# The LARC (Linear Algebra via Recursive Compression) Package and the MyPyLARC Tutorial Package

Steve Cuccaro (IDA/CCS)
Mark Pleszkoch (IDA/CCS)
Jenny Zito (IDA/CCS)

March 2020

## Contents

## 1 Introduction

LARC is a software package developed at IDA/CCS that stores matrices in a compressed format and performs operations on matrices and vectors while they are compressed. LARC assigns each scalar, vector, and matrix a unique *MatrixID*. These MatrixIDs are used with two hash tables, the *MatrixStore* and the *OperationStore*, to ensure that we never store the same matrix twice or carry out the same operation twice. We have developed several new techniques using a locality-sensitive scalar hash which allow LARC to reduce numerical precision issues and to preserve selected math identities. LARC is optimized for use on matrices with power-of-2 dimensions and is most effective when there is submatrix reuse, which is common in recursive matrices, self-similar matrices, matrices with tensor structure, and matrices with a limited number of distinct scalars (e.g., sparse matrices).

It only makes sense to use the LARC package if the application is expected to handle matrices that have some repeated internal structure. When repeated structure exists, LARC will store very large matrices and carry out matrix operations impractical in standard compressed or uncompressed formats. Otherwise, the application will not benefit from LARC but will only incur the cost of matrix storage and operation memoization overhead.

Most of the LARC computations are carried out in C, but there is a user-friendly SWIG-generated Python wrapper. MyPyLARC is a tutorial and example package which makes use of the Python interface and provides templates for users to write their own package using LARC to perform matrix math. Both LARC and MyPyLARC are available via GitHub at https://github.com/LARCmath/LARC.

## 2 The LARC Matrix-Math Package

### 2.1 Quadtrees and Matrix Computation

For large matrices, matrix operations consume a large amount of computational time and memory. There is a long history of clever methods to speed calculations by dividing matrices into quarters recursively in order to carry out computation, for example:

- recursive block matrix methods for Cooley-Tukey discrete Fourier transforms (see Van Loan [7, pp. 20–22]);
- Strassen's matrix multiplication algorithm [6];
- quadtree decomposition to implement parallel matrix operations such as multiplication, division and trace by David Wise and colleagues [1, 8, 9, 10, 11].

To represent matrices and accomplish recursive matrix operations, LARC makes use of the technique of *quadtree* decomposition, where matrices are recursively divided into four quadrant submatrices. In Figure 1, we see an example of such a division, iterated until we finally reach 1×1 matrices containing single scalars.

$$\left[\begin{array}{cc|cc} 0 & 6 & 2 & 1 \\ 0 & 1 & 1 & 0 \\ \hline 5 & 2 & 4 & 1 \\ 3 & 0 & 2 & 1 \end{array}\right]$$

$$\left[\begin{array}{c|c} 0 & 6 \\ \hline 0 & 1 \end{array}\right] \qquad \left[\begin{array}{c|c} 2 & 1 \\ \hline 1 & 0 \end{array}\right] \qquad \left[\begin{array}{c|c} 5 & 2 \\ \hline 3 & 0 \end{array}\right] \qquad \left[\begin{array}{c|c} 4 & 1 \\ \hline 2 & 1 \end{array}\right]$$

$$[0]\ [6]\ [0]\ [1] \qquad [2]\ [1]\ [1]\ [0] \qquad [5]\ [2]\ [3]\ [0] \qquad [4]\ [1]\ [2]\ [1]$$

Figure 1: For LARC, matrix operations and storage depend on the recursive subdivision of each matrix into four quadrant submatrices.

For the historical work listed above, there was no reason to expect that many of the submatrices that show up in the quadtree decomposition would be repeated. However, LARC was designed to work for matrices in which there is a great deal of repetition of submatrices in the quadtree. We will see in the next section how LARC takes advantage of this repetition.

