EXTENDING THE PRODUCT REPLACEMENT ALGORITHM

HENRIK BÄÄRNHIELM, CHARLES LEEDHAM-GREEN, AND LAWRENCE PETTIT

Abstract. We present three extensions to the product replacement algorithm. First we prove that the well-known accumulator extension does not increase the mixing time. Then we present a new heuristic extension, called the *Prospector*, with the aim of finding random elements with short straight line programs in the given generators. Finally we analyse the behaviour of the well-known accelerator extension on a large generating set.

The Prospector has been implemented in Magma, and experimental evidence is provided which indicates that it is a very practical method.

1. Introduction

Let G be a black box group[11, pp. 17], defined by a generating set X. The problem that we wish to address is how best to find an element of G that belongs to some given subset S of G as a straight line program (abbreviated `SLP`) in X, by random search.

The only known algorithms for solving this problem that work well in practice are variations of product replacement[3], which is implemented in both the two major software systems for group theory, Magma[2] and GAP[5]. Better algorithms exist in many important special cases, but we are assuming here that we have a true black box group, as opposed to a permutation group, or a group of known isomorphism type. There are three aspects of this problem that we wish to consider.

(1) Product replacement in its original form consists of a team which is updated and the new element of the team is then returned. This has the defect that there is a systematic bias against certain group elements. Although this bias is generally harmless, it can be (and generally is) removed by adding an accumulator to the process[7]. Then the team and the accumulator are updated, and the new value of the accumulator is returned. There is no systematic bias in the value of the accumulator. A question of great theoretical interest is the mixing time of product replacement. When no accumulator is used this mixing time has been studied by Diaconis and Saloff-Coste, and by Pak who proved that the mixing time is polynomial[9, 10]. Our first objective is to prove that the mixing time for product replacement with an accumulator is not significantly worse than the mixing time without an accumulator; that it to say, the mixing time for the accumulator is not seriously worse than the mixing time for the team. This is done in Section 2.

(2) We are not simply looking for an element of G of a particular type; we are looking for such an element as an `SLP` in X. Clearly we should like this length to be as short as possible. If the proportion of elements of G of the required type is $1/\varepsilon$ then the length of the SLP will need to be at least $O(\log(\varepsilon))$; but the graph in which product replacement searches has high valency, so we can hope that the logarithm will be to a large base (or that the constant concealed by the O()-notation will be small). However, if we use

Date: 2007-04-28.

DRAFT

product replacement in the traditional way the length of the SLP will be the length of the search (or twice this length with an accumulator, as each step then requires two multiplications). In Section 3 we explore an idea whereby the length of the SLP may be closer to $O(\log(\varepsilon))$ rather than $O(\varepsilon)$. Our approach is heavily statistical, in that the process is based on sampling the elements that are returned, and not on any theoretical assessment of what elements should be returned. Thus the success or failure of the technique has to be decided experimentally. There is a balance to be considered between having a faster algorithm and a shorter straight line program. Our general bias is towards having a shorter SLP. There is an assumption that the SLP will be evaluated in a preimage of G in a larger group, and group operations may be slower in this larger group. This need for short SLPs has arisen as a practical issue in the matrix recognition project[8].

(3) We are concerned also with the question of what to do if the given generating set is large. There has been a suggestion that an "accelerator" should be used in this context[7]. In Section 4 we explore the use of an accelerator and other ideas, particularly within the context of obtaining short SLPs.

In summary, our objective is to construct a fast algorithm for finding random elements of a given type in a general black box group, and to provide a little more theoretical justification for using an accumulator.

2. The product replacement algorithm

We now give a brief but more formal description of the product replacement algorithm. It maintains an array A of length n of group elements, called the *seed* or the "team". Initially, A are filled with the generators of G. More specifically, if $X = \{x_1, \ldots, x_t\}$ then we let $A[i] = x_{i \bmod t}$ for $i = 1, \ldots, n$.

The original version of the algorithm, known as *Shake*, then works as in Algorithm 1 at each invocation.

Algorithm 1: Shake(A)

1 **Input:** The seed A.
2 **Output:** The next random element.
3 $i := \text{Random}(1, n)$
4 **repeat**
5 $\quad j := \text{Random}(1, n)$
6 **until** $i \neq j$
7 $A[i] := A[i]A[j]$
8 **return** $A[i]$

In [7] the variant known *Rattle* was introduced. The seed is then extended with an entry $A[0]$, called the *accumulator*. Each invocation is then as in Algorithm 2.

