Sparse Matrix Computations for Linear Algebra

Calvin Roth

I. Introduction

In this paper, I will discuss the factoring problem with heavy emphasis on the linear algebra that is employed to solve it. The problem of how to efficiently find the factorization of a large number is one with ancient methods as well as continued interest to mathematicians and computer scientists in the modern.

The primary interest to factoring is the RSA encryption system. RSA is a widely used public-key encryption scheme with its security resting on the assumption that factoring a large integer is hard, meaning takes exponential time in the number of digits of N. Specifically, RSA uses a large N which is the product of two primes. These primes should be approximately the same size with the loose conceptual worry that if one prime is allowed to be too small then it could be easy to guess a factor. In the extreme case where N might be divisible by a small prime like 2 or 3, one would always start with trial division up to some fixed value. If we are developing a general factoring algorithm not based on RSA we still might want to do some trial division at the start. The RSA problem generates an effectively endless stream of harder problems by just ramping up the size of the number to be factored.

The preferred way to solve this factoring problem is to generate a related matrix and find an element of its nullspace. This matrix tends to be large and very sparse. For instance, in the current record of solving a 250 digit RSA number, the team had a matrix with 400 million rows and about 250 non-zero elements per row. We call the algorithms in this broad scheme by how they generate the matrix, and the one that performs the best is called the Number Field Sieve.

Several papers do a good job of outlining the essence of one of the two main linear algebra algorithms, the block Wiedemann algorithm or the Block Lanczos, but not the other. In addition, they spent very few words on matrix generating step and just focus on the linear algebra. In contrast, other papers and books that discuss the subject tend to be more interested in the number theory and abstract algebra of the Number Field Sieve but hand wave the linear algebra. Here we will bridge this gap and discuss both to sufficient but not exhaustive degree.

Both linear algebra methods are efficient at solving linear systems of sparse finite fields in terms of time and space. For both, we use the matrix for matrix vector products almost entirely. Currently, among groups that aim break a record for biggest RSA number factored the Wiedemann algorithm is preferred despite the fact that Lanczos requires fewer steps in serial. The most expensive operation, the total number of matrix vector products, is $3N/64$ for the Weidemann algorithm and $2N/64$ for Lanczos algorithm. The Weidemann algo-

rithm is preferred because it is better in parallel then Lanczos.

II. Notation and Background

In general p will represent a prime number, N will represent some large odd integer we'd eventually like to factor. $\pi(X)$ will denote the number of primes less than X. $\mathbb{Z}_n, \mathbb{Z}/n\mathbb{Z}$ both mean the integers mod N and $GF(2), \mathbb{Z}_2, \mathbb{F}_2$ all mean the field of two elements, 0 and 1.

$(\cdot, \cdot)$ in this paper will always mean the inner product of two vectors. This has to be noted because in much of the number theory and cryptography literature $(\cdot, \cdot)$ is frequently used as a stand in for the gcd of two integers. Uppercase letters(except N) will represent matrices, lowercase letters are vectors, bold lowercase letters will indicate blocks of vectors. Since I will be using N for the integer we are factoring R will be the de facto number of rows of matrix.

We have due to Gauss a satisfactory equation for the expected number of primes from 2 to X inclusive. This formula is $\pi(X) \approx /ln(x)$. We say a polynomial is irreducible over a domain D when it can not be factored into two smaller degree polynomials that are not constants. For example, $x^2 - 2$ is irreducible over $\mathbb{Q}$ but not $\mathbb{R}$.