## 2.2 Matrix Storage via Recursive Representation

Compressed storage in LARC of a $2^m \times 2^n$ matrix depends on recursively dividing the matrix into four quadrant submatrices or, in the case of vectors, into two halves, until we get down to the scalar elements of the matrix. We refer to $\max(n, m)$ as the *level* of a LARC matrix, and a level-$k$ matrix is composed of quadrants of level $(k-1)$. In LARC, all storage and operations depend on this recursive subdivision.

Let us examine the concepts used in LARC via a simple example. Consider the matrix $M$ (from Figure 1), which we divide into four quadrant submatrices:

$$M = \begin{bmatrix} 0 & 6 & 2 & 1 \\ 0 & 1 & 1 & 0 \\ 5 & 2 & 4 & 1 \\ 3 & 0 & 2 & 1 \end{bmatrix} \longrightarrow \left[\begin{array}{cc|cc} 0 & 6 & 2 & 1 \\ 0 & 1 & 1 & 0 \\ \hline 5 & 2 & 4 & 1 \\ 3 & 0 & 2 & 1 \end{array}\right]$$

Let us call the submatrices $A$, $B$, $C$, and $D$:

$$A = \begin{bmatrix} 0 & 6 \\ 0 & 1 \end{bmatrix}, \; B = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}, \; C = \begin{bmatrix} 5 & 2 \\ 3 & 0 \end{bmatrix}, \; D = \begin{bmatrix} 4 & 1 \\ 2 & 1 \end{bmatrix}.$$

In LARC, each scalar, vector, and matrix is assigned a unique MatrixID and given a *MatrixRecord* that is stored in the MatrixStore hash table. Inside the MatrixRecord, a matrix is represented recursively by a list of the MatrixIDs from its four quadrant submatrices; we call this list its *MatrixValue*. If the matrix is $1 \times 1$, then its MatrixValue is simply the scalar itself. For example, the matrix $M$ above would be described recursively by

$$[\mathtt{ID}_A, \mathtt{ID}_B, \mathtt{ID}_C, \mathtt{ID}_D],$$

that consists of the MatrixIDs from its four quadrant submatrices $A, B, C, D$. This list defines the matrix in a concise representation. Each MatrixRecord includes the MatrixID, MatrixValue (recursive definition), and the level of the matrix. By recursively drilling down through the MatrixRecords for each quadrant submatrix, we have a complete description of the contents of the matrix.

For non-scalar matrices, each MatrixRecord in the MatrixStore is indexed by a hash of its four quadrant submatrix MatrixIDs from its recursive definition. In order to not store the same matrix twice, we use the MatrixStore hash table to confirm whether a matrix is new or has already been stored. We will see in Section 2.3 that all matrix operations will be carried out recursively in a way that produces the four MatrixIDs of the quadrant submatrices of the output matrix. Therefore, before creating a new MatrixRecord for the output, we hash the list of four MatrixIDs and look in the MatrixStore to see if there is already an existing matrix composed of those four submatrices, and if so return its MatrixID.

Scalars in the MatrixStore ($1 \times 1$ matrices) are handled slightly differently. We use a special locality-sensitive hash of the scalar value to index into the MatrixStore, and instead of checking to see if the scalar itself has already been stored, we determine whether a scalar which is 'close enough' to the desired scalar has already been stored. We explain how this is done in Section 2.5 on locality-sensitive hashing.

The recursive representation of matrices leads to compressed storage if there is repetition of submatrices. Consider these two matrices:

$$R = \left[\begin{array}{cccc|cccc} .8 & .8 & 7 & 7 & 7 & 7 & .8 & .8 \\ .8 & \text{-}2 & 7 & 7 & 7 & 7 & .8 & \text{-}2 \\ 7 & 7 & .8 & .8 & .8 & .8 & 7 & 7 \\ 7 & 7 & .8 & \text{-}2 & .8 & \text{-}2 & 7 & 7 \\ \hline 7 & 7 & 7 & 7 & 7 & 7 & .8 & .8 \\ 7 & 7 & 7 & 7 & 7 & 7 & .8 & \text{-}2 \\ 7 & 7 & 7 & 7 & .8 & .8 & 7 & 7 \\ 7 & 7 & 7 & 7 & .8 & \text{-}2 & 7 & 7 \end{array}\right] \quad S = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 7 & 7 & .8 & .8 \\ 0 & 1 & 0 & 0 & 7 & 7 & .8 & \text{-}2 \\ 0 & 0 & 1 & 0 & .8 & .8 & 7 & 7 \\ 0 & 0 & 0 & 1 & .8 & \text{-}2 & 7 & 7 \\ \hline .8 & .8 & 7 & 7 & 7 & 7 & 7 & 7 \\ .8 & \text{-}2 & 7 & 7 & 7 & 7 & 7 & 7 \\ 7 & 7 & .8 & .8 & 7 & 7 & 7 & 7 \\ 7 & 7 & .8 & \text{-}2 & 7 & 7 & 7 & 7 \end{array}\right] \tag{1}$$

We have chosen this example so that the number of unique quadrant submatrices is small. The MatrixIDs are assigned to the submatrices in depth-first order. Since we only save each matrix once, if we are building a new matrix and it contains a submatrix we have already seen, then we use the previously assigned MatrixID for that submatrix.

If we store $R$ first, we will assign the MatrixIDs for our example in the following order, starting from the $1{\times}1$ matrix in the upper-left corner of matrix $R$:

$$A = \left[\ .8\ \right],\ B = \left[\ \text{-}2\ \right],\ C = \begin{bmatrix} .8 & .8 \\ .8 & \text{-}2 \end{bmatrix},\ D = \left[\ 7\ \right],\ E = \begin{bmatrix} 7 & 7 \\ 7 & 7 \end{bmatrix},$$

$$F = \begin{bmatrix} .8 & .8 & 7 & 7 \\ .8 & \text{-}2 & 7 & 7 \\ 7 & 7 & .8 & .8 \\ 7 & 7 & .8 & \text{-}2 \end{bmatrix},\ G = \begin{bmatrix} 7 & 7 & .8 & .8 \\ 7 & 7 & .8 & \text{-}2 \\ .8 & .8 & 7 & 7 \\ .8 & \text{-}2 & 7 & 7 \end{bmatrix},\ H = \begin{bmatrix} 7 & 7 & 7 & 7 \\ 7 & 7 & 7 & 7 \\ 7 & 7 & 7 & 7 \\ 7 & 7 & 7 & 7 \end{bmatrix},\ R, \tag{2}$$

$$I = \left[\ 1\ \right],\ J = \left[\ 0\ \right],\ K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ L = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix},\ M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},\ \text{ and } S.$$

We see that there are a total of 15 matrices to which we have assigned MatrixIDs. It takes nine MatrixRecords to store the matrix and unique quadrant submatrices that determine $R$ recursively. When we store $S$, we only need an additional six MatrixRecords because we have already stored some of the quadrant submatrices of $S$ by storing $R$. In contrast, if these submatrices had not been stored in advance, then storing $S$ would require 14 MatrixRecords. In general, if we fill a $2^n \times 2^n$ matrix with distinct scalars, our quadtree representation will have one distinct $2^n \times 2^n$ matrix, four $(2^{n-1} \times 2^{n-1})$ matrices, sixteen $(2^{n-2} \times 2^{n-2})$ matrices, and so on, down to $2^{2n}$ $1{\times}1$ matrices (each holding a single scalar), giving a total of

$$1 + 4 + 4^2 + \cdots + 4^{n-1} + 4^n = \frac{4^{n+1} - 1}{3}$$

MatrixRecords. Thus, two $8{\times}8$ matrices with no scalars in common require 170 MatrixRecords, versus the 15 MatrixRecords required to store the highly compressible $R$ and $S$ matrices of our example.

In Figure 2, we give a table showing how storage works in LARC for the example matrices (1) and their quadrant submatrices (2.2). The table has a line per MatrixRecord, and each appears in the order that it was created and stored.

