CySparse's developers manual

Release 0.2.2

Nikolaj van Omme
Sylvain Arreckx
Dominique Orban

December 23, 2015

CONTENTS

Release 0.2

Date December 23, 2015

Welcome to **`CySparse`**'s developers manual!

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

Warning: This is the manual for **developers**.

CHAPTER ONE

INTRODUCTION

Maintening a library as **CySparse** is not a small task. This is partly due to:

- the mix of several programming languages (mainly **Cython**, **Python** and **C**);
- the use of templated source files;
- the use of several external tools (cygenja, Jinja2, Sphinx, LaTeX, etc.);
- the optimized code (several chuncks are highly optimized for speed);
- the coupling with NumPy;
- the language Cython that is not mature. They are doing a fantastic job but the language still has numerous bugs. If you've never used a compiler with bugs, welcome to this wonderful world were you might have to debug the compiler as much as your code (yep!);

CHAPTER

TWO

SPARSE MATRIX FORMATS

[THIS SECTION IS VERY DETAILED AND IS PROBABLY NOT INTENDED FOR THE COMMON USER]

This section describes the sparse matrix storage schemes available in **`Cysparse`**. In the next sections, we cover sparse matrix creation, population, view and conversion.

Basically, we use one general sparse matrix format to *create* a general sparse matrix: the `LLSparseMatrix` class. This class has a rich API and allows multiple ways to create and transform a sparse matrix. Once the matrix is ready to be used, it can be appropriately converted into a specialized sparse matrix format that is optimized for the needed operations. The `LLSparseMatrix` is also the only type of matrices that can be modified while the other types of matrices are **immutable** for efficiency.

Here is a list of existing formats and their basic use:

- Linked-list format (`LL`): a convenient format for creating and populating a sparse matrix, whether symmetric or general.
- 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.

The CSR and CSC formats are complementary and can be viewed as respectively a row- and column view of a sparse matrix.

These formats are well known in the community and you can find scores of documents about them on the internet.

2.1 Sorted column and row indices

[TO BE WRITTEN]

For all sparse matrix formats, we internally keep the indices sorted, i.e. in ascending order.

2.2 The `LL` sparse format in details

This format is implemented by the `LLSparseMatrix` class.

All matrix classes are straightforward with perhaps the only exception of the `LLSparseMatrix` class. This class is really a container where you can easily add or remove elements but it is not much more than that: a container. As such, it is **not** optimized for any matrix operation. There are some internal optimisations though to add or retrieve an element and we explain them here.

2.2.1 Elements are linked in chains (aka linked lists)

The `LLSparseMatrix` container is made of 4 one dimensionnal arrays and one pointer (`free`):

- `root[i]`: points to the first elements of row `i`;
- `col[k]`: contains the column index `j` of an element stored at the `k`-th place;
- `val[k]`: contains the value of the element stored at the `k`-th place;
- `link[k]`: contains the pointer to the next element in the chain where the `k`-th element belongs.

The pointer `free` points to the first free element in the chain of free elements.

Despite its name, *linked list* sparse matrix, this container doesn't use list of pointers but **only** arrays of indices. "Pointers" are indices in arrays and to denote a pointer to `NULL` the value `-1` if used. For instance, if `free == -1`, this means that the chain of free elements is empty.

Then chain can be traversed by following the `link` array:

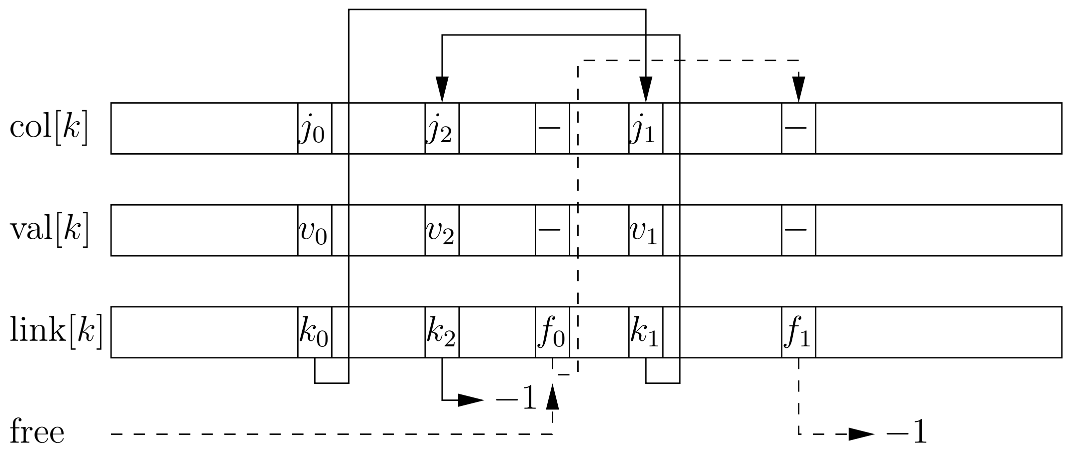

Fig. 2.1: Chains in an `LLSparseMatrix` matrix

Two chains are depicted in the picture. First, the chain with free elements. These are elements that where removed. The `free` pointer points to the first element of this chain and `link[free]` if f_0 which points to the second element in this chain. Whenever a new elements is added, it will take the place of this first element.

The second chain starts with element (j_0, v_0, k_0). k_0 points to the second element (j_1, v_1, k_1) and k_1 points to the second and last element (j_2, v_2, k_2).

Inside the arrays, the elements can be stored in any order.

2.2.2 Elements are "aligned" row wise (chains correspond to rows)

Each chain of elements corresponds to one row of the matrix. The pointer `root[i]` points to the first element of the `i` th row. If the above chain (k_0, k_1 and k_2) correspond to the only elements on row `i`, `root[i]` would point to element (j_0, v_0, k_0). If row `i` doesn't have any element, `root[i] == -1`.

To traverse the `i`th row, simply use:

```
k = self.root[i]

while k != -1:
    # we consider element A[i, j] == val
    j = self.col[k]
    val = self.val[k]
    ...
    k = self.link[k]
```

2.2.3 Inside a row, elements are ordered by column order (how to run through a `LL` matrix)

If the chain corresponding to row `i` is $k_0, k_1, \ldots, k_p$, then we know that the corresponding column indices are ordered: $j_0 < j_1 < \ldots < j_p$. When an element is added with the `put(i, j, val)` method, this new element is inserted in the right place, swapping pointers elements of `link` if necessary.

This means that looking for an element `A[i, k]`, one can simply use:

```
k = self.root[i]

while k != -1:

    if self.col[k] > j:
        # element doesn't exist
        break

    if self.col[k] == j:
        # element exists
        ...

    k = self.link[k]
```

2.2.4 Insertion of a new element in more details

The next figure represent the internal state of a `LLSparseMatrix`:

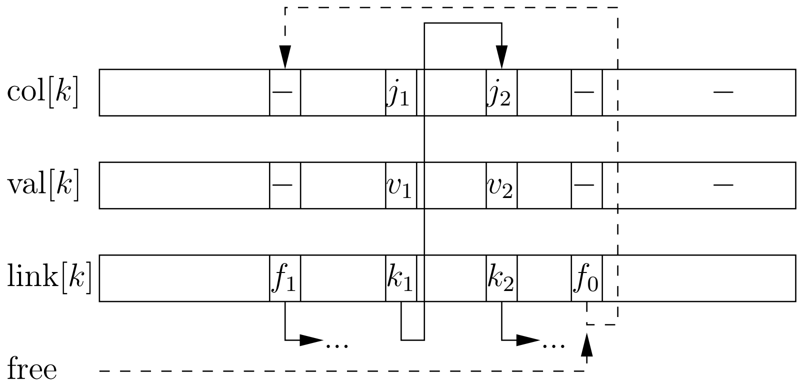

Fig. 2.2: **Before** insertion of element (j, v, k) in a `LLSparseMatrix` matrix

We have $j_1 < j_2$ and k_1 points to element k_2. Let's say we want to insert an new element (j, v, k) with column index j such that $j_1 < j < j_2$. To preserve the ordering, we have to insert this element **between** the elements k_1 and k_2 as shown on the following figure:

The element (j, v, k) was inserted in place of the first free element pointed by `free` and k_1 now points to this element. Notice also that now, `free` points to the next free element f_1.

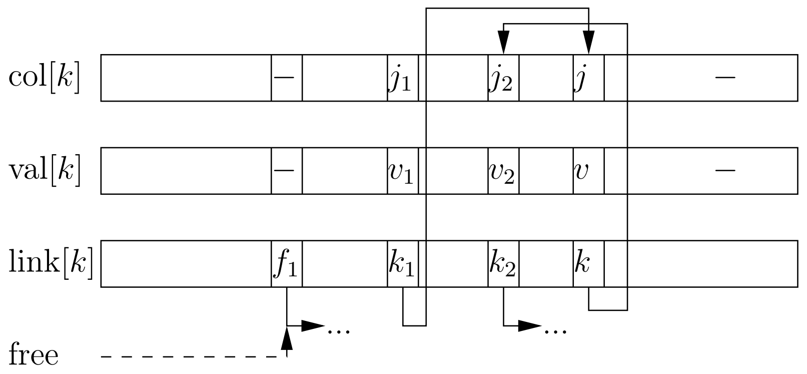

Fig. 2.3: **After** insertion of element (j, v, k) in a `LLSparseMatrix` matrix

2.2.5 Detailed example

For all sparse matrix formats, we'll detail an example. Let A be the following 3×4 sparse matrix:

$$A = \begin{bmatrix} 0 & 1 & 2 & 0 \\ 3 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 2.4: The example sparse matrix A

Notice that this matrix is sparse with 4 non zero entries, is non symmetric and has an empty row and column.

2.2.6 How to run through a `LL` matrix

To find all triplets (i, j, v):

```
for i from 0 <= i < nrow:
    k = self.root[i]
    while k != -1:
       j = self.col[k]
       v = self.val[k]

       k = self.link[k]
```

2.3 The `CSR` sparse format in details

This format is implemented by the `CSRSparseMatrix` class. This format use a row-wise representation, as the above `LL` Sparse format, i.e. elements are stored row by row.

2.3.1 Detailed example

Here are the three internal arrays for the example matrix:

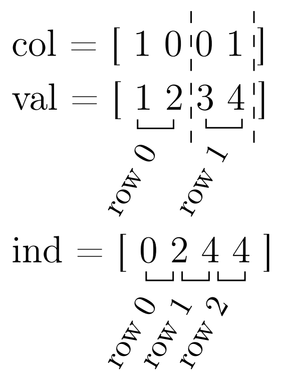

Fig. 2.5: The internal arrays of a `CSR` matrix

One can immediatly see that the values are stored row-wise in `col` and `val`: first the row `0`, than the row `1` (and nothing for row `2`). `ind` gives the first indices for each row: `ind[0] == 0` gives the start of row `0`, `ind[1] == 2` gives the start of row `1`, etc. This means that `ind[i+1] - ind[i]` returns the number of elements in row `i`.

2.3.2 How to run through a `CSR` matrix

To find all triplets (i, j, v):

```
for i from 0 <= i < nrow:
    for k from ind[i] <= k < ind[i+1]:
        j = col[k]
        v = val[k]
```

2.4 The `CSC` sparse format in details

This format is implemented by the `CSCSparseMatrix` class.

