Efficient Matrix Rank Computation with Application to the Study of Strongly Regular Graphs[*]

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

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

Zhendong Wan
Drexel University
Philadelphia, PA, USA
wan@cs.drexel.edu

ABSTRACT

We present algorithms for computing the p-rank of integer matrices. They are designed to be particularly effective when p is a small prime, the rank is relatively low, and the matrix itself is large and dense and may exceed virtual memory space. Our motivation comes from the study of difference sets and partial difference sets in algebraic design theory. The p-rank of the adjacency matrix of an associated strongly regular graph is a key tool for distinguishing difference set constructions and thus answering various existence questions and conjectures. For the p-rank computation, we review several memory efficient methods, and present refinements suitable to the small prime, small rank case. We give a new heuristic approach that is notably effective in practice as applied to the strongly regular graph adjacency matrices. It involves projection to a matrix of order slightly above the rank. The projection is extremely sparse, is chosen according to one of several heuristics, and is combined with a small dense certifying component. Our algorithms and heuristics are implemented in the LinBox library. We also briefly discuss some of the software design issues and we present results of experiments for the Paley and Dickson sequences of strongly regular graphs.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Performance

Keywords

matrix p-rank, out of core methods

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

1. INTRODUCTION

This is a study of the problem of computing the p-rank of an integer matrix with particular emphasis on the case when the matrix is large and dense, the rank is relatively small, and the prime is tiny (such as $p = 3$). We are motivated by problems arising in the study of the difference sets and partial difference sets of finite groups and their corresponding strongly regular graphs.

Matrix rank has been perhaps the most widely used exact linear algebra computation to date. In any event, it has been often the goal of computations requested by users of the LinBox software library [17] for exact linear algebra computations over the integers and over finite fields. Besides being of interest itself in numerous applications, rank plays an important role in solving singular systems and computing invariants. For example the rank is used in solving large sparse linear systems [15] computing homology groups [8], and in computation of Gröbner bases [10].

For integer matrices the expedient of choosing a word-size ($\leq 2^{32}-1$) prime number at random and computing modulo that prime is fast and sure, avoiding computation with large integers.

For sparse and structured matrices, LinBox uses blackbox methods for rank [21, 14, 3, 9]. These are probabilistic, requiring that a few vectors over $\boldsymbol{GF}(p)$ be chosen at random. For word-size primes this works well. However, if p is too small, it becomes impossible to guarantee that the probability of success is non-zero without moving the computation into an extension field $\boldsymbol{GF}(p^k)$.

For dense matrices Gaussian elimination is used and it is organized for efficient use of the highly tuned floating point BLAS kernel. This works well, even for small primes [7].

However in our application, the matrices are too large for direct elimination, the matrices are too dense for sparse elimination, and, as mentioned above, the small primes confound the probabilistic algorithms used in the blackbox approach. More importantly, perhaps, the very low rank expected invites us to find a more space and time efficient approach suitable to such very low rank. In addition, since these matrices are defined by formula, we can construct portions of the matrix at will. However, it is costly enough to generate an entry of a matrix (generally a few computations in $\boldsymbol{GF}(q)$, for q the dimension of the matrix) that our methods must avoid computing the entries excessively many times.

We were asked to compute 3-ranks for several families of matrices by Qing Xiang, who uses these computations in the study of difference set constructions [20]. In particular, we looked at the Paley and Dickson families of constructions but our methods should apply to the other families mentioned in [20] as well. The matrices in all these families are of order 3^{2k}. Their 3-ranks are around 2^{2k} (exactly this in the Paley case), and generally less than 2^{2k+1}. Values of k up to 4 pose no problems for straight-forward dense methods, while $k = 5$ begins to be a challenge (matrix order 59049). By the methods we develop here $k = 6$ (order 531441) is accomplished and $k = 7$ (order 4782969) is within reach. About half of the matrix entries are nonzero, so sparse representations and methods are of no help. Memory management is a key issue. For instance, $531441^2 = 3^{24}$ bytes is a quarter of a terabyte. For the matrices of order 3^{10} we have sufficient memory for direct elimination. For the matrices of order 3^{12}, blackbox methods still work directly, albeit slowly. At this point the heuristic methods presented in this paper are of significant benefit. For the next order of interest, 3^{14}, the memory and time efficiency we provide is essential and, in fact, use of parallelism and "out of core" strategies become necessary as well.

In the next section we summarize some definitions and basic facts about difference sets and strongly regular graphs that give rise to the matrix p-rank computations that are the motivation of this paper. Next we analyze algorithms for the problem, including a proof that a random projection method works well for small primes, and give a method of lower complexity (assuming the success of a heuristic component). It works well in practice as demonstrated in section 4 on experimental results. We finish with a brief discussion of software design issues for such large matrices and a brief concluding section.

2. DIFFERENCE SETS, STRONGLY REGULAR GRAPHS, AND P-RANK

The matrices considered in this paper arise from the study of difference sets and partial difference sets, which play important roles in several branches of discrete mathematics and coding theory. Difference sets and partial difference sets are constructed from finite groups. In turn these may be derived from finite fields and semifields. They are associated with graphs by the Cayley construction. A semifield has the properties of a field except for associativity of multiplication. Among the questions being pursued in the extensive literature of this area is to determine whether or not various distinct constructions yield isomorphic objects. The p-rank of an associated adjacency matrix of the Cayley graph is a useful invariant. If two constructions lead to matrices with distinct ranks, the objects are inequivalent.

In this section we provide some basic definitions and references on this topic. We also describe the construction of the specific examples we use in the experiments section to measure the effectiveness of the algorithms.

