# Constructing Permutation Representations for Large Matrix Groups

Gene Cooperman[1] Larry Finkelstein[1] Bryant York[2] Michael Tselman[1]

College of Computer Science
Northeastern University
Boston, MA 02115

**Abstract**

New techniques, both theoretical and practical, are presented for constructing a permutation representation for a matrix group. We assume that the resulting permutation degree, $n$, can be 10,000,000 and larger. The key idea is to build the new permutation representation using the conjugation action on a conjugacy class of subgroups of prime order. A unique signature for each group element corresponding to the conjugacy class is used in order to avoid matrix multiplication. The requirement of at least $n$ matrix multiplications would otherwise have made the computation hopelessly impractical. Additional software optimizations are described, which reduce the CPU time by at least an additional factor of 10. Further, a special data structure is designed that serves both as a search tree and as a hash array, while requiring space of only $1.6n \log_2 n$ bits.

The technique has been implemented and tested on the sporadic simple group $Ly$, discovered by Lyons [9], in both a sequential (SPARCserver 670MP) and parallel SIMD (MasPar MP-1) version. Starting with a generating set for $Ly$ as a subgroup of $GL(111, 5)$ [5], a set of generating permutations for $Ly$ acting on $9,606,125$ points is constructed as well as a base for this permutation representation. The sequential version required four days of CPU time to construct a data structure which can be used to compute the permutation image of an arbitrary matrix. The parallel version did so in 12 hours. Work is in progress on a faster parallel implementation.

## 1 Introduction

New techniques are demonstrated for constructing a permutation representation for finite matrix groups specified by a generating set of matrices. The techniques fit well into recent efforts by a number of researchers [4, 6, 8] on the problem of determining the structure of matrix groups defined over finite fields. In addition, Gerhard Michler [10] and his group at Essen make extensive use of permutation representations for matrix groups as a means for switching from one characteristic to another in the course of developing modular representations for the sporadic simple groups.

As a practical test for the ideas in this paper, we have constructed specific permutation generators for the sporadic simple group $Ly$ discovered by Lyons [9] from a matrix representation of dimension 111 over $GF(5)$ described in [5]. This example has independent interest. In 1972, Sims [12] announced the existence and uniqueness of $Ly$ and simultaneously gave a presentation for $Ly$, which was verified by performing a coset enumeration on a subgroup isomorphic to $G_2(5)$ with index $8,835,156$. Certainly, one could adapt this enumeration to construct specific permutation generators of this degree, but it has not yet been done. A more general approach, and one which we have followed, is to work directly with the matrix representation of $Ly$ and to construct a permutation representation of degree $9,606,125$ of $Ly$ acting on a conjugacy class $\mathcal{O}$ of $Z_3$ subgroups.

In general, our permutation representations will be constructed through the action of the matrix group on a "small" conjugacy class of subgroups of prime order. This approach can be justified, in part on theoretical grounds, by a result of Babai and Beals [4], which asserts that for a simple group $G$ there is always a conjugacy class of subgroups of prime order whose size is "comparable" to the degree of the smallest permutation representation for $G$.

We use randomization to gain a significant speedup in running time. This speedup is required for working with very large examples. Randomization is used in two ways. The first is to construct random elements of $G$ in order to identify suitable elements of prime order. A more formal method for producing *nearly uniformly distributed* random elements has been described by Babai [3], but its practicality has yet to be tested in an implementation.

The second use of randomization occurs in the enumeration of all elements of a conjugacy class $\mathcal{O}$ of subgroups of $G$ of prime order $p$. Suppose there exists a vector $v$ with the property that no non-identity element of $G$ fixes $v$. Then each $g \in G$ can be uniquely identified by a signature defined to be the image of $v$ under $g$. This allows one to rapidly test equality of two subgroups $\langle x \rangle$ and $\langle y \rangle$ of $\mathcal{O}$. In particular, $\langle x \rangle = \langle y \rangle$ if and only if the signature of $x$ is equal to the signature of $y^i$, for some $i$, $1 \leq i \leq p-1$. In Lemma 3.1 it is shown that under suitable conditions, if $v$ is chosen at random, then $v$ can be used to construct a signature with extremely high probability, which in our example of Lyon's group is at least $1 - 3.8 \times 10^{-23}$.

