

# SAND REPORT

SAND2003-1378  
Unlimited Release  
Printed April 2004

## **TSFCore: A Package of Light-Weight Object-Oriented Abstractions for the Development of Abstract Numerical Algorithms and Interfacing to Linear Algebra Libraries and Applications.**

Roscoe A. Bartlett  
Optimization/Uncertainty Estim

Michael A. Heroux  
Computational Math/Algorithms

Kevin R. Long  
Computational Sciences & Math

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,  
a Lockheed Martin Company, for the United States Department of Energy's  
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Telephone: (865) 576-8401  
Facsimile: (865) 576-5728  
E-Mail: [reports@adonis.osti.gov](mailto:reports@adonis.osti.gov)  
Online ordering: <http://www.doe.gov/bridge>

Available to the public from  
U.S. Department of Commerce  
National Technical Information Service  
5285 Port Royal Rd  
Springfield, VA 22161

Telephone: (800) 553-6847  
Facsimile: (703) 605-6900  
E-Mail: [orders@ntis.fedworld.gov](mailto:orders@ntis.fedworld.gov)  
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



# **TSFCore: A Package of Light-Weight Object-Oriented Abstractions for the Development of Abstract Numerical Algorithms and Interfacing to Linear Algebra Libraries and Applications.**

Roscoe A. Bartlett  
Optimization/Uncertainty Estim

Michael A. Heroux  
Computational Math/Algorithms

Kevin R. Long  
Computational Sciences & Math

Sandia National Laboratories,\* Albuquerque NM 87185 USA,

## **Abstract**

Engineering and scientific applications are becoming increasingly modular, utilizing publicly defined interfaces to integrate third party tools and libraries for services such as mesh generation, data partitioning, equation solvers and optimization. As a result, it is important to understand and model the interaction between these various modules, and to develop good abstract interfaces between the primary modules. One category of modules that is becoming increasingly important is abstract numerical algorithms (ANAs). ANAs such as linear and nonlinear equation solvers, methods for stability and bifurcation analysis, uncertainty quantification methods and nonlinear programming solvers for optimization are typically mathematically sophisticated but have surprisingly little essential dependence on the details of what computer system is being used or how matrices and vectors are stored and computed. As a result, using abstract interface capabilities in languages such as C++, we can implement ANA software such that it will work, unchanged, with a variety of applications and linear algebra libraries.

---

\*Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

In this paper we present a package of minimal but complete (with respect to basic required functionality and performance) object-oriented interfaces (implemented in C++) called TSF-Core which allows the development many of these ANAs and simplifies the development of interfaces to applications and linear algebra libraries.

## **Acknowledgement**

The authors would like to thank Paul Boggs, Rob Hoekstra, Rich Lehoucq, and Bart van Bloemen Waanders for their input and suggestions.

The format of this report is based on information found in [31].

## Contents

1	Introduction .....	9
2	Classification of linear algebra interfaces .....	10
3	TSFCore: Basic Requirements .....	12
4	TSFCore: Overview .....	13
5	TSFCore: Details and Examples .....	15
5.1	A motivating example sub-ANA : Compact limited-memory BFGS .....	16
5.2	<i>VectorSpace</i> .....	17
5.2.1	Creation of <i>Vector</i> and <i>MultiVector</i> objects .....	18
5.2.2	Creation of <i>Vector</i> and <i>MultiVector</i> objects from in-core views of data .....	18
5.2.3	General compatibility of <i>Vector</i> objects .....	19
5.3	<i>Vector</i> .....	20
5.3.1	<i>Vector::applyOp(...)</i> .....	20
5.3.2	Explicit access to <i>Vector</i> elements .....	25
5.3.3	Serial vectors and vector spaces .....	28
5.4	<i>LinearOp</i> .....	29
5.4.1	<i>LinearOp::apply(...)</i> .....	30
5.4.2	Optional support for adjoints .....	32
5.5	<i>MultiVector</i> .....	32
5.5.1	Accessing columns of a <i>MultiVector</i> as <i>Vector</i> objects .....	33
5.5.2	<i>MultiVector</i> sub-views .....	34
5.5.3	<i>MultiVector</i> support for <i>applyOp(...)</i> .....	35
5.5.4	<i>Vector</i> and <i>MultiVector</i> correspondence .....	36
5.5.5	<i>MultiVector</i> acting as a <i>LinearOp</i> .....	37
5.5.6	Explicit access to <i>MultiVector</i> elements .....	42
5.6	Aliasing of <i>Vector</i> and <i>MultiVector</i> arguments .....	44
6	An Example Abstract Numerical Algorithm : An Iterative Linear Solver .....	45
7	General Object-Oriented Software Design Concepts and Principles .....	50
7.1	General C++ idioms used by TSFCore .....	50
7.2	General object-oriented design concepts used by TSFCore .....	52
8	Nonessential but Convenient Functionality .....	53
8.1	Sub-vector views as <i>Vector</i> objects .....	54
8.2	Composition of <i>Vector</i> and <i>LinearOp</i> objects .....	54
8.3	Matlab-like notation and handle classes for linear algebra using operator overloading .....	55
9	Summary .....	56
	References .....	59

## Appendix

A	TSFCore C++ class declarations .....	61
---	--------------------------------------	----

## Figures

1	UML [13] class diagram : Interfaces between abstract numerical algorithm (ANA), linear algebra library (LAL), and application (APP) software. . . . .	12
2	UML class diagram : Major components of the TSF interface to linear algebra . . . .	14
3	A compact limited-memory representation of the inverse of a BFGS matrix. . . . .	16
4	Some standard vector operations declared in the header file <code>TSFCoreVectorStdOpsDecl.hpp</code> and defined in the <code>TSFCoreVectorStdOps.hpp</code> header file. . . . .	21
5	Carton of disturbed and locally replicated multi-vectors which are used in the example compact LBFGS sub-ANA when run in SPMD mode on four processors. The process boundaries are shown as dotted lines. The numbers for rows and columns of each multi-vector are also shown. . . . .	39
6	Carton of distributed-memory matrix-matrix product (i.e. block dot product) $T_1 = S^T V$ run in SPMD mode on four processors. This operation first performs local matrix-matrix multiplication with the entries of $S^T$ and $V$ on each processor using level-3 BLAS and then a global reduction summation operation is performed (using MPI) to produce $T_1$ which is returned to all of the processors. . . . .	40
7	Carton of local matrix-matrix product $T_3 = Q_{ss} T_1$ involving locally replicated multi-vectors run in SPMD mode on four processors. In SPMD mode this operation involves no processor-to-processor communication at all. . . . .	41
8	Carton of distributed-local matrix-matrix product $U = S T_3$ involving both distributed and locally multi-vectors run in SPMD mode on four processors. To perform this operation, the local elements in the distributed multi-vector $S$ are multiplied with the locally replicated multi-vector $T_3$ and the result of the matrix-matrix product are set to the local elements of the distributed multi-vector $U$ . In SPMD mode this operation requires no processor-to-processor communication at all. . . . .	42
9	A single-vector version of the preconditioned bi-conjugate gradient method (BiCG). . . . .	47
10	Implementation of an iteration of a multi-vector version of BiCG. . . . .	48



# **TSFCore: A Package of Light-Weight Object-Oriented Abstractions for the Development of Abstract Numerical Algorithms and Interfacing to Linear Algebra Libraries and Applications.**

## **1 Introduction**

One area of steady improvement in large-scale engineering and scientific applications is the increased modularity of application design and development. Specification of publicly-defined interfaces, combined with the use of third-party software to satisfy critical technology needs in areas such as mesh generation, data partitioning and solution methods have been generally positive developments in application design. While the use of third party software introduces dependencies from the application developer's perspective, it also gives the application access to the latest technology in these areas, amortizes library and tool development across multiple applications and, if properly designed, gives the application easy access to more than one option for each critical technology area, e.g., access to multiple linear solver packages.

One category of modules that is becoming increasingly important is abstract numerical algorithms (ANAs). ANAs such as linear and nonlinear equation solvers, methods for stability and bifurcation analysis, transient solvers, uncertainty quantification methods and nonlinear programming solvers for optimization are typically mathematically sophisticated but have surprisingly little essential dependence on the details of what computer system is being used or how matrices and vectors are stored and computed. Thus, by using abstract interface capabilities in languages such as C++, we can implement ANA software such that it will work, unchanged, with a variety of applications and linear algebra libraries. Such an approach is often referred to as *generic programming* [1].

