Numerical Techniques for Computing the Inertia of Products of Matrices of Rational Numbers*

John P. May
University of Delaware
Newark, DE, USA
jpmay@udel.edu

B. David Saunders
University of Delaware
Newark, DE, USA
saunders@udel.edu

David Harlan Wood
University of Delaware
Newark, DE, USA
wood@udel.edu

ABSTRACT

Consider a rational matrix, particularly one whose entries have large numerators and denominators, but which is presented as a product of very sparse matrices with relatively small entries. We report on a numerical algorithm which computes the inertia of such a matrix in the nonsingular case and effectively exploits the product structure. We offer a symbolic/numeric hybrid algorithm for the singular case. We compare these methods with previous purely symbolic ones. By "purely symbolic" we refer to methods which restrict themselves to exact arithmetic and can assure that errors of approximation do not affect the results. Using an application in the study of Lie Groups as a plentiful source of examples of problems of this nature we explore the relative speeds of the numeric and hybrid methods as well as the range of applicability without error.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Algorithm Design and Analysis; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra; I.1.4 [**Symbolic and Algebraic Manipulation**]: Applications

General Terms

Algorithms, Design, Performance

Keywords

lie group, matrix signature, product eigenvalue problem, symbolic/numeric hybrid method

1. INTRODUCTION

This is a study of the computation of the inertia of a rational matrix by a combination of exact linear algebra and numeric methods. Matrix inertia is a central concept of linear algebra. It is closely related to questions of Lyapunov stability, generalizing the discussion of stable, positive definite, and semi-definite operators. It plays a significant role in many applications.

The *inertia* of a real symmetric matrix (or Hermitian complex matrix) is the triple (n, z, p) whose entries are the numbers of, respectively, negative, zero, and positive eigenvalues. Real matrices A, B are *congruent* if there exists an invertible real matrix C such that $B = CAC^T$. By Sylvester's law of inertia [13; 8, §8.1.5], matrices are congruent if and only if they have the same inertia. Thus the inertia characterizes the congruence equivalence classes and every matrix is congruent to a unique diagonal matrix of the form

$$\begin{pmatrix} I_p & 0 & 0 \\ 0 & -I_n & 0 \\ 0 & 0 & 0_z \end{pmatrix}.$$

Because it characterizes conjugacy and because of its relation to stability in control theory and other domains, the inertia of a linear operator is an intensely studied concept. It turns out to be relevant to the study of Weyl groups as well. We use examples from this context to motivate and illustrate the symbolic/numeric methods we propose here. This work extends a previous effort using only the exact linear algebra methods [1].

The actual computational problem we consider is an instance of finding the inertia of a product of matrices

$$A = A_k A_{k-1} \cdots A_1. \tag{1}$$

A number of authors have sought information about such products of matrices including various decompositions [4, 9, 16, 17, 18, 19, 20, 21, 23] A recent review of theoretical results and numerical algorithms in this area [23] refers to this subject area as "product eigenvalue problems." A unifying theme is to extract desired results by means of processing the individual factors

$$A_k, A_{k-1}, \ldots, A_1$$

without explicitly forming the product matrix, A.

In section 2 we briefly introduce the Weyl group application. In section 3 we provide basic definitions and then summarize the available symbolic, or exact arithmetic, methods including one based on the product form. We also exploit the product viewpoint for the numeric approach described in sections 4 and 5. Computational results are reported in section 6. Our conclusion provides cautious optimism about the approach we have developed here and suggests several directions for further development.

*Research was supported by National Science Foundation Grant CCF-0515197

2. THE APPLICATION

Weyl groups have bases of reflections and are characterized by corresponding root systems. One of the questions concerning these is the unitary dual of the associated Hecke algebra. The irreducible root systems belong to 4 families A_n, B_n, C_n, D_n and 5 exceptional cases E_6, E_7, E_8, F_4, G_2. Of these E_8 has the largest Weyl group. It is of order 696,729,600. This group has 112 irreducible matrix representations, the largest of which is by 7168×7168 matrices. For each representation ρ and facet ν in the space of the root system an operator $J_\rho(\nu)$ is defined whose inertia reveals whether or not the corresponding representation of the Hecke algebra is unitary. For a full exposition of all this structure, see [14]. Through this connection, matrix inertia contributes to the study of unitary duality. For E_8, there are 1,070,716 facets of the root system. Multiply this by the number, 112, of irreducible representations to get the number of matrices whose inertia is of interest. The unitary representations of the Hecke algebra for E_8 have been characterized [3], so the inertia calculations confirm known results. But this E_8 framework gives us a rich set of examples, prototypical of what is needed to calculate the unitary duals for other real and p-adic Lie groups.