An integer is B-smooth is all of the numbers in it's prime factorization are less than or equal to B. For example, 15 is 5-smooth while 14 is not. We will be using B-smooth numbers often and there are a number of advantages to using them. The first is that if we know every number in our set is B-smooth then we can use the exponent vector notation of a number in place of the number itself. If we wrote $x = p_1^{k_1} p_2^{k_2} \cdots p_\ell^{k_\ell}$ then the exponent vector of x is $[k_1, k_2, \cdots, k_\ell]^T$. Since we will collect numbers with the same smoothness bound all of the vectors will have the same length. Adding two vectors corresponds to multiplying two numbers. Aside from this nice structure, B-smooth numbers are common and easy to recognize in batches. For quick recognition we can use an amortized approach built off of the Sieve of Erastoness to find B-smooth numbers with a just of $O(lg(lg(B)))$. As for the expected number of B-smooth numbers we use a result given by Erdos, Pommerance, Canfield that showed the number of numbers below X that are $X^{1/u}$ smooth is approximately Xu^{-u}[1]

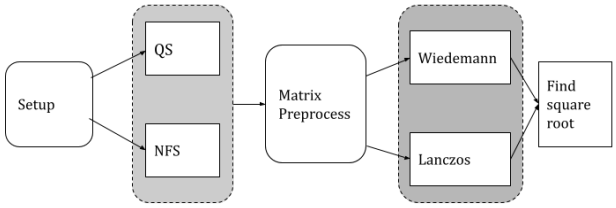

Fig. 1. General pipeline of steps in factoring matrix. The first step, preprocessing, typically involves trial dividing a small number of primes and testing that the number is not a power of a prime. Next, we generate a matrix using either the quadratic sieve or the number field sieve and do any optional matrix preprocessing to improve perform. We then apply either of two linear algebra algorithms to find a null vector of the matrix and finally using that vector we find the X and Y described in this section. The grayed boxes indicate the computationally intensive tasks.

III. Problem formulation

The standard factoring algorithms for factoring large integers is composed of two expensive steps. Step one is generate a set of vectors and step two is to find non-zero linear combination of them that sums to the zero vector. Note that since we will be working in $\mathbb{Z}_2$ every coefficient of our linear combination is 0 or 1. There are two well studied classical methods to generate the matrix, the quadratic sieve and the number field

sieve. The choice of matrix generating technique and linear algebra are independent, and their development happened somewhat in parallel. As the Number Field Sieve was being developed in the late 80s, the linear algebra algorithms discussed in this paper were being discussed but with an eye at the quadratic sieve and then seamlessly transitioned to be aimed at the number field sieve and the matrices it forms.

The goal finding a factor of a number is turned into the finding a nontrivial of square root of 1 mod N that is find $x \neq \pm 1 \mod N$ such that $x^2 = 1 \mod N$. It is worth pointing out that this surely exists if N is an odd composite from an elementary number theory consideration. If N has k unique prime factors then $\mathbb{Z}_N \cong \mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \cdots \times \mathbb{Z}_{p_k}$ by the Chinese remainder theorem. This means we can write an element of $x \in \mathbb{Z}_N$ as $(e_1, e_2, \cdots, e_k)$ where $e_i \equiv x \mod p_i$. A root is a number where every $e_i = \pm 1$ but N is only equivalent to ± 1 if every e_i is all 1 or all -1. In general, if a number has k unique prime factors, none of which are 2, then there are 2^k roots of 1 mod N and only 2 are exactly 1 and -1.

Example Suppose that $N = 3 * 5 * 7 = 105$ then one nontrivial root is 29, $29^2 \mod 105 \equiv 1$ and has an expansion via the Chinese remainder theorem, $29 \equiv -1 \mod 3, 29 \equiv -1 \mod 5, 29 \equiv 1 \mod 7$. For reference, the other nontrivial quadratic roots mod 105 are $34, 41, 64, 71$ and 76. We will see why immediately following this but $gcd(29-1, 105) = 7$ which is a factor of N.

