Ceygen Documentation

Release 0.4-pre

Matěj Laitl

May 05, 2014

CONTENTS

CHAPTER

ONE

CEYGEN

1.1 About

Ceygen is a binary Python extension module for linear algebra with Cython typed memoryviews. Ceygen is built atop the Eigen C++ library. Ceygen is **not** a Cython wrapper or an interface to Eigen!

The name Ceygen is a rather poor wordplay on Cython + Eigen; it has nothing to do with software piracy. Ceygen is currently distributed under GNU GPL v2+ license. The authors of Ceygen are however open to other licensing suggestions. (Do you want to use Ceygen in e.g. a BSD-licensed project? Ask!)

Cython is being developed by Matěj Laitl with support from the Institute of Information Theory and Automation, Academy of Sciences of the Czech Republic. Feel free to send me a mail to matej at laitl dot cz.

1.2 Features

Ceygen...

- **is fast** - Ceygen's primary raison d'être is to provide overhead-free algebraic operations for Cython projects that work with typed memoryviews (especially small-sized). For every function there is a code-path where no Python function is called, no memory is allocated on heap and no data is copied. Eigen itself performs rather well, too.
- **is documented** - see Documentation or hop directly to on-line docs.
- **supports various data types** - Ceygen uses Cython fused types (a.k.a. wannabe templates) along with Eigen's template nature to support various data types without duplicating code. While just a few types are pre-defined (float, double, ...), adding a new type is a matter of adding 3 lines and rebuilding Ceygen.
- **is extensively tested** - Ceygen's test suite validates every its public method, including errors raised on invalid input. Thanks to Travis CI, every push is automatically tested against **Python 2.6**, **2.7**, **3.2** and **3.3**.
- **is multithreading-friendly** - Every Ceygen function doesn't acquire the GIL unless it needs to create a Python object (always avoidable); all functions are declared nogil so that you can call them in prange blocks without losing parallelism.
- **provides descriptive error messages** - Care is taken to propagate all errors properly (down from Eigen) so that you are not stuck debugging your program. Ceygen functions don't crash on invalid input but rather raise reasonable errors.
- works well with NumPy, but doesn't depend on it. You don't need NumPy to build or run Ceygen, but thanks to Cython, Cython memoryviews and NumPy arrays are fully interchangeable without copying the data (where it is possible). The test suite currently makes use of NumPy because of our laziness. :-)

On the other hand, Ceygen...

- **depends on Eigen build-time**. Ceygen expects *Eigen 3* headers to be installed under `/usr/lib/eigen3` when it is being built. Installing Eigen is a matter of unpacking it, because it is a pure template library defined solely in the headers. Ceygen doesn't reference Eigen at all at runtime because all code is complited in.
- **still provides a very little subset of Eigen functionality**. We add new functions only as we need them in another projects, but we believe that the hard part is the infrastructure - implementing a new function should be rather straightforward (with decent Cython and C++ knowledge). We're very open to pull requests! (do include unit tests in them)
- **needs recent Cython** (currently at least 0.19.1) to compile. If this is a problem, you can distribute .cpp files or final Python extension module instead.
- **doesn't bring Eigen's elegance to Cython** - if you think of lazy evaluation and advanced expessions, stop dreaming. Ceygen will make your code faster, not nicer. Array expessions will help here.

A simple example to compute matrix product within a big matrix may look like

```
>>> cdef double[:, :] big = np.array([[1.,  2.,   2.,  0.,   0.,  0.],
>>>                                   [3.,  4.,   0., -2.,   0.,  0.]])
>>> ceygen.core.dot_mm(big[:, 0:2], big[:, 2:4], big[:, 4:6])
[[ 2. -4.]
 [ 6. -8.]]
>>> big
[[ 1.  2.   2.  0.   2. -4.]
 [ 3.  4.   0. -2.   6. -8.]],
```

where the dot_mm call above doesn't copy any data, allocates no memory on heap, doesn't need the GIL and uses vectorization (SSE, AltiVec...) to get the best out of your processor.