Let G be a finite group of order ν. A k-element subset D of G is called a (ν, k, λ) *difference set* if each non-identity element of G can be expressed as a "difference" (xy^{-1}, $x, y \in D$) of elements of D, in exactly λ ways. A k-element subset D of G is called a (ν, k, λ, μ) *partial difference set* if each non-identity element of D can be expressed as a difference of elements of D in exactly k ways and each non-identity element of $G - D$ can be expressed as a difference of elements of D in exactly μ ways. For example, Let D consist of all nonzero squares of $\boldsymbol{GF}(q)$, where q is a odd prime power. It is a $(q, (q-1)/2, (q-3)/4)$ difference set of the additive group of $\boldsymbol{GF}(q)$ if $q \equiv 3 \mod 4$. It is a $(q, (q-1)/2, (q-5)/4, (q-1)/4)$ partial difference set of the additive group of $\boldsymbol{GF}(q)$ if $q \equiv 1 \mod 4$. For surveys on difference sets and partial difference sets, we refer to [16, 22]. A difference set D in a finite group G is called *skew Hadamard* if G is the disjoint union of D, $D^{(-1)}$, and 1. The Paley difference set mentioned above is an example. More generally, a subset D of a finite group G of order ν ($\nu \equiv 1 \mod 4$) is called a *Paley type difference set* if D is a $(\nu, (\nu-1)/2, (\nu-5)/4, (\nu-1)/4)$ partial difference set.

A graph Γ with ν vertices is said to be a (ν, k, λ, μ)-strongly regular graph if each vertex is joined to exactly k other vertices, and any two adjacent vertices are both joined to exactly λ other vertices and two non-adjacent vertices are both joined to exactly μ other vertices. There is a strong relationship between strongly regular graphs and partial difference sets. Given a set D in additive group G with $0 \notin D$ and $-D = D$, we can construct a Cayley graph which is a graph Γ with elements of G as vertices such that two vertices $x, y \in G$ are joined if and only if $x - y \in D$. If the set in a group is a partial difference set, then its Cayley graph is a strongly regular graph. On the other hand, if a strongly regular graph has a regular automorphism group G, a partial difference set can be obtained. Therefore, strongly regular graphs with regular automorphism groups and partial difference sets are equivalent. The strongly regular Cayley graph constructed from the set of nonzero squares in $\boldsymbol{GF}(q)$ (q is a prime power and $q \equiv 1 \mod 4$) is called the *Paley graph*. Another interesting type of strongly regular graphs is the $\mathcal{P}^*$-graphs. Let q be an even power of a prime congruence to 3 modulo 4, g be a primitive element of $\boldsymbol{GF}(q)$, and the set $S = \{g^j \ : \ j \equiv 0, 1 \mod 4\}$. Then S is a Paley type partial difference set in $(\boldsymbol{GF}(q), +)$. The Cayley graph constructed from S is called the $\mathcal{P}^*$-graph. The $\mathcal{P}^*$-graph construction was used in [20].

Two difference sets D_1, D_2 of a group G are said to be *srg-equivalent* if the corresponding Cayley graphs are isomorphic. The p-rank computation of their adjacency matrices can be used to distinguish srg-inequivalent partial difference sets of $\boldsymbol{GF}(q)$, where q is a power of the prime p.

It was conjectured that Paley difference sets are the only examples of skew Hadamard difference sets in Abelian groups. This was disproved by Ding and Yuan [4]. It was conjectured by René Peeters that Paley graphs of non-prime order are uniquely determined by their parameters and the minimality of their relevant p-rank. This was disproved in [20].

Numerous other strongly regular graphs are constructed from partial difference sets defined as the set of squares in a semifield. In particular, we look at the partial difference sets coming from the Dickson family of finite commutative semifields. The matrix M corresponding to a semifield $(K, +, *)$ has dimension equal to $|K|$. The diagonal entries $M_{i,i}$ are 0 for all i. The entry $M_{i,j}$ is 1 if the difference of the i^{th} and j^{th} elements of K is in D, the set of squares in K, and $M_{i,j}$ is 0 otherwise. We will be computing the p-rank of $M - I$. The simplest example is the Paley matrix when $K = \boldsymbol{GF}(p^k)$. The other example we will consider is the Dickson matrix which exists for orders 3^{2k}. The defining semifield is $K = \boldsymbol{GF}(p^k) \times \boldsymbol{GF}(p^k)$, where '+' is defined coordinate-wise and '$*$' is defined by

$$(a, b) * (c, d) = (ac + gb^{\sigma} d^{\sigma}, ad + bc).$$

Here g is a primitive element in $\boldsymbol{GF}(p^k)$ and σ is a non-trivial automorphism of $\boldsymbol{GF}(p^k)$. This semifield multiplication is plainly commutative. It also distributes over addition, but fails to be associative. For our experiments we used the primitive element chosen by the Givaro [11] implementation of the field $\boldsymbol{GF}(q)$ and we used the Frobenius automorphism $\sigma(x) = x^p$. The p-rank is independent of the primitive element or automorphism chosen for the matrix construction. It is also independent of the order in which row and column indices are associated with semifield elements (a matter just of row/column permutations). In our experiments, we use `GivaroGFq`, the LinBox wrapper of the Zech's logarithm table implementation of the field $\boldsymbol{GF}(p^k)$ in Givaro. Our experiments are done in the case $p = 3$ with $2 \le k \le 6$.

