Generic Matrix Multiplication and Memory Management in LinBox[*]

Erich Kaltofen
Dept. of Mathematics
North Carolina State University
Raleigh, North Carolina
27695-8205, USA
kaltofen@math.ncsu.edu

Dmitriy Morozov
Department of Computer Science
Duke University
Durham, North Carolina
27708-0129, USA
morozov@cs.duke.edu

George Yuhasz
Dept. of Mathematics
North Carolina State University
Raleigh, North Carolina
27695-8205, USA
gyuhasz@math.ncsu.edu

ABSTRACT

We describe the design and implementation of two components in the LinBox library. The first is an implementation of black box matrix multiplication as a lazy matrix-times-matrix product. The implementation uses template meta-programming to set the intermediate vector type used during application of the matrix product. We also describe an interface mechanism that allows incorporation of external components with native memory management such as garbage collection into LinBox. An implementation of the interface based on SACLIB's field arithmetic procedures is presented.

Categories and Subject Descriptors

I.1.3 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation—*Languages and Systems*; D.1.5 [**Software**]: Programming Techniques—*Object-oriented programming*

General Terms

design, languages

Keywords

black box matrix, C++ templates, C++ allocator, system integration, garbage collection, memory management, exact linear algebra

1. INTRODUCTION

LinBox is a C++ template library that provides generic implementations of black box linear algebra algorithms [5]. The library was developed by a consortium of universities in Canada, France and the USA. See http://www.linalg.org for the list of participating researchers and for the open source code. Our goal is to supply "efficient black box solutions for a variety of problems including linear equations and matrix normal forms with the guiding design principle of re-usability" [5]. The LinBox library utilizes two different abstraction devices. The first is algorithmic, and it introduces the notion of a *black box matrix* [10][†], which is a matrix representation by a procedure that efficiently computes the product of the matrix times an arbitrary vector. The second abstraction is the programming methodology of *generic, reusable software design*. We use the C++ template instantiation mechanism to compile code for the most efficient ways of performing the arithmetic in the various entry fields [5, 6, 13]. This paper describes two components that exhibit the generic programming techniques LinBox provides. The first is the implementation of blackbox matrix multiplication and the second is the incorporation of external libraries that utilize garbage collection.

We have implemented matrix multiplication by a lazy matrix-times-matrix product for the black box matrix type of the LinBox library. Let A, B be black box matrices that have matrix-times-vector functions $y = Ax$ and $y = Bx$, where x and y are vector objects. In LinBox, the matrix-times-vector functions are named "apply", and are member template functions with parametric types for both the input vector x and the output vector y, which can be sparse or dense vector types or columns of matrices. The apply function for the matrix product $y = (A \cdot B)x$ is implemented as the function composition $z = Ax$; $y = Bz$. The issue is the choice for the vector data type of the intermediate vector z. Each black box matrix class defines a preferred input and a preferred output vector type. Our composed black box class now has a template parameter switch that lets the user choose at compile time different vector types for z: either the preferred output type of B or preferred input type of A. Or one may force a conversion from the preferred output vector type of B to the preferred input vector of A, or select a default intermediate vector type. Of course, neither must be done when the output/input types are the same. We use C++'s partial template specialization rules for building the proper instantiations of the template class for composition (see sections 2 and 3).

In sections 4–6, we describe an interface mechanism by which one can plug a library of functions whose objects are

[*]This material is based on work supported in part by the National Science Foundation (USA) under Grants CCR-0113121 and CCR-0305314.

[†]The term "black box" matrix seems to have first been coined in our paper.

garbage collected, such as Maple procedures or Java methods, into our LinBox algorithms. Our benchmark test is with SACLIB's [15, 2] modular integer arithmetic. The problem is that LinBox algorithms need to allocate temporary intermediate values which are arrays of SACLIB's modular digits that must be registered with SACLIB's garbage collector. Watt [16] gives a solution in the Aldor-Maple setting. Our solution is based on C++'s STL allocator template class, so that genericity in our algorithms is maintained and minimal reprogramming is needed.

2. DESIGN ISSUES OF THE COMPOSITION CLASS

The design of the composition class described in this paper is predicated on a change made in the *black box archetype* in LinBox. In the initial design, all black box matrices were templated by the vector type they expected as input and output vectors [5]. The current version of the *black box archetype* moves the vector type template parameter from the archetype itself to its member functions. The methods `apply` and `applyTranspose` are member template functions [9, 14.5.2], with two template parameters, an input vector type and an output vector type. A member template `apply` allows the design of a generic matrix times vector function that can be instantiated with several vector types that adhere to the vector object interface used. Different vector types may arise in the future, which then can be directly plugged into the matrix code.

