FFPACK: Finite field linear algebra package

Jean-Guillaume Dumas, Pascal Giorgi, Clément Pernet

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL
n° 5668

FFPACK: Finite field linear algebra package

Jean-Guillaume Dumas *(UJF Grenoble)*
Pascal Giorgi
Clément Pernet *(UJF Grenoble)*

Janvier 2004

Research Report N° 2004-2

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

FFPACK: Finite field linear algebra package

Jean-Guillaume Dumas *(UJF Grenoble)*
Pascal Giorgi
Clément Pernet *(UJF Grenoble)*

Janvier 2004

Abstract

The FFLAS project has established that exact matrix multiplication over finite fields can be performed at the speed of the highly optimized numerical BLAS routines. Since many algorithms have been reduced to use matrix multiplication in order to be able to prove an optimal theoretical complexity, this paper shows that those optimal complexity algorithms, such as LSP factorization, rank determinant and inverse computation can also be the most efficient.

Keywords: Word size Finite fields; BLAS level 1-2-3; Linear Algebra Package; Matrix Multiplication; LSP Factorization

Résumé

Le projet FFLAS a montré que le calcul d'un produit matriciel sur les corps finis peut être aussi rapide que les routines numériques BLAS. En algèbre linéaire exacte beaucoup d'algorithmes se réduisent au produit matriciel afin de prouver une complexité théorique optimale. Dans ce papier nous montrons que des algorithmes basés sur le produit matriciel, tels que la factorisation LSP, le calcul du rang, le calcul du déterminant et l'inversion peuvent être aussi les plus efficaces en pratique.

Mots-clés: Corps Finis-Mot machines; Niveau BLAS 1-2-3; Package d'Algèbre Linéaire; Produit Matriciel; Factorisation LSP

FFPACK: Finite field linear algebra package *

Jean-Guillaume Dumas[1]
Pascal Giorgi[2]
Clément Pernet[1]

[1]Laboratoire LMC, 50, av. des Mathématiques B.P. 53 38041 Grenoble.
[2]Laboratoire LIP, ENS de Lyon, 46, Allée d'Italie, F69364 Lyon Cedex 07.

1 Introduction

Exact matrix multiplication over finite fields can now be performed at the speed of the highly optimized numerical BLAS routines. This has been established by the FFLAS project [7]. Moreover, since finite field computations e.g. do not suffer from numerical stability, this project showed also an easy effectiveness of the algorithms with even better arithmetic complexity (such as Winograd's variant of Strassen's fast matrix multiplication) [17].

Now for the applications. Many algorithms have been designed to use matrix multiplication in order to be able to prove an optimal theoretical complexity. In practice those algorithms were only seldom used. This is the case e.g. in many linear algebra problems such as determinant, rank, inverse, system solution or minimal and characteristic polynomial. Over finite fields or over the integers those finite field linear algebra routines are used to solve many different problems. Among them are integer polynomial factorization, Gröbner basis computation, integer system solving, large integer factorization, discrete logarithms, error correcting codes, etc. Even sparse or polynomial linear algebra needs some very efficient dense subroutines [11, 9]. We believe that with our kernel, each one of those optimal complexity algorithms can also be the most efficient.

The goal of this paper is to show the actual effectiveness of this belief for the factorization of any shape and any rank matrices. The application of this factorization to determinant, rank, and inverse is presented as well.
Some of the ideas from FFLAS, in particular the fast matrix multiplication algorithm for small prime fields, are now incorporated into the Maple computer algebra system since its version 8. Therefore an effort towards effective reduction has been made within Maple by A. Storjohann[21]. Effective reduction for minimal and characteristic polynomial were sketched in [19] and A. Steel has reported on similar efforts within his implementation of some Magma routines. We provide a full C++ package available directly[1] or through LINBOX[2][6]. Extending the work undertaken by the authors et al.[17, 7, 3, 10], this paper focuses

*E-mail addresses: {Jean-Guillaume.Dumas,Clement.Pernet}@imag.fr, Pascal.Giorgi@ens-lyon.fr
[1] www-lmc.imag.fr/lmc-mosaic/Jean-Guillaume.Dumas/FFLAS
[2] www.linalg.org

on matrix factorization, namely the exact equivalent of the LU factorization. Indeed, unlike numerical matrices, exact matrices are very often singular, even more so if the matrix is not square ! Consequently, Ibarra, Moran and Hui have developed generalizations of the LU factorization, namely the LSP and LQUP factorizations [15]. In section 4 we deal with the implementation of those two routines as well as memory optimized (an in-place and a cyclic block) versions using the fast matrix multiplication kernel. Those implementations require the resolution of triangular systems with matrix right or left hand side. In section 3 the triangular system solver (`Trsm` routine using BLAS terminology) is therefore studied. Then, in section 5, we propose different uses of the factorization routine to solve other classical linear algebra problems. In particular, speed ratios are presented and reflects the optimal behavior of our routines.

2 Base fields

The algorithms we present in this paper are written generically with regards to the field over which they operate, as long as they provide some conversion functions from a field element to a floating point representation and backwards. As demonstrated in [7] this is easily done for prime fields and also for other finite fields, via a q-adic transformation. The chosen interface is that of the LinBox fields [22, §5.3].

For our experiments we use some classical representations, e.g. modular prime fields, primitive roots Galois fields, Montgomery reduction, etc. implemented in different libraries, as in [7, 5]. Still and all, when no special implementation is required and when the prime field is small enough, one could rather use what we call a `Modular<double>` field representation. Indeed, the use of the BLAS imposes conversions between the field element representations and a corresponding floating point representation. Hence a lot of time consuming conversions can be avoided whenever the field element representation is already a floating point number. This is the case for the LinBox prime field `Modular<double>`, where the exact representation of an element is stored within the mantissa of a double precision floating point number. Of course all the arithmetic operations remain exact as they are always performed modulo a prime number.

3 Triangular system solving with matrix hand side

In this section we discuss the implementation of solvers for triangular systems with matrix right hand side (or equivalently left hand side). This is also the simultaneous resolution of n triangular systems. Without loss of generality for the triangularization, we here consider only the case where the row dimension, m, of the the triangular system is less than or equal to the column dimension, n. The resolution of such systems is a classical problem of linear algebra. It is e.g. one of the main operation in block Gaussian elimination. For solving triangular systems over finite fields, the block algorithm reduces to matrix multiplication and achieves the best known arithmetic complexity. Therefore, from now on we will denote by ω the exponent of square matrix multiplication (e.g. from 3 for classical, to 2.375477 for Coppersmith-Winograd). Moreover, we can bound $m \times k$ by $k \times n$ rectangular matrix multiplication, using $R(m,k,n) \leq C_\omega min(m,k,n)^{\omega-2} max(mk, mn, kn)$ [14, (2.5)]. In the following

subsections, we present the block recursive algorithm and two optimized implementation variants.