In this paper we describe a set of basic interfaces for the Trilinos Solver Framework (TSF) called TSFCore as the common interface for (i) ANA development, (ii) the integration of an ANA into an application (APP) and (iii) providing services to the ANA from a linear algebra library (LAL). By agreeing on a simple minimal common interface layer such as TSFCore, we eliminate the many-to-many dependency problem of ANA/APP interfaces. TSFCore is not primarily designed to be the most convenient interface for the direct development of ANAs but it can be used in direct ANA development. Instead, TSFCore is designed to make it easier for developers to provide the basic functionality from APPs and LALs required for the implementation of ANAs.

While TSFCore provides a mechanism to express all of the functionality required to be directly used in ANA development it does not attempt to provide a full collection of methods that directly support the anticipated functionality needs of ANAs. Instead TSFCore relies on a simple

but powerful reduction and transformation operator mechanism [10] that can be used to express any element-wise vector reduction or transformation operation. More direct and convenient access to functionality that might be desired by a given ANA is provided in a companion project called TSFExtended [???]. Some extra functionality inbetween what is provided in the basic TSFCore interfaces described here and TSFExtended is contained in utility software that is also contained in the TSFCore namespace which will be referred to as TSFCore/utilities. Extended functionality can be very helpful in developing ANA code and some examples are discussed in Section 8.

It is difficult to describe a set of linear algebra interfaces outside of the context of some class of numerical problems. For this purpose, we will consider numerical algorithms where it is possible to implement all of the required operations exclusively through well defined interfaces to vectors, vector spaces and linear operators. The interfaces described here are the common denominator of all abstract numerical algorithms.

We assume that the reader has a basic understanding of vector reduction/transformation operators (RTOp) (see [10]), is comfortable with object-orientation [23] and C++, and knows how to read basic Unified Modeling Language (UML) [22] class diagrams. We also assume that the reader has some background in large-scale numerics and will therefore be able to appreciate the challenges that are addressed by TSFCore.

To motivate TSFCore, we discuss the context for TSFCore in large-scale (both in lines of code and in problem dimensionality) numerical software in Section 2. The major requirements for TSFCore are spelled out in Section 3. This is followed by an overview of the TSFCore linear algebra interfaces in Section 4 and a detailed discussion of the design of the TSFCore linear algebra interfaces in Section 5 including numerous examples. A complete example ANA for the iterative solution of simultaneous systems of linear equations (using a simple BiCG method) is described in Section 6. A discussion of some of the object-oriented and other general software design concepts and principles that have gone into the development of TSFCore is deferred to Section 7. Some of the nonessential but convenient functionality that is useful to direct ANA developers that is missing in TSFCore is described in Section 8.

## 2 Classification of linear algebra interfaces

Although we will discuss APPs, ANAs and LALs in detail later in this section, we want to briefly introduce these terms here to make them clear. Also, although there are certainly other types of modules in a large-scale application, we only focus on these three.

- Application (APP): The modules of an application that are not ANA or LAL modules. Typically this includes the code that is unique to the application itself such as the code that formulates and generates the discrete problem. In general it would also include other third-party software that is not an ANA or LAL module.

- Abstract Numerical Algorithm (ANA): Software that drives a solution process, e.g., an iterative linear or nonlinear solver. This type of package provides solutions to and requires services from the APP, and utilizes services from one or more LALs. It can usually be written so that it does not depend on the details of the computer platform, or the details of how the APP and LALs are implemented, so that an ANA can be used across many APPs and with many LALs.
- Linear Algebra Library (LAL): Software that provides the ability to construct concrete linear algebra objects such as matrices and vectors. A LAL can also be a specific linear solver or preconditioner.

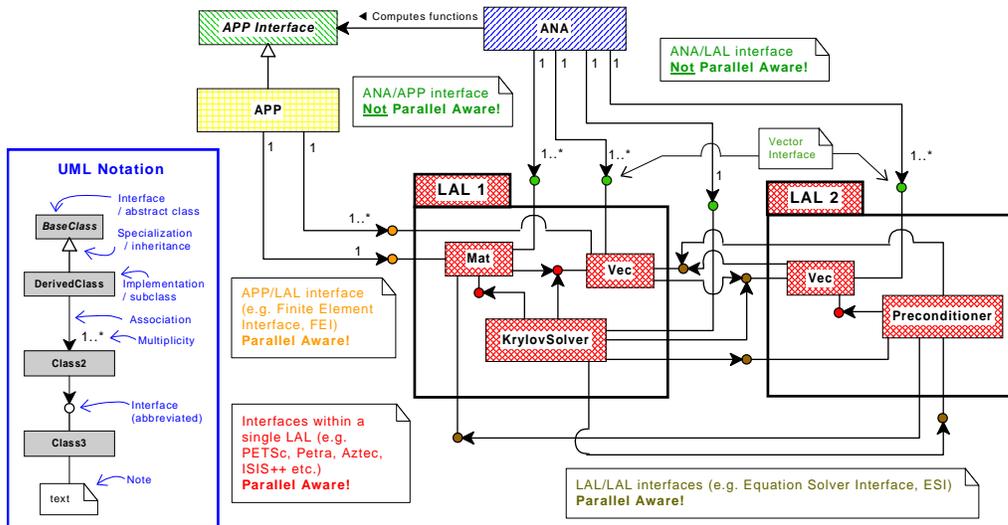
An important focus of this paper is to clearly identify the interfaces between APPs, ANAs and LALs for the purposes of defining the TSFCore interface.

The requirements for the linear algebra objects as imposed by an ANA are very different from the requirements imposed by an APP code. In order to differentiate the various types of interfaces and the requirements associated with each, consider Figure 1. This figure shows the three major categories of software modules that make up a complete numerical application. The first category is application (APP) software in which the underlying data is defined for the problem. This could be something as simple as the right-hand-side and matrix coefficients of a single linear system or as complex as a finite-element method for a 3-D nonlinear PDE-constrained optimization problem. The second category is linear algebra library (LAL) software that implements basic linear algebra operations [19, 2, 12, 28, 3, 26]. These types of software include primarily matrix-vector multiplication, the creation of a preconditioner (e.g. ILU), and may even include several different types of direct linear solvers. The third category is ANA software that drives the main solution process and includes such algorithms as iterative methods for linear and nonlinear systems; explicit and implicit methods for ODEs and DAEs; and nonlinear programming (NLP) solvers [35]. There are many example software packages [3, 28, 26, 16, 11] that contain ANA software.

The types of ANAs described here only require operations like matrix-vector multiplication, linear solves and certain types of vector reduction and transformation operations. All of these operations can be performed with only a very abstract view of vectors, vector spaces and linear operators.

An application code, however, has the responsibility of populating vector and matrix objects and requires the passing of explicit function and gradient value entries, sometimes in a distributed memory parallel environment. This is the purpose of a APP/LAL interface. This involves a very different set of requirements than those described above for the ANA/APP and ANA/LAL interfaces. Examples of APP/LAL interfaces include the FEI [17] and much of the current TSF.

Figure 1 also shows a set of LAL/LAL interfaces that allows linear algebra objects from one LAL to collaborate with the objects from another LAL. These interfaces are very similar to the APP/LAL interfaces and the requirements for this type of interface is also not addressed by TSFCore. The ESI [30] and much of the current TSF contain examples of LAL/LAL interfaces.



**Figure 1.** UML [13] class diagram : Interfaces between abstract numerical algorithm (ANA), linear algebra library (LAL), and application (APP) software.

TSFCore, as described in this paper, specifies only the ANA/LAL interface. TSFCore-based ANA/APP interfaces are described elsewhere (e.g. [8]).

### 3 TSFCore: Basic Requirements

Before describing the C++ interfaces for TSFCore, some basic requirements are stated.

1. TSFCore interfaces should be portable to all the ASC [36] platforms where SIERRA [20] and other ASC applications might run. However, a platform where C++ templates are fundamentally broken will not be a supported platform for TSFCore.
2. TSFCore interfaces should provide for stable and accurate numerical computations at a fundamental level.
3. TSFCore should provide a minimal, but complete, interface that addresses all the basic efficiency needs (in both speed and storage) which will result in near-optimal implementations of all of the linear algebra objects and all of the above mentioned ANA algorithms that use these objects. All other types of nonessential but convenient functionality (e.g. Matlab-like

syntax using operator overloading, see Section 8.3) will not be addressed by TSFCore. This extra functionality can be built on top the basic TSFCore abstractions (e.g. using TSF).