After the decision was made to have member template functions `apply` and `applyTranspose` in the black box archetype, the idea of preferred input and output vector types was introduced. Black box matrices may have efficient implementations of their apply methods when working with a particular vector type. If this is true, then a user or an algorithm working with the black box matrix in question could choose this input and output vector type accordingly to speed up computation. Placing `typedef`ed members `PreferredInputVector` and `PreferredOutputVector` in the definition of a black box matrix gives users access to the preferred input and output types of the matrix. Figure 1 shows the black box archetype with preferred input and output vectors.

The composition class is based on the lazy evaluation scheme for black box matrix multiplication. As a consequence, the class will use an intermediate vector in its calculations. Since the black box matrices themselves are not templated by the vector type, there are several ways to choose the vector type of the intermediate vector. Suppose A is user defined blackbox matrix with preferred input vector type dense and preferred output vector type sparse. Assume P is a preconditioner that can be used as a left or right multiplier and let P have preferred input and output vector types sparse. When P is used as a left preconditioner, the composed matrix PA should use a sparse vector as the intermediate type, since the types are the same. If P is used as a right preconditioner and code efficiency is the highest priority, then the input type of A, a dense vector, should be used as the intermediate type of AP. However, if memory is limited and a top priority, then the output type of P, a sparse vector, would be the best choice for the intermediate vector. Finally, if space is not a problem and if the cost of copying one vector type into another is made up for by the gain in efficiency of the apply methods, then a conversion between types would be the best choice for handling the intermediate vectors.

```
template <class _Field> class Blackbox{
  public:
    typedef _Field Field;
    typedef Vector_In PreferredInputVector;
    typedef Vector_Out PreferredOutputVector;
    // Constructors and destructor
    ...
    template<class OutVector, class InVector>
    OutVector& apply(OutVector& y,
          const InVector&);
    template <class OutVector, class InVector>
    OutVector& applyTranspose(OutVector& y,
          const InVector& x);
    const Field& field();
    size_t rowdim();
    size_t coldim();
  private:
    // Internal storage and methods
    ... };
```

Figure 1: Black box Archetype with preferred vectors

The design of the new composition class will provide the user with four methods for selecting the type of the intermediate vector. Assume you are composing two black box matrices A, B into the black box matrix AB. The default method will compare the preferred input of A and the preferred output of B, and if they are the same use that type, else use a dense vector as the default type. In addition, the preferred input type of A or the preferred output type of B can be used as the intermediate vector type. Finally, the user can use both types and do a conversion between the two types during the computation in apply methods. A selection method that allows users to control how the intermediate vector type is chosen will be provided. The intermediate vector type will be chosen upon instantiation of the composition class and each composition object will use one instance of the intermediate vectors to be used for all applications of the composed matrix.

There are two design ideas not incorporated into the composition class described within this paper that warrant explanation. A user may wish to provide an intermediate vector type to be used regardless of the preferred input and output types. This is not allowed in the implementation and design of the composition class currently, but the option may be added later. Second, the apply methods (not the entire class) could be templated to choose the intermediate vector type. Allowing the apply methods themselves to choose the intermediate vector type is not possible under the *black box archetype* in LinBox. The composition class must follow the *black box archetype*, since it is itself a black box matrix, and so the apply methods can only be templated by the input and output vector type. Further, having the apply methods choose the intermediate vector type would mean the creation of many temporary vectors, which could slow down running times. Having the class choose the type and construct one intermediate vector that will be reused many times is more efficient and less prone to memory leaks.

3. IMPLEMENTATION OF THE COMPOSITION CLASS

The composition class will use partial template specialization to implement all the features listed in section 2. Since partially specialized template classes are instantiated instead of primary template classes when the template parameters match the partial definition, it becomes possible to program a conditional "if-then-else" during compile time expansion (template "meta" programming, see [14] and [9, 14.5.4.2]).

Partial template specialization is used several ways in the composition class. First it is the driving mechanism behind the selection method presented to a user. The user can select how the intermediate vector is chosen by passing a flag as a template parameter. The flag is an enumerated type defined in the composition header file with the declaration in figure 2.

```
enum IntermediateVector { DEFAULT,
  INPUT, OUTPUT, CONVERSION };
```

Figure 2: Enumerated type `IntermediateVector`

This flag defines all the choices of how the intermediate vector type can be selected. The composition class is passed the user's choice as a template parameter as you can see in the declaration of the class `Compose`, shown in figure 3.

```
template <class _Blackbox1,
class _Blackbox2 = _Blackbox1,
IntermediateVector flag = DEFAULT>
class Compose;
```

Figure 3: Declaration of `Compose`

The `Compose` class is specialized by the third template parameter, with each specialization making the appropriate choice for the type of the intermediate vector. Further the flag is defaulted to `DEFAULT` so a user does not need to make a selection.