The key to efficiency is to quickly compute a "signature" for each point on which the group acts. For example, rather

[1]Partially supported by NSF Grant CCR-9204469.
[2]Partially supported by ARPA Grant MDA-972-93-1-0023.

than use the action of Lyons' group on $\mathcal{O}$ of degree 9,606,125, we could also have chosen the action on the right cosets of a subgroup isomorphic to $G_2(5)$ which leads to a permutation representation of minimal degree 8,835,156. Several possible signatures for the permutation action exist in this case. Nevertheless, none appeared to be as efficient to compute as the signature developed for the action on $\mathcal{O}$ and the computation would have taken substantially longer despite the relatively small decrease in the size of the point set.

Since this computation is near the limits of practical sequential computation, one must consider how to scale up further. The sequential computation required 64 Megabytes of memory and four CPU days on a SPARCserver 670MP. Machines with ten times as much memory are available, but ten times the CPU power is a more stringent requirement. Accordingly, we also discuss a parallel implementation on a MasPar MP-1 that performs eight times faster.

## 2 Conjugacy Class Approach

In this section, we assume that a "small" conjugacy class of subgroups of $G$ of prime order is available. It is easy to see that the smallest conjugacy class of cyclic subgroups must consist of cyclic subgroups of prime order. There are good heuristic techniques for finding representatives from each such conjugacy class, even when a character table is not available. See Linton's excellent discussion [7] of heuristic techniques for finding group elements with desired properties.

Often the smallest conjugacy class will consist of subgroups whose order is a small prime. As one constructs the elements of a conjugacy class, one can often estimate the size of the conjugacy class before the construction is completed. Assume, heuristically, that the candidates for new elements of a conjugacy class (i.e. conjugate of a previous subgroup by a random generator) are random with a uniform distribution. This is similar to the uniform hashing distribution that is often assumed in estimating the efficiency for hash functions. As one adds new elements to the conjugacy class $\mathcal{O}$, one expects the first few collisions with previously discovered elements to occur after $\sqrt{|\mathcal{O}|}$ elements have been added. This yields a rough estimate for $|\mathcal{O}|$, and one can guess which conjugacy class is smallest without having to find more than the square root of the number of elements for any conjugacy class.

In section 3.2.3, it is shown how the existence of a subgroup of $G$ which acts semi-regularly on $\mathcal{O}$ can be used to significantly speed up the enumeration of $\mathcal{O}$. In the case of Lyons' group, we know from the character table that a cyclic subgroup of order 67 must act semi-regularly on $\mathcal{O}$. Otherwise, such a subgroup must normalize, and hence centralize a subgroup of order 3 in $\mathcal{O}$, which is not possible. In the more general case, one initially has little initial knowledge of the structure of the group. Nevertheless, one can test an element from each conjugacy class of large cyclic subgroups, in order to find an element acting semi-regularly on $\mathcal{O}$. Although a general test may be expensive, it suffices to find a single example and to verify semi-regularity using the following result. Here, the *fixed point subspace* of a matrix is the eigenspace with eigenvalue 1.