If we have found such an root x then $gcd(x-1, N) \neq 1$ that is we have found a nontrivial factor of N. The following shows why:

$$\begin{aligned} x^2 &\equiv 1 \mod N \\ (x+1)(x-1) &\equiv 0 \mod N \\ (x-1) &\not\equiv 0 \mod N \\ (x+1) &\not\equiv 0 \mod N \\ (x-1)|N &\text{ Since } x+1 \neq kN \end{aligned}$$

But if $x - 1 = \{-1, 1\}$ then $kN/(x-1) = \pm kN = (x+1)$ so we should have a contradiction because we assumed that $x \neq -1 \mod N$. This is all to say the $(x-1)$ divides part of N(likwise for x+1) so $gcd(x-1, N) > 1$ so we have found a factor of N. This is a general framework that the leading factoring algorithms including the quantum factoring algorithm, Shors algorithm, all use. Shors algorithm uses quantum mechanics to find such an x but in the classical we make use of number theory and linear algebra.

To generate our matrix we will actually first make a larger pool of possible candidates S and hope to find a subset S' of S such that

$$\prod_{x \in S} (x^2 \mod N)$$

is square in the usual integers such as 4, 9, or 16. If we could do so then let

$$X = \prod_{x \in S} x$$

and

$$Y = \sqrt{\prod_{x \in S} (x^2 \mod N)}$$

. The two methods differ here in how they generate this pool of possible candidates but their are some core

essential similarities. Both as their names suggest use a sieving strategy to filter a pool of samples for possible candidates with a very low amortized cost per element. This is similar in spirit to classical Sieve of Eratosthenes for finding the factorizations of a set of consecutive numbers. Also both accept as candidates elements that can be fully described with a fixed length "exponent vector". The idea of an exponent vector is to turn questions but multiplying numbers into questions about adding vectors. The naming comes from the realization that if we know the prime factorization of $x = p_1^{k_1} p_2^{k_2} \cdots p_\ell^{k_\ell}$ and $y = p_1^{j_1} p_2^{j_2} \cdots p_\ell^{j_\ell}$ then $xy = p_1^{k_1+j_1} p_2^{k_2+j_2} \cdots p_\ell^{k_\ell+j_\ell}$. So we set a maximum vector length where the candidates are those elements that can be described by an exponent vector of that length. That is the candidates are those that are B-smooth.

But now we are ready to answer when is the product of numbers a square. The product is a square if when we add the exponent vectors the resulting vector is even in every component. The fact that we only care about the parity of each component leads us to work over $\mathbb{F}_2$. If each candidate's exponent vector is written as a column of a matrix A then we want to find a vector v such that $Av = 0$ but $v \neq 0$ or in linear algebra terms find an element of the nullspace of A but all are operations are $\mod 2$ and the only elements that any component is allowed to be is 0 or 1.

A. Quadratic Sieve

The quadratic sieve was the first algorithm developed to use this pattern. In the quadratic sieve, we generate the terms of the polynomial $x^2 \mod N$ starting at $x = \lceil \sqrt{N} \rceil$ and incrementing by 1 each step. If we have collected a large number of these terms then we can sieve for B-smooth values to represent the columns of our matrix. We start at $\lceil \sqrt{N} \rceil$ because that will lead to small initial residues when we compute $x^2 \mod N$ which will make it more likely any given sample is B-smooth and typically will have less factors which will generate a sparser matrix. If we store all the prime factorizations of this samples finding the square root of the choosen subset is easy. We have $Av = 0 \mod 2$ but $Av = w$ for some vector w. Then the exponent vector of the answer is just $w/2$.

We set the smoothness bound based on the input N. There is a balance between setting it too large and therefore needing to find many candidates and setting it too small where the probability that a given sample is B-smooth will be low. Using existing estimates for the number of B-Smooth numbers in an interval, we can optimize for the best choice of B. After this work we derive that the expected cost is $O(B^2) = O(exp(\frac{1}{2}\sqrt{ln(N) * ln(ln(N))}))$[2]. This is algorithm will take subexponential time. But by leveraging more number theory we can bring down the cost of this step.

B. The number Field Sieve

The number field sieve(NFS)[3] is the priemere classical factoring method but also more complicated than the quadratic sieve. Instead of looking at integers of the polynomial $x^2 \mod N$ we will instead be looking at polynomials.