The `DEFAULT` specialization requires that a comparison of the preferred input and output vector types be done and if they are the same, then we use that type, else we will use a dense vector for the intermediate type. This check and choice of types is done at compile time to avoid any unnecessary computations during run time. To perform this selection of types at compile time, a type choosing class is used. The `DEFAULT` specialization of `Compose` will instantiate a type choosing class, passing the preferred vector types as template parameters, and the template expansion mechanism will make the appropriate choice for the intermediate vector type. The code for the type choosing class `TypeChooser` is given in figure 4 and the instantiation of the class in `DEFAULT` is shown in figure 5.

The type choosing class in figure 4 compares the types `T` and `S`. If they are different then the top implementation of the class `TypeChooser` is instantiated by the compiler and the default type `D` is chosen. If the types `T` and `S` are the same then the second definition is instantiated and the type `T=S` is chosen.

The `CONVERSION` specialization compares the preferred intermediate vector types to specialize the apply methods. If the types are equal then no conversion needs to be done during an apply call, while a conversion between vector types must be performed if the types are not equal. As recommended by a referee, the comparison of types and choice of apply methods is made at compile time. The `CONVERSION` specialization contains a nested class that is templated by the preferred intermediate vector types and implements the apply methods. The encapsulated class has a partial specialization that eliminates the conversion of types when the two vector types are equal.

```
template <class T, class S, class D>
class TypeChooser {
  public:
    typedef D TYPE;
    TypeChooser() { } };
template <class T, class D>
class TypeChooser<T , T, D> {
  public:
    typedef T TYPE;
    TypeChooser() { } };
```

Figure 4: Type choosing class

```
typedef TypeChooser<
  typename Blackbox2::PreferredOutputType,
  typename Blackbox1::PreferredInputType,
  std::vector<Element> > VectorType;
```

Figure 5: Instantiation of `TypeChooser` in `DEFAULT`

4. EXTERNAL LIBRARIES AND MEMORY MANAGEMENT IN LINBOX

The memory management problems created when incorporating an external library into LinBox arise because LinBox uses C++ pointers and references as well as operators `new` and `delete` to address, manipulate, allocate, and deallocate memory. This fact ties the library to the details and representation of the physical memory and does not allow for the use of such abstractions as garbage collected memory or memory coming from a pre-allocated pool. It is important to introduce an interface that will provide a link between the external libraries and the LinBox objects that interact with them. The following paragraphs describe the LinBox objects that need such an interface.

LinBox uses several external libraries to perform field operations. The different implementations of the **field** arithmetic are contained in the objects that adhere to the *field archetype*, a common object interface defined by LinBox. It is these implementations that encapsulate the exact representation of the field elements and allow their manipulation by providing methods such as `add`, `mul`, etc. All these methods operate directly on the elements, and, therefore, need to be aware not only of how the elements are represented in the memory, but also how the memory is represented in the system.

Black box matrices may need to allocate field elements and then use field objects to operate on those elements. In such cases, a black box acts as a container and needs to be aware of the way in which elements are allocated and deallocated (the operations that replace operators `new` and `delete`, respectively) as well as how elements are addressed. **Vectors** in LinBox are usually containers of field elements, and in such case need to be provided with the same information as black boxes that contain elements: the ways in which elements can be allocated, deallocated, and addressed. Vectors may also be used to encapsulate the functionality of external vectors (for example, those native to the computer algebra systems that are using LinBox), and in such cases usually are accompanied by the field objects that describe how the elements stored in such external vectors should be manipulated. The last important part of the library that needs to be aware of how memory is represented and managed is the **algorithm** implementations.

A concrete example of the problems that may arise if an external library is to be used with LinBox is presented with SACLIB [2, 15]. SACLIB provides a number of facilities that can be of use to LinBox; from the perspective of memory management it is not important which specific facilities we employ, so we will concentrate on SACLIB's functions that perform field arithmetic over $\mathbb{Z}_p$ fields. What is important, however, is the fact that SACLIB uses a garbage collector to manage its memory, and applying SACLIB functions to the memory allocated using operator `new` or its underlying `malloc` will cause the program to crash since the elements will not be registered with the garbage collector. In fact, any memory that is used by SACLIB needs to be allocated by the library's own functions. For instance, to allocate a continuous array of elements, SACLIB provides a function `GCAMALLOC(n)` that takes the size of the array `n` as its argument. Since the memory is automatically garbage collected one may simply remove all the references to such an array in order for it to be returned into the pool of available memory.