**Lemma 2.1** *Let $z \in G \leq GL(n,q)$ be an element of prime order $r$ acting on a conjugacy class $\mathcal{O}$ of $G$ of subgroups of prime order $p$, with $p \neq r$. Let $d_1$ be the dimension of the fixed point subspace of $z$ and let $d_2$ be the dimension of the fixed point subspace of $x$, where $\langle x \rangle \in \mathcal{O}$. If $r$ does not divide $|GL(d_2,q)|$ and $d_1 < d_2$, then $\langle z \rangle$ acts semi-regularly on $\mathcal{O}$.*

*Proof:* It suffices to show that $z$ does not fix any points of $\mathcal{O}$. If $z$ in fact has a fixed point on $\mathcal{O}$, then $z$ normalizes $\langle y \rangle \in \mathcal{O}$ and hence must leave invariant the $d_2$-dimensional fixed point subspace of $y$. Since $r$ is relatively prime to $|GL(d_2,q)|$, $z$ must fix this subspace pointwise. But then $d_1 \geq d_2$, contradicting our hypothesis. □

**Remark** Let $g \in GL(n,q)$ have order $r$, relatively prime to $q$. Let $m$ be the dimension of the fixed point subspace of $g$ acting on the underlying vector space $V$. The dimension $m$ can be quickly computed by a randomized algorithm based on the following observation. If $v$ is chosen at random from $V$ according to the uniform distribution, then $\sum_{i=1}^{r} v^g$ satisfies the uniform distribution in the fixed point subspace of $g$. (The last statement is not true in general unless $r$ is relatively prime to $q$.) To construct a basis for the fixed point space of $g$, initialize $B \leftarrow \emptyset$. Execute a loop in which the basic step is to choose a random $v \in V$ and test if $u = \sum_{i=1}^{r} v^g \notin \langle B \rangle$. If the test fails, add $u$ to $B$. After $t$ consecutive successes, the probability is at least $1 - 1/q^t$ that $B$ is a basis for the fixed point subspace.

## 3 Algorithm

The algorithm is based on a data structure that serves both as a hash array and as a search tree for the conjugacy class of subgroups. We review the definition of hashing with open addressing to fix notation. Our definition covers only our own particular implementation of hashing, and makes no attempt at full generality. Let $\mathcal{O}$ be a set of objects. Let $A$ be a *hash array* of length larger than $|\mathcal{O}|$. (We use an array with $|A| = 1.6|\mathcal{O}|$.) The *hash function* $h_{\alpha,\beta}$ is defined in terms of $\alpha$ and $\beta$. The function $\alpha: \mathcal{O} \rightarrow [1,|A|]$ is a *primary hashing function on* $\mathcal{O}$ if for a random element $x \in \mathcal{O}$, $\alpha(x)$ is "nearly uniformly random". A *secondary hashing function*, $\beta: [1,|A|] \rightarrow [1,|A|]$ is an invertible function (i.e. a permutation on $[1,|A|]$) with long cycles under iteration by $\beta$.

We now restrict $\mathcal{O}$ to be the desired conjugacy class of subgroups of $G$. In implementations, it is most convenient to hash on an element of the subgroup, instead of on the subgroup as a whole. Each subgroup of $\mathcal{O}$ will be represented by exactly one generating element. Since the conjugacy class $\mathcal{O}$ consists of subgroups of order $p$, at most $p-1$ possible generators are possible for each subgroup, and this must be considered in building the hash table.

Let $x_{init}$ be a fixed element of $G$ with $\langle x_{init} \rangle \in \mathcal{O}$ and let `root_index` $= \alpha(x_{init})$. Let $S$ generate $G$. The entries in the hash array $A$ are either `NULL` or of the form $(i,g) \in [1,|A|] \times S$. For $x \in \mathcal{O}$, the *hash function* $h_{\alpha,\beta}: \mathcal{O} \rightarrow [1,|A|]$ probes the sequence $(f(x), \beta(f(x)), \beta^2(f(x)), \ldots)$ until it reaches the first index $i \in [1,|A|]$ for which either $A[i] =$ `NULL` or $A[i] = (i',g')$ *matches* $x$. We say that $A[i] = (i',g')$ *matches* $x$, with $\langle x \rangle \in \mathcal{O}$, if either $i' =$ `root_index` and $x = x_{init}$ or if $A[i']$ matches $y$ and $y^{g'} = x$. Finally, $h_{\alpha,\beta}$ returns $i$ and sets a condition variable to "`NULL`" or "`MATCH`".

