

SMOOTH SUBSUM SEARCH

A HEURISTIC FOR PRACTICAL INTEGER FACTORIZATION

MARKUS HITTMEIR

ABSTRACT. The two currently fastest general-purpose integer factorization algorithms are the Quadratic Sieve and the Number Field Sieve. Both techniques are used to find so-called smooth values of certain polynomials, i.e., values that factor completely over a set of small primes. As the names of the methods suggest, a sieving procedure is used for the task of quickly identifying smooth values among the candidates in a certain range. In the present paper, we present a novel approach based on representing such candidates as sums that are always divisible by several of the primes in the factor base. In a direct comparison with the Quadratic Sieve, the main advantage of our method is that the resulting values are smaller, which increases the likelihood of them being smooth. Using current implementations of the Self-initializing Quadratic Sieve in two Python packages for number-theoretical computations as benchmarks, a Python implementation of our approach runs faster in trials with inputs of 30 to 70 decimal digits. We discuss several promising avenues for further improvements and applications of the technique.

1. INTRODUCTION

Integer factorization is the task of computing the divisors of natural numbers. It is a problem with a long and fascinating history, and it is certainly among the most influential in algorithmic number theory. While there is a variety of algorithms significantly faster than the brute-force search for divisors, it is still an open problem to construct a technique that efficiently factors general numbers with hundreds to thousands of digits. The hardness of this problem is fundamental for the security of widely used cryptographical schemes, most prominently the RSA cryptosystem. Nevertheless, there is no proof for its hardness besides the fact that decades of efforts have failed to construct a more efficient technique.

Quite regularly, there are set new records¹ concerning the factorization of numbers of certain size, mostly due to improved implementations of the best available algorithms and advances in the hardware and computing power. In addition, the bound for the deterministic integer factorization problem has been improved multiple times in recent years ([11], [12], [10], [14]). On the other hand, there has only been little progress in the development of new techniques for practical integer factorization since the invention of the Number Field Sieve ([19]) in the 1990s. One of the earlier algorithms with sub-exponential runtime was by Dixon ([7]) in 1981.

2010 *Mathematics Subject Classification.* 11A51, 11Y05.

SBA Research (SBA-K1) is a COMET Centre within the framework of COMET – Competence Centers for Excellent Technologies Programme and funded by BMK, BMDW, and the federal state of Vienna. The COMET Programme is managed by FFG.

¹A recent record was the factorization of a 250 decimal digit number in February 2020 by Boudot et al., see https://en.wikipedia.org/wiki/RSA_Factoring_Challenge

From today’s perspective, it may be considered as a prototype for several other algorithms. To this group belong the Continued Fraction factorization method (CFRAC) described and implemented by Morrison and Brillhart ([24]) in 1975, the Linear Sieve by Schroepel and the Quadratic Sieve by Pomerance. The latter author analyzed and compared these algorithms in 1982 in [25]. In general, there is an extensive amount of literature on practical integer factorization algorithms. The reader may find information on the mentioned methods and other factorization techniques in the survey [18] and in the monographs [27] and [30]. These sources also discuss a variety of factorization algorithms for numbers N that satisfy certain properties, or that have prime factors of certain shape. For example, Fermat’s factorization method runs particularly fast if N has co-divisors that are very close, while Pollard’s P-1 technique and the Elliptic Curve Method (ECM) work particularly well for numbers with small prime factors. In the present paper, we are interested in so-called general-purpose factorization algorithms, which means that the runtime complexity estimate of the procedure only depends on the size of the input number N . Both the Quadratic Sieve and the Number Field Sieve belong to this group. From a practical point of view, the Quadratic Sieve (and its modifications) is the best general-purpose factorization algorithm for N up to around 100 digits, while the Number Field Sieve is faster for inputs beyond that. While methods such as CFRAC and ECM stay somewhat competitive up to a small input length (see [23]), factorization tools usually use ECM to rule out the existence of relatively small prime factors, but then switch either to the Number Field Sieve or to the Self-initializing Quadratic Sieve (SIQS), which is the fastest modification of the original Quadratic Sieve algorithm.²

Our contribution is the presentation and evaluation of a novel approach that we call *Smooth Subsum Search* (SSS). Just like CFRAC, the Linear Sieve and the Quadratic Sieve, SSS belongs to the group of Dixon-type algorithms. In addition to a detailed explanation of our algorithm and a discussion of its advantages and disadvantages compared to other techniques, we will discuss our current implementation in Python and several experiments on the runtime complexity of SSS. Our results demonstrate that SSS outperforms SIQS on semiprime³ numbers with 30 to 70 decimal digits. While these findings need to be corroborated in experiments with different implementations of SIQS in other programming languages and in other settings, they may indicate that SSS is the fastest available factorization algorithm for general numbers in this range.

The remainder of the paper is structured as follows: In Section 2, we discuss the Quadratic Sieve and its modifications in greater detail, together with an important subroutine of our own algorithm. Section 3 presents the main ideas of SSS. The results of our experiments can be found in Section 4, where SSS is compared to available implementations of the Quadratic Sieve in Python. Section 5 explores three ideas on how to improve the current implementation of SSS, in particular for numbers of larger size. Finally, we summarize and give concluding remarks in Section 6.

²In its standard settings, the tool *Msieve* (<https://sourceforge.net/projects/msieve/>) uses ECM for searching for small prime factors with up to 15 digits. Similarly, for numbers with 31 digits or more, the calculator of Dario Alpern (<https://www.alpertron.com.ar>) switches to SIQS after a quick search for small factors via ECM.

³Throughout the paper, we will use this term for numbers composed of two distinct prime factors. Under certain conditions, such numbers are the most difficult to factorize.

2. RELATED WORK

Let N be the number we want to factorize. We will always assume that N is odd, composite and not a perfect power of another number. The currently fastest general-purpose factorization algorithm is the General Number Field Sieve with a heuristic asymptotic runtime complexity of

$$\exp\left(\left(\sqrt[3]{64/9} + o(1)\right)(\log N)^{1/3}(\log \log N)^{2/3}\right).$$

For comparison, the heuristic asymptotic runtime complexity of the Quadratic Sieve is $\exp\left((1 + o(1))(\log N)^{1/2}(\log \log N)^{1/2}\right)$ ([26]). The Number Field Sieve is significantly more complicated and harder to implement than the Quadratic Sieve. As mentioned, the latter technique is also still faster in practice for factoring numbers up to around 100 digits. We now restrict our attention to the technical details of the Quadratic Sieve, as it is a suitable representative for the other Dixon-type methods. In addition, it will be used as a benchmark for our own algorithm in the experiments in Section 4.

The ultimate goal of the Quadratic Sieve and the other Dixon-type algorithms is to find two numbers X and Y such that $X^2 \equiv Y^2 \pmod{N}$. For such pairs, one can easily show that there is a high chance that $\gcd(X - Y, N)$ gives a nontrivial divisor. However, perfect squares are a rare occurrence. Since there are roughly \sqrt{N} square numbers below N , simply picking random values for X and hoping for $X^2 \pmod{N}$ to be square will take too long. So in order to find such X and Y , the first phase of all these algorithms tries to collect so-called *smooth relations*. Such a relation is a congruence $x^2 \equiv y \pmod{N}$ such that the prime factorization of y only consists of primes up to a certain bound B . In this case, the number y is called *B-smooth* (or just *smooth*). Let p_1, \dots, p_m be the primes up to B . The set $\{p_1, \dots, p_m\}$ of these primes is referred to as *factor base*. The smooth relations may then be written as $x^2 \equiv y = p_1^{e_1} \cdots p_m^{e_m} \pmod{N}$ for certain exponents $e_i \in \mathbb{N}_0$, $i = 1, \dots, m$. The first phase proceeds until enough of these relations are found, which usually means a bit more than the number m of primes in the factor base. As soon as this is the case, the second phase starts. Here, we want to find linear dependencies in the exponent vectors (e_1, \dots, e_m) of the smooth relations, i.e., a subset of these vectors that sum to the 0-vector. We can use well-known methods from linear algebra to identify such subsets and thereby find smooth relations that can be multiplied to obtain a new congruence of the form

$$x_1^2 x_2^2 \cdots x_k^2 \equiv \prod_{i=1}^m p_i^{e_{i,1} + e_{i,2} + \cdots + e_{i,k}} \pmod{N},$$

where $e_{i,1} + e_{i,2} + \cdots + e_{i,k} \equiv 0 \pmod{2}$. As a consequence, there exists a number Y such that $Y^2 = \prod_{i=1}^m p_i^{e_{i,1} + e_{i,2} + \cdots + e_{i,k}}$. Clearly, there also exists X such that $X^2 = x_1^2 x_2^2 \cdots x_k^2$, and we have found what we were looking for. One important parameter of these factorization techniques is the smoothness bound B . If B is very large, then it is easier to find smooth relations. On the other hand, we need to find much more of them for the second phase to work. In addition, the matrix in the linear algebra step will be very large. If B is rather small, then we will most likely not be able to find enough smooth relations.