5. STL ALLOCATORS

The problem of providing generic algorithms and generic abstractions for memory management is not unique to LinBox. It has been addressed and solved before by the C++ Standard Template Library [11]. STL employs the technique of allocators, objects "used to insulate implementers of algorithms and containers that must allocate memory from the details of physical memory" [14, page 567]. Allocators accomplish this function by providing a common interface that encapsulates the memory management functionality by providing standard names for types and functions that are involved in memory management such as pointers, references, functions to allocate, deallocate memory, and construct C++ objects in that memory. To understand better what functionality an allocator is supposed to provide, we examine the way in which the standard allocator may be declared. The standard allocator is provided by the STL in the header `<memory>` and is used by default by all the STL standard containers. The example in figure 6 is from [14, page 567]:

The functionality of each element of the allocator's design is implied by its name. For more information on the allocators and their use in the STL, see [11, Chapter 24] and [14, §19.4], here we will examine some of the important aspects of their design.

A notable element of the allocator design are the typedef declarations at the beginning of the `std::allocator` declaration. While in the implementation of the standard allocator the basic types (pointers and references) are defined to be called `pointer`, `reference`, and so on, one can easily imagine how, for example, a smart pointer class could be `typedef`ed to be called `pointer`. The fact that the allocators provide not only the functions that are used to allocate and deallocate memory, but also the data types used to represent the memory is very important: one can imagine a class that captures not only the information about the address of an object in the main memory, but also an address of the machine on which that object is stored, thus, providing a representation for distributed memory.

```
template <class T> class std::allocator
{ public:
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    pointer address(reference r) const { return &r; }
    const_pointer address(const_reference r) const
      { return &r; }
    allocator() throw();
    template <class U>
    allocator(const allocator<U>&) throw();
    ~allocator() throw();
    // space for n Ts
    pointer allocate(size_type n,
    allocator<void>::const_pointer hint = 0);
    // deallocate n Ts, don't destroy
    void deallocate(pointer p, size_type n);
    // initialize *p by val
    void construct(pointer p, const T& val)
      { new(p) T(val); }
    // destroy *p but don't deallocate
    void destroy(pointer p) { p->~T(); }
    size_type max_size() const throw();
    // in effect: typedef allocator<U> other
    template <class U> struct rebind
    { typedef allocator<U> other; } };
template<class T> bool operator==(
const allocator<T>&, const allocator<T>&) throw();
template<class T> bool operator!=(
const allocator<T>&, const allocator<T>&) throw();
```

Figure 6: STL Allocator

One has to note that the C++ Standard [9, 20.1.5 4] relaxes the requirements for the STL container implementations:

> Implementations of containers described in this International Standard are *permitted to assume* that their Allocator template parameter meets the following two additional requirements beyond those in Table 32:
>
> — All instances of a given allocator type are required to be interchangeable and always compare equal to each other.
>
> — The typedef members `pointer`, `const_pointer`, `size_type`, and `difference_type` are required to be `T*`, `T const*`, `size_t`, and `ptrdiff_t`, respectively.

However, at the same time it encourages the implementors of the libraries to not make such assumptions [9, 20.1.5 5]:

> Implementors are encouraged to supply libraries that can accept allocators that encapsulate more

general memory models and that support non-equal instances. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 32, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.

Unfortunately, common implementations of the STL used today, including the one supplied by the C++ compiler from the GNU Compiler Collection [7] (one of the key compilers targeted by LinBox), do make assumptions about the requirements on the typedef members of the allocators, which strictly limits the kinds of models that can be described.

One element of allocators' design that deserves closer attention is the member `struct rebind`. As the comment in the code in figure 6 states, `rebind` effectively `typedef`s its member `other` to be of type `allocator<U>`. This manipulation is provided so that storage for objects of a type other than the container element type can be managed. See [12, Chapter 4, p. 101]. For example, STL list nodes can thus be allocated via a user defined allocator.

6. ALLOCATORS IN LINBOX

6.1 Fields

As we have established earlier (in section 4) the part of the library whose addressing, allocation, deallocation, etc. requires special attention is the field elements: they most often occupy the most storage in any library operation, and they are the end target of black box manipulations. While all parts of the library need to be adjusted to be ready to accept different memory models, we should note that LinBox has already been designed in a way where all the details of the representation of field elements are hidden both from other parts of the library and the user by the field objects. In fact, field elements themselves do not even need to be classes (for example, in an implementation of $\mathbb{Z}_p$ field where p fits in a word, the elements themselves may be of type `unsigned int`), so it is only natural that we add another piece of information about elements — namely, how the memory in which they are stored is represented — to the field objects.

