IDA
Center for Computing Sciences

LARC: Linear Algebra via Recursive Compression

Steve Cuccaro, Mark Pleszkoch, and Jenny Zito, IDA/Center for Computing Sciences

https://github.com/LARCmath larc@super.org
LARC
Linear Algebra via Recursive Compression

Linear Algebra via Recursive Compression

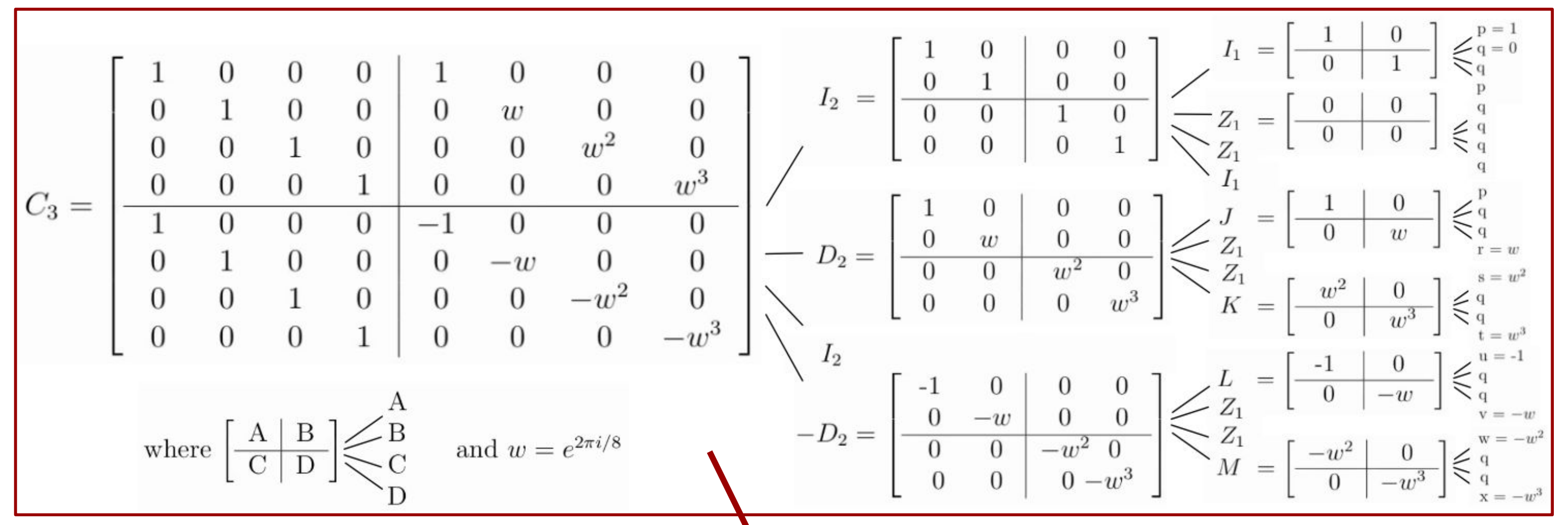
Some applications produce matrices that are too large to store or operate on in standard formats. LARC is a package for matrix math where matrix storage and operations are achieved recursively in compressed format. It uses a locality-sensitive hash technique to handle finite precision issues and mimic symbolic operations (e.g. multiplicative closure for roots of unity). There is a Python tutorial and example package for LARC available on GitHub.

Recursive storage and operations in LARC

We will show how LARC recursively compresses matrices for storage and then carries out matrix operations while in the compressed format by looking at an example application. Here is a recursive expression for the Fast Fourier Transform (FFT) [VL]

$$F_k = C_k \left[\begin{array}{c|c} F_{k-1} & Z_{k-1} \\ \hline Z_{k-1} & F_{k-1} \end{array}\right] P_k$$

where the subscript k (which we shall call the level) indicates a 2^k by 2^k matrix. F_k and F_{k-1} are FFT matrices, Z_{k-1} is all zeros, and C_k and P_k are highly structured sparse matrices. LARC stores matrices by dividing them up into quadrant submatrices and is designed to work best when the quadtree [W] (the tree of submatrices created by recursive subdivision) has some repetition. Consider for example the quadtree of the matrix C_3 which has many repeated submatrices.