The main difference between Dixon's method, CFRAC, the Quadratic Sieve and our algorithm SSS is in how the smooth relations are collected in the first phase. Dixon's algorithm chooses random values x and checks if $x^2 \pmod{N}$ is B -smooth.

CFRAC uses the convergents in the continued fraction expansion of \sqrt{N} . While the value of $x^2 \pmod{N}$ in Dixon's method is bounded by N , the main advantage of CFRAC over Dixon is that it generates congruences of the form $x^2 \equiv y \pmod{N}$ with $|y| \leq 2\sqrt{N}$, which increases the chances for y being smooth. In the basic version of the Quadratic Sieve, the values of the polynomial $f(x) = (x + \lceil \sqrt{N} \rceil)^2 - N$ are used as candidates for possibly smooth y . Note that

$$(2.1) \quad y = f(x) = x^2 + 2x\lceil \sqrt{N} \rceil + (\lceil \sqrt{N} \rceil)^2 - N,$$

which for small x gives a bound similar to the one obtained in CFRAC. The main advantage of the Quadratic Sieve compared to CFRAC is in how we check the smoothness of the candidates y . While CFRAC applies trial division or techniques for smoothness detection of batches of numbers (see [1]), the Quadratic Sieve uses *sieving*. The basic idea is as follows: Let again $\{p_1, \dots, p_m\}$ be the factor base \mathcal{F} . Our goal is to find arguments x such that $f(x)$ is divisible by many primes in \mathcal{F} . We first remove all primes p from \mathcal{F} for which Legendre's symbol $(N|p) = -1$, since N is a quadratic non-residue modulo such p and, thus, p cannot divide $f(x)$ for any value of x . Next, we use the well-known Tonelli-Shanks algorithm to compute the solutions of $f(x) \equiv 0 \pmod{p}$ for each remaining prime p in \mathcal{F} . Note that $f(x + kp) \equiv f(x) \pmod{p}$ for every $k \in \mathbb{Z}$. Hence, if $f(x_0)$ is divisible by p , so is $f(x_0 + kp)$ for every $k \in \mathbb{Z}$. We start the sieve by initializing an indexed array of pre-defined length L containing zeros, where each index corresponds to an argument x . For each $p \in \mathcal{F}$ and each x_0 with $f(x_0) \equiv 0 \pmod{p}$, we then raise the entries in the array at the indices $x_0 + kp$ for $k \in \mathbb{Z}$ by $\lceil \log p \rceil$. At the end of the sieving, we check the array for entries that are larger than a certain threshold, as for these it follows that $f(x)$ is divisible by a lot of primes in the factor base and thus likely to be smooth. Since the information about the actual divisors of such $f(x)$ is lost, we still need to apply either trial division or some other factorization algorithm suitable for finding small prime factors. However, the sieving procedure greatly reduces the total number of such applications compared to what is needed in CFRAC.

If we have not found enough smooth relations by applying the sieving procedure above, we may continue by increasing the length L of the sieving array. However, this will lead to the expression (2.1) getting quite large, which decreases the chances of $f(x)$ being smooth. The Multiple Polynomial Quadratic Sieve (MPQS) ([29]), an improvement of the basic version of the quadratic sieve, concerns the use of multiple polynomials $f_{a,b}(x) = (ax + b)^2 - N$ with $a, b \in \mathbb{Z}$. While switching to another polynomial allows the use of a new array and thus solves the problem of increasing values of the smoothness candidates, it is also somewhat expensive to initialize new polynomials and compute the solutions of $f_{a,b}(x) \equiv 0 \pmod{p}$ for all the primes p in the factor base. A remedy for this problem has been found in another improvement, the already mentioned Self-initializing Quadratic Sieve (SIQS) [3]. The idea is to choose a as a product of primes in the factor base, and b such that $b^2 - N = ac$ for some $c \in \mathbb{Z}$. Then $f_{a,b}(x) = a(ax^2 + bx + c)$, so $f_{a,b}(x)$ is smooth if and only if $ax^2 + bx + c$ is smooth. In addition, there are several different possible choices b for each choice of a that may be computed via the Chinese Remainder Theorem. The corresponding solutions to $f_{a,b}(x) \equiv 0 \pmod{p}$ for each prime p are related in a way that allows to change to a new polynomial much more easily. In particular, the expensive inversion of elements modulo the primes p only needs to be done once for every choice of a .

One further improvement that may be applied to all Dixon-type algorithms is the so-called *large prime variant* ([2]). If we have found a relation $x^2 \equiv y \pmod{N}$ where y is partially smooth, meaning the part r of its factorization that is not smooth is comparably small, then we may also save this so-called *partial relation*. As soon as we find two partial relations of the form

$$\begin{aligned} x_1^2 &\equiv y_1 \cdot r \pmod{N} \\ x_2^2 &\equiv y_2 \cdot r \pmod{N} \end{aligned}$$

where y_1 and y_2 are smooth, we may combine them to get a full relation, namely $(r^{-1})^2 x_1^2 x_2^2 \equiv y_1 y_2 \pmod{N}$. This improvement can speed up the search for smooth relations by about a factor of 2.

Finally, let us briefly discuss the smoothness detection algorithm by Bernstein that has already been mentioned in the context of CFRAC. For cases where sieving the values of a polynomial in the described sense is not an option, Bernstein describes another way to find smooth and partially smooth numbers in a given set. As mentioned in the abstract, SSS is not based on sieving. In Section 3, we will use the following procedure ([1, Algorithm 2.1]) instead. Let $\mathcal{F} = \{p_1, \dots, p_m\}$ be the factor base and $\mathcal{C} = \{x_1, \dots, x_n\}$ be a set of positive integers.

smooth_batch(\mathcal{F}, \mathcal{C})

- 1: Compute $z = p_1 \cdots p_m$ using a product tree.
 - 2: Compute $z \pmod{p_1}, \dots, z \pmod{p_m}$ using a remainder tree.
 - 3: For $k = 1, \dots, n$, compute $y_k = (z \pmod{p_k})^{2^e} \pmod{x_k}$ using repeated squaring. Here, e is the smallest nonnegative integer such that $2^{2^e} \geq x_k$.
 - 4: For $k = 1, \dots, n$, print $x_k / \gcd(x_k, y_k)$.
-

The algorithm outputs the part of the factorization of each x_k that is not smooth. For more information on smooth detection as well as on product and remainder trees, see [6, Section 3.3]. We now move on to the presentation of SSS.

3. SMOOTH SUBSUM SEARCH

Let us consider an integer polynomial $f : \mathbb{Z} \rightarrow \mathbb{Z}$ and a factor base \mathcal{F} of primes smaller than some bound B . The task is to find $x \in \mathbb{Z}$ such that $f(x)$ is B -smooth. In Section 2, we have discussed the Quadratic Sieve for solving this problem for the polynomial $f(x) = (x + \lceil \sqrt{N} \rceil)^2 - N$ in order to factor the number N . The algorithm presented in this section solves the same problem, but uses a heuristic search approach instead of sieving.

