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)

April 2021 (extended and revised from paper in March 2020 release)

Abstract

LARC (Linear Algebra via Recursive Compression) is a software package that stores matrices in a recursively compressed format and performs operations on them while they are compressed. LARC assigns each scalar, vector, and matrix a unique integer MatrixID. A 2^n by 2^n matrix is then recursively defined by a list of the four MatrixIDs of its quadrant submatrices. Functions depending on these MatrixIDs are used to produce hash values accessing three hash tables which store scalars, matrices and previously computed operations. These tables enable LARC to quickly retrieve previously stored matrices and operations, and thus to never store the same matrix twice or repeat an operation.

LARC contains a method called Regional Representative Retrieval that uses a locality sensitive hash and other techniques to snap nearby scalars together. Scalars that are important to identities can be preloaded, enabling LARC to mimic the behavior of symbolic computations over a ring.

The LARC matrix math package and a companion Python package MyPyLARC containing tutorials and sample applications are available on GitHub. This paper is intended as a companion paper to the associated GitHub code repository. It supplies motivations and explanations for LARC methodology as well as examples of applications for LARC and MyPyLARC.

Contents

1 Introduction

LARC (Linear Algebra via Recursive Compression) is a software package developed at IDA/CCS[1] that stores matrices in a recursively compressed format and performs operations on them while they are compressed. LARC assigns each scalar, vector, and matrix a unique integer ***MatrixID***. A 2^n by 2^n matrix is then recursively defined by a list of the four MatrixIDs of its quadrant submatrices.[2] Functions depending on these MatrixIDs are used to produce hash values accessing ***ScalarStore***, ***MatrixStore*** and ***OperationStore*** hash tables. These tables enable LARC to quickly retrieve previously stored matrices and operations, and thus to never store the same matrix twice or repeat an operation.

We have also developed several new techniques which substantively address numerical precision issues and, with high probability, allow LARC computations to mimic the behavior of selected identities involving symbolic expressions within an algebraic ring. In LARC each scalar stored in the ScalarStore will *claim* a small region of scalar space for which it is the unique representative. When a new scalar is calculated and it falls into a previously claimed region, the representative of that region is used as a replacement. This is enabled by a locality-sensitive hash which takes all scalars in a small range to the same hash value. This hash allows LARC to quickly retrieve the MatrixID of a previously stored scalar in that region or determine that no such scalar has been stored.

Since LARC stores each unique matrix and quadrant submatrix only once, LARC gains compression in its recursive representation of a matrix whenever there is quadrant submatrix reuse at any level of the recursion into smaller and smaller matrices. Examples of matrices that are highly compressible by LARC will be discussed later in the paper. Such matrices can arise from matrix algorithms which are intrinsically block or recursive in nature or from applications with a restricted number of scalars (e.g. sparse matrices). In these cases LARC is able to compress and store very large matrices and carry out matrix operations impractical in standard compressed or uncompressed formats.

Most of the LARC computations are carried out in C, but there is a user-friendly SWIG-generated Python wrapper. The Python interface is used by the MyPyLARC package. MyPyLARC provides tutorials, some example application experiments that use LARC to perform matrix math, and a template for users to implement their own experiments. Both LARC and MyPyLARC are available via GitHub at `https://github.com/LARCmath`.

This paper is intended as a companion paper to the associated GitHub code repository. It is intended to supply motivations and explanations for LARC methodology as well as examples

[1]IDA/CCS is the Center for Computing Sciences. CCS and its two sister organizations, the Centers for Communication Research in Princeton and La Jolla, form a Federally Funded Research and Development Center that is administered by IDA (the Institute for Defense Analyses). CCS research staff members work in a variety of areas involving algorithms for high performance computing, cryptography, network security and related cyber issues, signal processing, advanced techniques for analyzing extremely complex data sets, and alternative computing paradigms (see `https://www.ida.org/ida-ffrdcs/center-for-communications-and-computing`).

[2]While LARC will handle matrices of arbitrary dimension, it does so by padding each matrix with sufficient zeros to bring each dimension up to the nearest power of 2. Padding with zeros has little or no effect on LARC's ability to compress a matrix.

of applications for LARC and MyPyLARC. It is hoped that these will make the code both easy to use and easy to modify by future users.

The rest of the paper is organized as follows: in Section 2 we discuss the LARC matrix-math package with separate subsections on quadtree decomposition; recursive matrix storage and operations; Regional Representative Retrieval (our method for snapping nearby scalars together); supported scalar types, input-output formats, logging and metadata storage; and memory management techniques.

Section 3 discusses the MyPyLARC tutorial and sample application package. A subsection on a sparse block recursive discrete Fourier transform illustrates how the order of operations for an algorithm can affect its compressiblity in LARC. We describe a LARC implementation of a external benchmark challenge in the subsection on triangle counting in graphs. A subsection on a minimum energy physics problem illustrates an implementation in LARC of a power-method eigensolver. The subsections on reversible circuits and the quantum Sycamore circuit illustrate how gate operations can be translated into a linear algebra problem in LARC. We conclude this Section with a list of the tutorials and project directories available in MyPyLARC.

Section 4 discusses code and documentation availability.

Section 5 discusses some open research topics: Rayleigh dispersion for seismic events; a hybrid Lanczos-power method eigensolver; iterative methods that would vary region width to produce improved approximations; modifying LARC to allow improvement of the scalar values in the ScalarStore; a proposed special scalar type that would allow upper and lower bounding of probability calculations; various ways to calculate or approximate matrix inverses; Kronecker graphs for testing the MyPyLARC triangle counting code on large matrices; and integrating BLAS into LARC (or vice versa).

Section 6 contains our acknowledgments. It is followed by a reference section.

Appendix A discusses the details of Regional Representative Retrieval which is LARC's technique for snapping nearby scalars together. LARC's two modes for Regional Represenative Retrieval are Single-tile Probabilistic Retrieval (SPR) and Multi-tile Assured Retrieval (MAR). Before discussing the details, we discuss LARC's techniques for fast hash table access including the use of a locality sensitive hash for our ScalarStore and the use of "hashstamps" for search efficiency. This appendix includes an analysis of the "snapping" probability for SPR mode and MAR mode. It also discusses strategies for preloading scalars important to identities, and how to select the region width to both mimic symbolic computation and reduce memory. We conclude the appendix with a comparison of the relative advantages of MAR mode and SPR mode and an experimental comparison of the two modes.

Appendix B explains our algorithm for the conversion of standard sparse-row formats to LARC recursive format, while avoiding an intermediate representation as a dense matrix.

Appendix C examines the compressibility of various matrix families within LARC recursive format, and solves for the storage cost as a function of matrix dimensions.

Appendix D contains the open software license and copyright information.

2 The LARC Matrix-Math Package

2.1 Quadtree Decomposition and Matrix Computations

For large matrices, matrix operations consume a large amount of computational time and memory. There is a long history of clever methods to speed matrix calculations by using quadrant subdivision, for example:

- a sparse block-recursive version of the discrete Fourier transform based on the Danielson-Lanczos lemma [4] (see Section 3.1);
- Strassen's matrix multiplication algorithm [8];
- implementations of parallel matrix operations such as multiplication, division and trace by David Wise and colleagues [1, 10, 11, 12, 13].

Following the example of David Wise we call the tree structure of matrices created by recursive quadrant submatrix subdivision a ***quadtree decomposition***, or ***quadtree*** for short. LARC stores matrices using a recursive process based on their quadtrees, and will carry out matrix operations recursively while leaving matrices in this format. In Figure 1, there is an example of a quadtree decomposition, which has leaves that are 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: Quadtree Decomposition: for LARC, matrix operations and storage depend on the recursive subdivision of each matrix into four quadrant submatrices. Note that the children of each node are ordered according to the "Z" pattern (compass directions NW, NE, SW, SE) of positions within the matrix of submatrices.

For some of the historical work listed above, there was no reason to expect that many of the submatrices in the quadtree decomposition would be repeated. However, LARC is designed for applications in which there is a great deal of repetition of submatrices in the quadtrees. In the next section, we will describe how how LARC takes advantage of such repetition to reduce memory use and computation.

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.[3] 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 4×4 matrix M from Figure 1, which we divide into four 2×2 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:

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

For a $2^m \times 2^n$ matrix we call the exponents m and n the ***row level*** and ***column level*** respectively. For a square $2^m \times 2^m$ matrix we often simply refer to its ***level*** m. Notice that LARC expresses a level k matrix in terms of four matrices of level $k - 1$; thus the level 2 matrix M is composed of four level 1 quadrant submatrices $\{A, B, C, D\}$.

In LARC, each scalar, vector, and matrix is assigned a unique integer MatrixID. For a scalar (a 1×1 matrix), the ***ScalarRecord*** will be put into the ScalarStore hash table, and this record contains the value and the MatrixID of the scalar. Vectors and matrices are given a ***MatrixRecord***, which is put into the MatrixStore hash table. The MatrixRecord contains the MatrixID, row level, column level, and a recursive representation of the matrix called the ***SubMatList***. The SubMatList is a list of the integer MatrixIDs of the four quadrant submatrices of a matrix or of the two halves of a vector. For example, the matrix M above would have the SubMatList

$$[\mathtt{A_{ID}}, \mathtt{B_{ID}}, \mathtt{C_{ID}}, \mathtt{D_{ID}}]$$

that consists of the integer MatrixIDs from its four quadrant submatrices A, B, C, D. This list defines the matrix in a concise representation. We obtain a complete description of the contents of the matrix by recursively drilling down through the MatrixRecords for each quadrant submatrix until we reach the lists of four MatrixIDs corresponding to scalars.

Each MatrixRecord in the MatrixStore is indexed by a hash of its SubMatList. 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. In Section 2.3, we describe how matrix operations can be carried out recursively in a way that produces the SubMatList of the output matrix. Therefore, before creating a new MatrixRecord for the output and returning its MatrixID, we first hash the integers in the SubMatList and look in the MatrixStore to see if there is

[3] Recall that for dimensions that are not power-of-2, LARC has already padded the matrix with zeros.

already an existing matrix composed of those submatrices, and if so return its MatrixID instead.

For scalar matrices, LARC uses a special locality-sensitive hash of the scalar value to index into the ScalarStore, 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 2.4 on regional representative retrieval.

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. We think of the matrix as a quadtree of its submatrices, and traverse this tree.[4] The MatrixIDs are assigned to the submatrices in modified depth-first order. Our traversal ends when the root node (the top) has received a MatrixID. It can only receive its MatrixID when the MatrixIDs of its four children are known, so that we can hash the list and see if this matrix is already in the MatrixStore. If no previously stored matrix exists, LARC makes a new MatrixRecord with this list of children and returns the new record's MatrixID. This is a recursive process; if at any node we do not have the full list of children MatrixIDs, we traverse down the first child with no MatrixID. For scalar matrices (leaf nodes), the MatrixID can be found by hashing the scalar value and seeing if our regional representative retrieval technique returns the MatrixID of a previously stored ScalarRecord; if not, LARC creates a new ScalarRecord and returns that MatrixID.

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

$$A = \begin{bmatrix} .8 \end{bmatrix},\ B = \begin{bmatrix} \text{-}2 \end{bmatrix},\ C = \begin{bmatrix} .8 & .8 \\ .8 & \text{-}2 \end{bmatrix},\ D = \begin{bmatrix} 7 \end{bmatrix},\ E = \begin{bmatrix} 7 & 7 \\ 7 & 7 \end{bmatrix},$$
$$F = \begin{bmatrix} .8 & .8 & 7 & 7 \\ .8 & \text{-}2 & 7 & 7 \\ 7 & 7 & .8 & .8 \\ 7 & 7 & .8 & \text{-}2 \end{bmatrix},\ G = \begin{bmatrix} 7 & 7 & .8 & .8 \\ 7 & 7 & .8 & \text{-}2 \\ .8 & .8 & 7 & 7 \\ .8 & \text{-}2 & 7 & 7 \end{bmatrix},\ H = \begin{bmatrix} 7 & 7 & 7 & 7 \\ 7 & 7 & 7 & 7 \\ 7 & 7 & 7 & 7 \\ 7 & 7 & 7 & 7 \end{bmatrix},\ R, \tag{2}$$
$$I = \begin{bmatrix} 1 \end{bmatrix},\ J = \begin{bmatrix} 0 \end{bmatrix},\ K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},\ L = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix},\ M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},\ S.$$

[4]Matrix data is not normally presented in this fashion. Example conversions of various matrix formats to quadtree format are discussed in Section 2.6 and Appendix B.

We see that there are a total of 15 matrices to which we have assigned MatrixIDs. It takes three ScalarRecords and six MatrixRecords to store the matrix and unique quadrant submatrices that determine R recursively. We will refer to the total number of ScalarRecords and MatrixRecords needed to store a LARC matrix as the ***LARCsize*** of that matrix. Thus for our example, R has LARCsize nine. The recursive description of S, which has LARCsize 14, requires five ScalarRecords and nine MatrixRecords. However, when we store S after storing R, we only need an additional six records (two ScalarRecords and four MatrixRecords) because we have already stored the quadrant submatrices that S shares with R.

Table 1 shows how storage works in LARC for the example matrices (1) and their quadrant submatrices (2). The table has one line for each ScalarRecord and MatrixRecord required in its description, and each record appears in the order that it was created and stored.

Assigning MatrixIDs to ScalarRecords and MatrixRecords

MatrixID	Scalar Value or SubMatList	Level	
$\mathtt{A}_{\mathtt{ID}}$	.8	0	(1×1 matrix)
$\mathtt{B}_{\mathtt{ID}}$	-2	0	
$\mathtt{C}_{\mathtt{ID}}$	$(\mathtt{A}_{\mathtt{ID}}, \mathtt{A}_{\mathtt{ID}}, \mathtt{A}_{\mathtt{ID}}, \mathtt{B}_{\mathtt{ID}})$	1	(2×2 matrix)
$\mathtt{D}_{\mathtt{ID}}$	7	0	
$\mathtt{E}_{\mathtt{ID}}$	$(\mathtt{D}_{\mathtt{ID}}, \mathtt{D}_{\mathtt{ID}}, \mathtt{D}_{\mathtt{ID}}, \mathtt{D}_{\mathtt{ID}})$	1	
$\mathtt{F}_{\mathtt{ID}}$	$(\mathtt{C}_{\mathtt{ID}}, \mathtt{E}_{\mathtt{ID}}, \mathtt{E}_{\mathtt{ID}}, \mathtt{C}_{\mathtt{ID}})$	2	(4×4 matrix)
$\mathtt{G}_{\mathtt{ID}}$	$(\mathtt{E}_{\mathtt{ID}}, \mathtt{C}_{\mathtt{ID}}, \mathtt{C}_{\mathtt{ID}}, \mathtt{E}_{\mathtt{ID}})$	2	
$\mathtt{H}_{\mathtt{ID}}$	$(\mathtt{E}_{\mathtt{ID}}, \mathtt{E}_{\mathtt{ID}}, \mathtt{E}_{\mathtt{ID}}, \mathtt{E}_{\mathtt{ID}})$	2	
$\mathtt{R}_{\mathtt{ID}}$	$(\mathtt{F}_{\mathtt{ID}}, \mathtt{G}_{\mathtt{ID}}, \mathtt{H}_{\mathtt{ID}}, \mathtt{G}_{\mathtt{ID}})$	3	(8×8 matrix)
$\mathtt{I}_{\mathtt{ID}}$	1	0	
$\mathtt{J}_{\mathtt{ID}}$	0	0	
$\mathtt{K}_{\mathtt{ID}}$	$(\mathtt{I}_{\mathtt{ID}}, \mathtt{J}_{\mathtt{ID}}, \mathtt{J}_{\mathtt{ID}}, \mathtt{I}_{\mathtt{ID}})$	1	
$\mathtt{L}_{\mathtt{ID}}$	$(\mathtt{J}_{\mathtt{ID}}, \mathtt{J}_{\mathtt{ID}}, \mathtt{J}_{\mathtt{ID}}, \mathtt{J}_{\mathtt{ID}})$	1	
$\mathtt{M}_{\mathtt{ID}}$	$(\mathtt{K}_{\mathtt{ID}}, \mathtt{L}_{\mathtt{ID}}, \mathtt{L}_{\mathtt{ID}}, \mathtt{K}_{\mathtt{ID}})$	2	
$\mathtt{S}_{\mathtt{ID}}$	$(\mathtt{M}_{\mathtt{ID}}, \mathtt{G}_{\mathtt{ID}}, \mathtt{F}_{\mathtt{ID}}, \mathtt{H}_{\mathtt{ID}})$	3	

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

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} 1×1 scalar matrices, giving a LARCsize of

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

records. Thus, two 8×8 matrices with 128 distinct scalars require 170 records, versus the 15 required to store the highly compressible R and S matrices of our example.

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

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

A $2^n \times 2^n$ Hadamard matrix takes only $2n+1$ records to store recursively in LARC. Figure 2 shows the $2^3 \times 2^3$ Hadamard matrix H_3 in its LARC recursive form which consists of only two ScalarRecords and five MatrixRecords. The Hadamard matrix H_{100} is $2^{100} \times 2^{100}$ and requires only 201 records to store in LARC. (The compressibility, or lack of compressibility, of many standard families of matrices is examined in detail in Appendix C.)

H_3 is a $2^3 \times 2^3$ matrix.

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

MatrixRecords

Matrix ID	Scalar Value or SubMatList	Is the Matrix
$\mathtt{A}_{\mathtt{ID}}$	1	
$\mathtt{B}_{\mathtt{ID}}$	-1	
$\mathtt{C}_{\mathtt{ID}}$	$(\mathtt{A}_{\mathtt{ID}}, \mathtt{A}_{\mathtt{ID}}, \mathtt{A}_{\mathtt{ID}}, \mathtt{B}_{\mathtt{ID}})$	H_1
$\mathtt{D}_{\mathtt{ID}}$	$(\mathtt{B}_{\mathtt{ID}}, \mathtt{B}_{\mathtt{ID}}, \mathtt{B}_{\mathtt{ID}}, \mathtt{A}_{\mathtt{ID}})$	$-H_1$
$\mathtt{E}_{\mathtt{ID}}$	$(\mathtt{C}_{\mathtt{ID}}, \mathtt{C}_{\mathtt{ID}}, \mathtt{C}_{\mathtt{ID}}, \mathtt{D}_{\mathtt{ID}})$	H_2
$\mathtt{F}_{\mathtt{ID}}$	$(\mathtt{D}_{\mathtt{ID}}, \mathtt{D}_{\mathtt{ID}}, \mathtt{D}_{\mathtt{ID}}, \mathtt{C}_{\mathtt{ID}})$	$-H_2$
$\mathtt{G}_{\mathtt{ID}}$	$(\mathtt{E}_{\mathtt{ID}}, \mathtt{E}_{\mathtt{ID}}, \mathtt{E}_{\mathtt{ID}}, \mathtt{F}_{\mathtt{ID}})$	H_3

Figure 2: The H_3 Hadamard matrix is stored in LARC using seven matrix records.

While our examples show what happens with square matrices, we can easily extend this process to non-square 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.[5]

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 provides many basic unitary and binary operations including: matrix addition, matrix-matrix multiplication, scalar-matrix multiplication, Kronecker product, adjoint (complex conjugate transpose), and some matrix norms. In addition, LARC allows the user to define their own recursive operations with access to the full LARC infrastructure for operation storage and reuse.

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

[5]In the 2021 release, 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 $(\mathtt{A}_{\mathtt{ID}}, \mathtt{NULL}, \mathtt{B}_{\mathtt{ID}}, \mathtt{NULL})$ for column vectors where A and B are half-sized column vectors and $(\mathtt{C}_{\mathtt{ID}}, \mathtt{D}_{\mathtt{ID}}, \mathtt{NULL}, \mathtt{NULL})$ for row vectors where C and D are half-sized row vectors.

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,

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

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

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

Each unique operation completed by LARC is memoized[6] by the creation of an ***OperationRecord*** in the OperationStore. LARC implements the OperationStore as a hash table, with hash value determined by the MatrixIDs of the input matrices and the type of operation.

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. Furthermore, 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.

Every matrix operation takes one or more MatrixIDs as its input and returns the MatrixID of the result. For example the call for multiplying A and B would take as input the MatrixIDs A_{ID} and B_{ID}

$$\texttt{matrix\_mult}(A_{ID}, B_{ID})$$

[6] The word "memoized" comes from the the notion of making a memo. In computing, memoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls, and returning the stored result when the same inputs appear again.

and return the MatrixID C_{ID} of the matrix $C = A \times B$.[7]

Here is a summary of all the steps that occur when LARC carries out an operation:

- Check for mathematical identity shortcuts, and if found return appropriate MatrixID.
- Check to see if this operation has been previously performed. This is done by hashing(input MatrixIDs, name of operation) and looking in the OperationStore for a OperationRecord containing this information. If found, the record will also contain the MatrixID for the output of the operation, and this output MatrixID is returned.
- If the operation has not been previously carried out:
  - Recursively carry out the operation, which will produce the SubMatList corresponding to the result of the operation.
  - Determine the MatrixID of this result by hashing the integers in the SubMatList and looking in the MatrixStore to see if this SubMatList exists in a previously stored MatrixRecord. If found, this record will also contain the MatrixID for the larger matrix whose quadrant submatrices are given by the SubMatList. If not found, create a new MatrixRecord with this SubMatList and return the MatrixID of this new MatrixRecord.
  - Memoize the operation in the OperationStore by creating an OperationRecord with the operation name and the MatrixIDs of the inputs and result, indexed in the hash table by hashing(input MatrixIDs, name of operation).

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 cutdown of work is similar to the cutdown 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 be intractable.

2.4 Regional Representative Retrieval

LARC uses a technique we call ***regional representative retrieval*** to reduce the negative effects of limited numerical precision and to allow us to mimic symbolic computation. Before we explain the details, here is the motivation.

When running large applications, the earliest version of LARC was using large amounts of memory to save sets of scalars that were almost identical. The source of this problem was the finite precision in floating point operations, which can lead to the failure of a desired

[7]For speed of computation, the C code inside of LARC usually identifies a MatrixRecord by a pointer. From the Python interface we do not allow the user direct access to the memory (so that inexperienced users don't accidentally violate the fundamental rules). Thus Python functions refer to matrices and scalars by their MatrixID. Internally, LARC translates these Python functions to C functions using MatrixPTRs as the arguments.

identity to hold within its computer representation. For example, the identity $1-(1-a) \neq a$ when a is small, may fail to hold for a finite precision representation of the two different sides of the equation. If the math operations had been carried out symbolically the scalars would have been identical.

