FOXBOX: A System for Manipulating Symbolic Objects in Black Box Representation*

Angel Díaz[1] and Erich Kaltofen[2]

[1]Department of Mathematical Sciences, IBM T. J. Watson Research Center
Yorktown Heights, New York 10598; aldiaz@us.ibm.com
[2]Department of Mathematics, North Carolina State University
Raleigh, North Carolina 27695-8205; kaltofen@math.ncsu.edu
http://www.math.ncsu.edu/~kaltofen

Abstract

The FOXBOX system puts in practice the black box representation of symbolic objects and provides algorithms for performing the symbolic calculus with such representations. Black box objects are stored as functions. For instance: a black box polynomial is a procedure that takes values for the variables as input and evaluates the polynomial at that given point. FOXBOX can compute the greatest common divisor and factorize polynomials in black box representation, producing as output new black boxes. It also can compute the standard sparse distributed representation of a black box polynomial, for example, one which was computed for an irreducible factor. We establish that the black box representation of objects can push the size of symbolic expressions far beyond what standard data structures could handle before.

Furthermore, FOXBOX demonstrates the generic program design methodology. The FOXBOX system is written in C++. C++ template arguments provide for abstract domain types. Currently, FOXBOX can be compiled with SACLIB 1.1, Gnu-MP 1.0, and NTL 2.0 as its underlying field and polynomial arithmetic. Multiple arithmetic plugins can be used in the same computation. FOXBOX provides an MPI-compliant distribution mechanism that allows for parallel and distributed execution of FOXBOX programs. Finally, FOXBOX plugs into a server/client-style Maple application interface.

1 Introduction

FOXBOX is a software system that puts in practice the black box representation of symbolic objects and provides algorithms for performing the symbolic calculus with such representations. A classical application of the black box model is the factorization of the determinant of a matrix with symbolic entries. In FOXBOX a determinant object is a function which when supplied with values for the symbols computes the value for the determinant by Gaussian elimination. FOXBOX then creates by the algorithm in [25] another function that evaluates all irreducible factors. Any evaluation will produce the values of one and the same associate (scalar multiple) of each factor. This "factors box" in turn makes a series of calls to the determinant box; these calls can be executed in parallel if need be. FOXBOX also supplies a sparse interpolation algorithm [32] with which those irreducible factors that have not too many terms can be converted to the distributed sparse representation. Since both the construction and the evaluation of the factors box are quite efficient, it becomes possible in FOXBOX to factor a matrix determinant that as a polynomial would have a huge number of terms, and to obtain the sparse factors even in the presence of very dense ones. For example, one of our benchmarks computations in §5 finds the factors in standard "sparse" representation (there are 4982 non-zero monomials) of the determinant of a general 13 by 13 symmetric Toeplitz matrix in 15 hours and 25 minutes on a single processor of a Sun Ultra 2 computer.

The black box representation has evolved from our experience with the straight-line program representation [19, 20] and the DAGWOOD system [10]. In the straight-line program model the matrix determinant and the factors box would be restricted to so called single-assignment straight-line code. That model suffers from expression swell, as the length of the produced straight-line programs is in many cases proportional to the complexity of the algorithm that produced them. In [10] an irreducible quadratic factor of a determinant of a 16 by 16 matrix whose entries are chosen from 16 indeterminates, has almost 200,000 assignments. FOXBOX's factor box requires a comparable number of instruction for the evaluation of that factor, but the procedure for doing so is of almost fixed size. Furthermore, the matrix determinant box internally can test intermediate values for being zero and thus will not fail on certain inputs, while the straight-line code cannot in general avoid a zero division for all inputs without increasing the cost of evaluation [21].

The usefulness of implicit functional representations of mathematical objects has been demonstrated in several contexts. One example are the explicit and implicit LU-factorizations of a matrix. Here the inverse of a matrix is not explicitly computed and instead a forward and backward substitution is performed when the matrix inverse is multiplied by a vector. Another example from symbolic computation itself is the evaluation of a polynomial remainder sequence as is needed in Sturm-like rootfinding methods. The remainder sequence is implicitly computed from the polynomial quotient sequence, which greatly improves the running time of computing, say, the sign variation at a point. A final example is the representation of power series, which are infinite objects. Abelson and Sussman [3] describe a stream-based approach in which the *i*-th coefficient of the series is given by a procedure that is evaluated on demand thus removing the restriction of truncating the series object at a given point.

FOXBOX systematically introduces the implicit black box representation to symbolic computation. It is written in C++ but has a client/server style interface to Maple. As we have done much of our algorithm proto-typing in Maple, a package written entirely in Maple is also available. To avoid confusion we refer to this version by the name PROTOBOX. Aside from using as an underlying

*This article describes FOXBOX 1.1 (January 30, 1998). This material is based on work supported in part by the National Science Foundation under Grant No. CCR-9319776 and CCR-9712267.

Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. ISSAC'98, Rostock, Germany. ©1998 ACM 1-58113-002-3/ 98/ 0008 $ 5.00

model for symbolic expressions the black box model, the design of FOXBOX incorporates two more methodologies. First, following the Smalltalk-based system by Abdali et al. [2], Axiom [18, 31], and the Standard Template Library (STL) of C++ [28], our design incorporates data types parameterized by arbitrary coefficient domains and generic algorithms such as homomorphic imaging of black boxes. Thus FOXBOX can be compiled with an imported underlying domain arithmetic, and in fact we currently plug into the arithmetic of SACLIB 1.1 [16], GNU's MP [13], and Victor Shoup's fast modular polynomial arithmetic package NTL 2.0 [30]. C++ template classes allow us to define a precise interface while compiling FOXBOX and the imported packages in a seamless and efficient fashion. A second methodology incorporated into FOXBOX is an MPI-compliant [15] distribution mechanism that allows for parallel and distributed execution of FOXBOX programs when needed.

The realization of FOXBOX is intimately related to the development of new powerful algorithms. It is the efficient greatest common divisor or factor boxes and sparse interpolation algorithms that make the calculus of black boxes applicable to well-known problems in symbolic computation. In FOXBOX we have made improvements to many of the algorithms in the literature, which we describe in some detail in §4.