The matrices $J_\rho(\nu)$ for E_8 may be scaled to integer matrices. Measurements in [1] have shown that the integer entries are about $33\lg(n)$ bits long. But importantly these matrices are constructed as a product of 121 very sparse matrices with very short bit lengths for the non-zeroes. It is this product form that we exploit for the numeric method applied in this paper.

3. SYMBOLIC METHODS

The paper of Adams, Saunders, and Wan [1] is motivated by the goals of the Lie Atlas Group regarding E_8 and the other Weyl groups. In that paper the trade offs among three purely symbolic methods were studied. In this section we briefly summarize the two most useful of those methods called DSLU and BBSM. The DSLU is a Dense Symmetric LU decomposition [8, §4.1.2] and BBSM, Black Box Symmetric Minimal polynomial [5], is a blackbox method obtaining the inertia from the coefficient signs of a characteristic polynomial. The new numeric component we introduce in this paper is elimination based (like DSLU) but exploits the product form representation of the matrix (like BBSM). The blackbox method is the asymptotically faster of these two symbolic methods, but it was shown that the crossover in the run times would occur around the largest matrix under consideration. Inertia for matrices of the largest order, 7168, was not computed, but extrapolating from the times for smaller representations it was estimated that the run time would be more than a CPU year. Thus we are motivated to find faster methods or to apply parallelism.

In most cases a symmetric matrix A has a unique LDL^T decomposition where D is diagonal and L is unit lower triangular. When the LDL^T decomposition exists, the inertia is determined by the sign pattern on the diagonal of the congruent matrix D. Also in most cases the inertia can be determined from the sign pattern of the coefficients of the characteristic polynomial of A. The problematic case is when strings of successive zero coefficients occur. In our examples we have not encountered this problem.

For integer symmetric matrix A, the DSLU method is to compute the LU decomposition of A modulo a series of convenient (word sized) primes and obtain the diagonal D of the LDL^T by combining the diagonal matrices D_p of $A = L_p D_p L_p^T \pmod p$ using the Chinese remainder algorithm, CRA. If a symmetric matrix has an LU decomposition without permutations where L is unit lower triangular and U is upper triangular, then $U = DL^T$ where D is diagonal. Then we see that A is congruent to D so that the inertia is easily read off from the sign pattern of the entries of D. It is possible that A may not have an LU decomposition. In that case relatively simple adjustments can be made such as random conjugacy preconditioning, but that has not proven necessary in our examples. The i^{th} diagonal entry of D is a quotient of the i^{th} leading principal of A over the $i-1$ leading principal minor. Thus sizes are bounded by Hadamard's bound. The "early termination" strategy can be used in the CRA process. In our examples this saves about a $O(\lg(n))$ factor in the run times. The run time of DSLU for a dimension n matrix with entries of size $O^\sim(n)$ entries is in $O^\sim(n^4)$.

The BBSM method uses the matrix A in product form, $A = A_k A_{k-1} \cdots A_1$, where $k = 121$. The only computation required of the matrix is the matrix-vector product $y = Ax$. This may be computed by successive application of the sparse factors A_i. In practice, some of the A_i are diagonal and may be multiplied by adjoining factors without changing sparsity. When this is done for the representation of E_8 that we are using, $k = 94$ factors result. The idea of BBSM is to compute the characteristic polynomial of A or a diagonally congruent matrix. This is done by use of Wiedemann's algorithm [5, 15, 24] for the minimal polynomial including diagonal preconditioning if necessary to get the minimal polynomial to reveal the characteristic polynomial. It turns out that preconditioning is seldom necessary in our examples. Again, the polynomial computations are done modulo word sized primes and CRA is used to combine the images to get the integer characteristic polynomial. The coefficients of the characteristic polynomial are sums of minors, so again the number of primes can be determined either by upper bound calculation or the early termination strategy. The BBSM algorithm runs in $O^\sim(n^3)$, however this $O^\sim(n^3)$ comes with much larger constant coefficient (and $\lg(n)$ factors) than DSLU's $O^\sim(n^4)$. Two major components contribute to the unrevealed factor in the $O^\sim(n^3)$. One is that the fast floating point BLAS linear algebra kernels are used in the DSLU algorithm. The other is that matrix vector product is done $2n$ times in BBSM and the cost of each one is represented as $O^\sim(n)$. However the cost of an application of A to a vector is application of 121 sparse matrices each with about $n\lg(n)$ entries. So a hidden factor of around $2 \times 121 \times \lg(n)$ is present overall.

As a result, the algorithm to beat for matrices of the size we consider is DSLU, symbolic dense elimination. Thus we try to find improvements over DSLU because it is still too slow to compute all the inertias that must be computed for the study of Lie group representations. As reported in [1], the run times of DSLU are about a minute for $n = 200$, an hour for $n = 1000$, and a year for $n = 7168$.