$$C_3 = \left[\begin{array}{cccc|cccc} 1&0&0&0&1&0&0&0 \\ 0&1&0&0&0&w&0&0 \\ 0&0&1&0&0&0&w^2&0 \\ 0&0&0&1&0&0&0&w^3 \\ \hline 1&0&0&0&-1&0&0&0 \\ 0&1&0&0&0&-w&0&0 \\ 0&0&1&0&0&0&-w^2&0 \\ 0&0&0&1&0&0&0&-w^3 \end{array}\right]$$

$$I_2 = \left[\begin{array}{cc|cc} 1&0&0&0 \\ 0&1&0&0 \\ \hline 0&0&1&0 \\ 0&0&0&1 \end{array}\right] \quad D_2 = \left[\begin{array}{cc|cc} 1&0&0&0 \\ 0&w&0&0 \\ \hline 0&0&w^2&0 \\ 0&0&0&w^3 \end{array}\right] \quad -D_2 = \left[\begin{array}{cc|cc} -1&0&0&0 \\ 0&-w&0&0 \\ \hline 0&0&-w^2&0 \\ 0&0&0&-w^3 \end{array}\right]$$

$$I_1 = \left[\begin{array}{c|c} 1&0 \\ \hline 0&1 \end{array}\right] \quad Z_1 = \left[\begin{array}{c|c} 0&0 \\ \hline 0&0 \end{array}\right] \quad J = \left[\begin{array}{c|c} 1&0 \\ \hline 0&w \end{array}\right] \quad K = \left[\begin{array}{c|c} w^2&0 \\ \hline 0&w^3 \end{array}\right] \quad L = \left[\begin{array}{c|c} -1&0 \\ \hline 0&-w \end{array}\right] \quad M = \left[\begin{array}{c|c} -w^2&0 \\ \hline 0&-w^3 \end{array}\right]$$

where $\left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]$ and $w = e^{2\pi i/8}$

A unique matrixID and record is assigned to each unique quadtree submatrix. For 1 x 1 matrices the record also contains the scalar value from the matrix. For a larger sized matrix, the record's value is a list of the four matrixIDs of the quadrant submatrices of that matrix (a recursive definition).

Storing C_3 requires 19 MatrixRecords

	matrixID	value	level (size)
p	ID_1	1.0	0 (1×1)
q	ID_2	0.0	0
I_1	ID_3	$[ID_1,ID_2;ID_2,ID_1]$	1 (2×2)
Z_1	ID_4	$[ID_2,ID_2;ID_2,ID_2]$	1
I_2	ID_5	$[ID_3,ID_4;ID_4,ID_3]$	2 (4×4)
r	ID_6	$e^{2\pi i/8}$	0
J	ID_7	$[ID_1,ID_2;ID_2,ID_6]$	1
s	ID_8	$e^{2\cdot 2\pi i/8}$	0
r	ID_9	$e^{3\cdot 2\pi i/8}$	0
K	ID_{10}	$[ID_8,ID_2;ID_2,ID_9]$	1
D_2	ID_{11}	$[ID_7,ID_4;ID_4,ID_{10}]$	2
⋮	⋮	⋮	⋮
C_3	ID_{19}	$[ID_5,ID_{11};ID_5,ID_{18}]$	3 (8×8)

In LARC's FFT implementation we need C_k and P_k matrices for all $k < K$ the *max level*. Subscripts k are levels, matrix size is n x n with $n = 2^k$.