3.1 Scheme of the block recursive algorithm

The classical idea is to use the divide and conquer approach. Here, we consider the upper left triangular case without loss of generality, since the any combination of upper/lower and left/right triangular cases are similar: if U is upper triangular, L is lower triangular and B is rectangular, we call `ULeft-Trsm` the resolution of $UX = B$, `LLeft-Trsm` that of $LX = B$, `URight-Trsm` that of $XU = B$ and `LRight-Trsm` that of $XL = B$.

Algorithm `ULeft-Trsm`(A, B)
Input: $A \in {\mathbb{Z}_p}^{m \times m}$, $B \in {\mathbb{Z}_p}^{m \times n}$.
Output: $X \in {\mathbb{Z}_p}^{m \times n}$ such that $AX = B$.
Scheme
if m=1 **then**
$X := A_{1,1}^{-1} \times B$.
else *(splitting matrices into* $\lfloor \frac{m}{2} \rfloor$ *and* $\lceil \frac{m}{2} \rceil$ *blocks)*

$$\overbrace{\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}}^{X} = \overbrace{\begin{bmatrix} B_1 \\ B_2 \end{bmatrix}}^{B}$$

X_2 :=`ULeft-Trsm`(A_3, B_2).
$B_1 := B_1 - A_2 X_2$.
X_1 :=`ULeft-Trsm`(A_1, B_1).
return X.

Lemma 3.1 *Algorithm* `ULeft-Trsm` *is correct and its theoretical cost is bounded by* $\frac{C_\omega}{2(2^{\omega-2}-1)} n m^{\omega-1}$ *arithmetic operations in* $\mathbb{Z}_p$ *for* $m \leq n$.

Proof. The correctness of algorithm `ULeft-Trsm` can be proven by induction on the row dimension of the system. For this, one only has to note that

$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \text{ is solution} \iff \begin{cases} A_3 X_2 = B_2 \\ A_1 X_1 + A_2 X_2 = B_1 \end{cases}$$

Let $C(m,n)$ be the cost of algorithm `ULeft-Trsm` where m is the dimension of A and n the column dimension of B. It follows from the algorithm that $C(m,n) = 2C(\frac{m}{2}, n) + R(\frac{m}{2}, \frac{m}{2}, n)$. By counting each operation at one recursive step we have:

$$C(m,n) = \sum_{i=1}^{\log m} 2^{i-1} R(\frac{m}{2^i}, \frac{m}{2^i}, n)$$

Now, since $m \leq n$, we get $\forall i \; R(\frac{m}{2^i}, \frac{m}{2^i}, n) = C_\omega \left(\frac{m}{2^i}\right)^{\omega-1} n$ and therefore:

$$C(m,n) = \frac{C_\omega n m^{\omega-1}}{2} \sum_{i=1}^{\log m} \left(\frac{1}{2^i}\right)^{\omega-2}$$

which gives the $O(nm^{\omega-1})$ bound of the lemma.

3.2 Implementation using the BLAS "dtrsm"

Matrix multiplication speed over finite fields was improved in [7, 17] by the use of the numerical BLAS[3] library: matrices were converted to floating point representations (where the linear algebra routines are fast) and converted back to a finite field representation afterwards. The computations remained exact as long as no overflow occurred. An implementation of `ULeft-Trsm` can use the same techniques. Indeed, as soon as no overflow occurs one can replace the recursive call to `ULeft-Trsm` by the numerical BLAS *dtrsm* routine. But one can remark that approximate divisions can occur. So we need to ensure both that only exact divisions are performed and that no overflow appears. Not only one has to be careful for the result to remain within acceptable bounds, but, unlike matrix multiplication where data grows linearly, data involved in linear system grows exponentially as shown in the following.
The next two subsections first show how to deal with divisions, and then give an optimal theoretical bound on the coefficient growth and therefore an optimal threshold for the switch to the numerical call.

3.2.1 Dealing with divisions

In algorithm 3.1, divisions appear only within the last recursion's level. In the general case it cannot be predicted whether these divisions will be exact or not. However when the system is unitary (only 1's on the main diagonal) the division are of course exact and will even never be performed. Our idea is then to transform the initial system so that all the recursive calls to `ULeft-Trsm` are unitary. For a triangular system $AX = B$, it suffices to factor first the matrix A into $A = UD$, where U, D are respectively an upper unit triangular matrix and a diagonal matrix. Next the unitary system $UY = B$ is solved by any `ULeft-Trsm` (even a numerical one), without any division. The initial solution is then recovered over the finite field via $X = D^{-1}Y$. This normalization leads to an additional cost of:

- m inversions over $\mathbb{Z}_p$ for the computation of D^{-1}.
- $(m-1)\frac{m}{2} + mn$ multiplications over $\mathbb{Z}_p$ for the normalizations of U and X.

Nonetheless, in our case, we need to avoid divisions only during the numerical phase. cwTherefore, the normalization can take place only just before the numerical routine calls. Let β be the size of the system when we switch to a numerical computation. To compute the cost, we assume that $m = 2^i\beta$, where i is the number of recursive level of the algorithm `ULeft-Trsm`. The implementation can however handle any matrix size. Now, there are 2^i normalizations with systems of size β. This leads to an additional cost of:

- m inversions over $\mathbb{Z}_p$.
- $(\beta-1)\frac{m}{2} + mn$ multiplications over $\mathbb{Z}_p$.

This allows us to save $\left(\frac{1}{2} - \frac{1}{2^{i+1}}\right) m^2$ multiplications over $\mathbb{Z}_p$ from a whole normalization of the initial system. One iteration suffices to save $\frac{1}{4}m^2$ multiplications and we can save up to $\frac{1}{2}(m^2 - m)$ multiplications with $\log m$ iterations.