The `CSC` sparse matrix format is exactly the same as the CSR sparse matrix format but column-wise. Given a matrix A and a `CSR` representation of this matrix is exactly the same as a `CSC` respresentation of the transposed matrix A^t, i.e.

$$\text{CSR}(A) = \text{CSC}(A^t)$$

and everything we wrote about the `CSR` format transposes to the `CSC` format by exchanging rows for columns and vice-versa.

2.4.1 Detailed example

Here are the three internal arrays for the example matrix:

The values are stored column-wise in `row` and `val`: first the column `0`, than column `1` and finally column `2` (nothing for column `3`). `ind` gives the first indices for each column: `ind[0] == 0` gives the start of column `0`, `ind[1] == 1` gives the start of column `1`, etc. This means that `ind[j+1] - ind[j]` returns the number of elements in column `j`.

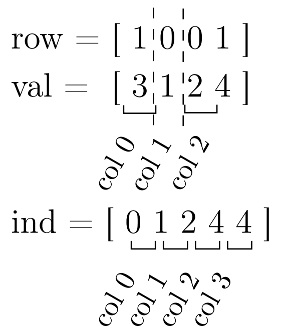

Fig. 2.6: The internal arrays of a `CSC` matrix

2.4.2 How to run through a `CSC` matrix

To find all triplets (i, j, v):

```
for j from 0 <= j < ncol:
    for k from ind[j] <= k < ind[j+1]:
        i = row[k]
        v = val[k]
```

CHAPTER

THREE

`CYSPARSE`'S SPARSE MATRIX CLASSES HIERACHY

3.1 The hierarchy at a glance

3.2 The `SparseMatrix` class

3.3 The `MutableSparseMatrix` class

3.4 The `ImmutableSparseMatrix` class

CHAPTER FOUR

CYSPARSE FOR LIBRARY MAINTAINERS

Warning: TO REWRITE COMPLETELY

The main difficulty to maintain the **CySparse** library is to understand and master the automatic code generation from templated source code. We use the template engine **Jinja2** and some hard coded conventions. We explain and justify these conventions in the following sections.

The sparse matrix formats are detailed in the section *Sparse Matrix Formats*.

4.1 Versions

The version is stored in the file `__init__.py` of the `cysparse` subdirectory:

```
__version__ = "0.1.0"
```

The version can be anything inside the quotes but this line has to be on its own and start with `__version__ = "` (notice the one space before and after the equal sign). See the function `find_version()` in the file `setup.cpy` for more details.

4.2 Meta-programming aka code generation

CySparse allows the use of different types at run time and most typed classes comes in different typed flavours. This feature comes with a price. Because we wanted to write the library completely in **Cython**, we decided to go for the explicit template route, i.e. we write **templated source code** and and use **explicit names** in the source code. This automatic generation process ask for some rigour and takes some time to master. If you follow the next conventions stricly, you should be fine. If you don't follow them then probably the code won't even compile or if it does you might generate difficult to find bugs. Trust me on this one.

Warning: Follow the conventions stricly to write templated source code.

4.2.1 Justifications

Following conventions is not always easy, especially if you don't understand them. In this sub-section we try to convince you or at least we try to explain and justify some choices I (Nikolaj) made (and try to follow).

These conventions were made with the following purpose in mind:

- respect the DRY (Don't Repear Yourself) principle;

- if the conventions are not followed, the code shouldn't compile;
- prefer static over dynamic dispatch;
- use typed variables whenever possible;
- keep the code simple whenever it doesn't sacrifice to efficiency even if the solutions are not Pythonesque;

We develop these key ideas in the following sub-sections.

Respect the DRY principle

Don't write the same code twice. This means of course than whenever you can factorize some common code, you should do so but in our case, because we lack the notion of *templates* (like **`C++`** templates), we **have** to repeat ourselves and rewrite the classes with different types. This is the main reason to use a template engine and templated code. That said, some code has been duplicated because I (Nikolaj) could not find how to make it work in **`Cython`**. One example is the proxy classes: they all share common code. I wasn't able to make them inherit from a base class [1].