1.3 Obtaining

Ceygen development happens in its github repository, `git clone git@github.com:strohel/Ceygen.git` -ing is the preferred way to get it as you'll have the latest & greatest version (which shouldn't break thanks to continuous integration). Released versions are available from Ceygen's PyPI page.

1.4 Building

Ceygen uses standard Distutils to build, test and install itself, simply run:

- `python setup.py build` to build Ceygen
- `python setup.py test` to test it (inside build directory)
- `python setup.py install` to install it
- `python setup.py clean` to clean generated object, .cpp and .html files (perhaps to force recompilation)

Commands can be combined, automatically call dependent commands and can take options, the recommended combo to safely install Ceygen is therefore `python setup.py -v test install`.

1.4.1 Building Options

You can set various build options as it is usual with distutils, see `python setup.py --help`. Notable is the *build_ext* command and its *–include-dirs* (standard) and following additional options (whose are Ceygen exten-

sions):

--include-dirs	defaults to */usr/include/eigen3* and must be specified if you've installed Eigen 3 to a non-standard directory,
--cflags	defaults to *-O2 -march=native -fopenmp*. Please note that it is important to enable optimizations and generation of appropriate MMX/SSE/altivec-enabled code as the actual computation code from Eigen is built along with the boilerplate Ceygen code,
--ldflags	additional flags to pass to linker, defaults to *-fopenmp*. Use standard *–libraries* for specifying extra libraries to link against,
--annotate	pass *–annotate* to Cython to produce annotated HTML files during compiling. Only useful during Ceygen development.

You may want to remove *-fopenmp* from *cflags* and *ldflags* if you are already parallelising above Ceygen. The resulting command could look like `python setup.py -v build_ext --include-dirs=/usr/local/include/eigen3 --cflags="-O3 -march=core2" --ldflags= test`. The same could be achieved by putting the options to a *setup.cfg* file:

```
[build_ext]
include_dirs = /usr/local/include/eigen3
cflags = -O3 -march=core2
ldflags =
```

1.5 Documentation

Ceygen documentation is maintained in reStructuredText format under `doc/` directory and can be exported into a variety of formats using Sphinx (version at least 1.0 needed). Just type `make` in that directory to see a list of supported formats and for example `make html` to build HTML pages with the documentation.

See `ChangeLog.rst` file for changes between versions or view it online.

On-line documentation is available at http://strohel.github.com/Ceygen-doc/

1.6 Bugs

Please report any bugs you find and suggestions you may have to Ceygen's github Issue Tracker.

CHAPTER

TWO

CEYGEN CHANGE LOG

This file mentions changes between Ceygen versions that are important for its users. Most recent versions and changes are mentioned on top.

2.1 Changes in 0.4 since 0.3

2.2 Changes in 0.3 since 0.2

- `eigen_version()` function introduced to get Eigen version.
- `llt` module introduced with Cholesky matrix decomposition.
- `dtype` enhanced to provide C char, short, int, long and float types in addition to C double type. `nonint_dtype` introduced for non-integer numeric types. If you get *no suitable method found* or *Invalid use of fused types, type cannot be specialized* Cython errors, specify the specialization explicitly: `ceygen.elemwise.add_vv[double](np.array(...), np.array(...))`. This unfortunately slows down compilation and makes resulting modules bigger, but doesn't affect performance and makes Ceygen more generic.
- `power_vs()` and `power_ms()` functions were added to the `reductions` module.