Thus, the hash array $A$ effectively encodes a search tree having a branching factor of at most $|S|$. If $A[i] = (i',g')$, then one can view the search tree as containing an edge from node $i$ to node $i'$ labeled by $g'$. The node $i$ is labeled by $x$ for which $A[i]$ matches $x$, and the node $i'$ is similarly labeled. If $A[i] = (i,g)$, it is clear from the recursive definition of "match" that one can find a word $w$ in $S$ such

that $h_{\alpha,\beta}(x_{init}^w) = i$. This provides a partial inverse for $h_{\alpha,\beta}$. With these tools, standard techniques of breadth-first search are then employed until no new elements of $\mathcal{O}$ are found. Two bit vectors, each with $|A|$ elements, are used to encode the elements of the last and next frontier set.

Thus, the space required by the algorithm is dominated by the space required for $A$. In the case of Lyons' group, we chose $|A| = 16,000,000$ and each entry $(i, g)$ fits in one 32-bit word, thus requiring about 64 megabytes for the full algorithm.

The time is dominated by the time to compute $h_{\alpha,\beta}(x_{init}^w)$ for $w$ a word in $S$. Usually, $\alpha(x_{init}^w)$ returns an index satisfying "`NULL`" or "`MATCH`", and $\beta(\alpha(x_{init}^w))$ does not need to be called. So, the time is dominated by the time to first compute $x_{init}^w$ and then to test if $\alpha(x_{init}^w)$ matches $(x_{init}^w)^j$ for some power $j$. Since the conjugacy class $\mathcal{O}$ consists of subgroups of order $p$, at most $p-1$ possible matches need to be computed in order to test if a conjugate of $x_{init}$ generates a subgroup in $\mathcal{O}$ which has already been identified. The time to test a match involves multiplying out a word in $S$ whose length is bounded by $2d+1$ where $d$ is the depth of the search tree. This is because the word in $S$ acts by conjugation on $x_{init} \in \mathcal{O}$ and each conjugation requires two multiplications. So, $p-1$ words of length $2d+1$ typically need to be multiplied out in computing $h_{\alpha,\beta}(x_{init}^w)$. In our experiment with Lyons' group, we employ several heuristics described below, so that most nodes are found at depth $d = 2$, and all nodes are found by depth $d = 3$.

### 3.1 Signatures of Group Elements

As described above, the algorithm requires approximately $|A|2d(p-1)$ matrix multiplications plus additional time for hash collisions, or at least 128,000,000 matrix multiplications in $GL(111, 5)$ (assuming $|A| = 16,000,000$, $d = 2$ and $p = 3$), which would be unacceptably slow. The key to making the algorithm fast is to avoid the time for matrix multiplication. Fortunately, the following lemma shows that for a randomly chosen vector $v_{init} \in V_{111}(5)$ the image of an element of $G$ on $v_{init}$ will uniquely determine the element. Given the generator $x' = x_{init}^w$ for a subgroup in $\mathcal{O}$, where $w$ is a word in $S$, one can then use the image $v_{init}^{w^{-1}x_{init}w}$ of $v_{init}$ under $x'$ as input to $h_{\alpha,\beta}$ instead of the matrix $x_{init}^w$. This results in a cost of $2|w|+1$ vector–matrix multiplications instead of $2|w|+1$ matrix multiplications, saving a factor of roughly $n$ in time, for $n = 111$.

**Lemma 3.1** *Let $m$ be the maximum dimension of any fixed point subspace of $G \leq GL(n, q)$. The probability that a randomly chosen vector is not fixed by any non-identity element of $G$ is at least $1 - |G|/q^{n-m}$.*

*Proof:* At most $(|G|-1)(q^m-1)$ non-zero vectors of $V$ will be fixed by a non-identity element of $G$. Thus, the probability that a randomly chosen vector is not fixed by any non-identity element of $G$ is at least $1 - q^m|G|/q^n$. □

**Remark** One can achieve finer estimates by also considering the size of the conjugacy class corresponding to $m$ and the second largest dimension, $m_2$, of a fixed point subspace. With the use of a character table, one can do still better. It is also clear that the use of $k$ independent initial random vectors increases the probability to at least $1-(|G|/q^{n-m})^k$.