With FOXBOX we hope to demonstrate an innovative approach to building symbolic computing software itself. One of our goals is to provide our algorithms to the non-specialist consumer of symbolic computation software. FOXBOX is designed as a component that can be easily custom assembled and incorporated into any computer algebra system because of its generic design. Furthermore, if a Maple user, say, wishes to access the C++ programmed black box algorithms in the same way as the PROTOBOX Maple procedures, we provide a client/server interface that links Maple and FOXBOX and works under many operating systems without having to customize either Maple or FOXBOX. As with "plug-and-play" hardware components, we establish that symbolic packages such as FOXBOX, which are written by a small team of specialists, can be immediately accessible to the many users of major systems.

Our paper is organized as follows. In §2 we explain our guiding design principles and describe the overall components of the FOXBOX system. The architecture of the FOXBOX server is summarized in §3. We proceed to summarize improvements to the major FOXBOX algorithms in §4. The benchmark timings on a set of challenge problems is presented in §5. In §6 we analyze our experience and state some future directions for the black box representation in symbolic computation.

2 The FoxBox Components

The FOXBOX distribution consists of a C++ object library that puts in practice the black box representation of symbolic objects and a C++ server that provides a portable interface to general purpose computer algebra systems. Several outstanding features of FOXBOX include the following:

- Manipulation of symbolic objects as black boxes
- An extensible component library for black box objects
- Efficiency through compilation
- Versatility of domain types and arithmetic
- Parallelism via an MPI compliant layer
- Conversion of black boxes to distributed representations
- Native Maple package (PROTOBOX) derived from our prototyping
- Maple interface to the FOXBOX server

We split FOXBOX into seven main components, namely: *base arithmetics*, *black box objects*, *common black box objects*, *black box algorithms*, *extended domain black box objects*, *homomorphic maps*, and *parallel black boxes*. This section will provide an overview of our guiding design methodology while covering the highlights of each FOXBOX component.

2.1 Overall Design Methodology

When designing a system like FOXBOX one is faced with the difficulty of fitting the resulting software into existing symbolic computation systems. Generic programming methodology allows us the use of existing high performance arithmetic packages and parallel computing software like MPI. Furthermore, FOXBOX's algorithms must be callable from familiar platforms like Maple and Mathematica and applicable to new user-defined black box objects. FOXBOX can be custom-assembled by a cafeteria-style selection process, where the user picks arithmetic packages and a launch platform and compiles the system. In January 1998 we have built a version for one of our factor challenges (see §5.1) using Shoup's new NTL 2.0 `zz_p` and `zz_pX` classes for 30 bit prime moduli. The compiled code still calls SACLIB's rational polynomial factorizer and FOXBOX does its internal arithmetic with GnuMP. One might even call Maple's own sparse interpolation procedure on the client side of our server §3. Since FOXBOX has a sparse interpolation function a main program in Maple is in this case not necessary. We believe, however, that mixing and matching makes FOXBOX efficient by new advances in fast arithmetic while maximally utilizing the many algorithms available across existing systems.

2.2 Base Arithmetic

The algorithms offered by the FOXBOX library are parameterized procedural schemata, namely C++ templates, that are completely independent of the underlying data representation. Hence, each algorithm can be utilized with any concrete representation of a field type or polynomial algorithm. We call such concrete representations a base arithmetic. Fundamental to the instantiation of such algorithms are our arithmetic *wrapper/adaptor classes*. External arithmetic packages are wrapped for adaptation to FOXBOX's function invocation standards. Inlining of member functions makes the adaptation efficient. The exact operations required by an arithmetic wrapper/adaptor depends on each particular black box algorithm. For example, the black box GCD algorithm does not request polynomial factorization from the external arithmetic package.

All of our algorithms are parameterized with both field and polynomial arithmetic, even though the polynomial arithmetic could be synthesized from field arithmetic (except polynomial factorization). Clearly, the efficiency of polynomial arithmetic is dependent on the representation of the field elements and a generic implementation of polynomial multiplication, for instance, would in most instances be quite inefficient. Victor Shoup's class `zz_pX` is a good example: it employs FFT based polynomial multiplication. However, the entire library only requires univariate polynomial arithmetic. Representations of bi-variate polynomials or rational functions, when needed as, for example, in bi-variate Hensel lifting, are constructed internally by FOXBOX. An arithmetic wrapper/adaptor may require a means of setting native arithmetic parameters. The code sample in Figure 1 is intended to initialize our SACLIB modular polynomial arithmetic wrapper/adaptor.

The function `SaclibInitEnv` initializes SACLIB by allocating memory and utilizing the address of `Stack` as the first variable located on the stack. FOXBOX utilizes GNU MP for its internal arbitrary precision integer arithmetic. All arithmetic wrapper/adaptors are expected to convert a GNU MP integer to its native format. An example of this is illustrated by the `SaclibSetPrime` function which initializes the SACLIB modular polynomial arith-

```
// SACLIB wrapper/adaptor
#include "PlugIns/saclib.h"
int main( int argc, char *argv[] )
{  Word Stack;
   // initialize SACLIB wrapper/adaptor
   SaclibInitEnv( 1000000, Stack );
   MP_INT MPPrime;
   mpz_init_set_str( &MPPrime, "32771", 10 );
   // set modulus
   SaclibSetPrime( &MPPrime );
   mpz_clear( &MPPrime );
   SaclibCleanUpEnv();  }
```

Figure 1: SACLIB wrapper/adaptor

metic wrapper/adaptor to GF(32771).

When switching to different arithmetics within a single FOXBOX application run it is possible to release resources associated with each wrapper/adaptor arithmetic by calling a corresponding "clean-up" function. In the case of our SACLIB wrapper/adaptor, the call to `SaclibCleanUpEnv` frees the previously allocated memory. While SACLIB allocates "heaps" of memory and employs such maintenance routines, other arithmetics may not require an explicit initialization/clean-up phase. Memory management in the presence of unknown memory models remains a difficult issue for the FOXBOX design.

2.3 Black Box Objects

Black box objects are C++ function objects derived from an abstract base class, namely `BlackBox< K >`. The `BlackBox` base class is parameterized by a coefficient domain `K` and serves as a framework which specifies the minimal interface required for all black boxes. Each black box object requires a function that provides its degree, number of variables and the probability of the correctness of its particular black box evaluation program. Furthermore, the overloading function operator is the evaluation of a black box object.