Let's start by defining the titular number field of the number field sieve. An extension of $\mathbb{Q}$ denoted by $\mathbb{Q}(\alpha)$ is the set of elements of $\mathbb{Q}$ appended with α expanded to have closure under the operations of standard operations

of addition, subtraction multiplication, and division. For example, for the set $\{a+b\sqrt[3]{3}|a,b\in\mathbb{Q}\}$ to be closed we must have $\mathbb{Q}(\sqrt{3})=\{a+b\sqrt[3]{3}+c\sqrt[3]{3}^2|a,b,c\in\mathbb{Q}\}$. We call the degree the number of terms needed expression any element of $\mathbb{Q}(\alpha)$ as polynomial of α with coefficients in $\mathbb{Q}$. For our $\mathbb{Q}(\sqrt[3]{3})$ the degree is 3. The degree need not be finite, for instance $\mathbb{Q}(\pi)$ does not have finite degree. A number field is a finite degree extension of $\mathbb{Q}$. We can also use the same definition for extension of $\mathbb{Z}$ for example $\mathbb{Z}(i)=\{a+bi|a,b\in\mathbb{Z}\}$ are known as the Gaussian integers.

The idea is to first calculate a related polynomial f(x) with degree d with integer coefficients. If α is a solution of f then the number field we will be concerned with is $\mathbb{Z}(\alpha)=\{a+b\alpha+c\alpha^2+\cdots+z\alpha^{d-1}|a..z\in\mathbb{Q}\}$. This polynomial will relate back to N via an integer with the property that $f(m)\equiv 0 \mod N$. Since will be be designing f this isn't a hard constraint to deal with. We will also define the nature map ϕ from an element in $\mathbb{Z}(\alpha)$ using f to $\mathbb{Z}$ via $h(a+b\alpha+\cdots+z\alpha^{d-1})=f(a+bm+\cdots+zm^{d-1}$. There is one more caveat where we want f to be irreducible. For the actual sieving and linear algebra. But if it is reducible as $f=g*h$ then $g(m)\times h(m)$ will be a factorization of N. We will note dicuss it here but factoring polynomials is relatively easy compared to factoring integers especially because f grows very slowly in terms on N[2] [4].

Our goal is to find a set of elements in number field such that their product is a square in the number field but also that the the product of each of these elements mapped to $\mathbb{Z}_N$ is a square mod N. If we call these elements $s_1,s_2,\cdots,s_k$ then what we are asking for is

$$u^2=\phi(\gamma)^2=\phi(\gamma^2)=\phi(\prod_{s_i}s_i)=\prod_{s_i}\phi(s_i)=v^2$$

. What we actually want is the u and v but we actually find γ and v by determining which s_i to include in our product.

The last essential piece to understand of the NFS is what exactly are our samples and what are their exponent vectors. Since our elements are polynomials in α which are harder to analyze then plain integers we pick the simpliest case, all of our polynomials will be of the form $a-b\alpha$ where a and b are co-prime integers. Instead, we will use the function $F(x,y)=x^d+c_{d-1}x^{d-1}y+c_{d-2}x^{d-2}y^2+\cdots+c_1xy^{d-1}+c_0y^d$. We generate samples by fixing either a or b in $a-\alpha b$ and finding (a,b) such that $F(a,b,)$ is B-smooth. There remain long number theoretic proofs that show this will work that will not be covered here[2].

Skipping past these difficulties, all these complications pay off and we get a run time for the matrix generating step of $O(exp((\frac{64}{9}^{1/3}+o(1))ln^{1/3}(N)(ln(ln(N)))^{2/3}))$. As a final note, the quadratic sieve and the number field sieve have running times that look similar, both have the form $exp(kln(n)^{\alpha}(ln(ln(n)))^{1-\alpha})$. This form has been well observed and in the cryptography community, a shorthand is used to write this runtimes called L-notation, $L_n[\alpha,k]=exp(kln^{\alpha}nln(ln(n))^{1-\alpha})$, and is commonly used for factoring and discrete logarithm problems[5].