4. NUMERIC APPROACH FOR THE NONSINGULAR CASE

The primary characteristic of symbolic computation with

rational numbers is that the computations are slow but *exact* — perhaps more exact than needed. The primary characteristic of numerical computation with floating point numbers is that the computations are fast but *approximate* — perhaps more approximate than needed. Hybrid symbolic/numerical computation seeks to blend speed with adequate accuracy.

Finding the inertia of a symmetric matrix of rational numbers would seem to require only modest accuracy. After all, consider the analogous problem for a polynomial with rational coefficients and real roots. Finding the "inertia" of a polynomial, that is, the number of roots that are negative, zero, and positive requires very little arithmetic with minimal accuracy. For this problem, rational arithmetic has excessive accuracy in that the inertia does not even depend on the exact coefficients of the polynomial, but only the pattern of their signs. In contrast, finding (bracketing) the roots of such a polynomial may require arbitrary high accuracy.

In the case of matrices rather than polynomials, it is known that finding the inertia of a matrix is far from trivial [11, 12]. This also implies that it cannot be easy to find the characteristic or minimal polynomial of a matrix [10]. Still, it would seem that finding the inertia of a matrix, which is trivial when its characteristic polynomial is known, should be easier than finding its eigenvalues, which is hard even when its characteristic polynomial is known.

Two exact methods for finding the inertia were presented in the previous section. In addition, there is an exact symbolic/numeric method [7]. Briefly, in that method, one truncates a matrix A to floating point and finds an approximate matrix of eigenvectors, Q. Then, using exact arithmetic, one computes Q^TAQ, which has the same inertia as A by the Sylvester law of inertia. The matrix Q^TAQ is not expected to be exactly diagonal. Nevertheless, one modifies this product with another congruence transformation $N^T(Q^TAQ)N$, where N is a diagonal matrix chosen to normalize the diagonal elements of this matrix to be near one. This exactly computed result is then truncated to floating point and the process is repeated.

Each iteration of the process requires $O(n^3)$ floating point eigendecompositions and $O(n^3)$ exact matrix products. The computational costs of such exact matrix products would depend on the size of the integers in the matrix A. Let n be the size of the symmetric matrix A of integers, and let $||A||_2$ denote the spectral norm of A, and let $\epsilon \approx 10^{-16}$ characterize the accuracy of floating point arithmetic. The total number of iterations for this method is bounded by $n \lg||A||_2/\lg(1/\epsilon)$. This bound includes the effect of large integer entries through their effect on $||A||_2$.

This would not be a practical method for many problems involving the product of a number of matrices. Let the number of matrices to be multiplied together be k. First of all, the product matrix would have to be computed using k exact multiplications of matrices with increasing large entries. For estimating the number of iterations, we note that in the worst case $||A||_2$ can be as large as the kth power of the largest norm among the factor matrices. Thus, the computational complexity is dominated by $n \lg \max_i ||A_i||_2{}^k/\lg(1/\epsilon)$ times the complexity of exact multiplications of matrices with large integer entries plus floating point eigendecompositions using $O(n^3)$ flops. The method presented below involves a total of $O(k\ n^3)$ floating point operations and, in practice, requires less time than forming the exact product of the A_is.

Methods that we will discuss in this section allow the factor matrices in the product $A = A_kA_{k-1}\cdots A_1$ to be rectangular and/or non-symmetric. For the factor matrices, eigendecompositions and their SVD generalizations would provide scant benefit. We want to find the inertia of a product of matrices. Even if the factor matrices are all real and square, eigendecompositions of the factors generally tell us little about the inertia of their product. Hence, we look to other decompositions and seek to exploit the product structure.

Because of the low operation counts of the usual algorithms, the method of choice to obtain the inertia of a symmetric real matrix is to compute the decomposition

$$A = LDL^T. \tag{2}$$

In numerical analysis, one refers to this as the $LDLT$ decomposition. In this decomposition, D is a diagonal matrix, L is a lower triangular matrix with ones on its diagonal and L^T denotes its transpose. If A is nonsingular, so is L. In this case L and D are uniquely determined by the matrix A. By the Sylvester law of inertia, A and $D = L^{-1}AL^{-T}$ have the same inertia, where L^{-T} denotes the inverse of the transpose of L. Naturally, the inertia of the diagonal matrix D is obvious, as are its eigenvalues.

The decomposition in (2) is a good method for finding the inertia of a matrix A if, firstly, the decomposition exists, and secondly, if the diagonal of the matrix D can be computed accurately enough to determine which of its elements are negative, zero, and positive. We present two examples.

The first example matrix,

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \tag{3}$$

does not have a $LDLT$ decomposition. Row or column pivoting (exchanging) produces the identity matrix, but this does not have the same inertia.