2.3 Changes in 0.2 since 0.1

- `reductions` module was added with vector, matrix, row-wise and column-wise sums.
- Simple benchmarks for many functions have been added, define `BENCHMARK` or `BENCHMARK_NUMPY` environment variable during test execution to run them; define `SAVE` environment variable to save timings into `.pickle` files that can be visualized by `support/visualize_stats.py`.
- Added code paths optimized for C-contiguous and F-contiguous matrices and vectors using fancy C++ dispatching code. Rougly 40% speed gains in `core.dot_mm()` (for common matrix sizes), 300% gains for `core.dot_mv()` and `core.dot_vm()` starting with 16*16, 30% gains for vector-vector operations and slight gains at other places.
- Internal Ceygen .pxd files (e.g. `eigen_cython.pxd`) are no longer installed.
- `-fopenmp` is now added by default to *build_ext* `cflags` and `ldflags` to enable parellelising `core.dot_mm()` in Eigen; speedups are noticeable for matrices 64*64 and bigger. Can be easily disabled.
- `dtype.vector()` and `dtype.matrix()` convenience functions added; their usage in other modules leads to speedups because it circumvents Cython shortcoming.

- `core.set_is_malloc_allowed()` added to aid in debugging and tests.

CHAPTER

THREE

CORE DATA TYPES AND FUNCTIONS

This module provides basic linear algebra operations such as vector and matrix products as provided by the <Eigen/Core> include.

3.1 Core Data Types

`ceygen.dtype.`**`dtype`**
Cython fused type, a selection of C char, short, int, long, float and double (Python `float`).

`ceygen.dtype.`**`nonint_dtype`**
Cython fused type for methods that cannot work with integer types (such as `inv()`).

`ceygen.dtype.`**`vector`**(*size*, *like*)
Convenience function to create a new vector (*cython.view.array*) and return a memoryview of it. This function is declared *with gil* (it can be called without the GIL held, but acquires it during execution) and is rather expensive (as many Python calls are done).

Parameters

- **size** (*int*) – number of elements of the desired vector
- **like** (`dtype *`) – dummy pointer to desired data type; value not used

Return type `dtype[:]`

`ceygen.dtype.`**`matrix`**(*rows*, *col*, *like*)
Convenience function to create a new matrix (*cython.view.array*) and return a memoryview of it. This function is declared *with gil* (it can be called without the GIL held, but acquires it during execution) and is rather expensive (as many Python calls are done).

Parameters

- **rows** (*int*) – number of rows of the desired matrix
- **cols** (*int*) – number of columns of the desired matrix
- **like** (`dtype *`) – dummy pointer to desired data type; value not used

Return type `dtype[:, :]`

3.2 Linear Algebra Functions

`ceygen.core.`**`dot_vv`**(*x*, *y*)
Vector-vector dot product, returns a scalar of appropriate type.

Parameters

- **x** (`dtype[:]`) – first factor
- **y** (`dtype[:]`) – second factor

Raises `ValueError` if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises `TypeError` if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to `Python array`, Cython array or a `NumPy array`.

Return type `dtype`

`ceygen.core.`**`dot_mv`**(*x*, *y*[, *out=None*])

Matrix-(column) vector product, returns a vector of appropriate type.

Parameters

- **x** (`dtype[:, :]`) – first factor (matrix)
- **y** (`dtype[:]`) – second factor (vector)
- **out** (`dtype[:]`) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get `ValueError`); the same *out* instance will be also returned. **Warning**: don't repeat *x* (or *y*) here, it would give incorrect result without any error. Perhaps there's an in-place variant instead?

Raises `ValueError` if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises `TypeError` if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to `Python array`, Cython array or a `NumPy array`.

Return type `dtype[:]`

`ceygen.core.`**`dot_vm`**(*x*, *y*[, *out=None*])

(Row) vector-matrix product, returns a vector of appropriate type. This is equivalent to dotvm(*y*.T, *x*) because there's no distinction between row and column vectors in Cython memoryviews, but calling this function directly may incur slightly less overhead.

Parameters

- **x** (`dtype[:]`) – first factor (vector)
- **y** (`dtype[:, :]`) – second factor (matrix)
- **out** (`dtype[:]`) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get `ValueError`); the same *out* instance will be also returned. **Warning**: don't repeat *x* (or *y*) here, it would give incorrect result without any error. Perhaps there's an in-place variant instead?