In both of these setups, the matrices we generate are extremely sparse. Typically, for the size of problems that have recently been solved there are typically on order of

200-400 non-zero elements per column but a number of rows in the hundreds of millions. This sparsity is in no sense randomly distributed either. Since the ith element of a column represents the power the ith prime is taken to then we should expect the matrix to be signifigantly more dense on top than bottom for the basic reason that a random number is more likely to be divisible by same primes like 3,5, and 7 compared to large primes.

IV. Linear Algebra set-up

Blocking A nature question about first seeing that we will be working with matrices where every element is 0 and 1 is how should we best store the matrix. The standard practice is to block slices of 32 or 64 columns together depending on the machine word size and treat each row as an int or long. The operations of addition and multiplication mod 2 are easily carried over the bitwise operations over ints and longs. To add two binary vectors x and y of 32 elements, we are simply taking the component-wise XOR of x and y and the component-wise multiplication of x and y is the component-wise AND. So we just use bitwise xor(ˆ) and bitwise and(&) to carry out the operations we will need over blocks of vectors. Although simple this is a quite impactful change as it will reduce the number of operations we have to do by a factor of 32 or 64.

Typically, directly finding a null vector of a large matrix is not a nice constraint to work with. So instead, the problem is written to find $By = z$ where the solution y and the matrix b can be connected back solving $Ax = 0$ where x is not 0. What we do is split A into the 64 rightmost columns, **a'** , and the rest A'. We try and find solutions to A'**x'** = **a'** without constraint and then we can set $A[x'| - I]^T = Ax' - a' = a' - a' = 0$. The final piece of the puzzle is that we will want the matrix we are working with to be symmetric as we had in the standard lanczos algorithm. To fix this we solve $AA^Tv = a'$ and $x' = A^Tv$. We should not ever directly use AA^T because it is likely to be dense so instead we need to compute AA^Tv will compute $v' = A^Tv$ and then Av'.

V. Lanczos type

The lanczos algorithm is one of the most popular algorithms for solving sparse linear systems so it perhaps isn't surprising that it is possible to adapt it to this domain.

A. The standard Lanczos

The application of the Lanczos algorithm to Real or Complex linear systems is well documented. In the standard lanczos algorithm we developed sequences v and w with the relation

$$w_0 = b$$
$$v_{n+1} = Aw_n$$
$$w_{n+1} = v_{n+1} - \alpha_n w_n - \beta_{n-1} w_{n-1}$$

Where α_n and β_{n-1} are set to make $w_i^T w_j = 0$ when $i \neq j$. We do so by setting $\alpha_n = \frac{w_n v_{n+1}}{(w_n^T w_n)^{-1}}$ and $\beta = \frac{w_{n-1}^T v_{n+1}}{w_{n-1}^T w_{n-1}}$.

The piece that breaks utterly breaks down if we tried blindly to apply this algorithm is our requirement for orthogonality specifically the part where we don't want $w_i^T w_i = 0$. In a finite field the chance that two random vectors are orthogonal is high. Specifically, for a finite field under prime p $(u, v)_{\mathbb{F}_p} = (u, v)_{\mathbb{R}} mod p$ i.e. there is

a $\frac{1}{p}$ chance any two random vectors are perpendicular. In our case, we want to work with p=2 so about 50% of time two randomly choosen vectors are perpendicular to each other. This includes vectors being self-orthogonal which would have whenever w has an even number of non-zero components. Back in the reals, breakdown where w_i^T is 0 or very small as already been studied[6] where they use information beyond just the last iteration of data.

B. Preprocessing

When the matrices that are generated by the previous steps are examined it is frequently observed that sometimes there is a row with exactly one non-zero element say in position i. Practically, we know that the ith component of the solution vector must be 0 because if not there would be now way to eliminate that element so we might as well remove that column. But importantly, the following Lanczos algorithm requires that this situation doesn't happen so as a preprocessing step we remove any columns that contain the single item of a row. Without doing so we may have a situation that was described in the original lookahead Lanczos algorithm where all the principal minors are singular which they called the incurable case.