In a second example, D is computed too inaccurately to determine the number of positive elements of the diagonal matrix D. MATLAB has a datatype for exact integers of limited size, so that entering the following matrix with $m = 10^8$ produces exactly

$$\begin{pmatrix} m-1 & m \\ m & m+1 \end{pmatrix} = \begin{pmatrix} 99999999 & 100000000 \\ 100000000 & 100000001 \end{pmatrix}. \tag{4}$$

The matrices L and D of the $LDLT$ decomposition found with a MATLAB computation have exact integers on their diagonals. However, one diagonal entry of D is the positive integer 99999999, and the other diagonal entry is 0. Thus, the computed D does not give the inertia correctly. A zero eigenvalue is impossible because the determinant of the matrix we entered is exactly -1, as is clear from the left hand side of (4).

For a matrix A expressed as a product of matrices,

$$A = A_kA_{k-1}\cdots A_1, \tag{5}$$

we develop a LDL^T decomposition of A in the following

way:

$$\begin{aligned} A &= A_k A_{k-1} \cdots A_3 A_2 A_1 \\ &= A_k A_{k-1} \cdots A_3 A_2 (L_1 D_1 U_1) \\ &= A_k A_{k-1} \cdots A_3 (A_2 L_1)(D_1 U_1) \\ &= A_k A_{k-1} \cdots A_3 (L_2 D_2 U_2)(D_1 U_1) \\ &= A_k A_{k-1} \cdots (A_3 L_2)(D_2 U_2)(D_1 U_1) \\ &\quad \cdots \\ &= (A_k L_{k-1}) \cdots (D_3 U_3)(D_2 U_2)(D_1 U_1) \\ &= (L_k D_k U_k) \cdots (D_3 U_3)(D_2 U_2)(D_1 U_1) \\ &= L_k (D_k U_k) \cdots (D_3 U_3)(D_2 U_2)(D_1 U_1) \\ &= L_k [D_k \cdots D_3 D_2 D_1] L_k^T. \end{aligned} \tag{6}$$

The above flow of equations ended rather abruptly, but the final transition depends on two simple insights. Firstly, The penultimate equation is a product of upper triangular matrices, aside from its first factor L_k. The *diagonal* of any product of upper triangular matrices is simply the product of the diagonals. In the present case, the diagonal of the product is the product of the D_i matrices, because the U_i matrices have all ones on their diagonals. This product of the D_i appears in the last equation. Secondly, no computation is needed to identify the remaining upper triangular matrix that remains on the right of the product of the D_i in (6). This matrix has to be the transpose of L_k because of the uniqueness of the $LDLT$ decomposition of (2).

In general, the sparsity of the matrices A_i in (6) will have little benefit. The operations count for (6) is bounded by k, the number of $n \times n$ matrices A_i times the complexity of a matrix multiplication and computing a LDU decomposition, namely, $O(n^3)$ flops.

The above algorithm, which we will denote NPLU, has the useful feature of processing the individual factors in $A_1, \ldots, A_{k-1}, A_k$ without explicitly forming their product. Instead, the k products $A_{i+1} L_i$ are formed.

We now give an alternative algorithm in which the products $A_{i+1} Q_i$ are formed, where Q_i is a real unitary matrix. The advantage is that multiplication of a matrix A_{k+1} by a unitary matrix does not change the norm or the condition number of A_{k+1}. In contrast, multiplication by a non-unitary matrix can increase both its norm and its condition number. Such increases are not necessarily detrimental, but increases can be avoided as a precaution by using a QR decomposition

$$A = QR, \tag{7}$$

where Q is unitary and R is upper triangular.

For a product of matrices, (6) can be replaced with a scheme that generates the QR decomposition of

$$A = A_k A_{k-1} \cdots A_1. \tag{8}$$

Of course, we ultimately want a $LDLT$ decomposition of A, to determine its inertia, but we first get a QR decomposition of A and then, at the very last moment, we find a LDU decomposition of Q. The overall scheme is very much like (6), but with a special twist at the end. The scheme goes as follows.

$$\begin{aligned} A &= A_k A_{k-1} \cdots A_3 A_2 A_1 \\ &= A_k A_{k-1} \cdots A_3 A_2 (Q_1 R_1) \\ &= A_k A_{k-1} \cdots A_3 (A_2 Q_1) R_1 \\ &= A_k A_{k-1} \cdots A_3 (Q_2 R_2) R_1 \\ &= A_k A_{k-1} \cdots (A_3 Q_2) R_2 R_1 \\ &\quad \cdots \\ &= (A_k Q_{k-1}) \cdots R_3 R_2 R_1 \\ &= (Q_k R_k) \cdots R_3 R_2 R_1 \\ &= (Q_k) R_k \cdots R_3 R_2 R_1 \\ &= (L_{k+1} D_{k+1} U_{k+1}) R_k \cdots R_3 R_2 R_1 \\ &= L_{k+1} [D_{k+1} \mathrm{diag}(R_k) \\ &\quad \cdots \mathrm{diag}(R_3) \mathrm{diag}(R_2) \mathrm{diag}(R_1)] L_{k+1}^T. \end{aligned}$$

The third from last equation gives the QR decomposition of A. To get the desired LDU decomposition of the product matrix A, we perform a final LDU decomposition of Q_k. We then go on to obtain the last equation just as we did in (6) obtaining the diagonal part of the $LDLT$ decomposition and the upper triangular L_{k+1}^T matrix on the far right.