Raises `ValueError` if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises `TypeError` if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to `Python array`, Cython array or a `NumPy array`.

Return type `dtype[:]`

`ceygen.core.`**`dot_mm`**(*x*, *y*[, *out=None*])

Matrix-matrix product, returns a matrix of appropriate type and dimensions. You may of course use this function to multiply matrices that are in fact vectors, you just need to pay attention to column-vector vs. row-vector distinction this time.

If both *x* and *y* are contiguous in some way (either C or Fortran, independently), this function takes optimized code path that doesn't involve memory allocation in Eigen; speed gains are around 40% for matrices around 2*2 – 24*24 size. No special markup is needed to trigger this. See also `set_is_malloc_allowed()`.

Parameters

- **x** (`dtype[:, :]`) – first factor
- **y** (`dtype[:, :]`) – second factor
- **out** (`dtype[:, :]`) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get `ValueError`); the same *out* instance will be also returned. **Warning**: don't repeat *x* (or *y*) here, it would give incorrect result without any error. Perhaps there's an in-place variant instead?

Raises `ValueError` if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises `TypeError` if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to `Python array`, Cython array or a `NumPy array`.

Return type `dtype[:, :]`

3.3 Miscellaneous Functions

`ceygen.core.`**`set_is_malloc_allowed`**(*allowed*)

Set the internal Eigen flag whether it is allowed to allocate memory on heap.

If this flag is `False` and Eigen will try to allocate memory on heap, it will assert which causes `ValueError` to be raised by Ceygen. This is useful to ensure you use the most optimized code path. Defaults to `True`. Note: for this to work, Ceygen defines *EIGEN_RUNTIME_NO_MALLOC* preprocessor directive before including Eigen.

See http://eigen.tuxfamily.org/dox/TopicPreprocessorDirectives.html

`ceygen.core.`**`eigen_version`**()

Return version of Eigen which Ceygen was compiled against as a tuple of three integers, for example (3, 1, 2).

Return type `tuple` of 3 `ints`

CHAPTER

FOUR

ELEMENT-WISE OPERATIONS

This module implements some basic element-wise operations such as addition or division.

Because aliasing is not a problem for element-wise operations, you can make the operations in-place simply by repeating *x* or *y* in *out*. Following examples are therefore valid and produce expected results:

```
ceygen.elemwise.add_mm(x, y, x)
ceygen.elemwise.multiply_vv(a, b, b)
```

Note: This module exists only as a stop-gap until support for element-wise operations with memoryviews are implemented in Cython. It will be phased out once Cython with Mark Florisson's array expressions pull request merged is released.

4.1 Vector-scalar Operations

ceygen.elemwise.**add_vs**(*x*, *y*[, *out=None*])

Add scalar *y* to each coefficient of vector *x* and return the resulting vector.

Note: there's no **subtract_vs**, just add opposite number.

Parameters

- **x** (dtype[:]) – first addend (vector)
- **y** (dtype) – second addend (scalar)
- **out** (dtype[:]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get ValueError); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype[:]

ceygen.elemwise.**multiply_vs**(*x*, *y*[, *out=None*])
Multiply each coefficient of vector *x* by scalar *y* and return the resulting vector.

Note: there's no **divide_vs**, just multiply by inverse number.

Parameters

- **x** (dtype[:]) – first factor (vector)
- **y** (dtype) – second factor (scalar)
- **out** (dtype[:]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get ValueError); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype[:]

ceygen.elemwise.**power_vs**(*x*, *y*[, *out=None*])
Compute *y*-th power of each coefficient of vector *x*.

