a factory method
```

Warning: An `LLSparseMatrix` can **only** be instantiated through a factory method.

5.3 Helpers

5.3.1 `size`

`size` is **not** an attribute...

CHAPTER

SIX

MUTABLE MATRICES: LLSPARSEMATRIX AND LLSPARSEMATRIXVIEW

6.1 The LLSparseMatrix class

The *mutable* `LLSparseMatrix` class is the base class to **construct** and **populate** a matrix. With it you can easily add or delete elements, rows, columns, assign sub-matrices, etc. Once your matrix is constructed, you create a new *optimized* and *immutable* matrix from it. You can choose between CSR, CSC and CSB. Each has its strength and weaknesses and we cover them in depth in their respective sections.

Warning: The `LLSparseMatrix` class is **not** optimized for matrix operations!

6.1.1 Creation

6.1.2 Population

6.1.3 Accessing elements

Elements can be accessed individually or by batch, i.e. several elements can be accessed at the same time and stored in a container.

Individual access

You can use the common `[]` operator:

```
L = NewLLSparseMatrix(...)

e = L[i, j]
```

where `i` and `j` are integer indices. At all time, bounds are checked and an `IndexError` is raised if an index is out of bound. **`Cython`** users can access the elements **whithout** bound checking if desired.

Notice that if one of the argument you pass to the `[]` operator is **not** an integer, you'll get an `LLSparseMatrixView` (see *The LLSparseMatrixView class*).

Warning: If one of the argument you pass to the `[]` operator is **not** an integer, you'll get an `LLSparseMatrixView`

Batch access

Basically, you can take a submatrix (and store its elements in a `NumPy` two dimensionnal array or another `LLSparseMatrix` or get a view to it, see *The LLSparseMatrixView class*) or a list of elements

6.2 The `LLSparseMatrixView` class

What is an `LLSparseMatrixView` good for?

There are basically two reasons to use a view instead of a matrix:

- efficiency: no matrix_copy is made and the `LLSparseMatrixView` is a light object. TO BE COMPLETED
- really fancy indexing: if you need a new matrix constructed by **any** combinations of rows and columns, including index repetitions. TO BE COMPLETED.

6.2.1 Views of views

It is possible to have views of... views as the following code illustrates:

```
A = MakeLLSparseMatrix(...)
A_view1 = A[..., ...]
A_view2 = A_view1[..., ...]
```

The second `LLSparseMatrixView` is **not** a view on a view but a direct view on the original matrix `A`. The only difference between the two objects `A_view1` and `A_view2` is that the indices given in the `[..., ...]` in `A_view1[..., ...]` refer to indices of `A_view1` **not** the original matrix `A`.

An example will clarify this:

```
pass
```

6.2.2 References to the base `LLSparseMatrix`

Whenever a `LLSparseMatrixView` is created, the corresponding `LLSparseMatrix` has its internal CPython reference count incremented such that even if the matrix object is (implicitely of explicitely) deleted, it still exist in Python internal memory and can even be retrieved again through the view:

```
A = MakeLLSparseMatrix(...)
A_view = A[..., ...]

del A

A_view[..., ...] = ...  # still works!

A = A_view.get_matrix() # A points again to the original matrix
```

In the code above, the `LLSparseMatrix` pointed by the variable `A` on the first line has never been deleted from memory. If you also delete **all** `LLSparseMatrixView` objects refering to the `LLSparseMatrix` object, then it is effictively deleted by the garbage collector.