A constructor provided by each derived black box class performs a particular black box manufacturing algorithm. Each derived black box common object or black box transformation may extend this minimum black box interface by providing additional functionality specific to a particular problem. Note that we call the black box transformations, such as factorization of a black box polynomial, "black box algorithms."

2.4 Extended Domain Black Box Objects

The FOXBOX library also provides for constructing black boxes that can evaluate at points that come from a domain that is an extension of the field over which it was constructed. *Extended domain black boxes* are derived from the abstract base class `BlackBoxEx< K, L >` which itself is derived from `BlackBox`. Extended domain common objects are parameterized by a coefficient domain `K` and by an extension domain `L`, their evaluation domain. Extended domain black box algorithms often employ a distinct method of evaluation from their black box algorithm counterparts. Naturally, extended domain black box algorithms require input common objects that can evaluate over an extension domain.

We apply extended domain black boxes to speed the factorization method. There we need to evaluate the input polynomials on truncated power series domains for the computation of the right side of the Hensel lifting problem (see [11, Chapter 6] for the Hensel lifting algorithm). Moreover, in symbolic computation the coefficient domain can in many instances be much smaller than the domain of values: a polynomial over the rational numbers can be evaluated at points that are, for example, full-fledged rational functions.

Template Prototype	
Common Objects	Extended Domain Common Obj's
`BlackBoxPoly<K>`	`BlackBoxPolyEx<K,L>`
`BlackBoxRatFunc<K>`	`BlackBoxRatFuncEx<K,L>`
`BlackBoxDet<K>`	`BlackBoxDetEx<K,L>`
`BlackBox-SymToeDet<K,KP>`	`BlackBox-SymToeDetEx<K,KP,L,LP>`
`BlackBoxVandDet<K>`	
`BlackBoxCauchyDet<K>`	

Table 1: FOXBOX common object library

```
#include <iostream.h>
// SACLIB wrapper/adaptor
#include "PlugIns/saclib.h"
// Toeplitz common object
#include "BlackBox/CommonObjects/bbtoeplitz.h"
typedef BlackBoxSymToeDet< SaclibQ, SaclibQX >
   BBSymToeDet;
int main( int argc, char *argv[] )
{  Word Stack;
   // initialize SACLIB wrapper/adaptor
   SaclibInitEnv( 1000000, Stack );
   int     N      = 4;
   int     DegDet = 4;
   // construct a symm. Toeplitz det. object
   BBSymToeDet SymToeDet( N, DegDet );
   SaclibCleanUpEnv();  }
```

Figure 2: Symmetric Toeplitz common object

2.5 Common Black Box Objects

FOXBOX provides a library of common objects for constructing implicit representations. One can construct a black box polynomial or rational function from a handle to an external C function; construct a determinant object, which evaluates via Gaussian elimination; or utilize specialized common objects for the determinants of Cauchy, Vandermonde, and Toeplitz matrices. Indeed, extensibility to other implicit representations is achieved via the development of user defined common objects. For example, our fast symmetric Toeplitz determinant method computes the determinant in $O(n^2)$ arithmetic operations via the subresultant PRS algorithm [24, Theorem 2].

Table 1 on page 32 depicts the current set of common objects found in the FOXBOX library. Each common object provides an implementation for the functionality required by its corresponding black box base class. While all common objects are parameterized by a coefficient domain `K`, several black box common objects are further parameterized by a polynomial algorithm `KP`. Each extended domain common object also requires arithmetic for an extension of the coefficient domain `K`, namely the parameter `L`. Our extended domain symmetric Toeplitz determinant common object also requires a corresponding extended domain polynomial arithmetic `LP`.

The C++ code provided in Figure 2 constructs a symmetric Toeplitz common object. In this example, our fast symmetric Toeplitz determinant procedure is instantiated to utilize the wrapper/adaptor to SACLIB's rational coefficient and polynomial arithmetic, `SaclibQ` and `SaclibQX` respectively. In C++ types are not values. However, we can write a C++ `typedef` in lieu for an assignment of a type to a name. The `SymToeDet` black box common object can provide rational number values for the determinant of a 4 by 4 symmetric Toeplitz matrix over the rationals. This determinant has a total degree 4.

2.6 Black Box Algorithms

The FOXBOX library provides black box algorithms for constructing factor, greatest common divisor, and numerator/denominator black boxes (see §4). FOXBOX also supplies a specialized sparse interpolation algorithm with which black boxes representing polynomials that do not have overly many terms can be converted to a distributed sparse representation. The only difference between what we categorize as black box algorithms and common objects is that a black box algorithm generates its result by probing values from another black box supplied as input.

Template Prototype
`BlackBoxFactors< K, KP, B >`
`BlackBoxFactorsEx< K, KP, B, L, LP >`
`BlackBoxGCD< K, KP, B >`
`BlackBoxNumDen< K, KP, B >`

Table 2: FOXBOX black box algorithm object library

Table 2 details template prototypes for the black box algorithm objects available in the FOXBOX library. Similar to common objects, each black box algorithm is parameterized by a coefficient domain `K`, polynomial algorithm `KP` and possibly an extend domain arithmetic `L` and `LP`. The `B` parameter specifies a particular input black box type. The type `B` is required for the following reason. Since a C++ compiler can inline the definition of the function at the site of the calls, by using `B` as an STL-style function object type we avoid, when evaluating the black box at a point, both function look-up via a pointer to a function and even the function call. In retrospect, however, the gain in efficiency is not as much as one might expect. Our experiments in [9] show the cost of dereferencing, and the savings may not justify the associated code bloat. However, code bloat can be easily avoided by use of the virtual function calling mechanism and by supplying the (instantiated) base classes `blackbox< K >` and `blackbox< K, L >` to the algorithm templates. The actual input black boxes are then typecast as references to the base classes.

2.7 Homomorphic Maps

The homomorphic imaging of a black box object is a generic algorithm that utilizes a supplied mapping function to convert between coefficient domains. The result of such a mapping is another black box that evaluates in the homomorphic image domain, such as the integers modulo a prime. More specifically, the homomorphic map produces a new black box where all of the "static" information, which characterizes the original black box, has been converted to its modular image. FOXBOX provides a homomorphic map for all of the black box algorithm objects (see Table 3).