We have developed a special method for handling scalars in LARC called regional representative retrieval that addresses these finite precision issues and allows scalars differing only by a tiny amount (determined by an initialization parameter) to be collapsed to a single value. LARC saves the first scalar from a set of nearly identical scalars, then uses a fast hash-table-based search method to allow replacement of a newly calculated scalar by the previously-stored (almost identical) scalar. We will see that this technique also allows LARC to mimic symbolic computation.

This is how LARC performs a fast hash-table-based search for nearly identical scalars. LARC divides the space of scalars into small ***tiles***. For complex scalars, the tiles are two dimensional and for real they are one dimensional. The ScalarStore hash table uses a locality-sensitive hash that combines two functions: a function T which calculates a ***tile label*** for a scalar S, and a function H which hashes tile labels (LARC has two modes, described below, which use different types of tile labels; in one case, the tile label is composed of integers and in the other case, it is the center coordinate of the tile). Then H(T(S)) is a locality-sensitive hash that hashes all scalars within a single tile to the same value. (See [7] or Wikipedia for more information on locality-sensitive hashing.) LARC uses its scalar hash to identify cases where a newly calculated scalar need not be stored, because it can be replaced by a previously stored scalar.

LARC has two different schemes available: SPR (Single-tile Probabilistic Retrieval), and MAR (Multi-tile Assured Retrieval).

- SPR replaces a newly calculated scalar with a previously stored scalar in the same tile, if one exists, or otherwise stores the new scalar. SPR succeeds with high probability in identifying a newly calculated scalar with an almost-identical previously stored scalar, only failing to collapse two close scalars that are on opposite sides of a tile boundary.
- MAR uses a more complicated scheme in which adjacent tiles can be grouped together to form a region. This region will contain a single stored scalar which lies someplace in the central portion of the region. Unlike SPR, MAR has a guarantee of identifying (snapping) a newly calculated scalar with some almost-identical previously stored scalar within a user specified distance. This is accomplished at the cost of having some extra records indexed by different hash values.

For either SPR or MAR we call the unique stored scalar in a region the ***representative*** of the region (for SPR, a region is always a single tile; for MAR, it is usually two tiles for rational or real scalars, and it is usually four tiles for complex scalars). More details on these two schemes are given in Appendix A.

The regional representative retrieval scheme enables LARC to mimic symbolic computation for selected mathematical expressions over some field. This is achieved by ***preloading*** (in full precision) a selection of scalars that appear in these expressions. Preloading, that is, placing scalars in the store immediately after initialization, ensures that they become the representatives of their regions. For example, in our implementation of a discrete Fourier transform, we preload the nth-roots of unity $1, w, w^2, \ldots, w^{n-1}$ where $w = e^{2\pi i/n}$. This ensures that when LARC calls

`product(MatrixID(`w^j`),MatrixID(`w^k`)),`

that the function returns `MatrixID(`w^m`)` where $m = j + k \pmod{n}$.

Prior to loading scalars for identities important to particular applications (e.g. the discrete Fourier transform), LARC preloads the scalars 0, 1, and larger matrices that are critical to the additive identity ($A + Z_n = A$) and multiplicative identity ($A \times I_n = A$), where Z_n and I_n are the 2^n by 2^n zero and identity matrices (level n). (We also set flags in the MatrixRecords for Z_n and I_n so that we can quickly find and shortcut these identities.)

When specifying the region width before preloading useful scalars, the user needs to be aware that the region width determines the minimum distance by which two desired scalars must be separated in order to prevent the second from snapping to the first. The routines in MyPyLARC/Tutorial (which is described in Section 3) have examples that can help the user understand the techniques and pitfalls of using preloading to preserve identities. Often these problems can be handled by choosing good initialization parameters (see Appendix A for details).

2.5 Scalar Types

LARC code is primarily written in C, and there is a Makefile in the main directory. If the user enters "make" in the command line, the default is for the LARC code to be compiled with a scalarType that is "REAL" which is C long double. The user has the option to compile with other scalar types by specifying the desired type when they use the make command. For example:

```
make TYPE=INTEGER
```

will compile LARC so that all the scalars in the scalar store and all scalar arithmetic are carried out on "INTEGER" which is C int64_t.

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

LARC scalarType	**Implemented data type**
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 imaginary parts in mpq_t
Clifford	an array of mpq_t basis coefficients

(More information on the GMP multiprecision data types can be found at the GNU Manuals Online website, `https://www.gnu.org/manual`. The functions for mpz_t and mpq_t types are found in the GMP manual; those for the mpfr_t type in the Mpfr manual; and those for the mpc_t type in the MPC manual.)

LARC supports a general Clifford algebra scalarType (`make TYPE=CLIFFORD`). A Clifford algebra implements exact calculations in a field extension to the rational numbers by select algebraic numbers, for example $\mathbb{Q}\left[\sqrt{2}, \sqrt{3}\right]$. To express a Clifford algebra in LARC the user first specifies the (ordered) basis for the algebra, e.g. $(1, \sqrt{2}, \sqrt{3}, \sqrt{6})$ for the example above. Then when the user wants to enter a scalar from the Clifford algebra, she enters the basis coefficients of the element, e.g. $x = 1 + 3/2\sqrt{3}$ would be specified by the list $(1, 0, 3/2, 0)$. LARC allows rational coefficients for basis elements. Complex Clifford algebras are expressed by extending the basis to include imaginary elements e.g. $(1, \sqrt{2}, \sqrt{-1}, \sqrt{-2})$. Multiplication in a Clifford algebra must be closed so care should be taken when selecting the basis to ensure the closure. Several example Clifford algebras already have their multiplication rules coded in LARC. Thus for example with the basis $(1, \sqrt{2}, \sqrt{3}, \sqrt{6})$, multiplying the scalars with basis coefficients $(0, 1, 0, 0)$ and $(0, 0, 0, 1)$ would result in the scalar given by the array of coefficients $(0, 0, 2, 0)$. LARC has built-in support for the real Clifford algebras $\mathbb{Q}\left[\sqrt{2}\right]$, $\mathbb{Q}\left[\sqrt{2}, \sqrt{3}\right]$, $\mathbb{Q}\left[\sqrt[3]{2}\right]$, and for their complex versions $\mathbb{Q}\left[i, \sqrt{2}\right]$, $\mathbb{Q}\left[i, \sqrt{2}, \sqrt{3}\right]$, $\mathbb{Q}\left[i, \sqrt[3]{2}\right]$. With these as examples, the user may modify LARC to add a new Clifford algebra to the list by adding its basis and defining its multiplication rules.

One application of the Clifford algebra scalarType $\mathbb{Q}\left[\sqrt{2}, \sqrt{3}\right]$ is for exploring discrete planar geometry (e.g., tilings, transformations, and symmetries) involving equilateral triangles, squares, regular hexagons, regular octagons, regular 12-gons, and regular 24-gons. This is because $\sin(\pi/12) = (\sqrt{6} - \sqrt{2})/4$ and $\cos(\pi/12) = (\sqrt{6} + \sqrt{2})/4$ are exact elements of that Clifford algebra, so that computations involving vertex coordinates of the aforementioned geometric shapes can be made exactly.

A second application of a Clifford algrebra scalarType, with basis $\mathbb{Q}\left[i, \sqrt{2}, \sqrt{3}\right]$, is described in Section 3.5, which discusses components of an IBM quantum density matrix simulator called the Sycamore Circuit.

Regional representative retrieval (RRR) is used for all scalarTypes. If the user does not desire scalars which are close to each other to be collapsed to a single value, RRR can be side-stepped (with high probability) by making the tile size extremely small with the choice of initialization parameters (see Appendix A). It is also possible to turn on an alert (ALERT_ON_SNAP) which warns the user whenever two scalars in the same region are collapsed together. This might be particularly useful when the user is attempting exact symbolic computation for rational numbers or for a Clifford algebra.

2.6 Input, Output, and Saving Metadata

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 and ScalarRecord. The MatrixRecords contain the MatrixID, the level, and the SubMatList which describes the matrix. The ScalarRecord contains the MatrixID and the numerical value of the scalar.

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).

The LARC compressed matrix files can also contain metadata which the user has added to the InfoStore record for that matrix. The InfoStore is a hash table which is designed to contain metadata for a small selection of important matrices. For example, if the user has a staged set of experiments in which an output matrix file produced by one experiment is used as the input for another experiment, the user can use the InfoStore record for this matrix (and hence the record in the output file) to keep information such as the date, input data type, code version, input parameters, data source, and general comments. Since there is no compression in the InfoStore, it is not intended for tracking information on very large sets of matrices.

LARC also has a logging facility which allows users to create a date-time stamped log directory containing files related to a program run. For example, these log files could contain information such as input parameters and data, and output results and matrices. Logging can also be useful when carrying out staged experiments.

LARC has input and output capabilities in a variety of matrix formats. Each of the following file formats contain a single matrix.

- ***Row major format*** has the dimensions of the matrix followed by a list containing all elements of the matrix with the elements in the first row listed first, followed by the second row, etc.
- ***Sparse format*** has some header information such as the dimensions of the matrix, followed by a list of nonzero elements of the matrix, with one element per line, specifying the matrix coordinates (i,j) and the value at that location (the order of these lines is not important).
- ***LARC compressed format*** is a JSON file (or a Python dictionary) reflecting the internal structure inside the MatrixStore and ScalarStore for the matrix. In JSON, there is one line for each unique submatrix of a matrix. The line for a scalar contains its MatrixID and the value of the scalar. The line for a larger matrix contains its MatrixID and a list of four MatrixIDs for its quadrant submatrices.

Note that as a LARC compressed format file is read into the current LARC MatrixStore and ScalarStore, the MatrixIDs from the file will be translated to new values. This is necessary because some of the submatrices may already be present in the current MatrixStore or ScalarStore, in which case we use their current MatrixIDs; unstored matrices or scalars are assigned MatrixIDs in numerical order as they are added to the stores. Also note that a single matrix may have nonidentical file versions depending on the state of the stores when the matrix is output. To check that two files represent the same matrix although the MatrixIDs differ, one can either read them both into the same LARC stores and verify they give the same MatrixID, or use the LARC Python utility `larcMatrix_files_matrix_equal.py`.

LARC can input or output row major format, but since this format contains an entry for every position in the matrix it is not feasible for LARC to use this format for large matrices. For very small matrices, LARC can also output the entire matrix in a “naive” or dense format, printed with one row per line.

When LARC reads sparse matrices, such as those in the Matrix Market Exchange Format, it converts to an intermediate format before translating to LARC compressed format without ever representing the matrix in a dense format. The details of this algorithm are in Appendix B. MyPyLARC contains Python routines to convert various other sparse formats to Matrix Market Exchange Format. The MyPyLARC package contains example code using the LARC calls to read graphs in Matrix Market format and implements a triangle counting example (the code is in MyPyLARC/Count_triangles and the application description is in Section 3.2). LARC does not currently have code to output matrices in Matrix Market Exchange Format or other sparse formats.

There are also examples in the MyPyLARC tutorials and sample applications that read row major format, and also examples that convert a LARC matrix to a Python dictionary.

2.7 Memory Management: Staging, Cleaning, and StopHogging

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 stages, each of which culminates in one or more matrices containing the results from that stage, then LARC can take advantage of this staged computation to reduce memory requirements. Each stage, after the first, starts by initializing LARC and reading one or more data files with the results from the last stage; each stage ends by writing its results into one or more files. In this way, unneeded Scalar-, Matrix- and OperationRecords from previous stages do not occupy memory. This staging can be accomplished by creating individual programs for each stage. Alternately, the user can write a single program which calls a function which shuts down LARC and frees its memory at the end of each stage.

Another way to reduce memory requirements is to use the routines that LARC has available for in-process ***cleaning***. By cleaning, we mean the removal of certain MatrixRecords from the MatrixStore and ScalarRecords from the ScalarStore, along with removal of those OperationRecords from the OperationStore that contain removed matrices. The cleaning routines track recursive dependencies in the MatrixStore so that no MatrixRecord or ScalarRecord 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. There are also `lock` and `hold` functions which prevent a matrix from being cleaned, either until

LARC is terminated or until the user specifically allows it to happen; use of these functions ensures that important matrices are not accidentally removed from the MatrixStore or ScalarStore.

A matrix that has been removed from the MatrixStore or ScalarStore may still be referred to in an OperationStore record. These obsolete OperationRecords are removed whenever they are found in the process of 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 phase in which few previously memoized operations are likely to be repeated.

On many applications that we have implemented, memory issues are much more challenging for LARC than speed of computation. When a LARC application requires too much memory to run on a desktop computer, we move to a bigger shared machine. To make it convenient for LARC users to be good citizens of shared resources, LARC contains a routine called `stopHogging` that is launched each time LARC is initialized, and will terminate a run if it is utilizing most of the free memory on a resource.

3 MyPyLARC Tutorial and Applications Package

To make it easy for new users to see how to use LARC, we have written the MyPyLARC tutorial and sample applications package. MyPyLARC is largely written in Python (a few C routines are used in certain applications), and the MyPyLARC applications and tutorials use the SWIG-generated LARC Python wrapper functions to call the LARC routines which perform most mathematical operations.

The MyPyLARC/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. See the list in Section 3.6.

In the next few sections we describe some of the sample applications that are available in MyPyLARC: a sparse block recursive discrete Fourier transform, an eigen-solver used for a lowest energy calculation, a routine for triangle counting in graphs, and a circuit simulation for a small portion of the Google Sycamore circuit.

Because LARC saves each unique matrix only once, and carries out each matrix operation only once, there is some automatic algorithmic optimization. However, when implementing any particular algorithm it is wise for the researcher to tailor the algorithm to further take advantage of LARC's strengths. Each of the following sections describes a particular problem and our implementation of a solution using LARC. We hope that these examples will provide some useful strategies for researchers who are interested in using LARC to tackle their own problems.

3.1 Block Recursive Discrete Fourier Transform

In the following example we discuss a block sparse representation of the discrete Fourier transform (DFT), and detail our ideas as to how to perform such an algorithm efficiently in LARC. We will see that decisions about how to group sequences of operations can greatly impact the efficiency of the algorithm.

The Danielson-Lanczos lemma [4] gives a block recursive definition for the DFT matrix. In our notation, this equation becomes

$$F_k = C_k(I_1 \otimes F_{k-1})P_k \tag{3}$$

where the subscript k (the level) indicates a matrix of size $n \times n$ where $n = 2^k$. Since in LARC all matrices are power-of-two dimensioned, this convention simplifies our notation. This means that the $2^k \times 2^k$ identity matrix is denoted as I_k, rather than I_n, and in particular $I_1 = \left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$.

The matrices in the product on the right-hand side of Equation 3 are highly structured and sparse. For example when $k = 3$, the first matrix in the product is:

$$C_3 = \left[\begin{array}{cccc|cccc}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & w_3 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & w_3^2 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & w_3^3 \\
\hline
1 & 0 & 0 & 0 & w_3^4 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & w_3^5 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & w_3^6 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & w_3^7
\end{array}\right]$$

where $w_3 = e^{(2\pi i)/8}$.

The second matrix in the product is a block diagonal matrix formed as the Kronecker product of the 2-by-2 identity matrix I_1 with the half-size DFT matrix F_2:

$$I_1 \otimes F_2 = \left[\begin{array}{cccc|cccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & i & -1 & -i & 0 & 0 & 0 & 0 \\
1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\
1 & -i & -1 & i & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & i & -1 & -i \\
0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 1 & -i & -1 & i
\end{array}\right]$$

The third matrix in the product is a structured permutation matrix:

$$P_3 = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right]$$

The mathematical description of the matrices C_k and P_k have recursive forms. The sparse matrix C_k is constructed from smaller diagonal matrices, as

$$C_k = \begin{pmatrix} I_{k-1} & D_{k-1} \\ I_{k-1} & -D_{k-1} \end{pmatrix}$$

where $D_{k-1} = \operatorname{diag}(1, w_n, w_n^2, \ldots, w_n^{\frac{n}{2}-1})$ with $n = 2^k$ and $w_n = e^{2\pi i/n}$ is the n-th root of unity.[8] Unlike the larger C_k matrices, C_1 is not sparse; it is the 2×2 Hadamard matrix $H_1 = \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$.

The matrix P_k is the inverse shuffle permutation matrix. Each row of the matrix has a single 1. In the top half of the matrix the 1's fall in the even indexed columns $0, 2, 4, \ldots$, while in the bottom half of the matrix the 1's fall in the odd columns $1, 3, 5, \ldots$. For $k \geq 2$, the matrix P_k can be described in terms of the quadrant submatrices of P_{k-1}, which are denoted below by $P_{k-1}[00]$, $P_{k-1}[01]$, $P_{k-1}[10]$, and $P_{k-1}[11]$ (and each have level $k-2$).

$$P_k = \left[\begin{array}{cc|cc} P_{k-1}[00] & P_{k-1}[01] & Z_{k-2} & Z_{k-2} \\ Z_{k-2} & Z_{k-2} & P_{k-1}[00] & P_{k-1}[01] \\ \hline P_{k-1}[10] & P_{k-1}[11] & Z_{k-2} & Z_{k-2} \\ Z_{k-2} & Z_{k-2} & P_{k-1}[10] & P_{k-1}[11] \end{array}\right] \text{ where } P_{n-1} = \begin{bmatrix} P_{k-1}[00] & P_{k-1}[01] \\ P_{k-1}[10] & P_{k-1}[11] \end{bmatrix}.$$

$$\text{where } P_0 = \begin{bmatrix} 1 \end{bmatrix};\ P_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; P_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Each scalar in the DFT matrix is a power of the n-th root of unity w_n. These roots of unity form a multiplicative group, that is, $w_n^\ell w_n^m = w_n^{\ell+m \pmod n}$. Choosing to prestore high precision approximations of these roots ensures that computations involving these roots mimic the symbolic mathematical identities such as this closure under multiplication. That is, when LARC calculates

$$\texttt{product(MatrixID(}w_n^\ell\texttt{),MatrixID(}w_n^m\texttt{))}$$

[8]Note that $w_n^{\frac{n}{2}} = -1$ so the matrix C_3 above has the w_3 to higher exponents in the bottom right quadrant, giving an alternative expression for the diagonal of the matrix $-D_{k-1}$.

it returns `MatrixID(`$w_n^{\ell+m \pmod n}$`)`. See Appendix A.7 for general tips on picking good initialization parameters and Appendix A.9 for experiments we ran to pick good input parameters specifically for the roots of unity equations.

LARC can take advantage of the limited number of unique scalars to achieve some compression, but since there is no other submatrix reuse, the size of the matrix still scales as $O(2^{2k})$. (The exact formula for the LARCsize of F_k is $(4^{k+1}-1)/3 - 4^k + 2^k$; see Table 2.) However, we are more interested in performing the DFT (i.e., multiplying the DFT matrix and some other vector/matrix) than in constructing the DFT matrix itself, and the matrices C_k and P_k compress well, so we look for a more efficient algorithm using only these matrices.

Expanding the expression for F_k in terms of these matrices, using the distributive law for Kronecker products, yields:

$$\begin{aligned} F_k &= C_k(I_1 \otimes F_{k-1})P_k \\ &= C_k(I_1 \otimes [C_{k-1}(I_1 \otimes F_{k-2})P_{k-1}])P_k \\ &= C_k(I_1 \otimes C_{k-1})(I_2 \otimes F_{k-2})(I_1 \otimes P_{k-1})P_k \end{aligned}$$

We continue such expansion until the matrix in the center is $I_k \otimes F_0$. Since $F_0 = [\,1\,]$, this central matrix is just the identity matrix, I_k. The resulting expression is:

$$F_k = C_k(I_1 \otimes C_{k-1}) \dots (I_{k-1} \otimes C_1) I_k (I_{k-1} \otimes P_1)(I_{k-2} \otimes P_2) \dots (I_1 \otimes P_{k-1})P_k$$

Since $P_1 = I_1$, we can simplify the above to

$$\begin{aligned} F_k &= C_k(I_1 \otimes C_{k-1}) \dots (I_{k-1} \otimes C_1)(I_{k-2} \otimes P_2) \dots (I_1 \otimes P_{k-1})P_k \\ &= \left(\prod_{i=0}^{k-1}(I_i \otimes C_{k-i})\right)\left(\prod_{i=2}^{k}(I_{k-i} \otimes P_i)\right) = \overline{C}_k\overline{P}_k \end{aligned}$$

While the expression above implies an algorithm for the DFT which creates two matrices and multiplies them in turn with a vector/matrix, this may not be the optimal algorithm. Any choice of grouping of the matrices in the fully-expanded expression is a valid algorithm for applying the DFT matrix, and some such groupings may be more efficient in LARC than others. In particular, it may be more efficient to perform more multiplies with sparser matrices, than fewer multiplies with more dense matrices.

The LARCsize of the P_k matrix is just $5(k-1)+2$ (for $k > 1$); as can be seen from the recursive description of these matrices, there are only two distinct scalars and a great deal of submatrix reuse. The product matrices $\overline{P}_k$ grow as in Table 2, with larger increases when going from odd to even k values. The compression for $\overline{P}_k$ is still good, being of order $O(2^k)$, so it may make sense to store $\overline{P}_k$ and use that; this will have to be tested.

In contrast, while the C_k are sparse and LARC compresses them well (the growth is roughly $O(2^k)$, though with LARCsizes larger than $\overline{P}_k$), the product matrices $\overline{C}_k$ are dense and grow as $O(2^{2k})$, so it will probably make more sense to perform k multiplies with the well-compressed $(I_i \otimes C_{k-i})$ than to construct the $\overline{C}_k$ matrix and use it.

k	P_k	$\overline{P}_k$	C_k	$\overline{C}_k$	F_k
0	1	1	–	–	1
1	3	3	3	3	3
2	7	7	9	8	9
3	12	12	19	23	29
4	17	28	37	74	101
5	22	45	71	261	373
6	27	109	137	976	1429
7	32	174	267	3771	5589
8	37	430	525	14822	22101
9	42	687	1039	58769	87893
10	47	1711	2065	234044	350549
11	52	2736	4115	934119	1400149
12	57	6831	8213	3732370	5596501
13	62	10929	16407	14921277	22377813
14	67	27313	32793	59668712	89494869
k	$5k-3$	$O(2^k)$	$O(2^k)$	$O(4^k)$	$\frac{4^k-1}{3}+2^k$

Table 2: LARCsizes of the matrices discussed in this note. They were calculated using scalarType MPComplex. For this run, the initialization parameter regionbitparam was set to 200. Strategies for parameter choices are discussed in Appendix A.