[3] `www.netlib.org/blas`

3.2.2 A theoretical threshold

The use of the BLAS routine trsm is the resolution of the triangular system over the integers (stored as `double` for `dtrsm` or `float` for `strsm`). The restriction is the coefficient growth in the solution. Indeed, the k^{th} value in the solution vector is a linear combination of the $(n-k)$ already computed next values. This implies a linear growth in the coefficient size of the solution, with respect to the system dimension. Now this resolution can only be performed if every element of the solution can be stored in the mantissa of the floating point representation (e.g. 53 bits for `double`). Therefore overflow control consists in finding the largest block dimension b, such that the result of the call to `dtrsm` will remain exact.

We now propose a bound for the values of the solutions of such a system; this bound is optimal (in the sense that there exists a worst case matching the bound when $n = 2^i b$). This enables the implementation of a cascading algorithm, starting recursively and taking advantage of the BLAS performances as soon as possible.

Theorem 3.2 *Let $T \in \mathbb{Z}^{n\times n}$ be a unit diagonal upper triangular matrix, and $b \in \mathbb{Z}^n$, with $|T| \leq p-1$ and $|b| \leq p-1$. Let $X = (x_i)_{i\in[1..n]} \in \mathbb{Z}^n$ be the solution of $T.X = b$ over the integers. Then, $\forall\ k \in [0..n-1]$:*

$$\begin{cases} (p-2)^k - p^k \leq 2\frac{x_{n-k}}{p-1} \leq p^k + (p-2)^k & \text{if } k \text{ is even} \\ -p^k - (p-2)^k \leq 2\frac{x_{n-k}}{p-1} \leq p^k - (p-2)^k & \text{if } k \text{ is odd} \end{cases}$$

The proof is presented in appendix A. The idea is to use an induction on k with the relation $x_k = b_k - \sum_{i=k+1}^{n} T_{k,i}x_i$. Two lower and an upper bounds for x_{n-k} are computed, depending whether k is even or odd.

Corollary 3.3 $|X| < \frac{p-1}{2}\left[p^{n-1} + (p-2)^{n-1}\right]$. *Moreover, this bound is optimal.*

Proof. We denote by $u_n = \frac{p-1}{2}\left[p^n - (p-2)^n\right]$ and $v_n = \frac{p-1}{2}\left[p^n + (p-2)^n\right]$ the bounds of the theorem 3.2. Now $\forall\ k \in [0..n-1]\ u_k \leq v_k \leq v_{n-1}$. Therefore the theorem 3.2 gives $\forall\ k \in [1..n]\ x_k \leq v_{n-1} \leq \frac{p-1}{2}\left[p^{n-1} + (p-2)^{n-1}\right]$

$$\text{Let } T = \begin{bmatrix} \ddots & \ddots & \ddots & \ddots & \ddots \\ & 1 & p-1 & 0 & p-1 \\ & & 1 & p-1 & 0 \\ & & & 1 & p-1 \\ & & & & 1 \end{bmatrix}, b = \begin{bmatrix} \vdots \\ 0 \\ p-1 \\ 0 \\ p-1 \end{bmatrix}$$

Then the solution $X = (x_i)_{i\in[1..n]} \in \mathbb{Z}^n$ of the system $T.X = b$ satisfies $\forall\ k \in [0..n-1]\ |x_{n-k}| = v_k$ Thus, for a given p, the dimension n of the system must satisfy

$$\frac{p-1}{2}\left[p^{n-1} + (p-2)^{n-1}\right] < 2^m \tag{1}$$

where m is the size of the mantissa so that the resolution over the integers using the BLAS trsm routine is exact. For instance, with a 53 bits mantissa, this gives quite small matrices, namely at most 55×55 for $p = 2$, at most 4×4 for $p \leq 9739$, and at most $p = 94906249$ for 2×2 matrices. Nevertheless, this technique is speed-worthy in most cases as shown in section 3.4.

3.3 Recursive with delayed modulus

In the previous section we noticed that BLAS routines within `Trsm` are used only for small systems. An alternative is to change the cascade: instead of calling the BLAS, one could switch to the classical iterative algorithm: Let $A \in {\mathbb{Z}_p}^{m\times m}$ and $B, X \in {\mathbb{Z}_p}^{m\times n}$ such that $AX = B$, then

$$\forall i, X_{i,*} = \frac{1}{A_{i,i}}(B_{i,*} - A_{i,[i+1..m]}X_{[i+1..m],*}) \tag{2}$$

The idea is that the iterative algorithm computes only one row of the whole solution at a time. Therefore its threshold is far below the one of the BLAS routine, namely it requires only

$$n(p-1)^2 < 2^m \tag{3}$$

Resultantly, an implementation of this iterative algorithm depends mainly on the matrix-vector product. The arithmetical cost of such an algorithm is now cubic in the size of the system, where blocking improved the theoretical complexity. Anyway, in practice fast matrix multiplication algorithms are not better than the classical one for such small matrices [7, §3.3.2]. In section 3.4 we compare both hybrid implementations with different thresholds to the pure recursive one.

Now we focus on the dot product operation, base for matrix-vector product. We use the results of [5], extending those of [7, §3.1]. There several implementations of a dot product are proposed and compared on different architectures. According to [5], where many different implementations are compared (Zech log, Montgomery, float, ...), the best implementation is a combination of a conversion to floating point representation with delayed modulus (for big prime and vector size) and an overflow detection trick (for smaller prime and vector size).

The first idea is to specialize dot product in order to make several multiplications and additions before performing the division (which is then delayed). Indeed, one needs to perform a division only when the intermediate result might overflow. Now, if the available mantissa is of m bits and the modulo is p, divisions happen at worst every n multiplications where n satisfies condition (3). There the best compromise has to be chosen between speed of computation and available mantissa. A double floating point representation gives actually the best performances for most of the vector and prime sizes [5]. Moreover one can then perform the division "à la NTL" using a floating point precomputation of the inverse: $a * b \mod p = a * b - \lfloor a * b * p^{-1} \rfloor * p$. The second idea is to use an integer representation and to let the overflow occur. Then one should detect this overflow and correct the result if needed. Indeed, suppose that we have added a product ab to the accumulated result t and that an overflow has occurred. The variable t now contains actually $t - 2^m$. Well, the idea is just to precompute a correction $CORR = 2^m \mod p$ and add this correction whenever an unsigned overflow has occurred. Now for a portable unsigned overflow detection, we use a trick of B. Hovinen [13]: since $0 < ab < 2^m$, an unsigned overflow has occurred if and only if $t + ab < t$. Of course, better performances are attained when several products are grouped (whenever possible) so that test and correction are also delayed [5]. Figure 1, shows the "quasi-optimal performances" obtained using both techniques on a pIII: the first part of the curve reflects a blocked version of the overflow and correction idea, the second and constant part of the curve