Matrices created in a recursive way, such as the Hadamard matrices, are extremely compressible. The Hadamard matrices are defined recursively by

$$H_1 = \left[\begin{array}{c|c} 1 & 1 \\ \hline 1 & \text{-}1 \end{array}\right], \qquad H_{n+1} = \left[\begin{array}{c|c} H_n & H_n \\ \hline H_n & -H_n \end{array}\right].$$

**MatrixRecords**

| **MatrixID** | **MatrixValue** | **Level** | |
|---|---|---|---|
| $\mathtt{ID}_A$ | .8 | 0 | (1×1 matrix) |
| $\mathtt{ID}_B$ | -2 | 0 | |
| $\mathtt{ID}_C$ | ($\mathtt{ID}_A$,$\mathtt{ID}_A$,$\mathtt{ID}_A$,$\mathtt{ID}_B$) | 1 | (2×2 matrix) |
| $\mathtt{ID}_D$ | 7 | 0 | |
| $\mathtt{ID}_E$ | ($\mathtt{ID}_D$,$\mathtt{ID}_D$,$\mathtt{ID}_D$,$\mathtt{ID}_D$) | 1 | |
| $\mathtt{ID}_F$ | ($\mathtt{ID}_C$,$\mathtt{ID}_E$,$\mathtt{ID}_E$,$\mathtt{ID}_C$) | 2 | (4×4 matrix) |
| $\mathtt{ID}_G$ | ($\mathtt{ID}_E$,$\mathtt{ID}_C$,$\mathtt{ID}_C$,$\mathtt{ID}_E$) | 2 | |
| $\mathtt{ID}_H$ | ($\mathtt{ID}_E$,$\mathtt{ID}_E$,$\mathtt{ID}_E$,$\mathtt{ID}_E$) | 2 | |
| $\mathtt{ID}_R$ | ($\mathtt{ID}_F$,$\mathtt{ID}_G$,$\mathtt{ID}_H$,$\mathtt{ID}_G$) | 3 | (8×8 matrix) |
| $\mathtt{ID}_I$ | 1 | 0 | |
| $\mathtt{ID}_J$ | 0 | 0 | |
| $\mathtt{ID}_K$ | ($\mathtt{ID}_I$,$\mathtt{ID}_J$,$\mathtt{ID}_J$,$\mathtt{ID}_I$) | 1 | |
| $\mathtt{ID}_L$ | ($\mathtt{ID}_J$,$\mathtt{ID}_J$,$\mathtt{ID}_J$,$\mathtt{ID}_J$) | 1 | |
| $\mathtt{ID}_M$ | ($\mathtt{ID}_K$,$\mathtt{ID}_L$,$\mathtt{ID}_L$,$\mathtt{ID}_K$) | 2 | |
| $\mathtt{ID}_S$ | ($\mathtt{ID}_M$,$\mathtt{ID}_G$,$\mathtt{ID}_F$,$\mathtt{ID}_H$) | 3 | |

Figure 2: Matrix storage for the matrices from (1) creates a MatrixRecord for each unique matrix and quadrant submatrix.

A $2^n \times 2^n$ Hadamard matrix takes only $2n + 1$ MatrixRecords to store recursively in LARC. Figure 3 shows the $8 \times 8$ Hadamard matrix $H_3$ and its compressed representation, which consists of only seven MatrixRecords. The Hadamard matrix $H_{100}$ is $2^{100} \times 2^{100}$ and requires only 201 records to store in LARC.

While our examples show what happens with square matrices, we can easily extend this process to nonsquare matrices of size $2^m \times 2^n$, including vectors (where either $m$ or $n$ is equal to 0). When we recursively divide vectors, we divide them in half, rather than into quarters. In our LARC code we keep track of both the row level $m$ and column level $n$. LARC's internal representation of vectors is similar to that of matrices, except that two of the MatrixIDs are set to a `NULL` value. Specifically, we store ($\mathtt{ID}_A$, `NULL`, $\mathtt{ID}_B$, `NULL`) for column vectors where $A$ and $B$ are half-sized column vectors and ($\mathtt{ID}_C$, $\mathtt{ID}_D$, `NULL`, `NULL`) for row vectors where $C$ and $D$ are half-sized row vectors.

$$\left[\begin{array}{cccc|cccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\
\hline
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\
1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\
1 & -1 & -1 & 1 & -1 & 1 & 1 & -1
\end{array}\right]$$

$H_3$ is 8×8 and requires seven records.