Algorithm 2: Rattle(A)

1 **Input:** The seed A.
2 **Output:** The next random element.
3 $i := \text{Random}(1, n)$
4 **repeat**
5 $\quad j := \text{Random}(1, n)$
6 **until** $i \neq j$
7 $k := \text{Random}(1, n)$
8 $A[i] := A[i]A[j]$
9 $A[0] := A[0]A[k]$
10 **return** $A[0]$

DRAFT

In [7] it is proved that the distribution of the elements returned from Rattle converges to the uniform distribution, which is not the case for Shake.

Normally one executes the algorithm s times as an initial *scrambling*, since at the beginning A consists of the generators of G, which might not be random. It follows from [10] that for Shake, the *mixing time*, the minimum number of scrambles that is necessary in order to produce nearly uniformly random elements of G, is polynomial in the size of the input, when n is large enough. In [7] it was conjectured that this is also true for Rattle.

Theorem 2.1. *The mixing time for Rattle is no worse than the mixing time for Shake.*

Proof. TODO □

The product replacement algorithm can be thought of as a random walk in a directed graph, where each vertex corresponds to an n-tuple of elements of G, given by the seed. There is an edge between two vertices if one can move between them using an invocation of the algorithm. Hence in general the vertices have very high valency. One can also note that the random walk determines a discrete Markov chain. Much can be said about the theoretical issues of the algorithm, see [9], but our concerns are different.

Since the algorithm constructs random elements directly from the generators of G, it is easy to also maintain `SLP`s of the elements of A and hence to return every random element as a straight line program in the generators of G. This is an essential feature which is used in many algorithms.

As the number of invocations of the product replacement algorithm increases, the elements that are returned become more random, but at the same time the lengths of their straight line programs increase. Hence one has to choose between good random elements, which might be necessary in order to find certain group elements, or short straight line programs, which might be necessary in order to evaluate them quickly on the generators of some other group. It might even be necessary to produce two different versions of an algorithm corresponding to these choices, which turn out to have different time complexity.

3. The Prospector

The dichotomy of the above situation is undesirable. We will describe a heuristic extension of the product replacement algorithm that tries to maintain short straight line programs while still producing elements that look random.

The idea is to try to detect when the elements that we find are random, and when we think that this is the case we save the seed at the current vertex. Then when we have moved a few steps we return to this vertex by resetting the seed. Later we might notice that the elements are in fact not random and then move on to another vertex. The similarity with a prospector looking for gold has led to the name *The Product Replacement Prospector*.

We call one instance of the ordinary product replacement algorithm a *random process*. The Prospector maintains two processes, a *sampler* process and a *seeker* process.

The seeker process is used by the Prospector to return random elements. The seed of this process will occasionally be reset as described above, to move back to a good vertex. We maintain `SLP`s of the seed in the seeker process in order to be able to return random elements as `SLP`s.

The sampler process is used to determine if we have good random elements. For this the Prospector needs as input a function $\tau : G \to \mathbb{N}$, which is used to produce test values of the elements. The sampler process is advanced as the ordinary product

DRAFT

replacement algorithm and the seed is never reset. We do not maintain SLPs of the seed in the sampler process.

More specifically, we choose parameters $m, t_l, c_a, a_l \in \mathbb{N}$, and at invocation l of the Prospector, the following happens:

(1) The next element $g_a \in G$ of the sampler process is fetched, and $\tau(g_a)$ is appended to a list T_a of sampler test values. Similarly $g_e \in G$ is fetched from the seeker process and $\tau(g_e)$ is appended to T_e. If $|T_a| > m$ then the first element of T_a is removed. Similarly if $|T_e| > m$.

(2) If $t_l \mid l$ and $|T_a| = m$ then the values of T_a are collected into buckets and a χ^2 test is performed that compares the bucket sizes either with their expected sizes, if these are known, or with the buckets from the previous χ^2 test using T_a, to see if there is any change. Note that these cases involve different types of χ^2 tests, as explained in the next section. The test uses significance level α_a.

(3) If c_a consecutive such tests are successful, the sampler process is stopped, so no more elements g_a are chosen and the list T_a is assumed to be a random distribution.