3. P-RANK ALGORITHMS

Our application produces very large dense matrices which turn out to have rather low rank. The families of matrices under study appear to have ranks significantly lower than the matrix order, for instance, the matrices coming from Paley graphs and $\mathcal{P}^*$-graphs. The Paley graph adjacency matrices A of dimension $n = p^e$ (n is congruent to 1 modulo 4) have the property [2]:

$$\text{rank}_p(2A + I) = (\frac{p+1}{2})^e.$$

The $\mathcal{P}^*$-graph adjacency matrices of dimension p^e (p is a prime congruent to 3 modulo 4 and e is an even number) have the property [20]:

$$\text{rank}_p(2A + I) = 2(\frac{p+1}{4})^e(3^{\frac{e}{2}} - 1).$$

For several other classes, such as the Dickson graphs, rank formulas are not known. The $\mathcal{P}^*$-graphs introduced in [20] are of interest because they disprove an earlier conjecture that the Paley graphs would have the smallest p-ranks. We have verified the $\mathcal{P}^*$-graph ranks for orders up to 3^{10}. For measuring the effectiveness of several algorithm choices, it suffices to study one class of known rank formula (Paley) and one class of unknown rank formula (Dickson). This gives a nice combination of reassurance (Paley computed ranks agree with formula) and drama (we determine Dickson ranks *de novo*).

3.1 Space Efficient Rank Algorithms

Since we are interested in computing on large cases where $O(n^2)$ memory is not available, we restrict our attention to methods that use $O(nr)$ memory for an $n \times n$ matrix of rank r and we eventually develop a method that uses $O(r^2 + nc)$ memory for a small c. In fact c is $O(\log(1/\epsilon))$, where ϵ is the probability of error we are willing to tolerate. Thus $c = 20$ gives less than one in a million chance of a wrong rank returned.

Algorithm 1: Rank of matrix $A \in \boldsymbol{GF}(p)^{n\times n}$, computed deterministically.

Step 1: Determine b such that b rows of A may be stored in main memory.

Step 2: Row reduce the leading b rows of A obtaining s linearly independent rows.

```
Step 3: WHILE unprocessed rows remain and $s < b$
    adjoin $b - s$ unprocessed rows
    row reduce to obtain $s'$ independent rows
    discard any dependent rows
    set $s \leftarrow s'$
```

Step 4: if $s \ge b$ return "fail" else return rank $= s$.

This algorithm is deterministic and requires $O(n^2r)$ arithmetic operations in $\boldsymbol{GF}(p)$ and uses $O(nr)$ memory. It can be quite fast in practice with the block elimination done using, for example, LU-Divine [7]. Also it is very well suited to a large dense matrix stored on disk or that can be generated on demand as is the case for our SRG problems. However on current hardware, the limiting cost is the $O(nr)$ memory usage. For instance the Paley graph matrix at exponent e has $n = 3^e$ and $r = 2^e$. If floats are used for the elements (for the sake of BLAS), then the memory needed is 4×6^e bytes. This is about 8GB when $e = 12$ and completely out of reach when e is larger.

Next we turn to a probabilistic method to address the memory issue and achieve a lower memory usage, but it comes at the cost of working over a moderate sized extension field of $\boldsymbol{GF}(p)$ where highly tuned arithmetic and BLAS type kernels are not currently available, but see [6] concerning BLAS. This algorithm is from [3].

Algorithm 2: Rank of matrix $A \in \boldsymbol{GF}(p)^{n\times n}$, probabilistic blackbox method.

Step 1: Choose the degree d of the field extension to be used. Compute two random nonsingular $n\times n$ diagonal matrices D and E over $\boldsymbol{GF}(p^d)$.

Step 2: Precondition to blackbox $B \leftarrow DA^TEAD$.

Step 3: Compute the minimal polynomial of B by Wiedemann's algorithm [21] over the extension field. If its degree is n and its least monomial with nonzero coefficient is of degree k, (and $k \ne 0$), return rank $r = n-k$.

This approach to rank requires $O(n^2rd)$ arithmetic operations in $\boldsymbol{GF}(p)$, where d is the degree of the extension field. It is Monte Carlo, with probability of failure no more than $(2n^2 - n)/p^d$. It uses $O(nd)$ memory to store a few vectors over the extension field. It also requires each entry of A to be either recalled from disk storage or regenerated by formula $O(r)$ times. In the presence of ample disk space but insufficient main memory, a storage scheme could be used with significant but constant overhead. Assuming $O(1)$ cost to generate an entry from formula, it can be more efficient to regenerate entries on demand. We have not explored this trade-off in detail. In either case, the matrix access cost is $O(n^2r)$ overall.

Finally, we present a straightforward probabilistic algorithm to compute the rank of A when we have a small upperbound for its rank.

Algorithm 3: Rank of matrix $A \in \boldsymbol{GF}(p)^{n\times n}$, probabilistic dense method. Input is A and probabilistic error tolerance ϵ.

Step 1: Choose $c \leftarrow \log_p(2/\epsilon)$ and choose b so that $b \ge \text{rank}(A) + c$ (This requires some prior knowledge about A - recursive doubling of b can be used if initial choice of b is too small)

Step 2: Project A to a $b \times b$ matrix $B \leftarrow LAR$. Where R and L^T are randomly chosen from $A \in \boldsymbol{GF}(p)^{n\times b}$.

Step 3: Compute r, the rank of B. If $r \leq b - c$ return r, else "fail" (or double b and repeat).

This requires $\mathrm{O}(n^2\, b)$ arithmetic operations in $\boldsymbol{GF}(p)$ and $\mathrm{O}(n\, b)$ storage for L, R, and intermediate results. If one begins by computing AR, then note that no more than $\mathrm{O}(nb)$ of A need be in memory at any one time and that blocks of A only need be generated (or taken from disk) once. Since b is $\mathrm{O}(r)$, we are using $\mathrm{O}(n^2r)$ time and $\mathrm{O}(nr)$ memory.