Parameters

- **x** (dtype[:]) – base (vector)
- **y** (dtype) – exponent (scalar)
- **out** (dtype[:]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get ValueError); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype[:]

4.2 Vector-vector Operations

ceygen.elemwise.**add_vv**(*x*, *y*[, *out=None*])
Vector-vector addition: *x* + *y*

Parameters

- **x** (dtype[:]) – first addend
- **y** (dtype[:]) – second addend

- **out** (dtype[:]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get ValueError); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype[:]

ceygen.elemwise.**subtract_vv**(*x*, *y*[, *out=None*])

Vector-vector subtraction: *x* - *y*

Parameters

- **x** (dtype[:]) – minuend
- **y** (dtype[:]) – subtrahend
- **out** (dtype[:]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get ValueError); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype[:]

ceygen.elemwise.**multiply_vv**(*x*, *y*[, *out=None*])

Vector-vector element-wise multiplication: *x* * *y*

Parameters

- **x** (dtype[:]) – first factor
- **y** (dtype[:]) – second factor
- **out** (dtype[:]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get ValueError); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype[:]

ceygen.elemwise.**divide_vv**(*x*, *y*[, *out=None*])
Vector-vector element-wise division: *x* / *y*

Parameters

- **x** (dtype[:]) – numerator
- **y** (dtype[:]) – denominator
- **out** (dtype[:]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get ValueError); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype[:]

4.3 Matrix-scalar Operations

ceygen.elemwise.**add_ms**(*x*, *y*[, *out=None*])
Add scalar *y* to each coefficient of matrix *x* and return the resulting matrix.

Note: there's no **subtract_ms**, just add opposite number.

Parameters

- **x** (dtype[:, :]) – first addend (matrix)
- **y** (dtype) – second addend (scalar)
- **out** (dtype[:, :]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get ValueError); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype[:, :]

ceygen.elemwise.**multiply_ms**(*x*, *y*[, *out=None*])
Multiply each coefficient of matrix *x* by scalar *y* and return the resulting matrix.

Note: there's no **divide_ms**, just multiply by inverse number.

Parameters

- **x** (dtype[:, :]) – first factor (vector)
- **y** (dtype) – second factor (scalar)
- **out** (dtype[:, :]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get ValueError); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype[:, :]

ceygen.elemwise.**power_ms**(*x*, *y*[, *out=None*])

Compute *y*-th power of each coefficient of matrix *x*.

Parameters

- **x** (dtype[:, :]) – base (matrix)
- **y** (dtype) – exponent (scalar)
- **out** (dtype[:, :]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get ValueError); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype[:, :]

4.4 Matrix-matrix Operations

ceygen.elemwise.**add_mm**(*x*, *y*[, *out=None*])

Matrix-matrix addition: *x* + *y*

Parameters

- **x** (dtype[:, :]) – first addend
- **y** (dtype[:, :]) – second addend
- **out** (dtype[:, :]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you

get `ValueError`); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises `ValueError` if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises `TypeError` if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to `Python array`, Cython array or a `NumPy array`.

Return type `dtype[:, :]`

`ceygen.elemwise.`**`subtract_mm`**(*x*, *y*[, *out=None*])
Matrix-matrix subtraction: *x* - *y*

Parameters

- **x** (`dtype[:, :]`) – minuend
- **y** (`dtype[:, :]`) – subtrahend
- **out** (`dtype[:, :]`) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get `ValueError`); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises `ValueError` if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises `TypeError` if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to `Python array`, Cython array or a `NumPy array`.

Return type `dtype[:, :]`

`ceygen.elemwise.`**`multiply_mm`**(*x*, *y*[, *out=None*])
Matrix-matrix element-wise multiplication: *x* * *y*

Parameters

- **x** (`dtype[:, :]`) – first factor
- **y** (`dtype[:, :]`) – second factor
- **out** (`dtype[:, :]`) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get `ValueError`); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises `ValueError` if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises `TypeError` if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to `Python array`, Cython array or a `NumPy array`.