C. Block Lanczos for finite fields

For the block Lanczos algorithm, will maintain more blocks than we did for the standard Lanczos but takes a similar pattern. The similar are that we compute a block $\mathbf{v}_{n+1} = A\mathbf{w}_n$ and we still compute α and β in nearly the same way as before but now we introduce a new sequence of vectors $\mathbf{x}_n$ in place of w and have $\alpha = \frac{\mathbf{x}_n^T\mathbf{v}_{n+1}}{\mathbf{x}_n^T\mathbf{x}_n}, \beta = \frac{\mathbf{x}_{n-1}^T\mathbf{v}_{n+1}}{\mathbf{x}_n^T\mathbf{x}_n}$. This new $\mathbf{x}_n$ is used to keep dot products from being 0. The final update corresponding to our initial iteration is $\mathbf{w}_{n+1} = \mathbf{v}_{n+1} - \alpha_n\mathbf{x}_n - \beta_{n-1}\mathbf{x}_{n-1} - d_{n-2}\boldsymbol{\gamma}_{n-2}$ where d is a new scalar and γ is a another new vector[7]. These variables take a bit too long to fully describe how we obtain them, in short the are found as the product of small matrices. In particular, to find $\mathbf{x}_n$ is at worst 128(using 64 bit word size) and is formed from the product of a $(64+k)\times(64+k)$ and $k\times k$ matrix.

The cost of the block-Lanczos algorithm is dominated by the $R/64$ sparse-matrix vector products as well as $5R/64$ inner products, and $10R/64$ scalar-vector products.

VI. Wiedemann type

There exists another strategy to attack the problem of finding a null vector of a finite field matrix. This method called the Wiedemann algorithm, comes from the need effectienly to solve linear algebra problems that arise in error correcting codes especially in the study of BCH codes. As we will learn this is the preferred method used in practice.

A. General framework

To solve $Ax = b$ the wiedemann algorithm attempts to find a polynomial p such that $p(A_K)b = 0$ where $A_s = \{b, Ab, \cdots A^ib\}$. The minimal polynomial of A_K will surely work but it isn't the only i solution. We can hope to find such a polynomial with a random process. Instead of directly finding the minimal polynomial of A_K we will find the minimum polynomial that generates a related sequence. By generate, I mean the coefficients should make a recurrence relation for the data that is a generating sequence of length L of the data $a^{(i)}$ is

$a_i = c_{i-L}a_{i-L-1} + c_{i-L+1}c_{i-L} + \cdots + c_i a_{i-1}$. The coefficient of this sequence will be the coefficients of our polynomial. Our sequence will be $a^{(i)} = (A^i b, u)$ for $i = 0, 1, \cdots 2R-1$. There is an algorithm derived from for finding smallest generating polynomial of binary codes in the study of BCH error correcting codes. This algorithm called the Berlekamp-Massey algorithm takes $O(R^2)$ time[8]. This polynomial will divide the minimum polynomial of A_S. Our hope is polynomial will also solve $p'(A)b = 0$. If we had the minimum polynomial f of A_S then we code do the following to solve $Ax = b$. Note that we normalize f so that the trailing coefficient is 1.

$$f(A)b = c_\ell A^\ell b + c_{\ell-1}A^{\ell-1} + \cdots + c_1 Ab + b$$
$$A(c_\ell A^{\ell-1}b + c_{\ell-1}A^{\ell-2} + \cdots + c_1 b) = -b$$
$$x = -(c_\ell A^{\ell-1}b + c_{\ell-1}A^{\ell-2} + \cdots + c_1 b)$$

The simplest thing to do if p' fails this condition is to just restart and pick a new y and generate a new sequence and polynomial. It was shown in the original paper that the expected number of restarts is $O(log(R))$ steps. It is nature to think we can do better, we found a polynomial that divides the minimum polynomial and this information can be put to use. In this serial algorithm the first iteration goes the same but when check if this polynomial is a solution we use the "residue" $f^{(1)}(A)b_0$ as b_1 and repeat the process. Each step the number of terms we need to generate for the berlekamp-massey algorithm decreases by the degree of last polynomial. The $f_{k+1}(A) - f_{k+1}(0)$ term is to take out the constant term. With probability $> 70\%$ this algorithm in find a solution in 3 iterations[9].