We must demonstrate that this algorithm finds the rank with the specified $1 - \epsilon$ probability of success.

THEOREM 1. *Assuming b is an upper bound of the rank of A as described in Step 1 of Algorithm 3 then r is the rank of A with probability $P(rank(A) = rank(B)) \geq 1 - 2/p^c$.*

PROOF. Let ρ denote the true rank of A and consider the columns of AR. Let us denote the columns of R by $r_1, \ldots r_b$. If these are uniformly randomly chosen then $s_1 = A\, r_1, \ldots, s_b = A\, r_b$ will be a uniform random sample of the column-space of A. Now, suppose that $t < \rho$ is the dimension of the span of $\{s_i\}$, and, without loss of generality, that $s_1, \ldots, s_t$ is a basis. That means that s_j, for $j \in \{t+1, \ldots, b\}$ are linearly dependent on the other s_is. The probability of this happening for each s_j, a random sample of the column space of A, is less than $1/p^{\rho-t} < 1/p$. Thus when the r_i are chosen randomly and independently, the probability that there are c random vectors which happen to lie in the t dimensional subspace of the ρ dimensional column space of A is less than $1/p^c$. Hence, the probability that the $\{s_i\}$ span the full column-space of A is at worst $1 - 1/p^c$. The same argument may then be applied to the row span of AR with random samples of this row span given by the rows of $L(AR)$. The probability of error is again less than $1/p^c$ and the overall probability of success is at least $(1 - 1/p^c)^2 \geq 1 - 2/p^c$. $\square$

In summary, randomly projecting down to a block size slightly larger than the rank suffices for a high probability of success even when the field characteristic is small. It would be useful to project with cheaper preconditioners L and R than full random matrices as we use here. For examples, Toeplitz, sparse, or butterfly matrices might work. These have been used as preconditioners to ensure useful properties for fields of large characteristic [21, 14, 3, 9]. We know of no argument to justify one of them for this context. But note that if $L = [L_1 \mid L_2]^T$ and $R = [R_1 \mid R_2]$, where L_1AR_1 has the same rank as LAR and L_2, R_2 are $n \times c$ fully random blocks, then our proof of the theorem applies. In other words, the larger parts of L and R, namely L_1 and R_1, could be chosen heuristically with fast computation in mind and the result validated by a relatively few fully random rows and columns. This is similar to what we propose next, a computation using the Schur complement.

Speed is important to our problem, but the memory issue is of even greater importance. We will use a heuristic somewhat like that just described, but follow with a validation that uses less than nr memory and has a deterministic variant.

3.2 Rank Isolating Heuristics

We discuss several heuristics which succeed on varying numbers of all possible matrices. At least one of them works on each matrix we have encountered in practice. All of these may be viewed as computing the leading $b \times b$ block of $B = LAR$, for some nonsingular $n \times n$ preconditioning matrices L and R and hoping its rank equals that of A. Three of them have the specific block pattern

$$L = \begin{pmatrix} I_b & X \\ 0 & I_{n-b} \end{pmatrix}, R = \begin{pmatrix} I_b & 0 \\ Y & I_{n-b} \end{pmatrix}.$$

Coupled with a certificate or Monte Carlo verification that the rank of the upper-left block of LAR is actually the rank of all of LAR, these heuristics make effective methods in practice.

Heuristic scheme: Rank of matrix $A \in \boldsymbol{GF}(p)^{n\times n}$.

Step 1: Let $b = 1$.

Step 2: Choose $b \times n - b$ block X and $n - b \times b$ block Y according to a specified heuristic (see below).

Step 3: For

$$A = \begin{pmatrix} B & C \\ D & E \end{pmatrix},$$

where B is $b \times b$, compute $M \leftarrow B + CY + XD + XEY$

Step 4: Compute $r = \mathrm{rank}(M)$.

Step 5: If $c = b - r$ is too small (see Theorem 1), then double b and go to step 2. Otherwise return the rank r. Optionally, first certify the rank.

The repeated doubling of the block size b assures that the largest block size is no more than $2r$ and this doubling is the source of a $\log(r)$ factor in the time estimates. In practice we have often been able to start with a more reasonable initial guess than $b = 1$ and effectively eliminate the $\log(r)$ factor.

Assume the cost of computing CY and XD are each a function $\mathrm{c}(n, b)$ of the shape. This $\mathrm{c}(n, b)$ is at most $\mathrm{O}(nb^2)$, for dense X and Y, and at least 0 for zero blocks X and Y. That gives away two of our heuristics. The other two lie somewhere between these extremes in cost. Let us assume E is decomposed in blocks of b columns so that $E = (E_2, E_3, \ldots, E_{n/b})$. Then $XE = (XE_2, XE_3, \ldots, XE_{n/b})$, and the overall cost of step 3 is $(n/b + 2)\mathrm{c}(n, b)$. The cost of the rank computation is $\mathrm{O}(b^3)$ with classical methods. Since we keep the block size within a constant multiple of the rank, overall the heuristic scheme costs $\mathrm{O}(\mathrm{c}(n, r)\, n \log(r)/r + r^3)$.

What heuristic might we try in step 2? Here are some ideas. They are given in increasing order of cost (and range of effectiveness), and after that the two certificate strategies are discussed.

Heuristic 1 is simply to hope the leading block has the same rank as A, i.e.,set $X = 0$ and $Y = 0$. Since $\mathrm{c}(n, r)$ is zero, the time is $\mathrm{O}(r^3)$. The memory is $\mathrm{O}(r^2)$.