We hypothesize that storing the $\overline{P}_k$ and the individual C_k may result in the most efficient implementation of the DFT. However, with any LARC implementation some experiments should be run, as efficiency is impacted by platform, memory, cache sizes and structure, and possibly other factors.

3.2 Triangle Counting in Graphs

The MyPyLARC Triangle_play project contains code to count the number of triangles in graphs. Triangle counting was a featured problem from the 2017 MIT/Amazon/IEEE Graph Challenge [17]. This Graph Challenge group publishes challenges every year to stimulate research, provide a benchmark for developers, and highlight any resulting innovations in graph algorithms. The group provides a set of graph input data, and after the competition publishes a summary paper that describes the algorithms and results for various groups who submitted to the competition. This gives developers an opportunity to compare their results with those of the community.

Although the competition was completed several years ago, our team decided to implement one of the algorithms for triangle counting on LARC. Because the adjacency matrix of a sparse graph is sparse, we expected that this problem might be a good fit for LARC. It also provided us with an opportunity to test our algorithm for efficiently reading sparse matrix data into the LARC recursive format (see Appendix B).

The MIT benchmark data for the triangle counting challenge problem was in a sparse format. Each file specifies an incidence matrix A of a graph G. The incidence matrix entry $A(i, j)$ is 1 if edge (i, j) is in the graph G and is 0 otherwise. The file only contains the coordinates of the nonzero entries (one line per edge).

We implemented one of the algorithms described in the results paper from this challenge (Algorithm 2 in [18]). This algorithm takes as input the incidence matrix A. It calculates $A^3 = A \times A \times A$; then sums the diagonal entries, giving trace(A^3); and finally divides this number by six. This yields a count of the number of triangles in the graph for the following reason. The graph G is undirected and entry $A(i, j)$ of the incidence matrix can also be interpreted as the number of paths containing a single edge that go from vertex i to vertex j of the graph G (it is 1 if they are connected and 0 otherwise). Thus A^2, which has entries $A^2(i, j) = \sum_k A(i, k)A(k, j)$, will count the number of paths containing two edges that connect vertex i to vertex j in the graph. Similarly $A^3(i, j)$ counts the number of paths with exactly three edges connecting vertex i to vertex j in G. A triangle with vertices $\{i, j, k\}$ in G is counted by six paths that contribute to the diagonal of A^3:

$$\begin{array}{ll} i \to j \to k \to i, & i \to k \to j \to i, \\ j \to i \to k \to j, & j \to k \to i \to j, \\ k \to j \to i \to k, & k \to i \to j \to k. \end{array}$$

Thus the trace, which is the sum of the diagonal entries, will count the number of triangles in G six times.

The largest challenge incidence matrix was $2,174,640$ by $2,174,640$ and contains $57,334,760$ entries which are "1". LARC zero pads this matrix to the next power of two dimensioned matrix 2^{22} by 2^{22} (note $2^{22} = 4,194,304$) and its JSON file has one line for each of the $111,797$ distinct submatrices of the matrix, including itself.

LARC does a great job of naturally compressing the benchmark data. This is true even though LARC format is output into a ASCII file using a JSON format and contains much additional information which we keep for convenience. For the MIT data set, LARC's compression gets stronger as the files become larger. For small files, both MIT native CSV sparse files and LARC format JSON files take up about 4kB. The largest MIT files of around 860 MB are reduced in size by a factor of 150 to 5 MB in LARC format.

We ran the LARC triangle counter on all the MIT benchmark data. We used triangle count data available at the SNAP site to verify our results on a sample of our runs.

Counting triangles with the LARC application code on the largest benchmark graph at the MIT site takes 12.5 minutes. All the other graphs took less time. It is hard to compare our run times with the results from other teams, since all of them are using parallel computers and optimized code, whereas we are running on a single processor with unoptimized code and tracking many statistics for development purposes. The competition submissions on a graph of roughly this size took between 100 seconds and less than 1 second. The published results do not specify which run time corresponded to what number of processors or memory size.

In order to test their triangle counting code against larger graphs, some of the authors, e.g. [16], generated large Kronecker graphs with up to approximately 10^{12} edges. The approach of modeling networks with Kronecker graphs is an easy way to generate large scale graphs [15]. A Kronecker graph is formed as follows. Start with incidence matrix K that specifies some initial graph (called the "initiator"), then for some integer k take the k-th Kronecker power of the matrix K with itself to give $M = K^{\otimes k} = \underbrace{K \otimes K \otimes \cdots K}_{k\ terms}$. It is easy to generate Kronecker graphs in LARC. We would like to implement generation of Kronecker graphs and test the triangle counting code at some point in the future.

3.3 Eigensolver with Physics Application

MyPyLARC's Eigen_Play project calculates the lowest-energy eigenvalue and associated eigenvector for a quantum Hamiltonian operator. Briefly, the method consists of two parts: discretizing the problem to create a matrix representation for a spatially-bounded version of the Hamiltonian; and applying the power method (that is, repeated multiplication of a random vector by the Hamiltonian matrix until the result has converged) to obtain the desired eigenstate. A full analysis of the problem can be found in the file `MyPyLARC/Eigen_play/physics_work/schroedinger.pdf`.

Here is an overview of the concepts. In quantum physics, the Hamiltonian operator $H(\mathbf{x})$ on functions $y(\mathbf{x})$ of vector $\mathbf{x}$ is an operator corresponding to total energy. It is a sum of Hermitian operators that has the form

$$H(\mathbf{x}) = c\nabla^2 + V(\mathbf{x})$$

where c is a constant, ∇^2 represents the Laplace operator and $V(\mathbf{x})$ is a real-valued multiplicative term corresponding to potential energy. We look for solutions to the eigenproblem

$$H(\mathbf{x})\,(y(\mathbf{x})) = c\nabla^2\,(y(\mathbf{x})) + V(\mathbf{x}) * y(\mathbf{x}) = \lambda y(\mathbf{x}).$$

This equation can have ***bounded*** solutions which have compact support over $\mathbf{x}$ (i.e., $y(\mathbf{x}) = 0$ outside of finite boundaries for the $\mathbf{x}$ variables) and/or ***unbounded*** solutions which do not have compact support, depending on the characteristics of the multiplicative $V(\mathbf{x})$ term.

The LARC application assumes a single spatial dimension x (scalar) and a potential $V(x)$ which supports at least one bounded solution. (Two example potentials are provided.) The result of the calculation is the unique bounded solution $y(x)$ which has no nodes; that is, $y(x)$ is zero outside the boundaries and is non-negative within the boundaries. This eigenvector of H is also the one with minimal eigenvalue.

In order to use LARC's matrix operations to solve this problem, it is necessary to work within finite boundaries in space. The differential equation above is turned into a $2^n \times 2^n$ matrix by discretizing space within these boundaries and assigning row/column indices to the $N = 2^n$ points. (Physically, this would be equivalent to modifying the $V(x)$ function to be infinite outside of the chosen bounds. If the bounds are chosen wisely, the values we wish to calculate will be unaffected.)

To get a matrix form for the problem equation, the boundary points x_1, x_N and some unit of distance ℓ are chosen such that $N = (x_N - x_1)/\ell$. (The boundary conditions are now expressed as $y(x_1 - \ell) = y(x_N + \ell) = 0$.) An approximation to the Laplacian operator applied to y is obtained using the central difference formula:

$$y''(x_i) \approx \ell^{-2} \left[y(x_{i+1}) - 2y(x_i) + y(x_{i-1}) \right].$$

At the endpoints, we take advantage of the assumption that $y(x) = 0$ beyond our boundaries to get

$$y''(x_1) \approx \ell^{-2}[y(x_2) - 2y(x_1)]$$
$$y''(x_N) \approx \ell^{-2}[-2y(x_N) + y(x_{N-1})].$$

The matrix that results is tridiagonal Toeplitz:

$$M' = \frac{c}{\ell^2} * \begin{bmatrix} 2 & -1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & -1 & 2 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & -1 & 2 \end{bmatrix}$$

The error for this approximation can be made as small as necessary by increasing N. Thus, the discretized solution vector $y(x_i)$ satisfies the equation

$$M' * y + V \odot y = E * y$$

where E is a scalar constant and the $\odot$ operator represents element-wise multiplication. To move fully to standard matrix notation, the matrix $\mathrm{diag}(V)$ is defined to be a diagonal matrix with the elements of $V(x)$ on the diagonal, and thus

$$My = (M' + \mathrm{diag}(V))y = Ey$$

The matrix M is tridiagonal but no longer Toeplitz, since every diagonal element can be different. It will still be highly compressible in LARC, as the $N \times N$ matrix has at most $N+2$ distinct scalars and large all-zero off-diagonal blocks.

The power method works by iteratively generating the eigenvector of a matrix M with eigenvalue of the largest magnitude. Note that any N-long vector can be expressed as a weighted sum of the eigenvectors v_i of M:

$$y_0 = \sum_{i=1}^{N} c_i v_i \text{ where } M v_i = \lambda_i v_i.$$

Without loss of generality, the eigenvectors are labeled in order of decreasing magnitude of their associated eigenvalues:

$$|\lambda_1| > |\lambda_2| \geq |\lambda_3| \geq \ldots |\lambda_N|$$

(The rate of convergence of the iteration depends on $|\lambda_1| - |\lambda_2|$, so strict inequality is necessary in this one comparison.) Here is the algorithm. Make a suitable choice for y_0 (we can guarantee $c_1 \neq 0$ by using a unit vector with its 1 located near the minimum of $V(x)$), then multiply y_0 by M and then normalize the result to obtain y_1, multiply y_1 by M and normalize to get y_2, and iterate on the y_i until vectors y_k and y_{k-1} are equal within some desired tolerance. This works because

$$y_k = \sum_{i=1}^{N} c_i^{(k)} v_i$$

where

$$c_i^{(1)} = \frac{c_i \lambda_i}{\sum_i c_i \lambda_i}; \; c_i^{(2)} = \frac{c_i^{(1)} \lambda_i}{\sum_i c_i^{(1)} \lambda_i};$$

and so on. Since $|\lambda_1|$ is strictly larger than the magnitudes of the other eigenvalues,

$$\lim_{k \to \infty} y_k = v_1.$$

The desired lowest-energy eigenvector of M may not naturally have the largest magnitude eigenvalue. However, if the potential $V(x)$ is replaced by $V'(x) = V(x) - C$ (with any constant C), the resulting matrix has the same eigenvectors v_i and eigenvalues $\lambda_i' = \lambda_i - C$. Thus, a sufficiently large value for C will guarantee that the lowest-energy eigenvector has the largest magnitude eigenvalue λ_1', and a simple adjustment recovers the desired value of λ.

Figure 3 shows the results of this algorithm when $V(x)$ is the Morse potential

$$V(x; D_e, x_e, a) = D_e \left(1 - e^{-a(x-x_e)}\right)^2 - D_e = D_e \left(e^{-2a(x-x_e)} - 2e^{-a(x-x_e)}\right)$$

This potential is an approximation for the potential of a diatomic molecule, with the variable x being the distance between the two atoms. When the parameters D_e, x_e, a are chosen such that the potential allows bound states, these states have analytic solutions. The lowest-energy solution is

$$y_0(z) = \left[\frac{2\lambda - 1}{\Gamma 2\lambda}\right]^{1/2} z^{\lambda - 1/2} e^{-z/2} \text{ where } z = 2\lambda e^{x-x_e}; \quad \lambda = \sqrt{2mD_e}/\alpha\hbar$$

with energy $E_0 = -\frac{\hbar^2 \alpha^2}{2m} \left(\lambda - 1/2\right)^2$.

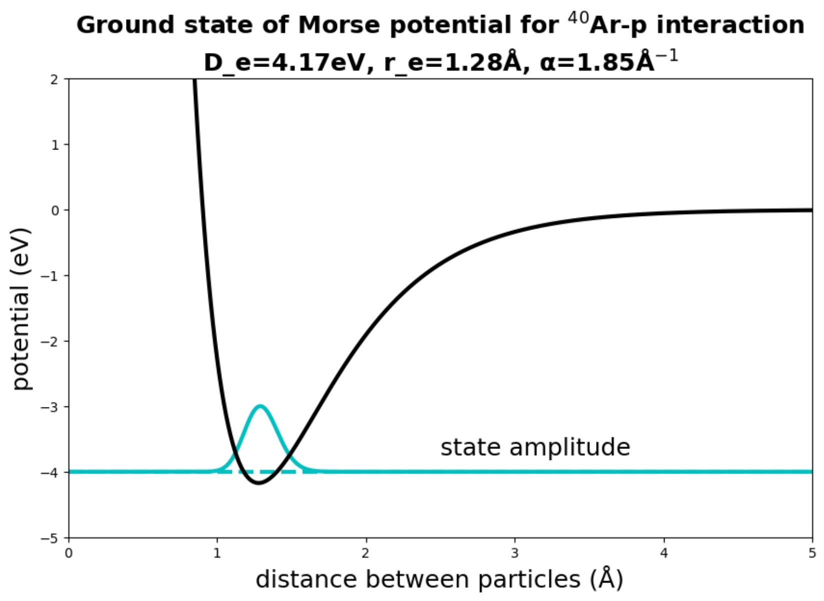


Figure 3: A plot of a Morse potential and its lowest eigenvector.

3.4 Reversible Circuits

The MyPyLARC/Gate_play/practiceGates.py program illustrates the construction of a circuit matrix using routines from MyPyLARC/src/gate.c. The circuit matrix is constructed from a series of gates, each represented by a permutation matrix, which are then multiplied together. The demonstration creates a reversible circuit using the universal gate library for reversible computing (NOT, CNOT, CCNOT).

In reversible computing, we speak of wires which carry binary values (off = 0, on = 1) and gates which when applied to a set of wires may change the value on one (or more) of the wires. A ***target*** wire is a wire whose value might change when a gate is executed. A ***control*** wire is a wire whose value determines whether a gate is executed. The reversible gates NOT, CNOT, and CCNOT have one or more target wires, and if the gate is executed, then the value on the target wires is toggled ($0 \to 1, 1 \to 0$). The gates CNOT and CCNOT also depend on one or more control wires which determine whether or not to execute the gate on the target. Generally, the value on the control wire(s) must be 1 for the gate to execute, but there is an option for the singly-controlled CNOT to have the gate execute when the value on the control wire is 0. As implemented, the function for building the NOT gate takes a list of wire numbers whose values will be negated unconditionally. The function for building the CNOT gate takes the wire number for a single control wire, the wire number for the single target wire, and a flag indicating whether the value of the target is toggled upon a control value of 0 or a control value of 1. The function for building the CCNOT gate takes the wire positions of two control wires and one target wire, and toggles the value on the target wire whenever both control wires have a value of 1.

For a k-wire circuit, LARC produces a $2^k \times 2^k$ permutation matrix that represents the action of the circuit. The input and output values on the wires are represented by 2^k-long vectors, with a single 1 in the position whose binary representation has bits corresponding to the values on the k wires. Although it is not demonstrated in Gate_play, the user could apply the circuit matrix to an input vector by using matrix-vector multiplication. Depending on the method for implementing the circuit in the matrix, applying the circuit to the input corresponds to a multiplication in which the input vector was in row or column format and on the left or right of the matrix respectively. These two different instantiations of the circuit are just transposes of each other. In the current code the circuit matrix was constructed assuming the input vector would be a row vector which would be right-multiplied by a circuit matrix.

In the example in MyPyLARC, we create two circuit matrices in which gates that are supposed to be commutative (i.e., $AB = BA$) are applied in different orders. For simple NOT, CNOT, and CCNOT gate matrices, it is sufficient (but not necessary) to determine whether the matrices for gates A and B commute by checking whether no target wire on A is a control wire on B and vice versa. A set of gates that commute may be placed in a circuit or written in the code in any order. After constructing the two matrices, we verify that they are indeed identical using the simple technique of loading them both into the same MatrixStore/ScalarStore and verifying that they have the same MatrixID. This verification could also be done using the program `LARC/src/python/json_files_matrix_equal.py` on LARC matrix files output from the two circuits.

3.5 Quantum Sycamore Circuit

We start this section with a description of the quantum density matrix formalism which allows a quantum circuit to be expressed as a series of matrix operations. This will echo the description given of reversible circuits above. Then we describe Google's Sycamore circuit and how LARC represents a portion of this circuit and carries out gate operations.

The Quantum Density Matrix Formalism

A quantum circuit diagram can be translated into a sequence of matrix operations in various ways. One of these is the quantum density matrix formalism.

For example, the diagram in Figure 4 shows a small quantum circuit called the teleportation circuit. There is a line for each qubit in the circuit. On the left hand side the initial states of the qubits are given, and time progresses to the right. Various actions occur over time including one and two qubit gates and measurements.

To convert this circuit to a density matrix representation, we will create a matrix which is expressed with respect to a particular basis. We assume that this is the $|0\rangle\langle 0|$, $|1\rangle\langle 1|$ basis in the following discussion. We are using Dirac notation, where $|\rangle$, pronounced ***ket***, indicates a column vector and $\langle|$, pronounced ***bra***, indicates the complex conjugate transpose row vector. So in particular, $|0\rangle$ is represented as the column vector $\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ and thus $|0\rangle\langle 0|$ is the density matrix $\left[\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right]$, and $|1\rangle$ is represented as the column vector $\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$ and thus $|1\rangle\langle 1|$ is the density matrix $\left[\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right]$. A general column vector is expressed as a sum of basis vectors

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}$$

where α and β are complex numbers. To get the corresponding density matrix we take the column vector given by the complex conjugate transpose $\langle\psi| = |\psi\rangle^\dagger$ and calculate the product

$$|\psi\rangle\langle\psi| = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}\begin{bmatrix}\alpha^* & \beta^*\end{bmatrix} = \begin{bmatrix}\alpha\alpha^* & \alpha\beta^*\\ \beta\alpha^* & \beta\beta^*\end{bmatrix}$$

where the superscript $*$ indicates complex conjugate.

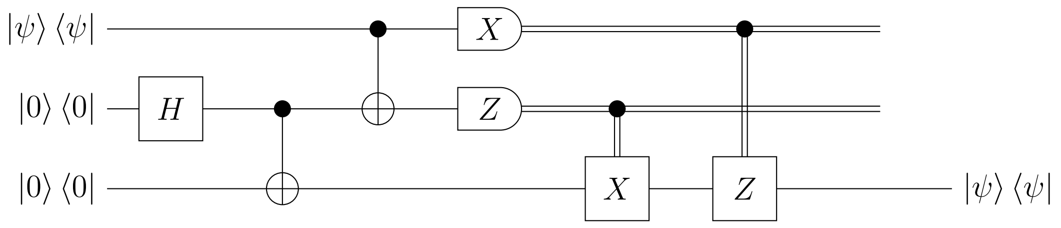


Figure 4: This quantum circuit is called the ***teleportation circuit*** since it transfers the initial state of the top-line qubit $|\psi\rangle\langle\psi|$ (on the left), to the final state of the bottom-line qubit (on the right). This is done by applying a sequence of gates and measurements: a 1-qubit Hadamard gate (H), two 2-qubit controlled `NOT` gates, measurements in the X and Z basis, and two classically controlled 1-qubit gates (X and Z).

To give the quantum density matrix representation of a circuit we will represent at each moment at time the state of the circuit as some 2^k-by-2^k matrix where k is the number of qubits. The action of the gates will also be an operation that involves a gate matrix which is also 2^k-by-2^k and is constructed as the Kronecker product of smaller 2-by-2 single qubit gates, and larger multi-qubit gates (and the identity if no gate occurs on some qubit during that time).

So for example, in the teleportation circuit in Figure 4, the initial state of the circuit is represented by the matrix

$$S_0 = |\psi\rangle\langle\psi| \otimes |0\rangle\langle 0| \otimes |0\rangle\langle 0| = \begin{bmatrix}\alpha\alpha^* & \alpha\beta^*\\ \beta\alpha^* & \beta\beta^*\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\ 0 & 0\end{bmatrix}$$

$$= \begin{bmatrix}\alpha\alpha^* & 0 & 0 & 0 & \alpha\beta^* & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ \beta\alpha^* & 0 & 0 & 0 & \beta\beta^* & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}.$$

Now let's look at the action of the H gate on the initial state of the circuit. A vertical line through the H gate, passes through two wires with no gates as well, and the combination of gates from this ***time slice*** is represented by the gate matrix

$$G_1 = I_1 \otimes H \otimes I_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \end{bmatrix}.$$

Then the action of a gate G_i on some state S_{i-1} produces state S_i by multiplying three matrices together:

$$S_i = G_i \; S_{i-1} \; G_i^\dagger$$

where the † indicates the complex conjugate transpose. People who are interested in how the more complicated gates involving multiple qubits or measurement can refer to [27].

Because the process of creating gate matrices is a Kronecker product of many smaller graphs and because circuits often reuse many basic components, we expect the gate matrices to be naturally compressible in LARC.

The Sycamore Circuit

Google carried out an experiment using the Sycamore quantum hardware [24]. The experiment created a quantum state on their noisy 53-qubit system with the intention of demonstrating quantum supremacy (a problem that can be done on a quantum computer that could not feasibly be performed on a classical computer). Originally they claimed that they could do their computation in 200 seconds compared to an estimated 10,000 years for a "state-of-the art classical supercomputer". However, researchers at IBM Watson responded almost immediately with a description of how this computation could be simulated classically in a few days using large amounts of secondary storage on the Summit supercomputer at Oak Ridge National Laboratory [25]. More recently a group has had success classically simulating this problem using tensor network methods [26].

The Sycamore circuit is built on a grid of 53 qubits. Each qubit can interact with neighbors in each of the cardinal directions. There are only four different gates that make up the circuit: three 1-qubit gates `sqrt_X`, `sqrt_Y`, and `sqrt_W`, where $W = (X + Y)/\sqrt{2}$; and a 2-qubit gate S to provide entanglement. The matrix for the 2-qubit Sycamore Gate is:

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & i & 0 \\ 0 & i & 0 & 0 \\ 0 & 0 & 0 & \omega \end{bmatrix}$$

where $i = \sqrt{-1}$ and $\omega = e^{i\pi/6} = (\sqrt{3} + i)/2$ is the primitive 12-th root of unity. They chose this gate because it most closely matched the behavior they could implement with their hardware.

```
                                 *                |           |
                              *     qubit - A - qubit - B - qubit - A -
                           *          |           |           |
     top edge ---->     *             C           D           C
          |          *                |           |           |
          |       *     qubit - A - qubit - B - qubit - A - qubit - B -
          V    *          |           |           |           |
            *             C           D           C           D
         *                |           |           |           |
      *     qubit - A - qubit - B - qubit - A - qubit - B - qubit - A -
   *          |           |           |           |           |
*             C           D           C           D           C
              |           |           |           |           |
qubit - A - qubit - B - qubit - A - qubit - B - qubit - A - qubit - B -
  |           |           |           |           |           |
```

Figure 5: This diagram shows a portion of the Sycamore circuit (tipped 45 degrees from the figures in [24] to make two-qubit-gate connections horizontal or vertical). The pairing patterns (A, B, C, D) determine which pairs of qubits will have the two-qubit S gate applied to them during a cycle in the Sycamore circuit. During each cycle an interior qubit will be linked by gate S with the one of its four closest neighbors determined by the pairing pattern used in this cycle.

