SZAKDOLGOZAT FELADAT

Marussy Kristóf
mérnök informatikus hallgató részére

Konfigurálható numerikus módszerek sztochasztikus modellekhez

A kritikus rendszerek – biztonságkritikus, elosztott és felhő-alapú alkalmazások – helyességének biztosításához szükséges a funkcionális és nemfunkcionális követelmények matematikai igényességű ellenőrzése. Számos, szolgáltatásbiztonsággal és teljesítményvizsgálattal kapcsolatos tipikus kérdés általában sztochasztikus analízis segítségével válaszolható meg.

A kritikus rendszerek elosztott és aszinkron tulajdonságai az állapottér robbanás jelenségéhez vezetnek. Emiatt méretük és komplexitásuk gyakran megakadályozza a sikeres sztochasztikus analízist, melynek számításigénye nagyban függ a lehetséges viselkedések számától. A modellek komponenseinek jellegzetes időbeli viselkedése és leginkább eltérő karakterisztikája a számításigény további jelentős növekedését okozhatja.

A szolgáltatásbiztonsági és teljesítményjellemzők kiszámítása markovi modellek állandósult állapotbeli és tranziens megoldását igényli. Számos eljárás ismert ezen problémák kezelésére, melyek eltérő reprezentációkat és numerikus algoritmusokat alkalmaznak; ám a modellek változatos tulajdonságai miatt nem választható ki olyan eljárás, mely minden esetben hatékony lenne. A hallgató feladata áttekinteni az irodalmat és megvizsgálni az ismert algoritmusokat.

A feladat megoldása a következő lépésekből áll:

1. Mutassa be az irodalomban ismert, markovi sztochasztikus rendszerek állandósult állapotbeli és tranziens viselkedésének vizsgálatára alkalmas numerikus algoritmusokat.
2. Az irodalom alapján implementáljon kiválasztott tranziens és állandósult állapotbeli analízis algoritmusokat.
3. Hasonlítsa össze futási idő és tárhely komplexitás szempontjából az implementált algoritmusokat.
4. Értékelje a megoldást és vizsgálja meg a továbbfejlesztési lehetőségeket.

Tanszéki konzulens: Vörös András, tudományos segédmunkatárs
Molnár Vince, doktorandusz
Külső konzulens: dr. Telek Miklós, egyetemi tanár

Budapest, 2015. október 7.

Dr. Jobbágy Ákos
egyetemi tanár
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

Contents

Chapter 4

Configurable data structure and operations

In this chapter, we present the linear algebra library that was developed as a foundation for configurable stochastic analysis.

The library is composed of a data structure and its related operations. The *data structure* provides abstraction for the numerical solution algorithms over the used matrix and vectors storage formats. Matrices stored as dense or sparse arrays, and even complex expression involving sums and Kronecker products that arise from matrix decompositions can be handled in a general way.

While direct read write access to elements is supported for most matrices and vectors, the majority of manipulations, such as vector–matrix products or vector additions, structure are performed as *operations*. Instead of being impelemented as methods of the data structure classes, operations are decoupled into separata entities. This allows operation execution with multiple dispatch, selecting optimized implementations accoringd to dynamic types of all operation arguments and other runtime properties, e.g. the numer of elements in the vector.

Another advantage of the decoupled operations framework is runtime configurability. The dispatch logic may be replaced between the execution of algorithms in the stochastic analysis workflow, therefore low level linear algebra operations may customized to suit the algorithm and the matrix decomposition in use, as well as the hardware. For example, paralell and sequential execution may be switched as neccessary.

Existing linear algebra and matrix libraries, such as [2, 5, 7, 9, 10], usually have unsatisfactory support for operations required in stochastic analysis algorithms with decomposed matrices, such as multiplications with Kronecker and block Kronecker matrices. Therefore, we have decided to develop out linear algebra framework from scratch in C#.NET specifically for stochastic algorithms as a basis of our stochastic analysis framework.

4.1 Data structure

The data structure library contains matrix and vector classes for stochastic analysis.

Client code interacts with the data strucutre through interfaces only, no classes are exposed on the public API. The main interfaces are `IVector` and `IMatrix` for vectors and matrices, respectively. The instances are created through an exposed static factory.

The interfaces are generic in the element type. For example `IVector<double>` and `IMatrix<double>` are used to work with double-precision floating point arithmetic. Due to language limitations, some classes must be implemented without genericity. In these cases, only `double` is currently supported, although re-implementation for single-precision floating point or other numeric types is trivial. The static factory handles selection of the appropriate non-generic type to instantiate if generic behavior is impossible.