This has proven to be useful, for example, with the Paley graph matrices where the leading $r \times r$ submatrix turned out to be nonsingular in every case.

Heuristic 2 is to set X and Y to a string of scalar matrix $b \times b$ blocks, $X = (l_2 I_b, l_3 I_b, \ldots, l_{n/b} I_b)$ and $Y =$

$(r_2 I_b, r_3 I_b, \ldots, r_{n/b} I_b)^T$, where the scalars l_i and r_i are chosen at random from $\boldsymbol{GF}(p)$. The random distribution used can be varied, but we used a uniform distribution. In this case the cost of step 3 in the scheme is $\mathrm{O}(n^2)$ because each entry of E contributes once, multiplied by a scalar from X and a scalar from Y, as a summand of an entry of XEY. Thus the run time is $\mathrm{O}(n^2 \log(r) + r^3)$. Since the space to store X and Y is just that of the $\mathrm{O}(n/r)$ scalars involved, the memory usage is $\mathrm{O}(r^2 + n/r)$.

This worked for the Dickson graph matrices over $\boldsymbol{GF}(3)$ about half the time, by which we mean that we probabilistically verified the computed rank about half the time in trials.

Heuristic 3 is to use butterfly matrices for L and R [3, 18]. A b-block product of a dense matrix and a butterfly costs $\mathrm{O}(b^2 \log(b))$, so a modified step 3 costs $\mathrm{O}(n^2 \log(b))$. With the doubling and the largest block size $b = \mathrm{O}(r)$, this leads to an $\mathrm{O}(n^2 \log(n) \log(r) + r^3)$ time cost and $\mathrm{O}(n \log(n) + r^2)$ memory cost for heuristic 3.

We did not pursue this in experiments, since the heuristics 1 and 2 are simpler, did work in practice, and are predicted to be faster. We conjecture that butterflies may work with provable error bound when r is a suitably small factor of the final block size used. Observe that they work when the block size is n [19]. Do they work when the block size is $r \log(n)$?

Heuristic 4: Finally, the scheme could be made into a Monte Carlo algorithm by using random dense blocks for X and Y.

If a suitable over-estimate for the rank is used as the final block size, this produces the correct rank with bounded error probability. Specifically if the final block size is $b = r + c$ where r is the leading block's rank, then the chance that r is not A's rank is less than $1/p^c$ by a similar argument to that used for theorem 1.

This method uses $\mathrm{O}(n^2 r \log(r))$ arithmetic operations in $\boldsymbol{GF}(p)$. It uses $\mathrm{O}(rn)$ memory if X or Y is stored, as at first may seem necessary. Heuristic 4 uses $\mathrm{O}(r^2)$ memory by computing the X and Y a $b \times b$ block at a time. The product step is then for $i \in \{1, \ldots, n/b\}$, for $j \in \{1, \ldots, n/b\}$, $B = B + X_i A_{i,j} Y_j$ where the indexed items are $b \times b$ blocks. Each block of X can be generated just when needed. However the blocks of Y must be available to be reused each time through the outer loop. For the claimed space cost these blocks of Y may be regenerated from a saved seed each time they are needed. This is where the dependence on a pseudo-random system enters in. Alternatively, X and Y may be saved on disk between uses, but then there is not a memory advantage over algorithm 1.

The heuristic methods are straightforward and seem to work well enough in practice. However, they do not provide truly quantifiable probability of error in principle. We will offer two methods to certify the rank in order to address this problem and allow us to use one of the earlier, cheaper heuristics.

It should be noted that even when the heuristic scheme is combined with a good certificate for rank, we do not obtain an algorithm for computing rank. This is because our method can return "fail" in place of a probabilistically correct rank and we cannot guarantee that there is a non-zero probability of obtaining a result other than "fail".

3.3 Rank Certificates

The first certificate is deterministic and the second is Monte Carlo but faster. Both certificates require than the initial $r \times r$ block of the matrix be non-singular. This means to verify the output of the heuristic scheme, the certificate will actually be run on the matrix LAR. This is important since the entries of LAR will, in general, be much more expensive to compute than those of A.

Rank Certificate 1: Given a number r, $0 \leq r \leq n$, and $A \in \boldsymbol{GF}(p)^{n \times n}$. If A has a non-singular initial $r \times r$ block, verify $r = \mathrm{rank}(A)$ otherwise return "fail".

Step 1: Let $A_{i,j}$ denote the $r \times r$ block starting at row $ri+1$ and column $rj + 1$ and let $k = n/r$ be the number of blocks (n can be padded to the next multiple of r).

Step 2: Compute $\mathrm{rank}(A_{1,1})$. If it is not r, return "fail". This may be done by LU decomposition, which will then also be useful for the next step.

Step 3: Verify that the Schur complement [1] of block $A_{1,1}$ is zero. The Schur complement may be computed in block form as $(A_{i,j} - A_{i,1} A_{1,1}^{-1} A_{1,j}), i \in 2..k, j \in 2..k$. The $A_{1,1}^{-1}$ may be computed explicitly or the LU of step 2 used in back-solving mode.

Step 4: If any block computed in step 3 is nonzero, return "fail" because $\mathrm{rank}(A)$ is greater than r. Otherwise return that $r = \mathrm{rank}(A)$ is verified.