Each ***cycle*** of the Sycamore circuit consists of two steps. Step 1: For each qubit, choose a random 1-qubit gate from the set `sqrt_X, sqrt_Y, sqrt_W`, then apply to all 53 qubits their selected gate. Step 2: Given a pairing pattern A of the qubits with grid neighbors (see Figure 5), apply the 2-qubit gate S to each pair linked in that pattern. During the next cycle, replace the pairing pattern A, with pattern B, and in each subsequent cycle replace the pairing pattern with the next pattern in the sequence (A B C D C D A B). This pattern was selected by Google because they believed that it made the circuit hard to simulate classically. There is a second pattern that Google uses to verify classical simulation results, called (E F G H E F G H).

To explore LARC's abilities on a sample quantum simulation, MyPyLARC/Sycamore_play contains a program demonstrating how one might code a small portion of the Sycamore circuit matrix using LARC. This code can be run using any of the complex types (`COMPLEX`, `MPCOMPLEX`, `MPRATCOMPLEX`) or with type `CLIFFORD` choosing the algebra $\mathbb{Q}(i, \sqrt{2}, \sqrt{3})$.

Description of the implementation of the 2-qubit Sycamore gate in LARC

It is simple enough to load the 4×4 matrix into LARC that corresponds to Sycamore's two-qubit gate S. The four quadrant submatrices of S are:

$$S_A = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \qquad S_B = \begin{bmatrix} 0 & 0 \\ i & 0 \end{bmatrix} \qquad S_C = \begin{bmatrix} 0 & i \\ 0 & 0 \end{bmatrix} \qquad S_D = \begin{bmatrix} 0 & 0 \\ 0 & \omega \end{bmatrix}$$

The challenging part of dealing with the 2-qubit Sycamore gate is embedding it within a larger n-qubit system. This means creating a 2^n by 2^n matrix (i.e., level $n =$ the number of qubits) that behaves like the identity operator on each of the remaining $(n-2)$ qubits.

Let the n qubits be numbered 0, 1, ..., $n-1$. If the two target qubits happen to be conveniently located at j and $j+1$, then the task of constructing the large matrix is simple. One merely takes the Kronecker product of j (2×2)-identity matrices, the S matrix, and then $n-j-2$ more (2×2)-identity matrices (which is $I_j \otimes S \otimes I_{n-j-2}$).

In the general case, where the target qubits are located at non-consecutive j_1 and j_2, the construction proceeds as follows. We build the large matrix iteratively, starting at the scalars (i.e., level 0) and working up one level at a time. Because the 2-qubit Sycamore gate behaves symmetrically with respect to the two target qubits, we can assume, without loss of generality, that $j_1 < j_2$. Then there are j_1 qubits before the first target, $j_2 - j_1 - 1$ qubits between the first and second targets, and $n - j_2 - 1$ qubits after the second target.

There are only four scalars that appear in the final gate; they are 0, 1, i, and ω. So we first construct the following level 0 matrices:

$$Z_0 = \begin{bmatrix} 0 \end{bmatrix} \qquad I_0 = \begin{bmatrix} 1 \end{bmatrix} \qquad J_0 = \begin{bmatrix} i \end{bmatrix} \qquad W_0 = \begin{bmatrix} \omega \end{bmatrix}$$

We next construct the following level k matrices for all levels $0 < k \leq n - j_2 - 1$:

$$Z_k = \left[\begin{array}{c|c} Z_{k-1} & Z_{k-1} \\ \hline Z_{k-1} & Z_{k-1} \end{array}\right] \qquad I_k = \left[\begin{array}{c|c} I_{k-1} & Z_{k-1} \\ \hline Z_{k-1} & I_{k-1} \end{array}\right]$$

$$J_k = \left[\begin{array}{c|c} J_{k-1} & Z_{k-1} \\ \hline Z_{k-1} & J_{k-1} \end{array}\right] \qquad W_k = \left[\begin{array}{c|c} W_{k-1} & Z_{k-1} \\ \hline Z_{k-1} & W_{k-1} \end{array}\right]$$

Mathematically, this is equivalent to:

$$Z_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes Z_{k-1} \qquad I_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes I_{k-1}$$

$$J_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes J_{k-1} \qquad W_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes W_{k-1}$$

Note: We will continue to need the Z_k matrices for all levels $0 \leq k < n$.

We next construct the four following level $n-j_2$ matrices, corresponding to the four quadrant submatrices S_A, S_B, S_C, and S_D of the S matrix above:

$$A_{n-j_2} = \left[\begin{array}{c|c} I_{n-j_2-1} & Z_{n-j_2-1} \\ \hline Z_{n-j_2-1} & Z_{n-j_2-1} \end{array}\right] \qquad B_{n-j_2} = \left[\begin{array}{c|c} Z_{n-j_2-1} & Z_{n-j_2-1} \\ \hline J_{n-j_2-1} & Z_{n-j_2-1} \end{array}\right]$$

$$C_{n-j_2} = \left[\begin{array}{c|c} Z_{n-j_2-1} & J_{n-j_2-1} \\ \hline Z_{n-j_2-1} & Z_{n-j_2-1} \end{array}\right] \qquad D_{n-j_2} = \left[\begin{array}{c|c} Z_{n-j_2-1} & Z_{n-j_2-1} \\ \hline Z_{n-j_2-1} & W_{n-j_2-1} \end{array}\right]$$

We next construct the following level k matrices for all levels $n - j_2 < k \leq n - j_1 - 1$:

$$A_k = \left[\begin{array}{c|c} A_{k-1} & Z_{k-1} \\ \hline Z_{k-1} & A_{k-1} \end{array}\right] \qquad B_k = \left[\begin{array}{c|c} B_{k-1} & Z_{k-1} \\ \hline Z_{k-1} & B_{k-1} \end{array}\right]$$

$$C_k = \left[\begin{array}{c|c} C_{k-1} & Z_{k-1} \\ \hline Z_{k-1} & C_{k-1} \end{array}\right] \qquad D_k = \left[\begin{array}{c|c} D_{k-1} & Z_{k-1} \\ \hline Z_{k-1} & D_{k-1} \end{array}\right]$$

Again, this is mathematically equivalent to:

$$A_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes A_{k-1} \qquad B_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes B_{k-1}$$

$$C_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes C_{k-1} \qquad D_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes D_{k-1}$$

We next construct the following level $n - j_1$ matrix:

$$M_{n-j_1} = \left[\begin{array}{c|c} A_{n-j_1-1} & B_{n-j_1-1} \\ \hline C_{n-j_1-1} & D_{n-j_1-1} \end{array}\right]$$

Finally, we construct the following level k matrix for all levels $n - j_1 < k \leq n$:

$$M_k = \left[\begin{array}{c|c} M_{k-1} & Z_{k-1} \\ \hline Z_{k-1} & M_{k-1} \end{array}\right] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes M_{k-1}$$

Then M_n is the desired 2^n by 2^n gate matrix which applies the 2-qubit Sycamore gate S to the pair of qubits with indices j_1 and j_2 while leaving the other qubits unaffected.

3.6 MyPyLARC Tutorial Routines and Project Directories

MyPyLARC consists of:

- a README.md file with some basic information;
- a Tutorial directory MyPyLARC/Tutorial;
- several example project directories (MyPyLARC/Count_Triangles, Eigen_play, FFT_play, Gate_play, Hash_play Roots_play,

- a src directory with C code that is used by the projects Gate_play
- a Makefile that integrates C and Python code for LARC and MyPyLARC,
- a project template that can be copied and modified, 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_method.py 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.

Here is the Tutorial/README file:

```
Contents of Directory MyPyLARC/Tutorial
=======================================
circulant.py
   This sample program creates a random circulant matrix.  These matrices
   compress well within LARC.

create_a_project.txt
   Gives tips to the a user for creating their own project directory
   and getting started writing their own application.

create_larc_matrix_compression_graph.ipynb
   This is a sample jupyter notebook routine that uses matplotlib
   to create a graph showing the LARC compression for various
   families of matrices.  A .png file of the output graph is placed
   in the Data/Out subdirectory.

Data
  A subdirectory containing two subdirectories (In and Out) which
  have respectively input data, and output data for various routines
  in the Tutorial directory.

examples_cleaning.py
  This sample program illustrates removing (i.e., cleaning) a matrix
  from the matrix store after it is no longer needed.

examples_infoStore1.py
  This sample program shows how to add metadata to the information
  store and associated to a particular matrix.

examples_infoStore2.py
```

```
  This sample program shows how to read metadata from the information
  store.

examples_localHash.py
  This sample program shows the effects of LARC's locality-sensitive hash.

examples_logging.py
  This sample program shows how to use logs to record the results of
  a run.  With these commands the user would be able to save off input,
  parameters, and results of experiments.

examples_math.py
  This routine illustrates various matrix operations on both square
  and non-square matrices.

examples_precision.py
  This routine tests for potential loss of precision caused by LARC's
  snap to region representative scheme (and locality sensitive hashing).
  It generates the 8th roots of unity, then checks to see if the
  square of w, where  w = 2nd power of the 8th root of unity
  is equal to i.

examples_roots_unity.py
  This routine generates the 24th roots of unity w, and then checks
  for the appropriate closure under multiplication of the various
  powers of w.  That is, it verifies that
     matrixProduct(matrixID(w^i),matrixID(w^j)) = matrixID(w^k)
  where k congruent to i+j mod 24.

example_unittest_matrix_store.py
  This contains unittests for various matrix building and access operations.
  To use the python unittest module and run this code type:
    python3 -m unittest example_unittest_matrix_store.py

example_unittest_op_store.py
  This contains sample set of unittests for the operations store of LARC.
  To use the python unittest module and run this code type:
    python3 -m unittest example_unittest_op_store.py

example_unittest_scalars.py
  This contains sample set of unittests for scalars in LARC.
  To use the python unittest module and run this code type:
    python3 -m unittest example_unittest_scalars.py

fft_nonParamFile_init.py
```

```
  This illustrates how to initialize LARC with parameters selected
  by the user.  The routine then has an example where matrices
  are generated that have components in a block sparse representation
  of the discrete Fourier transform.

fft_paramFile_init.py
  This illustrates how to initialize LARC by running code that checks
  the available memory then selecting an appropriate set of parameters
  from a stored file to use in initializing LARC.  The routine then
  has an example where matrices are generated that have components in
  a block sparse representation of the discrete Fourier transform.

fft_toeplitz.py
  This routine illustrates how two matrices which are each somewhat
  compressible in LARC behave when multiplied.  The matrices used are
  a permutation submatrices that we had from our block sparse
  fast Fourier transform (FFT), and a random Toeplitz matrix.
  The LARCsize of the product is returned.

newuser_instructions
  Advice on getting started with LARC and MyPyLARC including a
  suggested order to try the tutorials and compiling with different
  scalar types.

preloading_mults_pi.py
  This routine illustrates the way that region size for the locality
  sensitive hash is used by LARC to minimize the number of scalars
  that are creating by representing any scalar in a small region by
  a single representative.

sample_larc_initialization_and_use.ipynb
  This is a jupyter notebook file that shows how to initialize
  larc and to run a few computations.

test_block_diagonal.py
  This program creates a matrix and returns the matrixIDs of the block
  diagonal submatrices of that matrix.  It does testing for various
  error conditions.

test_hashstats.py
  This lets the user see which statistics are collected when the LARC
  src/larc.h has been modified to have the #define HASHSTATS uncommented.
  The program repeats various matrix operations in LARC and then gives
  the report on the hashchain lengths and other statistics.
```

```
test_kronecker.py
  This program creates kronecker products of various combinations of
  matrices, row vectors, and column vectors.

test_unittest_matmath.py
  This is a collection of unit tests for the various matrix functions
  in larc/src/matmath.c.
  See also working_example_unittest.py

toeplitz.py
  This routine creates a random Toeplitz matrix of level 3.

tut1_larc_overview.py
  This program illustrates LARC initialization and some basic LARC operations.

tut2_reporting.py
  This program prints out the matrix store report and the operations
  store report and the rusage report.  This runs fine, but we need to
  add some explanatory print statements.

tut4_matrix_build_and_io.py
  This code tests some basic matrix reading and building routines.

user_proj_matrix_symmetry.py
  This routine checks to see if a matrix is close to being symmetric
  by taking the difference between a matrix and its transpose
  and seeing how close that result is to the zero matrix.

user_proj_template.py
  This is the skeleton of a routine that we eventually hope can be
  used as a template for new code.  Right now, don't use it and copy
  some routine that already works as your starting place.
  See suggestions in create_a_project.txt

working_example_unittest.py
  This contains an example unit test, see also test_unittest_matmath.py
```

4 Code and Documentation Availability

The LARC linear algebra via recursive compression package and the MyPyLARC tutorial and sample application package are available publicly in two GitHub repositories LARC and MyPyLARC found at `https://github.com/LARCmath`. They may be used under an open source license agreement (see Appendix D).

LARC is written primarily in C and MyPyLARC is written primarily in Python. LARC has been built and tested under a variety of Linux platforms. LARC specifically requires the `gcc` compiler, and would require source-level modifications to work with other compilers. Beyond the standard build tools (e.g., `make`, `curses`, and `git`), other software dependencies for LARC are: Python 3.6 or later, GMP 5.0 or later, MPFR 4.0.0 or later, MPC 1.1.0 or later, SWIG 3.0.0 or later, and Doxygen 1.8.13 or later. Specific build directions and local makefile customizations are provided for the CentOS 8 and Ubuntu 18.04 platforms. In particular, when building LARC or MyPyLARC for the first time, you will be prompted to copy or create a local makefile customization file.

All LARC C routines have doxygen html documentation which is placed in the user's copy of the LARC/html directory upon compilation and can be viewed in a browser. Additional documentation exists in the README, the Tutorial directory, and the doc directory. Various LARC documents can be found in the MyPyLARC/doc directory including this paper, a poster describing LARC from SIAM AN 2020, and slides for a talk from SIAM LA 2021. There is also an earlier introduction to LARC in the LARC/doc subdirectory from 2015.

LARC is not currently optimized for memory or speed.

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 receive bug reports. If you want to contact the developers at CCS you can send email to larc@super.org.

5 Open Research Topics

We include short descriptions of several open research topics which we hope might appeal to other academic researchers, graduate students, and linear algebra package users.

5.1 Rayleigh Dispersion for Seismic Events

With Rayleigh dispersion, the longer wavelengths tend to go deeper underground giving them a straighter path between two points on the Earth's surface and an earlier arrival time than the higher frequency surface waves. The composition of different layers of the Earth affect the Rayleigh dispersion. Finding a Rayleigh dispersion model from a seismic event leads to a mathematical problem with a sequence of heptadiagonal (diagonal, three superdiagonals and three subdiagonals) stiffness matrices [19, 20]. In these papers, the authors look for the change point in which the determinant of this sequence of matrices changes sign. Since it is not a continuous sequence they can not solve for the point at which the matrix becomes singular (and its determinant becomes zero). When they find this change point it determines the velocities for the Rayleigh dispersion model. The authors use publicly available data collected from an actual seismic event. This problem might make an interesting project to program in LARC as heptadiagonal matrices are reasonably compressible in the LARC format (see Appendix C).

5.2 Hybrid Lanczos-Power Method Eigensolver

The Lanczos method is used to find the eigenvalues of large sparse real matrices. There is a block version of the Lanczos method (this is not to be confused with Montgomery's "Block Lanczos" which finds null spaces of a matrix over a finite field). The adaptive block method for the Lanczos eigensolver is called the ABLE Method [21, 22, 23].

In MyPyLARC/Eigen_play we have LARC code to find the largest magnitude eigenvalue using the power method. We have considered creating a hybrid method which would combine the block version of the Lanczos eigensolver and the power method. Lanczos could be used to make a smaller-than-full-size tridiagonal matrix, then the power method could be used on the smaller matrix to find a good approximation to its maximum eigenvalue.

5.3 Iterative Methods: Changing Region Width for Approximation Methods

The granularity of snapping is determined by the region width selected by the user. Large regions lead to a rough granularity with fewer scalars in the store. Running an algorithm on LARC with large regions may give an approximate solution to a problem with minimal computing resources. Once the approximate solution is available, it might be possible to use this as input to the same or a different program run on LARC with a smaller region width, thus obtaining a more accurate solution.

5.4 Modifying Scalar Values in the ScalarStore

LARC code currently fixes a scalar in the ScalarStore and snaps nearby values to the previously stored values. This may not be ideal for iterative methods in which the best known value for some quantity is expected to improve over time. The LARC code could be modified, so that there is an option to replace old stored scalar values with new "improved" values.

5.5 Upper and Lower Bounding of Probability Calculations

We have started experimenting with a new scalar type that would help us make approximate calculations during multiplication of stochastic matrices. Stochastic matrices (also called: probability matrices, transition matrices, substitution matrices or Markov matrices) are used to describe transitions in a Markov chain.

This new scalar type would approximate a probability value (inclusively between 0 and 1) by a sum of a small number of inverse powers of two, and represent that scalar by tracking the exponents. The idea is easiest to explain using an example. Say we want to approximate each scalar as a sum of no more than three inverse powers of two. For example, we might have a scalar $a = \frac{5}{8} = \frac{1}{2^1} + \frac{1}{2^3}$ and write its exponent representation as $(1, 3, \mathtt{NULL})$, and a scalar $b = \frac{11}{32} = \frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^5}$ with representation $(2, 4, 5)$.

Matrix multiplication requires that we define an addition and multiplication for our new scalar type. These operations will need to maintain our 3-term exponent limit and also the property that the scalar results remain within the $[0, 1]$ interval. An exact multiplication of our a and b above produces a result with more than 3 terms: $a \times b = (1, 3, \mathtt{NULL}) \times (2, 4, 5) = (\frac{1}{2^1} + \frac{1}{2^3}) \times (\frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^5}) = (\frac{1}{2^3} + \frac{1}{2^5} + \frac{1}{2^6} + \frac{1}{2^5} + \frac{1}{2^7} + \frac{1}{2^8}) = c_{\mathtt{exact}}$. This has exponents $(3, 5, 6, 5, 7, 8)$ before gathering or sorting the terms, and then after sorting (smallest on the left) and combining exponents we have $c_{\mathtt{exact}} = (3, 4, 6, 7, 8)$. As we are limiting the number of exponents to three for this scalar type, we need to approximate $c_{\mathtt{exact}}$.

First assume that we are trying to get a lower bound on our probability computations. In this case, the approximation to $c_{\mathtt{exact}}$ is accomplished by truncating after three exponents giving us $c_{\mathtt{lower}} = (3, 4, 6)$. If instead we wanted upper bounds on our probability computations, as before we calculate $c_{\mathtt{exact}}$, sorting and combining exponents as needed, then the approximation to our result is accomplished by retaining the numerically smallest three exponents, decrementing the largest of these if any exponents were dropped, and combining terms if needed. Thus in our example $c_{\mathtt{exact}}$ is replaced by the upper bound $c_{\mathtt{upper}} = (3, 4, 5)$.

It is interesting to note that the lower bounding process is more accurate in general than upper bounding. Given an exact scalar specified by four or more exponents $a_{exact} = (a_1, a_2, a_3, a_4, ...)$ the inaccuracy caused by lower bounding is no worse than $\frac{1}{2^{(a_4-1)}}$, whereas, the inaccuracy caused by upper bounding can be arbitrarily close to $\frac{1}{2^{(a_3)}}$.

Also note, that when an upper bounding addition operation is being carried out, we need to enforce the rule that no probability can be greater than 1. For example, if during a previous round operations the exact values were $a_{\mathtt{exact}} = (2, 3, 4, 5)$ and $b_{\mathtt{exact}} = (1, 5, \mathtt{NULL})$, they would be rounded up to get $a_{\mathtt{upper}} = (1, \mathtt{NULL}, \mathtt{NULL})$ and $b_{\mathtt{upper}} = (1, 5, \mathtt{NULL})$. Then when these upper-bounded terms are added, their sum is greater than 1, so we replace this sum with 1.

The extreme value 1 cannot be represented in the simple list-of-exponents notation. We think that the cases when the scalar is 0 or 1 are most easily handled by keeping flag bits `is_zero` and `is_one` and leaving all the exponents `NULL`. Flag bits let us quickly implement multiplicative and additive identities.

5.6 Matrix Inverses

Calculating the multiplicative inverse of a square matrix has some complexities in LARC. A sparse or otherwise easily compressible matrix rarely has an easily compressible inverse. This problem is magnified when using iterative inverse methods with LARC, because several versions of the matrix must be handled before we are finished.

Standard Gaussian elimination is not well suited to LARC because it involves rows and pivoting instead of a block recursive algorithm.

At first glance, block recursive techniques that involve Schur complements look like they might be an easy win for LARC. However, this method for calculating an inverse works only when the quadrant submatrices satisfy special conditions. It is possible that a matrix may be invertible, but can not be inverted using the Schur complement method. The current version of LARC has no structure to track all these possible cases.

The approximation method we are most interested in implementing would be to estimate the matrix inverse by using a truncated Taylor series which depends on the identity $(I - A)^{-1} = I + A + A^2 + A^3 \cdots$ for matrices A and identity I, where $\det(A) < 1$. Hence the inverse $M^{-1} = I + (I - M) + (I - M)^2 + (I - M)^3 \cdots$ can be estimated when $\det(I - M) < 1$ by truncating this sequence.

5.7 Kronecker Graphs and Triangle Counting

As we mentioned in Section 3.2, researchers testing their triangle counting code have used self-generated Kronecker graphs to mimic networks. Kronecker graphs should be easy to generate with LARC since it already has the Kronecker product coded. If the original incidence matrix K is not dense, then the resulting large Kronecker graph matrix $K^{\otimes k}$ is very compressible in LARC. After sample graphs are generated, the triangle counting code in MyPyLARC/Count_triangles could be applied and the results compared with the known formula for the number of triangles in a Kronecker graph. (See [15] for more information about modeling networks with Kronecker graphs.)