Again, the sparsity of the matrices A_i in (9) will have little benefit. The operations count for (6) with k matrices A_i of size $n \times n$ is bounded by $O(k\ n^3)$ flops.

We will call this version of NPLU using the QR decomposition NPLUQ. In practice we have found that NPLU is faster, but NPLUQ produces better accuracy so we generally prefer it. In the pseudo-code description of NPLUQ given below, we assume that the input matrix is non-singular. If needed non-singularity can be probabilistically certified as described in the following section.

Algorithm: NPLUQ: Numerical Product LU via QR

Input: a list of k $n \times n$ matrix factors: $\{A_i\}$ whose product is symmetric, non-singular, and admits an LU decomposition

Output: The inertia of the product $\prod_i A_i$

Step 1. Set $S_i \leftarrow 1$, for $i = 1 \ldots n$.

Step 2. FOR i FROM 1 TO k

Step 2a. Compute $QR = A_i$.

Step 2b. Determine if the signs of the diagonals of R were computed accurately. If not, return "Fail".

Step 2c. Set $S_j \leftarrow S_j \cdot \mathrm{sign}(R_{j,j})$, for $j = 1 \ldots n$.

Step 2d. Set $A_{i+1} \leftarrow A_{i+1} \cdot Q$.

Step 3. Where $QR = A_k$, compute $LDU = Q$.

Step 4. Set $S_j \leftarrow S_j \cdot \mathrm{sign}(D_{j,j})$, for $j = 1 \ldots n$.

Step 5. Return the inertia $neg = \#\{i \mid S_i < 0\}$, $zeros = 0$, $pos = \#\{i \mid S_i > 0\}$.

A technique one can use in Step 2b is to compute the QR decomposition of A_i^{-1} and compare the signs in $\mathrm{diag}(R)$ to those in the original decomposition of A_i.

It is possible to certify that the diagonals of R are computed correctly [22] but because the diagonals of R are only unique up to their signs [8, Theorem 5.2.2] and those signs

can be chosen in various ways [25], this may not help. In order to assess the accuracy of the signs one needs to study the LAPACK [2] routine `DGEQRF` (called by MATLAB to compute R) to find how it chooses the signs for R.

5. HYBRID APPROACH FOR THE SINGULAR CASE

In the case that the matrix A is not of full rank, an additional technique is needed since the LDLT decomposition will require pivoting to avoid 0's (except in the generic rank profile case). A matrix has *generic rank profile* if the leading principal minors are nonzero up to the rank. As seen with (3), a one-side permutation factor introduced to provide pivoting will cause the inertia of D to no longer be equivalent to that of A. To address this problem, we employ a symbolic-numeric hybrid method.

Though the matrix A has very large entries, its dimension is relatively small and the entries of its factors A_i are also relatively small. Thus, it is very efficient to compute the rank of A by choosing a random prime, p, reducing the A_i modulo p, and then computing the rank of their product using an LU decomposition modulo p. With high probability this will be the rank of the original rational A [6]. If A is not of full rank, by tracing an LU computation with symmetric pivoting we can also try to recover a truncated permutation P so that PAP^T is $r \times r$ and has full rank modulo p, and thus full rank over the rationals as well. This truncated permutation P consists of the first r rows of a permutation Q such that QAQ^T has generic rank profile.

In many examples from the Lie Atlas application, A has quite low rank. In one example, the 7168×7168 matrix has rank 4 (though the entries of the 4×4 matrix PAP^T are rational numbers several thousand bits in size). In this case (and many other less extreme cases), the exact algorithm described in Section 3 will work well. In a more typical example, the rank is much higher. In these higher rank cases, the algorithm NPLUQ described in Section 4 can be used, but with the first factor replaced with $A_1 P^T$, and the last factor replaced by $P A_k$, where P is the truncated permutation described above. This does not reduce to the $r \times r$ case, instead, the algorithm computes $n \times r$ matrices Q and $r \times r$ diagonals D at each step (for each A_i). Doing this, we get HPLUQ, the symbolic-numeric hybrid algorithm described below.

Algorithm: HPLUQ: Hybrid Product LU via QR

Input: a list of k $n \times n$ matrix factors: $\{A_i\}$ whose product is symmetric

Output: The inertia of the product $\prod_i A_i$

Step 1. Choose a random prime p, and compute the LU decomposition $A = \prod_i A_i$ modulo p (allowing pivoting as necessary). From the diagonal of U determine the rank r.