As a result, the field archetype and all the compliant field implementations get two new components:

- a typedef member `ElementAllocator`
- a method
  `ElementAllocator getElementAllocator() const`.

The first is the actual allocator type — the class that must adhere to the STL allocator requirements (see [9, 20.1.5]). The second member is the function that returns an instance of the allocator that the containers, algorithms and the user should use after possibly copying to a rebound allocator of appropriate type. This instance contains all the information necessary for memory management. A typical example of an allocator that needs additional information is a pool allocator.[‡]

We should also note that some fields may need to store field elements as part of the field's description. Such elements can no longer be just members of the field since fields may be allocated on the stack: rather, the references (or pointers) to such elements should be stored in the field itself, and the elements should be allocated using the field's allocator. See subsection 6.3 for the details of how the solution to the same problem with temporary elements used in the algorithm implementations should be implemented.

It is also interesting to note that the only modification necessary to the fields that have already been implemented to allow them to retain their current functionality is `typedef`ing `std::allocator<Element>` to be their `ElementAllocator` member, and defining `getElementAllocator()` member function to return `ElementAllocator()`. While such modifications are sufficient to retain their functionality, for many fields it is acceptable to operate on top of many different memory models[§], so many common field implementations may become template classes themselves, and allow a user to supply an allocator type which they would in turn pass to other library facilities as well as use to address the field elements. In such cases, LinBox shall follow the C++ Standard's encouragement ([9, 20.1.5 5] — cited in section 5), and use `ElementAllocator::reference` (`pointer`, etc.) in its fields' method declarations to accept more general memory models.

6.2 Vectors and black boxes

All the vectors used in LinBox internally adhere to the interfaces of various STL containers (most notably, `std::vector<Element>` for storing dense vectors of field elements, `std::vector< std::pair<size_t, Element> >` for storing sparse sequence vectors of elements, and `std::map< size_t, Element >` for storing sparse associative vectors). As it was mentioned above, STL defines its containers to be parametrized by an additional allocator type specifically for providing descriptions of alternative memory models. Since we have defined `Field::ElementAllocator` to adhere to the STL allocator object interface, and the declarations of containers are aware of which field objects are used (technically, they only need to be aware of the element type, but in reality such types are always obtained from the field object which is known in the context of the particular declaration), we simply require the library and the user to provide the allocator type to the vector declarations. As a result, the typical declarations of the vector objects are now of the form shown in figure 7. We also note that an allocator may contain auxil-

```
std::vector<Field::Element, Field::ElementAllocator>
// dense vectors,
std::vector<std::pair<size_t, Field::Element>,
        Field::ElementAllocator>// sparse sequence vectors
std::map<size_t, Field::Element, Field::ElementAllocator>
// sparse associative vectors
```

Figure 7: Vector declarations with allocators

[‡]A pool allocator is an excellent example of an allocator for different memory models because of its conceptual simplicity and usefulness: several POSIX facilities (such as shared memory and memory mapped files) describe the memory that they provide by supplying a pointer to the appropriate segment of memory, the size of which is known. With the pointer to the segment and its size one can construct a pool allocator that would abstract the memory to the library's facilities. A sample implementation of a pool_allocator can be found in Boost memory library [1].

[§]This is not the case, however, when the underlying library that implements field arithmetic is tied to some specific memory model. For an example, see section 7.2.

iary information that is necessary to describe the underlying memory model, so an instance of the allocator object has to be passed to a container. Internally, vectors are used primarily by various algorithms, so we consider an example of such use in subsection 6.3.

LinBox may also use "external vectors" — vectors that are native to and adhere to some internal representation of the computer algebra systems that are employing LinBox's functionality. In such cases, those vectors have to be adapted to conform to some interface that LinBox understands (e.g., the STL `vector` interface), so a wrapper class has to be provided. Then all the memory management information that is related to the representation of the vector depends on individual implementations and should be encapsulated inside that class.

While some of the black boxes provided by LinBox do not need to store field elements (for example, `Transpose` black box) the majority perform operations on field elements as part of the matrix times vector product. Such black boxes always have a field object passed to them, which now also contains information about how the memory used by the field elements needs to be managed. Black boxes in turn can simply pass allocators from such field objects to the underlying containers in the same way that is described above for vectors. If a black box uses temporary field elements in the implementation of its `apply` method, special attention needs to be paid to such elements, namely, to the fact that they cannot be allocated on the run-time stack as before. The same problem is present for algorithm implementations, so we discuss the details in the following section.