5.8 LARC with BLAS

There are several popular linear algebra tools with large user communities. One such tool is BLAS (Basic Linear Algebra Subprograms). It might be useful to the larger community to create a BLAS interface to allow access to LARC's functionality (e.g. recursive storage, recursive operations, regional representative retrieval, and conversion from sparse to recursive storage). Similarly, it might be helpful to LARC if it were able to call BLAS routines.

6 Acknowledgments

The majority of the current code in LARC and MyPyLARC is co-written by Steve Cuccaro and Jenny Zito. Mark Pleszkoch joined the CCS research staff and the LARC team in 2020, after previously contributing to LARC application code as a summer visitor. He has made significant contributions to the latest release. There are also 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/doc/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].

Particular thanks to Richard Bates for his alpha testing of parts of our package as a "newcomer" and for his very helpful contributions to writing the paper.

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 `https://github.com/LARCmath/LARC/doc/aboutLARC.pdf`.

[4] G.C. Danielson, C. Lanczos, "Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids", *J. Franklin Inst.*, vol. 233, (1942): 365-380, 435-452.

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

[6] V. Kolchin et.al, *Random Allocations*, V.H. Winston, 1978.

[7] J. Leskovec, A. Rajaraman and J. Ullman, "Mining of Massive Datasets, Chapter 3", `http://www.mmds.org`.

[8] V. Strassen, "Gaussian elimination is not optimal", *Numer. Math.* 13 (1969): 354–356.

[9] C. Van Loan, "Computational Frameworks for the Fast Fourier Transform," *Frontiers in Applied Mathematics Vol. 10*, SIAM, 1992.

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

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

[12] 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.

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

[14] J. Kepner et al., "Design, Generation, and Validation of Extreme Scale Power-Law Graphs", `https://arxiv.org/pdf/1803.01281.pdf`.

[15] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker Graphs: An Approach to Modeling Networks", *Journal of Machine Learning Research* 11 (2010): 985–1042;
`http://jmlr.csail.mit.edu/papers/volume11/leskovec10a/leskovec10a.pdf`.
For referencing: arXiv:1805.09675

[16] R. Pearce, T. Steil, B.W. Priest, and G. Sanders, *One quadrillion triangles queried on one million processors*, IEEE High Performance Extreme Computing Conference (HPEC), 2019, 1–5.

[17] S. Samsi et al, "GraphChallenge.org: Raising the Bar on Graph Analytic Performance", in *2018 IEEE High Performance Extreme Computing Conference (HPEC)*, 2018; http://dx.doi.org/10.1109/HPEC.2018.8547527.

[18] S. Samsi et al, "GraphChallenge.org Triangle Counting Performance", https://arXiv.org/2003.09269.

[19] E. Olafsdottir, S. Erlingsson and B. Bessason, "Tool for analysis of multichannel analysis of surface waves (MASW) field data and evaluation of shear wave velocity profiles of soils", *Canadian Geotechnical Journal* 55 (2), (2018): 217–233; https://doi.org/10.1139/cgj-2016-0302.

[20] E. Kausel and J. Roësset, "Stiffness matrices for layered soils", *Bulletin of the Seismological Society of America* 71 (6) 1981: 1743–1761. https://pubs.geoscienceworld.org/bssa/article-pdf/71/6/1743/2703662/BSSA0710061743.pdf.

[21] J. Demmel, *Applied Numerical Linear Algebra*, Society for Industrial and Applied Mathematics, 1997; https://epubs.siam.org/doi/pdf/10.1137/1.9781611971446.

[22] Z. Bai and D. Day *Block Lanczos Methods*, http://www.netlib.org/utk/people/JackDongarra/etemplates/node250.html.

[23] Z. Bai, D. Day and Q. Ye, "ABLE: An Adaptive Block Lanczos Method for Non-Hermitian Eigenvalue Problems", *SIAM J. Matrix Anal. Appl.* 20 (1999): 1060–1082.

[24] F. Arute, K. Arya, . . . J. Martinis, *Quantum supremacy using a programmable superconducting processor*, *Nature* 574 (2019): 505–510.

[25] E. Pednault, J. Gunnels, G. Nannicini, L. Horesh, R. Wisnieff, "Leveraging Secondary Storage to Simulate Deep 54-qubit Sycamore Circuits", https://arxiv.org/abs/1910.09534.

[26] Feng Pan and Pan Zhang, "Simulating the Sycamore quantum supremacy circuits", https://arxiv.org/abs/2103.03074.

[27] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information* (10th ed.), Cambridge University Press, 2010.

A Details of Regional Representative Retrieval

A.1 Introduction

As described in Section 2.4, LARC divides the space of scalars into ***regions*** and stores at most one ***representative*** scalar for each region. We call this method Regional Representative Retrieval. When a new scalar is calculated, if there is already a scalar that has the same or a nearly identical value, LARC would ideally not store the new scalar and instead use the previously stored scalar.

LARC has two regional representative retrieval modes:

- SPR (Single-tile Probabilistic Retrieval) mode and
- MAR (Multi-tile Assured Retrieval) mode.

For both of these modes, the size of the regions are determined by parameters passed to LARC in the initialization call.

The motivation for regional representative retrieval is to mimic symbolic or exact computation. Our technique involves preloading scalars associated with certain mathematical identities and causing "nearby" floating point calculation results to "snap" to prestored values.

SPR mode will succeed probabilistically in snapping to the desired prestored scalar. Failures occur when the closest previously stored scalar is on the other side of a region boundary from the newly calculated scalar.

For MAR mode we will be able to guarantee successfully snapping under certain restrictions. This is done by creating regions in such a way that the regional representative lies in the central area, so that newly calculated scalars that fall near the representative will be in its region. The initialization parameter must be chosen so that the snapping distance is greater than the expected error due to floating point computations. The snapping distance must also be small enough, and the preloaded scalars sufficiently separated from each other, so that desired distinct scalars do not coalesce.

Both SPR and MAR modes utilize a scalar hash table storage that allows fast location of close-by previously stored scalars. We explain how LARC uses locality sensitive hashing for our representive retrieval from the ScalarStore hash table in the next section. Then we will describe the details of SPR mode and MAR mode.

A.2 Fast Hash Table Access

The ScalarStore is implemented with a hash table whose size is a power of 2 determined by an initialization parameter. For example, if the ScalarStore hash table size exponent is set to be 30, there will 2^{30} different hash values which index into the hash table. The fundamental hash function used by LARC is a multiplicative hash [2] which takes 64-bit integers as input and output. LARC's hash is a Fibonacci hash (see Knuth [5]), which uses the golden mean as the multiplicative constant in order to achieve the optimally even distribution of entries in the hash table.

We have built hashes for the various ScalarTypes on top of our Fibonacci hash. For example, to implement the hash for multiprecision real, in which the bits are subdivided into 64-bit "limbs", LARC hashes a 64-bit limb, mod 2 adds this to the next limb, then hashes that sum. It continues to add each new limb to the output of the last hash and then hash the result until it has used up all the limbs. For floating point types, we ensure that we take advantage of the full entropy by converting the mantissa, sign, and exponent bits to an integer representation. The final step of all these hashes is to reduce the number of output bits from the hash so that it is equal to the hash table size exponent (e.g. 30 bits).

We use chaining [2] at each hash index to handle hash collisions. That is, for each fixed hash value (index in the hash table) we create a linked list of all records that were stored with that hash value. When searching for a record, one must check some information in the record itself to confirm that the correct record has been found.

LARC divides space into ***tiles*** and will never save more than one scalar from each tile. We can quickly check the ScalarStore for a previously stored scalar from a given tile if we send all scalars in that tile to the same hash chain (same hash value). This is done by using a locality-sensitive hash which works as follows: given a new scalar, LARC determines the tile in which the scalar lies, calculates the unique label for this tile and then hashes the ***tile label*** to determine the index into the hash table. In SPR mode, the tile label will be the center point of the tile, and in MAR mode, the tile label will be an integer (or pair of integers for complex types) which is described in Subsection A.4 on MAR mode.

As noted before, a LARC region always has at most one representative stored in the ScalarStore. In SPR mode, a region consists of a single tile. In MAR mode, each region may be composed of up to four tiles (for complex scalar types). When a region is composed of multiple tiles, we cannot expect each scalar in that region to hash to the same hash chain, since the tiles of the region are likely to end up in different hash chains. Therefore MAR mode will need an algorithm which finds the representative of a multi-tile region. This algorithm and other details of MAR and SPR modes are given in the following sections.

A.3 Single-tile Probabilistic Retrieval (SPR)

For SPR mode, we use the terms tile and region interchangeably, since each region consists of a single tile. When the user has a new scalar s, LARC needs to determine whether the ScalarStore hash table already contains that scalar or another scalar t which is sufficiently close to s to use instead. To accomplish this, LARC does the following.

1. **Compute the tile label for s.** In SPR mode this is the center of the tile (region) in which it lies, `SPR_tile_center`(s). Tile labels are unique.
2. **Determine which hash chain in the ScalarStore to search for a ScalarRecord for this region.** The index into the ScalarStore is the value `SPRhash`(s) which is given

by the standard LARC hash for this scalar type applied to the tile label:

$$\texttt{SPRhash}(s) = \texttt{Hash(SPR\_tile\_center}(s)\texttt{)}.$$

3. **Traverse the hash chain looking for a ScalarRecord for this tile.** Each such record contains a scalar t, and LARC checks whether s and t lie in the same region (tile) by determining whether their tile labels are the same.
   - If t is in the same region as s, then either it is equal to s or it is close enough to s that LARC will use it instead of s from now on (we say s snaps to t). LARC returns the stored MatrixID(t).
   - If t is not in the same region as s, continue traversing the hash chain.
4. **If no ScalarRecord exists for this region then add one.** If the end of the hash chain is reached without finding a scalar in the same region as s, LARC needs to add s to the ScalarStore. LARC creates a new ScalarRecord for s, adds the record to this hash chain, and returns the newly assigned MatrixID(s). The scalar s is now the unique representative of its region.

The location and widths of the SPR regions depend on two parameters that are passed at initialization of LARC, `regionbitparam` and `zeroregionbitparam`. In the following discussion we will abbreviate these as `rb` and `zrb` respectively.

Most regions are of width
$$w = 1/2^{\texttt{rb}}$$
and are centered about multiples of w. Users have the option to make the SPR region containing zero[9] larger than the standard regions, by setting the parameter `zrb` to be smaller than `rb`. When this is the case, the width of the region containing zero is given by the formula
$$w_z = 1/2^{\texttt{zrb}} - 1/2^{\texttt{rb}}.$$
This value approaches $1/2^{\texttt{zrb}}$ when `zrb` is very much smaller than `rb`. The reason for the slightly-odd formula is to make the region boundaries of the different-sized regions line up, and to maintain the property that all regions away from zero have the same size.

Here are a few examples setting $k = \texttt{rb} - \texttt{zrb}$. For $k = 1$, $w_z = w$; for $k = 2$, $w_z = 3w$; for $k = 3$, $w_z = 7w$; and for $k = 4$, $w_z = 15w$. In general for $k > 0$ the formula is $w_z = (2^k - 1)w$. When $k = 0$ LARC simply sets $w_z = w$. We have assumed that it never makes sense to have the regions around zero be smaller than other regions, so if the user (accidentally?) enters initialization parameters with the value of `zrb` set greater or equal to `rb`, then LARC ignores the `zrb` parameter and uses `rb` to calculate the width of all regions.

When scalars are any of the complex types, the region calculation occurs independently for real and imaginary parts. This means that when $w_z > w$, regions along the axes other than the origin are non-square rectangles; the rectangles are w wide along the axis and w_z wide perpendicular to the axis. To get default values for the region width parameters, the user

[9] Remember that LARC prestores zero, so that it is guaranteed to be the representative of its region.

```
..........  ..........  ................  ..........  ..........
.         | .         | .              . |         . |         .
. -3w+3wI | . -2w+3wI | .   +0w+3wI    . |  2w+3wI . |  3w+3wI .
.         | .         | .              . |         . |         .
._________| ._________| .______________. |_________. |_________.
..........  ..........  ................  ..........  ..........
.         | .         | .              . |         . |         .
. -3w+2wI | . -2w+2wI | .   +0w+2wI    . |  2w+2wI . |  3w+2wI .
.         | .         | .              . |         . |         .
._________| ._________| .______________. |_________. |_________.
..........  ..........  ................  ..........  ..........
.         | .         | .              . |         . |         .
.         | .         | .              . |         . |         .
.         | .         | .              . |         . |         .
. -3w+0wI | . -2w+0wI | .   +0w+0wI    . |  2w+0wI . |  3w+0wI .
.         | .         | .              . |         . |         .
.         | .         | .              . |         . |         .
.         | .         | .              . |         . |         .
..........| ..........| ................ |.......... |..........
._________  ._________  ________________  _________.  _________.
.         | .         | .              . |         . |         .
. -3w-2wI | . -2w-2wI | .   +0w-2wI    . |  2w-2wI . |  3w-2wI .
.         | .         | .              . |         . |         .
..........| ..........| ................ |.......... |..........
___________ ___________ ________________ __________. __________.
.         | .         | .              . |         . |         .
. -3w-3wI | . -2w-3wI | .   +0w-3wI    . |  2w-3wI . |  3w-3wI .
.         | .         | .              . |         . |         .
..........| ..........| ................ |.......... |..........
```

Figure 6: A diagram of the SPR regions near the origin for the case $k = 2$. Each region is labeled by its center point, in which $\mathtt{I} = \sqrt{-1}$. The center column along the imaginary axis will be three times the width of the other columns and the center row along the real axis is is three times the width of the other rows. Dotted lines indicate non-inclusive boundaries.

can enter -1 for either or both. The default value for `zrb` is to set it equal to `rb`, making all regions equal in size. The default for `rb` depends on the scalar type selected during compilation. For example, if LARC is compiled using C floating-point long double, then the default for `rb` is set to 56, eight bits less than machine precision for C long double.

Every point in scalar space belongs to exactly one region, so we must decide how to assign scalars that lie along region boundaries. For non-complex scalar spaces the region about zero is an interval `( , )` which does not include the boundaries. For regions on the positive side of zero we write the region `[ , )`, and for regions on the negative side of zero we write the region `( , ]`, to indicate that these regions include the boundary closer to zero and do not include the boundary further from zero. For complex scalarTypes, the region boundaries are marking off areas of the complex plane. The region around the origin includes none of its four boundaries. Regions along either axis include only the boundary closer to the origin. Regions away from both axes include the two boundaries closer to the axes and exclude the other two boundaries. Corner points of regions belong to the unique region which has two inclusive boundaries meeting at that corner. See Figure 6.

SPR is successful in collapsing a new scalar to a close-by previously stored scalar only if the two scalars lie in the same region, but will fail to find a nearby scalar if the two scalars lie on opposite sides of a region boundary. This means that SPR is only successful in snapping to a previously stored near-by scalar value with high probability. If one becomes aware that an important scalar lies close to a region boundary in SPR mode, and hence is not getting as many snaps as desired, it may help to change the parameters `rb` and `zrb`. SPR region boundaries shift into different positions whenever these parameters change. For example if `rb` = `zrb` and we look at `rb` $= 2, 3, 4$ the locations of the first few region boundaries greater than zero are: `rb` $= 2 : \{1/8, 3/8, 5/8, \ldots\}$; `rb` $= 3 : \{1/16, 3/16, 5/16, \ldots\}$; and `rb` $= 4 : \{1/32, 3/32, 5/32, \ldots\}$. Thus a scalar that lies close to a region boundary with one value of `rb` may lie near to the center of a region when `rb` is decreased or increased.

One also might try MAR mode as an alternative if scalars are failing to snap to the desired values in SPR mode. In MAR mode, there is guaranteed snapping to some previously-stored sufficiently-close scalar.

A.4 Multi-tile Assured Retrieval (MAR)

For MAR mode we make a distinction between regions and tiles. A MAR region is formed by one or more adjacent tiles:

- a ***primary tile***, which contains the unique scalar representative, and
- possibly some ***neighbor tiles*** that are adjacent to the primary tile.

After a primary tile is created, MAR attempts to extend the region by *claiming* desired neighboring tiles. For real scalar types MAR mode regions are generally composed of two adjacent tiles, the primary and one neighbor; for complex scalar types regions are generally a two-by-two square of adjacent tiles, the primary and three neighbors. However, a primary tile can only claim as neighbor tiles those tiles which do not already belong to another MAR region in the ScalarStore. Thus it is possible that some MAR regions for real types will only have one tile instead of the typical 2-tile shape; the MAR regions for complex types may have one, two, or three tiles instead of the typical 4-tile block.

MAR mode uses only one tile size parameter, regionbitparam (`rb`). All tiles are $1/2^{\texttt{rb}}$ wide. (MAR always sets zeroregionbitparam = regionbitparam.) MAR tiles have inclusive boundaries on the edges closer to negative infinity in both the real and imaginary directions.

LARC only stores a single scalar from each region in the ScalarStore; this scalar lies in the primary tile and is called the regional representative. MAR will attempt to claim adjacent neighbor tiles to add to the region in such a way that the regional representative will lie in the central portion of the multi-tile region. Specifically, the neighbor tiles are selected with the goal that the scalar will be at least 1/4 of the region width from all region boundaries. If MAR is able to claim all the desired neighbor regions, then the corner of the primary tile which is closest to the region representative will be the center point of the region.

Here is an example of the neighbor tile selection process. If a complex scalar lies in the North-East quarter of the primary tile, then MAR would attempt to claim the three tiles to its North, East, and North-East as the neighbor tiles in its region. MAR is greedy, in the sense that it will try to claim neighbor tiles to put in the region with the primary tile, but will only be able to claim neighbor tiles that have not yet been claimed as primary or neighbor tiles of other regions. Thus, although MAR tiles are all equal sized, some MAR regions may not have as many tiles as other regions if they were formed after another region claimed the neighbor tiles that they desire. Various concepts we have discussed concerning MAR tiles and regions are illustrated in Figure 7.

When the user has a new scalar s, LARC needs to determine whether the ScalarStore hash table already contains that scalar or another scalar t which is sufficiently close to s to use instead. To accomplish this in MAR mode, LARC does the following.

1. **Compute the tile label.** In MAR mode LARC calculates the tile label for s by multiplying the scalar by $2^{\texttt{rb}}$ and truncating the result to a multiprecise integer (right shift by `rb` and truncate fractional portion). For complex scalar types this is a pair of multiprecise integers. The tile label is called `MAR_tile_index`(s).

2. **Determine which ScalarStore hash chain to search.** The hash function `MARhash`(s) is calculated by applying the standard LARC hash for multiprecision integers to the tile label
$$\texttt{MARhash}(s) = \texttt{Hash(MAR\_tile\_index}(s)\texttt{)}.$$
The `MARhash`(s) is the index into the ScalarStore hash chain.

3. **Traverse the hash chain looking for a ScalarRecord with a matching tile label.** This record may correspond to either a primary tile or a neighbor tile for some region. There are three things that may happen.

   A. **LARC finds a ScalarRecord for a primary tile with a matching tile label.** In this case, LARC returns the MatrixID of the scalar from the primary tile. The scalar t in this primary tile is either equal to s or very close, and returning the MatrixID of t will cause s to snap to t.

   B. **LARC finds a ScalarRecord for a neighbor tile with a matching tile label.** In this case, the record contains a link that gives the location of the ScalarRecord of the primary tile for this region, and LARC returns the MatrixID of the scalar from the primary tile's ScalarRecord. The scalar t in this primary tile will not be equal to s since they lie in different tiles, but s is in the small region that has t as its unique representative, and s will snap to t.

   C. **LARC reaches the end of the hash chain without finding any ScalarRecord with a matching tile label.** This s does not fall inside of any previously claimed region and MAR mode needs to create a new region that contains s.

      – It creates a new ScalarRecord for a primary tile with the scalar s as its regional representative.

      – It adds this ScalarRecord to the end of the hash chain.

      – It attempts to claim the desired neighbor tiles of this primary tile. Using the tile label for a neighbor tile, MAR searches the appropriate hash chain to see if this tile is already contained in some region. If the tile has no record, then MAR claims it as a neighbor for the primary tile by doing the following:

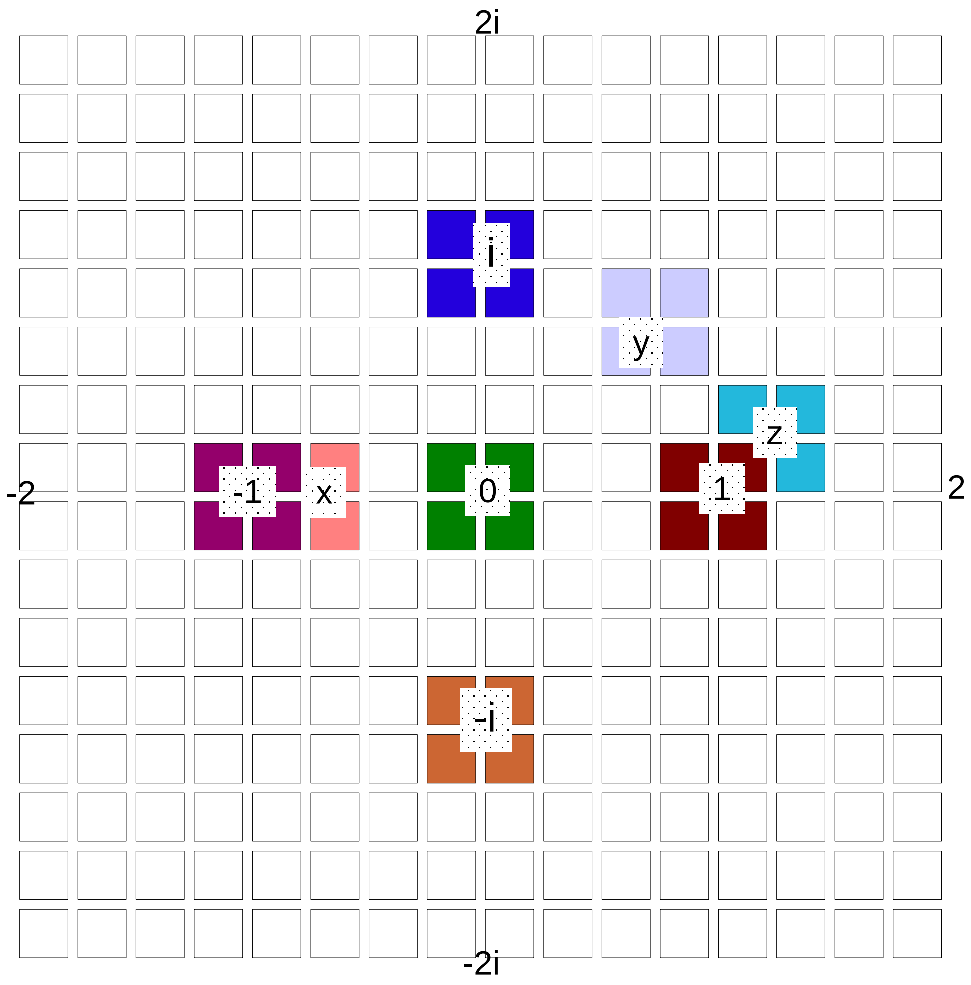


Figure 7: For display purposes we use large MAR tiles here of size 0.25×0.25 (`rb` $= 2$). The five preloaded integer-valued points 0, ± 1 and $\pm i$ become the centers of their 4-tile regions. Then we load the points $x = -\frac{1}{\sqrt{2}}$, $y = \frac{1+i}{\sqrt{2}}$, and $z = 1.25 + 0.25i$ to show various things that can happen when a primary tile tries to claim neighbor tiles while forming its region. The point y falls in the tile with lower left corner $0.5(1 + i)$ and is in the upper right quarter of that tile, thus y claims a region extending from $[0.5, 1.0)$ in both dimensions. Both x and z are near previously claimed regions, so are unable to claim all their desired neighbor tiles for their own regions.