Step 2. IF $r < n$

Compute the LU decomposition modulo p again using symmetric pivoting (return "Fail" if not possible – e.g. all diagonals 0). Record the pivots to compute P, an $r \times n$ truncated permutation P so that PAP^T has rank r.

ELSE IF $n = r$

Set P to the identity.

Step 3. Set $A_1 \leftarrow A_1 P^T$, $A_k \leftarrow P A_k$.

Step 4. Call NPLUQ on $\{A_i\}$ and return the computed inertia with $zeros = n - r$.

6. EXPERIMENTAL RESULTS

We have implemented the numerical algorithm NPLUQ (described in Section 4) in MATLAB and tested it on a large set of examples from the Lie Atlas problem.

The codes for the exact examples in this section were written in the LINBOX library. Both the numerical and exact examples were run on a machine with two 3.20 GHz Intel Dual Xeon processors and 6 GB of shared RAM. The computations used just one processor. As in Section 3, for all of these examples, the number of matrix factors is $k = 94$.

Table 1 shows the results of the MATLAB code run on all the representations of dimensions 200 to 500 for two facets ν_B and ν_C in the root space. The numbers in the matrix names in the first column are numbers assigned to representations in order of their dimension. The column "Bits" is the number of bits needed for each rational entry of the product matrix A. Both facets in table 1 are known to have full rank operators. The second facet

$$\nu_C = \left[0, \frac{1}{40}, \frac{1}{20}, \frac{3}{40}, \frac{1}{10}, \frac{1}{8}, \frac{3}{20}, \frac{23}{40}\right]$$

is known to have positive definite operators while the first facet

$$\nu_B = \left[\frac{1}{108}, \frac{7}{108}, \frac{11}{108}, \frac{79}{540}, \frac{109}{540}, \frac{149}{540}, \frac{209}{540}, \frac{751}{540}\right]$$

has about equal numbers of positive and negative eigenvalues. For comparison, timings for the exact DSLU algorithm are given. In order to check the accuracy of the numerical output, we compared the signs of the diagonals in the computed LDL^T decompositions. The column "Err #" reports the number of elements of the numerical D with the wrong sign. "Err %" gives this same number as a percentage of the total number of non-zero entries of D. In these examples NPLUQ encounters errors only on representation number 30. This suggests that the accuracy of this numerical algorithm depends more on the conditioning of the matrices than on the amount of round off error in the rational to floating point conversion (facet ν_B has larger entries but facet ν_C produces more errors). In figure 1 we can see that the run times of both algorithms grow at about the same quartic rate (the fit curves are degree 4 in n).

Computation on additional representations for facet B are shown in table 2. Some of these differ from results for the same examples in table 1 due to variations in run times from randomizations. These expanded results suggest that the number of miscomputed signs increases with the size of the problem. Also, it can be seen in figure 2 that the exact algorithm has a much greater time cost than the numerical algorithm (the fit lines shown are again degree 4 curves - we did not use a log scale on the time axis in order to emphasize the infeasibility computing the large exact examples).

We also ran the symbolic-numeric algorithm HPLUQ from section 5 on a few known semi-positive definite representa-