We start by discussing a suitable representation of the possible solutions x . We will see that this representation already restricts our attention to values x with increased likelihood of $f(x)$ being smooth. We are looking for preferably small values x such that $f(x)$ is divisible by several (powers of) the primes in \mathcal{F} . Let p_1, p_2, \dots, p_m be the odd primes in \mathcal{F} , and assume that $(N|p_i) = 1$ for $i = 1, \dots, m$. Then, the congruence $f(x) \equiv 0 \pmod{p_i}$ has two solutions $N_i = \{s_{i,1}, s_{i,2}\}$. Using the Chinese Remainder Theorem, we may easily represent those x for which $f(x)$ is divisible by specific primes in \mathcal{F} . Namely, with $x_i \in \{0, 1, 2\}$, we represent the candidates x via vectors

$$x \hat{=} (x_1, x_2, \dots, x_m).$$

If $x_i > 0$, we fix the value of $x \pmod{p_i} = s_{i,x_i} \in N_i$ and thereby make sure that $f(x) \equiv 0 \pmod{p_i}$ holds. If $x_i = 0$, we leave the value of $x \pmod{p_i}$ undefined, and it will ultimately be determined by the other values in the vector that we did fix. Of course we would like $f(x)$ to be divisible by as many primes in \mathcal{F} as possible. However, if we run the Chinese Remainder Theorem on vectors (x_1, x_2, \dots, x_m) where most of the x_i are nonzero, then the modulus M , i.e. the product of the primes p_i for which $x \pmod{p_i}$ is fixed, will be very large, and so will be the average values of the resulting x . The chances for $f(x)$ actually being smooth are then rather small. From this observation, it becomes quite clear that we would like to keep the quantity $|x|/M$ as small as possible in order to increase our chances for finding smooth numbers. The approach we currently use is to focus on a subset \mathcal{S} of the smaller primes in \mathcal{F} and, for generating a random solution, select only a few primes in \mathcal{S} for which the corresponding value x_i is nonzero. We call \mathcal{S} the *small factor base*. Assume that there are n primes in \mathcal{S} , i.e., let $\mathcal{S} = \{p_1, \dots, p_n\}$. Then $x_i = 0$ for $i > n$ in all considered representations (x_1, x_2, \dots, x_m) , and we may just denote them by (x_1, x_2, \dots, x_n) . Here is our function for retrieving the value x that corresponds to the representations (x_1, x_2, \dots, x_n) by applying the Chinese Remainder Theorem.

get_x(x_1, x_2, \dots, x_n)

- 1: Compute $M = \prod_{i: x_i \neq 0} p_i$
- 2: Return

$$x := \sum_{i: x_i \neq 0} \frac{M c_i}{p_i} \cdot s_{i,x_i} \pmod{M},$$

where $c_i = (M/p_i)^{-1} \pmod{p_i}$

Before discussing our implementation of SSS in greater detail, let us take a look at a sketch of the complete procedure for reference throughout of the remainder of this section.

(1) **Initialization**

We perform the following steps once before starting the main loop.

- (i) Compute the factor base \mathcal{F} and small factor base \mathcal{S} .
- (ii) Compute the roots of f modulo p_i for the odd primes p_i in \mathcal{S} .
- (iii) Compute a product tree of \mathcal{F} as preparation for **smooth_batch**⁴ (see Section 2).

(2) **Main Loop**

We repeat the following steps until we have found enough smooth relations.

- (i) Choose k random primes in \mathcal{S} and compute their product M . Compute several divisors of M and store them in a set \mathcal{M} .
- (ii) For $i = 1, \dots, n$, choose $x_i \in \{1, 2\}$ if p_i is among the k chosen primes, and set $x_i = 0$ otherwise. Let x be the value returned by **get_x**(x_1, x_2, \dots, x_n).
- (iii) Compute the residues of x modulo the elements m in \mathcal{M} , and store the values $|f(x)|/m$ in a set \mathcal{C} of candidates. To keep $|f(x)|$ small, we use the residues in the range $\{-\lceil m/2 \rceil, \dots, \lfloor m/2 \rfloor\}$.
- (iv) Run **smooth_batch**(\mathcal{F}, \mathcal{C}).

⁴We modified the code from <https://facthacks.cr.yp.to> for our implementation.

TABLE 1. Choice of factor base sizes m and n

Digits	m	n
≤ 18	60	12
$19 \leq 25$	150	30
$26 \leq 34$	200	40
$35 \leq 36$	300	60
$37 \leq 38$	400	80
$39 \leq 40$	500	100
$41 \leq 42$	600	120
$43 \leq 44$	700	140
$45 \leq 48$	1000	200
$49 \leq 52$	1200	240
$53 \leq 56$	2000	400
$57 \leq 60$	4000	800
$61 \leq 66$	6000	1200
$67 \leq 70$	10000	2000
≥ 71	30000	6000

The computation of the divisors of M in Step (2.i) is comparably expensive. For this reason, we actually run the Steps (2.ii) to (2.iv) multiple times with different choices for $x_i \in \{1, 2\}$ and, thus, different values of x . One can consider the Step (2.i) as a global search and the Steps (2.ii) to (2.iv) as a local search routine. The candidates we generate in the local search are, in some sense, subsums of the sum modulo M returned in Step 2 of `get_x`. In Section 5.1, we will actually discuss a possible improvement of the method based on the subset-sum problem. In any case, this is where the name of the method comes from.

3.1. Current implementation. Let us now discuss the details of our implementation⁵ and the used parameters. For the sizes m and n of the factor base \mathcal{F} and the small factor base \mathcal{S} , we aim⁶ at the values given in Table 1. Here, m is chosen similarly as for SIQS in various factorization tools. In particular, the choices for m between 26 and 70 digits are taken from the source code of the *primefac* package in Python, which uses similar parameters as Msieve v1.52 and is one of the implementations used in our experiments in Section 4. In order to find a good choice for n , which is a new parameter in our algorithm, we conducted a number of experiments. In the end, $n = m/5$ showed satisfying results.

In addition to the Steps (1.ii) and (1.iii) in the sketch above, we conduct two more precomputations to reduce the cost of the main loop.

- (1) We want to avoid the inversion in the coefficients $c_i = (M/p_i)^{-1} \pmod{p_i}$ used in Step 2 of the function `get_x`. Let μ be the product of all primes in \mathcal{S} and let $\gamma_i := (\mu/p_i)^{-1} \pmod{p_i}$. Then it is easy to check that, for every $M \mid \mu$ and every $i = 1, \dots, n$, we have

$$\frac{\mu\gamma_i}{p_i} \equiv \frac{Mc_i}{p_i} \pmod{M}.$$

⁵Repository: <https://github.com/sbaresearch/smoothsubsumsearch>

⁶Our implementation considers the first $2m$ primes for \mathcal{F} and then removes those p with $(N|p) = -1$ from the factor bases. Since N is a quadratic non-residue for about half of these primes, this leads to slightly varying, but overall similar cardinalities as those given in Table 1.

We hence precompute the global coefficients $\Lambda_i := (\mu\gamma_i)/p_i$ for $i = 1, \dots, n$ once and may then use them in the main loop, even though the modulus M is constantly changing.

- (2) In Step (1.ii), we compute the roots $\{s_{i,1}, s_{i,2}\}$ of f modulo the odd primes in the small factor base \mathcal{S} . In addition, we also compute the differences

$$\Delta_i := \Lambda_i \cdot (s_{i,2} - s_{i,1}).$$

If we then want to switch from a solution $x \hat{=} (x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n)$ to the solution $(x_1, \dots, x_{i-1}, 2, x_{i+1}, \dots, x_n)$ or vice versa, we may easily do that by computing $(x + \Delta_i) \pmod{M}$ or $(x - \Delta_i) \pmod{M}$, respectively.