There also exist *block* versions of these interfaces, IBlockVector and IBlockMatrix. A block object is conceptually a container of objects with scalar elements. For example, if $\mathbf{v} \in \mathbb{R}^{n_0+n_1+\cdots+n_{k-1}}$ is a block vectors with k blocks, $\mathbf{v}[i] \in \mathbb{R}^{n_i}$ $(0 \leq i < k)$ is a vector of real numbers with n_i elements, while $v[i][j]$ is the jth element of the ith block of $\mathbf{v}$. Hovewer, block interfaces do not extend from `IVector<IVector<T>>` and `IMatrix<IMatrix<T>>`, but a facility separate from ordinary indexing is provided for block access. This allows passing `IBlockVector<double>` and `IBlockMatrix<double>` objects to procedures consuming ordinary `IVector<double>` and `IMatrix<double>`.

4.1.1 Partials

Manipulations of subsequences of vector and mactrix elements, as well as conversion between flat and block object is performed by partial object wrappers.

Definition 4.1 A *partial vector* $\mathbf{v}[s{:}t{:}m]$ of a vector $\mathbf{v} \in \mathbb{R}^n$ is

$$\mathbf{v}[s{:}t{:}m] \in \mathbb{R}^m : v[s{:}t{:}m][i] = v[s + t \cdot i] \text{ for } i = 0, 1, \ldots, m,$$

where $0 \leq s \leq n, 1 \leq t, s + t(m-1) \leq n$.

The index s is the start of partial, t is the stride and m is the partial length. Matrix partials $A[s_1{:}t_1{:}m_1, s_2{:}t_2{:}m_2] \in \mathbb{R}^{m_1 \times m_2}$ are defined analogously.

The method `GetPartial` forms partial matrices and vectors. The returned object is always a wrapper which passes through and read and write indices to the original object after index manipulation. However, partial manipulation of large vectors, which was found to be a performance bottleneck upon profiling, is implemented with pointer arithmetic instead.

The `GetPartial` method itself is also passed through. This means forming a partial of partial ($\mathbf{v}[s_1{:}t_1{:}m_1][s_2{:}t_2{:}m_2]$) does not result in a chain of wrappers being created, but only a single wrapper object is places around the original after the neccassary index manipulations.

Block vectors and matrices may be be formed by splitting flat objects into blocks with partials, or by composition from unrelated objects.

4.1.2 Splitting and composition

Definition 4.2 A *split* of a vector $\mathbf{v} \in \mathbb{R}^n$ at $(n_0, n_1, \ldots, n_{k-1})$ is a block vector

$$\mathbf{v}_S \in \mathbb{R}^{n_0+n_1+\cdots n_{k-1}} : \mathbf{v}_S[i] = \mathbf{v}[N_i{:}1{:}n_i], \quad N_i = \sum_{j=0}^{i-1} n_i, \tag{4.1}$$

where $N_0 = 0$ and $n = N_{k+1} = n_0 + n_1 + \cdots n_{k-1}$.

Split matrices are defined analogously.

Definition 4.3 If $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{k-1}$ are real vectors of length $n_0, n_1, \ldots, n_{k-1}$, respectively, their *composition* $\mathbf{v}_C \in \mathbb{R}^{n_0+n_1+\cdots+n_{k-1}}$ is block vector $\mathbf{v}_C[i] = \mathbf{v}_i$.

Definition 4.4 If $A_{0,0}, A_{0,1}, \ldots, A_{0,l-1}, A_{1,0}, \ldots Ak-1, l-1$ are matrices such that $A_{i,j} \in \mathbb{R}^{n_i, m_j}$, their *composition* $A_C \in \mathbb{R}^{(n_0+n_1+\cdots+n_{k-1})\times(m_0+m_1+\cdots m_{l-1})}$ is block matrix $A_C[i,j] = A_{i,j}$.

The `Split` method builds block vectors and matrices from flat objects for blockwise access. Objects formed by `Split` can be split again arbitrarily, where the split command is forwarded to the original flat object.

In contrast, `Split` can only be applied to a composite object if it does not result in the creation of new partials. That is, a composite vector $\mathbf{v} \in \mathbb{R}^{n_0+n_1+\cdots+n_{k-1}}$ may only be split at $(m_0, m_1, \ldots, m_{l-1})$ if $k = l$ and $n_i = m_i$ for all $0, 1, \ldots, k-1$. Because composite objects are usually very large and are used in performance critical parts of algorithms, we decided to throw an exception instead of splitting even though arbitrary splitting of composite vectors and matrices would have been implemented easily.

4.1.3 Vectors

Vector data structures are used to store probability distributions of Markovian modells, as well as intermediate results of numerical algorithms.

The class hiearchy of vectors in our library is shown in Figure 4.1.