Return type `dtype[:, :]`

`ceygen.elemwise.`**`divide_mm`**(*x*, *y*[, *out=None*])
Matrix-matrix element-wise division: *x* / *y*

Parameters

- **x** (`dtype[:, :]`) – numerator
- **y** (`dtype[:, :]`) – denominator
- **out** (`dtype[:, :]`) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get `ValueError`); the same *out* instance will be also returned. *As an exception from the general rule*, you **may repeat** *x* (or *y*) here for this element-wise operation.

Raises `ValueError` if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises `TypeError` if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to `Python array`, Cython array or a `NumPy array`.

Return type `dtype[:, :]`

CHAPTER

FIVE

LU DECOMPOSITION-POWERED FUNCTIONS

This module contains algebraic functions powered by the LU matrix decomposition (as provided by the <Eigen/LU> include), most notably matrix inverse and determinant.

`ceygen.lu.`**`inv`**(*x*[, *out*=*None*])

Return matrix inverse computed using LU decomposition with partial pivoting. It is your responsibility to ensure that *x* is invertible, otherwise you get undefined result without any warning.

Parameters

- **x** (`nonint_dtype[:, :]`) – matrix to invert
- **out** (`nonint_dtype[:, :]`) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get `ValueError`); the same *out* instance will be also returned. **Warning**: don't repeat *x* (or *y*) here, it would give incorrect result without any error. Perhaps there's an in-place variant instead?

Raises `ValueError` if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises `TypeError` if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to `Python array`, Cython array or a `NumPy array`.

Return type `nonint_dtype[:, :]`

`ceygen.lu.`**`iinv`**(*x*)

Compte matrix inverse using LU decomposition with partial pivoting in-place. Equivalent to *x* = `inv(x)`, but without overhead. It is your responsibility to ensure that *x* is invertible, otherwise you get undefined result without any warning.

Parameters **x** (`nonint_dtype[:, :]`) – matrix to invert in-place

Raises `ValueError` if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises `TypeError` if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to `Python array`, Cython array or a `NumPy array`.

Returns Always `True` to allow fast exception propagation.

ceygen.lu.**det**(*x*)

Compute determinant of a square matrix *x* using LU decomposition.

Parameters **x** (dtype[:, :]) – matrix whose determimant to compute

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype

CHAPTER

SIX

CHOLESKY DECOMPOSITION-POWERED FUNCTIONS

This module contains algebraic functions powered by the Cholesky matrix decomposition (as provided by the <Eigen/Cholesky> include).

ceygen.llt.**cholesky**(*x*[, *out=None*])

Compute Cholesky decomposition of matrix *x* (which must be square, Hermitian and positive-definite) so that *x* = *out* * *out*.H (*out*.H being conjugate transpose of *out*)

Parameters

- **x** (nonint_dtype[:, :]) – matrix to decompose
- **out** (nonint_dtype[:, :]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get ValueError); the same *out* instance will be also returned. **Warning**: don't repeat *x* (or *y*) here, it would give incorrect result without any error. Perhaps there's an in-place variant instead?

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type nonint_dtype[:, :]

CHAPTER

SEVEN

REDUCTIONS

This module provides various reductions from matrices and vectors to scalars and from matrices to to vectors.

ceygen.reductions.**sum_v**(*x*)
Return sum of the vector *x*.

Parameters **x** (dtype[:]) – vector to sum up

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype

ceygen.reductions.**sum_m**(*x*)
Return sum of the matrix *x*.

Parameters **x** (dtype[:, :]) – matrix to sum up

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises TypeError if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to Python array, Cython array or a NumPy array.

Return type dtype

ceygen.reductions.**rowwise_sum**(*x*[, *out*])
Compute sum of the invidual rows of matrix *x*.

Parameters

- **x** (dtype[:, :]) – matrix to sum up
- **out** (dtype[:]) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get ValueError); the same *out* instance will be also returned. **Warning**: don't repeat *x* (or *y*) here, it would give incorrect result without any error. Perhaps there's an in-place variant instead?

Raises ValueError if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises `TypeError` if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to `Python array`, Cython array or a `NumPy array`.