For Lyons' group, the image of $v_{init}$ provides a unique signature of a group element with probability at least $1 - 3.8 \times 10^{-23}$. To determine this, we computed the dimension of the fixed point subspace for each conjugacy class of elements of prime order, since this dimension is maximized for such conjugacy classes. For Lyons' group, $m = 55$ (corresponding to the involutions) and $|G| = 5.2 \times 10^{16}$.

### 3.2 Software Optimizations

The first version of the program was tested on Lyons' group, and it is estimated that it would still have required more than a month of CPU time to complete. Hence, a series of software optimizations were successively applied to reduce the computation time to the four CPU days currently observed. We carry along and further refine the lower bound of $|A|2d(p-1)$ matrix-vector multiplications required from section signatures in order to illustrate how each successive optimization is expected to lower the CPU time. The actual number of matrix-vector multiplications required was usually larger by some proportional factor.

#### 3.2.1 Faster Matrix-Vector Multiplication

A lookup table of all linear combinations of each set of four adjacent rows under $GF(5)$ is kept for each matrix. This achieves a computation time for a matrix-vector multiplication in $GL(111, 5)$ of 900 microseconds on a SPARCserver 670MP. Such a lookup table was first used by Arlazarov et al. [1, 2], and was popularized by Parker in his software for the meataxe algorithm [11].

#### 3.2.2 Shallow Search Trees

We add 50 redundant generators chosen at random to the original generating set of size 2 and then add in all inverses. This decreases the depth of the search tree $d$, resulting in shorter words. If the number of generators $|S|$ is 2, then $d \geq \log_2|\mathcal{O}|/(\log_2|S|) > 30$. If $|S| = 104$, then $d \geq \log_2|\mathcal{O}|/(\log_2|S|) > 3.5$. Experimentally, the value of $d$ is observed to be close to the lower bound. Towards the end of the computation, when most of the elements of $\mathcal{O}$ have been discovered, we revert to the original, smaller generating set for finding the last ones.

#### 3.2.3 Fast Completion of Subgroup Orbits

We choose a subgroup, and note that the elements of the conjugacy class divide up into orbits under the action of this subgroup. In our case, the subgroup is of order 67, and all orbits are of length 67 or 1. All elements (matrices) in the subgroup are pre-computed. Every time we discover a new element of the conjugacy class, we can then conclude that all remaining elements of the orbit are not yet in the hash table. Thus, for each remaining element $\langle x \rangle$ of the orbit, we need only probe for the next `NULL` slot. If $A[\alpha(\beta^j(x))]$ is not `NULL` for some $j$, then we can skip the test for a match, since $A[\alpha(\beta^j(x))]$ cannot match $x$. If $\ell$ is the length of the orbit, then this reduces the approximate lower bound on the time to $|A|2d((p-1)/\ell + (\ell-1)/\ell)$ matrix-vector multiplications. Furthermore, when all orbits are of length $\ell$, each application of a generator will find $\ell$ elements, and so $d \geq \log_2(|\mathcal{O}|)/(\log_2(|S|\ell)) > 1.8$ in this case. Experimentally, we observe that 6,968 ($|S|\ell$) new elements are found after the first level ($d = 1$), and 9,536,646 new elements are found during the second round (for $d = 2$). An obvious generalization, albeit potentially less efficient, exists for non-cyclic subgroups of $G$.

In our situation, we invoke Lemma 2.1 to show that no element of $\mathcal{O}$ commutes with an element of order 67, and

so the orbits of an element of order 67 acting on $\mathcal{O}$ all have length 67. To see this, set $\ell = 67$ and let $\mathcal{O}$ be the "small" conjugacy class of subgroups of order 3. Direct computer calculation shows $d_1 = 3$ and $d_2 = 21$ in the lemma. The lemma will often have applicability, since elements of larger order tend to have smaller dimensional fixed point subspaces.