Template Prototype
`BlackBoxFactorsHMap< K, KP, F, FP, B, FB, H >`
`BlackBoxGCDHMap< K, KP, F, FP, B, FB, H >`
`BlackBoxNumDenHMap< K, KP, F, FP, B, FB, H >`

Table 3: FOXBOX homomorphic map library

Each homomorphic map is parameterized by the pre-image arithmetic (`K` and `KP`) and by a homomorphic image arithmetic `F` and `FP`. A "mapping" function object `H` converts between the coefficient domains. The parameters `B` and `FB` specify input black boxes to black box algorithms, the first of which evaluates over `K` and the second over `F`. The result of a homomorphic image of a black box algorithm computes its value by probing `FB` at values in `F`. If `FB` is not available, FOXBOX supplies a class, namely `BlackBoxMod< K, F, B, H>`, which will wrap/adapt a black box of type `B` so as to provide values modulo the homomorphic map. The mapping function object `H` is employed to convert the modular input to the original domain and the output to its modular representation. Modular wrapping, however, is likely to be less efficient than giving the explicit evaluation type `FB`.

In our challenge computations (see §5.1) a factor black box is constructed with SACLIB's rational polynomial arithmetic and mapped to a new black box that evaluates with NTL's fast modular polynomial arithmetic. This combination of utilizing specialized base arithmetics for different steps in the solution of a particular problem is one example of our "plug-and-play" software design methodology detailed in §2.1. In the factor challenge in §5.1, for example, the Toeplitz determinant factors are constructed by probing the Toeplitz determinant over SACLIB rationals. However, they are evaluated in the sparse interpolation algorithm by probing the same determinant over NTL integers modulo a prime. Nonetheless, both probing procedures are compiled from the same generic template member function. In fact, an error in this function was revealed modulo a small prime, when the subresultant PRS modulo this prime became non-normal and an incorrect subresultant was returned.

2.8 Parallel Black Boxes

Our sparse conversion algorithm is ideally suited for parallelization: the algorithm probes the polynomial at selected points and then performs the interpolation task by use of the obtained values. Therefore, the evaluation at the different points can be done on different computers. Each black box object is characterized by a small amount of pre-computed static information. An initial phase transmits this static information to each processor allowing for subsequent remote evaluations.

The parallel black box interface adds three member functions `Distribute`, `Wait` and `Kill` for administering remote construction, evaluation and termination of black box objects. Each class is derived from their black box algorithm counter part and extends the inherited data members and member functions by the parallel black box interface.

The parallel black box interface can be implemented utilizing different parallelization techniques. By virtue of this abstraction, applications that employ the parallel black box interface can remain independent of such techniques. For example, one can plug into the functionality offered by parallel systems such as DSC [6], PVM [12], or MPICH [15] simply by providing a corresponding derived class.

The current set of objects provided by the FOXBOX parallel black box library are realized via MPI compliant (Message Passing Interface) calls. Message-passing is a parallel programming technique used on MPP systems, workstation clusters, and other distributed memory systems. The Message Passing Interface standardization effort has produced a library specification intended for the portable development of message-passing applications. Implementations of MPI exist for heterogeneous workstations clusters, the Cray T3D, 64-bit mips3 and mips4 SGI machines, and Microsoft Windows to name a few. MPI and its workstation cluster implementation lack several features found in our distributed computing environment (DSC) such as process scheduling and dynamic process creation. However, providing a MPI compliant mechanism for parallelizing black boxes broadens FOXBOX's applicability.

A MPI parallel FOXBOX application consists of a C++ program that communicates with other processes by ultimately calling MPI routines. The initial loading of the executables onto the parallel machine is beyond the scope of the MPI interface. Each MPI implementation will have its own means of performing this task. However, once loaded, each processor executes different statements

within their copy of the same FOXBOX application program based on processor ranks. A typical application will consist of a driver and a parallel subtask portion. The driver will request evaluations from each parallel subtask. Each parallel subtask executes a copy of the MPI black box server object, which has the ability to accept messages from the interface provided by parallel black boxes. The first call to a `Distribute` member function sends an object's static information to a particular processor, as well as a point for evaluation. Subsequent calls simply send evaluation points. The call to `Wait` blocks the driver program until a particular remote evaluation can be processed. Each parallel subtask can be destroyed by a call to `Kill`.

3 The FOXBOX Server

The FOXBOX distribution provides a server application that allows the user of a general purpose computer algebra system to access the FOXBOX components in calculator style fashion. Since the FOXBOX server essentially provides for remotely invoking C++ black box object methods, there is quite a bit of overlap between the components of the FOXBOX server and FOXBOX programming library.

Throughout the development of examples, §3.1 describes the overall components of the FOXBOX server. The main design goal for an interface between FOXBOX and a computer algebra system was that this interface had to be easily portable between different systems. In §3.2 we detail the design behind the FOXBOX interface, which yields a fairly portable implementation.

3.1 Accessing the FOXBOX Server

A FOXBOX application specifies an underlying base arithmetic at compile time by template class arguments. The FOXBOX server application utilizes SACLIB's rational and modular polynomial arithmetic. However, since the components within the FOXBOX programming library are parameterized procedural schemata (see §2.2), the FOXBOX server can easily be ported to take advantage of other base arithmetics. Currently, the FOXBOX server provides for construction and evaluation of various types of black box objects. The server is also capable of converting such objects to sparse format.

As an example, let us consider the problem of computing a factor of the determinant of a 4 by 4 symmetric Toeplitz matrix utilizing the Maple interface to the FOXBOX server.

The following code fragment illustrates the initialization of the FOXBOX interface in a Maple session.

```
> read ‘bridge.mpl‘:
> FoxBoxInitEnv( ‘bpid.bbs‘, ‘cpid.bbs‘,
>  ‘command.bbs‘, ‘ans.bbs‘ ):
> Mod := 32771: Seed := 103069:
> FoxBoxSetPrime( Mod ):
```

The `read` command is utilized to import the Maple specific FOXBOX interface code into the Maple environment. Control of the remote black box objects is achieved via TCP/IP communication on a dedicated port. The FOXBOX server library function `FoxBoxInitEnv` establishes such a communication connection to a remote FOXBOX server as well as initializes several internal variables. Maple character strings formed by back quotes specify file names utilized by the FOXBOX server interface. The only restriction for each file name argument is that they be unique. The `FoxBoxSetPrime` procedure initializes the FOXBOX modular arithmetic.