Return type `dtype[:]`

`ceygen.reductions.`**`colwise_sum`**(*x*[, *out*])

Compute sum of the invidual columns of matrix *x*.

Parameters

- **x** (`dtype[:, :]`) – matrix to sum up
- **out** (`dtype[:]`) – memory view to write the result to. Specifying this optional argument means that Ceygen doesn't have to allocate memory for the result (allocating memory involves acquiring the GIL and calling many expensive Python functions). Once specified, it must must have correct dimensions to store the result of this operation (otherwise you get `ValueError`); the same *out* instance will be also returned. **Warning**: don't repeat *x* (or *y*) here, it would give incorrect result without any error. Perhaps there's an in-place variant instead?

Raises `ValueError` if argument dimensions aren't appropriate for this operation or if arguments are otherwise invalid.

Raises `TypeError` if you pass an argument that doesn't support buffer interface (e.g. a plain list). Use preferrably a Cython memoryview and resort to `Python array`, Cython array or a `NumPy array`.

Return type `dtype[:]`

CHAPTER

EIGHT

CEYGEN DEVELOPMENT

This document should serve as a reminder to me and other possible Ceygen hackers about Ceygen coding style and conventions.

8.1 Development Guidelines

Some special and important files:

- `eigen_cpp.h` - low-level implementation of a tiny C++ Eigen subclass that is used to create wrappers around Cython arrays.
- `eigen_cython.pxd` - exports BaseMap C++ class defined in *eigen_cpp.h* to Cython along with other Eigen methods.
- `dtype.{pxd,pyx}` - defines the base scalar fused (template-like) type that all other functions use, along with functions to create vectors and matrices.
- `dispatch.{h,pxd}` - contains fancy code and Cython declarations for so-called dispatchers: tiny helpers that call more optimized Eigen functions (in fact, the same functions with different template parameters) for column-contiguous, row-contiguous matrices and contiguous vectors.

All other *.{pxd,pyx} are public Ceygen modules.

Please always use appropriate *Dispatcher from *dispatch.pxd* instead of calling methods from *eigen_cython.pxd* directly, because declarations from *eigen_cython.pxd* don't contain `except +` keyword for performance reasons (i.e. you would leak C++ exceptions raised by Eigen code without converting them to Python exceptions).

8.2 Tests and Stress Tests

All public functions should have a unit test. Suppose you have a module `ceygen/modname.pyx`, then unit tests for all functions in `modname.pyx` should go into `ceygen/tests/test_modname.py`. There is a couple of "standard" environment variables recognized in tests:

- `BENCHMARK` - run potentially time-consuming benchmarks of Ceygen code
- `BENCHMARK_NUMPY` - also run some benchmarks with NumPy backend to see difference
- `SAVE` - save timings into `.pickle` files that can be visualized by `support/visualize_stats.py`.

8.3 Releasing Ceygen

Things to do when releasing new version (let it be **X.Y**) of Ceygen:

8.3.1 Before Tagging

1. Set version to **X.Y** in *setup.py* (around line 37)
2. Ensure *ChangeLog.rst* mentions all important changes
3. Ensure that *README.rst* is up-to-date
4. (Optional) update **short description** in *setup.py*
5. (Optional) update **long description** *README.rst*

8.3.2 Tagging & Publishing

1. Do `./setup.py sdist` and check contents, unpack somewhere, run tests incl. benchmarks
2. git tag -s **vX.Y**
3. ./setup.py register sdist upload –sign
4. Build and upload docs: `cd ../ceygen-doc && ./synchronise.sh`
5. If **short description** changed, update it manually at following places:
   - https://github.com/strohel/Ceygen
6. If **long description** changed, update it manually at following places:
   - http://scipy.org/Topical_Software
   - http://www.ohloh.net/p/ceygen

8.3.3 After

1. Set version to **$NEXT_VERSION-pre** in *setup.py*
2. Add header for the next version into *ChangeLog.rst*

CHAPTER

NINE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

C

INDEX