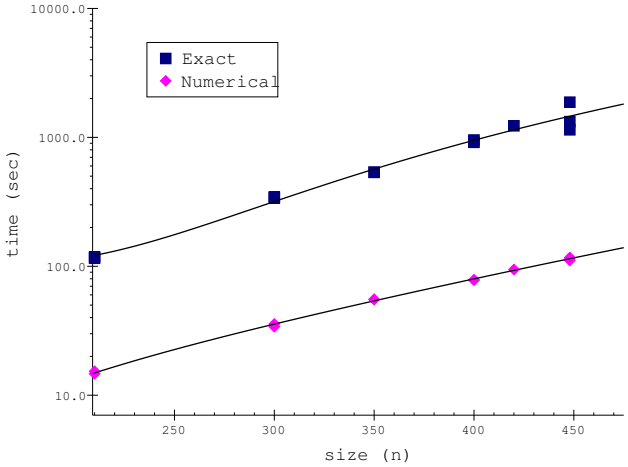

Figure 1: Facet ν_C from Table 1

Matrix	Dim.	Bits	Time NPLUQ	Err #	Err %	Time DSLU
23B	210	1190	13s	0	0%	322s
24B	210	1223	13s	0	0%	431s
25B	300	1215	53s	0	0%	1175s
26B	300	1240	34s	0	0%	1355s
27B	350	1222	57s	0	0%	1568s
28B	350	1238	59s	0	0%	1718s
29B	400	1253	95s	0	0%	2578s
30B	400	1217	84s	2	1%	2171s
31B	420	1251	94s	0	0%	3297s
32B	448	1237	117s	0	0%	4254s
33B	448	1233	117s	0	0%	3102s
34B	448	1252	113s	0	0%	3555s
23C	210	565	15s	0	0%	119s
24C	210	487	15s	0	0%	115s
25C	300	554	36s	0	0%	338s
26C	300	490	34s	0	0%	345s
27C	350	526	55s	0	0%	535s
28C	350	490	55s	0	0%	537s
29C	400	538	78s	0	0%	915s
30C	400	604	78s	200	50%	954s
31C	420	624	94s	0	0%	1229s
32C	448	537	117s	0	0%	1144s
33C	448	610	111s	0	0%	1875s
34C	448	556	116s	0	0%	1324s

Table 1: Times and errors for facets ν_B and ν_C.

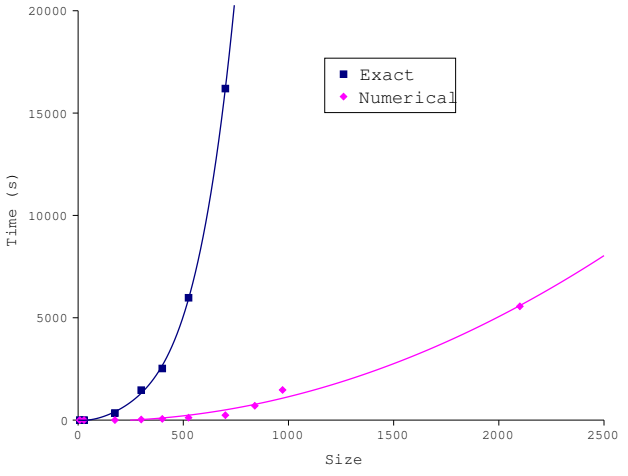

Figure 2: Facet ν_B from Table 2

tions of the facet

$$\nu_A = \left[0, \frac{1}{6}, \frac{1}{6}, \frac{1}{4}, \frac{1}{3}, \frac{5}{12}, \frac{7}{12}, \frac{23}{12}\right].$$

The timings are recorded in table 3. These timings do not include the cost of computing the rank of A and finding the permutation P since this computation is the same in both cases. From the few examples we have computed, it is clear that the projections make the algorithm less stable than the full rank version of the algorithm. However, the preliminary results seem promising. The hybrid algorithm HPLUQ is still quite a bit faster than DSLU so there is a lot of room to add additional processing to correct the errors shown in table 3.

Matrix	Dim.	Bits	Time NPLUQ	Err #	Err %	Time DSLU
03B	8	880	~0s	0	0%	~0s
05B	28	1052	1s	0	0%	2s
21B	175	1221	2s	0	0%	346s
25B	300	1215	29s	0	0%	1464s
30B	400	1217	59s	2	1%	2522s
35B	525	1244	118s	0	0%	5977s
42B	700	1242	239s	0	0%	16197s
45B	840	1249	700s	10	1%	25058s
50B	972	1242	1473s	218	22%	37063s
72B	2100	1259	5559s	?	?	-

Table 2: More runs for facet ν_B

Matrix	Dim.	Rank	Bits	Time HPLUQ	Err #	Err %	Time DSLU
05A	28	6	271	˜0s	0	0%	1s
23A	210	1	321	5s	0	0%	36s
24A	210	45	385	8s	7	16%	57s
25A	300	2	347	13s	0	0%	98s
26A	300	52	409	18s	10	19%	128s
27A	350	3	348	19s	0	0%	150s
28A	350	33	395	24s	6	18%	170s
29A	400	72	419	37s	15	21%	193s
30A	400	8	365	28s	1	13%	145s
42A	700	124	412	102s	18	15%	530s
50A	972	30	405	234s	3	10%	1161s

Table 3: Low rank operators: facet ν_A

7. CONCLUSION

The results presented here are still preliminary. The goal of this project is to be able to compute the inertia of the dimension 7168 representations. We expect that these should take around 5 hours using NPLUQ with double precision floating point arithmetic. However, our current implementation requires both further optimization and more robust error detection. We expect that by increasing the computation time (for example by employing extended precision floating point approximations of the A_is and computations of the QR factorizations) we can compensate for errors and still be able to compute the inertia of the dimension 7168 case in the order of CPU days (extrapolating from table 2) rather than the CPU year projected in [1] for the full symbolic solution by DSLU.

Our experiments thus far seem to indicate that errors in the numerical inertia calculations increase with the size. The data seems to indicate that errors come from factors A_i which have high condition number. Representation 30 in table 1 illustrates this well. It is quite difficult for the numerical algorithm but it is not larger than many other similarly sized examples. We are pursuing several strategies to handle these errors. For examples, by estimating the condition, we hope to be able to diagnose computations that were unstable and may have miscomputed the signs of $\mathrm{diag}(R)$, and then switch to extended precision computations or symbolic techniques in only these unstable cases.