The next code fragment issues FOXBOX server library calls intended to remotely instantiate two 4 by 4 symmetric Toeplitz determinant common objects. One evaluates over the rationals and the other modulo a prime. Each Maple constructor returns an integer index that serves as a remote black box object identifier. The call to `BlackBoxFactors` creates a factors black box over the rationals that can evaluate each irreducible factor of the previously constructed symmetric Toeplitz determinant common object.

```
> SymToeQ  := BlackBoxSymToe( BBNET_Q,4,-1,1.0 ):
                SymToeQ := 0
> SymToeZP := BlackBoxSymToe( BBNET_ZP,4,-1,1.0 ):
                SymToeZP := 1
> FactorsQ := BlackBoxFactors( BBNET_Q, SymToeQ,
>   Mod, 1.0, Seed ):
                FactorsQ := 2
```

The parameters for the Maple black box object constructors mirror those utilized by the FOXBOX programming library. The reader is referred to [5] and [8] for a more elaborate exposition of each FOXBOX server library procedure call. The result of the `BlackBoxHomomorphicMap` Maple constructor is a homomorphic image of the previously computed factors black box object. Such an image evaluates over the integers modulo a prime.

```
> FactorsZP := BlackBoxHomomorphicMap( BBNET_FACS,
>   FactorsQ, SymToeZP ):
                FactorsZP := 3
> FactorZP  := BlackBoxSelectValue( BBNET_ZP,
>   FactorsZP, 0 ):
                FactorZP := 4
```

The `BlackBoxSelectValue` function call serves as an n to 1 multiplexor which is utilized to select the first factor. The code below converts the first factor into its distributed sparse representation by employing the homomorphic map of the factors black box object to interpolate the selected factor modulo a prime. As an example, we provide two methods of sparse conversion. The first performs a remote conversion utilizing our modified Zippel algorithm.

```
> FB1 := SparseConversion( BBNET_ZP, FactorZP,
>   [ x1, x2, x3, x4 ], [ 4, 4, 4, 4 ], 4, Mod );
FB1 :=
          2       2                     2
  32768 x1  + 3 x2  + 32768 x1 x2 + 3 x3  +
              6 x2 x3 + 32768 x2 x4 + 32768 x1 x4
```

The call to the `SparseConversion` FOXBOX server library procedure requires as input a base arithmetic flag, an index to a black box representing a polynomial, a bound on the total degree of the input polynomial black box, a degree bound for each variable, and cardinality from which to choose random field elements. The result of this call is a vector of monomials and corresponding degrees. This representation is converted to a Maple polynomial by matching the input variables to each monomial degree pair. The second method employs Maple's sparse multivariate modular polynomial interpolation function.

```
> f := proc( x1, x2, x3, x4, p)
>    local FactorValue;
>    FactorValue   := BlackBoxEval( BBNET_ZP,
>      BBNET_FAC_HMAP_EVAL, FactorZP,
>      [ x1, x2, x3, x4 ] );
>    RETURN( FactorValue );
> end:
> readlib(sinterp):
> FB2 := sinterp( f, [x1, x2, x3, x4 ], 4, Mod );
FB2 :=
          2       2                     2
  32768 x1  + 3 x2  + 32768 x1 x2 + 3 x3  +
              6 x2 x3 + 32768 x2 x4 + 32768 x1 x4
```

The Maple `sinterp` function call requires a procedure that given integers and a prime number returns the value of a polynomial modulo the input prime. In our example, this procedure calls the `BlackBoxEval` FOXBOX server library procedure to evaluate the homomorphic map of the factors black box. Clearly, our remote interpolation generates the same result in less time since it employs an improved algorithm, utilizes compiled code, and does not incur the communication overhead of transmitting evaluations.

3.2 Underlying Interface Architecture

As stated in the introduction, the primary design goal for our interface between FOXBOX and a computer algebra system was that this interface had to be easily portable between different systems. Most general purpose computer algebra systems provide a method of invoking commands in the host operating system. Therefore, we chose a mechanism by which the FOXBOX server functions are invoked through a "system" call. Drawing from an idea utilized by our DSC interface to Maple [4], that system call executes an interface program which sends a signal to a concurrent daemon process. It is that single daemon process which forwards each request via a TCP/IP connection to the FOXBOX server. Thus, we avoid any dependence on calls to functions written in C from within a computer algebra system. Furthermore, similar to OpenMath "phrase books" [1] that translate (both ways) between the application specific representation of a mathematical concept and its representation as an OpenMath object, "bridges" to different computer algebra systems require only a small amount of customized code. Thus, our bridges are the only application-dependent portion of the FOXBOX server interface. Naturally, each different computer algebra system requires its own particular bridging code.

4 Algorithmic Improvements

We briefly summarize several improvements to algorithms from the literature that are made in their FOXBOX implementations.

4.1 Sparse Conversion

We implemented Zippel's [32] sparse interpolation algorithm with the following modifications. Instead of interpolating the polynomial $f(x_1,\ldots,x_n)$ we interpolate $f(x_1x_0,\ldots,x_nx_0)$. We thus have added a variable in the outer loop of Zippel's algorithm, which costs additional time, but we can prune the support structure for the undetermined coefficient vector when determining $f(x_1x_0,\ldots,x_ix_0,a_{i+1}x_0,\ldots,a_nx_0)$, where $a_1,\ldots,a_n$ are the anchor points for the sparse interpolation process. The degree in x_i in a term whose degree in x_0 is d cannot be higher than d due to the substitution above. Clearly, this trick does not always yield a lower count of polynomial evaluations. In the example $N = 10$ of Table 5 of Section 5 pruning reduced the number of polynomial values needed from 4,675 to 2,623.