* Create a neighbor tile record in the hash chain determined by the tile label for that neighbor tile.
* Place a link to the primary ScalarRecord inside the neighbor record.
* Modify the primary ScalarRecord by adding a link back to the newly created neighbor record.

Now that the new region has been created, MAR returns the new MatrixID for s which is in the primary tile ScalarRecord.

To get a default value for `regionbitparam` the user can enter -1. For example if the Scalar-Type is C `long double`, the default for `rb` is set to eight bits less than machine precision for C `long double`.

A.5 Hashstamps for Search Efficiency

In this section we would like to describe some more details of the ScalarStore hash table, especially something we call the hashstamp which is used to save time searching for previously stored scalars.

Each of the four LARC hash tables (MatrixStore, ScalarStore, OpStore, and InfoStore) contains an array of size 2^n where n is the hash-table exponent for that table. The array is indexed by the n-bit hash values and contains a pointer to the first hashnode of its hash chain. Each node in the hash chain is associated only with the one hash value that was the index into the array. Each hashnode contains 256 bits so that records are nicely aligned in cache. There are three 64-bit pointers, a 32-bit counter and a 32-bit field that is used for various purposes including flags. There are pointers `prev_node_PTR` and `next_node_PTR` to the previous node and next node in the hash chain, and a pointer `record_PTR` to a record of the appropriate type for what is being stored (MatrixRecordPTR, ScalarRecordPTR, OpRecordPTR and InfoRecordPTR). The 32-bit counter records the number of times that this node was used to recover stored information. One of the flag bits is used to indicate if the counter reached its full capacity.

Each region has a single ScalarRecord that contains the value of the scalar that is the regional representative as well as some other information. Each tile in the region has its own hashnode which is in the appropriate hash chain determined by hash(tile_label). All of the hashnodes corresponding to a tile of a region are linked to the single ScalarRecord that contains the regional representative for that region. For MAR mode, the ScalarRecord contains a list of pointers back to the hashnodes for each of its tiles.

The hashstamp is a short sequence of bits stored in the flag region of the hashnode that will save time when searching the ScalarStore for a previously stored scalar. The hashstamp is a function only of the tile label (the function is different than the hash used for the ScalarStore index), and thus has the property that the hashstamps of any two scalars lying in the same tile have the same value. Thus, if the hashstamp found in the hashnode is different from the hashstamp of the tile LARC is searching for, LARC can move its search to the next hashnode in the hash chain without having to access the ScalarRecord.

We expect that occasionally there will be two tiles that share both the same hash value (index into the ScalarStore hash table) and the same hashstamp. That is, given two different tiles a and b, letting their hash chain values be $H(a)$ and $H(b)$, and their hashstamp values be $S(a)$ and $S(b)$, we might coincidentally have $(H(a), S(a)) = (H(b), S(b))$. Therefore, when LARC sees a match in our hashstamp values, it still needs to make the check inside the ScalarRecord to verify that the tile is the one that it is looking for.

It is important for the good functioning of the hashstamp that coincidental matches of the pair $(H(a), S(a)) = (H(b), S(b))$ don't happen more often than statistically expected at random. Since even the best proofs of independence for hashes are vulnerable to subtle implementation errors, we verified experimentally that our statistics look random. The experiment works as follows: we iterated through a large grid of tiles calculating H and S for each tile; then we made counts of each time we saw a distinct (H, S) pair. The statistic we were most interested in was the maximum count. This should correlate to the statistic called maximum occupancy in the problem in which m balls are tossed with uniform probability into n bins. For this experiment m and n were equal. We verified that the maximum occupancy was approximately $\frac{\log n}{\log \log n}$, which is the value predicted for random data for the case $m = n$ [6]. We also used a random number generator to implement an experiment tossing m balls into n bins (matching the m, n values from our hash experiment). We visually compared the two distributions and saw a very close match.

A.6 Analysis of Snapping Probability for SPR and MAR modes

Two scalars that are a certain distance δ apart have a probability of collapsing (snapping) to the same value in LARC. The probability functions for SPR mode and MAR mode are different, but both depend on the choice of `regionbitparam` (`rb`).

Assume that we store randomly chosen scalars in the ScalarStore, then calculate some new scalar t whose minimum distance to any of the previously stored scalars is δ. MAR mode has a guarantee to snap to one of the previously stored scalars as long as δ is less than a function of `rb` we will derive below. In contrast, for any nonzero distance δ, SPR mode always has some possibility of failing to snap to any of the previously stored scalars.

Remember in SPR mode all regions consist of single tiles and are $W_S = 1/2^{\texttt{rb}}$ wide, whereas in MAR mode the standard region is two tiles wide and is of width $W_M = 2/2^{\texttt{rb}} = 1/2^{\texttt{rb}-1}$. Assume for the moment that we are not looking at special cases, that is, we are far from the axes in SPR mode (so can avoid thinking about `zeroregionbitparam`) and that we are looking at a MAR region that is full size (2-tile region in real types, and 4-tile square region in complex types). Also remember that the regional representative s can fall anywhere in the region for SPR mode. In MAR mode the region is constructed so that the representative will fall in the central portion of the region, thus s is at least $W_M/4$ from all region boundaries.

For each of SPR and MAR, we will first calculate for real types the probability that our new scalar t snaps to a previously stored scalar s, assuming scalars are locally distributed uniformly and given that

$$\delta = |t - s|$$

is the distance between t and s. For the purpose of our analysis we will assume that the newly calculated t and all the previously stored scalars fall in random locations inside their regions. After calculating the probability of snapping for real scalars, we will discuss the two dimensional version when scalars are complex.

A.6.1 Probability of Snapping in SPR mode

Real Scalars: Here is the calculation of snapping probability for real scalar types in SPR mode. Let s be a previously stored scalar that is δ from t. Assume without loss of generality that t is greater than s. If δ is either very large or zero then the probability of t snapping to this s is easily determined under the locally uniform probability assumption:

- If the width of the region W_S is less than δ then there is no possibility of snapping since s and t must be in different regions. Thus the probability of snapping is 0.
- When $\delta = 0$, the two scalars are identical and t snaps to s trivially. Thus the probability of snapping is 1.

For δ between 0 and W_S, either s and t are in the same region or in adjacent regions:

```
       delta                                            delta
       s----t                                           s----t
 |------------------|                          |------------------|
 W_S width of region                           W_S width of region
```

The probability that t and s are in the same region, and thus t snaps to s, is given by

$$P_S(\delta, W_S) = \frac{W_S - \delta}{W_S}.$$

Hence the probability of failure to snap is

$$1 - P_S(\delta, W_S) = \frac{\delta}{W_S}.$$

See Figure 8.

Complex Scalars: Now we want to calculate the probability in SPR mode of a newly formed scalar t snapping to a previously stored scalar s in the case that scalars are complex valued. The scalar t will only snap to s if they are in the same region. The variables to consider are the region width $W_S = 1/2^{\texttt{rb}}$, and the distances δ_R and δ_I from t to s in the real and imaginary directions:

$$\delta_R = |t_R - s_R| \qquad \delta_I = |t_I - s_I|.$$

Assuming that the previously stored scalars as well as t are in random locations, the imaginary and real components are independent of each other. Thus the probability of them jointly satisfying the condition to snap is the product of the probabilities for snapping in each of the real and imaginary directions. Using the probability function $P_S()$ that was calculated in the discussion of real scalars, we then have that the probability function for complex scalars of t snapping to s is

$$P_S(\delta_R, W_S) * P_S(\delta_I, W_S).$$

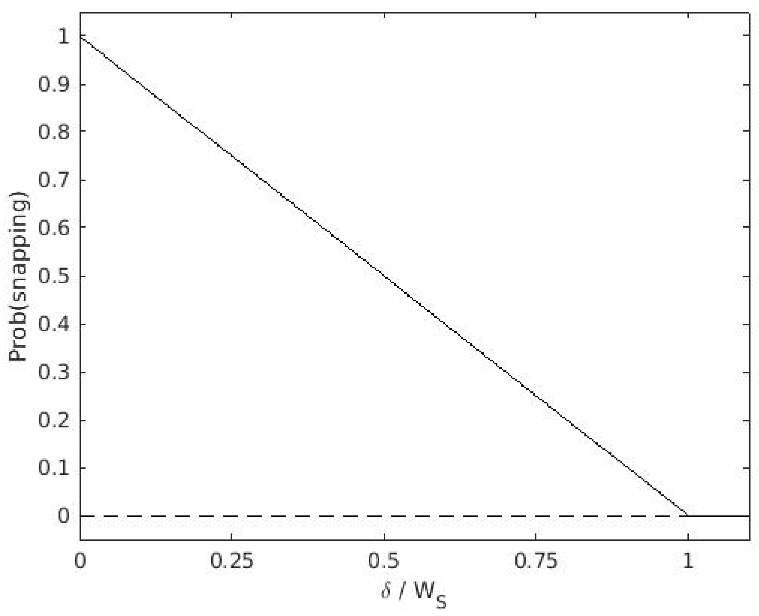


Figure 8: For SPR mode real scalars, this is the probability that a newly calculated scalar snaps to a previously stored scalar a distance δ away (or equivalently that the scalars are in the same region). The SPR regions are $W_S = 1/2^{\texttt{rb}}$ wide.

See Figure 9.

Note that a newly calculated scalar t will snap to a previously stored scalar s in its own region, regardless of whether there is a closer scalar s' in the scalar store. In fact s' and t can be arbitrarily close, while lying on opposite sides of a region boundary.

In the above analysis of snapping probability for SPR mode, we assumed that the parameters `rb` and `zrb` were equal. If this assumption is not made, the analysis is essentially the same but there are different tile widths depending on whether the tile contains zero in either the real or imaginary coordinate.

Remember that one goal of snapping is to collapse two scalars together that were symbolically equivalent. The discussion above leads to the conclusion that there is always a chance of failure to snap to a symbolically equivalent value in SPR mode, even for scalars that are arbitrarily close. The snapping probabilities calculated in this section will be used to analyze success and failure modes for mimicking symbolic/exact computation in Section A.7.

A.6.2 Probability of Snapping in MAR mode

Real Scalars: Here is the calculation of snapping probability for real scalar types in MAR mode. Recall MAR regions are $W_M = 1/2^{\texttt{rb}-1}$ wide and that MAR regions were created so that the regional representative s is in the central portion of the region, which has width $W_M/2$, so that the scalar s is at least $W_M/4$ from each region boundary.

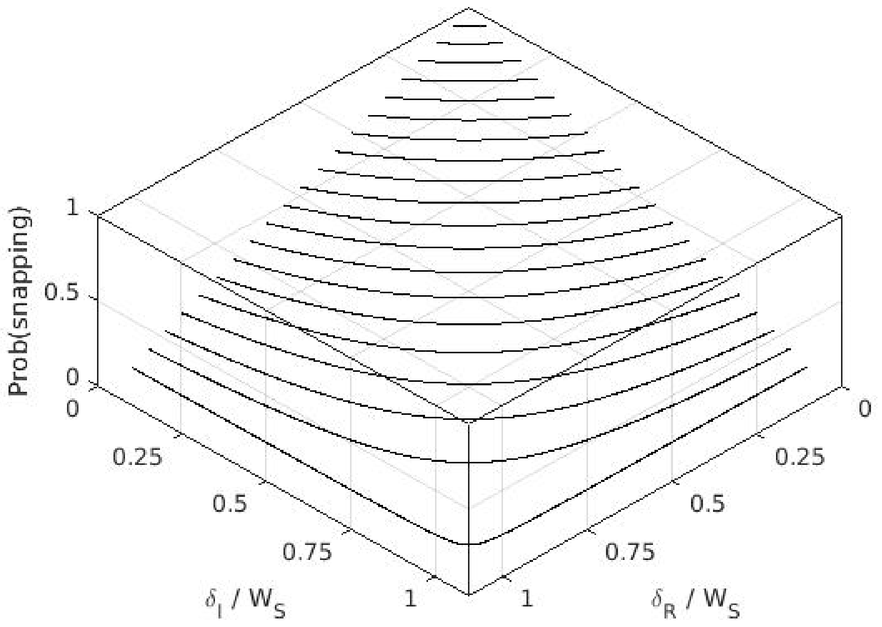


Figure 9: For SPR mode complex scalars, this is the probability that a newly calculated scalar snaps to a previously stored scalar when they are separated by δ_R in the real direction and δ_I in the imaginary direction. The SPR regions are $W_S = 1/2^{\mathtt{rb}}$ wide.

Let s be a previously stored scalar that is δ from t. Assume without loss of generality that t is greater than s. The scalar t snaps to s if they are in the same region. The probability of t snapping to s depends on the size of δ in comparison to W_M. There are three cases (see diagrams in Figures 10 and 11).

1. When $\delta < W_M/4$, then s and t will always be in the same region, because s is at least $W_M/4$ from the region boundary. Thus the probability of snapping is 1.
2. When $\delta > 3W_M/4$, then s and t must lie in different regions since s is at most $3W_M/4$ from all region boundaries. Thus the probability of snapping is 0.
3. When $W_M/4 < \delta < 3W_M/4$, then the probability that t and s are in the same region is given by
$$P_M(\delta, W_M) = \frac{(3W_M/4) - \delta}{W_M/2}.$$

One way to understand this equation is to see that it describes the straight line connecting the point $P_M = 1$ at $\delta = W_M/4$ to the point $P_M = 0$ at $\delta = 3W_M/4$ (we expect a straight line since we assumed that that s came from a locally uniformly distributed set, and the position of t on its right has been determined by the distance δ between s and t).

A second way to understand the equation is to create a variable y which is the distance from s to the left edge of the central portion of the region. When y and δ sum to $3W_M/4$,

t is on the right border of the entire region. See Figure 11. Thus when s is situated so that $y < (3W_M/4) - \delta$, t is in the region and snaps to s. When s is anywhere in the remainder of the central portion, t does not snap to s. Thus the probability that t snaps to s is the ratio of $(3W_M/4) - \delta$, the length of the segment from the left edge of the central portion to s when t is positioned at the right region boundary, to $W_M/2$, the total length of the central portion.

```
                  delta                                  delta
                  s---t                          s-------------------t
 |-------------------------|          |-------------------------|
   W_M/4|    W_M/2    |W_M/4            W_M/4|    W_M/2    |W_M/4
 |-------------------------|          |-------------------------|
      W_M region width                      W_M region width
```

Figure 10: The probability of snapping is 1 on the left (Case 1) when $\delta < W_M/4$. The probability of snapping is 0 on the right (Case 2) when $\delta > 3W_M/4$.

```
        |  y      |  delta |
        |--------s--------t
 |-------------------------|
   W_M/4|    W_M/2    |W_M/4
 |-------------------------|
  W_M region width
```

Figure 11: The probability of snapping, $((3W_M/4) - \delta)/(W_M/2)$, in Case 3 can be calculated by placing t at the right boundary of the region then taking the ratio of the length of the line segment y to the width of the central region.

As seen in Figure 12, the probability of snapping in MAR mode is qualitatively different than it was in SPR mode (Figure 8). In SPR mode there is always some probability that snapping will not occur, no matter how close t is to s. In contrast, MAR mode will always snap a newly calculated scalar t to a previously stored scalar if there are already one or more scalars in the ScalarStore that are within $W_M/4 = 1/2^{(\texttt{rb}+1)}$ distance from t.

This is not a guarantee that MAR mode with the right initial parameters will always find the nearest previously stored scalar. If two regions share a boundary, a newly calculated scalar may fall in one region, but be closer to the representative of the other region. Specifically, under the assumption of fully formed regions, a newly formed scalar t in region `A` will snap to its representative `a`, but t can be as close as $W_M/4$ from representative `b` of the adjacent region `B`, while it may be as far as $3W_M/4$ from `a`. See Figure 13.

This situation is even worse if the condition of only having fully formed regions is relaxed. That is, there are regions which failed to claim desired neighbor tiles because these tiles had already been claimed by an earlier claimed region (as its primary or neighbor tiles). Specifically, newly formed scalar t in the full-sized region `A` will snap to its representative `a`,

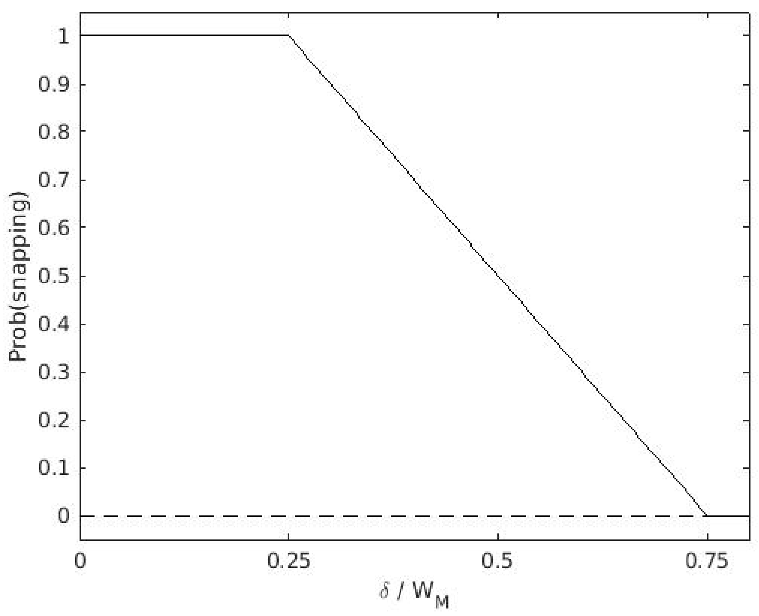


Figure 12: For MAR mode real scalars, this is the probability that a newly calculated scalar snaps to a previously stored scalar a distance δ away (or equivalently that the scalars are in the same region). The standard MAR region is $W_M = 1/2^{(\texttt{rb}-1)}$ wide.

but t can be arbitrarily close to representative `b` of the adjacent single-tile region `B`, while as before, it may be as far as $3W_M/4$ from `a`. See Figure 14.

What can be guaranteed in MAR mode:

- The earlier that a scalar is stored in the ScalarStore, the more likely it is to have a full-sized region.
- All full-sized regions provide a buffer within the region of width at least $W_M/4$ around the representative and any newly calculated scalar within that buffer will snap to this representative. In particular, if the buffer width is wider than numerical precision inaccuracy then LARC MAR mode is able to mimic exact/symbolic computation.
- A newly formed scalar never snaps to any scalar that is more than $3W_M/4$ away.
- If a newly formed scalar is within $W_M/4$ of one or more previously stored scalars, then it is guaranteed to snap to something.

Complex Scalars: Now we want to calculate the probability in MAR mode of a newly formed scalar t snapping to a previously stored scalar s in the case that scalars are complex valued. The scalar t will only snap to s if they are in the same region. The variables to consider are the region width $W_M = 1/2^{\texttt{rb}-1}$, and the distances δ_R and δ_I from t to s in the real and imaginary directions:

$$\delta_R = |t_R - s_R| \qquad \delta_I = |t_I - s_I|.$$

```
     |      Region A          |        Region B        |
     | with representative a  | with representative b  |

     |-----------|-----------|-----------|-----------|

     |     |a    |     |    t|     |b    |     |     |

-----|-----------|-----------|-----------|-----------|------
          tile1      tile2       tile3        tile4
```

Figure 13: Example of when newly calculated scalar t snaps to scalar a, in spite of being closer to another previously stored scalar b.

```
     |Earlier formed Region A|  Region B |
     | with representative a |   with b  |

     |-----------|-----------|-----------|

     |     |a    |     |    t|b    |     |

-----|-----------|-----------|-----------|-----
         tile1      tile2       tile3
```

Figure 14: The single-tile region B was formed after the region A, and was unable to claim tile2 as a neighbor tile. In this case, t can be arbitrarily close to b, but still snaps to a.

Assuming that the previously stored scalars as well as t are in random locations in their regions, the imaginary and real components are independent of each other. Thus the probability of them jointly satisfying the condition to snap is the product of the probabilities that they each would have snapped in the real and imaginary directions. Using the probability function $P_M()$ that was calculated in the discussion of real scalars, we then have that the probability function for complex scalars of t snapping to s is

$$P_M(\delta_R, W_M) * P_M(\delta_I, W_M)$$

Notice that in the square where δ_R and δ_I are both less than $W_M/4$, this probability function is 1. See Figure 15.

If all preloaded scalars have full-sized regions in MAR mode, then a newly calculated scalar is guaranteed to snap to a previously stored scalar that is less than $W_M/4$ in both the real and imaginary directions. Does this guarantee that when numerical precision of the calculation is smaller that $W_M/4$ that a symbolic identity in a ring maps to an identity for the stored scalars within LARC operations utilizing snapping? For example, if one preloads $a = \sqrt{2}$, $b = \sqrt{3}$, and $c = \sqrt{6}$ to high accuracy as LARC stored scalars and they have `MatrixIDs` a_{ID}, b_{ID}, and c_{ID} respectively, is it the case that the operation `multiply`(a_{ID}, b_{ID}) returns c_{ID}?

