# FOXBOX: A SYSTEM FOR MANIPULATING SYMBOLIC OBJECTS IN BLACK BOX REPRESENTATION

By

Angel Luis Díaz

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

Erich Kaltofen, Thesis Adviser

Mukkai S. Krishnamoorthy, Member

David R. Musser, Member

Michael M. Skolnick, Member

Stephen M. Watt, Member

Rensselaer Polytechnic Institute
Troy, New York

February 1997
(For Graduation May 1997)

# FOXBOX: A SYSTEM FOR MANIPULATING SYMBOLIC OBJECTS IN BLACK BOX REPRESENTATION

By

Angel Luis Díaz

An Abstract of a Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Examining Committee:

Erich Kaltofen, Thesis Adviser

Mukkai S. Krishnamoorthy, Member

David R. Musser, Member

Michael M. Skolnick, Member

Stephen M. Watt, Member

Rensselaer Polytechnic Institute
Troy, New York

February 1997
(For Graduation May 1997)

# CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# ACKNOWLEDGMENTS

I am externally gratefully to Erich Kaltofen for all of his assistance and support throughout my academic career. Over the years, Erich as eagerly accepted numerous roles which have included those of teacher, mentor, researcher, co-author, and travel companion. I am truly thankful for his tireless dedication and flexibility in each role. I could also never thank Erich enough for all of his efforts in regards to introducing me to the field of Computer Algebra. I am also indebted to Erich and his wife Hoang for all of their hospitality in New York and North Carolina. Both of you always went out of your way to see that I was comfortable and for that I am most appreciative.

I would also like to thank my committee members Mukkai Krishnamoorthy David Musser, Michael Skolnick, and Stephen Watt who graciously agreed to serve and help throughout this journey. Many thanks are extend to Erich Kaltofen, Stephen Watt, Robert Sutor and David Musser whose advice and assistance was invaluable when seeking employment.

I must also thank my good fiends from our research group at Rensselaer. Markus Hitz, Austin Lobo, Thomas Valente, and Lakshman Yagati were a constant source of academic assistance and comic relief which were equally important to my survival in the Ph.D program. I wish all of you continued success and happiness.

Thanks to the labors of the Rensselaer Computer Science Department's laboratory staff. David Hollinger and Nathan Schimke were always willing to provide assistance and support our endeavors, no matter the time of day or night.

A special thanks go out to my many Rensselaer friends who have offered encouragement and support. Among those who helped me maintain a connection to reality these past nine years are Joyce Brock, Vincent Calingsan, Sandy Charette, Steve Ferrara, Edward Fitzgerald, Robert Fountain, Robert Ingalls, Francis Kraner, Mike Mao, Dave Park, Pam Paslow, Ora Schongar Altug Yugner, and David Zaccaria.

I am quite grateful and would like to respectfully thank several institutions which have offered support throughout my academic career, including the Rensselaer Graduate Fellowship Program, National Science Foundation, GTE Fellowship Program and the Rensselaer Computer Science Department.

Finally, I thank my future wife Amy Roarke for her love and understanding. Without her support I would have never been to make this thesis a reality. I am looking forward to our new life together.

# ABSTRACT

The dreaded phenomenon of expression swell in symbolic computation can be palliated by adopting implicit representations for symbolic objects, such as straight-line programs or so-called black box representations. In the latter, each expression is a symbolic object, more specifically, a computer program with a set of statically initialized data, which takes as input a value for each variable and then produces the value of the symbolic object it represents at the specified point.

In this thesis we introduce FoxBox, a software package that puts in practice the black box representation of symbolic objects and provides algorithms for performing the symbolic calculus with such representations. Improved versions of the algorithms found in Kaltofen and Trager [*Journal of Symbolic Computing*, vol. 9, nr. 3, p. 311 (1990)] and Kaltofen and Díaz [*International Symposium on Symbolic and Algebraic Computation '95*, p. 232] are discussed. Also we describe an interpolation scheme based on a Zippel's algorithm [*Journal of Symbolic Computing*, vol. 9, nr. 3, p. 375 (1990)] that optimizes the number of required black box evaluations.

The design of FoxBox is intended to demonstrate how plug-in software components can be employed for generally used symbolic systems. Our implementation incorporates data types parameterized by arbitrary coefficient domains and generic algorithms. By providing a mechanism for interfacing to general purpose computer algebra systems, we broaden FoxBox's applicability. Furthermore we provide a distribution mechanism that allows for parallel and distributed execution of FoxBox programs independent of the underlying parallel architecture.

Finally, we present the results of several challenge problems which exercise our FoxBox library and represent the first symbolic solutions of such problems.

# CHAPTER 1
# INTRODUCTION

This chapter will present the reader with an overview of the black box representation for symbolic objects and provide some insight into the development of our system for manipulating such objects. In §1.1, we describe the impetus behind the black box representation, concomitantly setting the stage for the overall system description found in section §1.2. In §1.3 we furnish a guide recounting the highlights of each subsequent chapter.

## 1.1 The Black Box Representation

The proliferation of general purpose mathematical software and specialized mathematical libraries has provided a wealth of easily accessible operations for manipulating polynomials and rational functions. However, as observed in Moses [Mos71], the canonical representations used by these mathematical software tools sometimes lead to exponential intermediate expression swell.

By using an implicit representation for symbolic objects, such as by straight-line programs [Kal85a, Kal85b, FIKL88] or by black box representations [KT90, DK95], multivariate polynomials and rational functions can be manipulated efficiently without excessive intermediate expression swell.

Actually, the black box representation improves on the straight-line program representation in that the resulting objects are very small in size. In the black box representation (see Figure 1.1), each expression is a symbolic object, more specifically, a computer program with a set of statically initialized data. Such a

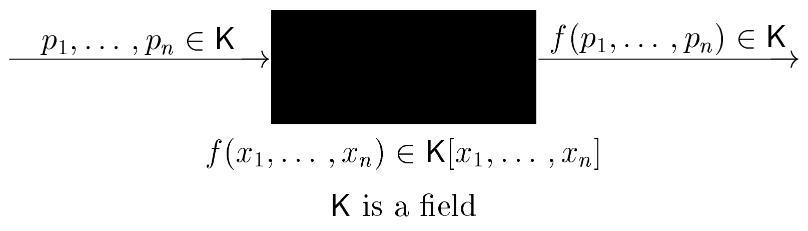

$p_1, \dots, p_n \in \mathsf{K}$
$f(p_1, \dots, p_n) \in \mathsf{K}$
$f(x_1, \dots, x_n) \in \mathsf{K}[x_1, \dots, x_n]$
$\mathsf{K}$ is a field

**Figure 1.1: Black box representation of a symbolic object**

program takes as input a value for each variable and then produces the value of the symbolic object it represents at the specified point.

Within the black box representation, the polynomial factorization (§3.2) and greatest common divisor (§3.3) problems, as well as the problem of extracting the numerator and denominator of a rational function (§3.4), can all be solved in random polynomial-time [KT90, DK95]. Each of the algorithms produce black boxes (computer programs) which evaluate the answer polynomial at arbitrary points by making calls to the black boxes given as the input. The algorithms are Monte-Carlo in the sense that their output, the constructed evaluation programs, are correct with controllably high probability.

The black box approach to computing polynomial greatest common divisors, polynomial factorization, and rational function numerator and denominator separation has several advantages over other methods. The black box algorithms excel when using objects that cannot be manipulated by sparse techniques due to their denseness, such as determinants of matrices with polynomial entries. The implication is that the algorithms should be applied to large problems. The complexity of all black box algorithms is polynomial in the number of indeterminants of the input polynomial. Hence, the algorithms are very competitive on problems with many variables. Also, the generated black boxes can be translated rapidly to sparse format by using our pruning sparse interpolation algorithm (§3.1) or used as input to other black box algorithms for further computations. Finally, the algorithms probe the input black box(es) at selected points and then perform a particular task via the utilization of the obtained values. Therefore, the evaluation at the different points can be done in parallel on different compute nodes.

The usefulness of implicit functional representations of mathematical objects has been demonstrated in several contexts [DK97b]. One example is the LU-factorization 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 [AS85] 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.

In summary, the black box representation permits the manipulation of multivariate polynomials and rational functions that are exponential in size with many variables more efficiently than is allowed by traditional methods. Additionally, when needed, one can rapidly convert a black box representation of a polynomial into its canonical form. What follows is an overall description of our system, FoxBox, which allows a scientist or engineer to readily take advantage of the black box representation. Indeed, the realization of FoxBox was intimately related to improvements to many of the algorithms in the literature (§3) while its implementation demonstrates how "plug-in" software components can be employed for generally utilized symbolic systems (§2).

## 1.2 An overview of FoxBox

FoxBox is a software package that puts into 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 §3.2 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, which if need be can be executed in parallel. FoxBox also supplies a sparse interpolation algorithm (see §3.1) with which those irreducible factors that have not too many terms can be converted to the distributed sparse representation which is common in symbolic computation systems. Since both the construction and the evaluation of the factors box is quite efficient, in FoxBox it thus becomes possible 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 §4 finds the factors (in standard representation) of the determinant of a general 13 by 13 symmetric Toeplitz matrix in $388^h 57'$ on 15 workstations of varying architectures.

The black box representation has evolved from our experience with the straight-line program representation [Kal88, Kal89] and the Dagwood system [FIKL88]. 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 [FIKL88] an irreducible quadratic factor of a determinant of a 16 by 16 matrix, whose entries are chosen from 16 indeterminants, 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 [Kal92]

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 do much of our algorithm prototyping in Maple, a version written entirely in Maple also exists. Aside from using as an underlying 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. [ACS86], Axiom [JS92, WBD$^+$94], and the Standard Template Library (STL) of C++ [MS96], 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 [HNS95], GNU's MP [Gra93], and Victor Shoup's fast modular polynomial arithmetic package NTL [Sho95, Sho97].

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 [GLS94] distribution mechanism that allows for parallel and distributed execution of FoxBox programs when needed. The benchmark problems that we have solved with FoxBox (§4) are such extensive symbolic computation tasks that the use of multiple processors for their solution is necessary.

The creation 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 §3. One improvement fundamentally affected the design of FoxBox itself. Borrowing an idea from our factorization algorithm of polynomials in straight-line representation [Kal89] our factor box evaluator can be dramatically improved in performance for small degree factors if the black box for the original polynomial can be probed at truncated power series as values for the variables. This need led us to the following natural design modification. It is now possible to evaluate any factor box, say, at points that come from a domain that is an extension of the field over which the polynomial was factored. One thus can, for instance, factor a polynomial over the rational numbers and evaluate the factors at complex rationals. A second notable improvement concerns the implementation of Zippel's sparse interpolation algorithm [Zip90]. In FoxBox the black box polynomial functions can be fairly complex, as they are constructed from basic boxes such as matrix determinants by sophisticated transformations such as the factor algorithm. The efficiency of sparse interpolation is thus significantly influenced by the number of probes, which we optimize by a novel term pruning technique during Zippel's algorithm.

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 incorporated into any major computer algebra system because of its generic design. Furthermore, if a Maple user, say, wishes to access the black box algorithms in a calculator-like fashion, which Odlyzko suggests is the major use of symbolic math software [Odl96], our client/server interface provides a link between Maple and FoxBox that will work under most any operating system 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.

## 1.3 Organization of Thesis

We begin in chapter 2 by describing our FoxBox system which puts into practice the black box abstraction and serves an example of our methodology for implementing symbolic software as "plug-in" components. Indeed, with the design of FoxBox, we furnish an efficient C++ implementation which follows the black box model while providing data types parameterized by arbitrary coefficient domains and generic algorithms. We introduce each of the major components within FoxBox and provide an exposition of the underlying software architecture for each component. Such components include our so-called base arithmetics, black box objects, common black box objects, black box algorithms, extended domain black box objects, homomorphic maps and parallel black boxes. Finally, we also describe our FoxBox server application, which allows the user of a general purpose computer algebra system to access the FoxBox components in calculator style fashion.

In Chapter 3 we provide a full report of our pruning sparse interpolation based on Zippel's algorithm [Zip90]. Our sparse conversion employs a novel term pruning technique with the overall goal of reducing the number of black box evaluations, since such evaluations are the most costly portion of each black box algorithm. We continue by providing an exposition of several algorithms originally described in Kaltofen and Trager [KT90] for manipulating multivariate polynomials and rational functions given by black boxes for their evaluations. For each black box algorithm, we describe our efficacious algorithmic improvements aimed at providing an efficient realization for the calculus of black boxes. For the black box factorization algorithm, we present two mechanisms for reducing the costly bi-variate interpolation associated with factor box evaluation. First, we apply our term pruning technique to reduce by half the complexity and number of black box evaluation points required by the bi-variate interpolation. Secondly, we describe the notion of extended domain black boxes which can dramatically improve performance of the bi-variate interpolation step for small degree factors. We also furnish a full algorithmic description and analysis of a new algorithm for computing the greatest common divisor of polynomials in black box format. Our algorithm builds on the description of Kaltofen and Trager [KT90] by employing a modular GCD approach. Finally, we describe a technique first observed in our GCD algorithm which markedly increases performance of the black box reduced numerator and denominator algorithm.

In Chapter 4, we report on the results of several benchmark problems which exercise each of the components within FoxBox. We provide solutions and timings for "challenge" computations which represented the first ever symbolic solutions of such problems. We utilize several FoxBox components to furnish the results of computing a factor of a symmetric Toeplitz determinant over a cluster of workstations. More specifically, we provide solutions for symmetric Toeplitz determinants of dimension $10, 11, 12$ and 13. We also provide the timings required to retrieve the distributed representation of the GCD of two Vandermonde matrices with two variables in common. Each symbolic object is represented in its black box format. For this benchmark problem, results are provided for Vandermonde matrices of dimensions $10, 15, 20, 25$ and 30. Finally, we nest several black box objects to attain a single factor of the numerator of a Cauchy determinant for dimensions 5 through 12.

Chapter 5 recounts our contributions to the calculus of black boxes while concomitantly laying out several future challenges.

# CHAPTER 2
# THE FOXBOX SYSTEM

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 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 implementation derived from our prototypical efforts

– Maple interface to the C++ FoxBox server

In §2.1 we demonstrate an innovative approach to building symbolic computing software, and its application in the design of the FoxBox system. In §2.2, we provide an overview of the major components within FoxBox, then detail the underlying software architecture in §2.3. Finally, in §2.4, we describe the FoxBox server which enables the user of a general purpose computer algebra system to access the FoxBox components.

## 2.1 Design Methodology

Our so called "double sided fan" design (Figure 2.1) is intended to demonstrate how plug-in software components can be employed by generally used symbolic systems. Blades on the bottom represent base arithmetic packages and support libraries while each blade on the top represents a different method of system access.

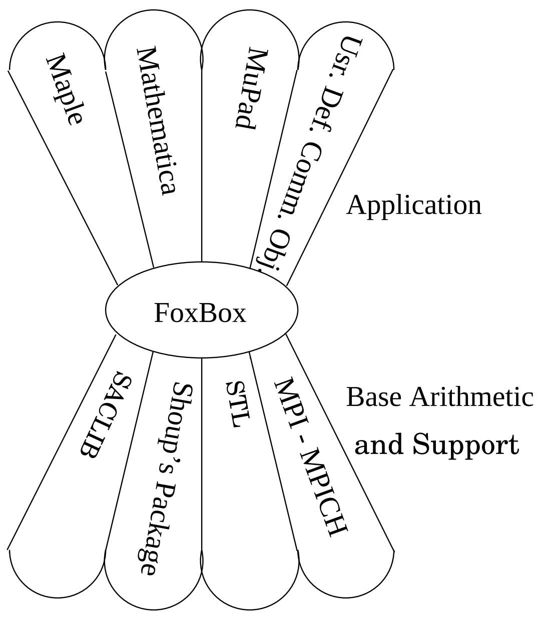


**Figure 2.1: Double sided fan design**

By applying this design methodology, our FoxBox system is able to take advantage of specialized arithmetic packages and support libraries without loss of efficiency; at the same time FoxBox remains versatile with respect to the user interface. When one designs symbolic computation software as components that can be easily incorporated into any computer algebra system and that can be compiled with several underlying base arithmetics, the software can take on many different arrangements. Drawing from our double sided fan analogy, $n$ top blades and $m$ bottom blades would allow $nm$ possible instantiations of one symbolic computation software system. Indeed, in the following section we provide an example of a C++ FoxBox application which takes advantage advantage of two underlying base arithmetics. Our illustration utilizes the specialty offered by each arithmetic for different stages of a symbolic computation.

## 2.2 Overview of FoxBox Components

There are seven major components within FoxBox, 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 cover the highlights of each component, leaving an exposition of our underlying software architecture to §2.3.

### 2.2.1 Base Arithmetic

The algorithms offered by the C++ FoxBox library are parameterized procedural schemata that are completely independent of the underlying data representation. Hence, a single expression of 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.* Native arithmetic packages are wrapped and adapted to express a particular polynomial algorithm or field type along with its corresponding access operations. Such flexibility implies that the black box algorithms have a broader utility. By inlining the member functions provided by the wrapper/adaptor classes, dedicated arithmetic packages can be utilized for specialized applications of the black box algorithms in an efficient manner.

The exact operations required by an arithmetic wrapper/adaptor depends on each particular black box algorithm. However, the entire library only requires univariate polynomial arithmetic, even when manipulating implicit representations of multivariate polynomials or rational functions. The representations of bi-variate polynomials or rational functions required by the black box algorithms are constructed by FoxBox itself.

An arithmetic wrapper/adaptor also provides one or more means of setting native arithmetic parameters. The code sample in figure §2.2 is intended to initialize our SACLIB modular polynomial arithmetic wrapper/adaptor.

In order to utilize FoxBox components, an application must employ the preprocessor `#include` directive to make available each particular component. The C++ FoxBox library distribution organizes each header file in accordance to FoxBox's overall component structure.

The function `SaclibInitEnv` initializes SACLIB by allocating memory and utilizing the address of `Stack` as the first variable located on the stack. FoxBox uti-

```
#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 modulo
   SaclibSetPrime( &MPPrime );

   mpz_clear( &MPPrime ); SaclibCleanUpEnv();
}
```

**Figure 2.2: SACLIB wrapper/adaptor**

lizes 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 arithmetic wrapper/adaptor to GF(32771).

A FoxBox application releases 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.

### 2.2.2 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. A constructor provided by each black box performs a particular black box manufacturing algorithm. Using a black box as a function is analogous to the evaluation of a black box. Since a C++ compiler can inline the definition of the function at the site of the calls, using functional objects not only avoids the overhead of an indirect function call (as occurs when using pointers to functions), it even eliminates the cost of a direct call. Each distinct black box common object and algorithm may extend this minimum black box interface by providing additional functionality specific to a particular problem.

### 2.2.3 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 Cauchy, Vandermonde, and Toeplitz matrices. Indeed, extensibility to other implicit representations is achieved via the development of user defined common objects. For example, our special purpose symmetric Toeplitz determinant algorithm computes the determinant in $O(n^2)$ operations utilizing a technique based on subresultant computations.

The C++ code provided in Figure 2.3 constructs a symmetric Toeplitz common object. In this example, our special purpose symmetric Toeplitz determinant is instantiated to utilize the wrapper/adaptor to SACLIB's rational coefficient and polynomial arithmetic, `SaclibQ` and `SaclibQX` respectively. Due to the potential of deeply nested template arguments, a C++ FoxBox application utilizes the `typedef` facility for creating new data type names. We utilize capitalized names, `BBSymToeDet` for example, to emphasize `typedef`s. The `SymToeDet` black box common object can provide values for the determinant of a 4 by 4 symmetric Toeplitz matrix. This determinant has a total degree 4.

### 2.2.4 Black Box Algorithms

The C++ FoxBox library provides black box algorithms for constructing factor, greatest common divisor, and numerator/denominator black boxes (see §3). FoxBox also supplies a specialized sparse interpolation algorithm with which black

```
#include <iostream.h>
#include "PlugIns/saclib.h"
#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 symmetric Toeplitz determinant object
   BBSymToeDet SymToeDet( N, DegDet );

   SaclibCleanUpEnv();
}
```

**Figure 2.3: Symmetric Toeplitz common object**

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 from a sophisticated process that probes values from another black box supplied as input.

Expanding on our previous example, the code in figure§2.4 retrieves the distributed representation of a factor of a 4 by 4 symmetric Toeplitz determinant. The `BBFactors` type definition is used to generate an instance of our factors black box algorithm which employs SACLIB's rational coefficient and polynomial arithmetic. Once constructed, a `BBFactors` object evaluates the irreducible factors of a symmetric Toeplitz black box object given by the `BBSymToeDet` type definition. The `BlackBoxSelector` auxiliary function servers as a $n$ to 1 multiplexor (in the narrow sense) which is utilized to select a particular factor. In the previous example where the exact type of an black box object is known at compile time, efficiency is ob-

```
#include <iostream.h>
#include "PlugIns/saclib.h"
#include "BlackBox/CommonObjects/bbtoeplitz.h"
#include "BlackBox/Algorithms/bbfactors.h"
#include "BlackBox/Aux/bbselector.h"
#include "BlackBox/Aux/bbvector.h"
#include "BlackBox/Algorithms/bbsparse.h"

typedef BlackBoxSymToeDet< SaclibQ, SaclibQX >
   BBSymToeDet;
typedef BlackBoxFactors< SaclibQ, SaclibQX, BBSymToeDet >
   BBFactors;
typedef BlackBoxSelector< SaclibQ, BBFactors >
   BBFactor;

int main( int argc, char *argv[] ) {
   Word Stack; int i;
   // initialize SACLIB wrapper/adaptor
   SaclibInitEnv( 1000000, Stack );
   MP_INT  MPCard; mpz_init_set_si( &MPCard, 32771 );
   int     N = 4; int DegDet = 4;

   // construct symmetric Toeplitz determinant object
   BBSymToeDet SymToeDet( N, DegDet );
   // construct a factors black box
   double    Prob  = -1.0; int Seed = 103069;
   BBFactors Factors( SymToeDet, Prob, Seed, &MPCard );

   // interpolate the first factor
   BBFactor                 FirstFactor( Factors, 0 );
   BBSPARSEAUX_VarVector    Vars; BBSPARSEAUX_DegVector Degs;
   SaclibQX                 SaclibQXElm[BBSPARSEAUX_MAXTERMS];
   BBSPARSEAUX_VecDegVector AnsDegs; bbvector< SaclibQ > AnsMons;
   int IsRestart = BBDEFS_FALSE; char* ProbName = NULL;
   for( i=1; i<=N; i++ ) { Vars.push_back(i);
      Degs.push_back(DegDet); }
   SparseInterp( FirstFactor, Vars, Degs, DegDet, &MPCard,
      AnsDegs, AnsMons, SaclibQXElm, ProbName, IsRestart );

   mpz_clear( &MPCard ); SaclibCleanUpEnv();
}
```

**Figure 2.4: Black box factors and sparse conversion**

tained at the cost of potential *code bloat*. The code bloat problem can be alleviated by generating instances of black box algorithms which utilize our black box base class. This technique is described in §2.3.2.