If the conventions are not respected, the code shouldn't compile

To enforce the use of the conventions, we try to enforce them by the compiler (whether the **`C`**, the **`Cython`** or **`Python`** compiler). Often, you'll find that templated code have guards to ensure that types are recognized and otherwise to generate garbish that won't compile.

The name convention is written explicitely: if you don't respect it, you won't be able to use the **`generate_code.py`** script. This is on purpose.

Prefer static over dynamic dispatch

Even if **`Python`** is a dynamic language, efficient **`Cython`** code **needs** typing. This typing can be done dynamically with long and tedious `if/then` combinations or we can let the compiler do the dispatch in our place at compile time whenever possible. This is the main reason why there are as many `LLSparseMatrixView` classes as there are `LLSparseMatrix` classes. Strictly speaking, we don't need more `LLSparseMatrixView` classes than the number of index types but then you need to dynamically dispatch some operations like the creation of a corresponding ‘‘

Use typed variables whenever possible

`Cython` really shines when it can deduce some static typing, especially in numeric loops. Therefor try to type variables **if** you know their type in advance [2].

Our hope is to keep a nice balance between the difficulty of coding and the easiness to maintain the code. When generating automatically code, these two don't necessarily go hand in hand.

If you find some code that doesn't follow these conventions, report it or even better change it!

4.2.2 Types

Basic types

For different reasons [3] (???)

[1] See https://github.com/PythonOptimizers/cysparse/issues/113 for more about this issue.
[2] Use your intelligence and knowledge of **`Cython`**. Know when it makes a difference to type a variable.
[3] we use **`C99`** for its superiority compared to **`ANSI C`** (**`C89`** or **`C90`** which is the same). Among others:

We use the following basic types:

CySparse	C99 types
`INT32_t`	`int`
`UINT32_t`	`unsigned int`
`INT64_t`	`long`
`UINT64_t`	`unsigned long`
`FLOAT32_t`	`float`
`FLOAT64_t`	`double`
`FLOAT128_t`	`long double`
`COMPLEX64_t`	`float complex`
`COMPLEX128_t`	`double complex`
`COMPLEX256_t`	`long double complex`

Two categories of types

We allow the use of different types at two levels:

- for the indices (`INT32_t` and `INT64_t`)[4];
- for the matrix elements (**all** the basic types).

Add (or remove) a new type

4.2.3 Conventions

File names and directories

To keep the generation of code source files as simple as possible, we follow some conventions. This list of conventions is **strict**: if you depart from these conventions, the code will **not** compile.

- **Don't** use fused types: this feature is too **experimental**.
- Template files have the following extensions:

Cython	**CySparse** template	File type
`.pxd`	`.cpd`	Definition files.
`.pyx`	`.cpx`	Implementation files.
`.pxi`	`.cpi`	Text files to insert verbatim.

  For python files:

Python	**CySparse** template	File type
`.py`	`.cpy`	Python module files.

- Any *template* directory must **only** contain the template files and the generated files. This is because all files with the right extension are considered as templates and all the other files are considered as generated (and can be thus automatically erased). This clear distinction allows also to have a strict separation between automatically generated files and the rest of the code.

- the INFINITY and NAN macros;
- its complex types;
- inline functions;

[4] We don't want to enter into the debate unsigned vs signed integers. Accept this as a fact. Beside, we use internally negative indices.

- Index types are replaced whenever the variable `@index@` is encountered, Element types are replaced whenever the variable `@type@` is encountered.
- Generated **file names**:
  - for a file `my_file.cpx` where we only replace an index type `INT32_t`: `my_file_INT32_t.pyx`;
  - for a file `my_file.cpx` where we replace an index type `INT32_t` **and** an elment type `FLOAT64_t`: `my_file_INT32_t_FLOAT_t.pyx`.
- Generated **class/method/function names**:

`Jinja2` conventions

4.2.4 Automatic generation scripts

All generated files can be generated by invoking a **single** script:

```
python generate_code.py
```

4.3 Conventions

4.3.1 Names

4.3.2 Types

All classes are typed and *almost* all algorithms used specialized typed variables. Many algorithm are specialized for **one** type of variable. This allows to have optimized algorithms but at the detriment of being able to mix types. For instance, most of the methods of sparse matrices only works for **one** `dtype` and **one** `itype`.