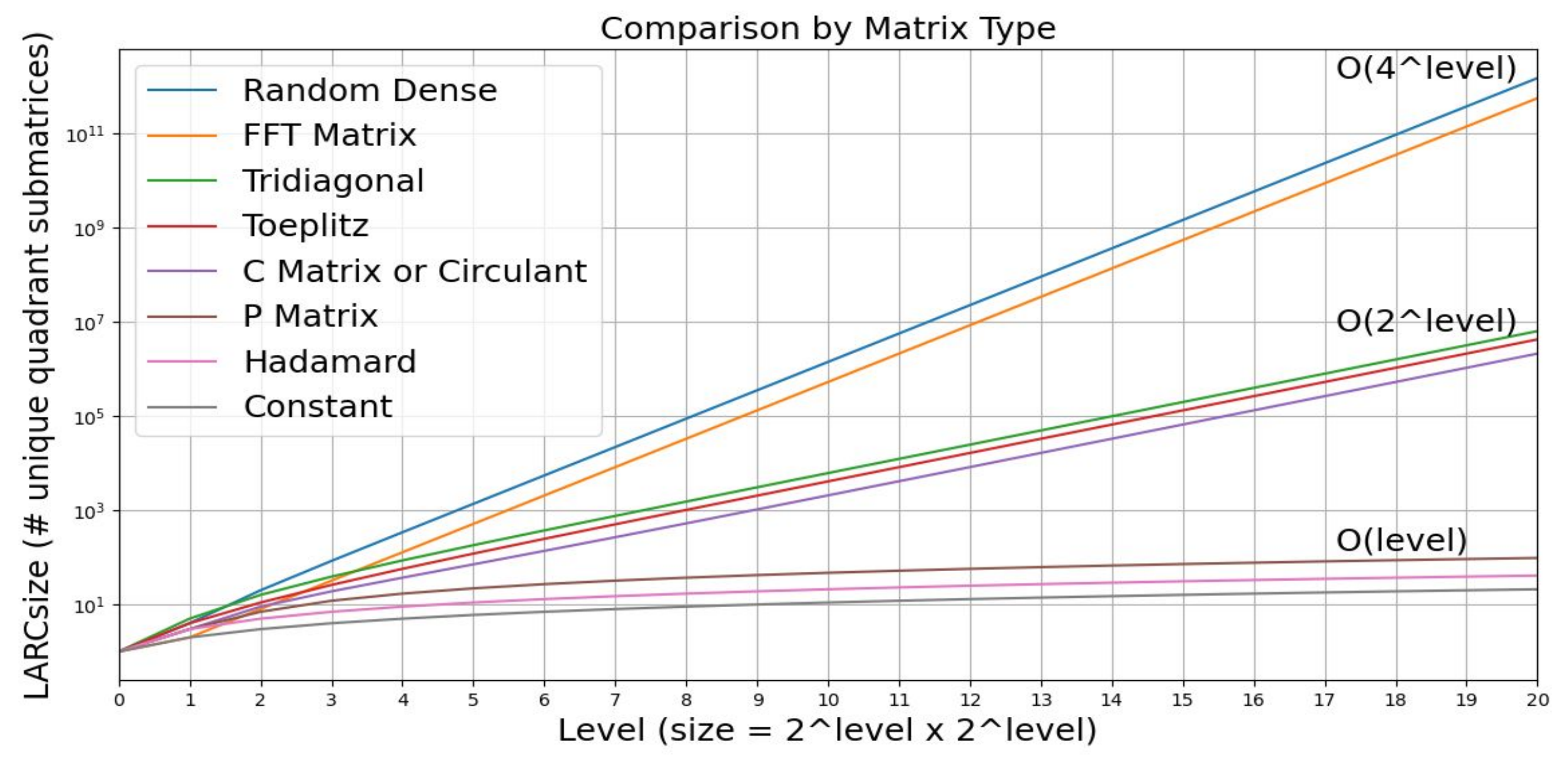
$$C_k = \left[\begin{array}{c|c} I_{k-1} & D_{k-1} \\ \hline I_{k-1} & -D_{k-1} \end{array}\right]$$

where I_{k-1} is a $2^{k-1} x\, 2^{k-1}$ identity matrix, D_{k-1} is a diagonal matrix with entries $(1, w, w^2, \ldots, w^{(n/2)-1})$ for $w = e^{2\pi i/n}$ the n-th root of unity. The matrix P_k is a permutation matrix, recursively defined by:

$$P_0 = [\,1\,];\ P_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix};\ \text{and } P_{k+1} = \begin{bmatrix} P_{k,0} & P_{k,1} & 0 & 0 \\ 0 & 0 & P_{k,0} & P_{k,1} \\ P_{k,2} & P_{k,3} & 0 & 0 \\ 0 & 0 & P_{k,2} & P_{k,3} \end{bmatrix} \text{ where } P_k = \begin{bmatrix} P_{k,0} & P_{k,1} \\ P_{k,2} & P_{k,3} \end{bmatrix};$$

Compression of Various Matrix Classes

These sparse matrices compress with small "LARCsize", which is the number of matrix records needed to store a matrix, or equivalently, the number of unique quadrant submatrices.