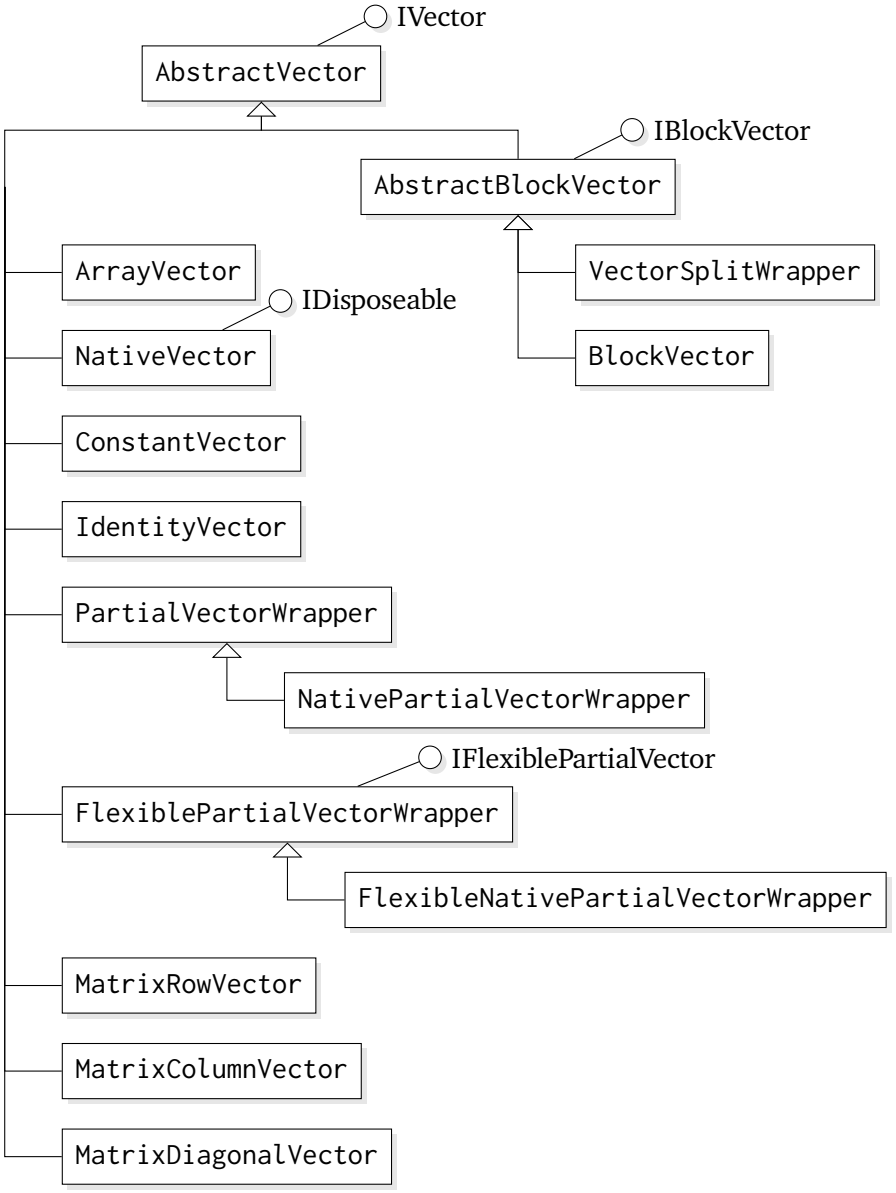

Figure 4.1 Inheritance hierarchy of vectors.

The abstract base classes `AbstractVector`, `AbstractBlockVector` are at the root of the inheritance hierarchies. The data structure may be extended by inheriting from these classes, or by implementing the publicly exposed interfaces directly.

Vector datatypes are available for the storage of general vectors, as well as for some special cases.

ArrayVector The basic vector datatype provided is `ArrayVector`, which stores vector elements in a Common Language Runtime (CLR) array. This class is completely generic, i.e. any CLR value or reference type may be used as an element.

The Microsoft .NET implementation of the CLR allows arrays of size up to 2 GiB even on 60-bit platforms. While on .NET 4.5, this limitation may be lifted with the `gcAllowVeryLargeObjects` configuration directive[1], this setting is cumbersome to use. Therefore, no vectors larger than 2 GiB should be stored as array vectors.

NativeVector To work around the 2 GiB memory limitation on CLR arrays, we implemented `NativeVector` which stores vector elements on the unmanaged heap. We also found unmanaged allocation reduce the pressure on the garbage collector, therefore provide the benefit of faster allocations.