4.3.3 How to expose `enum`s to `Python`

Even if recently **`Cyhton`** exposes automagically `enums` to **`Python`** (see https://groups.google.com/forum/#!topic/cython-users/gn1p6znjoyE), don't count on it. The convention is to expose equivalent strings to the user. This string is then translated internally by the corresponding `enum`. For instance, the `enum` value `UMFPACK_A` **`UmfPack`** system parameter can be given as the string *'UMFPACK_A'* by the user (as a parameter to a *solve()* method for instance). Internally, this string is translated:

```
def solve(..., umfpack_sys_string='UMFPACK_A', ...):
    cdef:
        int umfpack_sys = UMFPACK_SYS_DICT[umfpack_sys_string]
    ...
```

In **`Cython`** code, you are free to directly use the `enum` itself.

4.4 Class hierarchy

The class hierarchy may seems strange at first and indeed is strange. In my wildest dreams (I = Nikolaj) I would like to have a base class `MatrixLike` from which all other classes inherit. Something like this [5]:

[5] Note that some classes don't exist yet.

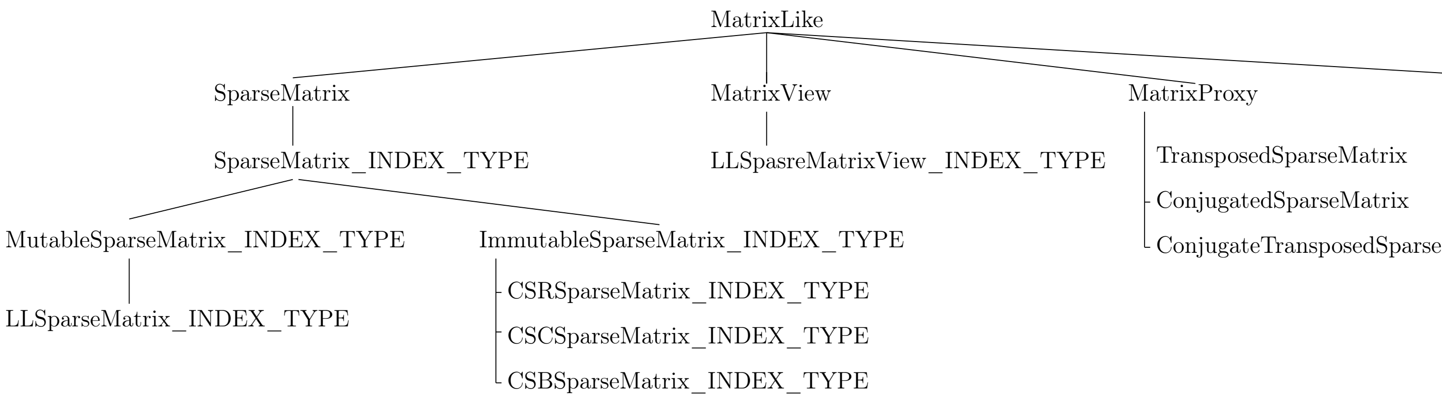
MatrixLike
SparseMatrix
MatrixView
MatrixProxy
SparseMatrix_INDEX_TYPE
LLSpasreMatrixView_INDEX_TYPE
TransposedSparseMatrix
ConjugatedSparseMatrix
ConjugateTransposedSparse
MutableSparseMatrix_INDEX_TYPE
ImmutableSparseMatrix_INDEX_TYPE
LLSparseMatrix_INDEX_TYPE
CSRSparseMatrix_INDEX_TYPE
CSCSparseMatrix_INDEX_TYPE
CSBSparseMatrix_INDEX_TYPE