(4) If $t_l \mid l$ and $|T_e| = m$ then similarly the values of T_e are collected and a χ^2 test using significance level α_e is performed that compares them with the values in T_a. If the test is successful, the current vertex is assumed to be good.

(5) If $a_l \mid l$ then, if the current vertex is good, we reset the seeker seed to the last known good seed, or we save the current seeker seed as the last known good seed.

The parameters have the following interpretations:

- a_l The length of the "arm" that we traverse in the graph before we reset the seed and move back to a good vertex.
- m The sizes of the test batches in the χ^2 tests for the sampler and seeker processes.
- t_l The number of invocations between the χ^2 tests.
- c_a The number of consecutive tests that must be successful until we conclude that we probably have good enough random elements and the sampler process is stopped.

3.1. **The χ^2 tests.** Given a list of values T, we construct a list of buckets B. The number of buckets n_b is the number of distinct elements of T, so each bucket is labelled by such an element. The value of each bucket is the multiplicity of its label in T. In other words, we consider T as a multiset and let the buckets contain the multiplicities of each element.

Then we can perform a χ^2 test between B and the expected values E of the buckets. If we have a group G where the distribution of the values from the function t are known, then we use the classical Pearson's test, but in our situation it is more likely that this is not the case. Then we want to check if B and E come from the same distribution, *i.e.* we perform a χ^2 for homogeneity. We describe this test now.

Let $r_1 = \sum_{i=1}^{n_b} B[i]$, $r_2 = \sum_{i=1}^{n_b} E[i]$, $c_i = B[i] + E[i]$ for $i = 1, \ldots, n_b$, and let $r = r_1 + r_2 = \sum_{i=1}^{n_b} c_i$. Let S be the $2 \times n_b$ matrix with rows B and E. The test statistic is then

$$X = r \sum_{i=1}^{2} \sum_{j=1}^{n_b} \frac{(S_{i,j} - r_i c_j / r)^2}{r_i c_j} \tag{3.1}$$

and X is approximately χ^2-distributed with $f = n_b - 1$ degrees of freedom, assuming the null hypothesis. If t_l is small, so that about $1/10$ or more of the values

in B are less than 5 then we use the well-known Yates' correction for continuity on X: subtract 0.5 from each numerator, inside the brackets.

Now to perform the test with significance level α, we need a threshold value X_α, such that the probability that $X > X_\alpha$ is less than α if the null hypothesis is true. In our case α is given in the input, so we use the well-known Wilson-Hilferty normal approximation[6]. If we let

$$Y = \frac{\sqrt[3]{X/f} - 1 - 2/(9f)}{\sqrt{2/(9f)}} \tag{3.2}$$

then Y is approximately $N(0,1)$-distributed. Hence if Φ is the normal cumulative distribution function, the threshold is $Y_\alpha = \Phi^{-1}(1-\alpha) = \sqrt{2}\,\mathrm{erf}^{-1}(1-2\alpha)$, where $\mathrm{erf}(x)$ is the error function. Since in our case α will be close to 0, we use the well-known approximation to the inverse error function $\mathrm{erf}^{-1}(z)$ that is valid when z is close to 1:

$$\mathrm{erf}^{-1}(z) \approx \sqrt{\frac{\log \frac{2}{\pi(z-1)^2} - \log\log \frac{2}{\pi(z-1)^2}}{2}}. \tag{3.3}$$

3.2. **Analysis.** Given the immense difficulties of proving anything at all about the ordinary product replacement algorithm, we have little hope in proving that the Prospector produces good enough random elements with short `SLP`s. We will instead highlight the essential questions, and provide experimental evidence for our answers.

It is clear that the length of the straight line programs will be kept short if we reset the seed of the seeker process. The first question is instead the following. Assume we search randomly for elements of G that satisfy a certain property, and assume that the sampler process has stopped and the seeker process has found a good vertex. Hence our statistical tests have concluded that the elements appear to be random. Is it then sufficient to search in a neighbourhood of size a_l of the good vertex in the graph, in order to find an element of G with the required property? Do we have to make significantly more invocations of the seeker process in this neighbourhood in order to find such an element, compared to if we used the ordinary product replacement algorithm?