6.3 Algorithms

Just like any other part of the library, when a LinBox algorithm needs to manipulate field elements it is passed a field object that encapsulates information about the field including the representation of the field elements. When an algorithm needs to create a vector, it needs to allocate such a vector by passing the allocator type and object to it, so a typical declaration of a vector now may be of the form shown in figure 8. Here `f` is the field object (of the type `Field`) and `n` is the size of the vector that is being declared.

```
std::vector<Field::Element, Field::ElementAllocator>
v(n, Field::Element(), f.getElementAllocator());
```

Figure 8: Declaration of LinBox vectors

An issue that requires special attention are temporary elements that an algorithm may allocate. Such elements may no longer be allocated on the stack due to the fact that a memory model that the given field uses may not allow it: for example, SACLIB (discussed in section 7.2) implements specific routines to scan the stack for its `Words`; however, the authors of that library could make a requirement (for example, for the sake of system-independence) that the memory used by the library can only be allocated using the library's own functions, i.e., placing data on the stack would not be allowed. When an algorithm needs temporary elements for its implementation, it has to utilize the field's allocator for their allocation, construction, etc. While one may use the allocator's methods `allocate`, `construct`, etc. directly, it is advisable to employ the "resource acquisition is initialization" technique [14, page 366] to avoid potential mistakes. See figure 9 for an example. The same technique should be used in the implementations of black boxes, fields, and any other places that use temporary elements or where elements could be placed on the stack (for example, as members of field objects that could themselves be placed on the stack).

```
some_algorithm()
{ Field f;
  . . .
  std::vector<Field::Element, Field::ElementAllocator>
  tmp_vec(2, Field::Element(), f.getElementAllocator());
  Field::ElementAllocator::reference one = tmp_vec[0];
  Field::ElementAllocator::reference two = tmp_vec[1];
  . . .}
```

Figure 9: Creating temporary elements using allocators

7. SAMPLE CODE

This section provides some sample code that illustrates the design and implementation ideas described previously.

7.1 Composition

The following shows how to use the `Compose` class in LinBox. First, figure 10, shows a blackbox matrix that extends the current implementation of diagonal matrices, and provides preferred input and output vector types. Figure 11 is an example of a program that illustrates how to use the `Compose` class and all of the options for choosing the intermediate vector type.

```
template<class _Field, class _Input,
class _Output> class MyDiagonal{
  public:
    //Diagonal matrix data structures
    typedef _Input PreferredInputType;
    typedef _Output PreferredOutputType;
    // Implementation of required functions
    . . .};
```

Figure 10: Diagonal matrix implementing preferred vectors

The code in figure 11 has an example of each option for selecting the intermediate vector type. The black box matrix `AA` will use the preferred input type of `A`, which is a dense vector. The matrix `BC` will use the output type of `C` which is an STL list of pairs. The matrix `CB` uses the `DEFAULT` option, and since the input type of `C` and the output type of `B` are the same, then the intermediate vector will be a sparse sequence vector. Finally the matrix `BB` uses the `CONVERSION` option. Because the input and output types of `B` are different, then the composed matrix will have to do a conversion between the two types.

7.2 Allocators

Next is an illustration of allocators in LinBox based on the SACLIB library. "SACLIB is a library of C programs derived from the SAC2 system" [2, 15]. It includes facilities for list processing, integer, modular number, and rational number arithmetic, polynomial arithmetic, linear algebra,

```
int main(int argc, char **argv){
  //Declarations
  typedef MyDiagonal<Field, Vector<Field>::Dense,
  Vector<Field>::Dense > Blackbox1;
  typedef MyDiagonal<Field, Vector<Field>::Dense,
  Vector<Field>::SparseSeq > Blackbox2;
  typedef MyDiagonal<Field,
  Vector<Field>::SparseSeq,
  std::list<std::pair<size_t, Field::Element> > >
  Blackbox3;
  //Initialization
  Blackbox1 A(F, d1);
  Blackbox2 B(F, d2);
  Blackbox3 C(F, d3);
  //Compose the matrices and apply them to vectors
  Compose<Blackbox1, Blackbox1, INPUT>
  AA(&A, &A);
  Compose<Blackbox2, Blackbox3, OUTPUT>
  BC(&B, &C);
  Compose<Blackbox3, Blackbox2> CB(&C, &B);
  Compose<Blackbox2, Blackbox2, CONVERSION>
  BB(&B, &B);
  y1=AA.apply(y1, x1);
  y2=BC.apply(y2, x2);
  y3=CB.apply(y3, x3);
  y4=BB.apply(y4, x4);
  return 0; } //End main
```

Figure 11: Composition test program

computing polynomial GCD and resultants, polynomial factorization.

