A User-Friendly Hybrid Sparse Matrix Class in C++

Conrad Sanderson, **Ryan R. Curtin**

`http://arma.sourceforge.net/`

July 26, 2018

Introduction

Here's our problem:

The existing landscape of sparse matrix libraries often requires a user to be knowledgeable about sparse matrix storage formats to write efficient code.

Introduction

Here’s our problem:

The existing landscape of sparse matrix libraries often requires a user to be knowledgeable about sparse matrix storage formats to write efficient code.

Here’s our solution:

We provide a new hybrid storage format that automatically (and lazily) converts its internal representation to the best format for a given operation.

Introduction

Outline:

Introduction

Outline:

1. The existing sparse matrix landscape
2. Some sparse matrix representations
3. Our hybrid format approach
4. Simulations and comparisons
5. Conclusion

MATLAB sparse matrix usage

MATLAB has only one sparse matrix format: **compressed sparse column** (CSC).

MATLAB sparse matrix usage

MATLAB has only one sparse matrix format: **compressed sparse column** (CSC).

This means that insertion operations can be very slow:

> Because sparse matrices are stored in compressed sparse column format, there are different costs associated with indexing into a sparse matrix than there are with indexing into a full matrix.

`https://www.mathworks.com/help/matlab/math/accessing-sparse-matrices.html`

MATLAB sparse matrix usage (2)

So, a loop like this can be very inefficient:

```
sp_matrix(1, 1) = 5.0;
for i=2:5000,
  sp_matrix(i - 1, i) = 3.0;
  sp_matrix(i, i) = 5.0;
  sp_matrix(i, i - 1) = 3.0;
end
```

MATLAB sparse matrix usage (2)

So, a loop like this can be very inefficient:

```
sp_matrix(1, 1) = 5.0;
for i=2:5000,
  sp_matrix(i - 1, i) = 3.0;
  sp_matrix(i, i) = 5.0;
  sp_matrix(i, i - 1) = 3.0;
end
```

This means that when using MATLAB with sparse matrices, **some operations have to be written carefully.**

MATLAB sparse matrix usage (2)

So, a loop like this can be very inefficient:

```
sp_matrix(1, 1) = 5.0;
for i=2:5000,
  sp_matrix(i - 1, i) = 3.0;
  sp_matrix(i, i) = 5.0;
  sp_matrix(i, i - 1) = 3.0;
end
```

This means that when using MATLAB with sparse matrices, **some operations have to be written carefully.**

Sparse matrices should instead be created using special code:

```
S = sparse(I, J, SV, M, N)
```

scipy sparse matrix usage

`scipy` implements seven separate sparse matrix formats.

scipy sparse matrix usage

`scipy` implements seven separate sparse matrix formats.

- `bsr_matrix`: block sparse row matrix
- `coo_matrix`: coordinate list matrix
- `csc_matrix`: compressed sparse column matrix
- `csr_matrix`: compressed sparse row matrix
- `dia_matrix`: sparse matrix with diagonal storage
- `dok_matrix`: dictionary-of-keys based matrix *(close to RBT)*
- `lil_matrix`: row-based linked list sparse matrix

Each of these formats is applicable to different use cases, but the user must manually convert between each.

scipy sparse matrix usage (2)

Here is an example program:

```
X = scipy.sparse.rand(1000, 1000, 0.01)

# manually convert to LIL format
# to allow insertion of elements
X = X.tolil()
X[1,1] = 1.23
X[3,4] += 4.56

# random dense vector
V = numpy.random.rand((1000))

# manually convert X to CSC format
# for efficient multiplication
X = X.tocsc()
W = V * X
```

Other libraries

- `SPARSKIT`: contains 16 formats, no automatic conversions
- Eigen: contains only one format (a CSC variant)
- R (`glmnet`, `Matrix`, and `slam`): one format each
- Julia: CSC format only

Even if more than one format is available, the user is responsible for manually converting between formats for the sake of efficiency.