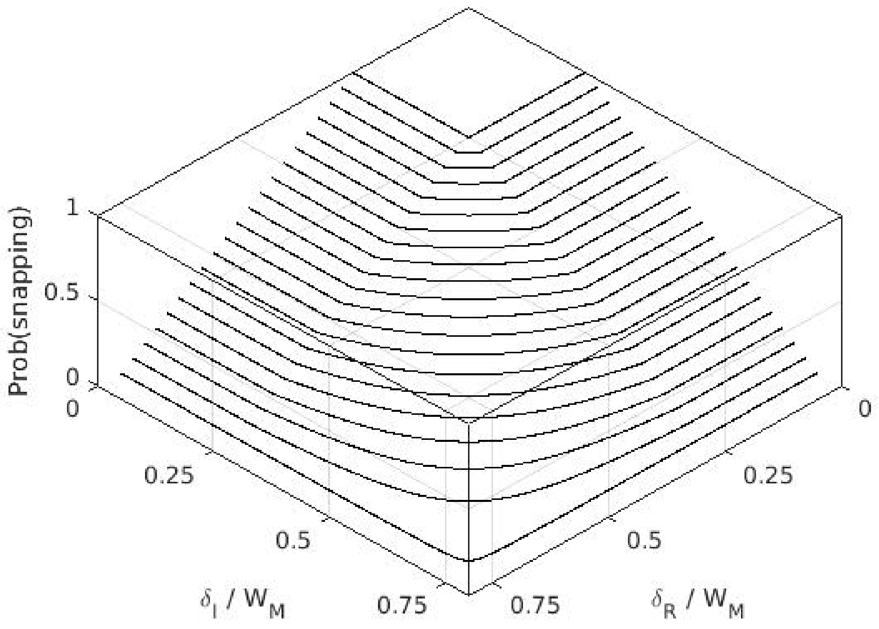


Figure 15: For MAR mode complex scalars, this is the probability that a newly calculated scalar snaps to a previously stored scalar when they are separated by δ_R in the real direction and δ_I in the imaginary direction. The standard MAR region is $W_M = 1/2^{(\texttt{rb}-1)}$ wide.

We will discuss symbolic mimicking, numerical precision, and good choices for initialization parameters in Section A.7.

A.7 Using Preloading and Selecting Region Width to Mimic Symbolic Computation and Reduce Memory

In order to mimic symbolic/exact computation, one needs to make good choices for which scalars to preload and what value to use for the region width parameter. Here are some tips and examples.

A.7.1 Preloading selected scalars

In this example, the user wants to ensure closure under multiplication of the n-th roots of unity. That is, LARC should return `MatrixID(`$w^{(j+k) \mod n}$`)` when we call the multiplication routine

$$\texttt{multiply(MatrixID(}w^j\texttt{),MatrixID(}w^k\texttt{))}$$

where the primary n-th root of unity is $w = e^{2i\pi/n}$. We will preload w^k for $k = 1, 2, \ldots, (n-1)$. Some of these roots of unity may already be preloaded, for example $1, -1, i$, and $-i$ are always preloaded. LARC has a routine `k_th_power_of_n_th_root_of_unity_matID` for calculating and loading the roots of unity. It uses a multiprecision MPC routine to calculate these roots to high precision. Depending on the scalar type being used, this precise calculation may be reduced in precision before being stored. So for example, you will get the full precision of this routine if you are using `TYPE=MPCOMPLEX` and you will only get C long double precision if you are using `TYPE=COMPLEX`.

LARC can preload high precision approximations to algebraic numbers such as $\sqrt{2}$, $1/\sqrt{2}$, $\sqrt{3}$, and $\sqrt{6}$. These and some others are already coded as options for preloading (see the function `get_enum_matID`). For example, one could mimic symbolic computations over the field over the field $\mathbb{Q}(i, \sqrt{2})$ by choosing `TYPE=MPRATCOMPLEX` and preloading a high precision approximation to the algebraic number $\sqrt{2}$. Another example would be an application using Hadamard matrices, in which case we would preload the high precision approximation to the scalar $1/\sqrt{2}$.

When using `TYPE=CLIFFORD` and implementing a particular Clifford algebra such as $\mathbb{Q}(\sqrt{2})$, the rational numbers adjoined with the algebraic number $\sqrt{2}$, LARC does not load a multiprecision approximation to this scalar. In this Clifford algebra the algebraic number $\sqrt{2}$ is a basis element. For example the scalar with value $5 + 3\sqrt{2}$ would be represented (without approximation) by the list of basis coefficients $(5, 3)$.

If an advanced user would like to use an algebraic number that is not listed as an option for `get_enum_matID`, the user can modify their copy of the LARC C code base to extend the function called by `get_enum_matID` so that it can preload their special scalar.

When a user is preloading scalars into the ScalarStore, they may occasionally run into a situation where two of the scalars are so close that they fall in the same region and LARC snaps the second one to the first preloaded value. This is rare, and can almost always be avoided by modifying the region widths so that all the preloads are in separate regions. (One can also turn on warnings by uncommenting the `#define ALERT_ON_SNAP` in larc.h, so that alerts are given when one scalar snaps to another.)

A.7.2 Selecting the Region Width

Selecting a good region width depends on the details of the application implementation. The choice of the scalarType (which may have different implementations or number of bits of accuracy in different computing environments) will result in numerical precision limits. In some applications, the user will want a set of symbolic equalities to hold in their LARC calculations, and this will determine the number of and distances between scalars that are prestored. The parameters must be chosen so that regions are small enough to differentiate these scalars, but not so small that numerical precision errors cause the LARC calculation corresponding to the underlying symbolic equality to fail to produce the same representatives from the LARC ScalarStore. Even if there is not an application-specific set of scalars to be prestored, the user may wish to set the parameters to collapse scalars together that are "close" in order to find or approximate the solution to the problem with the resources available.

Each scalar that is preloaded in the LARC ScalarStore is chosen to be a good computer representation of a scalar value V that can be written as one or more algebraic expressions in some ring. This exact value V might not be exactly representable in the computer, for example $\sqrt{2}$ has an infinitely long binary representation. Let us say that R is the closest scalar value that is representable by the computer. The difference between the true value V and the best computer representation R is no more than the last bit of accuracy. In some representations for floating point this accuracy varies as the number becomes larger.

The value of R may be difficult to compute, so in practice we may choose not to compute R, but instead use some less accurate computation. Let C_0 be the first scalar that we calculate and store in the LARC ScalarStore when we are attempting to represent V. The scalar C_0 will differ from V for a combination of different reasons. For example, say that we are attempting to represent $V = 20 * \sqrt{2} * \pi$. First there is limited precision caused by the finite representation of $\sqrt{2}$ and π. Even using a multiprecision package for preloading these values, there are likely to be additional limitations caused by the choice of scalarType. Additional error is created when carrying out the operations (in this case two multiplies and a square root). Say that C_i is a value that is calculated by a different expression for V at a later point. For example, C_1 could be the scalar we calculate using the expression $\sqrt{\pi^2 * 800}$, which is symbolically equivalent to the original expression for V. In general, the C_i's are likely to differ since the rearranged order of operations will introduce different errors. One would hope that the C_i would snap to C_0.

Thus the exact value V is likely to differ both from R, the best possible representation storable in the computer, and from the C_i, the results of computations implementing various formulas for V. We expect C_0 to be close to R if the expression is simple and carried out using high quality multi-precision packages. However, complicated expressions and implementations using limited precision scalars, like scalarType `COMPLEX` which is only C long double, will be further from R.

One would like V, R, and C_0 to be in the same LARC region so that the originally stored scalar C_0 is a good approximation for V. Furthermore, C_0 should be a good snapping target for subsequently calculated C_i, that is, they should have a high probability of falling in the same region.

Based on the discussion above, the following is claimed when operating in MAR mode:

- Let $a_R = |V - R|$. If $a_R < (W_M/4)$ then V and R are in the same region.
- The distance from R to the first calculated and stored representative, C_0, is a function of both the numerical precision of any prestored scalars required and of each of the operations carried out to compute C_0. If we were able to track some upper bound for the precision of C_0, then we could choose to set the region width so that $(W_M/4) - a_R > |R - C_0|$. From this it would follow that R and C_0 lie in the same region.

It is beyond the scope of this paper to track the precision of any particular user's computations. However, one can get reasonable results by using the following method to estimate the numerical precision and select the region-width parameters.

Here is a sample method that might be used to select W_M in MAR mode. Define the "expected error", E_C, of the computed scalars experimentally to be the average value over all $i \neq j$ of $|C_i - C_j|$ where C_i are the computer-generated scalar approximations from many equivalent expressions for V. Similarly, the standard deviation of the expected error, σ_C, is obtained as the standard deviation of these differences. Depending on the number of scalars that we expect to generate in a particular application, we will pick the region width so that $E_C + m * \sigma_C + a_R < W_M/4$, where m is some number of standard deviations away from the expected value as desired by the user.

Note that if one has chosen a scalarType with insufficient precision and then made the region width large enough to satisfy this equation, then there may be circumstances where scalars that are not "symbolically" equivalent snap together in these large regions.

Choosing the upper bound for region width depends on the set of non-equivalent symbolic values V_i which need to be distinguished in the application. The minimum distance for pairs (i, j) of $|V_i - V_j|$ should be greater than $3 * W_M/4$ to guarantee that no V_i snaps to another.

In SPR mode, although there is no guarantee of snapping, it is possible to guarantee two values do not snap together by setting the tile width W_S to be less than the smallest separation $|V_i - V_j|$ over the set of non-equivalent symbolic values.

Finally, when actually selecting the input parameter regionbitparam (`rb`), recall that the standard MAR regions are $W_M = 1/2^{\texttt{rb}-1}$ wide and the standard SPR regions are $W_S = 1/2^{\texttt{rb}}$ wide.

A.8 Comparison of MAR mode and SPR mode Advantages

Region Shape and Width

SPR mode regions have the following widths:

- the width for a tile centered at zero is $1/2^{\texttt{zrb}} - 1/2^{\texttt{rb}}$,

- otherwise the tile is $1/2^{\texttt{rb}}$ wide, and
- for complex scalars, the tile width in real and imaginary directions is determined separately following these rules.

When $\texttt{rb} = \texttt{zrb}$, all SPR mode regions have the same width $1/2^{\texttt{rb}}$. The scalar representative can fall anywhere inside the SPR region.

SPR mode allows us to make larger regions around axes so we can strengthen the snapping to zero. This might be useful in complex scalar value applications in which some class of objects is known to have no imaginary part.

MAR mode regions are composed of one or more tiles of width $1/2^{\texttt{rb}}$. The standard region will be two tiles wide so we use $W_M = 1/2^{\texttt{rb}-1}$. If the regions are standard size the region representative will fall in the central half of the region (at least $W_M/4$ from any boundary). Regions will fail to be two tiles wide, if during the building of the region one or more of the desired tiles has already been claimed by a previously stored scalar. Thus MAR mode region shapes depend on the order of preloading and operations.

Locations of Region Boundaries

Both MAR mode and SPR mode were designed so that integers, and more generally multiples of inverse powers of two, can lie in the center of regions. For MAR mode the tile boundaries are always at multiples of inverse powers of two. MAR region boundaries are not necessarily on the same line as the tiles claimed for the region depend on the location of the representative in its tile. For SPR mode, all the region boundaries fall in straight lines even when the regions have different sizes (when $\texttt{rb} \neq \texttt{zrb}$). However, for different values of `rb`, the location of the region boundaries in SPR mode shift in an interesting fashion and never line up with those for a different `rb`.

Guarantee of Snapping, Guarantee of Separating

MAR mode has a guarantee of snapping to a previously stored close by scalar if one exists. Specifically, if a scalar t is at least as close as $W_M/4$ to any prestored scalar, it will snap to some prestored scalar that is within $3W_M/4$. SPR mode always has a probability of not snapping when some scalar that is close has been previously stored. MAR mode will not snap two values that are at least $3W_M/4$ apart. SPR mode will not snap two values that are at least W_S apart.

Even with MAR mode, we cannot guarantee snapping for all expressions over an infinite field, but usually with the right scalar type and input parameters one can do well for simple expressions in a field. For example, in the field $\mathbb{Q}[\sqrt{3}]$, if we preload $\sqrt{3}$ and $-\sqrt{3}$, simple symbolically equivalent expressions will usually snap together. For example,
MatrixID($(2+\sqrt{3})(\sqrt{3})$) = MatrixID($3+2\sqrt{3}$).

LARC Memory Use

For some applications MAR mode will have fewer scalars and fewer matrices than SPR mode (assuming parameters are chosen so that region size is the same, $W_M = W_S$). This is because MAR mode is guaranteed to snap scalars of distance up to $W_M/4$ together, whereas SPR mode will fail to snap them together if they happen to fall on opposite sides of a SPR region boundary.

A.9 Experimental Comparison of SPR mode and MAR mode

In order to appropriately set the LARC region bit parameter, we recommend that the user run one or more experiments using data that is representative of the desired application. The region tiles need to be small enough in order to be able to separate computational data values that are symbolically distinct; otherwise, snapping could introduce error into the computation. Within this limitation, it is highly desirable that the region tiles be large enough so that different computational data values that are symbolically equal will collapse to a single stored value.

Experiments

This section describes two such experiments. In both experiments, the `COMPLEX` data type was used, which is implemented in LARC as a C language "long double complex" type.

In the first experiment, powers of the roots of unity were manipulated. (This kind of computation is similar to what is performed in a Fourier transform.) First, the values $1^{k/n} = e^{2\pi ki/n}$ for $0 \leq k < n \leq 200$ were computed to high precision using the GNU MPC (Multi-Precision Complex) package and preloaded into the LARC store. Then, for $0 \leq k_1, k_2 < n \leq 200$, $e^{2\pi k_1 i/n}$ times $e^{2\pi k_2 i/n}$ was computed and compared to $e^{2\pi(k_1+k_2)i/n}$.

This experiment was a success, in that it turned out to be possible to set the region bit parameter to satisfy both the lower and upper requirements. In MAR mode, as long as the region bit parameter was at least 14, none of the preloaded answers collapsed to a different answer, and as long as the region bit parameter was at most 62, all of the symbolically equivalent computations collapsed to the same answer. For SPR mode, the corresponding bit parameters results were 13 and 49.

In the second experiment, three complex numbers of unit norm were chosen: $(\sqrt{3} + \sqrt{6}i)/3$, $(4 + 3i)/5$, and $(12 + 5i)/13$. Then, each of these numbers was raised to a non-negative integer value, and then multiplied together. In this experiment, we did not expect to see any different exponent choices resulting in symbolically equal values, so we could only determine a minimum region bit parameter to avoid collapsing symbolically different values. After observing ten million such products, the minimum region bit parameter was found to be 41 for MAR mode and 40 for SPR mode.

Experimental Details

For the first experiment, the 86 incorrect snaps occurred during the preload of the correct answers in MAR mode with a region bit parameter of 13. The first incorrect snap was:

```
=10=>> MAR SNAP  FROM: -0.70385161282591149606+I*0.71034703288066402649
                   TO: -0.70369146714878502295+I*0.71050567841642927214
=16=>> MAR SNAP  FROM: -0xb.42f9e8a52fb1a30p-4+I*0xb.5d94d9b08787facp-4
                   TO: -0xb.4251fbde0d79db5p-4+I*0xb.5e3b33c6bbd4552p-4
Above snap occurred for load of 1^(64/171) collapsing to 1^(61/163).
```

For the first experiment, in MAR mode with a region bit parameter of 63, the following results were observed:

```
FINAL RESULTS: # of tests = 1353400,  # of snaps = 915455.
# of failures due to SNAPPING = 933, smallest denominator=23.
# of failures due to NOT SNAPPING = 5085, smallest denominator=13.
# of successes due to SNAPPING = 914522.
# of successes due to NOT SNAPPING = 432860.
```

For the first experiment, the 292 incorrect snaps occurred during the preload of the correct answers in SPR mode with a region bit parameter of 12. The first incorrect snap was:

```
=10=>> SPR SNAP  FROM: 0.62781212467209856148+I*0.77836491192416001195
                   TO: 0.62800687247767942808+I*0.77820779237990394780
=16=>> SPR SNAP  FROM: 0xa.0b84b9f7fba675bp-4+I*0xc.742ec411174bf7fp-4
                   TO: 0xa.0c50ef2f472bac2p-4+I*0xc.738a039c3987bc6p-4
Above snap occurred for load of 1^(23/162) collapsing to 1^(22/155).
```

For the first experiment, in SPR mode with a region bit parameter of 50, the following results were observed:

```
FINAL RESULTS: # of tests = 1353400,  # of snaps = 920531.
# of failures due to SNAPPING = 41, smallest denominator=157.
# of failures due to NOT SNAPPING = 9, smallest denominator=157.
# of successes due to SNAPPING = 920490.
# of successes due to NOT SNAPPING = 432860.
```

For the second experiment, in MAR mode with a region bit parameter of 40, the following results were observed:

```
x = (sqrt(3)+I*sqrt(6))/3 = 0.57735026918962576451+I*0.81649658092772603273
y = (12+I*5)/13 = 0.92307692307692307694+I*0.38461538461538461538
z = (4+I*3)/5 = 0.80000000000000000001+I*0.60000000000000000002

=10=>> MAR SNAP  FROM: 0.97804025475428777218-I*0.20841607442851555729
                   TO: 0.97804025475442555898-I*0.20841607442781470431
=16=>> MAR SNAP  FROM: 0xf.a60d89c5757363cp-4-I*0xd.56b05fe86530e54p-6
```

```
                 TO: 0xf.a60d89c577dfed2p-4-I*0xd.56b05fe833df736p-6
ANALYSIS: x^(4753321)y^(5132421)z^(113379) -> x^(517895)y^(510522)z^(17076),
      so  x^(4235426)y^(4621899)z^(96303) -> 1.
```

For the second experiment, in SPR mode with a region bit parameter of 39, the following resuls were observed:

```
x = (sqrt(3)+I*sqrt(6))/3 = 0.57735026918962576451+I*0.81649658092772603273
y = (12+I*5)/13 = 0.92307692307692307694+I*0.38461538461538461538
z = (4+I*3)/5 = 0.80000000000000000001+I*0.60000000000000000002

=10=>> SPR SNAP  FROM: 0.41745588936878674471+I*0.90869718852395485096
                   TO: 0.41745588936813305130+I*0.90869718852424271379
=16=>> SPR SNAP  FROM: 0xd.5bcc740b91adf8ep-5+I*0xe.8a06102ad72c6bcp-4
                   TO: 0xd.5bcc740b7aae068p-5+I*0xe.8a06102adc3cd70p-4
ANALYSIS: x^(4753322)y^(5132422)z^(113379) -> x^(517896)y^(510523)z^(17076),
      so  x^(4235426)y^(4621899)z^(96303) -> 1.
```

Combining the results of the two experiments (roots of unity and the more general norm one complex numbers) a good regionbitparam value for MAR mode would be in range from 41 to 62 and for SPR mode would be in the range would be 40 to 49 (both compiled with type `COMPLEX`).

B Converting Sparse Row Format to LARC Format

In this section we discuss how LARC converts sparse matrix formats (such as Matrix Market Sparse format) into the recursive compressed LARC matrix format.

One of the most common sparse row formats for matrices is to have one line per nonzero entry which contains the row and column coordinates followed by the value in that entry. The file usually has a header or accompanying metadata file that gives information such as what type of scalar values are used (real, integer, complex, etc.), the size of the matrix, and the number of nonzero entries.

LARC can efficiently convert such a sparse matrix format into LARC compressed format with minimal overhead. The process works by converting the sparse matrix into an intermediate format which looks like a truncated quadtree, in which every leaf is a maximal submatrix in which either all entries are zero or only a single entry is nonzero.

For example, say we have a 7 by 7 sparse matrix with seven non-zero entries that has the following entries (row, column, value) with indices starting from 0.

row	col	value
1	6	7
2	4	8
4	1	1
5	0	4
5	1	5
5	3	6
6	6	8

We will pad this out to the next power-of-two dimensioned matrix, which in this example corresponds to the 8 by 8 matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 5 & 0 & 6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

When we are done inputting the sparse matrix into our intermediate format, we will have a quadtree-like structure (Figure 16) in which every internal node has four children listed in Z order (NW, NE, SW, SE). The leaves have various sizes 2^k by 2^k and are either an all-zero matrix Z_k or a matrix $S_k(i, j, v)$ with a single non-zero entry v in position (i, j) (given in 0-based indices). The internal branching nodes B are numbered in the order that they are created.

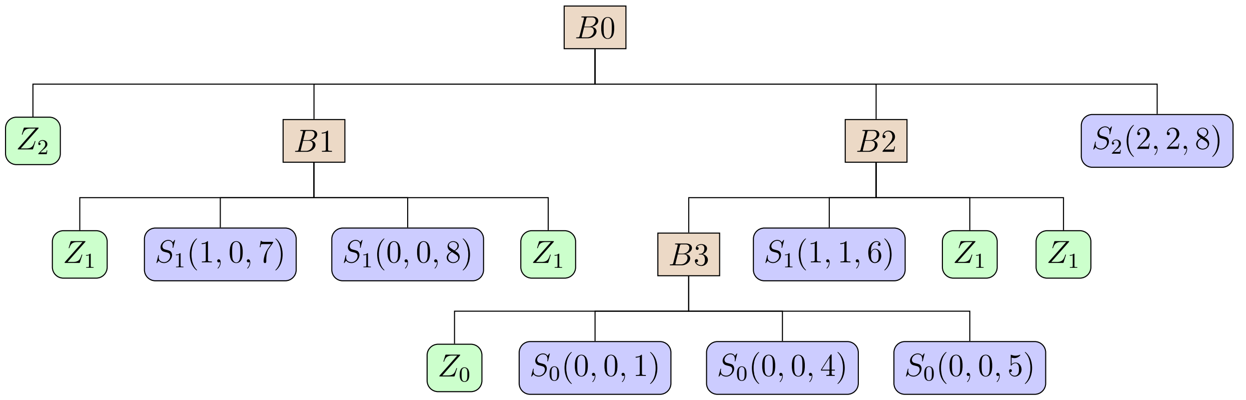
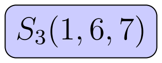


Figure 16: Input sparse matrix data is converted to this quadtree-like structure before being converted to LARC format.