**MatrixRecords**

| **Matrix ID** | **Matrix Value** | **Is the Matrix** |
|---|---|---|
| $A$ | 1 | |
| $B$ | $-1$ | |
| $C$ | $(A,A,A,B)$ | $H_1$ |
| $D$ | $(B,B,B,A)$ | $-H_1$ |
| $E$ | $(C,C,C,D)$ | $H_2$ |
| $F$ | $(D,D,D,C)$ | $-H_2$ |
| $G$ | $(E,E,E,F)$ | $H_3$ |

Figure 3: How the $H_3$ Hadamard matrix is stored in LARC.

## 2.3 Matrix Operations

LARC can carry out linear algebra operations on compressed matrices (leaving them in the compressed format), as long as the algebraic operation can be described recursively in terms of quadrant submatrices. LARC currently contains several basic unitary and binary operations including: matrix addition, matrix-matrix multiplication, scalar-matrix multiplication, Kronecker product, adjoint (complex conjugate transpose), and some matrix norms.

We define block-recursive algorithms for addition and multiplication, compatible with our compressed matrix storage format, as follows. Given matrices $A$ and $B$ in terms of their quadrants,

$$A = \left[\begin{array}{c|c} A_0 & A_1 \\ \hline A_2 & A_3 \end{array}\right] \quad \text{and} \quad B = \left[\begin{array}{c|c} B_0 & B_1 \\ \hline B_2 & B_3 \end{array}\right],$$

the result $C$ of adding $A$ and $B$ has quadrant submatrices expressed as

$$C = A + B = \left[\begin{array}{c|c} (A_0 + B_0) & (A_1 + B_1) \\ \hline (A_2 + B_2) & (A_3 + B_3) \end{array}\right],$$

and the result $D$ of multiplying $A$ and $B$ has quadrant submatrices expressed as

$$D = A \times B = \left[\begin{array}{c|c} (A_0 \times B_0) + (A_1 \times B_2) & (A_0 \times B_1) + (A_1 \times B_3) \\ \hline (A_2 \times B_0) + (A_3 \times B_2) & (A_2 \times B_1) + (A_3 \times B_3) \end{array}\right].$$

Vector operations are simply special cases of matrix operations. For example, multiplication of a row vector and a matrix is defined as follows. Given a row vector $A$ and a matrix $B$ in terms of their quadrants,

$$A = \left[ \begin{array}{c|c} A_0 & A_1 \end{array} \right] \text{ and } B = \left[ \begin{array}{c|c} B_0 & B_1 \\ \hline B_2 & B_3 \end{array} \right],$$

the result $D$ of multiplying $A$ and $B$ has two "quadrant" submatrices expressed as

$$D = A \times B = \left[ \begin{array}{c|c} (A_0 \times B_0) + (A_1 \times B_2) & (A_0 \times B_1) + (A_1 \times B_3) \end{array} \right].$$

Operations are memoized and saved as an *OperationRecord* in the OperationStore. Just as with the MatrixStore, LARC implements this as a hash table. The OperationStore hash function uses the type of operation and the MatrixIDs of the input matrices.

LARC takes advantage of mathematical identities in several ways. MatrixRecords of identity matrices and all zero matrices contain flags marking them as such. Then when a matrix $M$ is added to a zero matrix, or multiplied by an identity matrix, the MatrixID of $M$ is returned without any computational work. These zero and identity flags are used for initial checks in many LARC operations, e.g. adjoint of these matrices is trivial. Also, since matrix addition is commutative, we use the trick of sorting the MatrixIDs of the inputs before including them in the hash computation for the OperationStore. Thus having memoized the operation $A + B$ serves to memoize $B + A$ as well.