4. ANAs developed with TSFCore should be able to transparently utilize different types of computing environments such as SPMD<sup>1</sup>, client/server<sup>2</sup> and out-of-core<sup>3</sup> implementations.
5. The work required to implement adapter subclasses (see the “Adapter” pattern in [23]) for and with TSFCore should be minimal and straightforward for all of the existing related linear algebra and ANA interfaces (e.g. the linear algebra interfaces in MOOCHO [7] and NOX [29]). This requirement is facilitated by the fact that the TSFCore interfaces are minimal.

A hand-coded program (e.g. using Fortran 77 and MPI) should not provide any significant gains in performance in any of the above categories in any computing environment. If a hand-coded algorithm in Fortran 77 with MPI can significantly improve storage requirements, computational speed or numerical stability. There are many numerical algorithms that can not be considered to be “abstract” and therefore TSFCore and like abstract interfaces should not be used for such algorithms.

## 4 TSFCore: Overview

The basic linear algebra abstractions that make up TSFCore are shown in Figure 2. Complete C++ class declarations for these interfaces are given in Appendix A. The key abstractions include vectors, vector spaces and linear operators. All of the interfaces are templated on the `Scalar` type (the UML notation for templated classes is not used in the figure for the sake of improving readability).

Vector space is the foundation for all other linear algebra abstractions. Vector spaces are abstracted through the `VectorSpace` interface. A `VectorSpace` object acts primarily as an “Abstract Factory” [23] that creates vector objects (which are the “products” in the “Abstract Factory” design pattern).

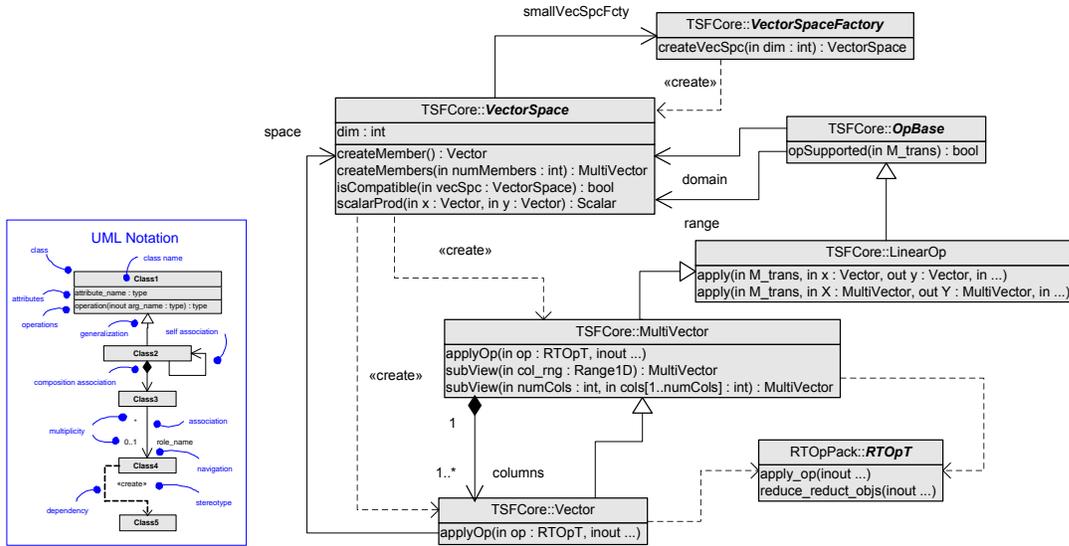
Vectors are abstracted through the `Vector` interface. The `Vector` interface is very minimal and really only defines one nontrivial function `applyOp(...)`. The `applyOp(...)` function accepts user-defined (i.e. ANA-defined) reduction/transformation operator (RTOp) objects through the templated RTOp C++ interface `RTOpPack::RTOpT`. A set of standard vector operations is provided as nonmember functions using standard RTOp subclasses (see Section 5.3). The set of operations is also easily extensible. Every `Vector` object provides access to its `VectorSpace` (that was used to create the `Vector` object) through the function `space()` (shown in Figure 2 as the role name `space` on the association connecting the `Vector` and `VectorSpace` classes).

---

<sup>1</sup>Single Program Multiple Data (SPMD): A single program running in a distributed-memory environment on multiple parallel processors

<sup>2</sup>Client/Server: The ANA runs in a process on a client computer and the APP and LAL run in processors on a server

<sup>3</sup>Out-of-core: The data for the problem is stored on disk and is read from and written to back disk as needed



**Figure 2.** UML class diagram : Major components of the TSF interface to linear algebra

The *VectorSpace* interface also provides the ability to create *MultiVector* objects through the *createMembers(numMembers)* function. A *MultiVector* is a tall thin dense matrix where each column in the matrix is a *Vector* object which is accessible through the *col(...)* function. *MultiVectors* are needed for near-optimal processor cache performance (in serial and parallel programs) and to minimize the number of global communications in a distributed parallel environment. The *MultiVector* interface is useful in many different types ANAs as described later. The interface class *Vector* is derived from *MultiVector* so that every *Vector* is a *MultiVector*. This simplifies the development of ANAs in that any ANA that can handle *MultiVector* objects should automatically be able to handle *Vector* objects as well. The somewhat complex relationship between *Vector* and *MultiVector* is described in Section ???.

*VectorSpace* also declares a virtual function called *scalarProd(x,y)* which computes the scalar product  $\langle x, y \rangle$  for the vector space. This function has a default implementation based on the dot product  $x^T y$ . Subclasses can override the *scalarProd(x,y)* function for other, more specialized, application-specific definitions of the scalar product. There is also a *MultiVector* version *VectorSpace::scalarProds(...)* (not shown in the figure). Finally, *VectorSpace* also includes the ability to determine the compatibility of vectors from different vector spaces through the function *isCompatible(vecSpC)* (see Section 5.2.3). The concepts behind the design of the *VectorSpace*, *Vector* and *MultiVector* interfaces are discussed later in Sections 5.2, 5.3 and 5.5 respectively.

Another important type of linear algebra abstraction is a linear operator which is represented by the interface class *LinearOp*. The *LinearOp* interface is used to represent quantities such as a Jacobian matrix. A *LinearOp* object defines a linear mapping from vectors in one vector space (called the domain) to vectors in another vector space (called the range). Every *LinearOp* object provides access to these vector spaces through the functions `domain()` and `range()` (shown as the role names `domain` and `range` on the associations linking the *OpBase* and *VectorSpace* classes). The exact form of this mapping, as implemented by the function `apply(...)`, is

$$y = \alpha op(M)x + \beta y \tag{1}$$

where  $M$  is a *LinearOp* object;  $x$  and  $y$  are *Vector* objects; and  $\alpha$  and  $\beta$  are Scalar objects. Note that the linear operator in (1) is shown as `op(M)` where `op(M) = M` or `MT` (depending on the argument `M_trans`). This implies that both the non-transposed and transposed (i.e. adjoint) linear mappings can be performed. However, support for transposed (adjoint) operations by a *LinearOp* object are only optional. If an operation is not supported then the function `opSupported(M_trans)` will return `false` (see Section 5.4.2). Note that when `op(M) = MT`, then  $x$  and  $y$  must lie in the range and domain spaces respectively which is the opposite for the case where `op(M) = M`.

In addition to implementing linear mappings for single *Vector* objects, the *LinearOp* interface also provides linear mappings of *MultiVector* objects through an overloaded function `apply(...)` which performs

$$Y = \alpha op(M)X + \beta Y \tag{2}$$

where  $X$  and  $Y$  are *MultiVector* objects. The *MultiVector* version of the `apply(...)` function has a default implementation based on the *Vector* version. The *Vector* version `apply(...)` is a pure virtual function and therefore must be overridden by subclasses. The issues associated with supporting the *MultiVector* version versus the *Vector* version of this function are described in Section 5.5.4.

Section 5 goes into much more detail behind the design philosophy for the core interfaces and the use of these interfaces by both clients and subclass developers.