Most likely these questions cannot be answered for a general G, but our experiments indicate that for many interesting G the answers are "yes" and "no", respectively.

The other question is what test function τ to use. We want a test such that the distribution of images of τ on a sample of elements from G differ significantly depending on whether or not the sample comes from a uniformly random distribution. And optimally we want a test that works well for every G. This is probably impossible, but it might be possible to have one test that works reasonably well for permutation groups, and one for matrix groups. These are the contexts that we are most interested in.

For a matrix group $G \leqslant \mathrm{GL}(d,q)$ we could take $\tau : G \to \{1,\dots,d\}$ such that $\tau(g)$ is the number of factors of the characteristic polynomial of g.

For a permutation group $G \leqslant \mathrm{Sym}(d)$, we could try one of the following tests:

- $\tau(g)$ is the number of cycles of g
- $\tau(g)$ is the length of the longest cycle of g
- $\tau(g)$ is the number of fixed points of g

Evidently, in the first two cases the codomain of τ is $\{1,\dots,d\}$ and in the last case it is $\{0,\dots,d\}$. Our experiments indicate that all these tests work well in practice.

DRAFT

3.2.1. *Experimental evidence.* The Prospector has been implemented in MAGMA. We have performed tests to see how it compares with the ordinary product replacement algorithm. Hence we executed the MAGMA built-in product replacement machinery and our Prospector implementation simultaneously on a number of groups:

- $\mathrm{Sym}(d)$ for various d. Here we used a random generating set of size 3, and searched for elements with a cycle of length d. The proportion of these elements is $1/d$. The function τ was "number of cycles".
- $\mathrm{SL}(d, q)$ for various d, q. Here we used a random generating set of size 2, and searched for elements whose characteristic polynomial had a factor of degree d. The proportion of these elements is approximately $1/d$. The function τ was "number of factors of characteristic polynomial".

In each case, we made 10 consecutive searches for an element of the required kind. We recorded the average number of random selections r_1 and r_2 that was necessary in order to find such an element, for the ordinary product replacement algorithm and the Prospector, respectively. We also recorded the average length l of the resulting SLPs from the Prospector. Of course, for the ordinary product replacement algorithm, the lengths of the resulting SLPs were about $2r_1$.

The values of the various parameters that we used was: $n = 10$, $s = 0$, $a_l = 10$, $m = 10$, $t_l = 20$, $c_a = 5$. We used two different significance levels: $\alpha_a = \alpha_e = 0.01$ and $\alpha_a = \alpha_e = 0.05$. In all the tests, the significant levels were the same, so we used a table to find the correct χ^2 threshold value rather than using the approximations described in Section 3.1. Tables 1 and 2 contain our experimental results. In addition, for each group we have included the expected number of selections e for finding an element of the required kind.

Group	l	r_1	r_2	e
Sym(300)	21	303	349	300
Sym(500)	28	534	692	500
Sym(1000)	23	1330	1580	1000
$\mathrm{SL}(50, 11^3)$	22	33	49	50
$\mathrm{SL}(100, 11^3)$	24	75	127	100
$\mathrm{SL}(150, 11^3)$	25	117	166	150

TABLE 1. Experimental evidence with $\alpha_a = \alpha_e = 0.01$

Group	l	r_1	r_2	e
Sym(300)	235	353	385	300
Sym(500)	294	526	542	500
Sym(1000)	343	1358	1427	1000
$\mathrm{SL}(50, 11^3)$	25	52	75	50
$\mathrm{SL}(100, 11^3)$	41	87	222	100
$\mathrm{SL}(150, 11^3)$	82	208	232	150

TABLE 2. Experimental evidence with $\alpha_a = \alpha_e = 0.05$

We also performed these experiments on various sporadic groups, using some of the permutation and matrix representations available in the WEB-ATLAS[1]. We used the standard generators available in MAGMA. The elements we searched for were those with a particular order o, such that the proportion of these elements were moderately small. To calculate suitable o and the corresponding e for each group, we used the AtlasRep package in GAP, derived from the ATLAS[4]. Table 3 contains our experimental results in those cases that we used permutation representations.

DRAFT

The permutation group degree d is listed. Table 4 contains our experimental results in those cases that we used matrix representations. The matrix group degree d and field size q are listed. Some representations had elements of the required order as part of their standard generators, and those representations were deliberately avoided, since they would give the Prospector a big advantage. We also chose at most one permutation or matrix representation for each group. With the sporadic groups, we only used significant level 0.01.