The black box factors algorithm instantiated in figure §2.4 requires as input a seed value which can be utilized to provide consistency between runs, a black box symmetric Toeplitz object and a measure of reliability. In this example, supplying a value of $-1$ as the success probability measure indicates that random field elements should be selected from a field of cardinality `MPCard`. The probability of success measure associated with this cardinality is computed and consequently inserted into the `Prob` variable.

The factors black box object extends on the minimum interface imposed by our black box base class by adding several member functions, which allows access to other important information such as the number of factors and factor exponents. Each distinct black box common object and algorithm may extend the minimum black box interface by providing additional functionality specific to a particular problem.

Our sparse conversion implementation utilizes a subset of the functionality of STL vectors for maintaining monomials and degree information. We provide such a subset as an auxiliary package for users who may not have access to STL (`bbvector.h`). The sparse conversion algorithm requires a vector of variable identifiers (`Vars`), bound on the degree of each variable (`Degs`), and bound on the total degree in order to populate `AnsMons` and `AnsDegs`. Since several of the benchmark problems that we have solved with FoxBox (see §4) are rather extensive symbolic computations, our sparse conversion algorithm provides a "check pointing" mechanism which helps to ensure fault tolerance. In our example, this functionality has been deactivated by virtue of the `IsRestart` flag. As a final note, the actual base arithmetic utilized by the `SparseInterp` template function is specified by its invocation parameters.

### 2.2.5 Extended Domain Black Box Objects

The FoxBox C++ library also provides for constructing black boxes that can evaluate at points not only from a particular domain but also at points that come from a domain which 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`. Extended domain black box algorithms often employ a distinct method of evaluation from their black box algorithm counterparts. Furthermore, extended domain black box algorithms require as input common objects that can evaluate over an extension domain.

The code displayed in figure §2.5 constructs the extended domain factors black box of a 4 by 4 symmetric Toeplitz determinant. In this example, the symmetric Toeplitz determinant common object `SymToeDetEx`, can be probed at truncated power series, as well as, at rational numbers as values for the variables. The extended domain factor black box algorithm constructs the factor black box by probing `SymToeDetEx` at rational numbers and performs subsequent evaluations by probing `SymToeDetEx` at truncated power series.

The function `SaclibSetTruncDeg` provided by the SACLIB wrapper/adaptor sets the appropriate truncation degree for the truncated rational polynomial arithmetic. The reader is directed to observe that the code samples provided in figure §2.4 and figure §2.5 are similar as a result of our underlying design methodology discussed in §2.1. Hence, the program detailed in figure §2.5 can easily be extended to retrieve the same distributed representation of a factor of a 4 by 4 symmetric Toeplitz determinant as in our previous example (figure§2.4) by simply adding the appropriate sparse conversion code.

### 2.2.6 Homomorphic Maps

The homomorphic imaging of a black box object is a generic algorithm which utilizes a supplied mapping function to convert between coefficient domains. The result of such a mapping is another black box which evaluates over the integers

```
#include <iostream.h>
#include "PlugIns/saclib.h"
#include "BlackBox/CommonObjects/bbtoeplitz.h"
#include "BlackBox/Algorithms/bbfactors.h"

typedef BlackBoxSymToeDetEx< SaclibQ, SaclibQXT >
   BBSymToeDetEx;
typedef BlackBoxFactorsEx< SaclibQ, SaclibQX, SaclibQXT,
   BBSymToeDetEx > BBFactorsEx;

int main( int argc, char *argv[] )
{
   Word Stack; int i;
   // initialize SACLIB wrapper/adaptor
   SaclibInitEnv( 1000000, Stack ); SaclibSetTruncDeg( 5 );
   MP_INT  MPCard; mpz_init_set_si( &MPCard, 32771 );
   int N = 4; int DegDet = 4;

   // construct an extended domain symm. Toe. det. object
   BBSymToeDetEx SymToeDetEx( N, DegDet );

   // construct an extended domain factors black box
   double Prob = -1.0; int Seed = 103069;
   BBFactorsEx FactorsEx( SymToeDetEx, Prob, Seed, &MPCard );

   mpz_clear( &MPCard ); SaclibCleanUpEnv();
}
```

**Figure 2.5: Extended domain black boxes**

modulo a prime. In fact, interoperability between distinct base arithmetics can be achieved by suppling the homomorphic imaging algorithm with an appropriate mapping operator.

Following our running example, the code in figure §2.6 demonstrates how to perform the homomorphic map of a factors black box. This particular instance of the factor homomorphic map algorithm (`BBFactorsQMapZP`), requires as input a factor black box constructed over the rationals, a mapping operator `SaclibQShoupZP` which converts SACLIB rational numbers to NTL's modular representation and a symmetric Toeplitz determinant common object (of type `BBSymToeDetZP`) which

```
#include <iostream.h>
#include "PlugIns/saclib.h"
#include "PlugIns/shoup.h"
#include "PlugIns/maps.h"
#include "BlackBox/CommonObjects/bbtoeplitz.h"
#include "BlackBox/Algorithms/bbfactors.h"
#include "BlackBox/HomomorphicMaps/bbfachmap.h"

typedef BlackBoxSymToeDet< SaclibQ, SaclibQX >
   BBSymToeDetQ;
typedef BlackBoxSymToeDet< ShoupZP, ShoupZPX >
   BBSymToeDetZP;
typedef BlackBoxFactors< SaclibQ, SaclibQX, BBSymToeDetQ >
   BBFactorsQ;
typedef BlackBoxFactorsHMap< SaclibQ, SaclibQX, ShoupZP,
   ShoupZPX, BBSymToeDetQ, BBSymToeDetZP, SaclibQShoupZP >
   BBFactorsQMapZP;

int main( int argc, char *argv[] )
{
   Word Stack; int i;
   // initialize SACLIB and Shoup wrapper/adaptors
   SaclibInitEnv( 1000000, Stack );
   MP_INT  MPCard; mpz_init_set_si( &MPCard, 32771 );
   ShoupInitEnv( &MPCard );
   int N = 4; int DegDet = 4;

   // construct symmetric Toeplitz determinant object over Q and ZP
   BBSymToeDetQ  SymToeDetQ( N, DegDet );
   BBSymToeDetZP SymToeDetZP( N, DegDet );

   // construct a factors black box over Q
   double Prob = -1.0; int Seed = 103069;
   BBFactorsQ FactorsQ( SymToeDetQ, Prob, Seed, &MPCard );

   // map the factors black box to Shoup's modular arithmetic
   SaclibQShoupZP   h;
   BBFactorsQMapZP  FactorsZP( SymToeDetZP, FactorsQ, h );

   mpz_clear( &MPCard ); SaclibCleanUpEnv(); ShoupCleanUpEnv();
}
```

**Figure 2.6: Homomorphic map of a factor black box**

evaluates over the integers modulo a prime. The consequent homomorphic image black box evaluates utilizing NTL's fast modular polynomial arithmetic.

Note that although the static information "housed" within the factors black box over the rationals includes polynomials, all that is required is a mapping operator to convert between coefficient domains. FoxBox employs a generic algorithm which converts between polynomial algorithms utilizing the coefficient domain mapping supplied to the factor homomorphic map algorithm. In fact, this is one example of several optimized internal procedures utilized by FoxBox to alleviate the user from providing additional functionality.

### 2.2.7 Parallel Black Boxes

Once constructed, a black box algorithm utilizes a small amount of precomputed static information for evaluation. Hence, black box objects are ideally suited for parallelization: an initialization phase transmits the static information to each processor allowing for subsequent probes. FoxBox provides a parallel version of each black box algorithm. Each parallel black box object is derived (by inheritance) from its counterpart. The parallel black box interface adds three member functions for administering its remote evaluation, namely `Distribute`, `Wait` and `Kill`. Our sparse conversion algorithm, for example, utilizes these member functions for parallel execution.

Message-passing is a parallel programming technique used on MPP (Massively Parallel Procesor) systems, workstation clusters, and other distributed memory systems. The Message Passing Interface (MPI) effort produced a library specification intended for the portable development of message-passing applications. Hence, providing a MPI compliant mechanism for parallelizing black boxes broadens FoxBox's applicability. All of the parallel black box member functions are realized via MPI compliant calls.

In order for a FoxBox application to exploit the existence of multiple processors within the MPI framework, the user must issue a directive to the operating system. This has the effect of placing a copy of an executable program on each processor. Thus different processors execute different statements within their copy of

```
#include <iostream.h>
#include "PlugIns/saclib.h"
#include "PlugIns/shoup.h"
#include "PlugIns/maps.h"
#include "BlackBox/CommonObjects/bbtoeplitz.h"
#include "BlackBox/Algorithms/bbfactors.h"
#include "BlackBox/HomomorphicMaps/bbfachmapmpi.h"
#include "BlackBox/Aux/bbselectormpi.h"
#include "BlackBox/MPI/bbservermpi.h"
#include "BlackBox/Aux/bbvector.h"
#include "BlackBox/Algorithms/bbsparse.h"

typedef BlackBoxSymToeDet< SaclibQ, SaclibQX >
   BBSymToeDetQ;
typedef BlackBoxSymToeDet< ShoupZP, ShoupZPX >
   BBSymToeDetZP;
typedef BlackBoxFactors< SaclibQ, SaclibQX, BBSymToeDetQ >
   BBFactorsQ;
typedef BlackBoxFactorsHMapMPI< SaclibQ, SaclibQX, ShoupZP,
   ShoupZPX, BBSymToeDetQ, BBSymToeDetZP, SaclibQShoupZP >
   BBFactorsQMapZP;
typedef BlackBoxSelectorMPI< ShoupZP, BBFactorsQMapZP >
   BBFactorZP;
typedef BlackBoxServerMPI< ShoupZP, ShoupZPX,
        BBSymToeDetZP >
   BBServerMPI;

int main( int argc, char *argv[] ) {
   MPI_Init(&argc, &argv);
   int  MyRank, NumProc, N = 4, DegDet = 4;
   char FileName[BBDEFS_FILEZ];
   MP_INT MPCard; mpz_init_set_ui( &MPCard, 32771  );
   MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
   MPI_Comm_size(MPI_COMM_WORLD, &NumProc);
   ....
   mpz_clear( &MPCard ); ShoupCleanUpEnv(); MPI_Finalize();
}
```

**Figure 2.7: Parallel sparse conversion - skeleton**

the same program based on processor ranks. To this end, FoxBox provides a *MPI black box server object* which has the ability to accept messages from parallel black boxes to construct/evaluate/destroy black box algorithms and common objects.

The combined code in figures §2.7 and §2.8 retrieves the distributed representation of a factor of a 4 by 4 symmetric Toeplitz determinant modulo a prime in parallel. Our code illustration imports the following: 1) wrapper/adaptors for SACLIB's rational arithmetic and NTL's fast modular polynomial arithmetic; 2) arithmetic conversion operators; 3) the symmetric Toeplitz common object library; 4) the pruning sparse conversion algorithm; 5) the MPI black box server; 5) parallel versions of the factors homomorphic map algorithm and the $n$ to 1 factor multiplexor. Within the FoxBox distribution, each parallel component header file ends with a `mpi.h` suffix.

As with our arithmetic wrappers/adaptors, the MPI standard requires an initialization and clean-up phase. The function calls `MPI_Init` and `MPI_Finalize` provide this functionality. Each executable image utilizes the MPI function `MPI_Comm_rank` to identify its process rank and `MPI_Comm_size` for determining the number of processes executing the program.

In this particular illustration we employ the parallel version of our sparse conversion application which consists of "driver" and "parallel subtask" portion (see figure §2.8). The driver portion of the code only executes on the processor which has rank `BBMPI_DRIVER_RANK` and performs the sparse conversion algorithm. The parallel subtask portion executes on each of the remaining processors and executes a copy of the MPI black box server object, namely `ServerMPI`. The sparse conversion algorithm dispatches construction/evaluation/shutdown requests to each available processor.

## 2.3 Underlying Software Architecture

In our C++ implementation, the specialization to particular choices of field type and polynomial algorithm occurs at compile time. This is accomplished by means of template and ordinary class definitions. Classes provide important mechanism for localizing code and channeling access through a well defined interface.

```
if ( MyRank == BBMPI_DRIVER_RANK ) {
   // initialize SACLIB and Shoup wrapper/adaptors
   Word Stack; int i;
   SaclibInitEnv( 1000000, Stack );

   // construct a factors black box over Q
   int Seed = 103069; double Prob = -1.0;
   BBSymToeDetQ  SymToeDetQ( N, DegDet );
   BBFactorsQ FactorsQ( SymToeDetQ, Prob, Seed, &MPCard );

   // map the factors black box to Shoup's modular arithmetic
   ShoupInitEnv( &MPCard );
   SaclibQShoupZP   h;
   BBSymToeDetZP    SymToeDetZP( N, DegDet );
   BBFactorsQMapZP  FactorsZP( SymToeDetZP, FactorsQ, h );
   SaclibCleanUpEnv();

   // interpolate the first factor in parallel
   BBFactorZP            FirstFactorZP( FactorsZP, 0 );
   BBSPARSEAUX_VarVector Vars; BBSPARSEAUX_DegVector Degs;
   ShoupZPX              ShoupZPXElm[BBSPARSEAUX_MAXTERMS];
   BBSPARSEAUX_VecDegVector AnsDegs; bbvector< ShoupZP > AnsMons;
   int IsRestart = BBDEFS_FALSE; char* ProbName = NULL;
   for( i=1; i<=N; i++ ) { Vars.push_back(i);
      Degs.push_back(DegDet); }
   SparseInterp( FirstFactorZP, Vars, Degs, DegDet, &MPCard,
      AnsDegs, AnsMons, ShoupZPXElm, ProbName, IsRestart,
      NumProc );
}
else {
   ShoupInitEnv( &MPCard );
   BBSymToeDetZP SymToeDetZP( N, DegDet );
   sprintf( FileName, "%dx%dtoesacQXshoupZPX", N, N );
   BBServerMPI ServerMPI( BBMPI_DRIVER_RANK, MyRank, SymToeDetZP,
                        FileName, BBMPI_CLOCK_ON );
}
```

**Figure 2.8: Parallel sparse conversion - driver/subtask**

Since a class is a type in C++, local instantiations can be automatic without the use of free store or garbage collection. This allows circumventing heap allocation, the most common performance bottleneck in real systems.

The black box common objects and algorithms are realized as template function objects. Ordinary class definitions are utilized to wrap/adapt a particular polynomial algorithm or field type along with its corresponding access operations. The native arithmetic wrapper/adaptor is passed as a template argument to our black box objects. In C++, we can syntactically indicate which types are parameters. However, the language does not provide a mechanism capable of expressing the requirement that the types supply certain operations. Instead, in §2.3.1 we provide operation prototypes along with their corresponding informal semantic descriptions with an emphasis on implementation in C++.

In §2.3.2 we detail the interface specification defined by our so called *black box abstraction*. Each black box common object and algorithm supplies an implementation for the black box specification and may build on this interface by adding several custom operations.

### 2.3.1 Native Arithmetic Wrappers/Adaptors

By expressing the black box common objects and algorithms in terms of polynomial and field access operations, we permit a single expression of a black box object to be utilized with any concrete base arithmetic. Native base arithmetic packages are wrapped and adapted [GHJV94] to express a particular polynomial algorithm or field type along with its corresponding access operations. The black box common objects and algorithms are instantiated by utilizing arithmetic wrapper/adaptor classes. Such flexibility implies that the black box objects have a broader utility:

- End users can employ black box common objects and algorithms while remaining in a “familiar” setting with respect to a particular arithmetic package.
- Efficient and dedicated arithmetic packages can be utilized for specialized applications of the black box algorithms.

- By taking advantage of inlining, a C++ implementation is possible without overhead.

Since inlining is purely an optimization, it should be used only when the benefit in run-time or space outweighs the cost and inconveniences imposed by its use. The ideal candidate for inlining is a function that performs a simple action such as returning a value, incrementing a value, or calling another function. Such functions proliferate where data hiding is used, as in our black box objects and our arithmetic wrapper/adaptor classes. Care must be taken when using inline functions. The definition of an inline function has to be available for inlining to be done. This implies that all callers of an inline function must be recompiled if its definition changes. The usual function call implementation protects users against that. In our final analysis, the performance gained by careful inlining outweighs the inconveniences.

The components within FoxBox utilize GNU MP for internal arbitrary precision integer arithmetic. All arithmetic wrapper/adaptors are expected to convert a GNU MP integer to the format of the underlying base arithmetic. A base arithmetic which requires an explicit initialization/clean-up phase should encapsulate these operations by providing corresponding functions. Such initialization/clean-up functions should be invoked before and after a FoxBox application.

The exact operations required by an arithmetic wrapper/adaptor depends on each particular black box object. In what follows, we provide an exposition of the requirements for the field and polynomial arithmetic wrapper/adaptor classes that are common to all black box objects. We also highlight those operations which are specific to a particular black box algorithm. As an illustration of our specification, §2.3.1.1 and §2.3.2 also provide the actual class definitions for our arithmetic wrapper/adaptor of SACLIB's rational univariate polynomial arithmetic.

#### 2.3.1.1 Field arithmetic

All black box objects are parameterized by a field wrapper/adaptor. A field wrapper/adaptor encapsulates a native representation of a field element as a C++ class (user defined type), as well as provides the set of operations that can be applied to such types.

## Constructors

The black box objects utilize the default constructor when allocating arrays of field elements. In C++, no initializers can be specified for arrays. Arrays of objects of a class with constructors can be created by the C++ operator `new` only if the class has a default constructor. In this case, the default constructor will be called for each element of the array. Hence, we require that a field wrapper/adaptor provide a default constructor.

A native arithmetic wrapper/adaptor is required to provide a constructor for the purpose of creating field types from a pointer to a GNU MP integer. Components within FoxBox do not directly access native format field elements. However, a constructor which instantiates a field wrapper/adaptor from a base arithmetic element may be beneficial for implementation purposes, but certainly this constructor is not required.

The black box algorithms utilize a C-string in base 10 representation of a native format field element for archival, interprocess communication, and tracing purposes. A field wrapper/adaptor is expected to construct an instance from a base 10 C-string representation.

A class object can be copied in two ways, by assignment and by initialization including function argument passing and function value return. In C++, these two operations are implemented by an assignment operator and a copy constructor. The utility of copy constructors spans to the situation in which a compiler instantiates temporary objects. If these member functions are not defined by the programmer, they will be defined as memberwise assignment and memberwise initialization. Such a scenario can be problematic for some base arithmetics. It is expected that the field wrapper/adaptor handle such situations gracefully.

The C++ destructor provides a method of destroying values in class types immediately before the object containing them is destroyed. When implementing a C++ field wrapper/adaptor, it is often necessary to reverse the effect of a construction allowing for the freeing of system resources. For this reason, we include a field wrapper/adaptor destructor in the specification.

## Operators and Public Member Functions

Operators are used to provide notational convenience. C++ allows for the overloading of function names and operators. The black box objects utilize a subset of the C++ operators that are expected to manipulate field wrapper/adaptors. Table §2.1 details a list of the required operators.

| Operator | Description |
|---|---|
| $=$ | Assignment |
| $==$ | Test for equality |
| $!=$ | Test for inequality |
| $<<$ | Output |
| $+$ | Addition |
| $-$ | Subtraction |
| $*$ | Multiplication |
| $/$ | Division |

**Table 2.1: Field wrapper/adaptor operators**

Table 2.2 depicts each public member function a FoxBox component may require. The member functions `DistinctSelect`, `Card`, and `toString` are required by all black box algorithms. The `Exp` member function is utilized by the symmetric Toeplitz black box, factors black box and the sparse conversion algorithm. Finally, `Denom` and `LCM` provide the necessary functionality for the numerator and denominator black box.

All black box algorithms require a function, `Card` that provides the field cardinality. In the case of an infinite field, a value of `-1` should be returned.

The black box algorithms perform univariate interpolation on a transformed version of the input black box(es). This interpolation requires the use of a “stream” of distinct field elements. On invocation, it is expected that the member function `DistinctSelect` return a distinct field element. An initial assignment to a field wrapper/adaptor provides the starting point from which distinct elements are chosen.

| Member Function | Description |
|---|---|
| `DistinctSelect` | All black box algorithms |
| `Card` | All black box algorithms |
| `toString` | All black box algorithms |
| `Exp` | Factor, SparseInterp, Toeplitz |
| `Denom` | NumDen |
| `LCM` | NumDen |

**Table 2.2: Field wrapper/adaptor public member functions**

As previously explained, the black box algorithms convert between a terminated C-string in base 10 to a native format. The member function `toString` populates a supplied character array with the string base 10 representation of the value of the encapsulated native field element.

The `Exp` member function raises the value of the encapsulated field element to the power of a supplied integer. The `Denom` function returns the denominator of the encapsulated field element. Finally in the case where the `LCM` is unique up to an associate, the `LCM` member function returns the least common multiple of an arbitrary number of elements. An array of elements and the number of elements in that array are supplied as arguments to the `LCM` function.

### Random Iterator

Many of the black box algorithms utilize randomization. The algorithms are Monte-Carlo in that their output, the constructed evaluation programs, are correct with controllably high probability (see §3). For this purpose, a random number iterator `RandomIterator` class with constructor `RandomIterator(const int Seed, const MP_INT* Card)` must be provided by the field wrapper/adaptor. The constructor generates a random iterator object which provides random field elements from a set of cardinality `MPCard` seeded at `Seed`. An overloaded function call operator supplies random field elements. It is expected that identical sets of random numbers are generated by different runs with the same seed and cardinality parameters, providing consistency between runs.

### Example

Figure §2.9 provides an example of an efficient C++ arithmetic wrapper/adaptor to SACLIB's rational number arithmetic. It is intended that this example aid in the exposition of the exact arguments and return values of each operator and member function outlined in the field wrapper/adaptor specification. As noted in the introduction of this section, a native field and polynomial arithmetic wrapper/adaptor can be implemented in C++ without overhead by utilizing inlining. The ideal candidate for inlining is a function that performs a simple action such as returning a value, incrementing a value, or calling another function. Such behaviors are exhibited in our implementation of SACLIB's native arithmetic wrappers/adaptors.

#### 2.3.1.2 Polynomial arithmetic

Several of the components implemented within FoxBox are parameterized by a univariate polynomial algorithm. As in the case of the field wrapper/adaptor, polynomial wrapper/adaptors are implemented as a C++ class upon which instantiation contains or refers to a polynomial in some native format. It is expected that a particular polynomial wrapper/adaptor provide the set of functions and operators that can be applied to such user defined types.