## 5 TSFCore: Details and Examples

A basic overview of the interface classes shown in Figure 2 was provided in Section 4. In the following sections, we go into more detail about the design of these interfaces and give examples of the use of these classes. Note that in all the below code examples it is assumed that the code is in a source file which include the appropriate header files.

$$H = B^{-1} = \gamma I + \underbrace{\begin{array}{|c|} \hline \text{---} \\ \hline \end{array}}_{S, \gamma Y} + \underbrace{\begin{array}{|c|} \hline \text{---} \\ \hline \end{array}}_Q + \underbrace{\begin{array}{|c|} \hline \text{---} \\ \hline \end{array}}_{S^T, \gamma Y^T}$$

**Figure 3.** A compact limited-memory representation of the inverse of a BFGS matrix.

### 5.1 A motivating example sub-ANA : Compact limited-memory BFGS

To motivate the following discussion and to provide examples, we consider the issues involved in using TSFCore to implement an ANA for the compact limited-memory BFGS (LBFGS) method described in [15]. BFGS and other variable-metric quasi-Newton methods are used to approximate a Hessian matrix  $B \in \mathbf{R}^{n \times n}$  of second derivatives. This approximation is then used to generate search directions for various types of optimization algorithms. The Hessian matrix  $B$  and/or its inverse  $H = B^{-1}$  is approximated using only changes in the gradient  $y = \nabla f(x_{k+1}) - \nabla f(x_k) \in \mathbf{R}^n$  of some multi-variable scalar function  $f(x)$  for changes in the variables  $s = x_{k+1} - x_k \in \mathbf{R}^n$ . A set of matrix approximations  $B_k$  are formed using rank-2 updates where each update takes the form

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}. \quad (3)$$

In a limited-memory BFGS method, only a fixed maximum number  $m_{\max}$  of updates are stored

$$XS = \begin{bmatrix} s_1 & s_2 & \dots & s_m \end{bmatrix} \in \mathbf{R}^{n \times m} \quad (4)$$

$$Y = \begin{bmatrix} y_1 & y_2 & \dots & y_m \end{bmatrix} \in \mathbf{R}^{n \times m} \quad (5)$$

where  $m \leq m_{\max}$  is the current number of stored updates and  $S$  and  $Y$  are multi-vectors (note that the subscripts in (4)–(5) correspond to column indexes in the multi-vector objects, not iteration counters  $k$ ). When an optimization algorithm begins,  $m = 0$  and  $m$  incremented each iteration until  $m = m_{\max}$  after which the method starts dropping older update pairs  $(s, y)$  to make room for newer ones. In a compact LBFGS method, the inverse  $H$  (shown in Figure 3) of the quasi-Newton matrix  $B$  (where when the index  $k$  is dropped, it implicitly refers to the current iteration  $B_k$ ) on is

approximated using the tall thin multi-vectors  $S$  and  $Y$  along with a small (serial) coordinating matrix  $Q$  (which is computed and updated from  $S$  and  $Y$ ). The scalar  $\gamma$  is chosen for scaling reasons and  $H_0 = B_0^{-1} = \gamma I$  represents the initial matrix approximation from which the updates are performed. A similar compact formula also exists for  $B$  which involves the same matrices (and requires solves with  $Q$ ). In an SPMD configuration, the multi-vectors  $Y$  and  $S$  may contain vector elements spread over many processors. However, the number of columns  $m$  in  $S$  and  $Y$  is usually less than 40. Because of the small number of columns in  $S$  and  $Y$ , all of the linear algebra performed with the matrix  $Q$  is performed serially using dense methods (i.e. BLAS and LAPACK). A parallel version of the compact LBFGS method is implemented, for example, as an option in MOOCHO. TSFCore supports efficient versions of all of the operations needed for a near-optimal parallel implementation of this LBFGS method.

The requirements for this sub-ANA will be mentioned in several of the following sections along with example code.

## 5.2 VectorSpace

The basic design of the *VectorSpace* interface was taken directly from the Hilbert Class Library (HCL) [24] which is also used in *AbstractLinAlgPack* (the basic linear algebra interfaces in MOOCHO [???]).

We now show a simple code example as to the use of the *VectorSpace* and *Vector* interfaces. The following code snippet shows a function that performs several types of tasks:

```
template<class Scalar>
void TSFCore::foo0( const VectorSpace<Scalar>& vecSpc, const LinearOp<Scalar>& M )
{
  TEST_FOR_EXCEPTION(!vecSpc.isCompatible(*M.domain()),std::logic_error,"Error!"); // Check compatibility
  Teuchos::RefCountPtr<Vector<Scalar> > x = vecSpc.createMember(); // Create new vector x
  Teuchos::RefCountPtr<Vector<Scalar> > y = M.range()->createMember(); // Create new vector y
  assign(x.get(),1.0); // x = 1.0
  M.apply(NOTRANS,*x,y.get()); // y = M*x
  M.apply(TRANS,*y,x.get(),0.5,0.1); // x = 0.5*M*y + 0.1*x
}
```

The above code snippet shows how memory management in TSFCore is handled – through the templated smart reference-counted pointer class `Teuchos::RefCountPtr<>` (see Section 7). The vector objects pointed to by the objects `x` and `y` are accessed in various ways in the last three lines. For instance, in the statement

```
assign(x.get(),1.0);
```

the raw C++ pointer (of type `Vector<Scalar>*`) to the underlying vector object is returned using the function `RefCountPtr<>::get()`. The function `assign(...)` is implemented through an `RTOp` object and its implementation is shown in Section 5.3.1. The next statement

```
M.apply(NOTRANS, *x, y.get());
```

shows the created vectors being passed into the `apply(...)` function of a *LinearOp* object. The expression `*x` invokes the function `RefCountPtr<>::operator*()` which returns a reference (of type `Vector<Scalar>&`) to the underlying vector object.

### 5.2.1 Creation of *Vector* and *MultiVector* objects

As stated above, one of the major roles of a *VectorSpace* object is as an abstract factory for *Vector* and *MultiVector* objects. The primary creational functions are:

```
template<class Scalar>
class VectorSpace {
public:
    ...
    virtual Teuchos::RefCountPtr< Vector<Scalar> > createMember() const = 0;
    virtual Teuchos::RefCountPtr< MultiVector<Scalar> > createMembers(int numMembers) const;
    ...
};
```

These creational functions return smart reference-counted pointers to the created *Vector* and *MultiVector* objects. Note that the multi-vector creational function `createMembers(...)` takes an argument `numMembers` that determines the number of columns in the multi-vector object.

### 5.2.2 Creation of *Vector* and *MultiVector* objects from in-core views of data

In very rare occasions in order to achieve near-optimal performance it is necessary that *Vector* and *MultiVector* objects be created as views of raw data that is owned by the client. The functions that allow this are:

```
template<class Scalar>
class VectorSpace {
public:
    ...
    virtual Teuchos::RefCountPtr<Vector<Scalar> > createMemberView(
        const RTopPack::MutableSubVectorT<Scalar> &raw_v ) const;
    virtual Teuchos::RefCountPtr<const Vector<Scalar> > createMemberView(
        const RTopPack::SubVectorT<Scalar> &raw_v ) const;
    virtual Teuchos::RefCountPtr<MultiVector<Scalar> > createMembersView(
        const RTopPack::MutableSubMultiVectorT<Scalar> &raw_mv ) const;
    virtual Teuchos::RefCountPtr<const MultiVector<Scalar> > createMembersView(
        const RTopPack::SubMultiVectorT<Scalar> &raw_mv ) const;
    ...
};
```

These functions create both mutable and non-mutable views of raw data. Note that efficient implementations of such views are only possible for vector and multi-vectors that have all data stored locally or in-core. Such views can be created efficiently for serial and SPMD vectors and multi-vectors. Serial and SPMD subclasses should override these functions to create the views in an efficient way.

These view functions all have default implementations that create temporary *Vector* or *Multi-Vector* objects (using the basic *createMember()* and *createMembers()* functions respectively), then copy in data (using the explicit element access functions described in Section ??? and ???) and then for mutable views (just before the vector or multi-vector is destroyed) copy the data in the temporary vector and multi-vector objects back into the raw data arrays. Because of these default implementations, these functions are supported automatically by every vector and multi-vector implementation and these default implementations may not be a performance problem in many different use cases.

### 5.2.3 General compatibility of *Vector* objects

There is one important aspect that distinguishes `TSFCore::VectorSpace` from vector space interfaces in HCL and TSF for instance. In HCL 1.0, the compatibility of vector spaces is tested with a virtual `operator==(...)` function. This implies that vector spaces will be compatible only if they are of the same concrete type and have the same setup. Ideally, however, we do not want to require that only vectors and vector spaces with the same *concrete* type to be compatible but instead we would like to allow vectors and vector spaces of the same *general* type to be compatible. To see the difference, consider parallel programs running in an SPMD configuration where vector elements are partitioned across processors and communication is handled using MPI [21]. There are several different linear algebra libraries that are designed to work in such an environment such as Aztec [27], Epetra [25] and PETSc [3]. TSFCore adapter subclasses would be created for vectors and vector spaces for each of these packages. In principle, all implementations of SPMD MPI vectors that have the same partitioning of elements to processors should be compatible, regardless of which underlying libraries are involved. The RTop design, given the appropriate *VectorSpace* and *Vector* interfaces, allows the seamless integration of vectors of different *concrete* types given the same *general* type. If all of these adapter subclasses inherited from the node interface classes *MPIVectorSpaceBase* and *MPIVectorBase* (see the Doxygen documentation) which include an appropriate set of abstract functions (like determining compatibility of maps and access to local vector data), then Epetra vectors should be transparently compatible with PETSc and Aztec vectors and so on. This type of interoperability is demonstrated for serial vectors and vector spaces in Section 5.3.3

## 5.3 *Vector*

The core design principles behind the *Vector* interface and the *applyOp(...)* function (which accepts RTOp objects) are described in [10]. The benefits of the RTOp approach can be summarized as follows.