Equipped with the quantities Λ_i and Δ_i , we may now move on to the core of the SSS algorithm, which is the search function. Let \mathcal{R} and \mathcal{P} be the sets in which we save full and partial relations, respectively.

search($k, \mathcal{R}, \mathcal{P}$)

- 1: Choose k random indices in $\{1, \dots, n\}$ and store them in a set \mathcal{I} .
 - 2: Compute the product M of all primes $p_i \in \mathcal{S}$ with $i \in \mathcal{I}$.
 - 3: Compute all divisors of M with exactly $k, k-1, k-2$ and $k-3$ prime factors. Store them in a set \mathcal{M} .
 - 4: For $i = 1, \dots, n$, let $x_i = 1$ if $i \in \mathcal{I}$, and set $x_i = 0$ otherwise. Compute $x \leftarrow \text{get_x}(x_1, x_2, \dots, x_n)$.
 - 5: **for** $i \in \mathcal{I}$ **do**
 - 6: $x \leftarrow (x + \Delta_i) \pmod{M}$
 - 7: Initialize $\mathcal{C} = \emptyset$
 - 8: **for** $m \in \mathcal{M}$ **do**
 - 9: If $p_i \mid m$, **continue**
 - 10: $\bar{x} \leftarrow x \pmod{m}$
 - 11: If $2\bar{x} < m$, add $|f(\bar{x})|/m$ to \mathcal{C} . Otherwise, add $|f(\bar{x} - m)|/m$ to \mathcal{C} .
 - 12: Run **smooth_batch**(\mathcal{F}, \mathcal{C}). Let g_1, \dots, g_ℓ be the returned values.
 - 13: **for** $i = 1, \dots, \ell$ **do**
 - 14: If $g_i = 1$, add the corresponding \bar{x} to \mathcal{R} .
 - 15: If $1 < g_i < 128 \cdot p_m$, add the corresponding \bar{x} to \mathcal{P} .
-

The parameter k in the search function and the size n of the small factor base together have a direct impact on the average magnitude of the values x we are dealing with. Similarly to our approach for choosing n , we have conducted a number of experiments to derive a suitable choice for k . In the current implementation, we are using $k = 8$. In Step 3, we then compute all divisors of M with 5, 6, 7 and 8 prime factors. It follows that \mathcal{M} consists of 93 elements. In the application of **get_x** in Step 4, we use the precomputed values Λ_i . Here, we are starting with the solution (x_1, \dots, x_n) that consists of 0's and 1's. In Step 6, we make sure that we are considering a different value for x in each run of the for-loop to avoid redundancies in the found smooth relations. By adding Δ_i to the previously used x , we basically exchange the corresponding 1 in the solution vector by a 2. Step 9 also removes redundancies in the computation, as the change applied to x in Step 6 does not affect the values of $x \pmod{m}$ in the cases where p_i divides m . For all other elements in \mathcal{M} , we then compute $x \pmod{m}$ and, depending on whether $|\bar{x}|$ or $|\bar{x} - m|$ is smaller, evaluate $|f|$ at this value, divide by m and add the result to \mathcal{C} .

Finally, we apply **smooth_batch** to \mathcal{C} and check if the resulting non-smooth parts of any of the candidates equals 1, in which case we have found a full relation, or is smaller than $128 \cdot p_m$, in which case we have found a partial relation (see Section 2). We note that, after the random choice of indices in the global search in Step 1, the local search and the remainder of the function are deterministic.

We now simply repeat **search**($k, \mathcal{R}, \mathcal{P}$) until we have found enough full and partial relations. As soon as this is the case, we proceed to the linear algebra stage of the algorithm, which is the same as for other Dixon-type algorithms. Besides the already mentioned approaches for improvements in Section 5, there are also minor tweaks and variations of the algorithm that make it run a bit faster. Let us discuss two of them:

- (1) Using a different polynomial than $f(x) = (x + \lceil \sqrt{N} \rceil)^2 - N$. While SSS does not really gain anything by changing polynomials during the algorithm like in SIQS, the search seems to run faster for certain polynomials compared to others. In particular, consider $f_\alpha(x) = (x + \lceil \sqrt{\alpha N} \rceil)^2 - \alpha N$ for small choices of $\alpha \in \mathbb{N}$. One approach for finding a good choice of α is the Knuth-Schroeppel function ([29, p.335]). However, we have not used this improvement in the experiments in Section 4.
- (2) Let η be the product of all primes in \mathcal{F} , which is computed in the product tree during precomputations (see Step (1.iii) of the sketch). Before starting the main loop, we multiply η by powers of the primes in the factor base until each prime power dividing η is larger than 2^{15} . This allows to reduce the exponent e in Step 3 of **smooth_batch**. In fact, our implementation does not run Step 3 at all. We might miss a few smooth relations, but the runtime savings due to the omitted repeated squarings make up for that. In slightly different form, this improvement is already mentioned in Bernstein's original paper on the smooth-batch algorithm. In fact, a long list of other ideas for speedups of the procedure can be found in [1, Section 3]. Since **smooth_batch** appears to be the bottleneck of our implementation, we suspect that there is still room for improvement.

3.2. Comparison to SIQS. The described implementation of SSS can be considered as a brute-force search for solutions x modulo products of primes M such that $|x|/M$ is small. Considering the fact that all the solutions found by the algorithm satisfy $|x| \ll M/2$ (due to Step 11 in **search**), we obtain

$$|f(x)|/M = |(x^2 + 2x\lceil \sqrt{N} \rceil + (\lceil \sqrt{N} \rceil)^2 - N)|/M \ll \lceil \sqrt{N} \rceil + \frac{2\sqrt{N} + 1}{M} + \frac{M}{4}$$

as the worst-case upper bound for all smoothness candidates. We will see in Section 4 that, in practice, most of the candidates are actually multiple factors smaller than \sqrt{N} . Considering again the bound (2.1) of the candidates in the Quadratic Sieve, we observe that there the converse statement is true, and its candidates are on average multiple factors larger than \sqrt{N} depending on the size of the x we are sieving for. Since a similar bound also holds for SIQS, the reduced candidate size and the resulting increase in our chances to find smooth relations is the main advantage of SSS over SIQS. Another advantage is that SSS is arguably simpler than SIQS. The core functions of our approach discussed in this section can be implemented in less than 100 lines of code.

A disadvantage of SSS is that we are not able to use sieving in order to identify smooth candidates. Sieving (as described in Section 2) is much faster than **smooth_batch**, and SIQS can check more candidates in the same time. While SSS and SIQS appear to have the same asymptotic runtime complexity, there are differences in the hidden constants, which are affected by the mentioned facts, i.e.

- (1) that the candidates in SSS are smaller, and
- (2) that SIQS uses sieving.

The outcome of a practical comparison of these two algorithms basically depends on which of the two benefits (1) and (2) is larger. The experiments in Section 4 show that, in the considered input range, (1) appears to outweigh (2).

3.3. SSSf: A filter for smoothness candidates. We have already mentioned that the smooth-batch procedure appears to be the bottleneck of SSS. This becomes more noticeable on inputs N with about 55 digits or more. We now propose an improvement that works like a filter for the candidates in \mathcal{C} in Step 12 of **search**. Before applying **smooth_batch** with the complete factor base \mathcal{F} , we apply it with a subset of its smallest primes. To be more concrete, we split up \mathcal{F} into two disjoint subsets \mathcal{F}_1 and \mathcal{F}_2 . \mathcal{F}_1 contains the smallest primes in \mathcal{F} such that $\rho = |\mathcal{F}|/|\mathcal{F}_1|$ for some preset proportion ρ , and \mathcal{F}_2 contains all remaining primes. We then replace Step 12 in **search** by the following procedure.

smooth_filter($\mathcal{F}_1, \mathcal{F}_2, \mathcal{C}, \delta$)

- 1: Initialize $\mathcal{C}_f = \emptyset$
 - 2: Run **smooth_batch**($\mathcal{F}_1, \mathcal{C}$). Let g_1, \dots, g_ℓ be the returned values.
 - 3: **for** $i = 1, \dots, \ell$ **do**
 - 4: If $g_i < 10^{d/2-\delta}$ for the number d of decimal digits of N , add g_i to \mathcal{C}_f .
 - 5: Run **smooth_batch**($\mathcal{F}_2, \mathcal{C}_f$) and proceed with Step 13 of **search**.
-

The main goal of **smooth_filter** is to reduce the number of candidates that have to be checked in the smooth-batch procedure for divisibility by the complete factor base. In order to achieve that, we first apply **smooth_batch**($\mathcal{F}_1, \mathcal{C}$) to check for divisibility by the smallest primes. If the resulting non-smooth part of a g_i is not significantly smaller than $\lceil \sqrt{N} \rceil$ in Step 4, we directly discard this candidate.