Native vectors utilize the `unsafe` facilities[2] provided by the C# language, including the access to memory through pointers and direct memory management through `AllocHGlobal` and `FreeHGlobal`. Therefore, the linear algebra library must be compiled with unsafe language features enabled. As an alternative, `NativeVector` may be disabled with conditional compilation directive and replaced by a wrapper around `ArrayVector`, forgoing the benefits of unmanaged allocation.

Due to language limitations, `NativeVector` must be implemented for any primitive type desired to be used as vector elements. Currently, only `double` is supported.

The use of unmanaged memory requires manual deallocation to avoid memory leaks. Because the C# language does not provide deterministic destructors, the `IDisposeable` pattern[3] must be used.

As an alternative means of memory management, an interface `IBufferProvider` may be used to allocate and track multiple vectors. A `IBufferProvider` itself implements `IDisposeable`, a single C# `using` block may free several vectors in the same scope, easing the burden of manual disposal. This approach is illustrated in Listing 4.1.

ConstantVector A constant vector is a vector with equal elements.

[1] `https://msdn.microsoft.com/en-us/library/hh285054(v=vs.110).aspx`
[2] `https://msdn.microsoft.com/en-us/library/chfa2zb8.aspx`
[3] `https://msdn.microsoft.com/en-us/library/system.idisposable(v=vs.110).aspx`

Listing 4.1 Manual memory management for `NativeVector`.

```
// Create and dispose a NativeVector of length 100.
using (var vector = Vectors.NewDisposeableVector<double>(100))
{
    vector[0] = 1.0;
}

// Dispose using IBufferProvider.
var factory = new DisposingBufferProviderFactory();
using (var bufferProvider = factory.Make())
{
    var v1 = bufferProvider.GetVector<double>(100);
    var v2 = bufferProvider.GetVector<double>(100);
}
// Both v1 and v2 are disposed here.
```

Two important special vector may be realized as `ConstantVector` instances in stochastic analysis, the vector of all zeroes $\mathbf{0}$, and the vector of all ones $\mathbf{1}$,

$$\texttt{Vectors.Constant<double>}(n,\ 0) = \mathbf{0} \in \mathbb{R}^n,$$
$$\texttt{Vectors.Constant<double>}(n,\ 1) = \mathbf{1} \in \mathbb{R}^n.$$

Because constant vectors require only $O(1)$ storage space instead of $O(n)$, this is an important optimization in equations involving $\mathbf{0}$, $\mathbf{1}$ and its scalar multiples.

IdentityVector An identity vector is vector with all but one zero elements and a single 1 element. Formally,

$$\texttt{Vectors.Identity<double>}(n,\ i) = \mathbf{e}_i \in \mathbb{R}^n, \quad e_i[j] = \delta_{i,j} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

`IdentityVector` is an $O(1)$ space optimization for storing special vectors, similar to `ConstantVector`.

PartialVectorWrapper and FlexiblePartialVectorWrapper Taking a partial vector of a vector results in the creation of `PartialVectorWrapper` object, which passes through read and write actions to the underlying vector after the neccessary index manipulations. Hence a new object is allocated at every call to `GetPartial`.

Listing 4.2 Ambigous use of `FlexiblePartialVectorWrapper`.

```
1  var vector = Vectors.NewArray<double>(100);

3  var part = vector.GetPartial(5, 2, 20);
4  var subPart = part.GetPartial(2, 1, 5);
5  // Unambigous, subPart = vector[5:2:20][2:1:5].

7  var flexible = vector.GetFlexiblepartial(5, 2, 20);
8  var subFlexible = vector.GetPartial(2, 1, 5);
9  flexible.SetPartial(6, 2, 20);
10 // Ambigous, is subFlexible = vector[5:2:20][2:1:5] or vector[6:2:20][2:1:5]?
```

Long delegation chains are eliminated by *collapsing* partial vectors. When the method `GetPartial` is invoked on `PartialVectorWrapper`, it performs index manipulations and passes delegates to the `GetPartial` method of the original vector. Thus, further partials do not have reference to the partial vector they were created from, but only to the underlying vector.

`FlexiblePartialVectorWrapper` alleviate allocation costs by providing a partial vector whose indices can be changed after construction. For example, if a flexible partial vector $\mathbf{v}[s{:}t{:}m]$ is available, it can be changed to $\mathbf{v}[s'{:}t'{:}m']$ without allocating a new instance whenever the need arises. The functionality is exposed to consumers through an interface.

Flexible partials cannot have their partials taken, because the collapse of the delegation chain makes the propagation of index changes impossible. This problem is illustrated in Listing 4.2.