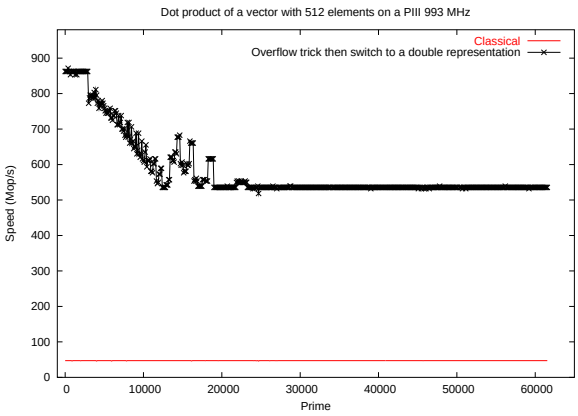

Figure 1: Speed improvement of dot product by delayed division, on a PIII, 993 MHz

(for bigger primes) is the very good behavior of a simple delayed division with a double floating point representation and "à la NTL" divisions.

3.4 "Trsm" implementations behavior

As shown in section 3.1 the block recursive algorithm `Trsm` is based on matrix multiplications. This allows us to use the fast matrix multiplication routine of the FFLAS package [7]. This is an exact wrapping of the ATLAS library[4][23] used as a kernel to implement the `Trsm` variants. In the following we denote by "pure rec", the implementation of the recursive `Trsm` described in section 3.1, by "BLAS", the variant of section 3.2 with optimal threshold and by "delayed$_t$", the variant of section 3.3 with a variable threshold t. In our comparisons we use the fields presented in section 2 as base fields and the version 3.4.2 of ATLAS. Performances are expressed in million of finite field operations (`Mfops`) per second for $n \times n$ dense systems.

n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
pure rec.	853	1216	1470	1891	2059	2184
BLAS	**1306**	**1715**	**1851**	**2312**	**2549**	**2660**
delayed$_{100}$	1163	1417	1538	1869	2042	2137
delayed$_{50}$	1163	1491	1639	1955	2067	2171
delayed$_{23}$	1015	1465	1612	2010	2158	2186
delayed$_3$	901	1261	1470	1937	2134	2166

n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
pure rec.	810	1225	1449	1886	2037	2184
BLAS	1066	1504	**1639**	**2099**	**2321**	**2378**
delayed$_{100}$	1142	1383	1538	1860	2019	2143
delayed$_{50}$	**1163**	**1517**	**1639**	1955	2080	2172
delayed$_{23}$	1015	1478	1612	2020	2146	2184
delayed$_3$	914	1279	1449	1941	2139	2159

Table 1: Comparing speed (Mfops) of `Trsm` using `modular<double>`, on a P4, 2.4GHz (Upper table is over Z_5, lower table is over Z_{32749})

[4] http://math-atlas.sourceforge.net

n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
pure rec.	571	853	999	1500	1708	1960
BLAS	688	1039	1190	**1684**	**1956**	**2245**
delayed$_{150}$	799	**1113**	909	1253	1658	2052
delayed$_{100}$	**831**	1092	**1265**	1571	1669	2046
delayed$_{23}$	646	991	1162	1584	1796	2086
delayed$_3$	528	755	917	1369	1639	1903

n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
pure rec.	551	786	1010	1454	1694	1929
BLAS	547	828	990	1449	1731	1984
delayed$_{100}$	703	958	1162	1506	1570	1978
delayed$_{50}$	**842**	**1113**	**1282**	**1731**	**1890**	**2174**
delayed$_{23}$	653	952	1086	1556	1800	2054
delayed$_3$	528	769	900	1367	1664	1911

Table 2: Comparing speed (Mfops) of `Trsm` using `givaro-ZpZ`, on a P4, 2.4GHz (Upper table is over Z_5, lower table is over Z_{32749})

One can see from table 1 that the "BLAS" `Trsm` variant with a `Modular<double>` representation is the most efficient choice for small primes (here switching to BLAS happens for $n = 23$ when $p = 5$ and $m = 53$). Now for big primes, despite the very small granularity (switching to BLAS happens only for $n = 3$ when $p = 32749$ and $m = 53$), this choice remains the best as soon as the systems are bigger than 1000×1000. This is because grouping operations into blocks speeds up the computation. However in the case of smaller systems, the "delayed" variant is more efficient, due to the good behavior of dot product. Then, table 2 shows that the conversions from machine integers to floating points numbers, needed by the "BLAS" variant, can become too big a price to pay. Therefore, the "delayed" variant is the best choice. However, for large matrices, conversions ($O(n^2)$) are dominated by computations ($O(n^\omega)$), and the "BLAS" variant is again the fastest one, provided that the field is small enough. Finally, one would rather use a `Modular<double>` representation and the "BLAS" `Trsm` variant in most cases. However, when the base field is already specified one can search for delayed thresholds which could provide slightly better performances.

4 Triangularizations

We now come to the core of this paper, namely the matrix multiplication based algorithms for triangularization over finite fields. The main concern here is the singularity of the matrices. Moreover, practical implementations need to efficiently deal with the rank profile, unbalanced dimensions, memory management, recursive thresholds, etc. Therefore, in this section we present three variants of the recursive exact triangularization. First the classical `LSP` of Ibarra et al. is sketched. In order to reduce its memory requirements, a first version, `LUdivine`, stores L in-place, but temporarily uses some extra memory. Our last implementation is fully in-place without any extra memory requirements and corresponds to Ibarra's `LQUP`. From both `LUdivine` and `LQUP` one can easily recover the `LSP` via some extractions and permutations.