Matrix Operations in Compressed Format

Implementing matrix operations by dividing a matrix into blocks has been used historically to reduce computational cost and more efficiently use computational resources such as cache [St][W]. 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. For example, for matrices A and B with quadrants submatrices labelled as shown, here are the block recursive algorithms for addition and multiplication.

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

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

LARC has block recursive implementations of other matrix operations such as Kronecker product, adjoint (complex conjugate transpose), and trace. The operations are "memoized" for reuse in a hash table indexed by the type of operation and the matrixIDs of the input matrices, so that LARC only need carry out any operation once.

LARC Reduces Numerical Precision Issues by Using a Locality Sensitive Hash

When two or more scalars arise that should be equal, but instead differ by some small amount due to numerical precision issues, not only would LARC have stored multiple copies of what should be the same scalar, it also would have stored the many matrices which would be equal except for these errors. To avoid this and consequent inefficiency of operation memoization, we have developed a new technique involving locality-sensitive hashing. LARC divides the space of scalars into tiny regions based on a user parameter, then the hash for the MatrixRecord table sends any scalar to the hash location determined by the center of the region that scalar is in. LARC will store at most one scalar from any region, and only need search one hash bucket to see if a previous "representative" from that region has been stored.

Mimicking Symbolic Computation in LARC

LARC uses its locality-sensitive hash to mimic symbolic computation for selected mathematical identities. This is achieved by preloading (in full precision) a selection of scalars involved in these identities. In our FFT implementation we preload $\{w^j\}$ the n^{th}-roots of unity; this assures that when LARC calls $\mathrm{product}((\mathrm{matrixID}(w^j), \mathrm{matrixID}(w^k))$ that the function returns $\mathrm{matrixID}(w^m)$ where $m = j+k \ (mod\ n)$. The first scalars we preload into LARC are 0 and 1 because of their importance in identities. We also preload all identity and zero matrices and set flags in their matrix records so that we can quickly identify and short cut any operation identities.

Algorithm Implementation Options in LARC

When implementing an algorithm on LARC the choice of how to group operations can make a big difference in compression and runtime. We saw in the graph on LARC compression that the FFT matrix is not very compressible, but C_k and P_k are. We have various choices about how to group operations. We can write the FFT recursion in terms of the Kronecker product $\otimes$ as $F_k = C_k\,(I_1 \otimes F_{k-1})\,P_k$ and then recursively substitute in for smaller FFT matrices. We can choose to group the P_i terms

$$\overline{P}_k = (I_{k-2} \otimes P_2) \ldots (I_1 \otimes P_{k-1}) P_k = \prod_{i=2}^{k} (I_{k-i} \otimes P_i)$$

and/or to group the C_i terms

$$\overline{C}_k = C_k (I_1 \otimes C_{k-1}) \ldots (I_{k-1} \otimes C_1) = \prod_{i=0}^{k-1} (I_i \otimes C_{k-i})$$