**Constructors**

As with the field wrapper/adaptor, we require the following: 1) a default constructor for the allocation of arrays of polynomial elements; 2) an assignment operator and a copy constructor for copying polynomial wrapper/adaptor objects; and 3) a destructor which provides a method of deallocating a native form univariate polynomial element immediately before the polynomial wrapper/adaptor is destroyed.

Several FoxBox components create univariate monomials from a field wrapper/adaptor object and degree or simply create the constant polynomial from a field wrapper/adaptor object. Hence the appropriate constructors must be furnished. Finally, a constructor which instantiates a monomial from a native arithmetic field element may be beneficial for implementation purposes, but is not required for any of the FoxBox components.

```
class SaclibQ {
private:
 Word SaclibQElm;
public:
 friend SaclibQX;
 inline friend ostream& operator<<( ostream&, const SaclibQ& );

 SaclibQ( );
 SaclibQ( const MP_INT*);
 SaclibQ( const char* );
 SaclibQ( const SaclibQ& );
 SaclibQ( const Word& SaclibQElm_i );
 ~SaclibQ( );

 inline SaclibQ&   operator=( const SaclibQ& );
 inline friend int operator==( const SaclibQ&, const SaclibQ& );
 inline friend int operator!=( const SaclibQ&, const SaclibQ& );

 inline SaclibQ operator+(const SaclibQ&);
 inline SaclibQ operator-(const SaclibQ&);
 inline SaclibQ operator*(const SaclibQ&);
 inline SaclibQ operator/(const SaclibQ&);

 inline SaclibQ DistinctSelect( );
 inline MP_INT  Card( );
 inline SaclibQ Exp( int );
 inline SaclibQ Denom( );
 inline SaclibQ LCM( SaclibQ*, int );
 inline void    toString( char* );

 class RandomIterator {
    public:
     RandomIterator( const int, const MP_INT* );
     ~RandomIterator( );
     SaclibQ operator()( );
     private:
      long RandStateArray[32];
      int RandSeed;
      MP_INT MPCard;
      Word SLCard; };
};
```

**Figure 2.9: SACLIB rational number wrapper/adaptor**

### Operators and Public Member Functions

The black box objects utilize a subset of the C++ operators that are expected to manipulate polynomial wrapper/adaptors. Table §2.3 is a list of the required operators.

| **Operator** | **Description** |
|---|---|
| $=$ | Assignment |
| $==$ | Test for equality |
| $!=$ | Test for inequality |
| $[]$ | Extract the coefficient of $x^n$ |
| $<<$ | Output |
| $+$ | Addition |
| $-$ | Subtraction |
| $*$ | Multiplication |
| $/$ | Exact Division |

**Table 2.3: Polynomial wrapper/adaptor operators**

Table 2.2 depicts each public member function a FoxBox component may require from a polynomial arithmetic wrapper/adaptor. The member function `Eval` is required by all black box algorithms. The factor black box requires the functionality offered by `Deg`, `Rem`, `QuoRem`, `GCD`, `EEA` and `Factors`. The greatest common divisor black box requires `Deg`, `LCoeff`, `GCD`. The numerator and denominator black box depends on `Deg`, `LCoeff` and `QuoRem`. The algorithm which solves the Vandermonde system that arises in our sparse conversion algorithm employs the `Diff` member function. Finally, the symmetric Toeplitz common object employs `Deg`, `LCoeff` and `Rem` for producing an evaluation.

The `Eval` member function produces the value of the encapsulated univariate polynomial at a specified point. The argument for `Eval` is a pointer to a field wrapper/adaptor object which remains consistent with the evaluation specification detailed by the black box abstraction (see §2.3.2). The `Deg` `LCoeff`, and `Diff` functions returns the degree, the leading coefficient and derivative of its encapsulated native polynomial respectively.

| Member Function | Object |
|---|---|
| `Eval` | All black box algorithms |
| `Deg` | Factor, GCD, NumDen, Toeplitz |
| `LCoeff` | GCD, NumDen, Toeplitz |
| `Rem` | Factor, Toeplitz |
| `QuoRem` | Factor, NumDen |
| `GCD` | Factor, GCD |
| `EEA` | Factor |
| `Factors` | Factor |
| `Diff` | SparseInterp |

**Table 2.4: Polynomial wrapper/adaptor public member functions**

The polynomial quotient and remainder of two polynomial wrapper/adaptor objects populate the last two arguments of the `QuoRem` member function. The polynomial remainder function `Rem` returns the remainder of two polynomial wrapper/adaptor objects via its third argument. Clearly this member function can interface to the same native arithmetic function which implements `QuoRem`. Similarly, the `GCD` member function inserts the greatest common divisor of its first two arguments into its third argument. The factorization black box object requires an extended Euclidean algorithm for computing partial fractions. Given two polynomial objects $a$ and $b$, the `EEA` member function computes $g$, $s$, and $t$ such that $g = \mathrm{GCD}(a, b) = a * s + b * t$. Finally, the `Factors` member function performs the univariate factorization of an input polynomial object and returns the integer content of the input polynomial, an array of factors, and an array of exponents via its argument list. The reader is referred to figure §2.10 for more information regarding the exact prototype for each operator and public member function.

A polynomial wrapper/adaptor which implements all of the specifications detailed above will provide the necessary functionality for each FoxBox component. However, a polynomial wrapper/adaptor supplying a subset of the specified functionality can be utilized in several situations since the particular choice of algorithm dictates the minimal interface required. For instance, the black box GCD algorithm requires a univariate GCD function and does not employ a univariate factorization routine. Hence, an arithmetic lacking polynomial factorization may be furnished to the GCD algorithm without consequence.

Actually, the amount of functionality required by an arithmetic wrapper/adaptor is minimized by providing template classes that build on the supplied functionality. Our internal support template classes, like adapter classes, convert the interface of a polynomial wrapper/adaptor into another interface. Furthermore, our support wrappers add functionality by providing an implementation based on the selected polynomial arithmetic. For example, given a univariate polynomial arithmetic which provides univariate factorization, we deliver a template wrapper which implements the truncated bivariate polynomial arithmetic and specialized Hensel lifting algorithm required by the black box factorization algorithm (see §3.2).

#### Example

Figure §2.10 provides an example of an efficient C++ arithmetic wrapper/adaptor to SACLIB's univariate polynomial arithmetic with rational coefficients. As in the field wrapper/adaptor example, the intention of this example is to aid in the exposition of the exact arguments and return values of each operator and member function outlined in the previous specification.

### 2.3.2 The Black Box Abstraction

The FoxBox class library supplies a C++ base class, namely `BlackBox`, which implements a framework for the black box abstraction. All of FoxBox's black box objects are derived from this black box base class. Consequently, each black box object furnishes at the very least an interface which adheres to the specification detailed in figure §2.11. The utilization of a base class to express commonality among black box objects allows the implementation of algorithms that can manipulate black boxes independently of type. Hence, such a framework should be utilized for user defined black box objects.

In some circumstances, it may be necessary or convenient for a compiler to generate a temporary object. In C++, when a compiler instantiates a temporary object, it must call the object's corresponding constructor and destructor. By providing a copy constructor, black box objects are able to properly handle such situations.

```
class SaclibQX {
private:
 Word SaclibQXElm;
public:
 inline friend ostream& operator<<( ostream&, const SaclibQX& );

 SaclibQX( );
 SaclibQX( const Word& );
 SaclibQX( const Word&, const  int& );
 SaclibQX( const SaclibQ&, const  int&  );
 SaclibQX( const SaclibQ& );
 SaclibQX( const SaclibQX& );
 ~SaclibQX( );

 inline SaclibQX& operator=( const SaclibQX& );
 inline friend int operator==( const SaclibQX&, const SaclibQX& );
 inline friend int operator!=( const SaclibQX&, const SaclibQX& );

 inline SaclibQX operator*( const SaclibQX& );
 inline SaclibQX operator/( const SaclibQX& );
 inline SaclibQX operator+( const SaclibQX& );
 inline SaclibQX operator-( const SaclibQX& );

 inline SaclibQ  operator[] ( int );
 inline int      Deg( );
 inline SaclibQ  Eval( const SaclibQ* );
 inline SaclibQ  LCoeff( );
 inline void     Rem( const SaclibQX&, const SaclibQX&, SaclibQX& );
 inline void     QuoRem( const SaclibQX&, const SaclibQX&,
                         SaclibQX&, SaclibQX&);
 inline void     GCD( const SaclibQX&, const SaclibQX&, SaclibQX& );
 inline void     EEA( const SaclibQX&, const SaclibQX&, SaclibQX&,
                      SaclibQX&, SaclibQX& );
 inline void     Factors( const SaclibQX&, int&, SaclibQ&,
                          unsigned int&, unsigned  int*, SaclibQX* );
 inline SaclibQX Diff( );
};
```

**Figure 2.10: SACLIB univariate rational polynomial wrapper/adaptor**

```
template< class K >
class BlackBox {
   protected:
      int          Degree;
      unsigned int NumVariables;
      double       Probability;
   public:
      BlackBox( );
      ~BlackBox( ) { }
      virtual inline K* operator() ( K* ) { }
      inline unsigned int Deg( );
      inline unsigned int NumVars( );
      inline double Prob( );
};
```

**Figure 2.11: Black box abstract base class**

An overloaded function call operator is utilized for black box evaluation. The black box object takes as input a value for each variable as an array of field elements and then produces the value of the symbolic object it represents at the specified point. The interpretation of this return value depends on the particular black box object. For example, a black box polynomial common object returns a C++ pointer to a field element representing the value of the polynomial at that point. On the other hand, a black box representing the factors of a polynomial returns a pointer to the first element of an array of values, one value for each factor. In the latter, the interface remains consistent by virtue of using pointers. The return value can be used as the value of the first factor or manipulated via pointers to access the values of the remaining factors.

| `BlackBox::Degree()` | |
|---|---|
| Value | Interpretation |
| $< 0$ | degree not known |
| $\geq 0$ | exact or upper bound |

**Table 2.5: Return Values for `BlackBox::Degree()`**

Each black box object requires a function that provides the degree of its corresponding symbolic object. Table 2.5 provides a description of the two possible

```
template< class K, class L >
class BlackBoxEx : public BlackBox< K > {
   public:
      BlackBoxEx( ) : BlackBox< K >( ) { }
      ~BlackBoxEx( ) { }
      virtual inline L* operator() ( L* ) { }
};
```

**Figure 2.12: Extended domain black box abstract base class**

ranges a degree value may hold.

Our black box algorithms can be implemented without a dependency on variable names. The only requirement is a knowledge of the number of variables. The function `BlackBox::NumVars()` provides this functionality.

The probability of the correctness of a black box evaluation program can be retrieved by the `BlackBox::Prob()` function. The return value is a float between 0 and 1. This value can be used by the programmer to measure the reliability of a computed result. Please refer to §3 for a mathematical description of the estimate of success corresponding to each black box algorithm.

The FoxBox class library supplies a C++ base class `BlackBoxEx` (figure §2.12) which implements a framework for the extended domain black box abstraction. Extended domain black boxes can evaluate at points not only from a particular field but also at points that come from a domain which is an extension of the field over which it was constructed. All of FoxBox's extended domain black box objects are derived from this black box base class. This class extends the `BlackBox` class by adding a function object which takes as input a value for each variable as an array of extended domain elements. Its return value is a pointer to the value of the symbolic object it represents at the specified point.

By utilizing pointers or references to the base black box class, the black box algorithms may access any of the derived black box classes in a uniform way. In the case that the exact type of an object is known at compile time, the virtual function call mechanism need not be used. Instead, an inline member function call can be used for efficiency. A direct call is a few memory references more efficient than a virtual function [ES95], but the real advantage of direct calls is for inline functions where the difference in run-time overhead can be significant. This allows the circumvention of the virtual function call mechanism at a cost of potential code bloat.

In what follows we provide the implementation notes behind our black box specification for each FoxBox component. This portion will also describe any extended functionality a particular FoxBox component may offer. The reader is referred to the C++ FoxBox library source code [DK97a] for a more detailed explanation.

#### 2.3.2.1 Common Black Box Objects

Table 2.6 depicts the current set of common objects found in the FoxBox library. Each common object provides an implementation for the functionality imposed by the black box abstraction. These particular common objects do not offer additional functionality over the black box abstraction. Thus, only the construction and evaluation operations will be detailed.

| Template Prototype | |
|---|---|
| Common Objects | Extended Domain Common objects |
| `BlackBoxPoly< K >` | `BlackBoxPolyEx< K, L >` |
| `BlackBoxRatFunc< K >` | `BlackBoxRatFuncEx< K, L >` |
| `BlackBoxDet< K >` | `BlackBoxDetEx< K, L >` |
| `BlackBoxSymToeDet< K, KP >` | `BlackBoxSymToeDetEx< K, KP, L, LP >` |
| `BlackBoxVandDet< K >` | |
| `BlackBoxCauchyDet< K >` | |

**Table 2.6:** FoxBox **common object library**

The first template parameter of each common object is a coefficient domain (`K`). Extended domain common objects are also parameterized by an extension domain (`L`). The constructors for `BlackBoxPoly` and `BlackBoxRatFunc` have the same form.

```
typedef K ( *FPTRK )( K* );
BlackBoxPoly( FPTRK PolyK, int NumVars, int Deg, double Prob );
BlackBoxRatFunc( FPTRK RatFK, int NumVars, int Deg, double Prob );
```

The first argument is a function which is utilized for point generation, the second indicates the number of variables, the third a degree bound and the fourth a probability of correctness. In the case of the black box polynomial, the degree parameter sets the total degree. This same parameter is utilized to indicate a bound on the total degree of the denominator of the black box rational function. The confidence that an evaluation is correct is expressed by a value between 0 and 1.

In the extended domain case, both the constructors for `BlackBoxPolyEx` and `BlackBoxRatFuncEx` require an additional function. Depending on a particular evaluation domain a corresponding input function will be employed for point generation.

```
typedef K ( *FPTRK )( K* );
typedef L ( *FPTRL )( L* );
BlackBoxPolyEx( FPTRK PolyK, FPTRL PolyL, int NumVars, int Deg,
   double Prob );
BlackBoxRatFuncEx( FPTRK RatFK, FPTRL RatFL, int NumVars, int Deg,
   double Prob );
```

New implicit representations as well as external libraries can easily be introduced into FoxBox via this method. For example, an external C function could serve as a wrapper which links to a Maple routine. Such a routine could evaluate a determinant of a particular type of matrix.

FoxBox provides a domain independent Gaussian elimination algorithm which can be utilized for computing the determinant of a matrix. The `BlackBoxDet` and `BlackBoxDetEx` common objects employ Gaussian elimination for evaluation. Constructing a black box determinant common object requires similar arguments to those described above.

```
typedef void ( *FPTRK )( K*, K [BBDET_MAXN][BBDET_MAXM] );
typedef void ( *FPTRL )( L*, L [BBDET_MAXN][BBDET_MAXM] );
BlackBoxDet( FPTRK FillMatrixFuncK, int NumVars, int Deg,
   double Prob, int n, int m );
BlackBoxDetEx( FPTRK FillMatrixFuncK, FPTRL FillMatrixFuncL,
    int NumVars, int Deg, double Prob, int n, int m );
```

The main difference lies in the input function which is intended to populate an array with elements from a particular domain. This array represents a matrix from which the black box computes the determinant for evaluation. The dimension of this matrix is provided by the `n` and `m` parameters. As an example, a "matrix filling" function could map each value from an array of points $P$ to a square matrix which has as its $(i, j)$-th entry $P[i]^{(j-1)}$. In this case, the black box evaluates the determinant of a Vandermonde matrix.

FoxBox provides several common objects which can take advantage of a particular matrix structure (Cauch, Toeplitz, Vandermonde) to provide a specialized algorithm for the determinant computation. In fact, extensibility to other implicit representations can be achieved by importing C++ user defined common objects which derive from either the `BlackBox` or `BlackBoxEx` abstract base classes.

Our symmetric Toeplitz determinant common object utilizes a technique based on subresultant computations for the determinant computation. Hence, the symmetric Toeplitz determinant common object employs an extra template parameter which supplies a polynomial algorithm. The constructors

```
BlackBoxSymToeDet( int NumVars, int Deg );
BlackBoxSymToeDetEx( int NumVars, int Deg );
```

produce an $n$ by $n$ symmetric Toeplitz determinant common object where $n$ is specified by the number of variables. In the case of the extended domain symmetric Toeplitz common object, a template parameter which supplies polynomial arithmetic for the extension domain (`LP`) is also required.

The constructor

```
BlackBoxCauchyDet( int NumVars, int Deg );
```

generates a $n$ by $n$ Cauchy determinant common object. In this case, $n$ is the number of variables divided by 2.

Our black box Vandermonde determinant common object provides a constructor for building Vandermonde objects customized for our GCD challenge problem (see §4.2).

```
BlackBoxVandDet( int NumVars, int Deg, int StartIndex,
   int EndIndex );
```

From this interface, one can construct a Vandermonde common object which utilizes only a subset of the input points for evaluation. For example, one can construct a Vandermonde common object which can evaluate given points $(x_1, x_2, x_3, x_4)$, but only utilizes $(x_1, x_2)$ to compute its determinant. The index delimiters indicating which "window" of variables to employ for evaluation is specified by the `StartIndex` and `EndIndex` parameters.

#### 2.3.2.2 Black Box Algorithms

Table 2.7 details template prototypes for the black box algorithm objects available within the C++ FoxBox library. Each algorithm black box object is derived from either the `BlackBox` or `BlackBoxEx` base classes. This section will supply interface details for each algorithm black box object, as well as describe our sparse interpolation implementation which converts black boxes representing polynomials to a distributed sparse format.

| Template Prototype | |
|---|---|
| Algorithms | Extended Domain Algorithms |
| `BlackBoxFactors< K, KP, B >` | `BlackBoxFactorsEx< K, KP, B, L, LP >` |
| `BlackBoxGCD< K, KP, B >` | |
| `BlackBoxNumDen< K, KP, B >` | |

**Table 2.7:** FoxBox **black box algorithm library**

The first template parameter of each algorithm object is the coefficient domain (`K`), the second is a polynomial algorithm (`KP`), and the third a black box type (`B`). For our extended domain factorization algorithm, we also require extension domain arithmetic (`L` and `LP`).

Each black box algorithm object utilizes a particular set of static information for evaluation. This static information uniquely describes a particular black box instance. Utilizing this "signature", one can archive constructed black boxes. To this end each black box algorithm provides overloaded input and output operators for file input (`ifstream`) and file output (`ofstream`). Each black box instance file is tagged identifying a particular black box algorithm.

Common arguments utilized by the constructor of each black box algorithm objects include a seed, a probability measure and a cardinality. The seed value is utilized by the native arithmetic wrapper/adaptor random field element generator. This value enables control of the stream of random elements which can be utilized to provide consistency between runs. The cardinality of the set from which to choose random field elements is determined by a probability measure between 0 and 1. Given a probability measure, each particular black box algorithm computes the corresponding cardinality and returns this cardinality via a supplied argument. In another mode of operation, by supplying a value of $-1$ as the probability measure, random field elements are selected from a field of predetermined cardinality supplied as a parameter. The probability measure associated with this cardinality is computed and consequently returned via a constructor argument.

It should be noted that all three black box algorithms evaluate one and the same associate (scalar multiple) of the goal polynomial (see §3). In what follows is a description of the remaining aspects of each black box algorithm found in the FoxBox C++ library.

**Factorization**

The black box factorization algorithm takes a black box for a multivariate polynomial and produces a black box object that will evaluate all individual irreducible factors.

```
BlackBoxFactors( B& BBPoly, double& Prob, int Seed, MP_INT* Card );
BlackBoxFactorsEx( B& BBPoly, double& Prob, int Seed, MP_INT* Card );
```

The construction phases for both of the above black box factorization algorithms are identical. Also, both algorithms provide a function object which requires as input a value for each variable. This function object returns a pointer to the first element of an array of field elements representing the values of each factor. However, the extended domain factors black box employs a different method of evaluation.

Furthermore, the extended domain factors black box probes the input black box at points that are from a domain which is an extension of the field over which it was constructed.

Both the factor black box and its extended domain version augment the black box abstraction by providing member functions `GetNumFacs` and `GetExp` for retrieving the number of factors and factor exponents respectively.

### Greatest Common Divisor

Our black box greatest common divisor algorithm constructs a black box representing the GCD of an arbitrary number of black box polynomials.

```
BlackBoxGCD( B** BBArray, int NumBBs, MP_INT* Card, double& Prob,
   int Seed );
```

Each black box object is supplied to the greatest common divisor black box constructor as an array of pointers to black box objects. The catalyst for this design was the need to compute the black box greatest common divisor of a heterogeneous set of black box objects. A reference to a derived class may be implicitly converted to a reference to an accessible base class. In the case that `B` is a black box base class, heterogeneity can be achieved by the standard conversion from a derived class pointer to a base class pointer.

The black box greatest common divisor extends the interface specified by the black box abstraction by providing the public member function `GetNumBlackBoxes` for accessing the number of input black boxes.

### Numerator and Denominator Separation

The black box numerator and denominator algorithm computes the values of the numerator and the denominator of a rational function black box.

```
BlackBoxNumDen( B& BBRatFunc, MP_INT* Card, int DenBound,
   double& Prob, int Seed );
```

Without the knowledge of a degree bound for the denominator, the construction phase may loop forever (see §3.4). An overloaded function operator provides for evaluation. The return value is a pointer to an array of two field elements. The first represents the value of the numerator and the second the value of the denominator.

The black box numerator and denominator provides the public member functions `GetDegNum` and `GetDegDen` for accessing the degrees of the numerator and denominator respectively.

The reader is referred to §5.2 where we discuss plans to provide an extended domain version of this algorithm. The performance of the Cauchy challenge problem (see §4.3) will improve dramatically by probing the numerator/denominator black box at truncated power series.

### Sparse Conversion

Our sparse conversion implementation is parameterized by a coefficient domain (`K`), polynomial algorithm (`KP`) and black box type (`B`).

```
template< class K, class KP, class B >
void SparseInterp( B& BBPoly, vector< int > Vars, vector< int > Degs,
   int TotalDeg, MP_INT* Card, vector< vector< int > > AnsDegs,
   vector< K > AnsMons, KP* PolyArray, char* ProbName,
   int& IsRestart, int NumProc = 0 );
```

The algorithm converts a black box representing a multivariate polynomial into a distributed sparse format. This is achieved by utilizing STL vectors for maintaining monomials and degree information. Our sparse conversion algorithm interpolates one variable at a time. In fact its performance can depend on the order in which each variable in interpolated (see §3.1). Furthermore, the algorithm reduces the number of black box evaluations by a term pruning technique (see §3.1.1). This technique utilizes degree bounds for each variable and a bound on the total degree (`TotalDeg`). The user specifies a variable order by supplying a vector of variable identifiers (`Vars`) and a corresponding vector of degree bounds (`Degs`). The algorithm probes the black box by utilizing random elements from a set of cardinality

`Card`. By "setting" the `IsRestart` flag, our sparse conversion algorithm will save "check pointing" information in a file indicated by `ProbName`. This mechanism ensures fault tolerance by securing interpolation state information on disk. Our sparse conversion algorithm solves a transposed Vandermonde system of equations by utilizing univariate polynomial arithmetic. The `PolyArray` argument solely serves as a mechanism of specifying a polynomial algorithm. The `NumProc` argument indicates the number of processors available processors for parallel black box evaluation.