Group	d	o	l	r_1	r_2	e
Co_2	4600	4	25	421	754	870
Co_3	552	5	27	369	187	250
M_{24}	2024	3	57	106	296	344
Fi_{22}	3510	5	22	798	469	600
HS	4125	3	25	208	704	360
McL	7128	3	25	998	832	941
Suz	1782	4	29	195	144	224
Ru	4060	5	62	243	372	231
HS	5600	3	26	364	573	360

Table 3. Experimental evidence for sporadics

Group	d	q	o	l	r_1	r_2	e
Co_1	24	2	8	64	183	209	292
Co_3	23	11	5	24	365	273	250
M_{22}	45	11	2	59	320	381	384
HS	77	11	3	28	433	411	360
McL	21	3	3	27	605	1171	941
He	51	25	5	25	211	261	300
HS	49	3	3	63	238	358	360

Table 4. Experimental evidence for sporadics

As can be seen in the tables, the Prospector usually seem to require slightly more random selections. However, we have performed χ^2 tests to see if the number of selections is significantly larger than the expected number. Using significance level 5% there has not been a single instance where the number of selections has been significantly too large.

The experiments were carried out using Magma V2.13-4, on a PC with an Intel Pentium 4 CPU running at 2.4 GHz and with 512 MB of RAM. The results indicate that the Prospector is a very practical extension of the product replacement algorithm.

4. Large generating sets

We now turn to the question of how to find random elements from a group given by a generating set that is large and very "non-random". Typical examples are $\mathrm{Sym}(d)$ generated by its transpositions, or $\mathrm{SL}(d, q)$ generated by its transvections. The Coxeter presentation of $\mathrm{Sym}(d)$ provides a more specific example: $\mathrm{Sym}(d)$ generated by the $n-1$ transpositions $(i, i+1)$ for $1 \leqslant i < n$, and $(n, 1)$.

As in our earlier considerations, the problem is essentially practical. With these bad generating sets the mixing time is too large to be really tractable. In [7] it is suggested that one could use an *accelerator* to somewhat decrease the mixing time, and we will analyse this experimentally. The idea is to choose one element of the

DRAFT

seed as the accelerator, and to use a modified version of the product replacement algorithm which is heavily biased to choose the accelerator at each step. One then mixes the bad generating set in this way, then chooses a small number of random elements as a new generating set and continues using ordinary product replacement (or the Prospector). More specifically, each invocation of Rattle with accelerator is as in Algorithm 3.

Algorithm 3: RattleWithAccelerator(A)

1 **Input:** The seed A.
2 **Output:** The next random element.
3 $i := \text{Random}(2, n)$
4 $j := \text{Random}(2, n)$
5 $k := \text{Random}(1, n)$
6 $A[i] := A[i]A[1]$
7 $A[1] := A[1]A[j]$
8 $A[0] := A[0]A[k]$
9 **return** $A[0]$

We have performed an experiment where we compared the mixing time of ordinary product replacement with the version where an accelerator is used. The groups we used were $\text{Sym}(d)$ on the above Coxeter generators, for various d. For each d, we searched for a d-cycle without performing any initial scrambling, and the mixing time is here defined as the number of invocations necessary to find this element. The results are shown in Figure 4.1. As can be seen, the accelerator clearly decreases the mixing time.