Fig. 4.1: The ideal (?) class hierarchy

While this makes perfect sense at first, it is not very practical with the current situation of **`Cython`** and its inability to really use *templates*. For instance, the `MatrixLike` class should have `nrow` and `ncol` as attributes but this cannot be done for the moment as both attributes are better typed [6]. Thus, `nrow` and `ncol` must be defined in `SparseMatrix_INDEX_TYPE`, and then again in `LLSparseMatrixView_INDEX_TYPE` and then again... We could define a base `MatrixLike_INDEX_TYPE` class and so on.But the point is that `MatrixLike` would be quite empty. Basically, I tried to keep it simple and without too many inheritance. The resulting class hierarchy is far from optimal (and **is** strange [7]) but is - in my view - a good compromise between code complexity (maintenance), code duplication and ease of use but also Cython's limitations (See [1]).

The current situation is:

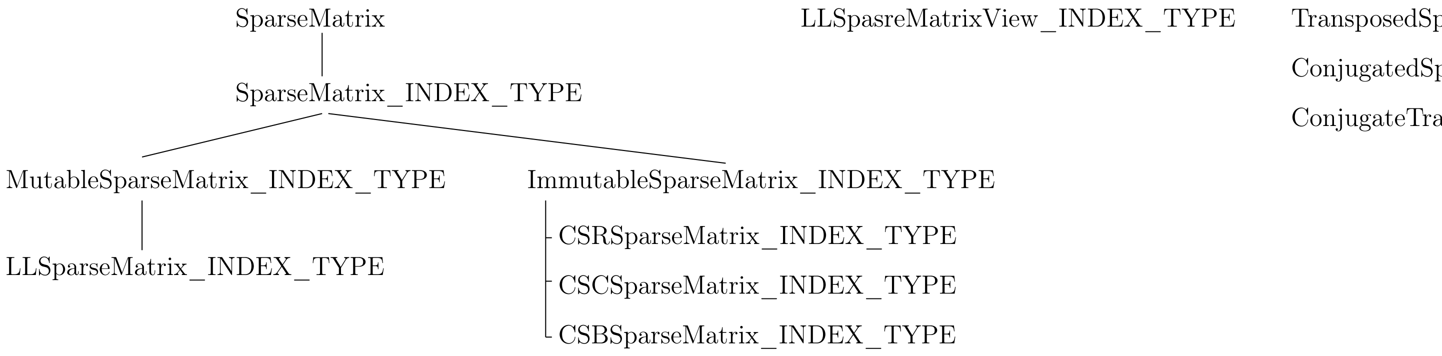
SparseMatrix
LLSpasreMatrixView_INDEX_TYPE
TransposedSp
ConjugatedSp
ConjugateTra
SparseMatrix_INDEX_TYPE
MutableSparseMatrix_INDEX_TYPE
ImmutableSparseMatrix_INDEX_TYPE
LLSparseMatrix_INDEX_TYPE
CSRSparseMatrix_INDEX_TYPE
CSCSparseMatrix_INDEX_TYPE
CSBSparseMatrix_INDEX_TYPE

Fig. 4.2: Current class hierarchy

which involves some code duplication and the use of global functions.

Some pages of this documentation display equations via the jsMath package. They should look reasonably good with most setups but the best rendering is obtained by installing the TeX fonts. Please refer to http://www.math.union.edu/~dpvc/jsMath/users/welcome.html.

[6] Of course, one could argue that we could use non typed attributes in `MatrixLike`.

[7] Especially with the `SparseMatrix` class split in two (`SparseMatrix` and `SparseMatrix_INDEX_TYPE`)), `LLSparseMatrixView_INDEX_TYPE` on its own and non typed proxies.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search