#### 2.3.2.3 Homomorphic Maps

In FoxBox, computing the homomorphic image of a black box object is a generic algorithm which employs a supplied mapping function to convert between coefficient domains. The result of such a mapping is another black box which evaluates over 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. All evaluations of the mapped black box occur modulo a prime. FoxBox provides a homomorphic map for all of the black box algorithm objects (see Table 2.8).

| 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 2.8:** FoxBox **homomorphic map library**

Each homomorphic map is parameterized by a field arithmetic (`K` and `KP`) and an arithmetic modulo a prime (`F` and `FP`). A "mapping" function object `H` is utilized to convert between coefficient domains. Conversion between polynomial algorithms is accomplished by utilizing the supplied mapping function. The parameters `B` and `FB` specify input black boxes, the first of which evaluates utilizing arithmetic specified by `K` and and the second employing arithmetic specified by `F` for evaluation. The result of a homomorphic image of a black box algorithm computes its value by probing `FB` at values modulo a prime. If `FB` is not available, FoxBox supplies a class, namely `BlackBoxMod< K, F, B, H>`, which will wrap/adapt a black box so as to provide values modulo a prime. The mapping function object `H` is employed to convert the input to the original domain and the output to its modular representation. Modular wrapping is likely to be less efficient than homomorphic imaging.

Interoperability between distinct base arithmetics can be achieved by suppling the homomorphic imaging algorithm with an appropriate mapping operator. For instance, a black box can be constructed utilizing SACLIB's rational polynomial arithmetic and mapped to a new black box that evaluates by employing 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.

#### 2.3.2.4 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 using 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. Table 2.9 details each parallel black box object provided by FoxBox. Each class is derived from its black box algorithm counterpart 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-in to the functionality offered by parallel systems such as DSC [DHK$^+$95], PVM [GBD$^+$94], or MPICH [GLS94] simply by providing a corresponding derived class.

| Template Prototype |
|---|
| `BlackBoxFactorsMPI< K, KP B >` |
| `BlackBoxGCDMPI< K, KP B >` |
| `BlackBoxNumDenMPI< K, KP B >` |
| `BlackBoxFactorsHMapMPI< K, KP, F, FP, B, FB, H >` |
| `BlackBoxGCDHMapMPI< K, KP, F, FP, B, FB, H >` |
| `BlackBoxNumDenHMapMPI< K, KP, F, FP, B, FB, H >` |

**Table 2.9:** FoxBox **parallel black box library**

For instance, our DSC system distributes computations over many computers. The system currently runs on workstations with the Unix operating system communicating via the DARPA Internet standard IP/TCP/UDP protocols. DSC affords a heterogeneous approach to problem solving in the following sense: DSC supports the distribution of C, C++, Lisp and Maple code; it has been tested on several architectures; and it distributes over our local area network as well as over the Internet to off-site compute nodes. DSC hides processor scheduling from the user and provides mechanisms for automatic fault tolerance. Each member function described by the parallel black box interface can be implemented by employing API calls from the DSC C++ programming library. This would allow a parallel FoxBox application to take advantage of all of the aforementioned features.

The current set of objects provided by the FoxBox parallel black box library are realized via MPI (Message Passing Interface) compliant 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 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. In summary, MPI was designed for high performance on both massively parallel machines and on workstation clusters; is widely available, with both freely available and vendor-supplied implementations; and was developed by a broadly based committee of vendors, implementors, and users. 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 an MPI-compliant mechanism for parallelizing black boxes broadens FoxBox's applicability.

An 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`.

Drawing from our experiences with DSC, the parallel black box interface can be enhanced to provide additional functionality such as a choice of distribution strategies. These enhancements are a focus of future work detailed in §5.2.

## 2.4 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 the FoxBox C++ programming library.

Through the development of examples, §2.4.1 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. Finally, §2.4.2 details the design behind the FoxBox interface, which yields a fairly portable implementation.

### 2.4.1 Accessing the FoxBox Server

A C++ 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 C++ 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 and 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 which serves as a remote black box object identifier. The call to `BlackBoxFactors` creates a factors black box over the rationals which 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 C++ FoxBox programming library detailed in §2.3.2. The reader is referred to §2.3.2 and [DK97a] 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 (see §3.1).

```
> 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 which given integers and a prime 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.

As a final example, let us consider computing the greatest common divisor of two 3 by 3 Vandermonde matrices which have two variables in common.