Primary drawbacks

- Each format has its own efficiency and usage drawbacks
- Users must generally manually convert between formats
- Users must understand the efficiency issues related to each format
- Non-expert users can't *just use it*

Coordinate list format

Simple storage of each nonzero point.

```
[[0 2 0 0
  1 0 4 0
  0 0 5 0
  0 3 0 0
  0 0 0 6]]
```

cols	0	1	1	2	2	3
rows	1	0	3	1	2	4
values	1	2	3	4	5	6

Coordinate list format

Simple storage of each nonzero point.

```
[[0 2 0 0
  1 0 4 0
  0 0 5 0
  0 3 0 0
  0 0 0 6]]
```

cols	0	1	1	2	2	3
rows	1	0	3	1	2	4
values	1	2	3	4	5	6

- Insertion: **hard**
- Ordered access: **easy**
- Random access: **medium**
- Programming difficulty: **easy**

Compressed Sparse Column (CSC) format

Storage of each nonzero format with pointers to the start of each column.
Column indices don't need to be stored.

```
[[0 2 0 0
  1 0 4 0
  0 0 5 0
  0 3 0 0
  0 0 0 6]]
```

column offsets	0	1	3	5	6	
row indices	1	0	3	1	2	4
values	1	2	3	4	5	6

Compressed Sparse Column (CSC) format

Storage of each nonzero format with pointers to the start of each column.
Column indices don't need to be stored.

```
[[0 2 0 0
  1 0 4 0
  0 0 5 0
  0 3 0 0
  0 0 0 6]]
```

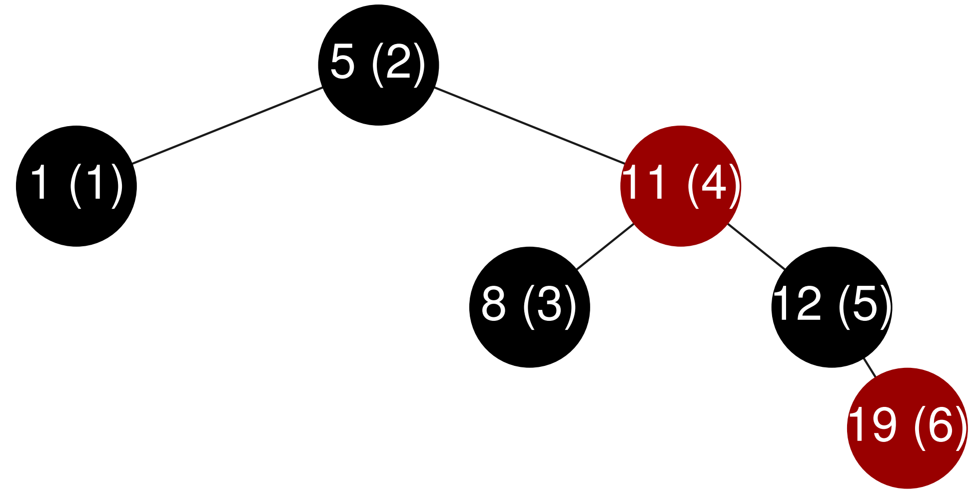
column offsets	0	1	3	5	6	
row indices	1	0	3	1	2	4
values	1	2	3	4	5	6

- Insertion: **hard**
- Ordered access: **easy**
- Random access: **easy**
- Programming difficulty: **hard**

Red-black tree (RBT) format

Store nonzeros in a balanced tree structure for easy insertion.