$b_0 = b$
$k = 0$
$d = 0$
$y_0 = 0$
while Not converged **do**
  u_{k+1} = Random Vector
  $Ts \leftarrow (u_{k+1}, A^i b_k)_{i=0}^{2(R-d)}$
  $f = Generating - Poly(Ts)$
  $y_{k+1} = y_k + A^{-1}(f_{k+1}(A)b_k - f_{k+1}(0)b_k)$
  $b_{k+1} = b_0 + Ay_{k+1}$
  $d_{k+1} = d_k + deg(f_{k+1})$
  Test if $A(-y_k) = 0$
  $k++$
end while

These two original proposals are already efficient in terms of memory and time. During one iteration with the restarting strategy, we compute 2R inner products and 2R Matrix-vector products plus $O(R)$ more matrix-vector products for a total of $3R$ matrix vector products per iteration. Storage is also reasonable. In the process of computing the $(u, A^i b)$ sequence we only need to store the previous $A^{i-1}b$ product and the result. So overall we need $4R$ storage aside from the matrix A: 1R for b, 1R for u, 1R for the previous iterate $A^{i-1}b$, and up to R scalars being the result of each inner product.

The improvements don't stop there though. The easiest improvement is a very top level parallelism. Instead of computing $E[log(R)]$ iterations with the restart strategy we can in parallel compute k independent generating polynomials each with it's own u and then return the LCM of this polynomial. But the more clever idea is to use blocking.

Instead of u and b we will use $\overline{u}, \overline{b}$ to represent blocks of vectors. The horizontal sizes of $\overline{u}, \overline{b}$ do not need to match so we will let $\overline{u} : R \times n$ and $\overline{b} : R \times m$. This process will not work for any matrix. But it has been show that we show use pre-conditioners to ensure this

true. Kaltofen showed that UAW will with probability $>= 1 - \frac{Rank(A)(Rank(A)+1)}{2}$[10] where U is a random upper triangular toeplitz matrix and W is a random lower triangular topelitz matrix both with 1s one the diagonal. That is $U = \begin{pmatrix} 1 & u_2 & u_3 & \cdots & u_R \\ 0 & 1 & u_2 & \cdots & u_{R-1} \\ 0 & 0 & 1 & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix}$ and $W = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ w_2 & 1 & 0 & \cdots & 0 \\ w_3 & w_2 & 1 & \ddots & 0 \\ w_R & w_{R-1} & \cdots & w_1 & 1 \end{pmatrix}$ So now our procedure functions by picking a random $\overline{u}, \overline{y} = A\overline{z}$ for a random z and compute the sequence $a^{(i)} = (u, A^i y)$ for $i = 0, \frac{R}{n} + \frac{R}{m}$[11]. The number of terms to evaluate here takes the place of our $2R$ iterations we required in for the non-blocking case. Luckily, we can extend the berlekamp-massey algorithm that we used to find the shortest linear recurrence over scalars to sequences of matrices[12].