#### 3.2.4 Elimination of Spurious Matches through Check Bits

We maintain an array of check bits for each entry in our hash table. This is used to efficiently recognize hash collisions. When $h_{\alpha,\beta}(x) = i$, we store four check bits in $B[i]$ derived from the computer encoding of $x$. Then whenever we must check if $A[i]$ matches $y$, with $\langle y \rangle \in \mathcal{O}$, we first check if $B[i]$ equals the four check bits for $y$. If not, we can quickly eliminate the possibility of a match. This has the effect of replacing $p-1$ by $\max((p-1)/16, 1)$ in the formula for the minimum number of matrix-vector computations. It is especially important in the latter phase, when few new orbits are discovered.

#### 3.2.5 Pre-computation of Common Subwords

Some matrix-vector multiplications can be saved through pre-computation (matrix multiplication) of subwords. This is especially valuable in conjunction with the optimization on subgroup orbits in section 3.2.3. The subword corresponding to the initial element of the orbit can be precomputed at the first encounter, saving computations during the rest of that orbit.

#### 3.2.6 Caching Common Prefixes

The image of $v_{init}$ under prefixes of words in $S$ can be cached, instead of repeatedly computed.

### 4 Machines

Our SPARCserver 670MP is a general user machine, but it has four CPU's and usually at least one CPU was free for our experiment. So, CPU contention was minimal. It had 128 Megabytes of memory. Since the process was using about 154 Megabytes (with a smaller active working set), and we were competing with a general user population, memory contention may have been significant, but it did not greatly affect the overall result. The SPARCserver is an older 24 MHz machine that is not superscalar. Our tests of matrix-vector multiplication indicate that newer workstations would achieve a factor of two to four times better performance.

The sequential software for enumeration of a conjugacy class of elements of order 3 in a 111-dimensional matrix group over GF(5) was written in AKCL 1-615, a dialect of COMMON LISP. The matrix-vector multiplication routine was written in C. The implementation consists of approximately 1,750 lines of code for all routines and comments, and is self-contained.

The MasPar MP-1 has 4K processing elements (PEs) with 64KB of memory per PE. The machine architecture consists of a front-end computer (DECstation 5240), an asynchronous control unit (ACU) and a data parallel unit (DPU) or backend array of PEs. The PEs of the DPU are 4-bit RISC machines with indirect addressing capability and which are collected into clusters of 16 with very fast local communication.

The parallel software is written in C and MPL (a MasPar enhancement of C with syntax for specifying parallel operations on the DPU and communication between the front-end, ACU and DPU). It consists of about 1,500 lines of C and MPL code. It uses the LISP code to initialize the data structures. Further, it does not implement all of the software optimizations, due to the differing architecture.

### 5 Parallel Implementation

In the initial design of the parallel code we decided to leverage the existing serial code and to use the back–end simply as a fast matrix–vector multiply engine. This yielded an implementation that performed in one day of CPU time. The goal was to develop a fast systolic matrix–vector multiply routine in which the matrices and vectors are stored in the DPU in a distributed fashion. Since there are 16 PEs per cluster we extended the size of the problem from 111 to 112 (note: $112 = 7 \times 16$). This allowed us to store 7 components (7 4–bit quantities) per 32-bit word. Since the computation occurs over $GF(5)$ only 3 bits are required to represent each field element. The chosen representation wasted 1 bit per element and one 4-bit quantity per 32-bit word, but this simplified the computation. Thus, one 112-vector is represented by a single parallel (plural) integer variable distributed across the 16 PEs of a cluster. A $112 \times 112$ matrix is represented by a single plural integer variable stored over 112 clusters.