All of SACLIB's objects are presented to the user via their handles each of which occupies one word in memory (which SACLIB conveniently calls `Words`). While most of the SACLIB's functions manipulate lists, the library also provides a facility to allocate *garbage collected arrays*: arrays that can both be referred to by the SACLIB structures (and garbage collected when they become inaccessible) and contain handles to other SACLIB structures that will be taken care of by the garbage collector. The specific functions that are of interest are `GCAMALLOC(n)` that allocates a garbage collected array of size `n` and `GCA2PTR(A)` that returns a pointer to the actual elements of the array, thus, allowing the user to refer to the elements directly without using the supplementary accessor functions. One cannot place a reference to a SACLIB structure in the dynamically allocated memory since the garbage collector will not be aware of such references, will collect the structures that it will consider inaccessible, and further behavior of the program will be undefined.

In order to make SACLIB's facilities available to LinBox one has to define not only a field object (see `SacLibModularField` in figure 13) that uses SACLIB's functions to implement arithmetic over $\mathbb{Z}_p$, but also an allocator (`SacLibAllocator`) that communicates to the algorithms and containers how the memory needs to be allocated for SACLIB. It is interesting to note that `SacLibAllocator` has very few differences from the standard allocator described in section 5, so here we present only those key differences in figure 12. The main differences appear in methods `allocate` and `deallocate`. In method `allocate`, after computing the size of the garbage collected array that needs to be allocated, the allocator calls function `GCAMALLOC` to allocate the actual array, then it adds it to a globally registered list

```
Word allocated_GCA_list = NIL;
int num_of_allocators = 0;
template<class T>
class SacLibAllocator
{ public:
    . . .
    SacLibAllocator() throw()
    { if (num_of_allocators == 0)
      { GCGLOBAL(&allocated_GCA_list);
        // register allocated_GCA_list with
        // the garbage collector
      }
      ++num_of_allocators;
    }
    pointer allocate(size_type n,
    const void* hint = 0)
    {int size_to_alloc =
     n * sizeof(T) / sizeof(Word) +
     (n * sizeof(T) % sizeof(Word) == 0 ? 0 : 1);
     Word h = GCAMALLOC(size_to_alloc, GC_CHECK);
     allocated_GCA_list = COMP(h,
     allocated_GCA_list);
     return (pointer) GCA2PTR(h);
    }
    void deallocate(pointer p, size_type n)
    { std::pair<Word, Word> res =
      remove_from_list(p, allocated_GCA_list);
      allocated_GCA_list = res.second;
    }
    . . . };
```

Figure 12: SacLib Allocator

`allocated_GCA_list` to make sure the array will be accessible and will not be removed by the garbage collector, and returns the pointer to the actual elements stored in the array. Method `deallocate`, in turn, removes a previously allocated array from the `allocated_GCA_list`, the actual collection of the memory occupied by the array occurs during the next invocation of the garbage collector.

`SacLibModularField` provides the allocator and uses the appropriate SACLIB functions to implement various field operations, see figure 13.

SACLIB has to be initialized using its function `BEGINSACLIB` before it can be used in a program, and after SACLIB is uninitialized using `ENDSACLIB`, all of its structures will be unavailable, therefore, all the `SacLibModularFields` should be destroyed by the time `ENDSACLIB` is called. To achieve this goal one should again use "resource acquisition is initialization" technique by putting all of the operations that utilize SACLIB in an unnamed scope as in figure 14.

8. CONCLUDING REMARKS

First, we address the issue of code efficiency for our generic framework. All decisions in matrix composition are resolved at compile time and there is no loss of efficiency due to genericity. Similarly, if a standard STL `std::allocator` is used, the templates are compiled out, and the generated code incurs no additional run-time overhead. Our customization of an allocator to handle SACLIB objects can introduce an inefficiency: each time a SACLIB Word is allocated, it is prepended to a list that is registered with the SACLIB garbage collector, which can be both time and space inefficient. However, as an STL allocator, `SacLibAllocator` can

```
class SacLibModularField
{ public:
    typedef Word Element;
    typedef SacLibAllocator<Element> Allocator;
    typedef ElementAllocator::reference
            ElementReference;
    typedef ElementAllocator::const_reference
            ElementConstReference;
    typedef ElementAllocator::pointer ElementPointer;
    . . .
    SacLibModularField (const integer& m,
    const ElementAllocator& alloc = ElementAllocator())
    : _alloc(alloc) {
      _modulus = _alloc.allocate(1);
      . . . }
    ElementReference add (ElementReference x,
    ElementConstReference y,
    ElementConstReference z) const
       { return x = MISUM(*_modulus, y, z); }
    . . .
    ElementAllocator getElementAllocator() const
    { return _alloc; }
  private:
    ElementAllocator _alloc;
    ElementPointer _modulus;
    // cannot be just Element because the field could
    // be allocated in the dynamic memory, see section 6.1
    . . . };
```