As in the previous algorithm, there is a problem here for memory usage in that some of the blocks in the first row and column will have to be stored between uses when forming the blocks of the Schur complement. The most space efficient scheme we know is "block the blocks" in $(k-1)^{1/2}$ size meta-blocks. In this case, each of $A_{1,i}$, for $i \in \{2, \ldots, k\}$ will be constructed $(k-1)^{1/2}$ times. With this plan the memory usage is $\mathrm{O}(r(n/r)^{1/2}) = \mathrm{O}((rn)^{1/2})$ (this is in the spirit of baby step, giant step, see [13]).

The algorithm calls for an $r \times r$-matrix inversion and $(k-1)^2$ $r \times r$-matrix multiplications requiring $\mathrm{O}(k^2 r^3) = \mathrm{O}(n^2 r)$ arithmetic operations. Add to this the cost of constructing the $A_{i,j}$ blocks, which may be $\mathrm{O}(r^2)$ or greater depending on which preconditioner is being used with A. With the preconditioners we have used, the cost of generating the first row and column is much higher than the cost for generating the other rows and columns. For instance, the cost to generate the (i, j) block may be $\mathrm{c}(i)\,\mathrm{c}(j)$, where $\mathrm{c}(1) = n$, and $\mathrm{c}(l) = r$ for $l > 1$. In this case, the total time for the algorithm Rank Certificate 1 is $\mathrm{O}((k-1)^2 r^2 + (k-1)nr + n^2 + (k-1)^{3/2} nr)$, which is $\mathrm{O}(n^2 (n/r)^{1/2})$.

Finally, we describe a faster, probabilistic, rank certificate based on the Schur complement:

Rank Certificate 2 Given $A \in \boldsymbol{GF}(p)^{n \times n}$, rank candidate r (with A expected to have leading principal r-minor nonzero) and error probability bound $\epsilon > 0$, return $r = \mathrm{rank}(A)$ verified with probability of error less than ϵ, or "rank is greater" or "fail" (from step 2),

Steps 1,2: . . . are as in Certificate 1. Also compute $c = \log_p(1/\epsilon)$ and let X be a $n \times c$ random matrix with X_i denoting the i-th $r \times c$ block. Let W be an $r \times c$ matrix.

```
Step 3: {Compute the Schur complement multiplied by X.}
    For
        $A = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$, $B$ is $r \times r$.
    Compute and compare $EX$ and $DB^{-1}CX$ in a just-in-time fashion:
    $W_i \leftarrow B^{-1}A_{1,i}X_i$, $i = 2 \dots k$. { $W = B^{-1}CX$ }
    FOR ($i$ from 2 to $k$)
        $T \leftarrow A_{i,1}W_i$, { $i$-th block of $DB^{-1}CX$ }
        $U \leftarrow \sum_{j=2}^{k} A_{i,j}X_j$, { $i$-th block of $EX$ }
        IF $U \neq T$, return "rank is greater".
Step 4: Rank is verified. Return "rank is r".
```

THEOREM 2. *If certificate 2 verifies the rank r, the probability of error is less than ϵ. It runs in $O(r^3 + n^2c)$ arithmetic operations in $\boldsymbol{GF}(p)$ and uses $O(r^2 + nc)$ memory. Each block of A is accessed once.*

PROOF. Correctness of the just-in-time check for the zero Schur complement is by an argument similar to that we gave in theorem 1 (and also used for example in [12, Theorem 2.2]. If the Schur complement is nonzero, the chance that a matrix random-vector product is zero is at most $1/p$.

The inversion of the leading block in step 2 requires $O(r^3)$ in classical elimination. The computation of W in Step 3 uses $(k-1)r^2c$ operations, which is $O(nrc)$. Each iteration of the second for-loop costs r^2c for T and $(k-1)r^2c$ for U and rc for the equality check. Thus the time for the loop as a whole is in $O(n^2c)$. The space and A access assertions are evident. □

3.4 Failed Certificate Correction

It may be that the above rank certifications fail. The first way they can fail is if the B, the upper left block, is singular. In this case, a new preconditioner or a smaller upper block will need to be computed by rerunning one of the heuristics in Section 3.2. The other cause for failure is that the Schur compliment $S = E - DB^{-1}C$ is not 0. In this case, we may still recover the rank of A with high confidence. So long as B was full rank, we have

$$A = \begin{pmatrix} B & C \\ D & E \end{pmatrix} = \begin{pmatrix} I & 0 \\ DB^{-1} & I \end{pmatrix} \begin{pmatrix} B & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} I & B^{-1}C \\ 0 & I \end{pmatrix},$$

and we see that $\text{rank}(B) + \text{rank}(S) = \text{rank}(A)$. If most of the rank was contained in B, that means we can effectively use Algorithm 3 to compute the rank of the Schur complement computed in the rank certification step. Specifically, we compute the rank of $(E - DB^{-1}C)X = SX$. If the rank of this $(n-r) \times c$ matrix is significantly less than c, then with high probability it is the rank of S and we report $\text{rank}(A) = \text{rank}(B) + \text{rank}(S)$. This increases the space cost if we store all of the T and U values computed in step 3 of Certificate 2. We can avoid this by instead computing a random $c \times (n-r)$ matrix Y and accumulating $V = YSX$ as $V \leftarrow V + Y(U - T)$ at each iteration of the loop in step 3. It should be noted that this method comes with only an *a posteriori* computed probability of correctness. However, it may provide enough confidence in its answer to avoid a possibly very lengthy recomputation of the entire method.

dimension	Rank	CPU Time	RAM Usage
$3^4 = 81$	$2^4 = 16$	0.0097s	30kB
$3^6 = 729$	$2^6 = 64$	0.03s	120kB
$3^8 = 6561$	$2^8 = 256$	0.93s	1.2MB
$3^{10} = 59049$	$2^{10} = 1024$	63.0s	18MB
$3^{12} = 531441$	$2^{12} = 4048$	4632s	195MB

Table 1: The Paley SRG example computed with Heuristic 1 and Certificate 2.

dimension	Rank	CPU Time	RAM Usage
$3^4 = 81$	20	0.021s	35kB
$3^6 = 729$	85	0.35s	180kB
$3^8 = 6561$	376	33.3s	2.0MB
$3^{10} = 59049$	1654	1799s	36MB
$3^{12} = 531441$	7283	168076s	516MB

Table 2: The Dickson SRG example computed with Heuristic 2 and Certificate 2.