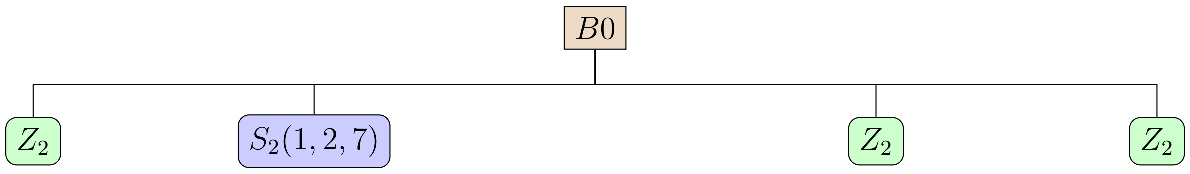
In this representation, like in LARC, the first child of the top node will correspond to the NW quadrant of the whole matrix, the second child will correspond to the NE quadrant, the third child to the SW, and the fourth child to the SE.

To create this intermediate quadtree-like structure, we process each non-zero entry in turn as we read the sparse matrix file. Initially we have an all-zero 8 by 8 matrix

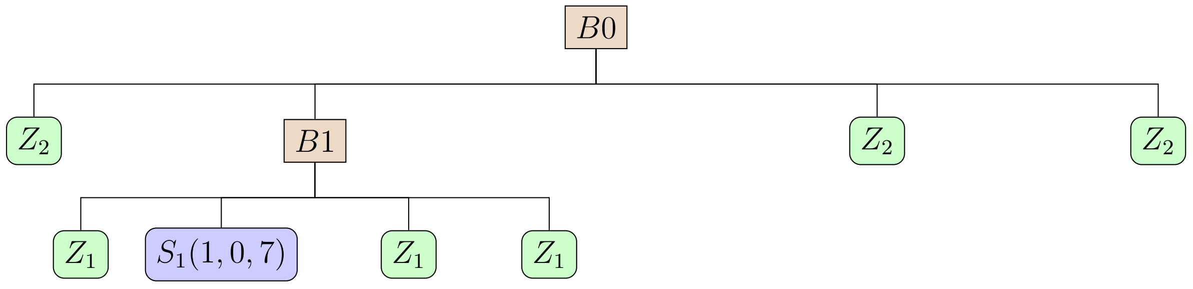
Adding the 7 in position (1,6), we just replace the all zero matrix with the appropriate $S_3(i, j, v)$ matrix.



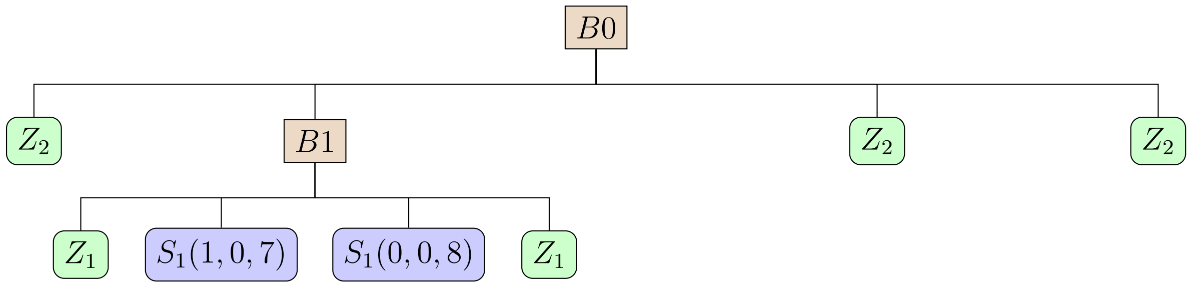
We next attempt to add the 8 in position (2,4). This would create two nonzeros in the 8 by 8 matrix so we need to create a larger tree to still satisfy our rules. We add an internal branch node `B0` at the 8 by 8 level, which notionally starts with four all-zero children. Now we will need to have a $S_2(i', j', 7)$ matrix as one of the four children of `B0` that describes where the 7 falls. Whenever we push a value $S_{k+1}(i, j, v)$ down to a lower level $S_k(i', j', v)$ we take advantage of the fact that indices are 0-based to make the following simple calculations: $i' = i \mod 2^k$ and $j' = j \mod 2^k$. Furthermore since the quadrants of a 2 by 2 matrix are ordered with their row and column indices being (0,0), (0,1), (1,0), and (1,1), we can determine the location of $S_k(i', j', v)$ as one of the four children of the new branch node, by looking at the high bits of i and j as $(k+1)$-long binary numbers. This can be expressed by $(i \text{ div } 2^k, j \text{ div } 2^k)$, where a div b is the integer part of a/b. In our example when we push down $S_3(1, 6, 7)$ it becomes the second child, the child indexed with $(0, 1)$, and the node is labeled $S_2(1, 2, 7)$.



Now we try to modify this tree to add the 8 in position (2,4). First we determine which child of `B0` would contain the 8, by the same calculation we did to place the 7 (looking at the high bits of the indices (2,4) expressed as 3-bit binary numbers). Since this gives the second child (position (0,1)) we see that both the 7 and 8 lie in the same 4 by 4 submatrix and we need to again create an internal node `B1` with four children and push the 7 down into a 2 by 2 matrix.



Now when we try to add the 8, it is not in the same submatrix as the 7, it is in the third position indexed by (1,0), so we can simply replace the third Z_1 with $S_1(0,0,8)$.



While the original list of input sparse elements happened to be in row major order, the algorithm succeeds with any ordering. If the ordering of the input matrix were permuted, then the only difference in the quadtree-like intermediate structure would have been in the numbers assigned to the branch nodes.

The final step is to convert the intermediate quadtree-like structure into LARC recursive format. LARC traverses the branch nodes in the reverse ordering of their numbering. At each step we need to define a LARC matrix record by the list of the MatrixIDs of its four quadrant submatrices. The zero matrices have all been pre-stored, so LARC calls the function which returns their MatrixIDs, and there is a simple recursive function that returns the MatrixID corresponding to $S_k(i,j,v)$.

For n by n matrices, with r nonzero entries, the worst case for number of nodes used by the quadtree-like structure has an upper bound $2r\log_2(n)+1$. This bound is closest to being achieved by putting pairs of nonzero entries in the same 2 by 2 matrices and then placing these 2 by 2's in as distinct as possible branches of the quadtree. The bound follows by noting that the height of the quadtree-like structure is at most the level $L = \log_2(n)$ and seeing that long paths down to the bottom level pair of nonzero entries (the 2 by 2 matrix) will add 4 nodes at each level (3 zero nodes and one branch at each level other than the lowest). However, for a random sparse matrix, the node count in the quadtree-like structure should be well below this bound.

Notice that unlike the LARC compressed format, the quadtree-like representation does not compress if we see the same leaf twice. Once we convert to the LARC format this compression takes place automatically.

The important point is that if we have a really large, really sparse matrix to input to LARC, we avoid the cost in space and time of having to convert to a dense matrix format.

The MyPyLARC package contains example code using the LARC calls to read graphs in Matrix Market "sparse" format and implements a triangle counting example.

C Compressibility of Various Matrix Families

The ***LARCsize*** of a matrix is the number of ScalarRecords and MatrixRecords needed to specify that matrix in LARC. This is equal to the number of unique quadrant submatrices inside the matrix's quadtree (the recursive division into quadrant submatrices). Some matrices have a great deal of repetition in these submatrices so they have small LARCsize compared to the size of the matrix. For example, although the $2^n \times 2^n$ integer Hadamard Matrix H_n is not sparse (it has no zero entries), it has a small LARCsize of $2n+1$ because its recursive definition requires only two records of the next smaller level. The integer Hadamard matrices are specified by

$$H_0 = 1; -H_0 = -1; H_1 = [H_0, H_0, H_0, -H_0]; -H_1; \dots ; H_n = [H_{n-1}, H_{n-1}, H_{n-1}, -H_{n-1}].$$

In contrast, a $2^n \times 2^n$ completely uncompressible matrix filled with distinct scalars would 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} 1×1 matrices (each holding a single scalar), giving a LARCsize of

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

Such an uncompressible matrix takes up more space in LARC and other "compression schemes" than in a dense format (like row major format) because of the overhead of the scheme.

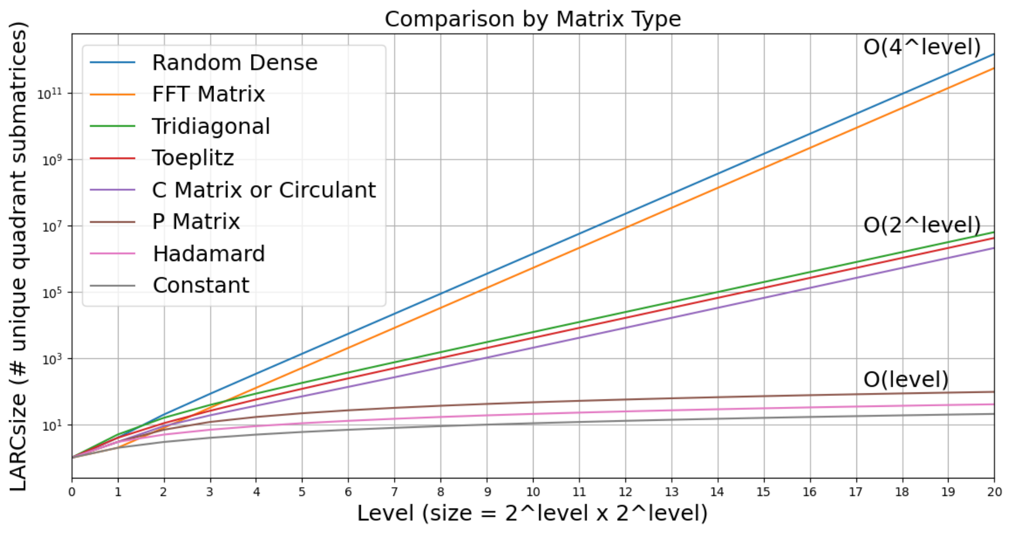


Figure 17: Asymptotic LARCsize for various families of matrices. This is generated by MyPyLARC/Tutorial/create_larc_matrix_compression_graph.ipynb, a Jupyter notebook.

We will express the growth of the LARCsize for a family of matrices in two ways: 1) as a function of the level n, and 2) as a function of the number of possible scalars in a matrix which is $N = 2^n \times 2^n = 4^n = 2^{2n}$. As seen in Figure 17, there are three natural groups of the asymptotic growth curves for these various families of matrices. The lowest grouping has linear growth $O(n) = O(\log_2(N))$. The middle grouping has exponential growth of the form $O(2^n) = O(\sqrt{N})$. The last grouping has exponential growth of the form $O(4^n) = O(N)$. Each grouping has a section below.

C.1 Linear growth $O(n) = O(\log_2(N))$

We describe three families of matrices in this section that have no more than some fixed constant number of distinct quadrant submatrices at each level. The constant matrices have 1 submatrix at each level, the integer Hadamard matrices have no more than 2 quadrant submatrices at each level, and the inverse shuffle permutation matrices have no more than 5 quadrant submatrices at each level.

Constant Matrices

Let M be a 2^n by 2^n matrix where all entries are the same scalar s. Then the quadrant submatrices of M are all equal, and recursively downward each of their quadrant submatrices are equal, so M is specified by only one MatrixRecord at each level from 0 to n. So the LARCsize of M is $n + 1$.

Integer Hadamard Matrices

The integer Hadamard matrices contain two scalars, 1 and -1. For example,

$$H_3 = \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & \text{-1} & 1 & \text{-1} \\ \hline 1 & 1 & \text{-1} & \text{-1} \\ 1 & \text{-1} & \text{-1} & 1 \end{array}\right]$$

For these matrices, as we noted above, we only need two different quadrant submatrices to build the matrix of the next level. The matrix H_n is described by $[H_{n-1}, H_{n-1}, H_{n-1}, -H_{n-1}]$, and in turn the two different quadrant submatrices H_{n-1} and $-H_{n-1}$ are themselves specified by only two different submatrices: $H_{n-1} = [H_{n-2}, H_{n-2}, H_{n-2}, -H_{n-2}]$ and $-H_{n-1} = [-H_{n-2}, -H_{n-2}, -H_{n-2}, H_{n-2}]$. This gives LARCsize $2n + 1$.

Inverse Shuffle Permutation Matrices

Here is the $2^n \times 2^n$ inverse shuffle permutation matrix P_n for $n = 3$.

$$P_3 = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right]$$

The P_n matrices, seen in the block recursive expression for the discrete Fourier transform in Section 3.1, are expressible recursively. The first three matrices in the sequence are

$$P_0 = \begin{bmatrix}1\end{bmatrix}; \quad P_1 = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}; \quad P_2 = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

and for $n \geq 2$, we have that

$$P_n = \left[\begin{array}{cc|cc} P_{n-1}[00] & P_{n-1}[01] & Z_{n-2} & Z_{n-2} \\ Z_{n-2} & Z_{n-2} & P_{n-1}[00] & P_{n-1}[01] \\ \hline P_{n-1}[10] & P_{n-1}[11] & Z_{n-2} & Z_{n-2} \\ Z_{n-2} & Z_{n-2} & P_{n-1}[10] & P_{n-1}[11] \end{array}\right] \text{ where } P_{n-1} = \left[\begin{array}{cc} P_{n-1}[00] & P_{n-1}[01] \\ P_{n-1}[10] & P_{n-1}[11] \end{array}\right].$$

This is an interesting recursion because it describes the matrix P_n in terms of the quadrant submatrices of P_{n-1}. In the above equation, the quadrant submatrices of P_{n-1} are level $n-2$, but they are denoted by $P_{n-1}[00]$, $P_{n-1}[01]$, $P_{n-1}[10]$, and $P_{n-1}[11]$. (We apologize for breaking the convention that the subscript always denotes the level.)

The LARCsize for the small cases P_0, P_1, and P_2, are respectively 1, 3, and 7. For $n \geq 3$, each P_n matrix has four different quadrant submatrices (children), but these child matrices only require a total of five different grandchild matrices (of level $n-2$) to describe. For sufficiently large n the following is true. There are two scalar matrices: [0], [1]. There are five distinct 2×2 matrices: the all-zero matrix and the four matrices which have a single 1 in each corner and are otherwise all zero. At the top $(2^n \times 2^n)$ level there is a single matrix which has four distinct children. At every other level, there is the all-zero matrix and four other matrices which have the form $[A, B, 0, 0]$, $[0, 0, A, B]$, $[C, D, 0, 0]$, $[0, 0, C, D]$. This yields a LARCsize of $2 + 5 + 5 + \cdots + 5 + 4 + 1 = 5n - 3$.

C.2 Exponential growth of the form $O(2^n) = O(\sqrt{N})$

Diagonal Matrices

Let D_n be a general 2^n by 2^n diagonal matrix constructed from 2^n unique non-zero scalars. For example, one such matrix is

$$D_3 = \left[\begin{array}{cc|cc|cccc} a & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & c & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & d & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & e & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & f & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & g & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & h \end{array}\right]$$

As you can see in the example, the off diagonal quadrant submatrices are entirely zero, and the diagonal quadrant matrices are themselves diagonal matrices. Working up from the bottom, we see there are 2^n non-zero scalars and of course the scalar 0, for a total of $2^n + 1$ 1×1 matrices. At level 1, there are 2^{n-1} distinct nonzero matrices in the quadtree and the all-zero 2×2 matrix, for a total of $2^{n-1} + 1$ different matrices. This pattern continues up to level $n - 1$, which has 3 unique quadtree matrices. Finally, level n has a single matrix. Summing these numbers we find that the LARCsize of the worst case diagonal matrix (with all distinct scalars on the diagonal) is

$$n \text{ all-zero matrices } + \sum_{k=0}^{n} 2^k \text{ non-zero matrices } = 2 * 2^n + n - 1.$$

Tridiagonal Matrices

Let T_n be a general 2^n by 2^n tridiagonal matrix constructed from $3 * 2^n - 2$ unique non-zero scalars. For example, one such matrix is

$$T_3 = \left[\begin{array}{cccc|cccc} \mathrm{a} & \mathrm{b} & 0 & 0 & 0 & 0 & 0 & 0 \\ \mathrm{c} & \mathrm{d} & \mathrm{e} & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathrm{f} & \mathrm{g} & \mathrm{h} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathrm{i} & \mathrm{j} & \mathrm{k} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & \mathrm{l} & \mathrm{m} & \mathrm{n} & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathrm{o} & \mathrm{p} & \mathrm{q} & 0 \\ 0 & 0 & 0 & 0 & 0 & \mathrm{r} & \mathrm{s} & \mathrm{t} \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathrm{u} & \mathrm{v} \end{array}\right]$$

The analysis of this case is somewhat similar to that for diagonal. At level 0, there are $3 * 2^n - 2$ non-zero scalars plus 1 zero scalar. At level 1, there are $3 * 2^{n-1} - 2$ non-zero quadrant submatrices plus 1 zero matrix. This pattern continues through level $(n - 2)$. At level $(n - 1)$ there are 4 non-zero quadrant submatrices, and at level n there is one matrix. The total is

$$(n-1) \text{ all-zero matrices } + (5 + \sum_{k=0}^{n-2} (3 * 2^{n-k} - 2)) \text{ non-zero matrices } = 6 * 2^n - n - 6$$

Toeplitz Matrices

Let A be a general 2^n by 2^n Toeplitz matrix constructed from $2^{n+1} - 1$ unique scalars. For example,

$$\mathrm{Toep}_3 = \begin{bmatrix} \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{f} & \mathrm{g} & \mathrm{h} \\ \mathrm{i} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{f} & \mathrm{g} \\ \mathrm{j} & \mathrm{i} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{f} \\ \mathrm{k} & \mathrm{j} & \mathrm{i} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} \\ \mathrm{l} & \mathrm{k} & \mathrm{j} & \mathrm{i} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} \\ \mathrm{m} & \mathrm{l} & \mathrm{k} & \mathrm{j} & \mathrm{i} & \mathrm{a} & \mathrm{b} & \mathrm{c} \\ \mathrm{n} & \mathrm{m} & \mathrm{l} & \mathrm{k} & \mathrm{j} & \mathrm{i} & \mathrm{a} & \mathrm{b} \\ \mathrm{o} & \mathrm{n} & \mathrm{m} & \mathrm{l} & \mathrm{k} & \mathrm{j} & \mathrm{i} & \mathrm{a} \end{bmatrix}$$

Note that at every level of quadrant submatrices, those that lie diagonally from each other are equal. Thus, the number of quadrant submatrices of level k for the level n Toeplitz matrix (worst case) is $2^{n-k+1}-1$, and the total number of distinct quadrant submatrices for Toeplitz (worst case) is

$$\sum_{k=0}^{n}(2^{n-k+1}-1) = 4 * 2^n - n - 3$$

Interestingly, this puts the asymptotic growth of Toeplitz matrices between the growth of diagonal and tridiagonal matrices.

Circulant Matrices

Let A be a general 2^n by 2^n circulant matrix constructed from 2^n unique non-zero scalars. For example,

$$\mathrm{Circ}_3 = \begin{bmatrix} \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{f} & \mathrm{g} & \mathrm{h} \\ \mathrm{h} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{f} & \mathrm{g} \\ \mathrm{g} & \mathrm{h} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{f} \\ \mathrm{f} & \mathrm{g} & \mathrm{h} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} \\ \mathrm{e} & \mathrm{f} & \mathrm{g} & \mathrm{h} & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} \\ \mathrm{d} & \mathrm{e} & \mathrm{f} & \mathrm{g} & \mathrm{h} & \mathrm{a} & \mathrm{b} & \mathrm{c} \\ \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{f} & \mathrm{g} & \mathrm{h} & \mathrm{a} & \mathrm{b} \\ \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{f} & \mathrm{g} & \mathrm{h} & \mathrm{a} \end{bmatrix}$$

Circulant matrices are a special case of the Toeplitz matrices. As with the Toeplitz matrices, at every level of quadrant submatrices, those that lie diagonally from each other are equal. Moreover, we now see submatrix repetition across the diagonal.

The number of distinct quadrant submatrices of level k for a n level circulant matrix is 2^{n-k}. Thus the LARCsize for the worst case circulant matrix of level n is $\sum_{i=0}^{n} 2^{n-i} = 2 * 2^n - 1$.

In summary, the last four families of matrices considered all grow asymptotically like $O(2^n)$, with the constant multiplier of 2^n being 2 for diagonal and circulant matrices, 4 for Toeplitz matrices and 6 for tridiagonal matrices.

C.3 Exponential growth of the form $O(4^n) = O(N)$

General Matrices

Let M_n be a completely general 2^n by 2^n matrix with every scalar distinct (4^n unique scalars).

As described in the beginning of this section the LARCsize for such a matrix of level n is $\sum_{i=0}^{n} 4^{n-i} = \frac{4^{n+1}-1}{3} = \frac{4}{3}4^n - \frac{1}{3}$.

Symmetric Matrices

Let Sym_n be a completely general symmetric 2^n by 2^n matrix constructed from $2^{n-1}(2^n+1)$ distinct scalars. For example

$$\mathrm{Sym}_3 = \begin{bmatrix} \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{f} & \mathrm{g} & \mathrm{h} \\ \mathrm{b} & \mathrm{i} & \mathrm{j} & \mathrm{k} & \mathrm{l} & \mathrm{m} & \mathrm{n} & \mathrm{o} \\ \mathrm{c} & \mathrm{j} & \mathrm{p} & \mathrm{q} & \mathrm{r} & \mathrm{s} & \mathrm{t} & \mathrm{u} \\ \mathrm{d} & \mathrm{k} & \mathrm{q} & \mathrm{v} & \mathrm{w} & \mathrm{x} & \mathrm{y} & \mathrm{z} \\ \mathrm{e} & \mathrm{l} & \mathrm{r} & \mathrm{w} & 0 & 1 & 2 & 3 \\ \mathrm{f} & \mathrm{m} & \mathrm{s} & \mathrm{x} & 1 & 4 & 5 & 6 \\ \mathrm{g} & \mathrm{n} & \mathrm{t} & \mathrm{y} & 2 & 5 & 7 & 8 \\ \mathrm{h} & \mathrm{o} & \mathrm{u} & \mathrm{z} & 3 & 6 & 8 & 9 \end{bmatrix}$$

As you can see the example, the symmetry $a_{ij} = a_{ji}$ does not lead to any quadrant submatrix repetition beyond the scalar level. At the scalar level, we do have $\frac{1}{2}4^n + \frac{1}{2}2^n$ distinct scalars, as compared to the 4^n distinct scalars for a completely general matrix. Thus the total number of distinct quadrant submatrices for the symmetric case is $\frac{5}{6}4^n + \frac{1}{2}2^n - \frac{1}{3}$.

Note that the structure inside of Hermitian matrices, where a_{ij} is the complex conjugate of a_{ji}, does not translate to any reduction in LARCsize beyond the general case.

D Copyright and Open Source Distribution License