Furthermore, we use the algorithm by Kaltofen and Lakshman [22] in the substep of solving a transposed Vandermonde system, although we have implemented one intermediate step, namely evaluation of a univariate polynomial at many points, with standard quadratic time polynomial arithmetic. The use of the algorithm from [22] reduces the required space complexity to linear in the number of terms of the sparse polynomial. At the current time our polynomial wrapper/adapter neither requires nor implements a method for multipoint polynomial evaluation, and we therefore cannot generically plug into NTL 2.0's fast code. However, the necessary changes to FOXBOX are simple by migrating to the polynomial wrapper/adapter classes all univariate polynomial algorithms, like multi-point evaluation and subresultant PRS, that are needed internally. A specific wrapper/adapter can then either call our algorithms or the ones provided by the base arithmetic. We will certainly do that in the upcoming revision of FOXBOX.

4.2 Factorization

A major bottleneck in the complexity of the black box factors constructed by the Kaltofen-Trager algorithm [25] is the necessity to interpolate in Step A of that algorithm the bivariate polynomial in X and Y

$$\begin{aligned} f(X+b_1, Y(p_2-a_2(p_1-b_1)-b_2)+a_2X+b_2,\\ \ldots, Y(p_n-a_n(p_1-b_1)-b_n)+a_nX+b_n), \end{aligned} \qquad (1)$$

where f is the polynomial in n variables that is factored. Here a_i, b_i and p_i are elements in a field `L`. Pruning as discussed in §4.1 above reduces the number of evaluations of f to about half. Since the image (1) is used for lifting the the factors, it is possible to further reduce the work for factors of small degree d by allowing the black box for f to be probed at points in the truncated power series domain $\mathtt{M} = \mathtt{L}[Y]/(Y^{d+1})$. FOXBOX's design allows for construction of extended domain objects (see §2.4) in which case one can switch to univariate interpolation over the truncated power series domain `M`. The complexity of evaluating a factor is then speeded by a factor of magnitude $O(d^2/\deg(f))$ provided that the extended black box for f over `M` runs a factor $O(d^2)$ slower, i.e., assuming classical power series arithmetic.

4.3 Greatest Common Divisor

Our implementation has followed the algorithm in [7] with a minor space improvement. In Step B the polynomial $\gamma(X,Y)$ is not explicitly constructed. Instead, its value $\gamma(p_1,1)$ is computed incrementally during the bivariate interpolation process.

4.4 Separation of Numerator and Denominator

Similarly as in the GCD algorithm, the Kaltofen-Trager algorithm [25] for evaluating the numerator and denominator black boxes can be terminated early when in Step A of their algorithm the assignment $i_1 \leftarrow 1$ is made.

5 Challenge Problems

We now report on the results of several benchmark problems, which exercise each of the components within FOXBOX. For each benchmark problem we provide the total CPU time. We used the GNU C/C++ compilers (version 2.7.2). Each of our benchmarks are problems that cannot be solved by traditional symbolic methods due to exponential intermediate expression swell. Hence, the benchmark problems reported herein represent the first symbolic solutions of such problems.

5.1 Factorization Challenge

Consider a symmetric $n \times n$ Toeplitz matrix S_n,

$$S_n = \begin{bmatrix} a_{n-1} & a_{n-2} & \ldots & a_1 & a_0 \\ a_{n-2} & a_{n-1} & \ldots & a_2 & a_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_1 & a_2 & \ldots & a_{n-1} & a_{n-2} \\ a_0 & a_1 & \ldots & a_{n-2} & a_{n-1} \end{bmatrix}.$$

The elements along the leading diagonal or along any other diagonal parallel to the leading diagonal are equal and the matrix is symmetric. The determinant of a symmetric Toeplitz matrix, S_n, has two factors over the rational numbers $\mathbb{Q}$. We selected the problem of computing a factor of a symmetric Toeplitz matrix to benchmark FOXBOX's symmetric Toeplitz common object, the factors black box object, and the sparse conversion algorithm. The solution to such a problem within the calculus of black boxes is to first construct a symmetric Toeplitz common object and factors black box. This factors black box evaluates both irreducible factors of the determinant of the aforementioned symmetric Toeplitz common object.

Table 4 provides total CPU times for the construction of the factors black box of 10 different symmetric Toeplitz matrices. Our application employs the SACLIB rational polynomial arithmetic wrapper/adaptor as the base arithmetic and executes utilizing 60 megabytes of memory.

The second phase in the solution of the factorization benchmark problem is to compute the homomorphic image of a factor black box object. The result of such a map is a new black box object

N	CPU Time	N	CPU Time
11	$0^h02'$	16	$0^h43'$
12	$0^h05'$	17	$1^h05'$
13	$0^h09'$	18	$1^h42'$
14	$0^h16'$	19	$2^h30'$
15	$0^h26'$	20	$3^h42'$

Table 4: Total CPU times (hours^{h}minutes$'$) required to construct a factors black box (over $\mathbb{Q}$) that can evaluate both irreducible factors of the determinant of a symmetric Toeplitz matrix. The processor is a Sun Ultra 1/170 (128MB), Solaris 2.5.

that evaluates the value of the pre-image modulo a prime. Finally, we perform a modular sparse conversion to retrieve the factor's distributed representation. This portion of the benchmark application utilized the NTL 2.0 modular polynomial arithmetic wrapper/adaptor for 30 bit prime moduli. Table 5 provides the complete CPU times for the factorization benchmark problem.

N	CPU Time	Degree	# Terms
10	$1^h20'$	5	931
11	$1^h34'$	5	847
12	$10^h14'$	6	5577
13	$15^h24'$	6	4982

Table 5: CPU times (hours^{h}minutes$'$) to retrieve the distributed representation of a factor from the factors black box of a symmetric Toeplitz determinant black box. Construction is over $\mathbb{Q}$ evaluation is in $\mathrm{GF}(10^8+7)$ for $N = 10, 11$, and 12 (Pentium 133, Linux 2.0) and $\mathrm{GF}(2^{30}-35)$ for $N = 13$ (Sun Ultra 2 168MHz, Solaris 2.4).

It appears from our empirical data that for even dimension Toeplitz matrices, both factors of the determinant are of degree $N/2$ and have an identical number of terms. For odd dimension Toeplitz matrix determinants, one factor is of degree $\lfloor N/2 \rfloor$ and the other has degree $\lceil N/2 \rceil$. It can be observed that for odd N the factor with degree $\lfloor N/2 \rfloor$ has fewer terms than a factor of an $N-1$ dimension Toeplitz matrix determinant. Indeed, providing an exact formulation for the number of terms and degree of each factor of an N dimensional Toeplitz matrix is a subject of future work.