Another set of partial wrappers handle partials of `NativeVector` instances. In these cases, a (*base pointer*, *stride*, *length*) triple amy be queried for use in low-level operations. Hence indexing logic may be skipped in favor of direct access.

To create a unified interface, the same triple may be queried from `NativeVector` instances themselves, where *base pointer* is the pointer to the allocated buffer, *stride* $= 1$ and *length* is the length of the vector itself.

Matrix vector wrappers To facilitate common manipulations of matrices, our library provides wrappers to for read and write access of parts of matrices as vectors.

`MatrixRowVector` and `MatrixColumnVector` accesses a row or a column of matrix, respectively. If $A \in \mathbb{R}^{n \times m}$,

$$A.\texttt{GetRow}(i) = \mathbf{r} \in \mathbb{R}^m, \quad r[j] = a[i, j],$$

$$A.\texttt{GetColumn}(j) = \mathbf{c} \in \mathbb{R}^n, \quad c[i] = a[i,j]$$

for $0 \leq i < n$, $0 \leq j < m$.

If $n = m$, i.e. A is square, `MatrixDiagonalVector` may provide access to the diagonal of the matrix,

$$A.\texttt{GetDiagonal}() = \mathbf{d} \in \mathbb{R}^n = \mathbb{R}^m, \quad d[i] = a[i,j].$$

Block vectors `VectorSplitWrapper` reifies vector splitting accoring to eq. (4.1) on page 25. The split vector is backed by a composition of partial vectors, thus it acts as a composite vector. However, when the split wrapper is used as an instance of `IVector`, commands, including re-splitting, are delegated to the underlying vector instead.

Composition of vectors according to Definition 4.3 on page 25 is represented by `BlockVector`. The constructor of `BlockVector` is passed a sequence of vectors which will constitute the block vector. Because `BlockVector` implements `IVector`, the composite vector may be used as a normal vector, however, splitting is limited to avoid performance penalties associated

4.1.4 Matrices

The class hiearchy of vectors in our library is shown in Figure 4.2. The abstract base classes `AbstractVector`, `AbstractBlockVector` are at the root of the inheritance hierarchies.

ArrayMatrix

SparseMatrix and NativeSparseMatrix Sparse matrices are stored in Compressed Column Storage (CCS) format, i.e. an array or values and row indices are stored for each column of the matrix, as illustrated in Figure 4.3. This facilitates multiplication from left with row vectors.

While other sparse matrix formats, such as sliced LAPACK are more amenable to parallel and SIMD processing Kreutzer et al. [8], CCS was selected due to implementation simplicity and the small number of nonzero entries in each column of the matrix, which reduces the potential benefits of SIMD implementations.