When LARC is asked to perform an operation, it uses any identity short cuts, then checks the OperationStore to see if the operation has already been performed, and if so returns the MatrixID of the result found in the OperationRecord. If the operation has not been memoized, then LARC carries out the operation. Addition, multiplication and all other matrix operations in LARC are carried out recursively in a way that produces the four MatrixIDs of the quadrant submatrices of the output matrix. As with any matrix LARC attempts to store, it hashes the list of quadrant MatrixIDs and then looks in that hash chain of the MatrixStore to see if a matrix with those quadrant submatrices has already been stored. If the matrix is found its MatrixID is returned, otherwise a new MatrixRecord is created and its MatrixID is returned. Finally, the operation is now memoized in the OperationStore.

Even if the top level of an operation has not been memoized, LARC may have previously memoized operations further down the quadtree recursion, allowing LARC to skip the work for all branches below the point of that memoization. This cut down of work is similar to the cut down of memory that occurred in the MatrixStore when a submatrix of a larger matrix was previously stored. The combination of compressed storage, carrying out operations in compressed format, and memoizing operations allows LARC, and application packages which use LARC, to carry out computations that would otherwise seem to be intractable.

## 2.4 Scalar Types

At compile time, LARC users select the underlying data type for scalars inside of LARC matrices by compiling with a command such as:

```
make TYPE=INTEGER
```

The scalarType also determines the appropriate scalar arithmetic. The scalarTypes currently supported by LARC are:

| | |
|---|---|
| Real (default) | C long double |
| Integer | C int64_t |
| Complex | C long double complex |
| MPInteger | GMP multiprecision mpz_t |
| MPRational | GMP multiprecision mpq_t |
| MPReal | GMP multiprecision mpfr_t |
| MPcomplex | GMP multiprecision mpc_t |
| MPRatComplex | a LARC structure with real and imag in mpq_t |

## 2.5 Locality-Sensitive Hashing: Numerical Precision and Identities

Finite precision can lead to math identities going wrong, such as $1-(1-a) \neq a$. In LARC, when we have two or more scalars that differ only by some small amount due to numerical precision issues, it becomes a large problem. Not only do we need to store multiple copies of what should be the same scalar, we also need to store many matrices which are essentially equal, but for these small errors.

We have developed a technique involving locality-sensitive hashing [5]. Locality-sensitive hashing is an algorithmic technique that hashes similar input items into the same hash buckets with high probability. In our case 'similar' means that two scalars are close to equal. The locality-sensitive hash ensures that scalars that would be in the same region hash to the same value, making it fast to find a previously stored representative.

When attempting to store a scalar value in LARC's MatrixStore, we first want to determine whether a sufficiently nearby scalar has already been stored, that we should use instead. We detect this with high probability by dividing the space of scalars into small regions, defined by a user-controlled parameter, and hashing all scalars that would be in the same region to the same hash bucket. The first scalar found in a particular region is stored in the MatrixStore and we call this scalar the *representative* of that region. Attempting to store any new value that would lie in an already occupied region will result in LARC returning the MatrixID of the original representative and not storing the new value.

This technique could clearly cause additional issues with failing identities when substitute scalars are used in mathematical operations. For example, if we have already stored the number 22 in LARC, and we try to store the scalar $s = 7\pi \approx 21.9911485751$, and we have set the size of regions to be large e.g. 1/32, then LARC will return the MatrixID for 22 instead of storing $s$. This can lead to further errors (see examples of how choices of initial parameters can affect this in MyPyLARC/Tutorial/preloading_mults_pi.py).

We have an additional technique called *preloading* which we use to guarantee that preselected identities are satisfied. For example, since zero is particularly important to mathematical identities, it is the first thing that we store when initializing LARC. This ensures that zero is the representative of its region, and any scalar within the region containing zero will be treated as zero. Depending on the application, we may also choose to preload a select set of scalars such as the $n^{\text{th}}$-roots of unity, to ensure they become the representatives of their regions and that mathematical identities with these scalars hold (see examples of this preloading to ensure identities in MyPyLARC/Tutorial/examples_roots_unity.py and fft_paramFile_init.py).