We denote SSS applied with this modification by SSSf. While SSSf is in fact slower than SSS for smaller inputs N with up to around 50 to 55 digits, the improvement is noticeable on inputs with 60, 65 or 70 digits. Besides the already discussed parameters m , n and k , SSSf comes with two additional parameters ρ and δ . We are currently using the values given in Table 2.

TABLE 2. Choices of ρ and δ in SSSf

Digits	ρ	δ
60	20	8
65	22	9
70	25	10

4. EXPERIMENTS

The approach described in Section 3 has been implemented in the programming language Python, a freely available high-level language that emphasizes code readability. The reader may therefore easily reproduce all experimental results presented in this section. On the other hand, Python is considerably slower than other programming languages (such as C), which restricted the size of the input numbers N we could factorize in our experiments. We have thus focused on small to medium-sized numbers in the range from 30 to 70 decimal digits. As already mentioned in Section 2, the currently best general-purpose factorization algorithm for integers in this range is the Self-initializing Quadratic Sieve (SIQS).

In order to compare SSS to SIQS, we have used currently available implementations of SIQS in Python libraries. The first we found is a recent implementation in *sympy*⁷, a well-known library for symbolic mathematics. The second implementation we found is part of *primefac*⁸, a package focused on primality tests and integer factorization. It is noted in the documentation that their SIQS implementation is taken mostly verbatim from another package, *PyFactorise*⁹. While we have found other implementations of the Quadratic Sieve in Python, these two appeared to be most refined and competitive in our experiments. Throughout the section, we will denote sympy’s implementation by sSIQS and the implementation by primefac/PyFactorise by pSIQS. With regards to the comparability of the approaches, we point out the following.

- We have applied all three implementations with the same code for the second phase (the linear algebra stage¹⁰) to make sure that any runtime differences only result from the performance in the first phase, where smooth relations are collected. We utilized the Gaussian elimination algorithm over \mathbb{F}_2 as it is implemented in primefac. The approach is based on [17].
- For all approaches, we used a factor base size m of \mathcal{F} close to the values in Table 1. For the length L of the sieved intervals in the SIQS implementations, we used the standard settings given in the source code of pSIQS.
- Both sSIQS and pSIQS do not use the Knuth-Schroeppel function to find a suitable multiplier α for N . As already discussed at the end of Section 3.1, we also did not make use of this improvement.
- In pSIQS, the large prime variant (see end of Section 2) is not implemented. This has to be kept in mind when considering the results, and is also the reason why we did not run pSIQS on inputs with 60 digits or more.
- We used SSS with the bound $128 \cdot p_m$ for saving partial relations (see Step 15 of **search**). The same bound is used in sSIQS.

Let us now discuss our experimental setup. We consider input numbers N with $d = 30, 35, 40, 45, \dots, 70$ digits. For each of these input sizes, we did the following.

- (1) Generate a random semiprime N with d digits, such that its two prime factors are about the same size.
- (2) Apply sSIQS to factorize N . If $d < 60$, apply pSIQS to factorize N .
- (3) If $d < 60$, apply SSS to N . For $d = 60, 65, 70$, apply SSSf to N .

⁷<https://docs.sympy.org/latest/modules/ntheory.html>

⁸<https://pypi.org/project/primefac/>

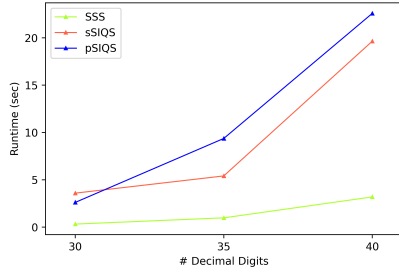
⁹<https://github.com/skollmann/PyFactorise>

¹⁰In general, this stage contributes only a tiny fraction to the complete runtime.

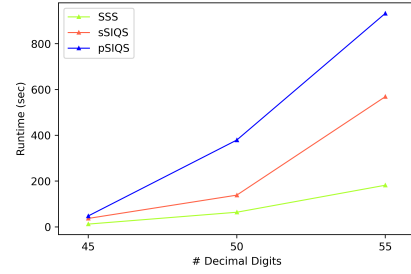
Due to the increase in general time complexity, we consider 10 different inputs N for $d \in \{30, 35, 40\}$, 5 different N for $d \in \{45, 50, 55\}$ and 2 different N for $d \in \{60, 65, 70\}$. After we have finished all runs for a certain d , we compute the mean and the standard deviation of the runtimes of all applied methods. The results¹¹ can be found in Table 3, and the mean runtimes are visualized in Figure 1 in form of line plots.

TABLE 3. Runtime complexity in seconds: Mean \pm STD

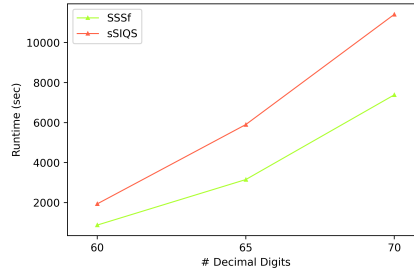
#Digits	SSS/SSSf	sSIQS	pSIQS
30	0.32 \pm 0.15	3.59 \pm 1.97	2.61 \pm 1.73
35	0.98 \pm 0.36	5.41 \pm 1.75	9.37 \pm 4.80
40	3.19 \pm 1.30	19.65 \pm 7.75	22.59 \pm 11.49
45	12.18 \pm 3.25	36.92 \pm 8.35	47.61 \pm 16.30
50	63.96 \pm 22.77	138.62 \pm 44.12	379.22 \pm 172.41
55	181.90 \pm 46.51	568.75 \pm 139.87	932.59 \pm 279.08
60	874.04 \pm 158.88	1934.37 \pm 192.66	—
65	3150.50 \pm 784.02	5900.01 \pm 1367.47	—
70	7387.15 \pm 1780.09	11411.35 \pm 1784.78	—



(A) 30 to 40 digits



(B) 45 to 55 digits



(C) 60 to 70 digits

FIGURE 1. Mean runtime complexity results as shown in Table 3

¹¹All computations have been conducted on a standard laptop with AMD Ryzen 7 5800H processor (8-core 3.2 GHz) and 16GB RAM.

For inputs with 30 – 40 digits, SSS is about 5 to 10 times faster than the SIQS approaches. For inputs with 45 – 55 digits, SSS is about 2.2 to 3 times faster than sSIQS. In this range, it becomes increasingly noticeable that pSIQS does not make use of the large prime variant, which renders it significantly slower than the other two approaches. For inputs with 60 – 70 digits, we hence only compared SSSf against sSIQS. There, SSSf is about 1.5 to 2.2 times faster than sSIQS. One observes that the runtime improvement of SSS over SIQS is particularly pronounced on the smaller input sizes. An already mentioned reason is that SSS is overall simpler and has much less overhead than the SIQS implementations. As the input size increases, this fact becomes less important and the advantage of the efficiency of sieving compared to the smooth-batch procedure becomes more and more pronounced. However, the usage of SSSf allows to maintain an improvement on all considered inputs.