DiagonalMatrix

NullMatrix and IdentityMatrix

PartialMatrixWrapper

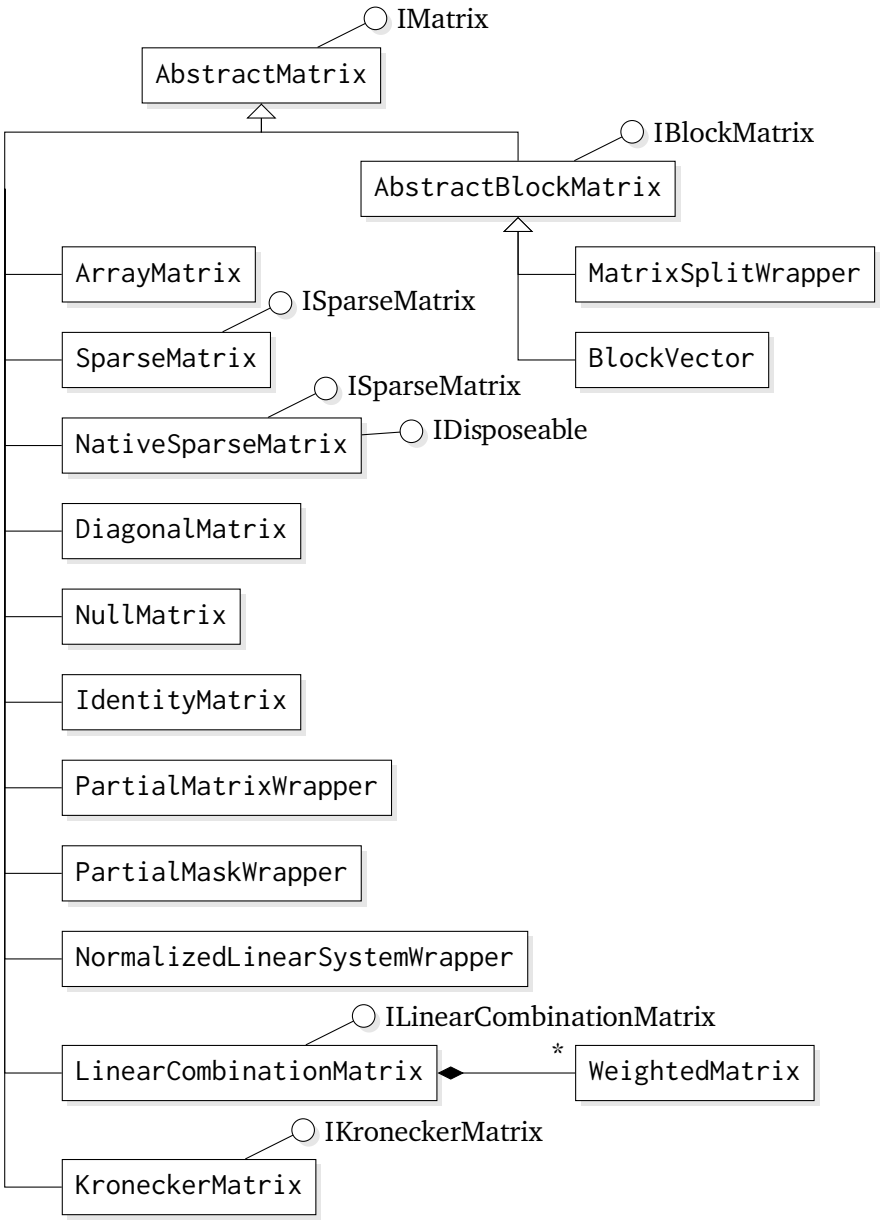

Figure 4.2 Inheritance hierarchy of matrices.

$$A = \begin{pmatrix} 1 & 0 & 0 & 2.5 \\ 3 & 1 & 0 & 0 \\ 4 & 0 & 0 & 1 \\ 5 & 0 & 0 & 0 \end{pmatrix} \qquad \begin{aligned} A = \{&\{(1,0),(3,1),(4,2),(5,3)\},\\ &\{(1,1)\},\\ &\{\},\\ &\{(2.5,0),(1,2)\}\} \end{aligned}$$

Figure 4.3 Compressed Column Storage of a matrix.

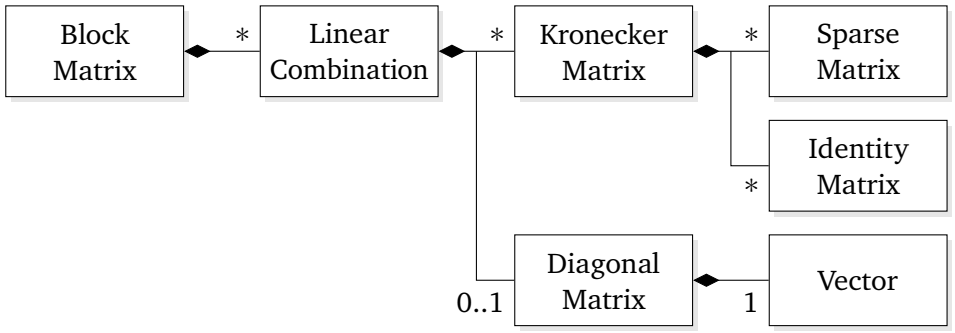

Figure 4.4 Data structure for block Kronecker matrices.

PartialMaskWrapper

NormalizedLinearSystemWrapper

LinearCombinationMatrix

KroneckerMatrix

4.2 Expression trees

Decomposed Kronecker and block Kronecker matrices are stored as algebraic expression trees as shown in Figure 4.4. Expression may contain Kronecker products (`KroneckerMatrix`), linear combinations (`LinearCombinationMatrix`) and block compositions (`BlockMatrix`).

The expression tree approach allows the use of arbitrary matrix decompositions that can be expressed with block matrices, linear combinations and Kronecker products. The implementation of additional opeartional primitives is also straightforward. The data structure forms a flexible basis for the development of stochastic analysis algorithms with decomposed matrix representations.

Algorithm 4.1 Parallel block vector-matrix product.

Input: block vector $\mathbf{b} \in \mathbb{R}^{n_0+n_1+\cdots+n_{k-1}}$,
block matrix $A \in \mathbb{R}^{(n_0+n_1+\cdots+n_{k-1})\times(m_0+m_1+\cdots+m_{l-1})}$
Output: $\mathbf{c} = \mathbf{b}A \in \mathbb{R}^{m_0+m_1+\cdots+m_{l-1}}$
1 **allocate** $\mathbf{c} \in \mathbb{R}^{m_0+m_1+\cdots+m_{l-1}}$
2 **parallel for** $j \leftarrow 0$ **to** $l-1$ **do**
3 $\mathbf{c}[j] \leftarrow \mathbf{0}$
4 **for** $i \leftarrow 0$ **to** $k-1$ **do**
5 $\mathbf{c}[j] \leftarrow \mathbf{c}[j] + \mathbf{b}[i]A[i,j]$ // Scaled addition of vector-matrix product

Algorithm 4.2 Product of a vector with a linear combination matrix.