4.1 LSP Factorization

The `LSP` factorization is a generalization of the well known block `LUP` factorization for the singular case [1]. Let A be a $m \times n$ matrix, we want to compute

the triple $< L, S, P >$ such that $A = LSP$. The matrices `L` and `P` are as in `LUP` factorization and `S` reduces to a non-singular upper triangular matrix when zero rows are deleted. The algorithm with best known complexity computing this factorization uses a divide and conquer approach and reduces to matrix multiplication [15]. Let us describe briefly the behavior of this algorithm. The

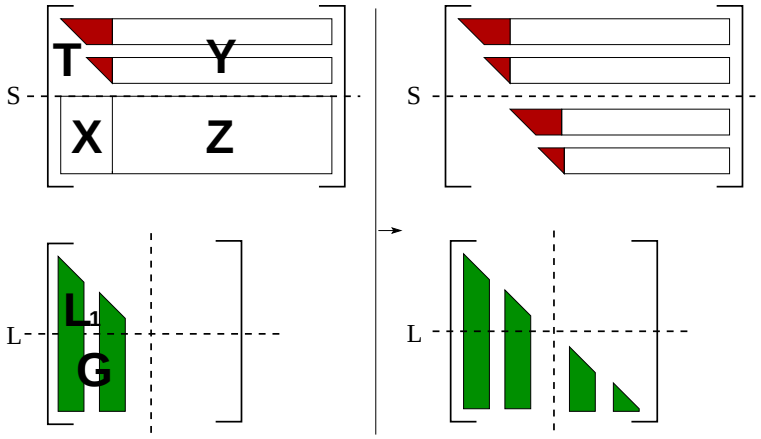

Figure 2: Principle of the LSP factorization

algorithm is recursive: first, it splits A in halves and performs a recursive call on the top block. It thus gives the T, Y and L_1 blocks of figure 2. Then, after some permutations ($[XZ] = [A_{21}A_{22}]P$), it computes G such that $GT = X$ via `Trsm`, replaces X by zeroes and eventually updates $Z = Z - GY$. The third step is a recursive call on Z. We let the readers refer e.g. to [2, (2.7c)] for further details.

Lemma 4.1 *Algorithm* `LSP` *is correct. The dominant term of its theoretical cost is bounded by* $\frac{C_\omega}{2^{\omega-1}-2} m^{\omega-1}\left(n + \frac{m}{2^{\omega}-2}\right)$ *arithmetic operations in* $\mathbb{Z}_p$ *for* $m \leq n$.

This refines Ibarra's original factor[15, Theorem 2.1] from $3n$ to $n + \frac{m}{2^{\omega}-2}$. Moreover, when each one of the intermediate block is of full rank, this factor even reduces to $n - m\frac{2^{\omega-2}-1}{2^{\omega-1}-1}$[18, Theorem 1]. And this nicely gives $\frac{2}{3}n^3$, when $\omega = 3$, $n = m$ and $C_\omega = C_3 = 2$.

The point here is that, L being square $m \times m$ does not fit in place under S. Therefore a first implementation produces an extra triangular matrix. The following subsections address this memory issue.

4.2 LUdivine

The main concern with the direct implementation of the `LSP` algorithm, is the storage of the matrix L: it can not be stored with its zero columns under S (as shown in figure 2). Actually, there is enough room under S to store all the non zero entries of L, as shown in figure 3. Storing only the non zero columns of L is the goal of the `LUdivine` variant. One can notice that this operation corresponds to the storage of $\tilde{L} = LQ$ instead of L, where Q is a permutation matrix such that $Q^T S$ is upper triangular. Consequently, the recovery of L from the computed $\tilde{L}$ is straightforward. Note that this $\tilde{L}$ corresponds to the echelon form of [16, §2] up to some transpositions.

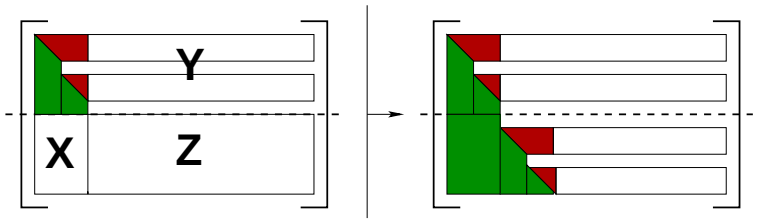

Figure 3: Principle of the LUdivine factorization

Further developments on this implementation are presented in [3, 18]. However, this implementation is still not fully in place. Indeed, to solve the triangular system $G = X.T^{-1}$, one has then to convert T to an upper triangular matrix stored in a temporary memory space. In the same way, the matrix product $Z = Z - GY$ also requires a temporary memory allocation, since rows of Y have to be shifted. This motivates the introduction of the `LQUP` decomposition.

4.3 LQUP

To solve the data locality issues, due to zero rows inside S, one can prefer to compute the `LQUP` factorization, also introduced in [15]. It consists in a slight modification of the `LSP` factorization: S is replaced by U, the corresponding upper triangular matrix, after the permutation of the zero rows. The tranpose of this row permutation is stored in Q.

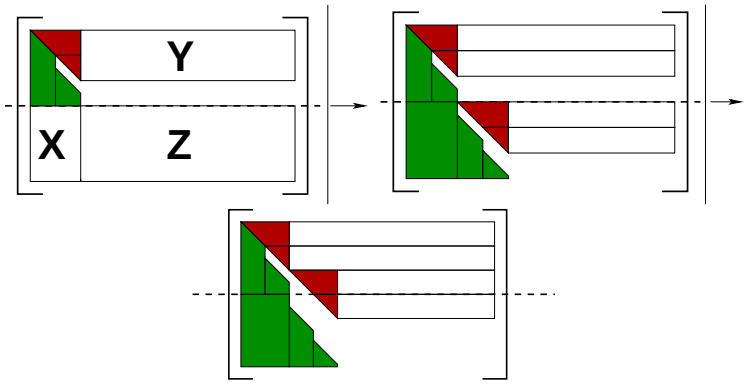

Figure 4: Principle of the LQUP factorization

This prevents the use of temporaries for Y and T, since the triangles in U are now contiguous. Moreover, the number of instructions to perform the row permutations is lower than the number of instructions to perform the block copies of `LUdivine` or `LSP`. Furthermore, our implementation of `LQUP` also uses the trick of `LUdivine`, namely storing L in its compressed form $\tilde{L}$. Thanks to all these improvements, this triangulation appears to be fully in place. As will be shown in section 4.4, it is also more efficient. Here again, the `LSP` and `LQUP` factorizations are simply connected via $S = QU$. So the recovery of the `LSP` is still straightforward.