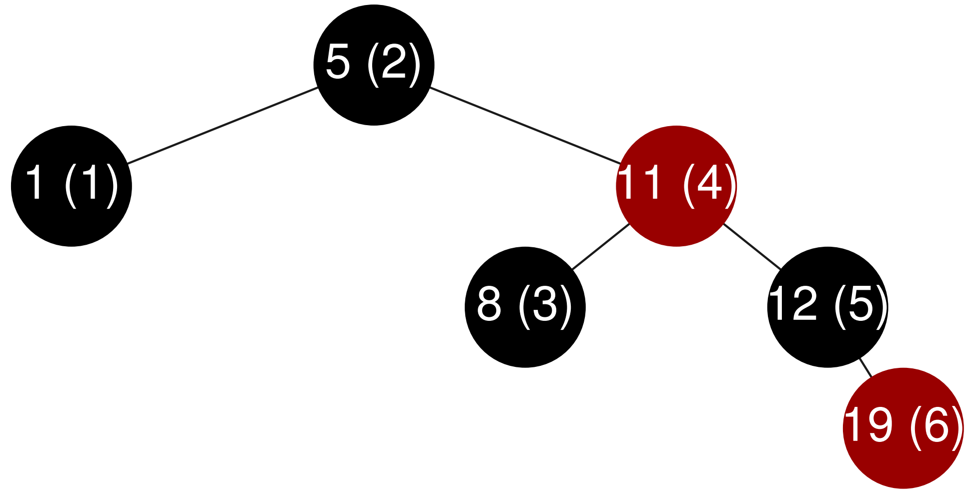
```
[[0 2 0 0
  1 0 4 0
  0 0 5 0
  0 3 0 0
  0 0 0 6]]
```



Red-black tree (RBT) format

Store nonzeros in a balanced tree structure for easy insertion.

```
[[0 2 0 0
  1 0 4 0
  0 0 5 0
  0 3 0 0
  0 0 0 6]]
```



- Insertion: **easy**
- Ordered access: **medium**
- Random access: **medium**
- Programming difficulty: **hard**

Hybrid format

format	insertion	ordered access	random access	difficulty
COO	hard	**easy**	medium	**easy**
CSC	hard	**easy**	**easy**	hard
RBT	**easy**	medium	medium	hard

A hybrid approach can get the best of each world.

Hybrid format

format	insertion	ordered access	random access	difficulty
COO	hard	**easy**	medium	**easy**
CSC	hard	**easy**	**easy**	hard
RBT	**easy**	medium	medium	hard

A hybrid approach can get the best of each world.

- **CSC** for structured operations where access patterns are regular (multiplication, addition, decompositions, etc.).

Hybrid format

format	insertion	ordered access	random access	difficulty
COO	hard	**easy**	medium	**easy**
CSC	hard	**easy**	**easy**	hard
RBT	**easy**	medium	medium	hard

A hybrid approach can get the best of each world.

- **CSC** for structured operations where access patterns are regular (multiplication, addition, decompositions, etc.).
- **RBT** for operations where access patterns are random, irregular, or unknown (insertion, deletion, etc.).

Hybrid format

format	insertion	ordered access	random access	difficulty
COO	hard	**easy**	medium	**easy**
CSC	hard	**easy**	**easy**	hard
RBT	**easy**	medium	medium	hard

A hybrid approach can get the best of each world.

- **CSC** for structured operations where access patterns are regular (multiplication, addition, decompositions, etc.).
- **RBT** for operations where access patterns are random, irregular, or unknown (insertion, deletion, etc.).
- **COO** for *low-programmer-resource* structured operations.

Hybrid format implementation

At all times inside the sparse matrix object we hold the following:

- CSC representation
- RBT representation
- flags indicating if CSC or RBT representations are up to date

Hybrid format implementation

At all times inside the sparse matrix object we hold the following:

- CSC representation
- RBT representation
- flags indicating if CSC or RBT representations are up to date

The representations in the matrix object are allowed to be out of sync!

Hybrid format implementation

At all times inside the sparse matrix object we hold the following:

- CSC representation
- RBT representation
- flags indicating if CSC or RBT representations are up to date

The representations in the matrix object are allowed to be out of sync!

The COO representation is created on-demand from CSC.

Transitions between states

We perform **on-demand** syncing between CSC and RBT.

Transitions between states

We perform **on-demand** syncing between CSC and RBT.

- **CSC operation:** we first ensure that our CSC representation is the most up-to-date.

Transitions between states

We perform **on-demand** syncing between CSC and RBT.

- **CSC operation:** we first ensure that our CSC representation is the most up-to-date. **If not we sync it.**

Transitions between states

We perform **on-demand** syncing between CSC and RBT.

- **CSC operation:** we first ensure that our CSC representation is the most up-to-date. **If not we sync it.**
- **RBT operation:** we first ensure that our RBT representation is the most up-to-date. **If not we sync it.**

Transitions between states

We perform **on-demand** syncing between CSC and RBT.