Figure 13: SacLib field implementation

```
#include <linbox/fields/saclib−modular−field.h>
. . .
int main(int argc, char* argv[])
{ BEGINSACLIB(&argc);
  {  // "SACLIB safety scope"
    integer m;
    . . .// initialize m to some large prime number
    SacLibModularField F(m);
    // perform field operations, create black boxes,
    //invoke algorithms, etc
    . . .
  } // end "safety scope"
  ENDSACLIB(SAC_FREEMEM); }//End main
```

Figure 14: SacLib test program

also allocate an array of SACLIB Words by a single call, thus allowing the application program to "pool" native memory chunks. One can also provide automatic memory blocking via the allocator mechanism, but we have not done so.

In its current state, the LinBox library contains numerous algorithms for sparse, structured and black box matrices. We have described a framework that permits black box matrix multiplication, which can be employed, for example, in the pre-conditioners needed in some of the algorithms [3]. We have also given a means to incorporate external memory managers, which allows the use of external garbage collected libraries in LinBox and which can implement a memory model where allocation is distributed over several computers.

Acknowledgments: We thank members of the LinBox project for their helpful comments on our design. We would also like to thank Matt Austern, Gillmer Derge, and David Musser for the useful information and pointers to the design intricacies of C++ that they have provided. We thank the two anonymous referees for their informative and constructive suggestions.

9. REFERENCES

[1] Boost C++ libraries, 2003. URL: `http://www.boost.org`.

[2] Brown, C. Saclib2.1 on Linux. `http://www.cis.udel.edu/~saclib/linux/SaclibLinux.html`, Mar. 2000.

[3] Chen, L., Eberly, W., Kaltofen, E., Saunders, B. D., Turner, W. J., and Villard, G. Efficient matrix preconditioners for black box linear algebra. *Linear Algebra and Applications 343–344* (2002), 119–146.

[4] Cohen, A. M., Gao, X.-S., and Takayama, N., Eds. *Proc. First Internat. Congress Math. Software ICMS 2002, Beijing, China* (Singapore, 2002), World Scientific.

[5] Dumas, J.-G., Gautier, T., Giesbrecht, M., Giorgi, P., Hovinen, B., Kaltofen, E., Saunders, B. D., Turner, W. J., and Villard, G. LinBox: A generic library for exact linear algebra. In Cohen et al. [4], pp. 40–50.

[6] Dumas, J.-G., Giorgi, P., and Pernet, C. FFPACK: Finite field linear algebra package. In Gutierrez [8], pp. 119–126.

[7] GNU compiler collection, 2003. FSF - Free Software Foundation Gnu Project: URL: `http://gcc.gnu.org`.

[8] Gutierrez, J., Ed. *ISSAC 2004 Proc. 2004 Internat. Symp. Symbolic Algebraic Comput.* (New York, N. Y., 2004), ACM Press.

[9] ISO/IEC. *International standard, programming languages—C++*. No. 14882:1998(E). American National Standards Institute, New York, 1998.

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

[11] Musser, D. R., Derge, G. J., and Saini, A. *STL Reference Guide*, second ed. Addison-Wesley, Reading, MA, 2001.

[12] Plauger, P. J., Stepanov, A. A., Lee, M., and Musser, D. R. *The C++ Standard Template Library.* Prentice Hall PTR, Upper Saddle River, New Jersey, 2001.

[13] Saunders, D., and Wan, Z. Smith normal form of dense integer matrices, fast algorithms into practice. In Gutierrez [8], pp. 274–281.

[14] Stroustrup, B. *The C++ Programming Language*, third ed. Addison–Wesley, Reading, Massachusetts, 1997.

[15] Vielhaber, H., Buchberger, B., Collins, G. E., Encarnación, M. J., Hong, H., Johnson, J. R., Krandick, W., Loos, R., and Mandache, A. N. A. M. SACLIB 1.1 user's guide. Tech. Rep. 93-19, RISC Linz, Linz, Austria, 1993.

[16] Watt, S. M. A study in the integration of computer algebra systems: Memory management in the Maple-Aldor environment. In Cohen et al. [4], pp. 405–410.