1. LAL developers need only implement one operation — *applyOp(...)* — and not a large collection of primitive vector operations.
2. ANA developers can implement *specialized* vector operations without needing any support from LAL maintainers.
3. ANA developers can optimize time consuming vector operations on their own for the platforms they work with.
4. Reduction/transformation operators are more efficient than using primitive operations and temporary vectors.
5. ANA-appropriate vector interfaces that desire built-in standard vector operations (i.e. *axpy* and norms) can use RTOp operators for the implementations of these operations (for example, see *TSFExtended::Vector*).

The *applyOp(...)* function is described in more detail in Section 5.3.1. Note that this approach does not hinder the development of convenience functions in any way. In fact, a set of basic operations is already available in the header file *TSFCoreVectorStdOpsDecl.hpp*. The declarations for the functions in this file are shown in Figure 4. Note, to use these template functions you should include the definitions from *TSFCoreVectorStdOps.hpp* (never directly *#include* a *xxxDecl.hpp* file unless you know what you are doing, instead, include the *xxx.hpp* file for all of the *TSFCore* code). Using one of these non-member vector functions is transparently obvious and there is not even one hint that the function *Vector::applyOp(...)* is involved.

### 5.3.1 *Vector::applyOp(...)*

Several important issues regarding the specification of the *Vector::applyOp(...)* function were not discussed in [10]. Before describing these issues, note that the *Vector::applyOp(...)* function is not directly called by a client (it is protected) but instead is called through a non-member (friend) function of the same name. This is done to provide a uniform way to deal with all of the allowed permutations of the number and types of vector arguments to this function when the function is called by the client. Therefore, we will only consider the prototype for the non-member function *TSFCore::applyOp(...)* which is

```

-----
// TSFCoreVectorStdOpsDecl.hpp
...
namespace TSFCore {
template<class Scalar> Scalar sum( const Vector<Scalar>& v );           // result = sum(v(i))
template<class Scalar> Scalar norm( const Vector<Scalar>& v );       // result = sqrt(<v,v>)
template<class Scalar> Scalar norm_1( const Vector<Scalar>& v );     // result = ||v||1
template<class Scalar> Scalar norm_2( const Vector<Scalar>& v );     // result = ||v||2
template<class Scalar> Scalar norm_inf( const Vector<Scalar>& v_rhs ); // result = ||v||inf
template<class Scalar> Scalar dot( const Vector<Scalar>& x
                                ,const Vector<Scalar>& y );         // result = x'*y
template<class Scalar> Scalar get_ele( const Vector<Scalar>& v, Index i ); // result = v(i)
template<class Scalar> void set_ele( Index i, Scalar alpha
                                ,Vector<Scalar>* v );             // v(i) = alpha
template<class Scalar> void assign( Vector<Scalar>* y, const Scalar& alpha ); // y = alpha
template<class Scalar> void assign( Vector<Scalar>* y
                                ,const Vector<Scalar>& x );       // y = x
template<class Scalar> void Vp_S( Vector<Scalar>* y, const Scalar& alpha ); // y += alpha
template<class Scalar> void Vt_S( Vector<Scalar>* y, const Scalar& alpha ); // y *= alpha
template<class Scalar> void Vp_StV( Vector<Scalar>* y, const Scalar& alpha
                                ,const Vector<Scalar>& x );       // y = alpha*x + y
template<class Scalar> void abs( Vector<Scalar>* y, const Vector<Scalar>& x ); // y(i) = abs(x(i))
template<class Scalar> void reciprocal( Vector<Scalar>* y
                                       ,const Vector<Scalar>& x ); // y(i) = 1/x(i)
template<class Scalar> void ele_wise_prod( const Scalar& alpha
                                       ,const Vector<Scalar>& x, const Vector<Scalar>& v, Vector<Scalar>* y ); // y(i)+=alpha*x(i)*v(i)
template<class Scalar> void ele_wise_divide( const Scalar& alpha
                                       ,const Vector<Scalar>& x, const Vector<Scalar>& v, Vector<Scalar>* y ); // y(i)=alpha*x(i)/v(i)
template<class Scalar> void linear_combination( const int m
                                               // y(i) = beta*y(i)
                                               ,const Scalar alpha[], const Vector<Scalar>* x[] // + alpha[0]*x[0](i)+...
                                               const Scalar &beta, Vector<Scalar> *y ); // +.alpha[m-1]*x[m-1](i)
template<class Scalar> void seed_randomize( unsigned int ); // Seed for randomize()
template<class Scalar> void randomize( Scalar l, Scalar u, Vector<Scalar>* v ); // v(i) = random(l,u)
} // end namespace TSFCore
-----

```

**Figure 4.** Some standard vector operations declared in the header file `TSFCoreVectorStdOpsDecl.hpp` and defined in the `TSFCoreVectorStdOps.hpp` header file.

```

template<class Scalar>
void TSFCore::applyOp(
    const RTopPack::RTopT<Scalar> &op
    ,const size_t num_vecs, const Vector<Scalar>* vecs[]
    ,const size_t num_targ_vecs , Vector<Scalar>* targ_vecs[]
    ,RTopPack::ReductTarget *reduct_obj
    ,const Index first_ele = 1, const Index sub_dim = 0, const Index global_offset = 0
    );

```

and has nine arguments: the RTop object that defines the reduction/transformation operation to be performed `op`; the non-mutable input vectors specified by `num_vecs` and `vecs[]` (`num_vecs==0` and `vecs==NULL` allowed); the mutable input/output vectors specified by `num_targ_vecs` and `targ_vecs[]` (`num_targ_vecs==0` and `targ_vecs==NULL` allowed); the input/output opaque reduction target object `reduct_obj` (must be set to `NONE` if no reduction is defined); the range of elements defining the sub-vector to apply the operator to specified by `first_ele` and `sub_dim`; and the global offset `global_offset` to use when applying coordinate-variant operators.

The role of the first five arguments in `TSFCore::applyOp(...)` should be clear from the discussion in [10]. However, the special handling of the object `reduct_obj` and the use cases where the last three arguments are important need to be carefully explained since they are critical to the success of this design. In short, what this specification allows is the ability to take *Vector* objects and then be able to put together abstract compositions of them to create new (logical) vector *Vector* objects. There are primarily four use cases that this specification is designed to support: (a) treating all of the elements in a *Vector* object as a single logical vector, (b) targeting an RTop operator to a specific element or range of elements, (c) creating a sub-view of an existing vector and treating it as a vector in its own right, and (d) creating a new, larger composite (i.e. block, or product) abstract vector out of a collection of other vector objects.

The first use case (a), where all of the elements in a *Vector* object are treated as a single logical vector, is the most common one. Here, the default argument values of `first_ele=1`, `sub_dim=0` (the value 0 is a flag to indicate that all of the remaining elements should be included) and `global_offset=0` are used and `TSFCore::applyOp(...)` is called with the vector arguments. For example, consider the invocation of an assignment-to-scalar transformation operator in the following function.

```

template<class Scalar>
void TSFCore::assign( Vector<Scalar>* y, const Scalar& alpha )
{
    TEST_FOR_EXCEPTION(y==NULL, std::logic_error, "assign(...), Error!"); // Validate input
    RTopPack::TopAssignScalar<Scalar> assign_scalar_op(alpha); // Create the operator
    Vector<Scalar>* targ_vecs[] = { y }; // Set up vector args
    applyOp<Scalar>(assign_scalar_op, 0, NULL, 1, targ_vecs, NULL); // Invoke operator
}

```

In the above function, the operator `assign_scalar_op` of type `RTopPack::RTopAssignScalar` only performs a transformation which does not require a reduction object. In these cases a `NONE` pointer is passed in for the reduction object `reduct_obj`.