```
> VandQ1  := BlackBoxVand( BBNET_Q, BBNET_VAND, 6, -1, 1.0 );
                                 VandQ1 := 5
> VandQ2  := BlackBoxVand( BBNET_Q, BBNET_VANDSHIFT, 6, -1, 1.0 );
                                 VandQ2 := 6
> GCDQ := BlackBoxGCD( BBNET_Q, [ VandQ1, VandQ2 ], Mod , 1.0,
          Seed );
                                  GCDQ := 7
> GCD  := SparseConversion( BBNET_Q, GCDQ, [x1, x2, x3, x4, x5, x6 ],
          [ 1, 1, 0, 0, 0, 0 ], 1, Mod );
                                GCD := - x1 + x2
FoxBoxCleanUp();
```

The FoxBox server Vandermonde determinant constructor provides for building Vandermonde determinant objects customized for our GCD challenge problem (see §4.2). The code detailed above constructs two Vandermonde determinant objects which evaluate by utilizing rational number arithmetic. The call to `BlackBoxGCD` returns an index to a newly constructed black box greatest common divisor of the previously instantiated Vandermonde determinant objects. Finally, the `SparseConversion` call performs a remote conversion of the black box greatest common divisor into a distributed representation.

While the current scope of the FoxBox server is to provide a calculator style interface to general purpose computer algebra systems, it is quite feasible to expand this notion to allow for the transparent exchange of symbolic objects given by black boxes between a FoxBox server and algebra system. In light of the fact that black box objects are characterized by a small amount of static data, the framework for a more sophisticated interaction between the FoxBox server and general purpose computer algebra systems is in place by virtue of the FoxBox server design (see §2.4.2). Such improvements are a focus of future work detailed in section §5.2.

### 2.4.2 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 [CDK94], 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" [ADS96] which translates (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.

Figure 2.13 provides an illustration of the different bands of the FoxBox server interface. Residing in band **A**, a small amount of system dependent code acts as a bridge between a computer algebra system (eg. Axiom, Maple, Mathematica) and a temporary process **T**. The temporary process in band **B** makes a request to a local daemon process **D** which in turn forwards the request to the FoxBox server which resides in band **C**. The temporary process **T** blocks the execution of the application program running on the algebra system until the desired result has been received. In an ideal situation the FoxBox server will execute on a compute node with a fast processor and a considerable amount of memory, essentially providing a service for many computer algebra "front ends".

The transfer of information between processes in bands **A** and **B** employs a

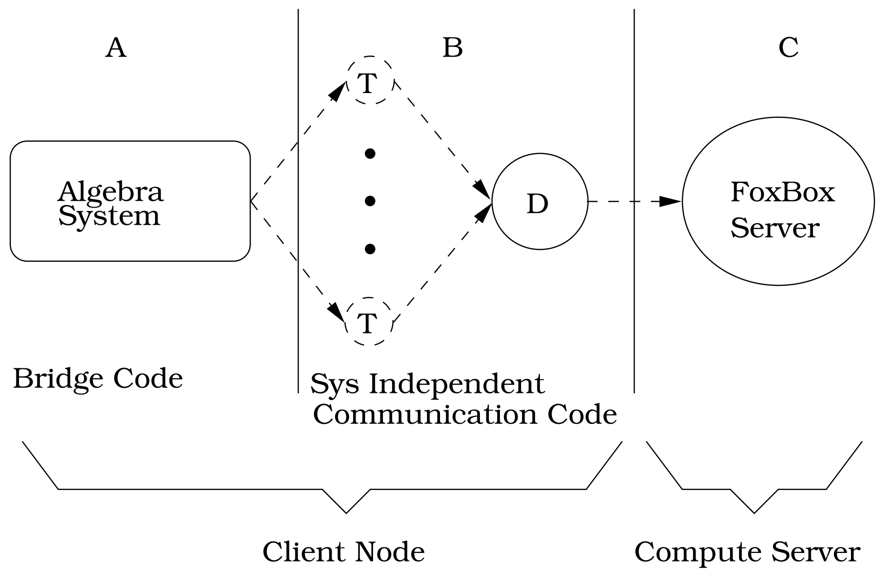


**Figure 2.13: Bands of the FoxBox server interface**

file based communication utilizing a specialized ASCII protocol. OpenMath aims to provide a complete protocol for the exchange of mathematical expressions in order to become the primary standard for the exchange of such information. OpenMath specifies a mapping between mathematical concepts and their concrete representations by defining so-called content dictionaries. Each content dictionary embodies semantic information from a single area of mathematics. In the same way that some areas of mathematics build on others, a content dictionary may depend on others. Our hopes are that the final version of OpenMath will allow for the introduction of a "black box" content dictionary hence broadening FoxBox's applicability. Such a content dictionary will further reduce the amount of bridge code required to interface to general computer algebra systems.

# CHAPTER 3
# THE BLACK BOX ALGORITHMS

We now provide an exposition of several algorithms for manipulating multivariate polynomials and rational functions given by black boxes for their evaluations. In §3.1, we provide a full description and illustration of our sparse interpolation algorithm which employs a novel term pruning technique. This technique is aimed at reducing the number of black box evaluations necessary to convert a black box polynomial into its distributed representation. In fact, our term pruning technique is also applicable to a special bi-variate polynomial interpolation which arises in our factor box evaluation algorithm. §3.2 outlines the aforementioned optimization and concomitantly describes the notion of extended domain black boxes which dramatically improves evaluation performance for small degree factors. In §3.3, we provide a detailed description and analysis of our algorithm for computing the greatest common divisor of polynomials in black box format. By employing a modular GCD approach, we improve on the black box GCD algorithm sketched in the literature. Finally, in §3.4, we describe a technique first observed in our GCD algorithm to markedly increase performance of the black box reduced numerator and denominator algorithm.

For clarity in our detailed algorithmic descriptions in §3.1 and §3.3, we shall state the actual computations in imperative mood and typeset them in italics font, while keeping the extensive comments in narrative and roman.

## 3.1 Sparse Conversion

We now discuss our version of Zippel's sparse interpolation algorithm for converting a polynomial from its' black box representation to a distributed format. The reader is referred to the sparse conversion algorithm presented by Kaltofen [Kal89] and the transposed Vandermonde linear system solver by Kaltofen and Lakshman [KL88]. Both are algorithms from which we base our modifications and hence we closely mirror their notational conventions and algorithmic description.

Zippel provides a probabilistic resolution to the interpolation problem. No information is given in regards to a bound on the number of terms. An estimate on the number of terms is reached as each new variable is introduced during the interpolation process. The algorithm is based on the assumption that during the task of interpolation many of the coefficients are zero and any zero coefficient is, with high probability, the image of a zero polynomial.

In FoxBox, the black box algorithm objects can become complex, because they are constructed from common objects such as matrix determinants by sophisticated transformations. The potential for nesting black box algorithms can further increase complexity. Hence, the efficiency of sparse interpolation is greatly influenced by the number of required evaluation points. We employ two novel techniques for reducing the number of overall evaluations required during Zippel's algorithm. First, we introduce an extra variable which provides an exact bound for the degree of each monomial. Secondly, utilizing these exact bounds of each monomial, we reduce the size of the resulting transposed Vandermonde system and hence the number of black box probes by "term pruning" after each interpolation step.

Our pruning sparse conversion algorithm proved quite useful for our black box factorization benchmark problem (see §4.1). For instance, one benchmark problem was to convert to sparse representation a factor of a 10 by 10 symmetric Toeplitz determinant given a black box which evaluates both irreducible factors. Our pruning sparse conversion algorithm only requires $2,623$ black box calls while the original version employs $4,675$ black box evaluations. For both runs, an exact degree bound $(d_i)$ for each variable was supplied. While exact degree information $d_i$ and $\deg(f)$ is extremely beneficial for the original version found in Kaltofen [Kal89], it is not so critical for our pruning sparse conversion algorithm. In fact, our technique for "zooming-in" on the degree of each monomial provides a mechanism which can quickly account for overestimated degree bounds.

In §3.1.1, we provide a formal presentation and description of our pruning sparse interpolation algorithm. We give a simple demonstration of our algorithm in §3.1.2.

### 3.1.1 The Pruning Sparse Interpolation Algorithm

Given a multivariate polynomial $f(x_1, \ldots, x_n) \in \mathsf{K}[x_1, \ldots, x_n]$ in its' black box representation, consider the representation of $f$ as the following sum of monomials:

$$f(x_1, \ldots, x_n) = \sum_{(e_1,\ldots,e_n)\in J} c_{e_1,\ldots,e_n} x_1^{e_1} \cdots x_n^{e_n}$$
$$0 \neq c_{e_1,\ldots,e_n} \in \mathsf{K}, \quad J \subset \mathrm{N}^n.$$

The sparse representation for $f(x_1, \ldots, x_n)$ is the vector

$$((e_1, \ldots, e_n, c_{e_1,\ldots,e_n}))_{(e_1,\ldots,e_n)\in J}.$$

At this point, we provide a detailed description of our algorithm. The full analysis and probabilistic measure follows from Kaltofen's [Kal89] original version. For our algorithm, the probability formulation depends on the singularity of the resulting Vandermonde system as well as the support structure at each interpolation step.

**Algorithm** *Pruning Sparse Conversion*

*Input:* $f(x_1, \ldots, x_n) \in \mathsf{K}[x_1, \ldots, x_n]$ given by a black box $B$, a bound $d_i \geq \deg(x_i), 1 \leq i \leq n$, a bound for the degree of $f$, the allowed failure probability, and an upper bound $t \leq (d_0+1)^n$ for the number of nonzero monomials permitted in the answer. Note that if the $\deg(f)$ is not known, a crude upper bound $\deg(f) \leq \prod_{i=1}^{i=n} d_i$ can be supplied.

*Output:* With high probability the sparse representation of a polynomial with no more than $t$ monomials or an indication that $f(x_1, \ldots, x_n)$ probably has more than $t$ monomials.

**Step 1:** *Select initial evaluation points*

*Pick initial random field elements* $a_1, \ldots, a_n$. Let the overline operator $\bar{}$ for any $h \in \mathsf{K}[x_1, \ldots, x_n]$ be the projection

$$\bar{h}(x_0, \ldots, x_n) = h(x_1x_0, x_2x_0, \ldots, x_nx_0)$$

which implicitly augments the number of variables to include $x_0$. We will utilize the augmented polynomial $\bar{f}(x_0, \ldots, x_n)$ for the purpose of interpolation. Note that the degree of $x_0$ is $\deg(f)$ and that $x_0$ is represented in each monomial. Furthermore, the degree of $x_0$ in each monomial is the total degree of $x_1, \ldots, x_n$ for each particular monomial in $f$. This will provide an exact bound for our prune step detailed below. In many cases, the $\deg(f) + 1$ extra evaluations introduced by the extra variable will be overcome by the savings brought by advanced knowledge of the exact degree measure of the monomials. However, for those instances where the cost of the extra evaluations are not surmounted, our algorithm can easily be modified to remove this procedure and prune with a degree bound instead.

**Step 2:** *Interpolation loop*

*For* $i \leftarrow 0, \ldots, n$ *Do: Step I then populate the set of pruned monomials* $P_{n+1}$ *with the vectors and corresponding coefficients of those* $(e_0, \ldots, e_n) \in J_{n+1}$ *whose* $d_{e_0,\ldots,e_n}$ *value is* 0. Both the set of vectors $J_{n+1}$ and degree bound $d_{e_0,\ldots,e_n}$ are defined in Step I.

$$P_{n+1} \leftarrow P_n \cup \{(e_0, \ldots, e_n, c_{e_0,\ldots,e_n}) \mid (e_0, \ldots, e_n) \in J_{n+1}, d_{e_0,\ldots,e_n} = 0\}$$

*Finally, return*

$$\sum c_{e_1,\ldots,e_n} x_1^{e_1} \cdots x_n^{e_n}.$$

This distributed representation is constructed from the pruned monomials in $P_{n+1}$. Note that any mention of $e_0$ can be safely excluded from the answer since the overline operator $\bar{\ }$ does not change the coefficients of the monomials in the original polynomial.

**Step I:** *Interpolate one more variable*

At this point, we have with high probability computed the sparse representation of

$$\bar{f}(x_0, \ldots, x_{i-1}, a_i, \ldots, a_n) = \sum_{(e_0,\ldots,e_{i-1}) \in J_i} c_{e_0,\ldots,e_{i-1}} x_0^{e_0} \cdots x_{i-1}^{e_{i-1}},$$
$$0 \neq c_{e_0,\ldots,e_{i-1}} \in \mathsf{K}, J_i \subset \mathrm{N}^{i-1}.$$

Here we interpolate one more variable.

*If* $i = 0$ *set*

$$j_0 \leftarrow \deg(f) + 1,\ \hat{J}_0 \leftarrow \{(\delta) \mid 0 \le \delta < j_i\},\ P_0 = \emptyset,\ J_0 = \{\emptyset\}$$

*and continue to* Step V.

For all other values of $i$ we have the previously computed set of exponent vectors $J_i$. Each exponent vector in $J_i$ can be associated with a corresponding nonzero coefficient $c_{e_0,\dots,e_{i-1}}$. *Set*

$$\hat{J}_i \leftarrow \{(e_0,\dots,e_{i-1},\delta) \mid (e_0,\dots,e_{i-1}) \in J_i, 0 \le \delta \le d_{e_0,\dots,e_{i-1}}, d_{e_0,\dots,e_{i-1}} > 0\},$$
$$j_i \leftarrow \operatorname{card}(\hat{J}_i)$$

where $d_{e_0,\dots,e_{i-1}} = \min[d_i, e_0 - e_1 - \dots - e_{i-1}]$. *Populate the set of pruned monomials* $P_i$ *with the vectors and corresponding coefficients of those* $(e_0,\dots,e_{i-1}) \in J_i$ *whose* $d_{e_0,\dots,e_{i-1}}$ *value is* $0$.

$$P_i \leftarrow P_{i-1} \cup \{(e_0,\dots,e_{i-1}, c_{e_0,\dots,e_{i-1}}) \mid (e_0,\dots,e_{i-1}) \in J_i, d_{e_0,\dots,e_{i-1}} = 0\}.$$

Note that $e_0$ is the exact degree of each monomial by virtue of the the overline operator $\bar{\ }$.

**Step V:** *Solve transposed Vandermonde system of equations*
*Select random points* $b_0,\dots,b_i$.
*For* $k \leftarrow 0,\dots,j_i - 1$ *Do:*
*Compute*

$$g_{k,i} \leftarrow \bar{f}(b_0^k,\dots,b_i^k,a_{i+1},\dots,a_n) - \sum_{(e_0,\dots,e_{i-1},c_{e_0,\dots,e_{i-1}}) \in P_i} c_{e_0,\dots,e_{i-1}} b_0^{ke_0} \cdots b_{i-1}^{ke_{i-1}}$$

*by evaluating* $\bar{B}(b_0^k,\dots,b_i^k,a_{i+1},\dots,a_n)$ *and adjusting this value to account for the monomials pruned thus far. If* $B$ *is not defined at* $b_0^k,\dots,b_i^k,a_{i+1},\dots,a_n$ *return "failure".*
*oD*

*Solve the $j_i$ by $j_i$ transposed Vandermonde linear system of equations*

$$\sum_{(e_0,\dots,e_i)\in\hat{J}_i} \gamma_{e_0,\dots,e_i} b_0^{e_0 k} \cdots b_i^{e_i k} = g_{k,i}, \quad 0 \le k \le j_i.$$

*If the system is singular return "failure".* We suggest the efficient algorithm by Kaltofen and Lakshman that solves a transposed Vandermonde system of equations in $\mathrm{O}(n^2)$ employing only $\mathrm{O}(n)$ space. There is a small chance that the solution of the above linear system is wrong if a "bad" anchor point $a_0, \dots, a_n$ is chosen. For details on the probability of choosing a rogue anchor point, see Zippel [Zip90]. *Set* $c_{e_0,\dots,e_i} = \gamma_{e_0,\dots,e_i}$ *where the right hand side ranges over all nonzero components of the solution of the above system.* Notice that the subscripts define the set $J_{i+1}$ for which each degree vector can be associated a nonzero $c_{e_0,\dots,e_i}$. *If the number of those nonzero coefficients becomes more than $t$, return "input polynomial has (probably) more than $t$ monomials." Set*

$$J_{i+1} \leftarrow \{(e_0, \dots, e_i) \mid \gamma_{e_0,\dots,e_i} \neq 0\}.$$

□

### 3.1.2 Demonstration of the Pruning Sparse Interpolation Algorithm

Let us consider as an example the problem of interpolating the polynomial $f(x_1, x_2, x_3) \in \mathrm{Q}[x_1, x_2, x_3]$ given by

$$f(x_1, x_2, x_3) = x_1^2 + x_1 + x_1x_2 + x_2^2 + x_2 + x_3.$$

In what follows we provide an annotated step by step trace of our pruning sparse interpolation algorithm.

**Step 1:** *Select initial evaluation points*

Our first task is to choose three random field elements. Let us select the following values $\{7, 6, 14\}$ as the anchor points $a_1, a_2, a_3$ respectively. Step 2 performs a variable by variable interpolation of a new polynomial $\bar{f}(x_0, x_1, x_2, x_3)$ given by:

$$\bar{f}(x_0, x_1, x_2, x_3) = f(x_1 x_0, x_2 x_0, x_3 x_0) = x_0^2 x_1^2 + x_0 x_1 + x_0^2 x_1 x_2 + x_0^2 x_2^2 + x_0 x_2 + x_0 x_3.$$

As illustrated above, by implicitly introducing another variable $x_0$ the algorithm can determine an exact degree bound for each monomial. Indeed, even for this small example there is an overall savings of two black box evaluations compared to Zippel's interpolation.

**Step 2:** *Interpolation loop*

$\mathbf{i = 0}$

We first seek the skeleton for $x_0$. For this example let us assume that we are provided with the following information regarding the degree of $f(x_1, x_2, x_3)$, upper bounds for each of the variables in $f$ and the total number of terms in $f$:

$$\deg(f) = 3, \quad d_0 = 2, \quad d_1 = 2, \quad d_2 = 2, \quad t = 6.$$

Also, let us assume that the choice of random elements come from a set of sufficiently large cardinality to guarantee a high probability of success. Our initial set of state variables are initialized as follows:

$$j_0 \leftarrow 3, \quad \hat{J}_0 \leftarrow \{(0), (1), (2)\}, \quad P_0 = \emptyset, \quad J_0 = \{\emptyset\}.$$

**Step V:** *Solve transposed Vandermonde system of equations*

For each interpolation step $i$ we randomly choose an an initial $i$-tuple $\vec{b}_i = \{b_0, \ldots, b_i\}$. Given an $i$-tuple $\vec{b}_i$ and a degree vector $\vec{e}$ we denote by $\vec{b}_i^{\vec{e}}$ the set $\{b_0^{e_0}, \ldots, b_i^{e_i}\}$ and define $(\vec{b}_i)^k$ as the set $\{b_0^k, \ldots, b_i^k\}$. We generate $j_i$ values of $\bar{f}(x_1, \ldots, x_n)$ from evaluation points constructed from successive powers of $\vec{b}_i$ and points selected from our initial set of anchor field elements $\vec{a}_{i+1} = \{a_{i+1}, \ldots, a_n\}$. The values for each of the monomials are $(\vec{b}_i^{\vec{e}})^k$, $\vec{e} \in \hat{J}_i$ for $k = 0, \ldots, j_i - 1$ . Thus we have the following transposed Vandermonde system of equations:

$$\begin{aligned}
\gamma_{e_{0,0},\dots,e_{0,i}} + \dots + \gamma_{e_{j_1,0},\dots,e_{j_i,i}} &= \bar{f}(\vec{b}_i^0, \vec{a}_{i+1}) \\
\gamma_{e_{0,0},\dots,e_{0,i}} b_0^{e_{0,0}}, \dots, b_i^{e_{0,i}} + \dots + \gamma_{e_{j_1,0},\dots,e_{j_i,i}} b_0^{e_{j_1,0}}, \dots, b_i^{e_{j_i,i}} &= \bar{f}(\vec{b}^1, \vec{a}_{i+1}) \\
&\vdots \\
\gamma_{e_{0,0},\dots,e_{0,i}} (b_0^{e_{0,0}}, \dots, b_i^{e_{0,i}})^{j_i-1} + \dots + \gamma_{e_{j_1,0},\dots,e_{j_i,i}} (b_0^{e_{j_1,0k}}, \dots, b_i^{e_{j_i,i}})^{j_i-1} &= \bar{f}(\vec{b}^{j_i-1}, \vec{a}_{i+1})
\end{aligned}$$

where

$$\hat{J}_i \leftarrow \{(e_{0,0}, \dots, e_{0,i}), (e_{1,0}, \dots, e_{1,i}), \dots, (e_{j_i,0}, \dots, e_{j_i,i})\}.$$

We choose $\vec{b}_0 = \{10\}$ and select $\vec{a}_1 = \{7, 6, 14\}$ from our set of anchor values. For this particular instance we do not need to account for pruned monomials and consequently have the following system of transposed Vandermonde equations:

$$\begin{aligned}
\gamma_0 + \gamma_1 + \gamma_2 &= 154 \\
\gamma_0 + \gamma_1 10 + \gamma_2 100 &= 12970 \\
\gamma_0 + \gamma_1 100 + \gamma_2 1000 &= 1272700
\end{aligned}$$

The solution of the above system is $\gamma_0 = 0, \gamma_1 = 27, \gamma_2 = 127$. We set the next degree vector to include those degrees whose monomials are non zero:

$$J_1 \leftarrow \{(1), (2)\}.$$

The monomials corresponding to each entry in $J_1$ are given by $c_1 = 27, c_2 = 127$ respectively.

**$\mathbf{i = 1}$**

At this point we must create our new set of exponent vectors $\hat{J}_1$ utilizing $J_1$ as a starting point. One can observe that a tight bound has already been found for all of the monomials. Each monomial in $f$ will have either degree 1 or 2. The new set of state variables employed to interpolate $x_1$ follows:

$$\hat{J}_1 \leftarrow \{(1,0), (1,1), (2,0), (2,1), (2,2)\}, \quad j_1 \leftarrow 5.$$

**Step V:** *Solve transposed Vandermonde system of equations*
We choose $\vec{b}_1 = \{13, 17\}$ and employ $\vec{a}_2 = \{6, 14\}$ to complete our evaluation point. The values for each of the $\vec{b}_1^{\vec{e}}$, $\vec{e} \in \hat{J}_1$ are

$$\{13, 221, 169, 2873, 48841\}$$

and each of the values of $\bar{f}(x_0, x_1, x_2, x_3)$ are given by

$$\{64, 72644, 2436048472, 116649904565948, 5690748462493657264\}.$$