- **CSC operation:** we first ensure that our CSC representation is the most up-to-date. **If not we sync it.**
- **RBT operation:** we first ensure that our RBT representation is the most up-to-date. **If not we sync it.**
- **COO operation:** we extract a COO representation on-demand.

Transitions between states

We perform **on-demand** syncing between CSC and RBT.

- **CSC operation:** we first ensure that our CSC representation is the most up-to-date. **If not we sync it.**
- **RBT operation:** we first ensure that our RBT representation is the most up-to-date. **If not we sync it.**
- **COO operation:** we extract a COO representation on-demand.

All of this syncing is handled automatically and is hidden from the user.

Extra optimizations with template metaprogramming

The C++ language allows us to collect the details of an operation as its type.

Extra optimizations with template metaprogramming

The C++ language allows us to collect the details of an operation as its type.

- `A.t().t()` $\rightarrow$ we can do no computation at all

Extra optimizations with template metaprogramming

The C++ language allows us to collect the details of an operation as its type.

- `A.t().t()` $\rightarrow$ we can do no computation at all
- `trace(A.t() * B)` $\rightarrow$ we can avoid the transpose and multiplication

Extra optimizations with template metaprogramming

The C++ language allows us to collect the details of an operation as its type.

- `A.t().t()` $\rightarrow$ we can do no computation at all
- `trace(A.t() * B)` $\rightarrow$ we can avoid the transpose and multiplication
- `C = 2 * (A + B)` $\rightarrow$ we can avoid generating a temporary for `A + B`

Extra optimizations with template metaprogramming

The C++ language allows us to collect the details of an operation as its type.

- `A.t().t()` → we can do no computation at all
- `trace(A.t() * B)` → we can avoid the transpose and multiplication
- `C = 2 * (A + B)` → we can avoid generating a temporary for `A + B`

This also allows us to skip format syncing when it isn't necessary.

Extra optimizations with template metaprogramming

The C++ language allows us to collect the details of an operation as its type.

- `A.t().t()` $\to$ we can do no computation at all
- `trace(A.t() * B)` $\to$ we can avoid the transpose and multiplication
- `C = 2 * (A + B)` $\to$ we can avoid generating a temporary for `A + B`

This also allows us to skip format syncing when it isn't necessary.
(These optimizations also apply to dense matrices in Armadillo.)

C. Sanderson. *Armadillo: An Open-Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments.* Technical report, NICTA, 2010.

C. Sanderson, R.R. Curtin. *Armadillo: C++ template metaprogramming for compile-time optimization of linear algebra.* PASC 2017.

API comparison

```
X = scipy.sparse.rand(1000, 1000, 0.01)

# manually convert to LIL format
# to allow insertion of elements
X = X.tolil()
X[1,1]  = 1.23
X[3,4] += 4.56

# random dense vector
V = numpy.random.rand((1000))

# manually convert X to CSC format
# for efficient multiplication
X = X.tocsc()
W = V * X
```