Recently we have developed an alternative scheme (still in progress), which finds the signature for a different word conjugation problem on each cluster. This yielded an implementation that currently performs in 12 hours of CPU time. In this model the vector is stored in the same representation as above. However, now all of the generating matrices are replicated on each cluster. Thus, a single matrix requires a plural integer array of 112 entries. This requirement reduces the number of matrices that may be pre-stored in the DPU, but increases the number of problems which can be solved in parallel to 256, the number of clusters. The communication time is measured at 2 of the 12 hours and is independent of the number of the processors. We expect a linear speedup in the remaining 10 hours of backend CPU time.

As part of the alternative scheme, we employ a new optimization especially suited to this software architecture. The DPU computes $\alpha(x)$ and returns $i$. In the first phase, we never test whether a computed $A[i]$ matches $x$. We note only whether $A[i]$ is `NULL` or not. If $A[i]$ is not `NULL`, then we assume that $x$ is already in the search tree and we discard $x$. By this heuristic, we may fail to enter an element $x \in \mathcal{O}$ to the search tree when we first see it. However, as long as we successfully add another element of the same orbit, we can then employ the fast completion of subgroup orbits discussed in section 3.2.3. This tells us that all remaining elements of the orbit do not match any existing element of the search tree, and we can probe for the next `NULL` slot.

If any elements of $\mathcal{O}$ are not discovered in this first pass, a second pass is made to find those elements that are still missing. The advantage of this scheme is that in the first pass, we do not have to find the first index $i$ in the sequence $(\alpha(x), \beta(\alpha(x)), \beta^2(\alpha(x)), \ldots)$ for which $A[i]$ matches $x$. This avoids the necessity of keeping 255 clusters idle while one cluster tries to find a match for an element later in the sequence.

## 6 Acknowledgements

The authors gratefully acknowledge discussions with the following people: Holger Gollan, Alexander Hulpke, Bill Kantor, Wolfgang Lempken, Steve Linton, Gene Luks, Klaus Lux, Gerhard Michler, Reiner Staszewski and Michael Weller. The authors especially acknowledge that the matrix generators for Lyons's group, for which these techniques were tested, were constructed by Robert Wilson and provided to us by Klaus Lux. Finally, we thank Bill Kantor for originally suggesting the "testbed" of Lyons's groups.

## References

[1] Aho, Hopcroft and Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1974, p. 245.

[2] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod and I.A. Faradzev, "On Economical Construction of the Transitive Closure of a Directed Graph", Dokl. Nauk SSSR **194**, pp. 487–488. English translation in Soviet Math. Dokl. **11**:5, pp. 1209–1210.

[3] L. Babai, "Local Expansion of Vertex-Transitive Graphs and Random Generation in Finite Groups", *Proc. 23rd ACM STOC* (1991).

[4] L. Babai and R. Beals, "Las Vegas Algorithms for Matrix Groups", to appear.

[5] J. H. Conway, R. T. Curtis, S. P. Norton, R. A. Parker, R. A. Wilson: *Atlas of Finite Groups*, Clarendon Press, Oxford (1985).

[6] C. Leedham-Green et al., Lecture at the MAGMA Conference, London, England, July, 1993.

[7] S. Linton, "The Art and Science of Computing in Large Groups", to appear in *Proceedings of the Computational Algebra and Number Theory conference (CANT)*, Sydney, November, 1992.

[8] E. M. Luks, "Computing in Solvable Matrix Groups", Proc. 33rd IEEE FOCS (1992), pp. 111–120.

[9] R. Lyons, "Evidence for a New Finite Simple Group", Journal of Algebra, **20** (1972), pp. 540–569.

[10] G. Michler, lecture at Northeastern University, October, 1993.

[11] R. Parker, "The computer calculation of modular characters. (The Meat-Axe)", in M. Atkinson (ed.), *Computational Group Theory*, Academic Press, London, pp. 267–274, 1984.

[12] C. C. Sims, "The Existence and Uniqueness of Lyons' Group", Proc. of the Gainseville Conference (1972), pp. 138–141.