Our pruning sparse conversion algorithm proved quite useful for this particular set of challenge problems. By pruning terms that have been marked as "completed" the pruning sparse conversion algorithm was able to dramatically reduce the overall size of the resulting linear systems and as a direct consequence reduced the number of black box evaluations. For instance, the problem of converting to sparse representation a factor of a 10 by 10 symmetric Toeplitz determinant required only 2623 black box calls while the algorithm without pruning employed 4675 black box evaluations. Indeed, it was a combination of the concepts of black box homomorphic maps, term pruning and NTL's fast modular arithmetic that provided the tools necessary for the successful completion of our factorization challenge problems.

5.2 Greatest Common Divisor Challenge

Vandermonde's matrix $V(P)$ formed from elements of a list $P = (x_1, \ldots, x_n)$ is a square matrix that has as its (i, j)-th entry $P[i]^{j-1} = x_i^{j-1}$, where $1 \le i, j \le n$. The determinant of Vandermonde's matrix can be expressed by the following equation:

$$\det(V(x_1, \ldots, x_n)) = \prod_{1 \le i < j \le n} (x_j - x_i).$$

Let us denote $V_1 = \det(V(x_1, \ldots, x_n))$ and $V_2 = \det(V(x_1, \ldots, x_k, y_{k+1}, \ldots, y_n))$. The greatest common divisor of V_1 and V_2 can be expressed as the following product:

$$G(x_1, \ldots, x_k) = \mathrm{GCD}(V_1, V_2) = \prod_{1 \le i < j \le k} (x_j - x_i).$$

We selected the problem of computing the greatest common divisor of V_1 and V_2 for $k = 2$ and $n = 10, 15, 20, 25, 30$. For all values of n, the final result of our benchmark problem is the polynomial $G(x_1, x_2) = x_1 - x_2$. These computations provide benchmark timings for FOXBOX's Vandermonde common object, greatest common divisor black box object, and sparse conversion algorithm. The solution of this problem within the black box framework is to first construct two Vandermonde black box objects. One representing V_1 and the other V_2. Secondly, we employ the greatest common divisor black box algorithm to create a black box representing the greatest common divisor of the aforementioned Vandermonde common objects. Finally, we perform a sparse conversion to retrieve the distributed representation of the previously constructed greatest common divisor black box.

Table 6 provides the complete CPU times and work measures for the greatest common divisor benchmark problems. Timings for our arithmetic over $\mathbb{Q}$ and in $\mathrm{GF}(10^{16}+61)$ employed SACLIB's rational polynomial arithmetic wrapper/adaptor and the NTL 1.0 fast modular polynomial arithmetic wrapper/adaptor respectively. These benchmark applications performed their computations on a Sun Ultra 1/170 with a resident set of 60 megabytes of memory.

N	$\mathbb{Q}$	$\mathrm{GF}(P)$
10	$0^h01'$	$0^h0'$
15	$0^h14'$	$0^h0'$
20	$1^h43'$	$0^h0'$
25	$8^h42'$	$0^h2'$
30	$35^h36'$	$0^h5'$

Table 6: Total CPU time (hours^{h}minutes$'$) required to retrieve the distributed representation of the GCD of two Vandermonde matrices with two variables in common, each symbolic object is in black box representation. Timings are provided for arithmetic over $\mathbb{Q}$ and in $\mathrm{GF}(10^{16}+61)$. The processor is a Sun Ultra 1/170 (256MB), Solaris 2.5.

6 Conclusions

The creation of FOXBOX demonstrate several new aspects of symbolic computation. First, we show that the black box representation of objects has pushed the size of symbolic expressions beyond what standard data structures could handle before. Second, we conclude that abstract domain types can be implemented in a general computer language without significant loss of efficiency, the so-called generic programming methodology. Third, we establish that special purpose components can be seamlessly incorporated into widely-used general purpose symbolic systems and that at the same time these components can employ varying external symbolic arithmetic packages. This is our "plug-and-play" design objective. Finally, we retain parallelizability of all of our algorithms.

The FOXBOX system is far from finished. We do not provide sparse interpolation procedures for non-standard bases [14] or black box linear algebra [23, 6]. We have not compared ours with alternative algorithms, for instance for factorization [29]. Servers to Magma, Mathematica, and MuPAD are not written. The system should be ported from our Unix platforms (Solaris, Linux, AIX) to Windows 95/NT. Distributed task management and scheduling

must be improved. The propagation of success probabilities in our randomized algorithms for nested black box construction is to be investigated. Wen-shin Lee has started to derive the necessary probability estimates.

FOXBOX is the most complex program that the authors have built so far, for it fits between the non-specialist consumer of symbolic mathematics software and the highly tuned expert software for performing fundamental tasks such as polynomial factorization. FOXBOX must run under different operating systems, allow for parallel computation, and be easy to reconfigure for new plugin and client systems. We believe that FOXBOX demonstrates how a small team of developers can produce efficient software of such wide applicability by using a general purpose symbolic system for rapid algorithmic prototyping and by generic programming methodology for producing compilable code into which multiple existing libraries can be plugged.

Acknowledgement: Plugging into fast single word modular polynomial arithmetic instead of full precision modular arithmetic was recommended to us by Michael Monagan. That speeded one of the factorization challenges in §5.1 twenty-six fold. The packing of our Maple code into PROTOBOX is done with the help of Wen-shin Lee. We thank the referees for their comments.

References

Note: many of Erich Kaltofen's publications are accessible through links in the online BIBTEX bibliography database at `www.math.ncsu.edu/~kaltofen/bibliography/`.

[1] ABBOTT, J., DÍAZ, A., AND SUTOR, R. S. A report on OpenMath a protocol for the exchange of mathematical information. *SIGSAM Bulletin 30*, 1 (1996), 21–24.

[2] ABDALI, S. K., CHERRY, G. W., AND SOIFFER, N. An object-oriented approach to algebra system design. In *Proc. 1986 Symp. Symbolic Algebraic Comput. Symsac '86* (New York, N. Y., 1986), B. W. Char, Ed., ACM, pp. 24–30.