## 2.6 Input and Output

LARC has its own compressed format to describe a single matrix; this can be used for input from a file to LARC and output from LARC to a file. These files use a JSON container and have a separate line for each unique MatrixRecord containing the MatrixID, the level, and the MatrixValue (which is either the list of MatrixIDs of the four quadrants or for scalar matrices, the value of the scalar). These LARC compressed matrix files can also contain metadata which LARC allows the user to record in an InfoStore indexed by the MatrixID. The metadata in the InfoStore allows a user to track parameters of interest such as scalar data type used, locality-sensitive hash parameter used, and comments. We designed the InfoStore to make logging of experimental parameters easy for input/output. Since there is no compression in the InfoStore, it is not intended for tracking information on very large sets of matrices.

The ability to store compressed matrices in files allows us to checkpoint important intermediate results in long runs. We can also write staged algorithms that output results and then start a new run loading the results from the previous program. This has the advantage of removing unneeded records and reducing memory requirements (see section 2.7 for more information on cleaning out storage).

In addition to the LARC compressed matrix format, LARC has the capability to read files containing matrices expressed in row-major format (header with dimensions, followed by items listed one at a time reading along rows). Using row-major format only makes sense for small matrices. LARC can also read sparse matrices in the Matrix Market Exchange Format (header with various info including number of nonzero items, followed by a line for each nonzero item with its coordinates and content). When LARC reads in a matrix, as always, it does not store copies of any matrix or submatrix that is already in the MatrixStore.

LARC can output small matrices in row-major format and larger matrices in LARC compressed format. LARC currently does not output matrices in Matrix Market Exchange Format.

## 2.7 Addressing Memory Challenges

The LARC package speeds computations by storing matrices and memoizing operations when the matrices involved have repeated submatrices. However, the memory requirements for some computations can become very large, so it is sometimes necessary to give up some previously stored matrices and memoized operations to permit continued computation.

If a computation can be divided into several programs which each culminate in one or more matrices containing the results from that stage, then LARC can take advantage of this staged computation. Each stage starts by initializing LARC, and reading one or more files with the results from the last stage; each stage ends by writing its results into one or more files. In this way, unneeded MatrixRecords and OperationRecords from previous stages do not occupy memory.

Another way to reduce memory requirements is to use the routines that LARC has available for *cleaning*. By cleaning, we mean the removal of certain matrices from the MatrixStore and removal of those OperationRecords from the OperationStore that contain removed matrices. The cleaning routines track recursive dependencies in the MatrixStore so that no MatrixRecord is removed if another MatrixRecord refers to it. This means that a request to remove a matrix will only be carried out if it does not violate this recursive closure condition. When a matrix is removed, LARC also removes any submatrices that are only used by this matrix.

When a matrix has been removed from the MatrixStore, it may still be referred to in an OperationStore record. These obsolete OperationRecords are removed whenever they are touched during the traversal of a OperationStore hash chain while searching for some memoized operation. There is also a cleaning option to empty the entire OperationStore if desired; this is most useful if the computation is entering a new stage in which few previously memoized operations are likely to be repeated.

# 3 Documentation and the MyPyLARC Tutorial and Examples Package

All LARC C routines have doxygen html documentation which is placed in the larc/html directory upon compilation and can be viewed in a browser.

An introduction to LARC can be found in the slide deck

MyPyLARC/larc/doc/aboutLARC.pdf.

These slides also contain a discussion of the block recursive implementation of the Cooley-Tukey discrete Fourier transform [7, pp. 20–22].

If you want to build a package using LARC for your matrix math, a great place to start is by looking at (and possibly copying and modifying) the MyPyLARC package. MyPyLARC is primarily composed of Python routines with examples and tutorials, and is intended for new users or developers that would like to see how to build a package on top of LARC.