If a reduction is being performed, the reduction object is initialized prior to a single call to `TSFCore::applyOp(...)` and then the reduction value is extracted. The following function shows an example where the norm  $\| \cdot \|_2$  is computed

```
template<class Scalar>
Scalar TSFCore::norm_2( const Vector<Scalar>& v )
{
    RTOPackROpNorm2<Scalar> norm_2_op;           // Create the RTOp operator object
    Teuchos::RefCountPtr<RTOPack::ReductTarget>
        norm_2_targ = norm_2_op.reduct_obj_create(); // Create (init) reduction object
    const Vector<Scalar>* vecs[] = { &v };      // Set up non-mutable vector args
    applyOp<Scalar>(norm_2_op,1,vecs,0,NULL,&*norm_2_targ); // Invoke the reduction operator
    return norm_2_op(*norm_2_targ);            // Extract reduction value
}
```

A great many implementations of RTOp operator subclasses are already available and wrapper functions to several of the more standard operations, including the above functions `assign(y, alpha)` and `norm_2(v)`, are defined in the header file `TSFCoreVectorStdOps.hpp` shown in Figure 4.

The second use case (b) is where the client targets an RTOp operator for a specific element or set of elements in a Vector object. Two important examples are getting and setting individual vector elements. This can be accomplished without having to write specialized RTOp subclasses for these cases. For example, getting an element can be performed using a standard RTOp subclass as is done in the following function.

```
template<class Scalar>
Scalar TSFCore::get_ele( const Vector<Scalar>& v, Index i )
{
    RTOPack::ROpSum<Scalar> sum_op;           // Create RTOp operator object
    Teuchos::RefCountPtr<RTOPack::ReductTarget>
        sum_targ = sum_op.reduct_obj_create(); // Create (init) reduction object
    const Vector<Scalar>* vecs[1] = { &v };  // Set up non-mutable vector args
    applyOp<Scalar>(sum_op,1,vecs,0,NULL,&*sum_targ,i,1); // Invoke the reduction operator
    return sum_opt(*sum_targ);              // Extract reduction value
}
```

In the above call to `TSFCore::applyOp(...)`, the argument `global_offset` is left at its default value of 0, since this argument is ignored by the RTOp object `sum_op` anyway (the sum operator is coordinate invariant).

Setting a vector element is performed in a similar manner using the same transformation RTOp operator subclass for assigning the elements of a vector that was used in the `assign(...)` function shown above. The following function shows how setting a vector element is performed using this transformation operator.

```
template<class Scalar>
void TSFCore::set_ele( Index i, Scalar alpha, Vector<Scalar>* v )
```

```

{
  TEST_FOR_EXCEPTION(v==NULL, std::logic_error, "set_ele(...), Error!"); // Validate input
  RTOpPack::TOpAssignScalar<Scalar> assign_scalar_op(alpha); // Create op object
  Vector<Scalar>* targ_vecs[1] = { v }; // Set up vector args
  applyOp<Scalar>(assign_scalar_op, 0, NULL, 1, targ_vecs, NULL, i, 1); // Invoke operator
}

```

Again, since the assignment operator is also coordinate invariant, the `assign_scalar_op` object ignores the `global_offset` argument so `global_offset` is left at its default value in the call to `TSFCore::applyOp(...)`.

For an example of the third use case (c), where a sub-view of an existing vector is treating as a vector in its own right, consider an optimization algorithm where the state  $y$  and design  $u$  variables are physically concatenated into a single serial vector  $x^T = [y^T \ u^T]$ . For example, if  $n_y = 10$  and  $n_u = 5$ , then the dimension of the vector  $x$  would be  $n_x = 15$ . There are parts of the algorithm where it is most convenient to treat all of the variables  $x$  the same and there are others where access to the individual state  $y$  and design  $u$  sub-vectors of  $x$  is required. Now suppose that a *Vector* object  $x$  is directly used by an optimization algorithm. When the optimization algorithm needs to apply an RTOp operator to the state variables  $y$ , it sets `first_ele=1` and `sub_dim=10` and then calls `TSFCore::applyOp(...)` (leaving the default value of `global_offset=0`). When the algorithm needs to apply an RTOp operator to the design variables  $u$ , it sets `first_ele=11` and `sub_dim=5` and then calls `TSFCore::applyOp(...)` (also leaving the default value of `global_offset=0`). In each case, if a reduction is being performed, the reduction object is initialized prior to a single call to `TSFCore::applyOp(...)` and then the reduction value is extracted just as in the first use case (a). For example, the following function computes the  $\| \cdot \|_2$  norms for the state and design sub-vectors given the vector object  $x$ .

```

template<class Scalar>
void TSFCore::compute_norm_2( const Vector<Scalar>& x, Index ny, Scalar* nrm_2_y, Scalar* nrm_2_u )
{
  const Index nx = x.space()->dim(), nu = nx - ny; // Get dimensions
  RTOpPack::ROpNorm2<Scalar> norm_2_op; // Create op object
  Teuchos::RefCountPtr<RTOpPack::ReductTarget>
    norm_2_targ = norm_2_op.reduct_obj_create(); // Create (init) reduction object
  const Vector<Scalar>* vecs[1] = { &x }; // Set up non-mutable vector args
  applyOp<Scalar>(norm_2_op, 1, vecs, 0, NULL, &*norm_2_targ, 1, ny); // Invoke the operator for y
  *nrm_2_y = norm_2_op(*norm_2_targ); // Extract the value of ||y||2
  norm_2_op.reduct_obj_reinit(&*norm_2_targ); // Reinitialize reduction object
  applyOp<Scalar>(norm_2_op, 1, vecs, 0, NULL, &*norm_2_targ, ny+1, ny+nu); // Invoke operator for u
  *nrm_2_u = norm_2_op(*norm_2_targ); // Extract the value of ||u||2
}

```

Finally, as an example of the fourth use case (d), where a new larger composite (i.e. block) abstract vector is created out of a collection of other abstract vectors, we use the same optimization example as above, except this time the vector  $x$  is actually represented as two separate *Vector* objects  $y$  and  $u$ . In this case, a new composite blocked or product vector

$$x = \begin{bmatrix} y \\ u \end{bmatrix}$$

is abstractly created which lies in a new product vector space  $\mathcal{X} = \mathcal{Y} \times \mathcal{U}$ . With that said, consider how the element with the maximum absolute value and its index can be determined for the full vector  $x$  given separate `Vector` objects for the state  $y$  and design  $u$  variables. This can be done with the predefined `RTOp` subclass `ROpMaxAbsEle` which is applied in the following function.

```
template<class Scalar>
void TSFCore::compute_max_abs_ele( const Vector<Scalar>& y, const Vector<Scalar>& u
    ,Scalar* x_max, Index* x_i )
{
    const Index ny = y.space()->dim(), nu = u.space()->dim(); // Get dimensions
    RTOpPack::ROpMaxAbsEle<Scalar> max_abs_ele_op; // Create op object
    Teuchos::RefCountPtr<RTOpPack::ReductTarget>
        max_abs_ele_targ = max_abs_ele_op.reduct_obj_create(); // Create (init) reduction object
    const Vector<Scalar>* vecs[1]; // Declare array
    vecs[0] = &y; // Set pointer to y
    applyOp<Scalar>(max_abs_ele_op,1,vecs,0,NULL,&*max_abs_ele_targ,1,0,0); // Reduce over y
    vecs[0] = &u; // Set pointer to u
    applyOp(max_abs_ele_op,1,vecs,0,NULL,&*max_abs_ele_targ,1,0,ny); // Combine with reduction over u
    *x_max = max_abs_ele_op(*max_abs_ele_targ).x_max(); // Extract reduction values
    *x_i = max_abs_ele_op(*max_abs_ele_targ).x_i(); // ...
}
```

The above reduction operation is not coordinate invariant and therefore the value of `global_offset` is critical in the calls to `TSFCore::applyOp(...)`.

Note that optimization algorithms are not the only ANAs that require the (logical) composition of individual `Vector` objects into a single vector. For example, SFE methods form a large blocked SFE system out of several smaller deterministic systems [37]. There can also be multiple levels of blocking such as embedding a blocked SFE set of state vectors  $y^T = [ \tilde{y}_1^T \ \tilde{y}_2^T \ \dots \ \tilde{y}_N^T ]$  into the blocked set of optimization variables  $x^T = [ y^T \ u^T ]$ . The basic functionality in `Vector::applyOp(...)` supports all of these examples through the above use cases.

### 5.3.2 Explicit access to `Vector` elements

Another important feature of the `Vector` interface regards the functions that can be used to gain explicit access to the vector elements (which are not shown in the UML diagram in Figure 2). First, it should be noted that requesting explicit access to vector elements is ill-advised in general (especially in an SPMD or client-server environment). However, there are instances where this is perfectly appropriate.