4.4 Comparisons

As shown in previous sections the three variants of triangularization mainly differ by their memory management. Indeed, the main operations remain matrix multiplications and triangular system solving. Therefore, the implementation of all these variants use the fast matrix multiplication routine of the FFLAS package [7] and the triangular system solver of subsection 3.2 as kernel. The results are impressive: for example, table 3 shows that it is possible to triangularize a 5000×5000 matrix over a finite field in 32.9 seconds. We now compare the three routine speed and memory usage with the same kernels: a `Modular<double>` representation (so that no conversion overhead occur) and the recursive with BLAS `Trsm`. For table 3, we used random dense square matrices (but with $3n$

n	*400*	*1000*	*3000*	*5000*	*8000*	*10000*
`LSP`	0.05	0.48	8.29	35.56	258.7	1297
`LUdivine`	0.05	0.47	7.94	33.24	451.8	1289
`LQUP`	0.05	0.46	7.79	32.9	183.6	1014

Table 3: Comparing real time (seconds) of `LSP`, `LUdivine`, `LQUP` over Z_{101}, on a P4, 2.4GHz

non-zero entries so as to have rank defficient matrices. The timings given in table 3 are close since the dominating operations of the three routines are similar. `LSP` is slower, since it performs some useless zero matrix multiplications when computing $Z = Z - GY$ (section 4.2). `LQUP` is slightly faster than `LUdivine` since row permutations involve less operations than the whole block copy of `LUdivine` (section 4.3). However these operations do not dominate the cost of the factorization, and they are therefore of little influence on the total timings. This is true until the matrix size induces some swapping, around 8000×8000.

Now for the memory usage, the fully in-place implementation of `LQUP` saves 20% of memory (table 4) when compared to `LUdivine` and 55% when compared to `LSP`. Actually, the memory usage of the original `LSP` is approximately that of `LUdivine` augmented by the extra matrix storage (which corresponds exactly to that of `LQUP`: e.g. $5000 * 5000 * 8bytes = 200Mb$). This memory reduction is

n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
`LSP`	2.83	8.71	17.85	71.16	160.35	444.17
`LUdivine`	1.60	4.90	10.00	39.99	89.98	249.86
`LQUP`	1.28	3.93	8.01	32.02	72.02	200.04

Table 4: Comparing memory usage (Mega bytes) of `LSP`, `LUdivine`, `LQUP` over Z_{101}, on a P4, 2.4GHz

of high interest when dealing with large matrices (further improvements on the memory management are presented section 4.5).

4.5 Data locality

To solve even bigger problems, say that the matrices do not fit in RAM, one has mainly two solutions: either perform out of core computations or parallelize the resolution. In both cases, the memory requirements of the algorithms to be used will become the main concern. This is because the memory accesses (either on hard disk or remotely via a network) dominate the computational cost. A classical solution is then to improve data locality so as to reduce the

volume of these remote accesses. In such critical situations, one may have to prefer a slower algorithm having a good memory management, rather than the fastest one, but suffering from high memory requirements. We here propose to deal with this concern in the case of rank or determinant computations of large dense matrices. The generalization to the full factorization case being direct but not yet fully implemented.

To improve data locality and reduce the swapping, the idea is to use square recursive blocked data formats [12]. A variation of the `LSP` algorithm, namely the `TURBO` algorithm [8], adapts this idea to the exact case. Alike the `LQUP` algorithm which is based on a recursive splitting of the row dimension (see section 4.3), `TURBO` achieves more data locality by splitting both row *and* column dimensions. Indeed the recursive splitting with only the row dimension tend to produce "very rectangular" blocks: a large column dimension and a small row dimension. On the contrary, `TURBO` preserves the squareness of the original matrix for the first levels. More precisely each recursive level consists in a splitting of the matrix into four static blocks followed by five recursive calls to matrix triangularizations (U, V, C, D, and Z, in that order on figure 5), six `Trsm` and four matrix multiplications for the block updates. In this first

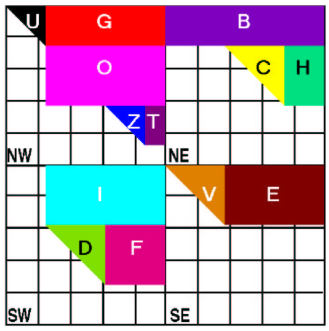

Figure 5: Principle of the TURBO decomposition

implementation, only one recursive step of `TURBO` is used, the five recursive calls being performed by the `LQUP` algorithm. For the actual size of matrices, the quite complex implementation of more recursive levels of `TURBO` is not yet mandatory.

Now for the comparisons of figure 6, we use the full `LQUP` factorization algorithm as a reference. Factorization of matrices of size below 8000 fit in 512Mb of RAM. Then `LQUP` is slightly faster than `TURBO`, implementation of the latter producing slightly more scattered groups. Now, the first field representation chosen (curves 1 and 2) is a modular prime field representation using machine integers. As presented in [7], any matrix multiplication occurring in the decomposition over such a representation is performed by converting the three operands to three extra floating point matrices. This memory overhead is critical in our comparison. `TURBO`, having a better data locality and using square blocks whenever possible, requires smaller temporary matrices than the large and very rectangular blocks used in `LQUP`. Therefore, for matrices of order over 8000, `LQUP` has to swap a lot while `TURBO` remains more in RAM. This is strikingly true for matrices between 8000 and 8500, where `TURBO` manages to keep its top speed.

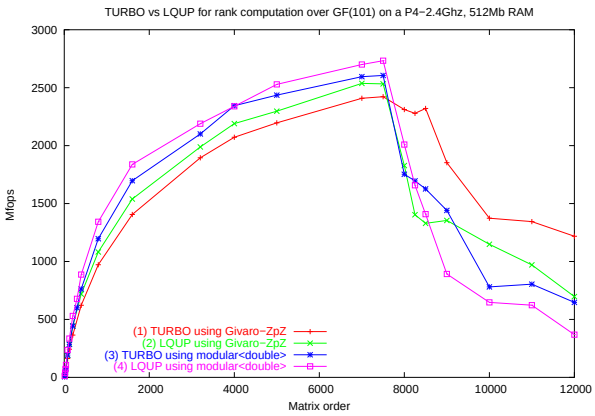

Figure 6: TURBO versus LQUP for out of core rank

To also reduce the memory overhead due to the conversions to floating point numbers, one can use the `modular<double>` field representation, as described in section 2. There absolutely no allocation is done beside the initial matrix storage. On the one hand, performances increase since the conversions and copy are no longer performed, as long as the computations remain in RAM (see curves 3 and 4). On the other hand, the memory complexities of both algorithms now become identical. Furthermore, this fully in-place implementation does not create small block copies anymore. Paradoxically, this prevents the virtual blocks from fitting in the RAM, since they are just a view of the large initial matrix. For this reason, both performance losses appear for matrices of order around 8000. However, the drop is lower for `TURBO` thanks to the recursive blocked data formats producing better data locality.