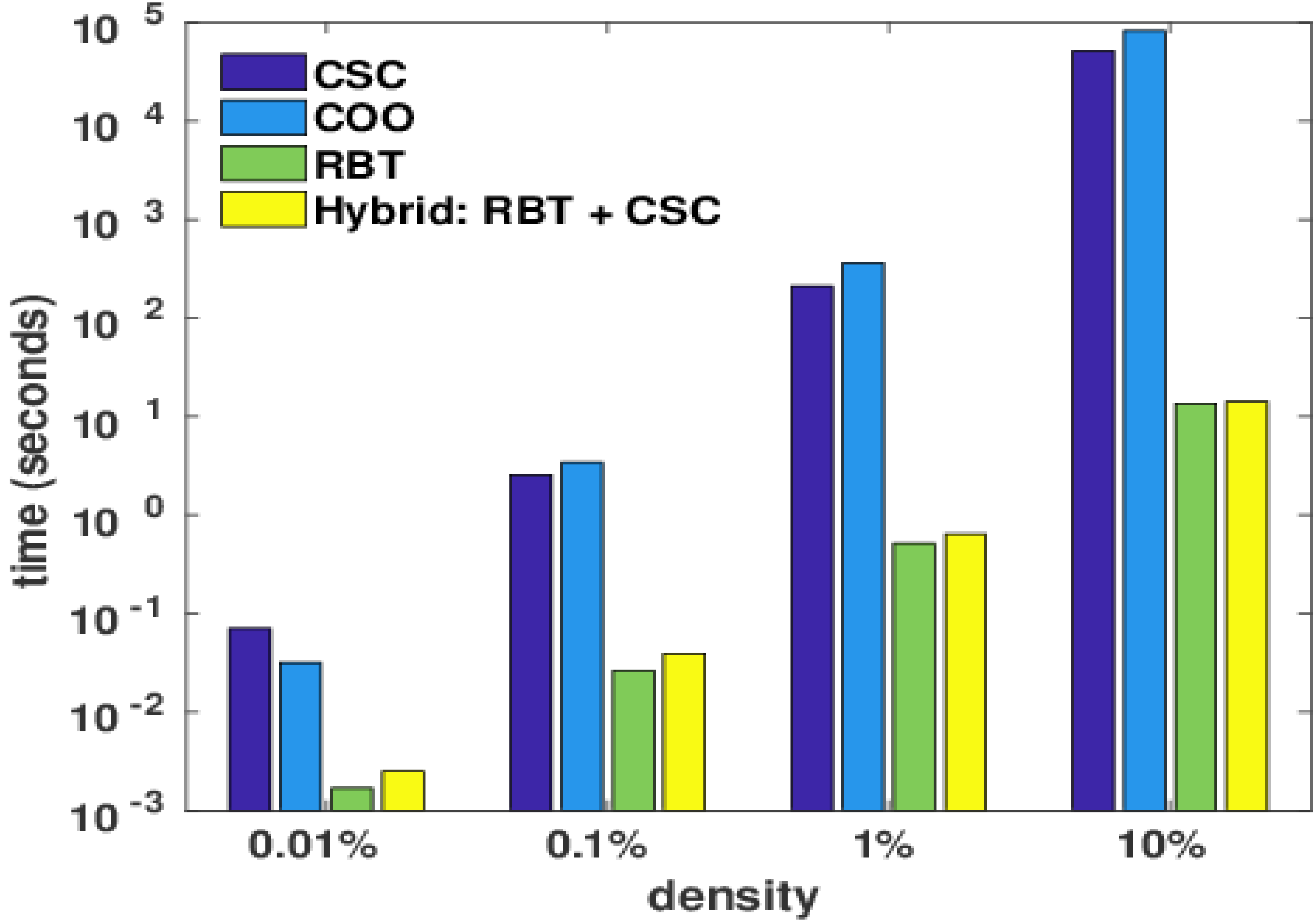
API comparison

```python
X = scipy.sparse.rand(1000, 1000, 0.01)

# manually convert to LIL format
# to allow insertion of elements
X = X.tolil()
X[1,1]  = 1.23
X[3,4] += 4.56

# random dense vector
V = numpy.random.rand((1000))

# manually convert X to CSC format
# for efficient multiplication
X = X.tocsc()
W = V * X
```

```cpp
sp_mat X = sprandu(1000, 1000, 0.01);

// automatic conversion to RBT format
// for fast insertion of elements

X(1,1)  = 1.23;
X(3,4) += 4.56;

// random dense vector
rowvec V(1000, fill::randu);

// automatic conversion of X to CSC
// prior to multiplication

rowvec W = V * X;
```