[3] ABELSON, H., AND SUSSMAN, G. J. *Structure and Interpretation of Computer Program*. MIT Press, Cambridge, Massachusetts, USA, 1985.

[4] CHAN, K. C., DÍAZ, A., AND KALTOFEN, E. A distributed approach to problem solving in Maple. In *Maple V: Mathematics and its Application* (Boston, 1994), R. J. Lopez, Ed., Proceedings of the Maple Summer Workshop and Symposium (MSWS'94), Birkhäuser, pp. 13–21.

[5] DÍAZ, A. FOXBOX *a System for Manipulating Symbolic Objects in Black Box Representation*. PhD thesis, Rensselaer Polytechnic Instit., Troy, New York, May 1997.

[6] DÍAZ, A., HITZ, M., KALTOFEN, E., LOBO, A., AND VALENTE, T. Process scheduling in DSC and the large sparse linear systems challenge. *J. Symbolic Comput. 19*, 1–3 (1995), 269–282.

[7] DÍAZ, A., AND KALTOFEN, E. On computing greatest common divisors with polynomials given by black boxes for their evaluation. In Levelt [27], pp. 232–239.

[8] DÍAZ, A., AND KALTOFEN, E. The FOXBOX source code. `http://www.math.ncsu.edu/~kaltofen/software/foxbox`, Jan. 1998. Version 1.1.

[9] ERLINGSSON, U., KALTOFEN, E., AND MUSSER, D. Generic Gram-Schmidt orthogonalization by exact division. In Lakshman Y. N. [26], pp. 275–282.

[10] FREEMAN, T. S., IMIRZIAN, G., KALTOFEN, E., AND LAKSHMAN YAGATI. DAGWOOD: A system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Software 14*, 3 (1988), 218–240.

[11] GEDDES, K. O., CZAPOR, S. R., AND LABAHN, G. *Algorithms for Computer Algebra*. Kluwer Academic Publ., Boston, Massachusetts, USA, 1992.

[12] GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R., AND SUNDERAM, V. *PVM Parallel Virtual Machine A Users' Guide and Tutorial for Network Parllel Computing*. MIT Press, Cambridge, Massachusetts, USA, 1994.

[13] GRANLUND, T. *The GNU Multiple Precision Arithmetic Library*. Free Software Foundation, Cambridge, MA, 1993.

[14] GRIGORIEV, D. Y., AND LAKSHMAN Y. N. Algorithms for computing sparse shifts for multivariate polynomials. In Levelt [27], pp. 96–103.

[15] GROPP, W., LUSK, E., AND SKJELLUM, A. *Using MPI Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, Massachusetts, USA, 1994.

[16] HONG, H., NEUBACHER, A., AND SCHREINER, W. The design of the SACLIB/PACLIB kernels. In *Design and Implementation of Symbolic Computation Systems* (Heidelberg, Germany, 1995), A. Miola, Ed., vol. 722 of *Lect. Notes Comput. Sci.*, Springer Verlag, pp. 288–302.

[17] *ISSAC '94 Proc. Internat. Symp. Symbolic Algebraic Comput.* (New York, N. Y., 1994), ACM Press.

[18] JENKS, R. D., AND SUTOR, R. S. *axiom The Scientific Computing System*. Springer Verlag, Heidelberg, Germany, 1992.

[19] KALTOFEN, E. Greatest common divisors of polynomials given by straight-line programs. *J. ACM 35*, 1 (1988), 231–264.

[20] KALTOFEN, E. Factorization of polynomials given by straight-line programs. In *Randomness and Computation*, S. Micali, Ed., vol. 5 of *Advances in Computing Research*. JAI Press Inc., Greenwhich, Connecticut, 1989, pp. 375–412.

[21] KALTOFEN, E. On computing determinants of matrices without divisions. In *Proc. 1992 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'92)* (New York, N. Y., 1992), P. S. Wang, Ed., ACM Press, pp. 342–349.

[22] KALTOFEN, E., AND LAKSHMAN YAGATI. Improved sparse multivariate polynomial interpolation algorithms. In *Symbolic Algebraic Comput. Internat. Symp. ISSAC '88 Proc.* (Heidelberg, Germany, 1988), P. Gianni, Ed., vol. 358 of *Lect. Notes Comput. Sci.*, Springer Verlag, pp. 467–474.

[23] KALTOFEN, E., AND LOBO, A. Factoring high-degree polynomials by the black box Berlekamp algorithm. In ISSAC'94 [17], pp. 90–98.

[24] KALTOFEN, E., AND LOBO, A. On rank properties of Toeplitz matrices over finite fields. In Lakshman Y. N. [26], pp. 241–249.

[25] KALTOFEN, E., AND TRAGER, B. Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators. *J. Symbolic Comput. 9*, 3 (1990), 301–320.

[26] LAKSHMAN Y. N., Ed. *ISSAC 96 Proc. 1996 Internat. Symp. Symbolic Algebraic Comput.* (New York, N. Y., 1996), ACM Press.

[27] LEVELT, A. H. M., Ed. *Proc. 1995 Internat. Symp. Symbolic Algebraic Comput. ISSAC'95* (New York, N. Y., 1995), ACM Press.

[28] MUSSER, D. R., AND SAINI, A. *STL Tutorial and Reference Guide C++ Programming with the Standard Template Library*. Addison-Wesley Publ. Comp., Reading, Massachusetts, 1996.

[29] RUBINFELD, R., AND ZIPPEL, R. A new modular interpolation algorithm for factoring multivariate polynomials. In *Algorithmic Number Theory* (Heidelberg, Germany, 1995), vol. 877 of *Springer Lecture Notes Comput. Sci.*, Springer Verlag, pp. 93–107.

[30] SHOUP, V. NTL: A library for doing number theory. Link on web document `http://www.cs.wisc.edu/~shoup/`, Univ. Wisconsin, Dept. Comput. Sci, 1998.

[31] WATT, S. M., BROADBERY, P. A., DOOLEY, S. S., IGLIO, P., MORRISON, S. C., STEINBACH, J. M., AND SUTOR, R. S. A first report on the $A^\sharp$ compiler. In ISSAC'94 [17], pp. 25–31.

[32] ZIPPEL, R. Interpolating polynomials from their values. *J. Symbolic Comput. 9*, 3 (1990), 375–403.