This behavior of course confirms that as soon as the RAM is full, data locality becomes more important than memory saves : `TURBO` over Givaro-Zpz is the fastest for matrices of size bigger than 8000, despite its bigger memory demand. This is advocating further uses of recursive blocked data formats and of more recursive levels of `TURBO`.

5 Rank, determinant, inverse

The `LQUP` factorization and the `Trsm` routines reduce to matrix multiplication as we have seen in the previous sections. Theoretically, as matrix multiplication requires $2n^3 - n^2$ arithmetic operations, the factorization, requiring at most $\frac{2}{3}n^3$ arithmetic operations, could be computed in about $\frac{1}{3}$ of the time. Now, the matrix multiplication routine `Fgemm` of FFLAS package can compute 5000×5000 matrix multiplications in only 67.58 seconds on a 2.4GHz pentium 4. This is achieved with pipelining within the P4 processor and with very good performances of the BLAS. This corresponds to 3300 millions of finite field arithmetic operations per seconds ! Well, table 5 shows that with $n \times n$ matrices we are not very far from these quasi-optimal performances also for the factorization:

Moreover, from the two routines, one can also easily derive several other algorithms:

n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
LQUP	0.05s	0.18s	0.46s	2.80s	7.79s	32.9s
FGEMM	0.04s	0.23s	0.62s	4.28s	14.72s	67.58s
Ratio	1.25	0.78	0.74	0.65	0.53	0.48

Table 5: Comparing Matrix Multiplication and Factorization over Z_{101}, on a P4, 2.4GHz

- The **rank** is the number of non-zero rows in U.
- The **determinant** is the product of the diagonal elements of U (stopping whenever a zero is encountered).
- The **inverse** is also straightforward:

Algorithm `Inverse`(A)
Input: $A \in {\mathbb{Z}_p}^{m \times m}$, non singular.
Output: $A^{-1} \in {\mathbb{Z}_p}^{m \times m}$.
Scheme

$$\begin{array}{rcl} L, U, P & := & \texttt{LQUP}(A). \quad \textit{(A is invertible, so Q is Id)} \\ X & := & \texttt{LLeft-Trsm}(L, Id). \\ A^{-1} & := & P^T \ \texttt{ULeft-Trsm}(U, X). \end{array}$$

Now, the inverse can then be computed with at most $\frac{2}{3}n^3 + 2n^3$ arithmetic operations which gives a theoretical ratio of $\frac{4}{3}$. Once again, table 6 proves that our implementation has pretty good performances: Indeed, operations performed in

n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
INV	0.18s	0.70s	1.79s	10.84s	32.33s	139.5s
FGEMM	0.04s	0.23s	0.62s	4.28s	14.72s	67.58s
Ratio	4.50	3.04	2.89	2.53	2.20	2.07

Table 6: Comparing Matrix Multiplication and Inverse over Z_{101}, on a P4, 2.4GHz

`LQUP`, or the `Trsm` are not grouped as well as in `Fgemm`. Therefore, the excellent performances of `Fgemm` make the ratio somewhat unreachable, although the `invert` routine is very fast. Note that, as the first `LLeft-Trsm` call is made on the identity it could be accelerated in a specific routine. Indeed, during the course of LUP, L^{-1} can actually be computed with only a $\frac{n^3}{3}$ overhead, thus reducing the theoretical ratio from 4/3 to 1.

6 Conclusions

We have achieved the goal of approaching the speed of the numerical factorization of any shape and any rank matrices, but for finite fields. For example, the `LQUP` factorization of a 3000×3000 matrix over a finite field takes 8.5 seconds where 6 seconds are needed for the numerical `LUP` factorization of `lapack`[5]. To reach these performances one could use blocks that fit the cache dimensions of a specific machine. In [7] we proved that this was not mandatory for matrix multiplication. We think we prove here that this is not mandatory for any dense

[5] www.netlib.org/lapack

linear algebra routine. By the use of recursive algorithms and efficient numerical BLAS, one can approach the numerical performances. Moreover, long range efficiency and portability are warranted as opposed to every day tuning with at most 10% loss for large matrices (see table 2 where delayed can beat BLAS only for big primes and with a specific empirical threshold).

Besides, the exact equivalent of stability constraints for numerical computations is coefficient growth. Therefore, whenever possible, we computed and improved theoretical bounds on this growth (see bounds 3.3 and [7, Theorem 3.1]). Those optimal bounds enable further uses of the BLAS routines.

Further developments include:

- A Self-adapting Software [4] (to switch to different algorithms during the recursive course of `Trsm` and `TURBO`), could be used to find the best empirical thresholds.
- The other case where our wrapping of BLAS is insufficient is for very small matrices (see tables 1 and 2). Here also, automated tuning would produce improved versions.
- The extension of the factorization to some other algorithms as shown for the Inverse (e.g. null-space computation as in [21]) is in progress.
- Finally, extending the out of core work of section 4.5 to design a parallel library is promising.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Dario Bini and Victor Pan. *Polynomial and Matrix Computations, Volume 1: Fundamental Algorithms*. Birkhauser, Boston, 1994.

[3] Morgan Brassel, Pascal Giorgi, and Clement Pernet. LUdivine: A symbolic block LU factorisation for matrices over finite fields using blas. In *East Coast Computer Algebra Day, Clemson, South Carolina, USA*, April 2003. Poster.

[4] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. *Lecture Notes in Computer Science*, 2660:759–770, January 2003.

[5] Jean-Guillaume Dumas. Efficient dot product over word-size finite fields. Rapport de recherche, IMAG, December 2003.

[6] Jean-Guillaume Dumas, Thierry Gautier, Mark Giesbrecht, Pascal Giorgi, Bradford Hovinen, Erich Kaltofen, B. David Saunders, Will J. Turner, and Gilles Villard. LinBox: A generic library for exact linear algebra. In Arjeh M. Cohen, Xiao-Shan Gao, and Nobuki Takayama, editors, *Proceedings of the 2002 International Congress of Mathematical Software, Beijing, China*, pages 40–50. World Scientific Pub, August 2002.

[7] Jean-Guillaume Dumas, Thierry Gautier, and Clément Pernet. Finite field linear algebra subroutines. In Teo Mora, editor, *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, Lille, France*, pages 63–74. ACM Press, New York, July 2002.

[8] Jean-Guillaume Dumas and Jean-Louis Roch. On parallel block algorithms for exact triangularizations. *Parallel Computing*, 28(11):1531–1548, November 2002.