The solution of the transposed Vandermonde system defined by the above state variables in terms of $\gamma$ is $\gamma_{1,0} = 20, \gamma_{1,1} = 1, \gamma_{2,0} = 36, \gamma_{2,1} = 6, \gamma_{2,2} = 1$. The next degree vector $J_2$ is identical to $\hat{J}_1$ since all of the solutions are non zero:

$$J_2 \leftarrow \{(1,0), (1,1), (2,0), (2,1), (2,2)\}.$$

The corresponding set of monomials for each degree vector in $J_2$ is $c_{1,0} = 20, c_{1,1} = 1, c_{2,0} = 36, c_{2,1} = 6, c_{2,2} = 1$.
**i = 2**
At this point we have enough information to prune the $x_1$ and $x_1^2$ terms. Our set of state information follows:

$$\hat{J}_2 \leftarrow \{(1,0,0), (1,0,1), (2,0,0), (2,0,1), (2,0,2), (2,1,0), (2,1,1)\},$$
$$j_2 \leftarrow 7 \quad P_2 \leftarrow \{(1,1,1), (2,2,1)\}.$$

By pruning we were able to reduce the size of $\hat{J}_2$ by two degree vectors.
**Step V:** *Solve transposed Vandermonde system of equations*
For this step the random points were chosen as $\vec{b}_2 = \{12, 25, 24\}$. The Vandermonde system of equations is described by the monomials $\vec{b}_2^{\vec{e}}$, $\vec{e} \in \hat{J}_2$ given by:

$$\{12, 288, 144, 3456, 82944, 3600, 86400\}$$

and evaluations:

$$\{17, 169800, 14344752096, 1215602996600448, 103055998085609319936,$$
$$8740464474326435884247040, 741608668658921404201047023616\}.$$

Note that the values of $\bar{f}(x_0, x_1, x_2, x_3)$ have been adjusted to account for the monomials we have pruned thus far. The solution in terms of $\gamma$ is $\gamma_{1,0,0} = 14, \gamma_{1,0,1} = 1, \gamma_{2,0,0} = 0, \gamma_{2,0,1} = 0, \gamma_{2,0,2} = 1, \gamma_{2,1,0} = 0, \gamma_{2,1,1} = 1$. The degree vector $J_3$ includes those degrees whose monomials are non zero:

$$J_3 \leftarrow \{(1,0,0), (1,0,1)(2,0,2), (2,1,1)\}.$$

The corresponding set of monomials for each degree vector in $J_3$ is $c_{1,0,0} = 14, c_{1,0,1} = 1, c_{2,0,2} = 1, c_{2,1,1} = 1$.

$\mathbf{i = 3}$

The interpolation of the final variable $x_3$ begins with pruning monomials which we know do not include $x_3$. For $f$ this includes all monomials but one. The following details the current state of the interpolation process after each completed monomial has been pruned:

$$\hat{J}_3 \leftarrow \{(1,0,0,0), (1,0,0,1)\}, \quad j_3 \leftarrow 2$$
$$P_3 \leftarrow \{(1,1,1), (2,2,1)(1,0,1,1), (2,0,2,1), (2,1,1,1)\}.$$

As a result of pruning, the size of the linear system in this stage has been decreased.

**Step V:** *Solve transposed Vandermonde system of equations*

The base evaluation point was chosen as $\vec{b}_3 = \{24, 3, 20, 3\}$. There is no need to pad the subsequent evaluation points with the initial anchor elements since we are now interpolating the final variable. The Vandermonde system of equations are described by the monomials $\vec{b}_3^{\vec{e}}$, $\vec{e} \in \hat{J}_3$,

$$\{24, 72\}$$

and evaluations of $\bar{f}(x_0, x_1, x_2, x_3)$,

$$\{1, 72\}.$$

The values of $\bar{f}$ have been adjusted by the monomials we have pruned thus far. The solution in terms of $\gamma$ is $\gamma_{1,0,0,0} = 0, \gamma_{1,0,0,1} = 1$. The subsequent degree vector $J_4$,

$$J_4 \leftarrow \{(1, 0, 0, 1)\}$$

has as its' corresponding monomial $c_{1,0,0,1} = 1$.

**Step 2:** *Augment set of pruned monomials and return $f$*
Finally, the distributed representation of $f$ is retrieved by converting the sparse representation stored in the augmented set $P_4$ given by:

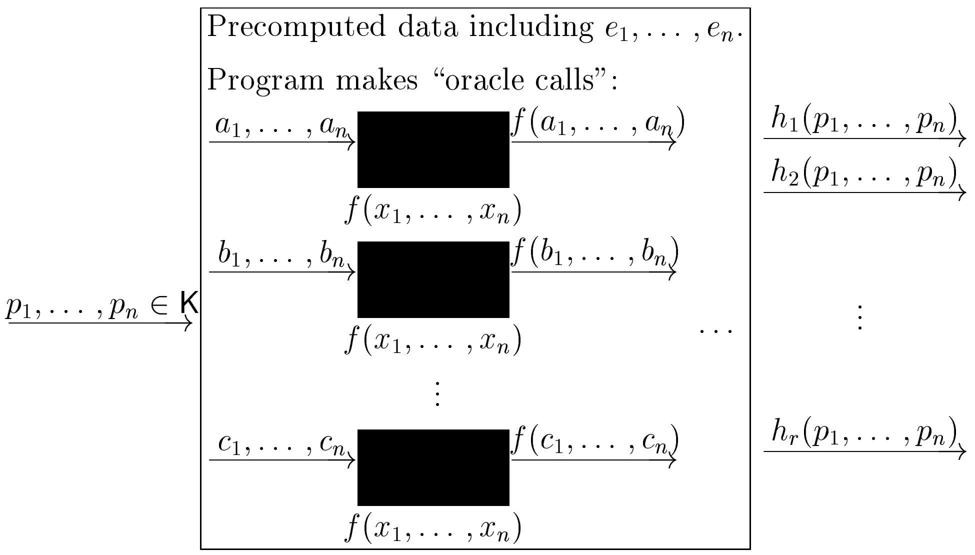
$$P_4 \leftarrow \{(1, 1, 1), (2, 2, 1), (1, 0, 1, 1), (2, 0, 2, 1), (2, 1, 1, 1), (1, 0, 0, 1, 1)\}.$$

## 3.2 Factorization

The input to the factor black box algorithm is a polynomial $f(x_1, \ldots, x_n) \in \mathsf{K}[x_1, \ldots, x_n]$ given by its' black box format (see §1.1). Also, a failure probability $\epsilon \ll 1$ is input. The result is an evaluation program for all irreducible factors of the input black box polynomial (Figure §3.1). Let us denote

$$f(x_1, \ldots, x_n) = \prod_{i=1}^{r} h_i(x_1, \ldots, x_n)^{e_i}, \quad e_i \geq 1.$$

as the factorization of $f$ into irreducibles. The factor evaluation program accepts as input $n$ arbitrary elements $p_1, \ldots, p_n \in \mathsf{K}$. The evaluation program then calls the black box for $f(x_1, \ldots, x_n)$ and determines with high probability $(1-\epsilon)$ the values for all irreducible factors $h_i(p_1, \ldots, p_n)$, $1 \leq i \leq r$. Furthermore, the factor evaluation program returns the corresponding exponents $e_1, \ldots, e_n$ with the same probability. During the construction phase, the algorithm actually chooses an associate for each factor $h_i(x_1, \ldots, x_n)$. For repeated invocations, the evaluation program returns the value scaled by that associate.



$$f(x_1, \ldots, x_n) = h_1(x_1, \ldots, x_n)^{e_1} \cdots h_r(x_1, \ldots, x_n)^{e_r}$$
$$h_i \in \mathsf{K}[x_1, \ldots, x_n] \text{ irreducible.}$$

**Figure 3.1: Black box factorization**

Note that with probability at least $1 - \epsilon$, the factor evaluation program is correct and that a correct program will always produce a valid associate for each of the factors. The probability of success depends on the selection of random field elements chosen during the construction phase of the algorithm. The relationship between the cardinality of the set from which the random elements are chosen, $R$, and the probability of success $\epsilon$ is given by

$$\mathrm{card}(R) \geq 6\deg(f)\, 2^{\deg(f)}/\epsilon.$$

In what follows, we provide a description of two improvements to the black box factor algorithm. The reader is referred to Kaltofen and Trager [KT90] for a detailed exposition and full analysis of this algorithm.

During the evaluation process, the projection of $f(x_1, \ldots, x_n)$ given by the overline operator $\bar{}$

$$\bar{f}(X, Y) = f(X + b_1,\ Y(p_2 - a_2(p_1 - b_1) - b_2) + a_2X + b_2,$$
$$\ldots,\ Y(p_n - a_n(p_1 - b_1) - b_n) + a_nX + b_n)$$

must be computed by standard interpolation. The bi-variate polynomial $\bar{f}(X,Y)$ is utilized to perform a Hensel lifting [Mus75] of the factorization of $\bar{f}(X,0)$. Given a particular factor of $\bar{f}(X,0)$ in $X$ whose bi-variate counterpart has degree $d$ in $Y$, the lifting process only requires the coefficients of $Y$ of $\bar{f}(X,Y)$ up to degree $d$. In fact, our implementation of the Hensel algorithm utilizes truncated bi-variate polynomial arithmetic to weed out unnecessary terms. Let us denote the factorization of $\bar{f}(X,Y)$ by

$$\bar{f}(X,Y) = \prod_{i=1}^{r} \bar{g}_i(X,Y)^{e_i}.$$

Since we lift all of the factors at the same time, one can set the truncation degree to $d = \max(\deg_Y(g_i(X,Y))), 1 \le i \le r$. The implication is that for small degree factors the bi-variate interpolation step can be dramatically improved in performance if the black box for the original polynomial can be probed at truncated power series as values for the variables. Such black boxes that can evaluate at points which come from a domain that is an extension of the field over which the polynomial was factored are called extended domain black boxes.

For the case of the truncated power series domain, the bi-variate interpolation algorithm probes $\bar{f}(X,Y)$ at $d_{fb} = \deg(\bar{f}(X,Y)) + 1$ distinct field elements for $X$ resulting in polynomials in $Y$.

$$\begin{aligned}
\bar{f}(d_0,Y) \bmod Y^d &= c_{0,0} + c_{0,1}Y + \cdots + c_{0,d-1}Y^{d-1} \\
\bar{f}(d_1,Y) \bmod Y^d &= c_{1,0} + c_{1,1}Y + \cdots + c_{1,d-1}Y^{d-1} \\
&\vdots \\
\bar{f}(d_{fb},Y) \bmod Y^d &= c_{d_{fb},0} + c_{d_{fb},1}Y + \cdots + c_{d_{fb},d-1}Y^{d-1}
\end{aligned}$$

Note that since the input black box performs arithmetic over an extension field given by $\mathsf{M} = \mathsf{K}[Y]/(Y^d)$ the resulting polynomials in $Y$ are of degree $d-1$. Finally, each coefficient of $Y$ in

$$\bar{f}(X,Y) \bmod Y^d = C_0(X) + C_1(X)Y + \cdots + C_{d-1}Y^{d-1}$$

can be retrieved by interpolating each corresponding coefficient band

$$C_i = \texttt{interpolate}(\{d_0, \dots, d_{fb}\}, \{c_{0,i}, \dots, c_{d_{fb},i}\}, X)$$

as a polynomial in $X$. The `interpolate` function computes a polynomial of degree less than or equal to $d_{fb}$ in the variable $X$ from a list of independent $(\{d_0, \dots, d_{fb}\})$ and dependent $(\{c_{0,i}, \dots, c_{d_{fb},i}\})$ values. As illustrated above the the extended domain factor object switches to univariate interpolation over the truncated power series domain $\mathsf{M}$. 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 $\mathsf{M}$ runs a factor $O(d^2)$ slower.

When the input black box can only evaluate points from the field over which the polynomial was factored, we provide a bi-variate interpolation algorithm that takes advantage of the special structure of $\bar{f}(X, Y)$. Indeed, borrowing from our "term pruning" method described in §3.1 we can reduce the number of black box evaluations in half.

For any $f$, the overline operator $\bar{}$ provides a map for each variable resulting in a bi-variate polynomial in $X$ and $Y$ that has the following form:

$$\bar{f}(X, Y) = c_{0,0}Y^d + (c_{0,1} + c_{1,1}X)Y^{d-1} + (c_{0,2} + c_{1,2}X + c_{2,2}X^2)Y^{d-2} + \cdots + (c_{0,d} + c_{1,d}X + \cdots + c_{d,d}X^d),$$

$$\text{where} \quad c_{i,i}, 0 \le i \le d, \in \mathsf{K}, \quad \text{and} \quad d = \deg(\bar{f}(X, Y)).$$

Note that if one views $\bar{f}(X, Y)$ as a polynomial in $Y$ with coefficients in $X$, the task of interpolating $\bar{f}(X, Y)$ can be modified to reduce the number of necessary black box evaluations. The first step is to select $d + 1$ distinct field elements $x_0, \dots, x_d$. Much like standard bi-variate interpolation, our algorithm iterates through each of $x_i$'s computing polynomials $\bar{f}(x_i, Y), 0 \le i \le d$. However, at the $i$-th stage instead of interpolating a full degree polynomial, we only compute a polynomial of degree $i$ in $Y$. This is achieved by successively pruning the $i$-th coefficient of $Y$ and adjusting the result of subsequent black box evaluations accordingly. We know that the $i$-th coefficient in $Y$ is "completed" by virtue of the projection imposed by the overline operator $\bar{\ }$. Our pruning bi-variate interpolation technique becomes quite important for input polynomials of large degree or those polynomials that are represented by so-called "nested" black boxes. Our algorithm proved quite useful in the completion of our numerator and denominator benchmarks problems detailed in §4.3. Clearly, the gains from only probing the numerator and denominator black box half as many times to evaluate the factors of the numerator were significant.

## 3.3 Greatest Common Divisor

In this section, we revisit the problem of computing the black box greatest common divisor (GCD) of several multivariate polynomials that are given by black boxes. In Kaltofen and Trager [KT90] a brief sketch of a possible solution was presented.

The black box GCD algorithm must fix a unique scalar multiple of the GCD that the produced black box will evaluate at the given points. The scalar multiple is fixed by performing a generic symbolic shift of the variables that makes the shifted GCD monic in the main variable $X$. By our shifts, we also avoid the computationally costly content and primitive part problem of GCD algorithms for polynomials in explicit representation like Brown's [Bro71], Zippel's [Zip79], or Char's et al. [CGG89] modular algorithms, and like Moses's and Yun's [MY73], Wang's [Wan80], or Kaltofen's [Kal85c] Hensel-lifting based algorithms. Note that the shifts do not introduce expression swell since the representation of the input polynomials is by black boxes. Values of the GCD are obtained by introducing a second variable $Y$ in the fashion of homotopy continuation [Dre77]. For $Y = 0$, the black box computes the value of the GCD at a predetermined point, while for $Y = 1$ we will have the value of the GCD at the inputs to the black box. Finally, the problem of computing the GCD of several polynomials is reduced to the problem of computing the GCD of a pair of polynomials by taking a random scalar sum, which is a technique first introduce by D. Spear (see Wang [Wan80], p. 57). Finally, we remark that our algorithm uses randomization and that the resulting black box for the GCD may with controllably small probability be incorrect. However, if the black box was determined correctly, the values for the GCD will always be correct.

Our improvement of the algorithm sketched in Kaltofen and Trager [KT90] is as follows [DK95]. By using a modular GCD approach for the arising bivariate GCD problem (in the variables $X$ and $Y$) we can greatly reduce the number of calls to the black boxes for the input polynomials. In fact, under fortunate circumstances the input black boxes need only be called as many times as the degree of the input polynomials in order to obtain the value of the GCD at a single point.

At this point, in §3.3.1 we provide a detailed description and in §3.3.2 a full analysis of our algorithm.

### 3.3.1 The black box GCD algorithm

We now give a detailed description of our algorithm.

**Algorithm** *Black Box GCD*

*Input:* A black box for each polynomial $f_i(x_1, \dots, x_n) \in \mathsf{K}[x_1, \dots, x_n]$ for $i = 1, \dots, r$, $r \geq 2$, where $\mathsf{K}$ is a field.

*Output:* A program (see Figure 3.2) that makes calls to the black boxes of the $f_i$'s and has with probability no less than $1-\epsilon$ the following property. The program accepts $n$ field elements $p_1, \dots, p_n$. It returns $g(p_1, \dots, p_n) \in \mathsf{K}$ where $g = \mathrm{GCD}_{1 \leq i \leq r}(f_i)$. Notice that $g(p_1, \dots, p_n)$ is determined only up to a multiple in $\mathsf{K}$. For repeated invocations with different arguments, it returns the value scaled by the same multiple. Notice also that the failure probability applies to the construction and not to the execution of the program. That is, with probability at least $1-\epsilon$ the output program is correct; a correct program will always produce the true values of the GCD.

**Step 1:** *Pick random field elements*

$$a_2, \dots, a_n, b_2, \dots, b_n, c_3, \dots, c_r$$

*from a sufficiently large finite subset* $\mathsf{R} \subset \mathsf{K}$. We will give the cardinality of this set in relation to $\deg(f_i)$ for $1 \leq i \leq r$ and $\epsilon$ in the statement of theorem 1 below.

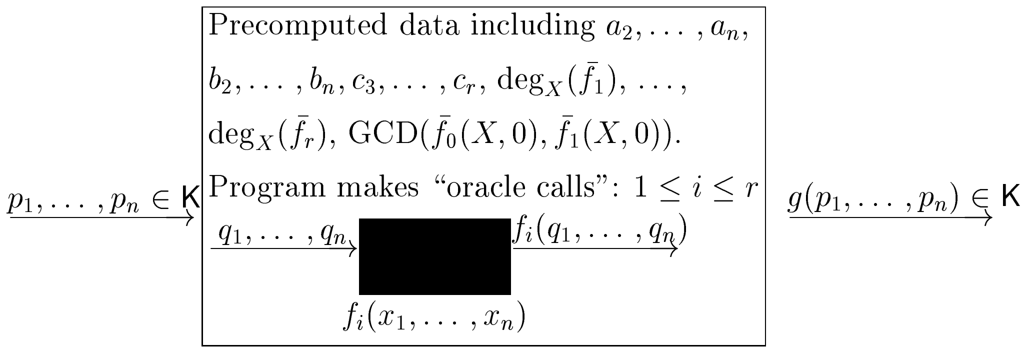

Precomputed data including $a_2, \dots, a_n$, $b_2, \dots, b_n, c_3, \dots, c_r$, $\deg_X(\bar{f}_1)$, …, $\deg_X(\bar{f}_r)$, $\mathrm{GCD}(\bar{f}_0(X,0), \bar{f}_1(X,0))$.
Program makes "oracle calls": $1 \le i \le r$
$p_1, \dots, p_n \in \mathsf{K}$
$q_1, \dots, q_n$
$f_i(q_1, \dots, q_n)$
$f_i(x_1, \dots, x_n)$
$g(p_1, \dots, p_n) \in \mathsf{K}$

**Figure 3.2: Black box greatest common divisor**

Let the overline operator $\bar{\ }$ for any $h \in \mathsf{K}[x_1, \dots, x_n]$ be the projection

$$\bar{h}(X,Y) = h(X, Y(p_2 - a_2p_1 - b_2) + a_2X + b_2, \\ \dots, Y(p_n - a_np_1 - b_n) + a_nX + b_n).$$

Note that $\bar{h}(p_1, 1) = h(p_1, \dots, p_n)$. We will be using the bivariate polynomials

$$\bar{f}_0(X,Y) = \bar{f}_2(X,Y) + \sum_{i=3}^{r} c_i\bar{f}_i(X,Y),\ \ \bar{f}_1(X,Y),$$
$$\gamma(X,Y) = \mathrm{GCD}(\bar{f}_0(X,Y), \bar{f}_1(X,Y)),$$

and the univariate GCDs

$$\bar{\gamma}_e(X) = \mathrm{GCD}(\bar{f}_0(X,e), \bar{f}_1(X,e)) \quad \text{for} \quad e \in \mathsf{K}.$$

Among the possible associates for the GCDs $\gamma$ and $\bar{\gamma}_e$ we always choose those whose leading coefficient is one and the same element of $\mathsf{K}$, namely the leading coefficient of $\bar{\gamma}_0$. We will show in the proof of Theorem 1 below that with high probability $\gamma = \bar{g}$.

**Step 2:** *By standard interpolation compute*

$$\bar{f}_0(X,0) = f_2(X, a_2X + b_2, \ldots, a_nX + b_n) + \sum_{i=3}^{r} c_i f_i(X, a_2X + b_2, \ldots, a_nX + b_n)$$

*and*

$$\bar{f}_1(X,0) = f_1(X, a_2X + b_2, \ldots, a_nX + b_n);$$

*then compute* $\bar{\gamma}_0(X) = \mathrm{GCD}(\bar{f}_0(X,0), \bar{f}_1(X,0))$.

The interpolation algorithms mentioned above and below need to know $\deg_X(\bar{f}_i)$ for all $1 \le i \le r$. Either the degree or an upper bound is supplied as input, or the degree can be probabilistically guessed as follows (see Kaltofen and Trager 1990). Pick a random $B \in \mathsf{R}$ and compute $\bar{f}_i(X,B)$ by determining a succession of polynomials $\bar{f}_i^{(d)}(X,B)$ for $d = 1,2,3,\ldots$ until $\bar{f}_i^{(d)}(X,B) = \bar{f}_i(X,B)$, where $\bar{f}_i^{(d)}(X,B)$ is the interpolate at $X = 0,1,\ldots,d$ of $\bar{f}_i(X,B)$. We test whether $\bar{f}_i^{(d)}(X,B) = \bar{f}_i(X,B)$ by evaluating at a random $A \in \mathsf{R}$: if $\bar{f}_i^{(d)}(A,B) = \bar{f}_i(A,B)$ then we declare $\bar{f}_i^{(d)}(X,B) = \bar{f}_i(X,B)$ and $\deg_X(\bar{f}_i) = d$, otherwise we continue the interpolation. The failure probability is bounded by $\deg(\bar{f}_i)/\mathrm{card}(\mathsf{R})$, which means that the degree in $X$ of $\bar{f}_i(X,Y)$ would have to be very large in order for the guessing procedure to be unreliable. Note that one may use $B = 0$ provided the leading coefficient of $\bar{f}_i(X,Y)$ does not depend on $Y$. One can estimate the probability that the random $a_i$ yield this condition, but we will not incorporate such an estimate in our analysis. Ultimately, if the probability of determining the degree correctly is to be guaranteed, an upper bound for all $\deg(f_i)$ must be provided by the user of this algorithm.

If not all polynomials $f_i$ for $i \ge 3$ are needed to determine $\bar{\gamma}_0$, those polynomials that are redundant meaning that they can be omitted without affecting the GCD, preferably the ones of highest degree, can be ignored for future considerations. An optimization at this point can significantly reduce the complexity of the output black box.

**Step 3:** This step constructs the programs for evaluation of $g$ at $p_1, \ldots, p_n$ as described in the output specifications. First $a_2, \ldots, a_n, b_2, \ldots, b_n, c_3, \ldots, c_r$, $d_i =$

$\deg_X(\bar{f}_i)$ for $1 \leq i \leq r$, $\bar{\gamma}_0$, and $\delta = \deg(\bar{\gamma}_0)$ are "hardwired" into that program. Then the following Steps A, and B are appended to the program.

**Step A:** Let $\mathsf{S} \subset \mathsf{K}$ containing $\{1\}$ be of cardinality at least $d_0 d_1 + \delta$. *First select* $i_1 = 1$ *and compute* $\bar{\gamma}_1(X)$. *If* $\deg(\bar{\gamma}_1(X)) = \delta$ *then* the black box can terminate early, therefore *return* $\bar{\gamma}_1(p_1)$ *as the requested evaluation. If* $\deg(\bar{\gamma}_1(X)) > \delta$ *compute* $\gamma$ *by using a modular* GCD *approach. To this end select* $i_1, i_2, \ldots, i_\delta$ *pairwise distinct elements in* $\mathsf{S}$ *such that* $\deg(\bar{\gamma}_{i_j}) = \delta$ *for* $0 \leq j \leq \delta$. Any element $e$ in $\mathsf{S}$ such that $\deg(\bar{\gamma}_e) > \delta$ must be discarded. Further we are guaranteed that only $d_0 d_1$ of such elements exist. If for any element $e$ in $\mathsf{S}$ we have $\deg(\bar{\gamma}_e) < \delta$ then the black box is invalid and the program returns an error indicating that a new black box is needed with different random elements. The polynomials $\bar{\gamma}_e$ are computed as GCDs of $\bar{f}_0(X, e)$ and $\bar{f}_1(X, e)$ which can be computed by interpolation using $d_i$ as degree bound for each $\bar{f}_i(X, e)$.

**Step B:** Let $I = \{i_0 = 0, i_1, i_2, \ldots, i_\delta\}$, let $P = \{\bar{\gamma}_{i_0}(X),\ \bar{\gamma}_{i_1}(X),\ \ldots, \bar{\gamma}_{i_\delta}(X)\}$ with each polynomial having its leading coefficient $\mathrm{ldcf}(\bar{\gamma}_0(X))$, and let `coeff(`$i$, $P$`)` for $0 \leq i \leq \delta$ be a function that returns an ordered coefficient list corresponding to the $i^{th}$ coefficient of $X^i$ of all the polynomials in $P$. Furthermore, let `interpolate`$(I, C, V)$ be a function that computes the polynomial in the variable $V$ which interpolates the points in $I$ at the values given by $C$.