4. EXPERIMENTS

The benchmarks in this section were written using the LinBox library and run on a machine with two 3.20 GHz Intel Dual Xeon processors and 6 GB of shared RAM. The runs used just one processor.

Table 1 and Table 2 collect the results of computing 3-rank using the heuristic scheme and certifying the rank with certificate 2. We chose not to implement certificate 1 since it is clearly much more expensive than certificate 2. In particular, it has to re-compute blocks of LAR $\sqrt{k}$ times and those computations are already the most time intensive part of the computation when using heuristic 2 with certificate 2.

Extrapolating from the table 1, heuristic 1 and Rank certificate 2, should compute the rank of the Paley matrix of dimension 3^{14} in about 4 days using about 2 gigabytes of RAM. Unfortunately, 3^{14} is past the limit of the Givaro library's fast implementation of Galois fields. Thus we expect an implementation using a slower field arithmetic implementation to require a factor of 3 or 4 more time.

Extrapolating from the table 2, heuristic 2 and rank certificate 2, should compute the rank of the Dickson matrix of dimension 3^{14} using about 1 CPU year and about 7.3 gigabytes of RAM. The drastic difference in timings compared to the Paley case, comes from applying the more expensive preconditioner in Heuristic 2.

In the both of the above sets of experiments, the dominating cost is that of the rank certificate. If the heuristic computes the rank correctly, the certificate generally accounted for about 75% of the total computation time. At least 75% of that cost was for the just-in-time computation of the (very large) block E.

We also computed the rank of the Paley matrix using algorithm 2 (blackbox) and algorithm 3 (dense elimination method after JIT computation of a projection). The time and memory usage for these computations are given in Tables 3 and 4.

As can be seen from table 3, the blackbox method uses much less RAM than the heuristic methods but at the cost

dimension	Rank	CPU Time	RAM Usage
$3^4 = 81$	$2^4 = 16$	0.020s	20kB
$3^6 = 729$	$2^6 = 64$	4.45s	128kB
$3^8 = 6561$	$2^8 = 256$	4131s	1.1MB
$3^{10} = 59049$	$2^{10} = 1024$	est. 30 days	est. 15MB

Table 3: The Paley SRG example computed with Algorithm 2: the Weidemann Blackbox method

dimension	Rank	CPU Time	RAM Usage
$3^4 = 81$	$2^4 = 16$	0.010s	20kB
$3^6 = 729$	$2^6 = 64$	0.780s	875kB
$3^8 = 6561$	$2^8 = 256$	208s	13.4MB
$3^{10} = 59049$	$2^{10} = 1024$	1496s	467MB
$3^{12} = 531441$	$2^{12} = 4048$	est. 5 days	est. 13GB

Table 4: The Paley SRG example computed with Algorithm 3: just-in-time projection plus dense elimination

of much more time. In table 4, it can be seen that the LAR projection algorithm, while faster than algorithm 2, is still far slower than the heuristic plus certification method. That is, in practice, the heuristic plus certification method allows us to compute the rank of a Paley matrix one size larger (i.e. dimension 9 times larger and rank 4 times higher) than could be computed with a dense elimination algorithm using the same computing resources.

5. SOFTWARE DESIGN

The heuristics and algorithms discussed in this paper were implemented using the LinBox library [5]. One new concept we added to the library was that of a "just in time matrix". That is, a matrix to which we have access to the entries, but those entries are not stored in RAM – they are computed (or retrieved) just in time for computation. Contrast this to the idea of a black-box matrix: while a black-box may not be stored in RAM it also does not allow access to its entries directly, it allows only matrix-vector products to be computed. A simple example of a matrix that admits a just in time implementation is a Hilbert matrix: H where $H_{i,j} = 1/(i+j)$.

In the library, we provide a generic `JITMatrix` class. This class is a derived class of `BlackBox` and thus provides all the features of a black box matrix and allows `JITMatrix` objects to be used in most LinBox algorithms. The `JITMatrix` class is also templated by a generator and a field. The generator is a function object which must take a pair of indices (i, j) and return an element of the field corresponding to the $(i, j)^{th}$ entry of the matrix. An instance of a `JITMatrix` will require only enough storage for the generator and the field. For example, in the case of $\boldsymbol{GF}(p^k)$ the field may store arithmetic tables, and in the case of a Hilbert matrix the generator may precompute and store all the reciprocals up to $1/(2\,n)$.

An entry of a Paley-type matrix is computed by determining if the difference of a pair of elements in a semifield is a square in the semifield. We implement our generator by precomputing the list of squares in the semifield (storing this takes space $O(n)$). The generator function then just computes a subtraction in the semifield, and does a look up. A generic Paley-type matrix class is possible where the user provides an object to do the semifield arithmetic.

However, for the Paley-type matrices we are using block methods so we chose a block just in time design. That is, we have created a `JITMatrix` whose entries are $b \times b$ dense matrices. In order to do this, we had to create a `BlockRing` which is an object for expressing the arithmetic functions on blocks consistent with the LinBox "field" concept. It wraps some of the functionality of a field for dense matrices. Of course the inversion and division functions are partial in a block ring (ring of square matrices). In fact we have not yet implemented these partial functions at all, since they are not needed in the algorithms.