Input: $\mathbf{b} \in \mathbb{R}^n$, $A = \nu_0 A_0 + \nu_1 A_1 + \cdots + \nu_{k-1} A_{k-1}$, where $A_h \in \mathbb{R}^{n\times m}$
Output: $\mathbf{c} = \mathbf{b}A \in \mathbb{R}^m$
1 **allocate** $\mathbf{c} \in \mathbb{R}^m$ if no target buffer is provided
2 $\mathbf{c} \leftarrow \mathbf{0}$
3 **for** $h \leftarrow 0$ **to** $k-1$ **do**
4 $\mathbf{c} \leftarrow \nu_h \cdot \mathbf{b}A_h$ // In-place scaled addition of vector-matrix product
5 **return c**

4.2.1 Efficient vector-matrix products

Iterative linear equation and transient distribution solvers require several vector-matrix products per iteration. Therefore, efficient vector-matrix multiplication algorithms are required for the various matrix storage methods (i.e. dense, sparse and block Kronecker matrices) to support configurable stochastic analysis.

Our data structure supports run-time reconfiguration of operations, for example, to switch between parallel and sequential matrix multiplication implementations for different parts of an algorithm, depending on the characteristics of the model and the hardware which runs the analysis.

Implemented matrix multiplication for the data structure (see Figure 4.4 on page 32) routines are

- Multiplication of vectors with dense and sparse matrices. Sparse matrix multiplication may be parallelized by splitting the columns of the matrix into chunck and submitting each chunk to the executor thread pool.

  Operations with vectors and sparse matrices are implemented in an `unsafe`[4]

[4] `https://msdn.microsoft.com/en-us/library/chfa2zb8.aspx`

Algorithm 4.3 The SHUFFLE algorithm for vector-matrix multiplication.

Input: $\mathbf{b} \in \mathbb{R}^{n_0 n_1 \cdots n_{k-1}}$, $A = A^{(0)} \otimes A^{(1)} \otimes \cdots \otimes A^{(k-1)}$, where $A^{(h)} \in \mathbb{R}^{n_h \times m_h}$
Output: $\mathbf{c} = \mathbf{b}A \in \mathbb{R}^{m_0 m_1 \cdots m_{k-1}}$

$n \leftarrow n_0 n_1 \cdots n_{k-1}, \quad m \leftarrow m_0 m_1 \cdots m_{k-1}$
tempLength $\leftarrow \max_{h=-1,0,1,\ldots,k-1} \prod_{f=0}^{h} m_f \prod_{f=h+1}^{k-1} n_f$
allocate $\mathbf{x}, \mathbf{x}'$ with at least *tempLength* elements
$\mathbf{x}[0{:}1{:}n] \leftarrow \mathbf{b}, \quad i_{\text{left}} \leftarrow 1, \quad i_{\text{right}} \leftarrow \prod_{h=1}^{k-1} n_h$
for $h \leftarrow 0$ **to** $k-1$ **do**
    if $A^{(h)}$ is not an identity matrix **then**
        $i_{\text{base}} \leftarrow 0, j_{\text{base}} \leftarrow 0$
        for $il \leftarrow 0$ **to** $i_{\text{left}} - 1$ **do**
            for $ir \leftarrow 0$ **to** $i_{\text{right}} - 1$ **do**
                $\mathbf{x}'[j_{\text{base}}{:}m_h{:}i_{\text{right}}] \leftarrow \mathbf{x}[i_{\text{base}}{:}n_h{:}i_{\text{right}}]A^{(h)}$
                $i_{\text{base}} \leftarrow i_{\text{base}} + n_h i_{\text{right}}, \quad j_{\text{base}} \leftarrow j_{\text{base}} + m_h i_{\text{right}}$
        Swap the references to $\mathbf{x}$ and $\mathbf{x}'$
    $i_{\text{left}} \leftarrow i_{\text{left}} \cdot m_h$
    if $h \neq k-1$ **then** $i_{\text{right}} \leftarrow i_{\text{right}} / n_{h+1}$
return $\mathbf{c} = \mathbf{x}[0{:}1{:}m]$

context. The elements of the data structures are not under the influence of the Garbage Collector runtime, but stored in natively allocated memory. This allows the handling of large matrices without adversely impacting the performance of other parts of the program, albeit the cost of allocations in increased.