```
$\gamma \leftarrow X^\delta$;
For $j \leftarrow 0, \ldots, \delta - 1$ Do:
    Let $\gamma = \gamma + X^j \cdot \texttt{interpolate}(I, \texttt{coeff}(i, P), Y)$;
```

*Finally* $\gamma(p_1, 1)$ *can now be returned as the requested evaluation.* Note that for implementation purposes it is more efficient to implicitly perform the construction of $\gamma(X, Y)$ by substituting $X = p_1$ and $Y = 1$ after each interpolation step. The final result is the desired field element $\gamma(p_1, 1)$. $\square$

The following theorem states the complexity and the failure estimate in relation to the cardinality of $\mathsf{R}$ mentioned in Step 1 of the above algorithm.

**Theorem 1** *The Black Box Polynomial GCD algorithm can construct its output program in polynomially many arithmetic steps as a function of* $r$ *and* $\deg(f_i)$ *for* $1 \leq i \leq r$. *For each* $i$, *it requires* $\deg(f_i) + 1$ *oracle calls to the black box of the* *polynomial if the degree of each polynomial is known, or* $\deg(f_i) + 2$ *oracle calls to the black box of the polynomial if only a degree bound is known. If the cardinality of the set* $\mathsf{R}$ *in Step 1 is chosen*

$$\mathrm{card}(\mathsf{R}) \geq \deg(f_1) \cdot (1 + 2 \max_{2 \leq i \leq r}\{\deg(f_i)\})/\epsilon$$

*then the algorithm succeeds with probability no less than* $1 - \epsilon$ *and the resulting program will always correctly evaluate the GCD at all points. That program in turn can be executed in polynomially many arithmetic steps and at best with* $\deg(f_i) + 1$ *many oracle calls to each black box of* $f_i$ *for all* $1 \leq i \leq r$*, and at worst with*

$$\deg(f_i) \cdot (\delta + \deg(f_1) \cdot \max_{2 \leq i \leq r}\{\deg(f_i)\})$$

*many oracle calls to each black box of* $f_i$*, where* $\delta = \deg(\mathrm{GCD}_{1 \leq i \leq r}(f_i))$.

Note that our analysis supposes that the degrees of the input polynomials (or upper bounds for them) are known. As explained in Step 2, it is possible to probabilistically determine them.

### 3.3.2 Complexity analysis

In this section we provide a proof for the complexity and probabilistic analysis of the Black Box GCD algorithm as given in Theorem 1. We will base our arguments on three lemmas, which we state and prove next. First, we shall establish a well-known condition under which GCD operations and homomorphic imaging commute. In order to avoid ambiguities, we sometimes write $\mathrm{GCD}_{\mathsf{D}}(f_1, f_2)$ meaning that the GCD is to be taken in the domain $\mathsf{D}$. Note that our condition is weaker than those established by Brown [Bro71], who assumes that no leading coefficient of the inputs an no leading coefficient of a non-zero remainder in the Euclidean chain is mapped to zero. Consequently, we will obtain better probabilistic estimates.

**Lemma 1** *Let* $\mathsf{E}$ *be a* UFD*,* $\mathsf{K}$ *a field,* $\phi\colon \mathsf{E} \to \mathsf{K}$ *be a ring homomorphism,* $f_1 = a_n x^n + \cdots + a_0$ *and* $f_2 = b_m x^m + \cdots + b_0 \in \mathsf{E}[x]$*,* $l = \mathrm{ldcf}_x(\mathrm{S}_\delta(f_1,\, f_2))$ *where* $\delta = \deg(\mathrm{GCD}_{\mathsf{E}[x]}(f_1, f_2))$ *and where* $\mathrm{S}_\delta(f_1,\, f_2)$ *is the subresultant formal degree* $\delta$ *of*

*$f_1$ and $f_2$ (see Brown and Traub [BT71]). If $\phi(l) \neq 0$ we have*

$$u \cdot \phi(\mathrm{GCD}_{\mathsf{E}[x]}(f_1, f_2)) = \mathrm{GCD}_{\mathsf{K}[x]}(\phi(f_1), \phi(f_2))$$

*and*

$$\delta = \deg(\mathrm{GCD}_{\mathsf{K}[x]}(\phi(f_1), \phi(f_2))) \tag{1}$$

*for $u \in \mathsf{K} \setminus \{0\}$.*

**Proof** From the condition $\phi(l) \neq 0$ we can infer that one or both of the $\mathrm{ldcf}_x(\phi(f_1))$ and $\mathrm{ldcf}_x(\phi(f_2))$ must be nonzero. Let $c_\delta = \mathrm{ldcf}_x(\mathrm{GCD}_{\mathsf{E}[x]}(f_1, f_2))$. Our first claim is that

$$\delta = \deg(\phi(\mathrm{GCD}_{\mathsf{E}[x]}(f_1, f_2))),$$

where $\delta$ was defined as $\deg(\mathrm{GCD}_{\mathsf{E}[x]}(f_1, f_2))$. We observe without loss of generality that if $\phi(a_n) \neq 0$, $c_\delta \mid a_n$ implies that $\phi(c_\delta) \neq 0$, which proves the claim. Let

$$\delta' = \deg(\mathrm{GCD}_{\mathsf{K}[x]}(\phi(f_1), \phi(f_2))).$$

Since $\phi$ is a ring homomorphism

$$\phi(\mathrm{GCD}_{\mathsf{E}[x]}(f_1, f_2)) \mid \mathrm{GCD}_{\mathsf{K}[x]}(\phi(f_1), \phi(f_2)) \tag{2}$$

and consequently,

$$\delta \leq \delta'. \tag{3}$$

Let $f_1, f_2, \ldots, f_k$ be the polynomial remainder sequence (PRS) of $f_1$ and $f_2$. Also, let $\hat{f}_1, \hat{f}_2, \ldots, \hat{f}_{\hat{k}}$ be the PRS of $\phi(f_1)$ and $\phi(f_2)$. Assume for purpose of contradiction that $\delta' > \delta$. According to the Fundamental Theorem of Subresultants $\mathrm{S}_j(\hat{f}_1, \hat{f}_2) = 0$ for $j = 0, \ldots, \delta' - 1$. Since $\delta$ is in the range $0, \ldots, \delta' - 1$, $\mathrm{S}_\delta(\hat{f}_1, \hat{f}_2)) = 0$.

Case 1: Both $\mathrm{ldcf}_x(\phi(f_1))$ and $\mathrm{ldcf}_x(\phi(f_2))$ are nonzero :

The dimension of the matrix corresponding to $\phi(\mathrm{S}_\delta(f_1, f_2))$ is $(n + m - 2\delta) \times (n + m - 2\delta)$. Since $\mathrm{ldcf}_x(\phi(f_1))$ and $\mathrm{ldcf}_x(\phi(f_2))$ are nonzero the degree of $\phi(f_1)$ and $\phi(f_2)$ does not change, and therefore the dimension of the matrix corresponding to

$\mathrm{S}_\delta(\phi(f_1), \phi(f_2))$ is also $(n+m-2\delta) \times (n+m-2\delta)$. From this it follows that when $\mathrm{ldcf}_x(\phi(f_1))$ and $\mathrm{ldcf}_x(\phi(f_2))$ are nonzero $\phi(\mathrm{S}_\delta(f_1, f_2)) = \mathrm{S}_\delta(\phi(f_1), \phi(f_2))$ and hence a contradiction.

Case 2: Without loss of generality, $\mathrm{ldcf}_x(\phi(f_1)) = 0$ and $\mathrm{ldcf}_x(\phi(f_2)) \neq 0$ :

Consider the following example.

$$\phi(\mathrm{S}_\delta(f_1, f_2)) = \phi(\mathrm{Det}(\begin{bmatrix} a_n & a_{n-1} & a_{n-2} & \cdots \\ 0 & a_n & a_{n-1} & \cdots \\ & \ddots & & \ddots \\ b_m & b_{m-1} & b_{m-2} & \cdots \\ 0 & b_m & b_{m-1} & \cdots \\ & \ddots & & \ddots \end{bmatrix}))$$

$$= \mathrm{Det}(\begin{bmatrix} 0 & \phi(a_{n-1}) & \phi(a_{n-2}) & \cdots \\ 0 & 0 & \phi(a_{n-1}) & \cdots \\ & \ddots & & \ddots \\ \phi(b_m) & \phi(b_{m-1}) & \phi(b_{m-2}) & \cdots \\ 0 & \phi(b_m) & \phi(b_{m-1}) & \cdots \\ & \ddots & & \ddots \end{bmatrix})$$

and

$$\mathrm{S}_\delta(\phi(f_1), \phi(f_2)) = \mathrm{Det}(\begin{bmatrix} \phi(a_{n-1}) & \phi(a_{n-2}) & & \cdots \\ 0 & \phi(a_{n-1}) & \phi(a_{n-2}) & \cdots \\ & \ddots & & \ddots \\ \phi(b_m) & \phi(b_{m-1}) & \phi(b_{m-2}) & \cdots \\ 0 & \phi(b_m) & \phi(b_{m-1}) & \cdots \\ & \ddots & & \ddots \end{bmatrix}).$$

The dimension of $\phi(\mathrm{S}_\delta(f_1, f_2))$ is again $(n+m-2\delta) \times (n+m-2\delta)$. However, since $\mathrm{ldcf}_x(\phi(f_1)) = 0$, there is a degree drop (in the above example by 1) in $\phi(f_1)$ and hence the dimension of $\mathrm{S}_\delta(\phi(f_1), \phi(f_2))$ is now $(n+m-2\delta-1) \times (n+m-2\delta-1)$. By minor expansion of the first column of $\phi(\mathrm{S}_\delta(f_1, f_2))$, we see that $\phi(\mathrm{S}_\delta(f_1, f_2)) = \mathrm{ldcf}_x(\phi(f_2)) \cdot \mathrm{S}_\delta(\phi(f_1), \phi(f_2))$. In the case more coefficients of $\phi(f_1)$ map to zero we see that $\phi(\mathrm{S}_\delta(f_1, f_2)) = \pm\ \mathrm{ldcf}_x(\phi(f_2))^\alpha \cdot \mathrm{S}_\delta(\phi(f_1), \phi(f_2))$, where $\alpha$ is the number of coefficients of $\phi(f_1)$ mapping to zero before the first nonzero coefficient is reached. We can be guaranteed that both $\phi(f_1)$ and $\phi(f_2)$ can not vanish identically since $\phi(l) \neq 0$. This is again a contradiction to the assumption.

From the contradiction between what was arrived at from the Fundamental Theorem of Subresultants and Case 1 and 2, we know that $\delta' \leq \delta$, hence,

$$\delta \geq \delta'. \tag{4}$$

From (3) and (4)

$$\delta = \delta'. \tag{5}$$

Finally (1) follows from (2) and (5). $\square$

The next lemma concerns the reduction of the GCD problem of many polynomials to computing the GCD of a pair of polynomials. Here we follow a strategy first used by Spear and extended to multivariate polynomials by Kaltofen (1988, Theorem 6.2). Note that the usage of only $r-2$ random elements can also be found in von zur Gathen et al.(1994).

**Lemma 2** *Let $f_i(x_1, \ldots, x_n) \in \mathsf{K}[x_1, \ldots, x_n]$ be nonzero polynomials for $i = 1, \ldots, r$, $r \geq 2$, $\mathsf{K}$ a field, $d = \deg(f_1)$ for $1 \leq i \leq r$, $\mathsf{R} \subset \mathsf{K}$. Then for randomly chosen $c_i \in \mathsf{R}$, $3 \leq i \leq r$ we have,*

$$\Pr(\mathrm{GCD}_{1\leq i\leq r}(f_i) = \mathrm{GCD}(f_1, f_2 + \sum_{i=3}^{r} c_i f_i)) \geq 1 - d/\mathrm{card}(\mathsf{R})$$

**Proof** We first show this lemma for $n = 1$. Let

$$\hat{f}_1 = f_1, \quad \hat{f}_2 = f_2 + \sum_{i=3}^{r} \gamma_i f_i \in \mathsf{E}[x], \qquad \mathsf{E} = \mathsf{K}[\gamma_3 \ldots \gamma_r],$$

$\gamma_3 \ldots \gamma_r$ be indeterminants, and let $g = \mathrm{GCD}_{1\leq i\leq r}(f_i)$. Clearly, $g \mid \hat{f}_1$, $g \mid \hat{f}_2$. The first claim is that $g = \hat{g}$ where $\hat{g} = \mathrm{GCD}(\hat{f}_1, \hat{f}_2)$. We observe that $\hat{g} \in \mathsf{K}[x]$, since $\hat{g}$ divides $f_1$. Now write $\hat{f}_2 = \hat{g}\hat{f}_2^*$, where $\hat{f}_2^* \in \mathsf{E}[x]$. We know that $\hat{g} \mid f_1$, if we evaluate $\hat{f}_2 = \hat{g}\hat{f}_2^*$ at $\gamma_i = 0$ for $3 \leq i \leq r$, then we see that $\hat{g} \mid f_2$. Evaluating this equation at $\gamma_i = 1$ and $\gamma_j = 0$, $i \neq j$ for $3 \leq i \leq r$ we get $\hat{g} \mid f_2 + f_i$ for $3 \leq i \leq r$

since we know that $\hat{g} \mid f_2$, hence $\hat{g} \mid f_i$ for $3 \leq i \leq r$. Therefore $\hat{g} \mid g$. Consequently since $g \mid \hat{f}_1$ and $g \mid \hat{f}_2$ we know that $g \mid \hat{g}$, hence we can conclude that $g = \hat{g}$ which proves the first claim. Now let $\phi_{c_3,\ldots,c_r} \colon \mathsf{E} \to \mathsf{K}$ be the ring homomorphism $\gamma_i = c_i$ for $3 \leq i \leq r$, $l \in \mathsf{E}[x]$ be the $\mathrm{ldcf}_x(\mathrm{S}_\delta(\hat{f}_1,\, \hat{f}_2))$ where $\delta = \deg(\mathrm{GCD}_{\mathsf{E}[x]}(\hat{f}_1, \hat{f}_2))$. By lemma 1 if for randomly chosen $c_i \in \mathsf{R}$ for $3 \leq i \leq r$ we have $\phi_{c_3,\ldots,c_r}(l) \neq 0$, then

$$\mathrm{GCD}_{\mathsf{K}[x]}(\phi_{c_3,\ldots,c_r}(\hat{f}_1), \phi_{c_3,\ldots,c_r}(\hat{f}_2)) = \phi_{c_3,\ldots,c_r}(\mathrm{GCD}_{\mathsf{E}[x]}(\hat{f}_1, \hat{f}_2)),$$

which implies the asserted event. Since the $\deg(l) \leq d$, the Schwartz (1980)/ Zippel (1979) Lemma then establishes the stated probability. The multivariate case can now be reduced to the univariate case as in Theorem 6.2 in Kaltofen (1988). $\square$

By use of Lemma 1 we can now justify the evaluations used in Step 1 of the Black Box GCD algorithm. Because of Lemma 2 we can restrict ourselves to the case of two polynomials.

**Lemma 3** *Let $\hat{f}_1$ and $\hat{f}_2$ be nonzero polynomials $\in \mathsf{K}[x_1, \ldots, x_n]$, $d_1 = \deg(\hat{f}_1)$, $d_2 = \deg(\hat{f}_2)$, $\mathsf{R} \subset \mathsf{K}$, $a_2, \ldots, a_n$, $b_2, \ldots, b_n$ be randomly chosen elements $\in \mathsf{R}$, $\phi \colon \mathsf{K}[x_1, \ldots, x_n] \to \mathsf{K}[X]$ be the ring homomorphism generated by $x_1 = X$, $x_i = a_i X + b_i$ for $2 \leq i \leq n$, $g_1 = \mathrm{GCD}(\hat{f}_1,\, \hat{f}_2)$ and let $g_2 = \mathrm{GCD}(\phi(\hat{f}_1), \phi(\hat{f}_2))$. Then we have*

$$\Pr\left(\phi(g_1) = g_2 \quad \text{and} \quad \deg(g_1) = \deg(g_2)\right) \geq 1 - 2d_1 d_2 / \mathrm{card}(\mathsf{R})$$

**Proof** Let $\psi \colon \mathsf{K}[\alpha_2, \ldots, \alpha_n][x_1, \ldots, x_n] \to \mathsf{K}[\alpha_2, \ldots, \alpha_n, \beta_2, \ldots, \beta_n][X]$ be the ring isomorphism generated by substituting $x_1 = X$, $x_i = \alpha_i X + \beta_i$ for $2 \leq i \leq n$, let $\phi' \colon \mathsf{K}[\alpha_2, \ldots, \alpha_n, \beta_2, \ldots, \beta_n] \to \mathsf{K}$ be the ring homomorphism generated by $\alpha_i = a_i$, and $\beta_i = b_i$ for $2 \leq i \leq n$, and let $\tilde{g}_1 = \mathrm{GCD}(\psi(\hat{f}_1)\ \psi(\hat{f}_2))$. We have $\phi = \phi'\psi$ as seen in the diagram in Figure 3.3.

Due to the nature of the mapping in the ring isomorphism $\psi$ we can see that $\deg(g_1) = \deg_X(\tilde{g}_1)$ since the $\alpha_i$ do not allow the vanishing of $\mathrm{ldcf}_X(\tilde{g}_1)$. Let $l = \mathrm{ldcf}_X(\mathrm{S}_\delta(\psi(\hat{f}_1),\, \psi(\hat{f}_2)))$, where $\delta = \deg(g_1)$ and $\mathrm{S}_\delta$ is the $\delta^{th}$ subresultant with respect to $X$. We know that $l \in \mathsf{K}[\alpha_2, \ldots, \alpha_n, \beta_2, \ldots, \beta_n]$ and from the Schwartz/

$$\begin{tikzcd} \mathsf{K}[\alpha_2, \ldots, \alpha_n][x_1, \ldots, x_n] \arrow[r, "\psi"] \arrow[dr, "\phi"'] & \mathsf{K}[\alpha_2, \ldots, \alpha_n, \beta_2, \ldots, \beta_n][X] \arrow[d, "\phi'"] \\ & \mathsf{K}[X] \end{tikzcd}$$

**Figure 3.3: Diagram of $\psi$, $\phi$ and $\phi'$**

Zippel Lemma

$$\Pr(\phi'(l) \neq 0) \geq 1 - \deg(l)/\mathrm{card}(\mathsf{R}).$$

The degree of the coefficients with respect to $X$ in $\psi(\hat{f}_1)$ and $\psi(\hat{f}_2)$ can be bounded from above by $d_1$ and $d_2$ respectively. There are $d_2 - \delta$ rows of entries in the matrix corresponding to $\mathrm{S}_\delta(\psi(\hat{f}_1),\ \psi(\hat{f}_2))$ of degree at most $d_1$ and there are $d_1 - \delta$ rows of entries of degree at most $d_2$. In the worst case of $\delta = 0$ we can bound the degree of all of the coefficients in above mentioned subresultant by $2d_1d_2$ and hence $l$. Using Lemma 1 with $\phi'$, $\psi(\hat{f}_1)$, $\psi(\hat{f}_2)$, $\mathsf{E} = \mathsf{K}[\alpha_2, \ldots, \alpha_n,\ \beta_2, \ldots, \beta_n]$ and assuming that $\phi'(l) \neq 0$ we can apply Lemma 1, which yields $\phi'(\tilde{g}_1) = g_2$ and $\deg(\tilde{g}_1) = \deg_X(g_2)$. □

Finally, we can prove Theorem 1.

*Proof of Theorem 1* The statements on the run time and required black box oracle calls of the algorithm and the returned program are easily verified. First we need to interpolate the univariate polynomials defined in Step 1, namely, $\bar{f}_0(X, 0)$ of degree at most $\max_{2 \leq i \leq r}\{\deg(f_i)\}$ and $\bar{f}_1(X, 0)$ of degree $\deg(f_1)$. There are $\deg(f_i) + 1$ many oracle calls to each individual black box of $f_i$ if the degree is known. If the polynomial is probabilistically guessed as described in Step 2, an extra oracle call for the check at $X = A$ is required. In either case the interpolation and single GCD needed to compute $\gamma_0(X) = \mathrm{GCD}(\bar{f}_0(X, 0), \bar{f}_1(X, 0))$ described in the algorithm can be accomplished in polynomial time.

The dominating work of the output program is Step A, the computation of GCDs of interpolated univariate polynomials. Let $d_0 = \deg(\bar{f}_0(X, 0))$, $d_1 = \deg(\bar{f}_1(X, 0))$, and let $\mathsf{S} \subset \mathsf{K}$ containing $\{1\}$ be of cardinality at least $d_0d_1 + \delta$.

As described, if $\deg(\bar{\gamma}_1) = \deg(\bar{\gamma}_0)$ the black box program for the GCD can terminate early only using $r+\sum_{i=1}^{r}\deg(f_i)$ many oracle calls. Both the interpolation and computation of $\bar{\gamma}_1(X)$ can be accomplished in polynomial time. The degree of $\gamma_0(X)$ is, with high probability (see Lemma 3), equal to $\delta$. Then, if $\deg(\bar{\gamma}_1) = \deg(\bar{\gamma}_0)$ the polynomial $\bar{\gamma}_1(X)$ is essentially the GCD described in Lemma 2 and is equal to the homomorphic image of $\mathrm{GCD}_{1\le i\le r}(f_i)$.

At worst $d_0d_1 + \delta$ elements from $\mathsf{S}$ are needed to compute $\bar{g}$. The GCD algorithm needs $\delta$ "lucky" values for $Y$ to interpolate $\bar{g}$, since the coefficients for $Y = 0$ are already available, and because there can be a maximum of $d_0d_1$ "unlucky" values, as we will argue below. There are $\deg(f_i)+1$ many oracle calls for each black box of $f_i$ needed to interpolate $\bar{f}_0(X,e)$ and $\bar{f}_1(X,e)$, that for every $e$ in $\mathsf{S}$. Hence, since only $d_0d_1 + \delta$ values from $\mathsf{S}$ must be used at the worst, the number of required black box oracle calls follows. Again both the interpolations of $\bar{f}_0(X,e)$ and $\bar{f}_1(X,e)$ and GCD computations can be accomplished in polynomial time. The number of "unlucky" values of $Y = e$ is derived as follows. Let $l(Y) = \mathrm{ldcf}_X(\mathrm{S}_\delta(\bar{f}_0(X,Y),\bar{f}_1(X,Y)))$. The degree of $l$ can be bounded by $d_0d_1$ and hence only $d_0d_1$ values can zero the leading coefficient of the subresultant, thus making the computed GCD invalid (cf. Lemma 1).

All that remains is to analyze the failure probabilities. From Lemma 2 we obtain that with probability at least $1 - \deg(f_1)/\mathrm{card}(\mathsf{R})$ we have

$$g = \mathrm{GCD}(f_1,\dots,f_r) = \mathrm{GCD}(f_1, f_2 + c_3f_3 + \cdots + c_rf_r),$$

when the elements $c_3,\dots,c_r$ are chosen at random from the set $\mathsf{R}$. Assuming that this is the case, Lemma 3 then yields that with probability at least $1 - 2\deg(f_0)\deg(f_1)/\mathrm{card}(\mathsf{R})$ we have $\mathrm{GCD}(\bar{f}_0(X,0),\bar{f}_1(X,0)) = \bar{g}(X,0)$, where the barred polynomials are defined in Step 1. Furthermore, the leading coefficient of $\bar{g}(X,Y)$ is independent of $Y$, since by Lemma 3 we also have $\deg(g) = \deg(\bar{g}(X,0))$. Since by virtue of homomorphic imaging we have that $\bar{\gamma}_0$ is a polynomial multiple of $\gamma(X,0)$, these conditions imply that $\gamma(X,Y) = \mathrm{GCD}(\bar{f}_0(X,Y),\bar{f}_1(X,Y)) = \bar{g}(X,Y)$ and that the polynomial $\gamma$ as determined in Step B is the image of one and the same associate of $g$, namely the one whose leading coefficient in $X$ has been preselected. Both events occur with probability no less than the product of the stated bounds,

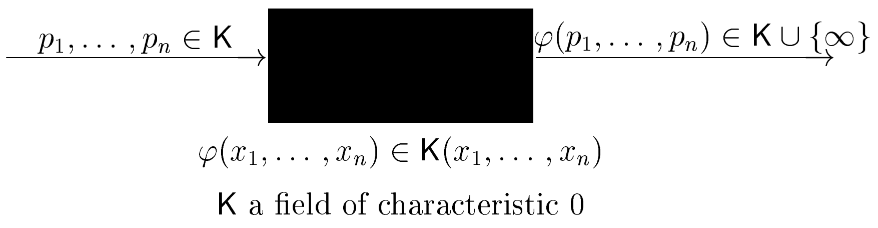

$p_1, \ldots, p_n \in \mathsf{K}$
$\varphi(p_1, \ldots, p_n) \in \mathsf{K} \cup \{\infty\}$
$\varphi(x_1, \ldots, x_n) \in \mathsf{K}(x_1, \ldots, x_n)$
$\mathsf{K}$ a field of characteristic 0

**Figure 3.4: Black box rational function**

which yields the given estimate. $\square$

## 3.4 Numerator and Denominator Separation

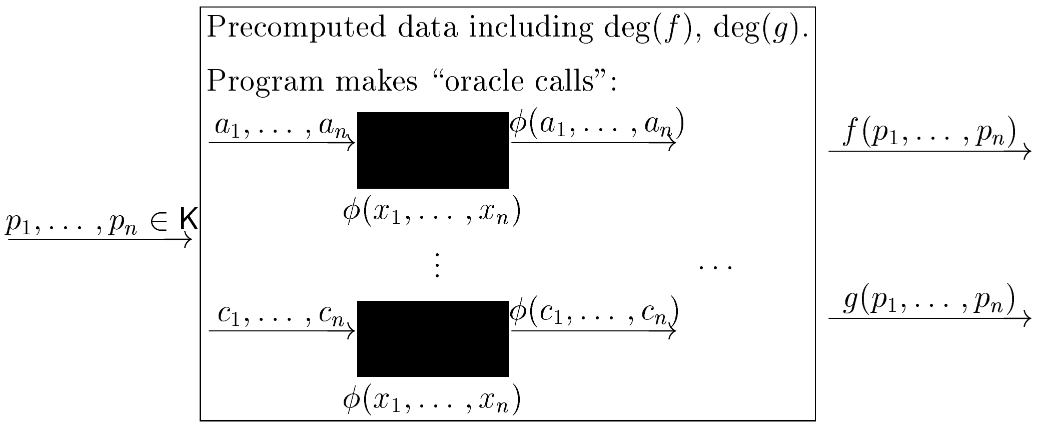
A black box rational function $\varphi(x_1, \ldots, x_n) \in \mathsf{K}$ takes as input values for each of the variables $p_1, \ldots, p_n$ and produces the value of the rational function at that point (see Figure §3.4). Furthermore, the rational function black box must be able to handle situations where the input point is a zero of the denominator. In such cases, it should return a "special" value or trigger an alarm.

The black box numerator and denominator algorithm takes as input a black box that evaluates the rational function

$$\varphi(x_1, \ldots, x_n) = f(x_1, \ldots, x_n)/g(x_1, \ldots, x_n) \in \mathsf{K}(x_1, \ldots, x_n)$$
$$\text{where } f, g \in \mathsf{K}[x_1, \ldots, x_n],\ \mathrm{GCD}(f, g) = 1,$$

a failure probability $\epsilon \ll 1$, and a degree bound $\bar{e} \geq \deg(g)$. Note that without a degree bound for the denominator, there is a chance that the algorithm enters an infinite loop.

The output (see Figure §3.5) is an evaluation program which accepts field elements $p_1, \ldots, p_n$ and with high probability $(1 - \epsilon)$ returns the values of both the numerator $f(p_1, \ldots, p_n)$ and denominator $g(p_1, \ldots, p_n)$ at the input point. It should be noted that as with the factor and greatest common divisor black box algorithms, the numerator and denominator construction algorithm chooses an associate for $f$ and $g$. Upon repeated probes, the algorithm returns a value of both the numerator and denominator scaled by the previously chosen multiple.



$$\phi(x_1, \dots, x_n) = \frac{f(x_1, \dots, x_n)}{g(x_1, \dots, x_n)}, f, g \in \mathsf{K}[x_1, \dots, x_n], \mathrm{GCD}(f, g) = 1.$$

**Figure 3.5: Black box numerator/denominator**

Note that with probability at least $1-\epsilon$, the output program is correct and that a correct program will always produce a valid associate for both the numerator and denominator. The probability of success depends on the selection of random field elements chosen during the construction phase of the algorithm. If the cardinality of the set from which the random elements are chosen $R$ is at least

$$\max\Big(2\,(2\deg(f)+1)\deg(g), 3m^2 - m\Big)\Big/\epsilon, \quad m = \max(\deg(f), \deg(g)),$$

then the resulting program correctly evaluates the numerator and denominator with probability no less than $1-\epsilon$.

In what follows, we provide a description of our improvement to the black box numerator and denominator separation algorithm originally described by Kaltofen and Trager. It is necessary that the reader refer to [KT90] for a detailed exposition and full analysis of this algorithm since we describe our modification utilizing notations and concepts defined there.

Like the previous black box algorithms, the numerator and denominator algorithm reduces the problem of computing with polynomials in $n$ variables to that of two. A special overline operator $\bar{}$ given by

$$\bar{h}(X,Y) = h(X, a_2X + b_2 + Y(p_2 - a_2p_1 - b_1), \dots, a_nX + b_n + Y(p_n - a_np_1 - b_n))$$
$$\text{where } a_i, b_i, p_i \in \mathsf{K}$$

maps both the numerator and denominator to the bi-variate polynomials $\bar{f}(X,Y)$ and $\bar{g}(X,Y)$ respectively. During the evaluation process, the images $\bar{f}(X,y_i)$ and $\bar{g}(X,y_i)$ for $1 \leq i \leq j = \max(\deg(f), \deg(g)) + 1$ are computed. From these candidates, the full bi-variate polynomials $\bar{f}$ and $\bar{g}$ are interpolated. By virtue of the overline operator, the value of $f$ and $g$ can be retrieved at a point $p_1, \dots, p_n$ by evaluating $\bar{f}(p_1, 1)$ and $\bar{g}(p_1, 1)$ respectively. Our improvement follows from an observation that while computing image candidates for $y_i$, we have for $y_i = 1$ the polynomials $\bar{f}(X,1)$ and $\bar{g}(X,1)$. At this stage if the degree of both $\bar{f}(X,1)$ and $\bar{g}(X,1)$ are the same as the previously computed bounds for the numerator and denominator respectively, we can terminate early and simply return the values of $\bar{f}(X,1)$ and $\bar{g}(X,1)$ at $X = p_1$. In many cases, this improvement markedly increases performance of the numerator and denominator algorithm since we can avoid $\max(\deg(f), \deg(g))$ many Euclidean algorithm steps. A similar observation was employed to improve the performance of our GCD algorithm detailed in §3.3. It turns out that in the examples of Table 4.4 in Section 4 this shortcut is always taken.

# CHAPTER 4
# BENCHMARK 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 and total "work". Our work metric is measured in SPECint92-hours[GA95]. SPEC (Standard Performance Evaluation Corporation) is a non-profit corporation formed to, "establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers". The results of each SPEC benchmark are expressed as the ratio of the wall clock time to execute one single copy of the benchmark, compared to a fixed SPEC reference time. The SPECint92 metric is a geometric mean of SPEC ratios of 8 benchmarks which were compiled with aggressive optimizations.

Each of our benchmarks are problems which can not 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. In §4.1, we provide the results of computing the factors of a symmetric Toeplitz determinant over a cluster of workstations. The timings of computing the greatest common divisor of the determinant of two Vandermonde matrices which share two variables in common are detailed in §4.2. Finally, in §4.3, we nest several black box objects to attain a single factor of the numerator of a Cauchy determinant.

## 4.1 Factorization Benchmark

For $n > 0$ the Toeplitz matrix given by $T_n = (t_{i-j}) = (a_{i-j+n-1}),\ i, j = 1, \ldots, n$ is a matrix of the form:

$$T_n = \begin{bmatrix} a_{n-1} & a_{n-2} & \cdots & a_1 & a_0 \\ a_n & a_{n-1} & \cdots & a_2 & a_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{2n-3} & a_{2n-2} & \cdots & a_{n-1} & a_{n-2} \\ a_{2n-2} & a_{2n-3} & \cdots & a_n & a_{n-1} \end{bmatrix}.$$

The elements along the leading diagonal or along any line parallel to the leading diagonal are equal and $T_n$ is completely determined by its' first row and column. A symmetric Toeplitz matrix $S_n$ can be defined as follows:

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