```
A = MakeLLSparseMatrix(...)
A_view = A[..., ...]

del A
del A_view

# matrix A is lost... and will be deleted by the garbage collector
```

CHAPTER

SEVEN

IMUTABLE MATRICES: `CSR-`, `CSC-` AND `CSBSPARSEMATRIX`

7.1 The `CSRSparseMatrix` class

7.2 The `CSCSparseMatrix` class

7.3 The `CSBSparseMatrix` class

CHAPTER

EIGHT

SPARSE MATRIX PROXIES

This section describes the sparse matrix proxies available in **Cysparse**. Sometimes, we can use a substitute for certain types of matrices instead of creating a full scale matrix. For instance, instead of creating the transpose of a matrix *A*, one can use *A.T*. *A.T* is a proxy and is thus **not** a real matrix. As such, it cannot replace a matrix in all circonstances but it can be used for the following operations:

- printing;
- accessing a value;
- multiplying with a **NumPy** vector;
- accessing basic attributes of a matrix (*shape*, *ncol*, *nrow*, ...);

8.1 Available proxies

Three basic proxies [1] are available:

- the transpose matrix proxy (`TransposedSparseMatrix` given by the `.T` attribute);
- the conjugate transpose matrix proxy (`ConjugateTransposedSparseMatrix` given by the `.H` attribute) and
- the conjugate matrix proxy (`ConjugatedSparseMatrix` given by the `.conj` attribute).

Each of them is unique (i.e. the user is not supposed to copy them) and automagically updates whenever the original matrix is changed (for `LLSparseMatrix` matrices).

These proxies are available for **all** sparse matrix and proxy types when they make sense. This means that the following code is legit:

```
A = NewSparseMatrix(...)

b = np.array(...)

A.T.H.conj.H.H * b
```

The expression `A.T.H.conj.H.H` has to be read for left to right: we start by taking the transpose matrix of matrix `A`, then the conjugate transpose matrix of the transpose matrix of matrix `A` etc. The chaining of proxies can be as long as memory permits and no penalty occurs, i.e. each proxy is only instantiated once.

The `H` and `conj` proxies are **only** available for complex matrices, i.e. matrices with a complex `dtype`.

Warning: The `H` and `conj` proxies are **only** available for complex matrices.

[1] Despite being *proxies* and **not** matrices, we still give them the name `...SparseMatrix`.

8.2 Basic operations

8.3 What if I need the full scale corresponding matrix?

For all proxies, the method `copy_matrix()` is available:

```
A = NewSparseMatrix(...)

Transpose_proxy = A.T

# real matrix
T = Transpose_proxy.copy_matrix()
```

`T` is now a real matrix of the same type as the original `A` matrix.

CHAPTER

NINE

MULTIPLICATION WITH A NUMPY VECTOR

One very common matrix operation is to multiply a sparse matrix with a dense **NumPy** vector. In fact, this operation is so common that we have written very specialized and optimized code for it. Because **NumPy** vectors are dense, the multiplication of a sparse matrix by a **NumPy** vector returns a new (dense) **NumPy**.

All **NumPy** vector multiplication operations work accross **all** sparse matrix types (`LLSparseMatrix`, `CSRSparseMatrix` and `CSCSparseMatrix`), **all** proxy types (`TransposedSparseMatrix`, `ConjugateTransposedSparseMatrix` and `ConjugatedSparseMatrix`) and this for **all real** and **complex** element types.

As all multiplications are **higly** optimized, you really should use the corresponding method and not use a combination of them equivalent to the one you seek. For instance,

$$A^H * x = conj(A^T * conj(x))$$

Because $A^H * x$ **is** implemented, it will be much faster than $conj(A^T * conj(x))$.

Here is the list of the available multiplications:

- $A * x$: `A.matvec(x);`
- $A^T * x$: `A.matvec_transp(x);`
- $A^H * x$: `A.matvec_htransp(x);`
- $conj(A) * x$: `A.matvec_conj(x);`

These operations can also be optained throught the corresponding proxies (with the same efficiency):

- `A.matvec(x)` is the same as `A * x`;
- `A.matvec_transp(x)` is the same as `A.T * x`;
- `A.matvec_htransp(x)` is the same as `A.H * x`;
- `A.matvec_conj(x)` is the same as `A.conj * x`;

9.1 Optimization

The following has been optimized:

- each type is using the dedicated corresponding **C** functions (for instance, the corresponding *conj* function from the standard lib);
- each operation is done at **C** level;
- no factor is copied in any way: we work on the raw **C** arrays;
- strided vectors have special dedicated code;

- we used the best loops to compute the multiplication for each type of matrix;

9.2 matvec()

9.2.1 Syntactic sugar

9.3 matvec_transp

9.4 matvec_htransp

9.4.1 Real matrices

9.5 matvec_conj

9.5.1 Real matrices

9.6 What about sparse vectors?

CySparse doesn't have a special sparse vector class. However, you can use a simple `SparseMatrix` object to represent your vector:

```
v = LLSparseMatrix(nrow=4, ncol=1)
v.put_triplet([0, 2], [0, 0], [1.0, 2.0])

A = LinearFillLLSparseMatrix(nrow=3, ncol=4)

print v
print A

C = A * v
print C
```

returns the expected results:

```
LLSparseMatrix [INT64_t, FLOAT64_t] of size=(4, 1) with 2 non zero values
<Storage scheme: General and without zeros>
 1.000000
   ---
 2.000000
   ---


LLSparseMatrix [INT64_t, FLOAT64_t] of size=(3, 4) with 12 non zero values
<Storage scheme: General and without zeros>
 1.000000   2.000000   3.000000   4.000000
 5.000000   6.000000   7.000000   8.000000
 9.000000  10.000000  11.000000  12.000000


LLSparseMatrix [INT64_t, FLOAT64_t] of size=(3, 1) with 3 non zero values
<Storage scheme: General and without zeros>
```

```
 7.000000
19.000000
31.000000
```

Of course, the result **is** a sparse matrix. Contrary to **NumPy** vectors, you need to give the right dimensions for the vector:

```
v = LLSparseMatrix(nrow=1, ncol=4)
A = LinearFillLLSparseMatrix(nrow=3, ncol=4)

A * v
```

will result in

```
IndexError: Matrix dimensions must agree ([3, 4] * [1, 4])
```

CHAPTER

TEN

IO FORMATS

Several IO formats are supported.

10.1 Text formats

10.1.1 Matrix Market format

The Matrix Market format is detailed in the Matrix Market website.

The different types of matrices that are supported are basically declared on the first line of the text file in what is called the *Matrix Market banner*:

%%MatrixMarket matrix SECOND_FIELD THIRD_FIELD FOURTH_FIELD

The start of the banner `%%MatrixMarket matrix` is mandatory. We detail the other fields:

- `SECOND_FIELD`: `coordinate` for sparse matrices and `array` for dense matrices;
- `THIRD_FIELD`: `complex` or `real` for the type of elements. Elements as stored as `C-double`. `integer` denotes integers and `pattern` is used when only the pattern of the matrix is given, i.e. only non zero values indices are given and not values.
- `FOURTH_FIELD`: denotes the storage scheme: `general`, `hermitian`, `symmetric` or `skew`.

Warning: Matrix Market matrices are **always** 1-based, i.e. the index of the first element of a matrix is `(1, 1)` not `(0, 0)`.

10.2 Binary formats

CHAPTER

ELEVEN

CYSPARSE FOR CYTHON USERS

I efficiency is a major concern to you, we strongly encourage you to use **Cython** to compile your own Python extension.

11.1 How to compile with Cython?

To compile and link your project with **CySparse**, simply download the complete source code of **CySparse** and refer to the `.pxd` files as needed.

11.2 Creation of matrices

Matrices cannot be instantiated directly in **Python** (see *LLSparseMatrix matrices must be instantiated by a factory method*). In **Cython**, the factory methods can be used **except** if this creates a *circular dependencies* between these methods. One solution is to simply invoke the protection mechanism for the creation of classes yourself:

```
from cysparse.cysparse.sparse_mat cimport unexposed_value
from cysparse.cysparse.ll_mat cimport LLSparseMatrix

LLSparseMatrix ll_mat = LLSparseMatrix(control_object=unexposed_value, ...)
```

By adding `control_object=unexposed_value` as argument, the `ll_mat` assertion in the `__cinit__()` constructor will be not be triered. Be carreful though as you are fully responsible for the creation of the matrix.

11.3 Accessing matrix elements

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search