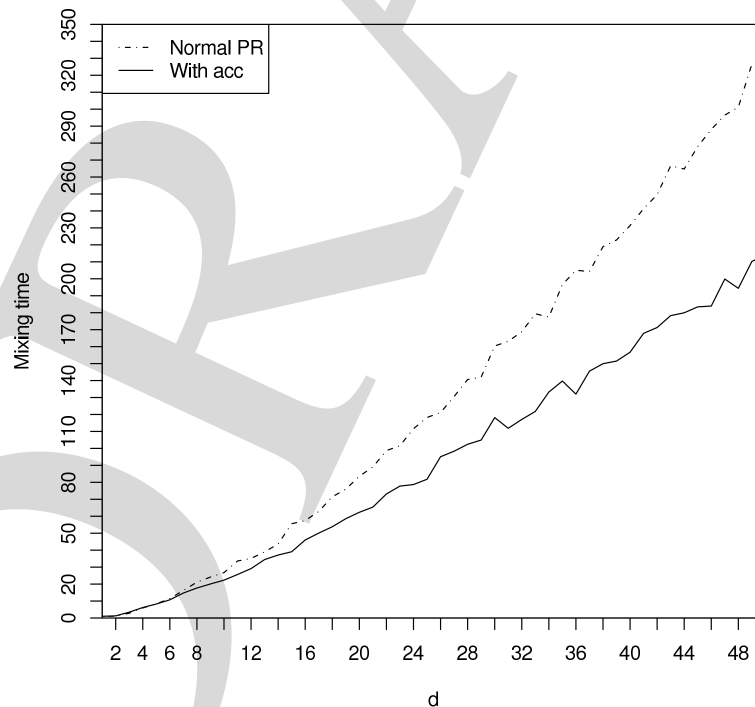

Figure 4.1. The mixing time for $\text{Sym}(d)$

DRAFT

References

1. R. Abbot, J. Bray, S. Linton, S. Nickerson, S. Norton, R. Parker, S. Rogers, I. Suleiman, J. Tripp, P. Walsh, and R. Wilson, *Atlas of Finite Group Representations*, http://brauer.maths.qmul.ac.uk/Atlas/.
2. Wieb Bosma, John Cannon, and Catherine Playoust, *The Magma algebra system. I. The user language*, J. Symbolic Comput. **24** (1997), no. 3-4, 235–265, Computational algebra and number theory (London, 1993). MR MR1484478
3. Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien, *Generating random elements of a finite group*, Comm. Algebra **23** (1995), no. 13, 4931–4948. MR MR1356111 (96h:20115)
4. J. H. Conway, R. T. Curtis, S. P. Norton, R. A. Parker, and R. A. Wilson, *Atlas of finite groups*, Oxford University Press, Eynsham, 1985, Maximal subgroups and ordinary characters for simple groups, With computational assistance from J. G. Thackray. MR MR827219 (88g:20025)
5. The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.4*, 2006, (http://www.gap-system.org).
6. M. M. Hilferty and E. B. Wilson, *The distribution of chi-square*, Proc. Natl. Acad. Sci. USA **17** (1931), no. 12, 684–688.
7. C. R. Leedham-Green and Scott H. Murray, *Variants of product replacement*, Computational and statistical group theory (Las Vegas, NV/Hoboken, NJ, 2001), Contemp. Math., vol. 298, Amer. Math. Soc., Providence, RI, 2002, pp. 97–104. MR MR1929718 (2003h:20003)
8. Charles R. Leedham-Green, *The computational matrix group project*, Groups and computation, III (Columbus, OH, 1999), Ohio State Univ. Math. Res. Inst. Publ., vol. 8, de Gruyter, Berlin, 2001, pp. 229–247. MR MR1829483 (2002d:20084)
9. I. Pak, *What do we know about the product replacement algorithm?*, Groups and computation, III (Columbus, OH, 1999), Ohio State Univ. Math. Res. Inst. Publ., vol. 8, de Gruyter, Berlin, 2001, pp. 301–347. MR MR1829489 (2002d:20107)
10. Igor Pak, *The product replacement algorithm is polynomial*, FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science (Washington, DC, USA), IEEE Computer Society, 2000, pp. 476–485.
11. Ákos Seress, *Permutation group algorithms*, Cambridge Tracts in Mathematics, vol. 152, Cambridge University Press, Cambridge, 2003. MR MR1970241 (2004c:20008)

School of Mathematical Sciences, Queen Mary, University of London, Mile End Road, London E1 4NS, United Kingdom
URL: http://www.maths.qmul.ac.uk/~hb/
E-mail address: h.baarnhielm@qmul.ac.uk

School of Mathematical Sciences, Queen Mary, University of London, Mile End Road, London E1 4NS, United Kingdom
URL: http://www.maths.qmul.ac.uk/~crlg/
E-mail address: c.r.leedham-green@qmul.ac.uk

School of Mathematical Sciences, Queen Mary, University of London, Mile End Road, London E1 4NS, United Kingdom
URL: http://www.maths.qmul.ac.uk/~pettit/
E-mail address: l.pettit@qmul.ac.uk

DRAFT