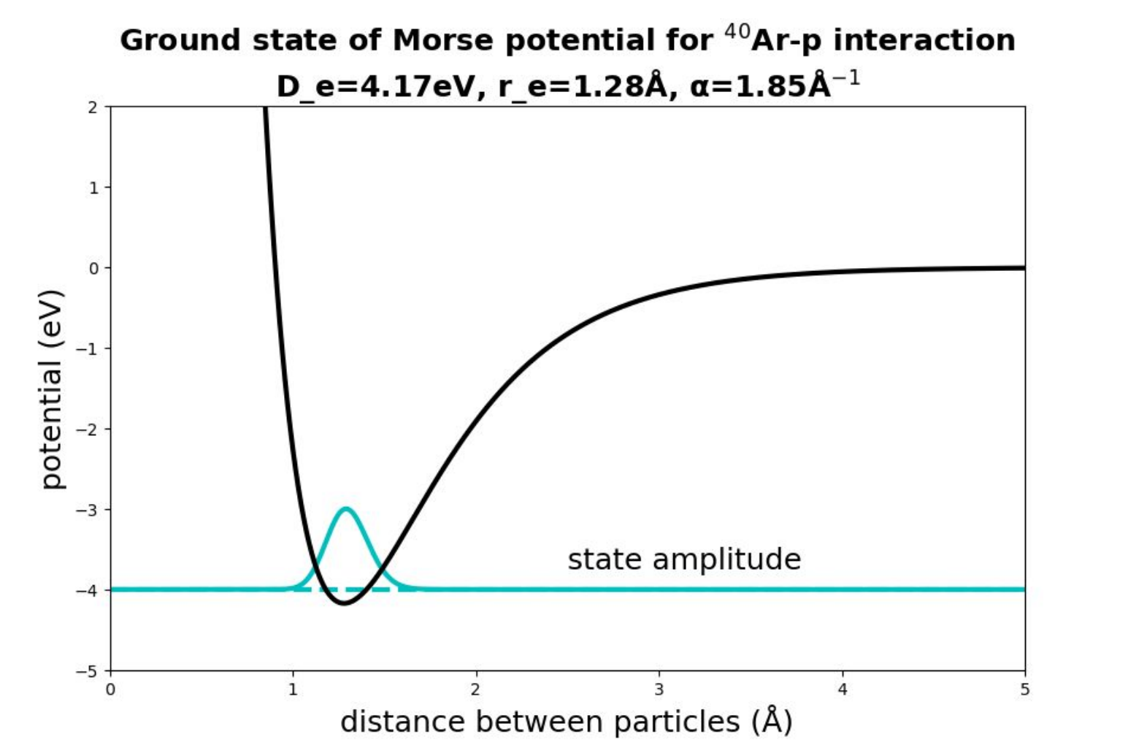
Thus $F_k = \overline{C}_k \overline{P}_k$, $F_k = \prod_{i=0}^{k-1} (I_i \otimes C_{k-i})\,\overline{P}_k$, ...

k	P_k	$\overline{P}_k$	C_k	$\overline{C}_k$	F_k
		…		…	
2	7	7	9	8	9
3	12	12	19	23	29
4	17	28	37	74	101
5	22	45	71	261	373
		…		…	
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 of level *k* versus LARCsize

Depending on platform, memory, and cache, storing $\overline{P}_k$ and the individual C_k may result in the most efficient implementation of the FFT.

Applications and Current Research



In addition to the FFT application featured in this poster, we have several sample applications available. One application is a ground state eigenvector and energy calculation (see figure). The 1-D Schrödinger equation H(f(x)) = E f(x) can be discretized and represented as a matrix eigenvalue problem My=Ey, with M symmetric and tridiagonal. We use the power method to solve for E_0 and y_0.

Other research areas we are exploring are: implementing Kronecker graphs for modeling networks [L10], developing a hybrid Lanczos method, and implementing a locality preserving hash.

Availability of LARC Package and Applications

Version 1.0 of the LARC matrix math package is implemented in C with a user friendly Python wrapper. The code is not currently optimized for any particular application. LARC can be compiled with different scalar types (including standard real, integer, complex; and multiprecision real, rational, complex and complex rational). We also provide the MyPyLARC tutorial and example package (including some Jupyter Notebook examples) which can be copied and modified as a starting point for your own projects. Both MyPyLARC and LARC may be downloaded from GitHub.

```
git clone https://github.com/LARCmath/LARC.git
git clone https://github.com/LARCmath/MyPyLARC.git
```

LARC contains doxygen documentation and there are slides [C17] and an explanatory paper on LARC and MyPyLARC available in the GitHub repositories. MyPyLARC contains tutorials and demos including the FFT and power method. A section in the paper describes other research ideas.

References

[C17] S. Cuccaro, J. Daly, J. Gilbert, and J. Zito, "LARC: Linear Algebra via Recursive Compression", GitHub:LARCmath LARC/doc, AACC, 2017.

[[L10] J. Leskovec et al, "Kronecker Graphs: An Approach to Modeling Networks", J. of Machine Learning Research (2010).

[Sh] A. Shukla and J. Futrell, "Ion Collision Theory", in Encyclopedia of Spectroscopy and Spectrometry, 1999, Academic Press, pp. 954–963.

[St] V. Strassen, "Gaussian elimination is not optimal", Num. Math. 1969.

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

[W] D. Wise and J. Franco, "Costs of Quadtree Representation of Non-dense Matrices," J. Parallel and Distributed Computing, 1990.