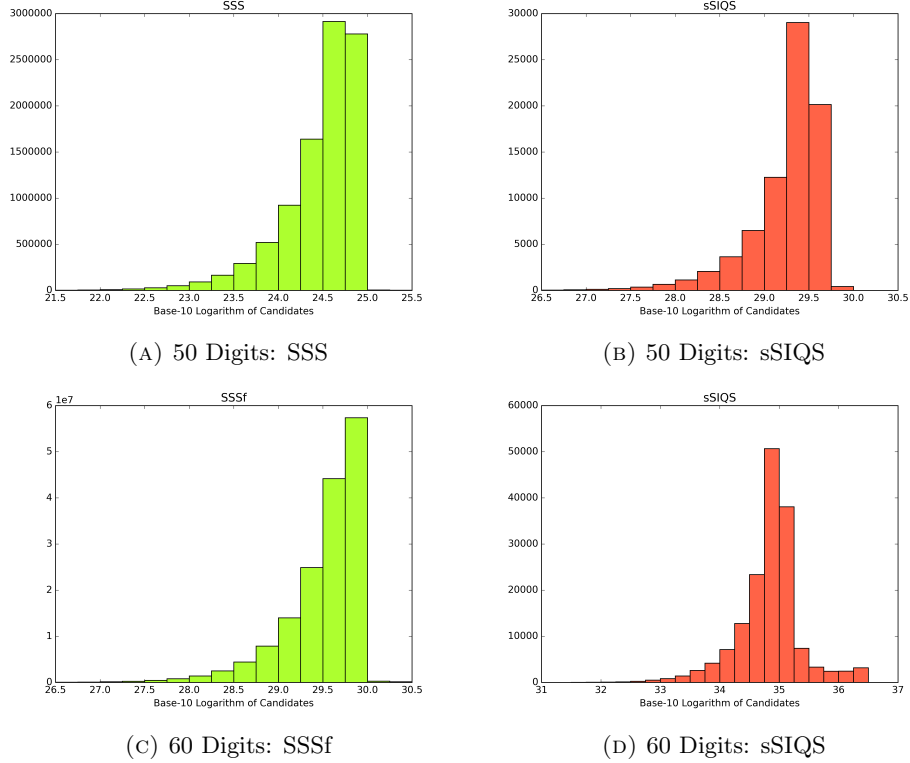


FIGURE 2. Size distributions of candidate values in SSS and sSIQS

In Section 3.2, we have already discussed the differences in the distribution of the size of candidate values in both SSS and SIQS in terms of theoretical bounds. In Figure 2, we show these distributions as they manifested in our practical experiments. On the left-hand side, we can see histograms of the base-10 logarithms of all candidates that occur in SSS and SSSf, i.e., that are members of \mathcal{C} in Step 12 in **search**. On the right-hand side, we consider similar histograms for all candidates in sSIQS that remain after the sieving procedure and are thus likely to be smooth.

Almost all candidates in SSS are smaller than \sqrt{N} , most of them between 5 to 10 times. The candidates in SIQS are on average $10^2 - 10^5$ times larger than \sqrt{N} . On the other hand, we can see on the y-axis that the number of candidates that survive the sieving (and that have to be factored in order to determine if they are really smooth) is around $10^2 - 10^3$ times smaller than the number of candidates we check in the smooth-batch procedure of SSS for inputs with 50-60 digits. While the total number of elements in the sieved intervals is much larger, this reduces the cost required for trial division.

While it would certainly be interesting to conduct experiments on numbers with 75-95 digits, an implementation in a faster programming language (such as C) would be necessary. Given the trend in Table 3, it is likely that there is a cutoff somewhere in the range 75-95 digits where our current approach for SSS and SSSf would start to be slower than SIQS. In order to make Smooth Subsum Search fit for larger input sizes, we need to replace the brute-force approach for finding residues x modulo M with small $|x|/M$ by something even faster. The next section considers possible avenues in this direction.

5. POSSIBLE IMPROVEMENTS

Our chances of finding smooth numbers with SSS depend on the ratio of the solution x and considered modulus M . The size of x directly affects the size of the considered candidate $f(x)$, while M divides $f(x)$ by definition and is a factor that is always smooth for all considered candidates. As a result, we would like to make x as small and M as large as possible. In our current implementation, we basically choose random moduli M and arbitrary values of x . For input numbers N that are larger than the ones considered in our experiments, it may make sense to refine the selection of the values of x and M . Three possible approaches come to mind:

- (1) For a fixed modulus M , try to find a related representation (x_1, \dots, x_n) such that the value x returned by `get_x`(x_1, x_2, \dots, x_n) is minimized.
- (2) For a fixed value x , try to find divisors m dividing M such that the ratio $(x \pmod{m})/m$ is minimized.
- (3) Use a different search technique to optimize both parameters at once by minimizing the objective function

$$\psi = |x|/M.$$

Section 5.1, Section 5.2 and Section 5.3 refer to these approaches respectively. Besides some proof-of-concept experiments, the described ideas have not been implemented yet. Testing their applicability to improve SSS is a subject for future research. Section 5.4 discusses some minor speedups of our code.

5.1. Lattice Reduction. We start with considering the approach of fixing M and finding a suitable subsum, i.e., a corresponding representation (x_1, \dots, x_n) such that x returned by `get_x`(x_1, x_2, \dots, x_n) is minimized. Let $M = \rho_1 \dots \rho_k$, where the ρ_i are primes in our factor base or (in contrast to our current implementation) powers of such primes. For $\gamma_i := (M/\rho_i)^{-1} \pmod{\rho_i}$ and $M_i := M\gamma_i/\rho_i$, we may write the possible values x returned by `get_x` as

$$(5.1) \quad s_{1,j_1}M_1 + s_{2,j_2}M_2 + \dots + s_{k,j_k}M_k \pmod{M},$$

where s_{i,j_i} are solutions of $f(x) \equiv 0 \pmod{\rho_i}$. Our ultimate goal is to find the choice of $(s_{1,j_1}, \dots, s_{k,j_k})$ for which the value of (5.1) is minimized.

In principle, this problem is a modular version of the *multiple choice subset-sum problem* (MCSS), which is discussed in [16, Sec. 11.10.1]. We consider k classes $N_i := \{s_{i,j_i}M_i \pmod{M} : f(s_{i,j_i}) \equiv 0 \pmod{\rho_i}\}$, and the elements of the classes are called *weights*. Denoting our weights as $\omega_{i,j_i} := s_{i,j_i}M_i \pmod{M}$, our goal is to minimize the sum $\omega_{1,j_1} + \dots + \omega_{k,j_k} \pmod{M}$. While MCSS is NP-complete, there are multiple ways to approach this problem and solve it efficiently for certain sizes of k or cardinalities of the classes N_i . For example, a time-space tradeoff is discussed in [13, Section 6], where a similar MCSS occurs in a more theoretical approach to integer factorization. There, the solution of the MCSS corresponds to the sum $S := p+q$ of the two prime factors of semiprime numbers $N = pq$. While it would be enough to solve one MCSS instance in order to find S (and thereby factor N), the problem of the approach in [13] in terms of efficiency is that both k and the classes N_i become quite large (around the size of $\log N$). In our present setting, however, we are basically free to choose k and may also control the cardinalities of the N_i by deciding in advance which prime powers to include as factors of M . We may hence choose from the large variety of available techniques for minimizing (5.1), one of which is based on lattice reduction.

It is well known that the LLL-algorithm ([20]) for lattice reduction can be used to solve a certain class of knapsack and subset-sum problems in polynomial-time ([28]). In particular, consider the standard subset-sum problem

$$(5.2) \quad a_1x_1 + \dots + a_\ell x_\ell = s,$$

where the a_i and s are integers. The task is to find $(x_1, \dots, x_\ell) \in \{0, 1\}^\ell$ such that (5.2) is satisfied. An important quantity is the density $d := \ell / \log_2(\max_i a_i)$. In [5] it is shown that an oracle for finding the shortest vector in a special lattice can be used to solve almost all subset-sum problems with $d < 0.9408$. In practice, this oracle is replaced by an application of the LLL-algorithm. The procedure works as follows: For $n := \lceil \frac{1}{2}\sqrt{\ell} \rceil$, we define the lattice $L \subseteq \mathbb{Z}^{\ell+1}$ generated by the rows $r_1, \dots, r_{\ell+1}$ of the matrix

$$\begin{pmatrix} 1 & 0 & \dots & 0 & na_1 \\ 0 & 1 & \dots & 0 & na_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & na_n \\ \frac{1}{2} & \frac{1}{2} & \dots & \frac{1}{2} & ns \end{pmatrix}.$$

If (x_1, \dots, x_ℓ) is a solution to (5.2), the vector $v = \sum_{i=1}^\ell x_i r_i - r_{\ell+1} = (y_1, \dots, y_\ell, 0)$ is an element of L , where $y_i \in \{-\frac{1}{2}, \frac{1}{2}\}$ and, hence, $\|v\|_2 \leq \frac{1}{2}\sqrt{\ell}$. Knowing v , it is easy to retrieve a solution to the subset-sum problem. Therefore, the hope of applying the LLL algorithm is that v will occur in the reduced basis of L .