Another strategy we will explore is combining more factors to reduce the number of QR computations in the algorithms NPLUQ and HPLUQ. We hope that the factor of two (or more) speed up that could result would offset any additional processing that would be required to handle any additional instability introduced.

8. REFERENCES

[1] J. Adams, B. D. Saunders, and Z. Wan. Signature of symmetric rational matrices and the unitary dual of Lie groups. In *ISSAC 05 Proc. 2005 Internat. Symp. Symbolic Algebraic Comput.*, pages 13–20. ACM Press, 2005.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK's user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[3] D. Barbasch. Unitary spherical spectrum for split classical groups. *ArXiv Mathematics e-prints*, September 2006. http://arxiv.org/abs/math/0609828.

[4] A. Bojanczyk, G. Golub, and P. van Dooren. The periodic Schur decomposition: algorithms and applications. *Proceedings of the SPIE*, 1770:31–42, 1992.

[5] L. Chen, W. Eberly, E. Kaltofen, W. J. Turner, B. D. Saunders, and G. Villard. Efficient matrix preconditioners for black box linear algebra. *Linear Algebra and Applications*, 343-344:119–146, 2002.

[6] J-G. Dumas, B. D. Saunders, and G. Villard. Integer Smith form via the valence: Experience with large sparse matrices from homology. In *ISSAC 00 Proc. 2000 Internat. Symp. Symbolic Algebraic Comput.*, pages 95–105. ACM Press, 2000.

[7] S. Fortune. Exact computations of the inertia symmetric integer matrices. In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 556–564, New York, NY, USA, 2000. ACM Press.

[8] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, Maryland, third edition, 1996.

[9] M. T. Heath, A. J. Laub, C. C. Paige, and R. C. Ward. Computing the singular value decomposition of a product of two matrices. *SIAM Journal of Scientific and Statistical Computing*, 7(4):1147–1159, October 1986.

[10] T. M. Hoang and T. Thierauf. The complexity of the minimal polynomial. In *26th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, Lecture Notes in Computer Science 2136, pages 408–420. Springer-Verlag, 2001.

[11] T. M. Hoang and T. Thierauf. The complexity of the inertia. In *22nd Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Lecture Notes in Computer Science 2556, pages 206–217. Springer-Verlag, 2002.

[12] T. M. Hoang and T. Thierauf. The complexity of the inertia and some closure properties of GapL. In *20th IEEE Conference on Computational Complexity (CCC)*, pages 28 – 37. IEEE Computer Society Press, 2005.

[13] R. Horn and C. R. Johnson. *Matrix Analysis*. Cambridge University Press, Cambridge, UK, 1985.

[14] James E. Humphreys. Reflection groups and Coxeter groups*, volume 29 of* Cambridge Studies in Advanced Mathematics. Cambridge University Press, Cambridge, 1990.

[15] E. Kaltofen and B. D. Saunders. On Wiedemann's method of solving sparse linear systems. In H. F. Mattson, T. Mora, and T. R. N. Rao, editors, *Proc. AAECC-9*, volume 539 of *Lect. Notes Comput. Sci.*, pages 29–38, New York, NY, 1991. Springer-Verlag.

[16] D. Kressner. *Numerical Methods and Software for General and Structured Eigenvalue Problems*. PhD thesis, TU Berlin, Institut für Mathematik, Berlin, Germany, 2004.

[17] D. Kressner. The periodic QR algorithm is a disguised QR algorithm. *Linear Algebra and its Applications*, 417(2–3):423–433, September 2006.

[18] C. Van Loan. A general matrix eigenvalue algorithm. *SIAM Journal on Numerical Analysis*, 12(6):819–834, 1975.

[19] S. Oliveira and D. E. Stewart. Exponential splittings of products of matrices and accurately computing singular values of long products. *Linear Algebra and its Applications*, 309(1–3):175–190, 2000.

[20] D. E. Stewart. A new algorithm for the SVD of a long product of matrices and the stability of products. *Electronic Transactions on Numerical Analysis*, 5:29–47, June 1997.

[21] G. W. Stewart. On graded QR decompositions of products of matrices. Technical Report CS-TR-3263, University opf Maryland, 1994.

[22] G. Villard. Certification of the QR factor R, and of lattice basis reducedness. In *Proc. of ISSAC 20007*. ACM Press, 2007. To appear.

[23] D. S. Watkins. Product eigenvalue problems. *SIAM Review*, 47(1):3–40, 2005.

[24] D. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Trans. Inf. Theory*, IT-32:54–62, 1986.

[25] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Claredon Press, Oxford, UK, 1965.