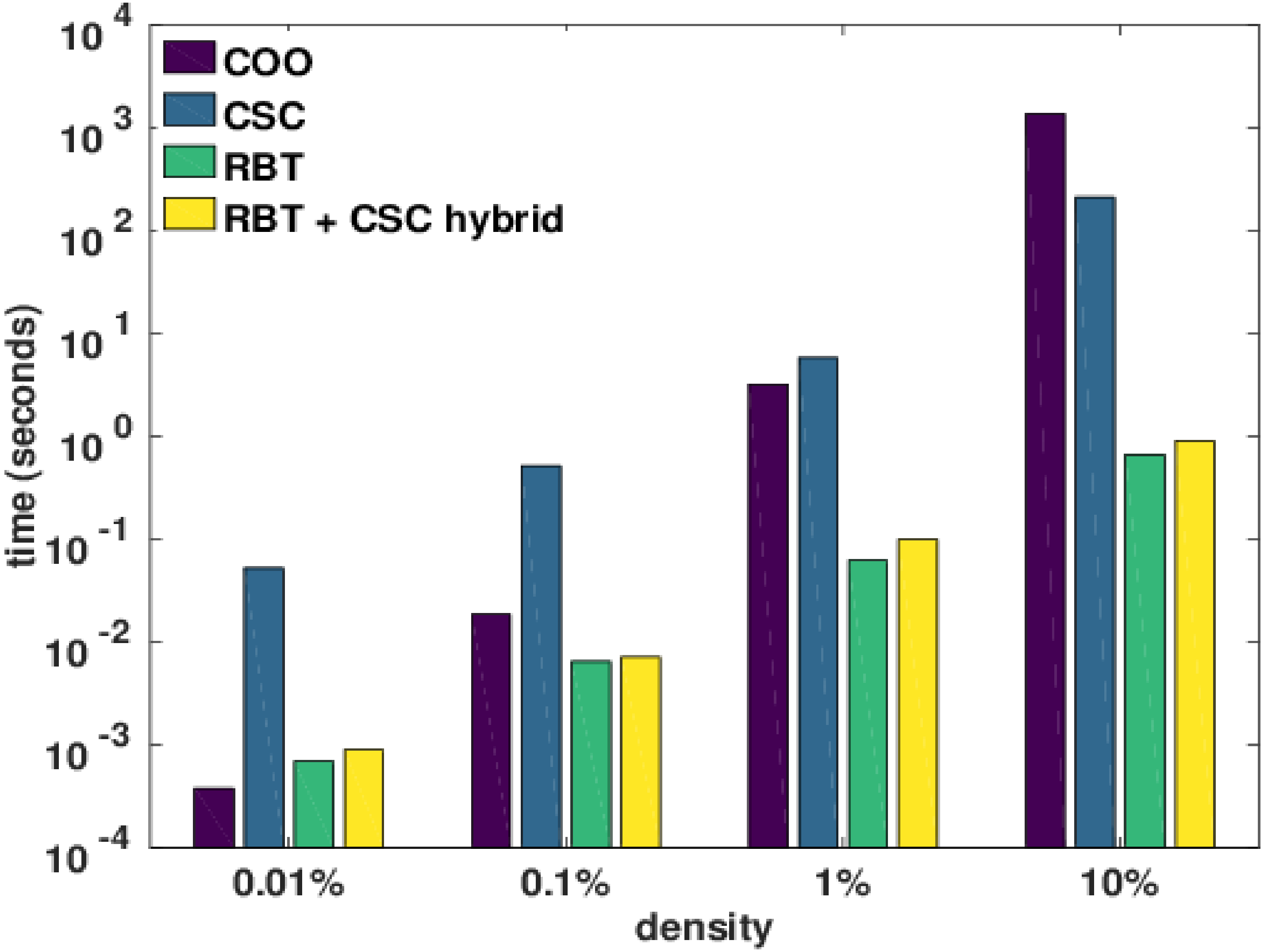
Random element insertion

10k x 10k sparse matrix. Time is to fill random locations to desired density.