[9] Jean-Guillaume Dumas and Gilles Villard. Computing the rank of sparse matrices over finite fields. In Victor G. Ganzha, Ernst W. Mayr, and Evgenii V.

Vorozhtsov, editors, *Proceedings of the fifth International Workshop on Computer Algebra in Scientific Computing, Yalta, Ukraine*, pages 47–62. Technische Universität München, Germany, September 2002.

[10] Pascal Giorgi. From BLAS routines to finite field exact linear algebra solutions, July 2003. ACA'2003, 9th International Conference on Applications of Computer Algebra, Raleigh, North Carolina State University, USA.

[11] Pascal Giorgi, Claude-Pierre Jeannerod, and Gilles Villard. On the complexity of polynomial matrix computations. In Sendra [20], pages 135–142.

[12] F. Gustavson, A. Henriksson, I. Jonsson, and B. Kaagstroem. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. *Lecture Notes in Computer Science*, 1541:195–206, 1998.

[13] Bradford Hovinen, 2002. Personal communication.

[14] Xiaohan Huang and Victor Y. Pan. Fast rectangular matrix multiplications and improving parallel matrix computations. In ACM, editor, *PASCO '97. Proceedings of the second international symposium on parallel symbolic computation, July 20–22, 1997, Maui, HI*, pages 11–23, New York, NY 10036, USA, 1997. ACM Press.

[15] Oscar H. Ibarra, Shlomo Moran, and Roger Hui. A generalization of the fast LUP matrix decomposition algorithm and applications. *Journal of Algorithms*, 3(1):45–56, March 1982.

[16] Erich Kaltofen, Mukkai S. Krishnamoorthy, and B. David Saunders. Parallel algorithms for matrix normal forms. *Linear Algebra and its Applications*, 136:189–208, 1990.

[17] Clément Pernet. Implementation of Winograd's matrix multiplication over finite fields using ATLAS level 3 BLAS. Technical report, Laboratoire Informatique et Distribution, July 2001. `www-id.imag.fr/Apache/RR/RR011122FFLAS.ps.gz`.

[18] Clément Pernet. Calcul du polynôme caractéristique sur des corps finis. Master's thesis, University of Delaware, June 2003.

[19] Clement Pernet and Zhendong Wan. LU based algorithms for the characteristic polynomial over a finite field. In Sendra [20]. Poster.

[20] Rafael Sendra, editor. *ISSAC'2003. Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation, Philadelphia, Pennsylvania, USA*. ACM Press, New York, August 2003.

[21] Arne Storjohann. Effective reductions to matrix multiplication, July 2003. ACA'2003, 9th International Conference on Applications of Computer Algebra, Raleigh, North Carolina State University, USA.

[22] Will J. Turner. *Blackbox linear algebra with the LinBox library*. PhD thesis, North Carolina State University, May 2002.

[23] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, January 2001. `www.elsevier.nl/gej-ng/10/35/21/47/25/23/article.pdf`.

Appendix

A Proof of theorem 3.2

THEOREM 3.2 Let $T \in \mathbb{Z}^{n\times n}$ be a unit diagonal upper triangular matrix, and $b \in \mathbb{Z}^n$, with $|T| \leq p-1$ and $|b| \leq p-1$. Let $X = (x_i)_{i\in[1..n]} \in \mathbb{Z}^n$ be the solution of $T.X = b$ over the integers.

Then

$$\forall\ k \in [0..n-1] \begin{cases} -u_k \leq x_{n-k} \leq v_k & \text{if } k \text{ is even} \\ -v_k \leq x_{n-k} \leq u_k & \text{if } k \text{ is odd} \end{cases} \tag{4}$$

where

$$\begin{cases} u_n = \frac{p-1}{2}\left[p^n - (p-2)^n\right] \\ v_n = \frac{p-1}{2}\left[p^n + (p-2)^n\right] \end{cases}$$

Proof. Let us define the induction hypothesis IH_l to be that the equations (4) are true for $k \in [0..l-1]$. When $l = 0$, $x_n = b_n$ which implies that $-u_0 = 0 \leq x_n \leq p-1 = v_0$. Thus IH_0 is proven. Let us suppose that $\forall j \in [0..l]$ IH_j is true, and prove IH_{l+1}. There are two cases: either l is odd or not !

If l is odd, $l+1$ is even. Now, by induction, an upper bound for x_{n-l-1} is

$$\begin{aligned}
& (p-1)\left(1 + \sum_{i=0}^{\frac{l-1}{2}} u_{2i} + v_{2i+1}\right) \\
\leq\ & (p-1)\left(1 + \sum_{i=0}^{\frac{l-1}{2}} \frac{p-1}{2}\left[p^{2i} - (p-2)^{2i} + p^{2i+1} + (p-2)^{2i+1}\right]\right) \\
\leq\ & (p-1)\left(1 + \sum_{i=0}^{\frac{l-1}{2}} \frac{p-1}{2}\left[p^{2i}(p+1) + (p-2)^{2i}(p-3)\right]\right) \\
\leq\ & (p-1)\left(1 + \frac{p-1}{2}\left[(p+1)\frac{p^{l+1}-1}{p^2-1} + (p-3)\frac{(p-2)^{l+1}-1}{(p-2)^2-1}\right]\right) \\
\leq\ & \frac{p-1}{2}\left[p^{l+1} + (p-2)^{l+1}\right] \\
\leq\ & v_{l+1}
\end{aligned}$$

Similarly, a lower bound for x_{n-l-1} is

$$\begin{aligned}
& -(p-1)\sum_{i=0}^{\frac{l-1}{2}} v_{2i} + u_{2i+1} \\
\geq\ & -\frac{(p-1)^2}{2}\sum_{i=0}^{\frac{l-1}{2}}\left[p^{2i} + (p-2)^{2i} + p^{2i+1} - (p-2)^{2i+1}\right] \\
\geq\ & -\frac{(p-1)^2}{2}\sum_{i=0}^{\frac{l-1}{2}}\left[p^{2i}(p+1) - (p-2)^{2i}(p-3)\right] \\
\geq\ & -\frac{p-1}{2}\left[p^{l+1} - (p-2)^{l+1}\right] \\
\geq\ & -u_{l+1}
\end{aligned}$$

Finally, If l is even, a similar proof leads to $-v_{l+1} \leq x_{n-l+1} \leq u_{l+1}$