In some cases, we found it necessary to precondition a Paley-type matrix. For simple preconditioners, it is possible to embed the preconditioner in the generator function object. In doing so, the preconditioned Paley-type matrix is still implemented as a `JITMatrix`. This allows us to compute with $n \times n$ Paley-type matrices where n^2 entries are too large to fit in memory but $r^2 + n$ entries are not.

6. CONCLUSION

We have addressed the problem of p-rank computation for very large matrices when p is small and the rank to be computed is relatively small. We have offered a heuristic (heuristic 2 coupled with certificate 2) that worked well on some very large matrices (of order about 5×10^5, hence containing 2.5×10^{11} entries). This method has as a very low a space complexity. It involves processing the matrix only a few times, whether from disk storage or from formula generation and thus is suitable for matrices that will not fit in main memory. It has faster asymptotic complexity than any method we know and it has fast practical performance due to the use of the fast level 3 BLAS matrix operation kernels.

We have applied this technique to assist in the study of strongly regular graphs, particularly to assist the the project to identify p-rank formulas for certain families of strongly regular graphs. To date we have ranks of matrices of orders up to 3^{12} in the Paley, P*, and Dickson sequences, as well as a few other isolated constructions. In the case of Paley and P* constructions this verifies proven formulas. No formula is known for ranks of the Dickson sequence. We would like to compute ranks for the 3^{14} and 3^{16} orders in pursuit of such a formula.

In the process we have started a design process for the software library LinBox which should prove useful for the implementation of block blackbox methods, out of core methods, and parallel methods. In immediate future work we will implement some of these features, in particular for the sake of computing the rank of the order 3^{14} Dickson rank.

7. REFERENCES

[1] M. Brookes. The matrix reference manual. `http://www.ee.ic.ac.uk/hp/staff/www/matrix/property.html#schurcomp`, 2005. [Online; accessed 22-January-2007].

[2] A. E. Brouwer and C. A. Van Eijl. On the p-rank of the adjacency matrices of strongly regular graphs. *J. Algebraic Comb.*, 1(4):329–346, 1992.

[3] L. Chen, W. Eberly, E. Kaltofen, W. Turner, B. D. Saunders, and G. Villard. Efficient matrix preconditioners for black box linear algebra. *LAA 343-344, 2002*, pages 119–146, 2002.

[4] C. Ding and J. Yuan. A family of skew Hadamard difference set. *J. Comb. Theory, Ser. A*, 113:1526–1535, 2006.

[5] J-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. Turner, and G. Villard. Linbox: A generic library for exact linear algebra. In *ICMS'02*, pages 40–50, 2002.

[6] J-G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In *Proc. of ISSAC'02*, pages 63 – 74. ACM Press, 2002.

[7] J.-G. Dumas, P. Giorgi, and C. Pernet. FFPACK: Finite field linear algebra package. In *Proc. of ISSAC'05*, pages 119–126, 2004.

[8] J-G. Dumas, B. D. Saunders, and G. Villard. Smith form via the valence: Experience with matrices from homology. In *Proc. of ISSAC'00*, pages 95 – 105. ACM Press, 2000.

[9] J.-G. Dumas and G. Villard. Computing the rank of large sparse matrices over finite fields. In *Proc. CASC'2002, The Fifth International Workshop on Computer Algebra in Scientific Computing*, pages 22–27. Springer-Verlag, 2002.

[10] J.-C. Faugère. Parallelization of Gröbner basis. In H. Hong, editor, *PASCO'94*, volume 5 of *Lecture notes series in computing*, pages 109–133, 1994.

[11] Thierry Gautier, Jean-Louis Roch, and Gilles Villard. Givaro, a C++ for algebraic computations. `http://www-lmc.imag.fr/Logiciels/givaro`.

[12] M Giesbrecht, A. Lobo, and B. D. Saunders. Certifying inconsistency of sparse linear systems. In *Proc. of ISSAC'98*, pages 113–119. ACM Press, 1998.

[13] E. Kaltofen. An output-sensitive variant of the baby steps / giant steps determinant algorithm. In *Proc. of ISSAC'05*, pages 138–144, 2002.

[14] E. Kaltofen and B. D. Saunders. On Wiedemann's method of solving sparse linear systems. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 539 of *LNCS*, pages 29–38, 1991.

[15] B. A. LaMacchia and A.M. Odlyzko. Solving large sparse linear systems over finite fields. *Lecture Notes in Computer Science*, 537:109–133, 1991.

[16] S. L. Ma. A survey of partial difference sets. *Designs, Codes and Cryptography*, 4:221–261, 1994.

[17] The LinBox Team. Linbox, a C++ library for exact linear algebra. `http://www.linalg.org/`.

[18] W. Turner. Preconditioners for singular black box matrices. In *Proc. of ISSAC'05*, pages 332–339, New York, NY, USA, 2005. ACM Press.

[19] W. Turner. A block wiedemann rank algorithm. In *Proc. of ISSAC'06*, pages 332–339, New York, NY, USA, 2006. ACM Press.

[20] G. Weng, W. Qiu, Z. Wang, and Q. Xiang. Pseudo-paley graphs and skew Hadamard difference sets from commutative semifields, 2006. Preprint.

[21] D. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Trans. Inform. Theory*, 32:54 – 62, 1986.

[22] Q. Xiang. Recent progress in algebraic design theory. *Finite Fields and Their Applications*, 11:622–653, 2005.