Using random projections to accelerate numerical linear algebra with the NAG Library

Philip Maybank *Lawrence Mulholland* *(Numerical Algorithms Group)*

Background

Historically NAG was a key contributor to the design and implementation of the widely used LAPACK software for numerical linear algebra. An implementation of LAPACK was subsequently incorporated into the NAG Library. Recently NAG has been exploring the use of random projection based algorithms for the solution of large-scale numerical linear algebra problems, that might be more efficient than current LAPACK algorithms. Our work to date has mainly focused on the case of dense matrices that fit into core memory, and on evaluating whether Randomized Numerical Linear Algebra (RNLA) algorithms outperform LAPACK in terms of computational efficiency. This work has led to the incorporation of new routines for RNLA into the next release (Mark 27) of the NAG Library. Applications where we expect these routines to be used include PCA and linear regression for large datasets.

Random projections for SVD

Halko *et al* (2011) presented a randomized SVD algorithm that is structured as follows:

1. Apply a random projection to input matrix, $Y = A\Omega$.
2. Apply pivoted QR decomposition to Y.
3. Form SVD by row extraction, $A \approx U\Sigma V^T$.

This algorithm will typically be faster than LAPACK's full SVD on large matrices and is well suited to problems where A is rank deficient, or only a small number of singular vectors / values are needed.

Random projections for iterative least squares

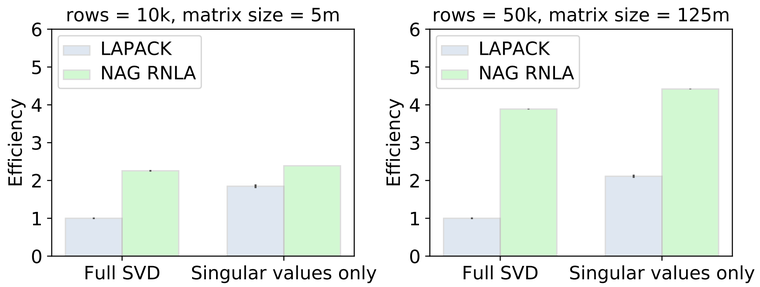
Avron *et al* (2010) presented a randomized least squares solver called Blendenpik. Blendenpik uses an iterative least squares algorithm called LSQR with a preconditioner that is generated as follows,

1. Apply a random projection to data matrix, $Y = A\Omega$.
2. Apply QR decomposition, $Y = QR$.

Then R^{-1} is used as a preconditioner in LSQR. Again this algorithm will typically be faster than LAPACK's least squares solver on large matrices. When A is ill-conditioned, LSQR with no preconditioner requires significantly more iterations to converge. The preconditioner obtained from the random projection solves this problem, i.e., LSQR will typically converge in 20 to 50 iterations even for ill-conditioned A.

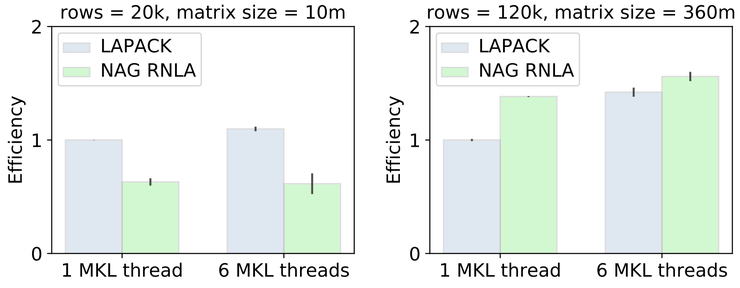
SVD benchmark

Figure 1: SVDs for two different problem sizes. The column dimension was $n = m/20$, the rank was $r = m/100$. The dimension of the random projection was set to $r + 5$. The SVDs were calculated using a pre-release Mark 27 build of the NAG Library for *Python* with MKL. The LAPACK solver is called through `lapackeig.dgesvd` and the NAG RNLA solver is called through `rnla.svd_rowext_real`. The relative timings are similar for single-threaded and multithreaded implementations (results not shown). The results were generated on a Windows machine with 16GB of RAM and 6 cores.



Iterative least squares benchmark

Figure 2: least squares solvers for two problem sizes and a fixed condition number of 10^6. The column dimension was $n = m/40$. The dimension of the random projection was set to $10\,n$. As above, the least squares problem was solved using a pre-release Mark 27 build of the NAG Library for *Python*. The LAPACK solver is called through `lapackeig.dgels` and the NAG RNLA solver is called through `linsys.real_gen_sparse_lsqsol` and `rnla.randproj_dct_real`. The timings were similar for a range of condition numbers (results not shown), and were generated on the same machine as the SVD benchmark.



Getting started with the NAG Library

Search for NAG Library for *Python* documentation, and check out the `NAGPythonLibraryTraining` repository on GitHub. Written as Jupyter notebooks, the training materials are designed to demonstrate a range of algorithms that are available in the NAG Library. The NAG Library for *Python* calls compiled Fortran code. Performance is similar to code that is written exclusively in Fortran. The NAG Library for *Python* has consistent user-friendly interfaces and documentation for a comprehensive library of algorithms. The NAG Library is an attractive alternative to open-source packages both for researchers prototyping novel algorithms and software developers writing production code.

How to contribute to the NAG Library

Students and academic researchers regularly contribute to NAG's work in a number of different ways. For example:

- Undergraduate student placements with programming experience adapt existing numerical codes for inclusion in the NAG Library.
- Graduate student placements in mathematical sciences review published academic research to identify and then implement new algorithms for the NAG Library.
- NAG-sponsored PhD students develop novel algorithms that are then implemented by numerical software developers at NAG.
- Knowledge Transfer Partnerships (KTPs) between NAG and universities fund postdoctoral researchers.
- Senior academics are invited to give technical seminars at NAG that help to shape the direction of NAG's development work.

Many of the skills and knowledge required to effectively write NAG Library code can be picked up within a few months assuming a moderate level of numerical programming experience and familiarity with command-line tools.

References

Avron H, Maymounkov P, Toledo S, (2010) Blendenpik: Supercharging LAPACK's least squares solver, *SIAM Journal on Scientific Computing*.

Halko N, Martinsson P G, Tropp J A, (2011) Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions, *SIAM review*.