MyPyLARC consists of:

- a README.md file with some basic information;
- a Tutorial directory MyPyLARC/Tutorial;
- several example project directories (MyPyLARC/FFT_play, Gate_play, and Roots_play);
- a src directory with C code that is used by the project Gate_play;
- a Makefile that integrates C and Python code for LARC and MyPy-LARC,
- a project template that you can copy and modify, Tutorial/user_proj_template.py, and an example of how to use this, Tutorial/user_proj_max-A-AT.py;
- sample initialization parameter files for the Tutorial and various projects, found in the directory MyPyLARC/InitParams (look at MyPyLARC/Eigen_play/power_meth for an example of setting initialization parameters depending on available memory);
- written text explanations in the files MyPyLARC/Eigen_play/how.we.did.this and MyPyLARC/Tutorial/newuser_instructions.

Many of the above are far from finished; MyPyLARC is a work in progress.

The Tutorial directory contains some interactive programs which introduce general concepts of LARC. There are routines to help the user learn about input/output, about initialization parameters for LARC, about creating and manipulating matrices with various operations, and examples of recursively structured code.

For the most part we expect that people who want to use LARC can do development in their own projects by copying and modifying what is done in MyPyLARC. However, if you are adding routines to, or otherwise modifying LARC, then you should check the functionality of your code and make sure you have not broken anything by compiling with

```
make unittests
```

which will run a suite of tests in each of the scalar data types.

While users are free to modify their own copies of LARC and MyPyLARC, at the current time we do not allow users to check their changes into the GitHub repository. However, we would be happy to get your suggestions for useful additions, and recieve bug reports. If you want to contact the developers at CCS you can send email to larc@super.org.

## 4 Acknowledgments

The majority of the current code in LARC and MyPyLARC is co-written by Steve Cuccaro and Jenny Zito. There are contributions to LARC or its applications from other CCS researchers, summer visitors, and from CCS integrees and interns. A list of many of our contributors is in larc/LARCcontributors.

The first version of LARC was created by Jenny Zito and John Gilbert in 2013, and coded in C by Zito. An early description of LARC by Cuccaro, John Daly, John Gilbert, and Zito is in the slide deck larc/doc/aboutLARC.pdf [3].

## References

[1] S. Abdali and D. Wise, “Experiments with Quadtree Representation of Matrices,” in *Symbolic and Algebraic Computation*, vol. 358 of *Lecture Notes in Computer Science*, ed. P. Gianni, Springer, 1989, 96–108.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed, MIT Press, 2009.

[3] S. Cuccaro, J. Daly, J. Gilbert, and J. Zito, “aboutLARC.pdf” Slides for talk on LARC given at Anne Arundel Community College April 2017 are included in GitHub release as larc/doc/aboutLARC.pdf

[4] D. Knuth, *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, 2nd ed., Addison-Wesley, 1997.

[5] J. Leskovec, A. Rajaraman and J. Ullman, “Mining of Massive Datasets, Chapter 3”, http://www.mmds.org.

[6] V. Strassen, “Gaussian elimination is not optimal”, *Numer. Math.* **13** 354–356, 1969.

[7] C. Van Loan, “Computational Frameworks for the Fast Fourier Transform,” *Frontiers in Applied Mathematics Vol. 10*, SIAM, Philadelphia, 1992.

[8] D. Wise, “Representing Matrices as Quadtrees for Parallel Processors: Extended Abstract,” *ACM SIGSAM Bulletin* 18, no. 3 (Aug. 1984): 24–25.

[9] D. Wise, “Representing Matrices as Quadtrees for Parallel Processors,” *Information Processing Letters* 20 (May, 1985): 195–199.

[10] D. Wise, “Parallel Decomposition of Matrix Inversion Using Quadtrees,” in *Proceedings of the 1986 International Conference on Parallel Processing*, ed. K. Hwang, S. Jacobs, and E. Swartzlander (1986), 92–99.nn

[11] D. Wise and J. Franco, “Costs of Quadtree Representation of Nondense Matrices,” *Journal of Parallel and Distributed Computing* 9, no. 3 (1990): 282–296.