If we apply the approach of lattice reduction for standard subset-sum directly to MCSS, the multiple-choice restriction (choosing exactly one element from each class N_i) is not accounted for in the definition of the lattice L , hence too many vectors representing invalid solutions will be in the reduced basis. We now discuss an idea for solving this problem. As mentioned, we consider the classes N_1, N_2, \dots, N_k , and each class contains the weights $\omega_{i,1}, \dots, \omega_{i,\kappa_i}$, where $i = 1, \dots, k$ and $\kappa_i := |N_i|$. Moreover, let $\alpha \in \mathbb{N}$ be a natural number which will be specified later. Our idea is to introduce a set of dummy subset-sum problems in the last few columns of the matrix by which the lattice is defined. The purpose is to force the lattice

reduction algorithms into favoring vectors that correspond to choosing exactly one weight from each class. For example, assume that $\kappa_i = 2$ for every i and that s is the target sum of our MCSS. Moreover, let $n := \lceil \frac{1}{2}\sqrt{k} \rceil$. In this case, the matrix defining the lattice is of the shape

$$(5.3) \quad \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & \alpha & 0 & \cdots & 0 & n\omega_{1,1} \\ 0 & 1 & 0 & 0 & \cdots & 0 & 0 & \alpha & 0 & \cdots & 0 & n\omega_{1,2} \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & \alpha & \cdots & 0 & n\omega_{2,1} \\ 0 & 0 & 0 & 1 & \cdots & 0 & 0 & 0 & \alpha & \cdots & 0 & n\omega_{2,2} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & 0 & 0 & 0 & \cdots & \alpha & n\omega_{k,1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 & \cdots & \alpha & n\omega_{k,2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \cdots & \frac{1}{2} & \frac{1}{2} & \alpha & \alpha & \cdots & \alpha & ns \end{pmatrix}.$$

So in addition to what we have seen in the matrix for the standard subset-sum problem, we add one additional column for each class, namely

$$(\alpha, \alpha, 0, 0, \dots, 0, \alpha)^T, \quad (0, 0, \alpha, \alpha, \dots, 0, \alpha)^T$$

and so on. It is clear how the definition of this matrix may be generalized to classes with more than two weights in them. Let us consider this general case, and set $\ell = \sum_i \kappa_i$. If (x_1, \dots, x_ℓ) is a binary vector encoding a solution to the MCSS (in the same way as described above for standard subset-sum), then the vector

$$v = \sum_{i=1}^{\ell} x_i r_i - r_{\ell+1} = (y_1, \dots, y_\ell, 0, \dots, 0, 0)$$

with $y_i \in \{-\frac{1}{2}, \frac{1}{2}\}$ will be in the lattice $L \subseteq \mathbb{Z}^{\ell+k+1}$, and satisfies $\|v\|_2 = \frac{1}{2}\sqrt{k}$.

Besides searching for a short vector in the lattice L , we may also reduce MCSS to a specific Closest Vector Problem (CVP). This can be achieved by considering the lattice L' defined by the matrix in (5.3) *without* the last row, and then defining the target vector $t := (0, \dots, 0, \alpha, \alpha, \dots, \alpha, ns)^T$. We already know that the vector in L' corresponding to the solution (x_1, \dots, x_ℓ) is very close to t . In the context of our research on the idea in [13], we conducted proof-of-concept experiments and noticed that this CVP approach works much better in practice than the SVP approach discussed above. One apparent reason is that there can be vectors in L that are still much shorter than $\frac{1}{2}\sqrt{k}$ in the euclidean norm. Most of these vectors relate to some combination of the weights in the last column of (5.3) that sums to 0 and omit the last row. The formulation of the CVP prevents this behavior.

We may transform our modular MCSS into an ordinary MCSS by defining the class $N_0 := \{0, M, 2M, \dots, (k-1)M\}$ and setting the approximative target sum $s = (k-1)M$. In addition, note that the LLL approach described above only works for an exact target sum. However, [15, Section 6] shows how one may reduce any approximate MCSS instance to an exact one by dividing all weights by the approximate bound. Of course, there are several other aspects and details to consider, and we will elaborate on this approach in future work. One goal would be to prove an analogue of the standard subset-sum density result in [5] for our adaptation to MCSS. Such a result would also inform the choice of M in our SSS setting. As already mentioned, we want to take M as large as possible. Using lattice reduction for subsequently finding a relatively small value x in the related subsum has the potential of increasing our chances of finding smooth values of polynomials f .

5.2. Small modular roots of linear polynomials. Let us now assume that we fix the value x (with regards to a certain modulus M) and try to find divisors m of M such that the ratio $(x \bmod m)/m$ is minimized. In principle, this is very close to what we are doing in our current implementation. However, we are basically brute-forcing the search for such a pair (x, m) by choosing all divisors m of M composed of five prime factors or more as possible moduli. One technique that might improve this step by allowing us to choose larger M is Coppersmith's method ([4]). In particular, consider the following result ([22, Theorem 3]).

Theorem 5.1. *Let M be an integer which has an unknown divisor $m \geq M^\beta$, where $0 < \beta \leq 1$. Let $0 < \varepsilon < \frac{1}{7}\beta$. Furthermore, let $g(X)$ be a univariate monic polynomial of degree δ . Then we can find all solutions x_0 for $g(X) \equiv 0 \pmod{m}$ with*

$$|x_0| \leq \frac{1}{2} M^{\frac{\beta^2}{\delta} - \varepsilon}.$$

The running time is polynomial in ε^{-1}, δ and $\log M$.

Assume that there is some divisor $m \geq M^\beta$ of M such that the residue x_0 of x modulo m satisfies $|x_0| \leq \frac{1}{2} M^{\frac{\beta^2}{\delta} - \varepsilon}$. Then we can find x_0 by running Theorem 5.1 with the polynomial $g(X) = X - x$. Having found x_0 , it is easy to determine the corresponding divisor m and, thus, a suitable pair (x_0, m) with low ratio. Solving linear equations modulo unknown divisors has also been studied in [21] and [9]. While this idea appears interesting in theory, it is not yet clear whether it is actually applicable to our problem. The main question is how to choose M and β to ensure the existence of pairs (x_0, m) that satisfy the bounds. If Theorem 5.1 is not suitable for application to our problem, we will explore other approaches in this direction.

5.3. A genetic algorithm. Finally, we assume an unrestricted search approach that chooses x and M together in an attempt to minimize the objective function $\psi = |x|/M$. For example, we could comb through different choices of x and M , save all pairs (x, M) for which ψ is below a certain bound in a set of candidates \mathcal{C} , and regularly apply the smooth-batch procedure to \mathcal{C} to find smooth values of $f(x)$. This idea was the starting point for the SSSf implementation that we applied to integers with 60 to 70 decimal digits. However, there may be more elaborated approaches to quickly generate (and identify) suitable pairs (x, M) . One such possibility is a *genetic algorithm* ([8]). In general, genetic algorithms solve an optimization problem by representing the search space by a number of individuals (the “population”) that represent possible solutions. Each individual is represented by a so-called “chromosome” that reflects its properties. In our case, we may just use the representation vectors (x_1, \dots, x_n) as discussed in the beginning of Section 3. The next step is to find the best individuals in the population with regards to a fitness function. Here, we could use a version of our objective function $\psi = |x|/M$. Finally, the best n individuals are selected as “parents” for the next generation, which is obtained by applying certain changes (the “crossovers” and “mutations”) to the chromosomes. For example, we could change a random value in a chromosome (mutation), or we could take the chromosomes from two parents (say, (x_1, \dots, x_n) and (y_1, \dots, y_n)) and generate a child chromosome as $(x_1, \dots, x_{i-1}, y_i, \dots, y_n)$ for some random index i (crossover). As soon as we have obtained enough chromosomes for the next generation, the whole process starts again, i.e., we choose the best individuals and apply mutation and crossover.