B. Relation to Toeplitz Matrices

Another way to view this procedure is obtained when we write out the linear system that our generating polynomial most satisfy. Specifically, this is a toeplitz matrix of equations:

$$\begin{pmatrix} a_R & a_{R-1} & \cdots & a_0 \\ a_{R+1} & a_R & \cdots & a_1 \\ \vdots & \cdots & \ddots & \vdots \\ a_{2R-1} & a_{2R-2} & \cdots a_{R-1} & \end{pmatrix} v = 0$$

In the blocking case we this basically the same system but with a block toeplitz matrix. In both cases, we can use the existing theory developed for quickly solving toeplitz matrices[13]. The total runtime of this algorithm will take no more than $(1+\frac{n}{m}+\frac{1}{n})R+\frac{2n^2}{m}+2n+2]$ matrix vector products and $O((m+n)R^2+(1+\frac{m}{n})R^2 log R * log(log(R))$ arithmetic operations used to solve the Toeplitz matrix. Importantly, we can parallelize this with q processors all of which have access to performing matrix vector products with the matrix A. In this case we will now need each processor to do $2R/m+4R/q+O(1)$ matrix vector products, $O(R^2 log R log(log(R)))$ other operations and be able to $O(R)$ store elements.

VII. Developments

Although this problems leads to a lot interesting ideas that pull from many areas of math this area of research is not active anymore. All the critical pieces of theory came out during the 90s and into the early 2000s. The more recent papers that write about the improving the number field sieve all centered around practical improvements such as improving the performance in parallel and distributed systems. I believe there are number of reasons for this lack of development. The fastest matrix generating procedure, the number field sieve, was developed by Pollard in the late 80s and no one has come up with a more sophisticated and faster method. This matters because in practice, the matrix generating step is the most costly step in the process, the linear algebra step is far from the limiting factor of this algorithm. In addition, the problem has the appearance of making significant progress since improvements in hardware over the decades can carry us. Lastly, in some sense the goal has been reached RSA when first thought up imagined factoring to be hard to solve eliminating the possibility of a brute force attack which isn't the case. The progress made on factoring has prompted the

cryptography research to focus more on other problems.

REFERENCES

[1] E. Canfield, P. Erdös, and C. Pomerance, "On a problem of oppenheim concerning "factorisatio numerorum"," *Journal of Number Theory*, vol. 17, no. 1, pp. 1–28, 1983.

[2] R. Crandall and C. B. Pomerance, *Prime numbers: a computational perspective*, vol. 182. Springer Science & Business Media, 2006.

[3] J. P. Buhler, H. W. Lenstra, and C. Pomerance, "Factoring integers with the number field sieve," in *The development of the number field sieve* (A. K. Lenstra and H. W. Lenstra, eds.), (Berlin, Heidelberg), pp. 50–94, Springer Berlin Heidelberg, 1993.

[4] A. K. Lenstra, H. W. Lenstra, and L. Lovász, "Factoring polynomials with rational coefficients," *Mathematische annalen*, vol. 261, no. ARTICLE, pp. 515–534, 1982.

[5] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 2018.

[6] B. N. Parlett, D. R. Taylor, and Z. A. Liu, "A look-ahead lă˘nczos algorithm for unsymmetric matrices," *Mathematics of computation*, vol. 44, no. 169, pp. 105–124, 1985.

[7] D. Coppersmith, "Solving linear equations over gf (2): block lanczos algorithm," *Linear algebra and its applications*, vol. 192, pp. 33–60, 1993.

[8] J. Massey, "Shift-register synthesis and bch decoding," *IEEE transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, 1969.

[9] D. Wiedemann, "Solving sparse linear equations over finite fields," *IEEE transactions on information theory*, vol. 32, no. 1, pp. 54–62, 1986.

[10] E. Kaltofen and B. D. Saunders, "On wiedemann's method of solving sparse linear systems," in *International Symposium on Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes*, pp. 29–38, Springer, 1991.

[11] D. Coppersmith, "Solving homogeneous linear equations over gf (2) via block wiedemann algorithm," *Mathematics of Computation*, vol. 62, no. 205, pp. 333–350, 1994.

[12] S. Sakata, "Extension of the berlekamp-massey algorithm to n dimensions," *Information and Computation*, vol. 84, no. 2, pp. 207–239, 1990.

[13] R. H.-F. Chan and X.-Q. Jin, *An introduction to iterative Toeplitz solvers*. SIAM, 2007.