One use case where explicit vector element access may be required is when the vector lies in the domain space of a `MultiVector` object. This, for example, is needed in the implementation of the compact LBFSG method described in Section 5.1 above. For the implementation of this compact LBFSG matrix, it is critical to be able to explicitly access elements in the domain space of  $Y$  and  $S$  in order to compute and update the coordinating matrix  $Q$ . Another situation when explicit access to vector elements is appropriate and needed is when the vector is in a small dimensional

design space in an optimization problem and where the ANA uses dense quasi-Newton methods to approximate the reduced Hessian of the Lagrangian (e.g. this is one option in MOOCHO).

Another use case where explicit element access is critical is when a vector is “in-core” and this allows the seamless integration of all serial vectors (see Section 5.3.3).

The functions in *Vector* support three different types of use cases with respect to explicit element access: (a) extracting a non-mutable view of the vector elements; (b) extracting a mutable view of the vector elements and then committing the changes back to the vector object; and finally, (c) explicitly setting the elements in the vector. The prototypes for these functions are shown below.

```
namespace TSFCore {

template<class Scalar>
class VectorSpace {
public:
    ...
    virtual bool isInCore() const;
    ...
};

namespace TSFCore {
template<class Scalar>
class Vector {
public:
    ...
    virtual void getSubVector( const RangeID& rng, RTOpPack::SubVectorT<Scalar>* sub_vec ) const;
    virtual void freeSubVector( RTOpPack::SubVectorT<Scalar>* sub_vec ) const;
    virtual void getSubVector( const RangeID& rng, RTOpPack::MutableSubVectorT<Scalar>* sub_vec );
    virtual void commitSubVector( RTOpPack::MutableSubVectorT<Scalar>* sub_vec );
    virtual void setSubVector( const RTOpPack::SparseSubVectorT<Scalar>& sub_vec );
    ...
};

} // namespace TSFCore
```

All of these functions have reasonably efficient default implementations based on fairly sophisticated RTOp subclasses and *Vector::applyOp(...)*. The default implementations of the *getSubVector(...)* functions require dynamic memory allocation. For most use cases, *Vector* subclasses usually do not need to override these functions for the sake of efficiency but may need to override them for other reasons (see the subclass *SerialVector* in Section 5.3.3 and the interface *MPIVectorBase* in the Doxygen documentation). The *VectorSpace* function *isInCore()* returns true if all of the vector’s elements are easily accessible to all of the calling processes and therefore these explicit vector access functions are an efficient way to get at the explicit elements. This function should not generally be called by typical client code but instead is designed to be used by more specialized types of purposes (e.g. see the class *MPIVectorSpaceBase* in the Doxygen documentation).

In the first use case (a) – extracting and releasing a non-mutable view of the vector elements – involves calling the const functions *getSubVector(...)* and *freeSubVector(...)* respectively.

These functions use the C++ class `RTOpPack::SubVectorT<>` that is build into the C++ interfaces for `RTOp` and was therefore a natural choice for this purpose. To demonstrate the use of these functions the following example function copies the elements from a `Vector` object into a raw C++ array.

```
template<class Scalar>
void fool( const Vector<Scalar>& x, Scalar v[] )
{
    RTOpPack::SubVectorT<Scalar> sub_vec;           // Create (int) subvector view object
    x.getSubVector(Range1D(),&sub_vec);           // Initialize the view object
    for( Index i = 0; i < sub_vec.subDim(); ++i ) // Loop through the explicit elements
        v[i] = sub_vec(i+1);                       // Extract values
    x.freeSubVector(&sub_vec);                       // Free the view of the vector x
}
```

In the statement

```
x.getSubVector(Range1D(),&sub_vec);
```

the constructed `Range1D()` object represents the full range of vector elements (this is similar to the colon `:` syntax in Matlab). Note that this function call may require dynamic memory allocation in order to create a strided view of the vector elements that is represented in the output argument `sub_vec`. The data pointed to by `sub_vec.values()` may be dynamically allocated which is why it is necessary to call

```
x.freeSubVector(&sub_vec);
```

after the view in `sub_vec` is no longer needed in order to possibly free dynamically allocated memory.

The process of extracting, modifying and committing a mutable view of vector elements, in the second use case (b), involves the non-const functions `getSubVector(...)` and `commitSubVector(...)` respectively. These functions use the C++ class `RTOpPack::MutableSubVectorT<>`. As an example, consider the following function that accepts a raw C++ array of values and then adds them to a `Vector` object's elements.

```
template<class Scalar>
void foo2( const Scalar v[], Vector<Scalar>* x )
{
    RTOpPack::MutableSubVectorT<Scalar> sub_vec; // Create (init) subvector view object
    x->getSubVector(Range1D(),&sub_vec);         // Initialize the view object
    for( Index i = 0; i < sub_vec.subDim(); ++i ) // Loop through the explicit elements
        sub_vec[i] += v[i];                       // add v[] to elements
    x->commitSubVector(&sub_vec);                 // Commit and free the view of x
}
```

The last use case (c) is where a client simply wants to set elements without creating a view. This is accomplished through the non-const function `setSubVector(...)`. This function uses yet another built-in RTOp C++ class called `RTOpPack::SparseSubVectorT<>`. This class is different from the `RTOpPack::SubVectorT<>` and `RTOpPack::MutableSubVectorT<>` classes in that `RTOpPack::SparseSubVectorT<>` also allows the representation of sparse vectors. This is very useful for quickly and efficiently setting up sparse *Vector* objects. For example, one way to initialize a *Vector* object to represent a column of identity (i.e. an “eta” vector  $e_i$ ) is to use a function like the following.

```
template<class Scalar>
void set_eta_vec( Index i, Vector<Scalar>* e_i )
{
    const Scalar av[] = { 1.0 };           // Create array for the values
    const Index  ai[] = { i   };           // Create array for the indexes
    RTOpPack::SparseSubVectorT<Scalar>
        sub_vec(0,e_i->dim(),1,av,1,ai,1,0,1); // ...
    x->setSubVector(sub_vec);               // Set all x = 0 except x(i) = 1.0
}
```

### 5.3.3 Serial vectors and vector spaces

One of the remarkable features of the design of the *VectorSpace* and *Vector* interfaces is that they allow, in principle, for all serial vectors of the same dimension to be automatically compatible with little work. Here we use the term serial to mean that all of the vector elements are stored in core in the same process where the ANA is running. While this may not sound remarkable at first thought consider the fact that there exist numerous C++ classes libraries that contain some concept of a serial vector [32, 38, 39, 40] which are all largely incompatible (except perhaps through explicit element access using `operator[]` or `operator()` but certainly only through compile time polymorphism (i.e. C++ templates)). With TSFCore, these incompatibilities are not an issue. The way that this works is exemplified by the subclasses *SerialVectorSpace* and *SerialVector* which are derived from the node subclasses *SerialVectorSpaceBase* and *SerialVectorBase* respectively.

The first step is for every serial *VectorSpace* subclass to implement the `isCompatible(...)` function in the same way as shown below (using *SerialVectorSpaceBase* as the example).

```
template<class Scalar>
bool SerialVectorSpaceBase<Scalar>::isCompatible( const VectorSpace<Scalar>& aVecSpc ) const
{
    return this->dim() == aVecSpc.dim() && this->isInCore() && aVecSpc.isInCore();
}
```

The above implementation makes the assumption that if the dimensions of the vector spaces are the same and both vectors are stored in core, then the vectors themselves should also be compatible (through the efficient use of the explicit sub-vector element access functions, first introduced in Section 5.3.2, as described below).

The second critical step is to have every serial *Vector* subclass override the explicit sub-vector access functions `getSubVector(...)` (both the `const` and `non-const` versions), `freeSubVector(...)` and `commitSubVector(...)` to perform these operations without calling the `applyOp(...)` function (see the subclass `SerialVector`).

The third step is to have every serial *Vector* subclass override and implement the function `applyOp(...)` in the same way as shown below (using the `SerialVectorBase` node subclass as the example).

```
template<class Scalar>
void TSFCore::SerialVectorBase::applyOp(
    const RTOpPack::RTOpT<Scalar> &op, const size_t num_vecs, const Vector<Scalar>* vecs[]
    ,const size_t num_targ_vecs, Vector<Scalar>* targ_vecs[]
    ,RTOpPack::ReductTarget *reduct_obj
    ,const Index first_ele, const Index sub_dim, const Index global_offset
    ) const
{
    ...
    in_applyOp_ = true;
    TSFCore::apply_op_serial(
        op,num_vecs,vecs,num_targ_vecs,targ_vecs,reduct_obj
        ,first_ele,sub_dim,global_offset
    );
    in_applyOp_ = false;
}
```

The implementation of the above `applyOp(...)` function is really quite simple and it uses a helper function `apply_op_serial(...)` that takes care of all of the details of calling the sub-vector extraction functions on the `Vector` objects. No dynamic casting is performed during this process and in the case of `SerialVector`, no dynamic memory allocation is performed either. Therefore, for sufficiently large serial vectors, the overhead of these function calls will be swamped by computation in the `RTOp` operators, yielding near-optimal performance.