The determinant of a symmetric Toeplitz matrix, $S_n$, has two factors over the rational numbers 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 4.1 provides total CPU times and work measures 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 that evaluates the value of the pre-image modulo a prime. Finally, we perform a parallel modular sparse conversion to retrieve

| $N$ | CPU Time | Work |
|---|---|---|
| 10 | $0^h1'$ | $4^\#$ |
| 11 | $0^h2'$ | $8^\#$ |
| 12 | $0^h5'$ | $21^\#$ |
| 13 | $0^h9'$ | $37^\#$ |
| 14 | $0^h16'$ | $67^\#$ |
| 15 | $0^h26'$ | $109^\#$ |
| 16 | $0^h43'$ | $181^\#$ |
| 17 | $1^h5'$ | $273^\#$ |
| 18 | $1^h42'$ | $428^\#$ |
| 19 | $2^h30'$ | $630^\#$ |
| 20 | $3^h42'$ | $932^\#$ |

**Table 4.1: Black box factorization benchmark results - construction**
Total CPU times (hours$^h$minutes$'$) required to construct a factors black box (over Q) which can evaluate both irreducible factors of the determinant of a symmetric Toeplitz matrix. The processor is a Sun Ultra1/170 (128MB) rated at 252 SPECint92. Total work is measured in units of (SPECint92-hours$^\#$).

the factor's distributed representation. This portion of the benchmark application utilized the SACLIB modular polynomial arithmetic wrapper/adaptor and an MPI compliant implementation of the parallel black box interface.

Table 4.2 provides the complete CPU times and work measures for the factorization benchmark problem. This benchmark application performed the actual sparse conversion algorithm on a Sun Ultra1/170 for the $N = 10$ case and on a Sun 20/50 for the remaining runs, each "driver" program executed with a resident set of 60 megabytes of memory. Each of the factor black box evaluations required by the sparse conversion algorithm was dispatched as a "parallel task." All of the parallel tasks performed modular evaluations utilizing only 10 megabytes of memory.

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

| $N$ | Task | | Degree | # Terms |
|---|---|---|---|---|
| 10 | (6) Sun 20/50 | $2^h54'$ | 5 | 931 |
| | (2) Sun 5/70 | $1^h07'$ | | |
| | (6) Sun 5/110 | $2^h27'$ | | |
| | (1) Ultra1/170 | $4^h52'$ | | |
| | **Total Work** | $1706^{\#}$ | | |
| 11 | (9) Sun 20/50 | $28^h25'$ | 5 | 847 |
| | (1) Sun 5/70 | $1^h04'$ | | |
| | (2) Sun 5/110 | $1^h33'$ | | |
| | (1) Sun 10/51 | $0^h44'$ | | |
| | (1) Sun 10/20 | $1^h58'$ | | |
| | (1) Ultra1/170 | $0^h25'$ | | |
| | **Total Work** | $2599^{\#}$ | | |
| 12 | (9) Sun 20/50 | $130^h15'$ | 6 | 5577 |
| | (1) Sun 5/70 | $5^h19'$ | | |
| | (2) Sun 5/110 | $7^h34'$ | | |
| | (1) Sun 10/51 | $3^h45'$ | | |
| | (1) Sun 10/20 | $9^h25'$ | | |
| | (1) Ultra1/170 | $2^h10'$ | | |
| | **Total Work** | $12080^{\#}$ | | |
| 13 | (9) Sun 20/50 | $333^h19'$ | 6 | 4982 |
| | (1) Sun 5/70 | $11^h03'$ | | |
| | (2) Sun 5/110 | $15^h53'$ | | |
| | (1) Sun 10/51 | $6^h35'$ | | |
| | (1) Sun 10/20 | $17^h34'$ | | |
| | (1) Ultra1/170 | $4^h33'$ | | |
| | **Total Work** | $29785^{\#}$ | | |

**Table 4.2: Black box factorization benchmark results - sparse conversion** CPU times (hr$^h$min$'$) to retrieve the distributed representation of the factors black box of a symmetric Toeplitz determinant black box. Construction is over Q evaluation is in GF($10^8 + 7$). 15 nodes, each rated by SPECint92: Sun 20/50(64MB) 76.9, Sun 5/70(64MB) 57.0, Sun 5/110(40MB) 78.6, Sun 10/51(128MB) 65.2, Sun 10/20(64MB) 39.8, Ultra1/170(128MB) 252. Parallel work is in SPECint92-hrs$^{\#}$.

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 (see §5.2).

Our modifications to Zippel's sparse conversion algorithm proved quite useful for our benchmark example. Figure 4.1 illustrates the number of evaluations required by two different sparse conversion algorithms for interpolating a factor of our 10 by 10 benchmark problem. The `SInterp` plot details the number of evaluations employed for each interpolation step of Kaltofen's version of Zippel's sparse

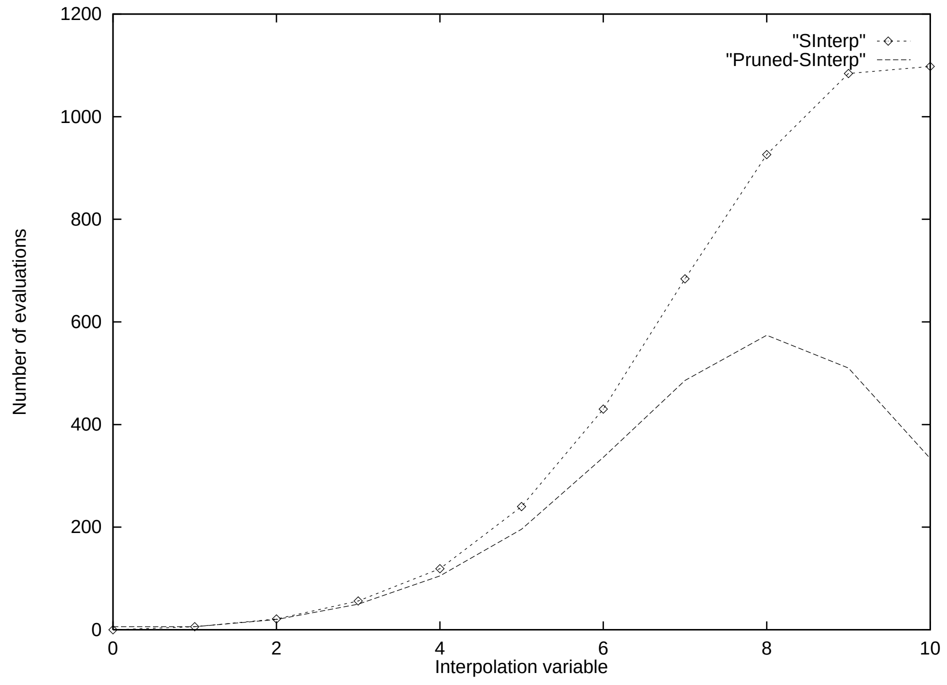


**Figure 4.1: Evaluations - sparse conversion vs. pruned sparse conversion**

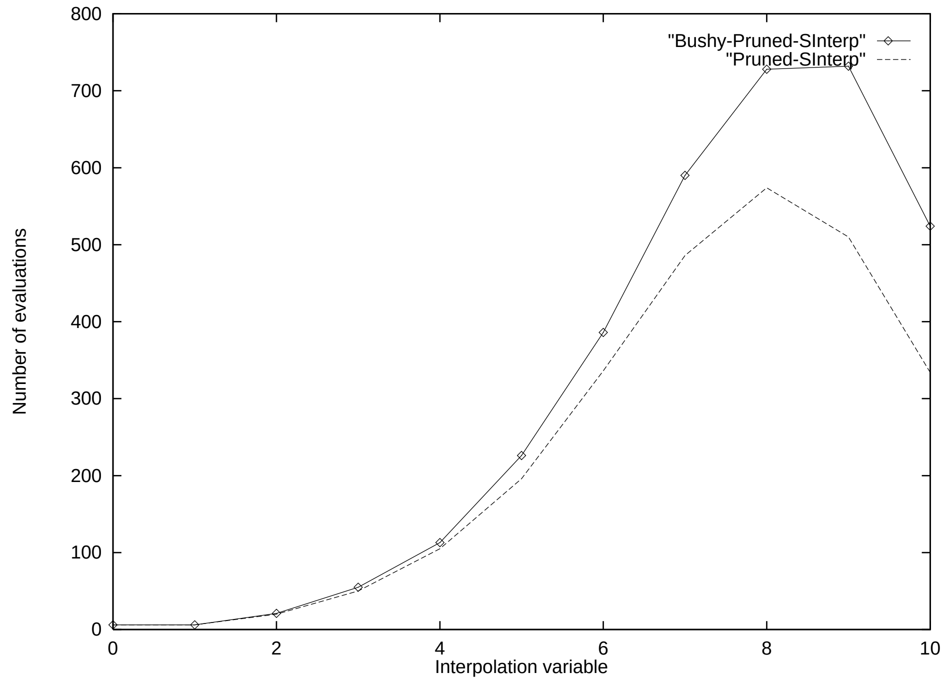


**Figure 4.2: Pruned evaluations for each sparse conversion step**

conversion algorithm [Kal89]. The `Pruned-SInterp` plot details the same information for our pruning sparse conversion algorithm. By pruning terms which 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 (see §3.1.1). Figure 4.2 depicts the number of black box calls required before (`Bushy-Pruned-SInterp`) and after (`Pruned-SInterp`) the pruning phase for each interpolation step of our pruning sparse conversion. Indeed, it was a combination of the concepts of black box homomorphic maps, term pruning and parallel black boxes which provided the tools necessary for the successful completion of our factorization benchmark problems.

## 4.2 Greatest Common Divisor Benchmark

For $n > 0$, Vandermonde's matrix $V(P)$ formed from elements of a list $P = \{x_1, \ldots, x_n\}$ is a square matrix which has as its' $(i, j)$-th entry $P[i]^{(j-1)}$. In its' expanded form, Vandermonde's matrix can be defined as follows:

$$V(x_1, \ldots, x_n) = \begin{bmatrix} 1 & x_1 & \ldots & x_1^{n-1} \\ 1 & x_2 & \ldots & x_2^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & x_n & \ldots & x_n^{n-1} \end{bmatrix}.$$

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 $V_1 = \det(V(x_1, \ldots, x_n))$ and $V_2 = \det(V(x_1, \ldots, x_k, y_{k+1}, \ldots, y_n))$, then 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 $G(x_1, x_2) = x1 - x2$. These computations provide benchmark timings for FoxBox's Vandermonde common object, greatest common divisor black box object, and sparse conversion algorithm.

| $N$ | Task | Q | GF(P) |
|---|---|---|---|
| 10 | total time | $0^h1'$ | $0^h0'$ |
| | work | $4^{\#}$ | $0^{\#}$ |
| 15 | total time | $0^h14'$ | $0^h0'$ |
| | work | $59^{\#}$ | $0^{\#}$ |
| 20 | total time | $1^h43'$ | $0^h0'$ |
| | work | $433^{\#}$ | $0^{\#}$ |
| 25 | total time | $8^h42'$ | $0^h2'$ |
| | work | $2192^{\#}$ | $8^{\#}$ |
| 30 | total time | $35^h36'$ | $0^h5'$ |
| | work | $8971^{\#}$ | $21^{\#}$ |

**Table 4.3: Black box greatest common divisor benchmark results**
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 Q and in GF($10^{16} + 61$). The processor is a Sun Ultra1/170(256MB) rated at 252 SPECint92. Total work is measured in units of (SPECint92-hours$^{\#}$).

The solution of this problem within the black box framework is first to construct two Vandermonde black box objects, one representing $V_1$ and the other $V_2$. Second, 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 4.3 provides the complete CPU times and work measures for the greatest common divisor benchmark problems. Timings for our arithmetic over Q and in GF($10^{16}+61$) employed SACLIB's rational polynomial arithmetic wrapper/adaptor and NTL's fast modular polynomial arithmetic wrapper/adaptor respectively. These benchmark applications performed their computations on a Sun Ultra1/170 with a resident set of 60 megabytes of memory.

## 4.3 Numerator and Denominator Benchmark

For $n > 0$, Cauchy's matrix can be defined in its expanded form as follows:

$$C(n,n) = \begin{bmatrix} \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \cdots & \frac{1}{x_1+y_n} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \cdots & \frac{1}{x_2+y_n} \\ \vdots & \vdots & & \vdots \\ \frac{1}{x_n+y_1} & \frac{1}{x_n+y_2} & \cdots & \frac{1}{x_n+y_n} \end{bmatrix}.$$

The determinant of Cauchy's matrix can be expressed by the following equation:

$$\det(C(n,n)) = \prod_{1 \le i < j \le n} (x_j - x_i)(y_j - y_i) \Bigg/ \prod_{1 \le i,j \le n} (x_i + y_j).$$

We selected as a benchmark the problem of computing a factor of the reduced numerator and denominator of $\det(C(n,n))$ for $n = 5 \ldots 12$. This particular problem involves the nesting of the black boxes. First, a black box numerator and denominator is constructed from a Cauchy common object. Secondly, a factor black box is constructed from the numerator of the previously computed numerator/denominator black box. Finally, the factor is retrieved in its distributed format by employing our sparse conversion algorithm. The sparse conversion algorithm probes the factor black box for the purpose of interpolation. For each value provided, the factor black box probes the numerator/denominator black box to compute its value; in turn, the numerator/denominator black box evaluates the Cauchy common object to produce the value for the factor black box.

Table 4.4 details the total CPU times and work measures for our benchmark problem for $n = 5 \ldots 12$. Each problem was executed on a Sun Ultra1/170, with all arithmetic in GF($10^{16}$+61). The benchmark application utilized NTL's fast modular polynomial arithmetic wrapper/adaptor. Our sparse conversion algorithm takes a variable by variable approach for interpolation and is sensitive to variable order. Since the black box factors object does not distinguish between factors, our results vary in the number of required evaluations. The number of evaluations required by the sparse conversion algorithm depends on which two variables are present in a

| $N$ | Total Time | Work | # Calls |
|---|---|---|---|
| 5 | $1^h29'$ | 374 | 18 |
| 6 | $3^h47'$ | 953 | 22 |
| 7 | $3^h44'$ | 940 | 10 |
| 8 | $24^h42'$ | 6224 | 28 |
| 9 | $75^h57'$ | 19139 | 38 |
| 10 | $52^h30'$ | 13230 | 12 |
| 11 | $282^h56'$ | 71299 | 30 |
| 12 | $684^h25'$ | 172473 | 38 |

**Table 4.4: Black box numerator/denominator benchmark results**
Total CPU times (hours$^h$minutes$'$) required to retrieve the distributed representation of a factor of the numerator of a Cauchy matrix determinant, each symbolic object is in black box representation. All arithmetic is in GF($10^{16} + 61$). The processor is a Sun Ultra1/170(256MB) rated at 252 SPECint92. Total work is measured in units of (SPECint92-hours$^{\#}$).

particular factor. Clearly, another interpolation technique that does not depend on variable order could reduce the number of required evaluations for this particular example. It is expected that future enhancements to FoxBox will provide a suite of interpolation routines from which a user can select an appropriate algorithm.

For this particular example where each of the factors are of small degree, the factor box evaluator can be dramatically improved by probing the numerator and denominator black box at truncated power series as values for the variables. An extended domain version of the numerator and denominator algorithm would provide this functionality and thus is a focus of future work as detailed in §5.2.

# CHAPTER 5
# RESEARCH CONTRIBUTIONS AND FUTURE DIRECTIONS

## 5.1 Summary of Research Contributions

We now summarize the contributions of our research:

**1.** We provide improvements to several algorithms found in the literature for manipulating black boxes:

- Sparse Interpolation

  We provide a full description of our Zippel based pruning sparse conversion which employs a novel term pruning technique with the overall goal of reducing the number of black box evaluations. Such evaluations being the most costly portion of each black box algorithm.

- Factorization

  We present two mechanisms for reducing the costly bi-variate interpolation associated with factor box evaluation. First, we apply our term pruning technique to reduce by half the complexity and number of black box evaluation points required by the bi-variate interpolation. Secondly, we describe the notion of extended domain black boxes which can dramatically improve performance of the bi-variate interpolation step for small degree factors.

- Greatest Common Divisor

  We furnish a full algorithmic description and analysis of a new algorithm for computing the greatest common divisor of polynomials in black box format.

- Reduced Numerator and Denominator Separation

  We describe a technique first observed in our GCD algorithm which, in most cases, markedly increases performance of the black box reduced numerator and denominator algorithm.

**2.** Through the implementation of FoxBox, we demonstrate a new methodology for designing "component" symbolic software as well as provide the first tool for manipulating symbolic objects given by black boxes: Several features include:

– 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 implementation derived from our prototypical efforts

– Maple interface to the C++ FoxBox server

**3.** Our benchmark computations represent the first ever symbolic solutions of such problems and hence prove the viability of the black box representation in symbolic computing. With FoxBox we provide solutions for the following challenge problems:

- Compute a factor of a 13 by 13 symmetric Toeplitz determinant.
- Provide the GCD of two 30 by 30 Vandermonde matrices with two variables in common.
- Attain a single factor of the numerator of a Cauchy determinant of dimension 12 by 12.

## 5.2 Future Challenges

### 5.2.1 Open Problems

The most costly portion of the black box factorization algorithm is the bi-variate interpolation step required for evaluation. While we provide two methods for alleviating the cost associated with this step, the bi-variate interpolation still represents a bottleneck area. We have begun looking into the possibility of Hensel [Mus75] lifting with respect to a non-standard basis. This would allow for the combination of the interpolation and lifting stages to occur in an optimal fashion. Also, alternate factorization algorithms will be considered, for instance [RZ95].

Our current implementation of Kaltofen and Lakshman's algorithm for solving a transposed Vandermonde system does not take advantage of several arithmetic optimizations necessary to achieve a quasi-linear running time. A potential bottleneck for our implementation of the pruning sparse conversion algorithm is the fact that at each interpolation step we solve one large linear system instead of several smaller systems as described in Zippel [Zip90]. Perhaps, a heuristic approach which gauges the cost of black box evaluation and the cost of linear system solving could be employed to dynamically "switch" interpolation techniques in order to further optimize performance. Note that this heuristic approach is not advantageous when the transposed Vandermonde system solver executes in quasi-linear time (see §3.1.1).

Clearly the black box representation of symbolic objects provides for computing with such objects which can not be manipulated by other means by alleviating intermediate expression swell. Currently we are conducting research into the possibility of developing black box algorithms for computing derivatives and Gröbner bases [BW93].

Finally, drawing from the empirical data generated from our factorization benchmark problem, we are currently attempting to provide an exact formulation for the number of terms and degree of each factor of an $N$ dimensional Toeplitz matrix determinant.

### 5.2.2 System Enhancements

As for improvements and modifications to our FoxBox system, several outstanding points remain. Most notably, the notion of extended domain black boxes. The solution of our numerator and denominator benchmark problem can be greatly improved with a extended domain version of this algorithm.

By virtue of the design of the FoxBox server, we can easily provide interfaces to several different "front-ends". In hopes of furnishing the functionality offered by FoxBox to a more wide spread audience, the possibility of suppling "bridge" code to different computer algebra systems is currently under investigation. Furthermore, the "calculator-style" interface can be relaxed by providing for the transparent exchange of black box objects between the FoxBox server and a general purpose computer algebra system. Our first attempt will be with Maple, where we already have a complete native implementation of the calculus of black boxes.

In the realm of black box parallelization, there remains room for improvement. We provide a simple API for distributing black boxes in a coarse grain fashion. However, drawing from our experiences with DSC, the parallel black box interface can be enhanced to provide additional functionality such as a choice of distribution strategies. Furthermore, the black box GCD and numerator/denominator algorithms can greatly benefit from the parallelization of their respective evaluation steps. Certainly, administering the many possible combinations of parallel tasks is an area of research.

Both the factorization and numerator/denominator algorithm computes and returns several values. Both algorithms can be modified to produce, say, one factor value or just the value of the numerator. In the case where just a single value is required, the modified factorization and numerator/denominator algorithms can reduce the overall cost of evaluation.

Currently, we provide one improved method of converting black boxes into sparse format. While this method optimizes the required number of black box evaluations, its' efficiency greatly depends on the order of variables in which one interpolates. Perhaps, FoxBox can be extended to provide a suite of interpolation algorithms targeted for different applications.

Associated with each black box algorithm is a probability measure. With the potential for nesting of black boxes, this measure can become unattainable. The reliability of the outermost black box depends on the reliability of each internally nested black box. A "self-tuning" feature that allows nested black boxes to "reconstruct" utilizing a different probability metric would alleviate the potential for an unattainable probability.

# LITERATURE CITED

[ACS86] S. K. Abdali, G. W. Cherry, and N. Soiffer. An object-oriented approach to algebra system design. In B. W. Char, editor, *Proc. 1986 ACM Symp. Symbolic Algebraic Comp.*, pages 24–30. ACM, 1986.

[ADS96] J. Abbott, A. Díaz, and R. S. Sutor. A report on OpenMath a protocol for the exchange of mathematical information. *SIGSAM Bulletin*, 30(1):21–24, 1996.

[AS85] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Program.* MIT Press, Cambridge, MA, 1985.

[Bro71] W. S. Brown. On Euclid's algorithm and the computation of polynomial greatest common divisors. *J. ACM*, 18:478–504, 1971.

[BT71] W. S. Brown and J. F. Traub. On Euclid's algorithm and the theory of subresultants. *J. ACM*, 18:505–514, 1971.

[BW93] T. Becker and V. Weispfenning. *Gröbner bases A Computational Approach to Commutative Algebra.* Springer Verlag, New York, N.Y., 1993.

[CDK94] K. C. Chan, A. Díaz, and E. Kaltofen. A distributed approach to problem solving in Maple. In R. J. Lopez, editor, *Maple V: Mathematics and its Application*, Proceedings of the Maple Summer Workshop and Symposium (MSWS'94), pages 13–21, Boston, 1994. Birkhäuser.

[CGG89] B. W. Char, K. O. Geddes, and G. H. Gonnet. GCDHEU: Heuristic polynomial GCD algorithm based on integer GCD computation. *J. Symbolic Comput.*, 7(1):31–48, 1989.

[DHK$^+$95] A. Díaz, M. Hitz, E. Kaltofen, A. Lobo, and T. Valente. Process scheduling in DSC and the large sparse linear systems challenge. *J. Symbolic Comput.*, 19(1–3):269–282, 1995.

[DK95] A. Díaz and E. Kaltofen. On computing greatest common divisors with polynomials given by black boxes for their evaluation. In A. H. M. Levelt, editor, *Proc. Internat. Symp. Symbolic Algebraic Comput. ISSAC '95*, pages 232–239, New York, N. Y., 1995. ACM Press.

[DK97a] A. Díaz and E. Kaltofen. The FoxBox source code. Link to ftp document `ftp://ftp.cs.rpi.edu/pub/kaltofen/foxbox/`, Rensselaer Polytechnic Instit., Dept. Comput. Sci, Troy NY, 1997.

[DK97b] A. Díaz and E. Kaltofen. Foxbox: A system for manipulating symbolic objects in black box representation. Report in preparation, Rensselaer Polytechnic Instit., Dept. Comput. Sci, Troy NY, 1997.

[Dre77] F. J. Drexler. Eine methode zur berechnung sämtlicher lösungen von polynomgleichungssystemen. *Numer. Math.*, 29:45–58, 1977. (In German.).

[ES95] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, Reading, MA, 1995.

[FIKL88] T. S. Freeman, G. Imirzian, E. Kaltofen, and Lakshman Yagati. DAGWOOD: A system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Software*, 14(3):218–240, 1988.

[GA95] R. Giladi and N. Ahituv. SPEC as a performance evaluation measure. *Computer*, 28(August):33–42, 1995.

[GBD$^+$94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine A Users' Guide and Tutorial for Network Parllel Computing.* MIT Press, Cambridge, Massachusetts, 1994.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, MA, 1994.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, Massachusetts, 1994.

[Gra93] T. Granlund. *The GNU Multiple Precision Arithmetic Library.* Free Software Foundation, Cambridge, MA, 1993.

[HNS95] H. Hong, A. Neubacher, and W. Schreiner. The design of the SACLIB/PACLIB kernels. In A. Miola, editor, *Design and Implementation of Symbolic Computation Systems*, volume 722 of *Springer Lect. Notes Comput. Sci.*, pages 288–302, Heidelberg, Germany, 1995. Springer Verlag.

[JS92] R. D. Jenks and R. S. Sutor. *axiom The Scientific Computing System.* Springer Verlag, New York, 1992.

[Kal85a] E. Kaltofen. Computing with polynomials given by straight-line programs I; Greatest common divisors. *Proc. 17th ACM Symp. Theory Comp.*, pages 131–142, 1985.

[Kal85b] E. Kaltofen. Computing with polynomials given by straight-line programs II; Sparse factorization. *Proc. 26th IEEE Symp. Foundations Comp. Sci.*, pages 451–458, 1985.

[Kal85c] E. Kaltofen. Sparse Hensel lifting. *Proc. EUROCAL '85, Vol. 2, Springer Lec. Notes Comp. Sci.*, 204:4–17, 1985.

[Kal88] E. Kaltofen. Greatest common divisors of polynomials given by straight-line programs. *J. ACM*, 35(1):231–264, 1988.

[Kal89] E. Kaltofen. Factorization of polynomials given by straight-line programs. In S. Micali, editor, *Randomness and Computation*, volume 5 of *Advances in Computing Research*, pages 375–412. JAI Press Inc., Greenwhich, Connecticut, 1989.

[Kal92] E. Kaltofen. On computing determinants of matrices without divisions. In P. S. Wang, editor, *Proc. Internat. Symp. Symbolic Algebraic Comput. ISSAC '92*, pages 342–349, New York, N. Y., 1992. ACM Press.

[KL88] E. Kaltofen and Yagati Lakshman. Improved sparse multivariate polynomial interpolation algorithms. *Proc. ISSAC '88, Springer Lect. Notes Comput. Sci.*, 358:467–474, 1988.

[KT90] E. Kaltofen and B. Trager. Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators. *J. Symbolic Comput.*, 9(3):301–320, 1990.

[Mos71] J. Moses. Algebraic simplification: A guide for the perplexed. *Commun. ACM*, 14:548–560, 1971.

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide C++ Programming with the Standard Template Library.* Addison-Wesley Publ. Comp., Reading, Massachusetts, 1996.

[Mus75] D. R. Musser. Multivariate polynomial factorization. *J. ACM*, 22:291–308, 1975.

[MY73] J. Moses and D. Y. Y. Yun. The EZ-GCD algorithm. *Proc. 1973 ACM National Conf.*, pages 159–166, 1973.

[Odl96] A. Odlyzko. Computer algebra and its applications: Where are we going? Paper accompanying a lecture at the Joint Annual Meeting of the German GI and the Austrian Computergesellschaft in Klagenfurt, Austria, September 1996.

[RZ95] R. Rubinfeld and R. Zippel. A new modular interpolation algorithm for factoring multivariate polynomials. In *Algorithmic Number Theory*, volume 877 of *Springer Lecture Notes Comput. Sci.*, pages 93–107, Heidelberg, Germany, 1995. Springer Verlag.

[Sho95] V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symbolic Comput.*, 20(4):363–397, 1995.

[Sho97] V. Shoup. NTL: A library for doing number theory. Link on html document `http://www.cs.wisc.edu/~shoup/`, Univ. Wisconsin, Dept. Comput. Sci, 1997.

[Wan80] P. S. Wang. The EEZ-GCD algorithm. *SIGSAM Bulletin*, 14/2:50–60, 1980.

[WBD$^+$94] S. M. Watt, P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morrison, J. M. Steinbach, and R. S. Sutor. A first report on the $A^\sharp$ compiler. In J. von zur Gathen and M. Giesbrecht, editors, *Proc. Internat. Symp. Symbolic Algebraic Comput. ISSAC '94*, pages 25–31, New York, N. Y., 1994. ACM Press.

[Zip79] R. Zippel. Probabilistic algorithms for sparse polynomials. *Proc. EUROSAM '79, Springer Lec. Notes Comp. Sci.*, 72:216–226, 1979.

[Zip90] R. Zippel. Interpolating polynomials from their values. *J. Symbolic Comput.*, 9(3):375–403, 1990.