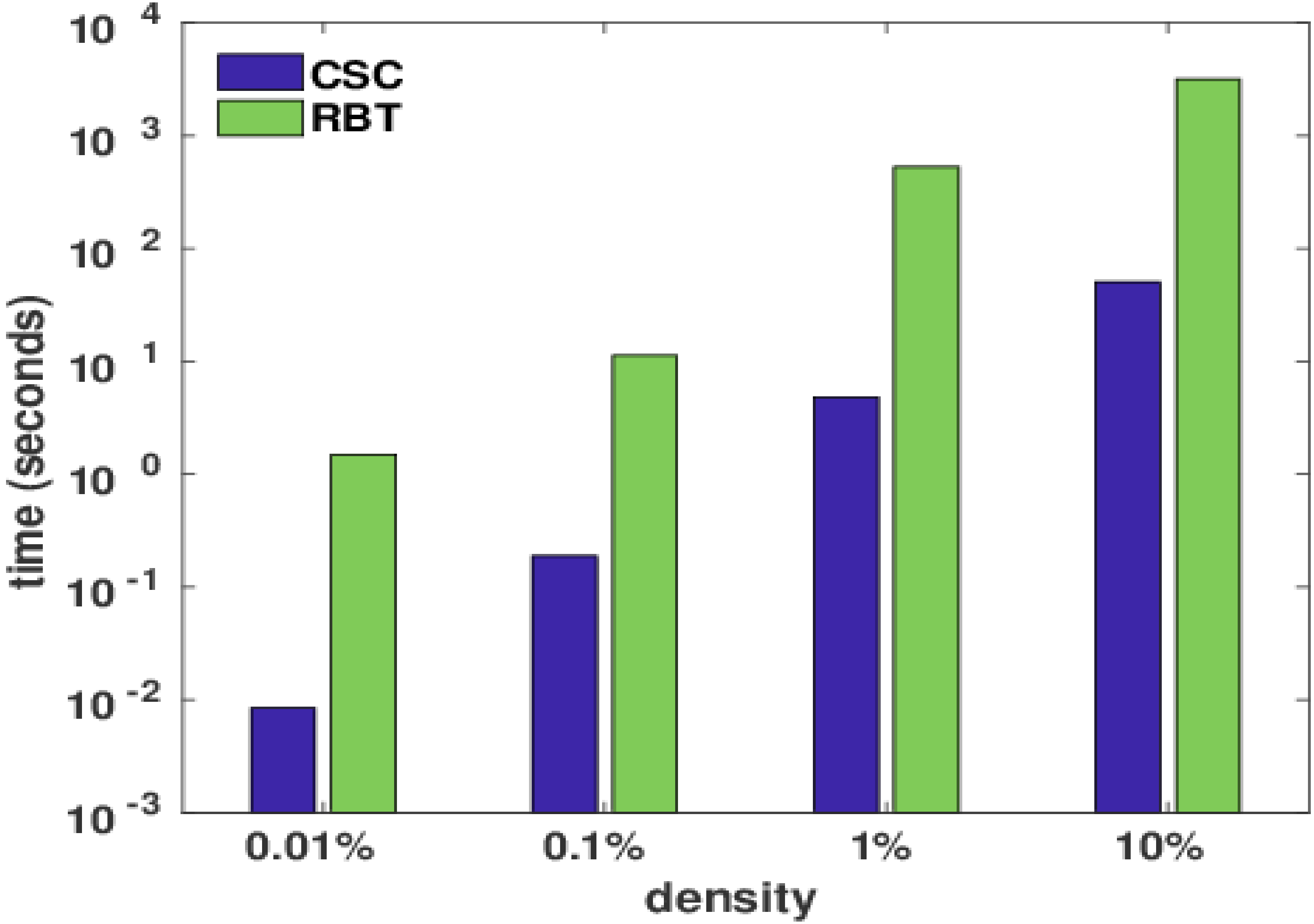
Ordered element insertion

10k x 10k sparse matrix. Time is to fill ordered random location to desired density.