Using the chromosomes and the fitness function as discussed above, we could immediately apply a genetic algorithm to solve the SSS problem. However, the main difficulty in improving the current implementation is in finding suitable mutation and crossover operations. Ideally, they are not totally random, but instead have a tendency of generating individuals with a smaller ratio of x and M than their parents. It may therefore make sense to combine the genetic algorithm with the ideas discussed in the previous subsections.

5.4. Further speedups. In addition to the Knuth-Schroeppel multiplier and Bernstein’s optimizations for the smooth-batch procedure discussed at the end of Section 3.1, there are certainly other ways to improve SSS. Here are four examples.

- The random choice of indices in the global search (Step 1 in **search**) is not optimized. Recent tests showed that removing a few of the smallest primes from \mathcal{S} leads to a small, but consistent speedup of SSS compared to the currently reported runtimes. It may hence make sense to not only introduce an upper bound (determined by n) for the primes in the small factor base, but also a lower bound.
- We may control the batch size (i.e., the cardinality of \mathcal{C}) in **smooth_batch**. At the moment, we just apply the smooth-batch procedure at the end of each run of the main loop in **search**. It might be better to collect more candidates until we have reached a certain number, and then applying **smooth_batch**. This is particularly true for SSSf, as the cardinality of the filtered candidate set \mathcal{C}_f might actually be quite small after just one run of the **search** loop.
- One can certainly fine-tune the currently used parameters. For example, our current choices of ρ and δ in SSSf (Table 2) are not necessarily optimal.
- Due to the simplicity of SSS, the whole procedure is cut out for parallelization. Different runs of **search** are independent from one another, so we may easily use more than one processor for the collection of smooth relations.

6. SUMMARY

This paper presented Smooth Subsum Search, a new heuristic search approach for finding smooth values of polynomials. We have applied SSS as part of an integer factorization algorithm. For inputs between 30 and 70 digits, we compared SSS to the Self-initializing Quadratic Sieve. Our results show that SSS runs multiple times faster for inputs with 30 to 55 digits. SSSf, an improved version of the original procedure, is still 1.5 to 2.2 times faster on the considered inputs with 60 to 70 digits. In addition, we have presented possible ideas for achieving speedups and making the method efficient for factoring larger numbers. Our future research will concern a detailed investigation of these approaches. In particular, the lattice-based approach in Section 5.1 for solving MCSS problems could have other applications besides integer factorization. We will also work on the optimizations of the current implementation discussed in Section 5.4. Finally, we intend to explore further applications of SSS in other algorithms that also depend on finding smooth values of polynomials.

Implementation. An implementation of SSS as well as the code for the runtime experiments conducted in Section 4 is available at <https://github.com/sbaresearch/smoothsubsumsearch>.

REFERENCES

- [1] D. J. Bernstein, *How to find smooth parts of integers*, <https://cr.yp.to/factorization/smoothparts-20040510.pdf>, 2004.
- [2] H. Boender, H. J. J. te Riele, *Factoring Integers with Large-Prime Variations of the Quadratic Sieve*, Exp. Math., 5: 257–273, 1996.
- [3] S. Contini, *Factoring integers with the self-initializing quadratic sieve*, Masters Thesis, U. Georgia, 1997.
- [4] D. Coppersmith, *Finding a Small Root of a Univariate Modular Equation*, Lecture Notes in Computer Science, 1070: 155–165, 1996.
- [5] M. J. Coster et al., *Improved low-density subset sum algorithms*, Computational Complexity, 2: 111–128, 1992.
- [6] R. Crandall, C. Pomerance, *Prime Numbers, A Computational Perspective*, Second Edition, Springer Science+Business Media Inc., New York, 2005.
- [7] J. D. Dixon, *Asymptotically Fast Factorization of Integers*, Mathematics of Computation, 36 (153): 255–260, 1981.
- [8] D. E. Goldberg, J. H. Holland, *Genetic Algorithms and Machine Learning*, Machine Learning, 3: 95–99, 1988.
- [9] M. Herrmann, A. May, *Solving Linear Equations Modulo Divisors: On Factoring Given Any Bits*, In: Pieprzyk, J. (eds) Advances in Cryptology - ASIACRYPT 2008, Lecture Notes in Computer Science 5350, Springer, Berlin, Heidelberg, 2008.
- [10] D. Harvey, *An exponent one-fifth algorithm for deterministic integer factorisation*, Math. Comp., 90 (332), 2937–2950, 2021.
- [11] M. Hittmeir, *A babystep-giantstep method for faster deterministic integer factorization*, Math. Comp., 87 (314): 2915–2935, 2018.
- [12] M. Hittmeir, *A time-space tradeoff for Lehman’s deterministic integer factorization method*, Math. Comp., 90 (330): 1999–2010, 2021.
- [13] M. Hittmeir, *Integer factorization as subset-sum problem*, arXiv:2205.10074, 2022.
- [14] D. Harvey, M. Hittmeir, *A log-log speedup for exponent one-fifth deterministic integer factorisation*, Math. Comp., 91 (335): 1367–1379, 2021.
- [15] N. Howgrave-Graham, A. Joux, *New generic algorithms for hard knapsacks*, EUROCRYPT 2010, 235–256, 2010.
- [16] H. Kellerer, U. Pferschy, D. Pisinger, *Knapsack Problems*, Springer, Berlin, 2004.
- [17] Ç. K. Koç and S. N. Arachchige, *A fast algorithm for gaussian elimination over $GF(2)$ and its implementation on the GAPP*, Journal of Parallel and Distributed Computing, 13 (1): 118–122, 1991.
- [18] A. K. Lenstra, *Integer Factoring*, Designs, Codes and Cryptography, 19: 101–128, 2000.
- [19] A. K. Lenstra, H. W. Lenstra (eds.), *The development of the number field sieve*, Lecture Notes in Math. 1554, Springer, Berlin, Heidelberg, 1993.
- [20] A. K. Lenstra, H. W. Lenstra, L. Lovász, *Factoring Polynomials with Rational Coefficients*, Ann. of Math., 261(4): 515–534, 1982.
- [21] Y. Lu, R. Zhang, L. Peng, D. Lin, *Solving Linear Equations Modulo Unknown Divisors: Revisited.*, In: Iwata, T., Cheon, J. (eds) Advances in Cryptology – ASIACRYPT 2015, Lecture Notes in Computer Science 9452, Springer, Berlin, Heidelberg, 2015.
- [22] A. May, *Using LLL-Reduction for Solving RSA and Factorization Problems*, In: Nguyen, P., Vallée, B. (eds) The LLL Algorithm. Information Security and Cryptography. Springer, Berlin, Heidelberg, 2009.
- [23] J. Milan, *Factoring Small to Medium Size Integers: An Experimental Comparison*, inria-0018864v2, 2010.
- [24] M. A. Morrison, J. Brillhart, *A Method of Factoring and the Factorization of F_7* , Mathematics of Computation, 29 (129): 183–205, 1975.
- [25] C. Pomerance, *Analysis and Comparison of Some Integer Factoring Algorithms*, In: H. W. Lenstra and R. Tijdeman, Eds., Computational Methods in Number Theory, Math Centre Tracts—Part 1, Math Centrum, Amsterdam, 89–139, 1982.
- [26] C. Pomerance, *A Tale of Two Sieves*, Notices of the AMS, 43 (12): 1473–1485, 1996.
- [27] H. Riesel, *Prime Numbers and Computer Methods for Factorization*, Progress in Mathematics (Volume 126), 2nd Edition, Birkhäuser Boston, 1994.

- [28] C. Schnorr, M. Euchner, *Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems*, Mathematical Programming, 66: 181–199, 1994.
- [29] R. D. Silverman, *The Multiple Polynomial Quadratic Sieve*, Mathematics of Computation, 48 (177): 329–339, 1987.
- [30] S.S. Wagstaff Jr., *The Joy of Factoring*, American Math. Society, Providence, RI, 2013.

Email address: `mhittmeir@sba-research.org`