- Multiplication with block matrices by delegation to the constituent blocks of the matrix (Algorithm 4.1 on page 33). The input and output vectors are converted to block vectors before multiplication. If parallel execution is required, each block of the output vector can be computed in a different task, since it is independent from the others.

- Multiplication by a linear combination of matrices is delegated to the constituent matrices (Algorithm 4.2 on page 33). An in-place scaled addition of vector-matrix product to a vector operation is required for this delegation. To facilitate this, each vector-matrix multiplication algorithm is implemented also as an in-place addition and in-place scaled addition of vector-matrix product, and the appropriate implementation is selected based on the function call aruments.

- Multiplications $\mathbf{b} \cdot \text{diag}\{\mathbf{a}\}$ by diagonal matrices are executed as elementwise

product $\mathbf{b} \odot \mathbf{a}$. The special case of multiplication by an identity matrix is equivalent to a vector copy.

- Multiplications by Kronecker products is performed by the SHUFFLE algorithm [1, 3] as shown in Algorithm 4.3 on page 34.

  The algorithm requires access to slices of a vector, denoted as $\mathbf{x}[i_0{:}s{:}l]$, which refers to the elements $x[i], x[i{+}s], x[i{+}2s], \ldots, x[i{+}(l{-}1)s]$. Thus, slices were integrated into the operations framework as first-class elements, and multiplication algorithms are implemented with support for vector slice indexing.

  SHUFFLE rewrites the Kronecker products as

$$\bigotimes_{h=0}^{k-1} A^{(h)} = \prod_{h=0}^{k-1} I_{\prod_{f=0}^{h-1} n_f \times \prod_{f=0}^{h-1} n_f} \otimes A^{(h)} \otimes I_{\prod_{f=h+1}^{k-1} m_f \times \prod_{f=h+1}^{k-1} m_f},$$

  where $I_{a \times a}$ denotes an $a \times a$ identity matrix. Multiplications by terms of the form $I_{N \times N} \otimes A^{(h)} \otimes I_{M \times M}$ are carried out in the loop at line 8 of Algorithm 4.3.

  The temporary vectors $\mathbf{x}, \mathbf{x}'$ are large enough store the results of the successive matrix multiplications. They are cached for every worker thread to avoid repeated allocations.

  Other algorithms for vector-Kronecker product multiplication are the SLICE [6] and SPLIT [4] algorithms, which are more amenable to parallel execution than SHUFFLE. Their implementation is in the scope of our future work.

References

[1] Anne Benoit, Brigitte Plateau, and William J Stewart. “Memory efficient iterative methods for stochastic automata networks”. In: (2001).

[2] BlueBit Software. *.NET Matrix Library 6.1*. Accessed October 26, 2015. URL: http://www.bluebit.gr/NET/.

[3] Peter Buchholz, Gianfranco Ciardo, Susanna Donatelli, and Peter Kemper. “Complexity of Memory-Efficient Kronecker Operations with Applications to the Solution of Markov Models”. In: *INFORMS Journal on Computing* 12.3 (2000), pp. 203–222. DOI: 10.1287/ijoc.12.3.203.12634.

[4] Ricardo M. Czekster, César A. F. De Rose, Paulo Henrique Lemelle Fernandes, Antonio M. de Lima, and Thais Webber. “Kronecker descriptor partitioning for parallel algorithms”. In: *Proceedings of the 2010 Spring Simulation Multiconference, SpringSim 2010, Orlando, Florida, USA, April 11-15, 2010*. SCS/ACM, 2010, p. 242. ISBN: 978-1-4503-0069-8. URL: http://dl.acm.org/citation.cfm?id=1878537.1878789.

[5] Extreme Optimization. *Numerical Libraries for .NET*. Accessed October 26, 2015. URL: http://www.extremeoptimization.com/VectorMatrixFeatures.aspx.

[6] Paulo Fernandes, Ricardo Presotto, Afonso Sales, and Thais Webber. “An Alternative Algorithm to Multiply a Vector by a Kronecker Represented Descriptor”. In: *21st UK Performance Engineering Workshop*. 2005, pp. 57–67.

[7] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. Accessed October 26, 2015. 2010. URL: http://eigen.tuxfamily.org.

[8] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. “A unified sparse matrix data format for modern processors with wide SIMD units”. In: *CoRR* abs/1307.6209 (2013). URL: http://arxiv.org/abs/1307.6209.

[9] Math.NET. *Math.NET Numerics webpage*. Accessed October 26, 2015. URL: http://numerics.mathdotnet.com/.

[10] Conrad Sanderson. “Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments”. In: (2010).