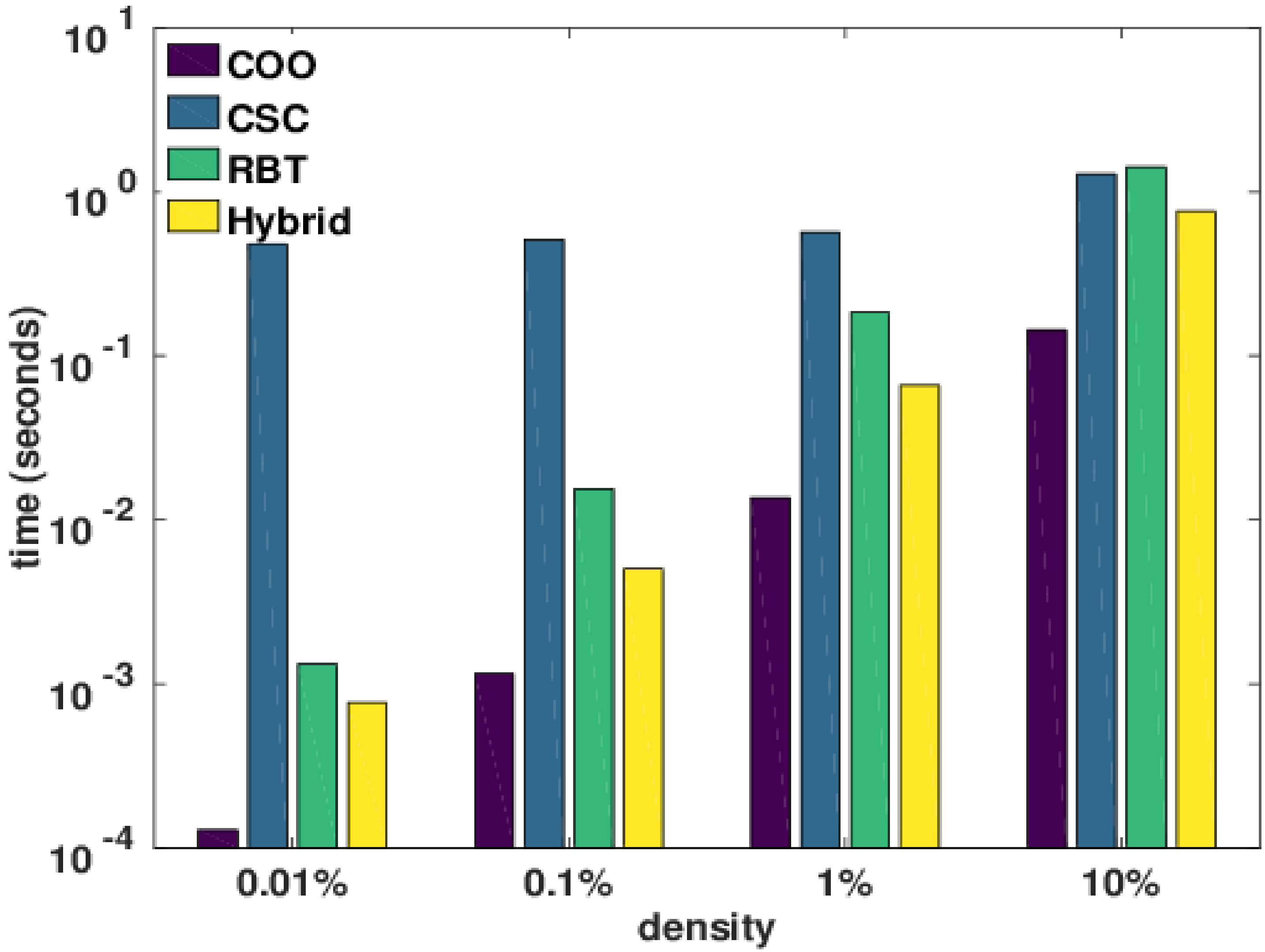
Multiplication

10k x 10k random sparse matrix. Time is for `A*B` operation.



repmat()

Time is to replicate 1k x 1k random sparse matrix into a 10k x 10k sparse matrix.



Conclusions

- Sparse matrix implementations are not very user friendly, because they often require the user to know details about internal storage.
- The CSC, COO, and RBT format provide good performance for the vast majority of use cases.
- We have created a hybrid format that can use whichever of these is best for the given task.
- The hybrid format performs automatic on-demand conversion between internal storage formats; the overhead is minimal.
- Use of this hybrid format means easy code for users.
- This is all available in Armadillo (`http://arma.sourceforge.net/`) as the `arma::sp_mat` class!
- Usable in R via **RcppArmadillo** and used in other higher-level libraries such as mlpack (machine learning) and others.

Questions and comments?