There are cases where it can not be determined until runtime whether a vector is serial or not. In these cases the concrete subclasses can not simply derive from the `SerialVectorSpaceBase` and `SerialVectorBase` node subclasses but must instead implement this this functionality themselves to be used when it is determined that the vectors are indeed serial (see the `Epetra` `TSFCore` adapter subclasses `TSFCore::EpetraVectorSpace` and `TSFCore::EpetraVector` for instance).

By using this simple approach to developing serial *VectorSpace* and *Vector* subclass, the details of putting together many different types of numerical algorithms becomes much easier.

## 5.4 *LinearOp*

This section continues the discussion started in Section 4 for the *LinearOp* interface and includes some examples.

### 5.4.1 *LinearOp::apply(...)*

The C++ prototype for the *Vector* version of *LinearOp::apply(...)* is

```
namespace TSFCore{
template<class Scalar>
class LinearOp : public virtual OpBase<Scalar> {
public:
    ...
    virtual void apply(
        ETransp M_trans, const Vector<Scalar> &x, Vector<Scalar> *y
        ,Scalar alpha = 1.0, Scalar beta = 0.0
        ) const = 0;
    ...
};
} // namespace TSFCore
```

where the type *ETransp* is the C++ enum

```
enum ETransp { NOTRANS, TRANS, CONJTRANS };
```

The use of an enum instead of a simple bool for the *M\_trans* argument is very important. The use of an enum disallows the implicit conversion from other types like char, int, double and any type of pointer. Using enums instead of bools requires more typing but greatly helps to avoid introducing bugs into the program that are extremely difficult to track down. In addition, the use of an enum allows for more than just two values such as is shown for the third value *CONJTRANS* which signifies the complex conjugate.

The fact that ANAs can be designed to handle both real and complex scalar types presents somewhat of a tricking problem for *LinearOp* implementations that only support real scalars (i.e. double). The problem is that the transpose argument value of *CONJTRANS* may be passed to a real-only implementation of *LinearOp* from an ANA designed to work properly for complex scalars. Real-valued *LinearOp* implementations should determine if transpose or non-transpose is to be used by checking for *M\_trans == NOTRANS* and then assume that if *M\_trans != NOTRANS* then the transpose should be used. For example, a real-valued implementation of *LinearOp::apply(...)* should look like:

```
void MyLinearOp::apply( const ETransp M_trans, const Vector<double> &x
    ,Vector<double> *y ,const double alpha=1.0 ,const double beta=0.0 ) const
{
    if( M_trans == NOTRANS ) {
        // Apply the non-transposed operator
        ...
    }
    else {
        // Apply the transposed operator
        ...
    }
}
```

This approach automatically makes the enum values of `CONJTRANS` and `TRANS` mean the same thing for real-valued implementations (as they should).

The *MultiVector* version of `LinearOp::apply(...)` has an identical prototype except the *Vector* arguments are replaced with *MultiVector* arguments. The *MultiVector* version has a default implementation based on the *Vector* version as described in Section 5.5.4.

In the above prototype, the scalars  $\alpha$  and  $\beta$  default to 1.0 and 0.0 respectively. Therefore, by leaving the default values, the default operation becomes

$$y = op(M)x$$

which is the same form that is declared in `HCLLinearOperator::apply(...)`. However, the scalars  $\alpha$  and  $\beta$  provide direct calls to BLAS functions and remove the need to create temporaries when performing long operations (see Section 5.5.5). For example, consider the following long expression

$$y = Au + \gamma B^T v + \eta Cw$$

where  $A$ ,  $B$  and  $C$  are *LinearOp* objects and  $y$ ,  $u$ ,  $v$  and  $w$  are *Vector* objects. Using `TSFCore`, this long operation can be performed as follows

```
template<class Scalar>
void TSFCore::long_expression(
    const LinearOp<Scalar>& A, const Vector<Scalar>& u
    ,Scalar gamma, const LinearOp<Scalar>& B, const Vector<Scalar>& v
    ,Scalar eta, const LinearOp<Scalar>& C, const Vector<Scalar>& w
    ,Vector<Scalar>* y
)
{
    A.apply(NOTRANS,u,y); // y = A*u
    B.apply(TRANS,v,y,gamma,1.0,1.0); // y += gamma*B'*v
    C.apply(NOTRANS,w,y,eta,1.0,1.0); // y += eta*C*w
}

```

where no temporary vectors are required. Note that if the arguments  $\alpha=1.0$  and  $\beta=0.0$  where fixed (as they are in HCL for instance), the above operation would have to be implemented as:

```
template<class Scalar>
void TSFCore::bad_long_expression(
    const LinearOp<Scalar>& A, const Vector<Scalar>& u
    ,Scalar gamma, const LinearOp<Scalar>& B, const Vector<Scalar>& v
    ,Scalar eta, const LinearOp<Scalar>& C, const Vector<Scalar>& w
    ,Vector<Scalar>* y
)
{
    Teuchos::RefCountPtr<Vector<Scalar> >
        t = A.range()->createMember(); // Create a temporary to store the intermediate products
    A.apply(NOTRANS,u,y); // y = A*u
    B.apply(TRANS,v,t.get()); // t = B'*v
}

```

```

axy( $\gamma$ ,*t,y);           // y +=  $\gamma$ *t
C.apply(NOTRANS,w,t.get()); // t = C*w
axy( $\eta$ ,*t,y);           // y +=  $\eta$ *t
}

```

Not only is the function `bad_long_expression(...)` slightly less efficient than `long_expression(...)` but it is also longer and more difficult to write. The arguments `alpha` and `beta` are important to achieve a near-optimal implementation and for ease of use.

Note that some implementations of *LinearOp* may not be able to apply the operator with a value of  $\beta \neq 0$  without creating at least one temporary vector (or multi-vector). However, this is a minor performance issue in most use cases and if  $\beta \neq 0$  is required then a temporary will have to be created by someone anyway.

## 5.4.2 Optional support for adjoints

The *LinearOp* interface only optionally supports transposed (adjoint) matrix-vector multiplications. If the function `opSupported(M_trans)` returns `false`, then the argument `M_trans`, when passed to `apply(...)`, will result in an `OpNotSupported` exception being thrown. This specification, while not ideal from an object-orientation purest point of view, does satisfy the basic principles outlined in Section 7.

## 5.5 MultiVector

While the concepts of a *VectorSpace* and *Vector* are well established, the concept of a multi-vector is fairly new. The idea of a multi-vector was motivated by the library Epetra [25] which contains mostly concrete implementations of distributed-memory linear algebra classes using MPI [21]. A key issue is how multi-vectors and vectors relate to each other. In Epetra, the vector class is a specialization of the multi-vector class. This makes sense from an implementation point of view. The Epetra approach takes the view that a vector *is a* type of multi-vector. An arguably more natural view from an abstract mathematical perspective is that multi-vectors are composed out of a set of vectors where each vector represents a column of the multi-vector. This is the view that multi-vectors *have* or *contain* vectors and this is the approach that has been adopted for TSFCore as shown in Figure 2.

Note that a multi-vector is not the same thing as a blocked or product vector. In fact, multi-vectors and product vectors are orthogonal concepts and it is possible to have product multi-vectors. Product vectors and vector spaces are discussed in Sections 5.3.1 and 8.2.

All of the below examples will involve the compact LBFGS implementation described above in Section 5.1. For these examples we will consider interactions with the two principle *Multi-Vector* objects `Y_store` and `S_store` which each have  $m_{\max}$  columns.

### 5.5.1 Accessing columns of a *MultiVector* as *Vector* objects

The columns of a *MultiVector* object can be accessed using the const or non-const *col(j)* functions which return `RefCountPtr<>` objects which points to an abstract *Vector* view of a column. The prototypes for these functions are shown below.

```
namespace TSFCore{
template<class Scalar>
class MultiVector : virtual public LinearOp<Scalar> {
public:
    ...
    virtual Teuchos::RefCountPtr<Vector<Scalar> >      col(const Index j) = 0;
    virtual Teuchos::RefCountPtr<const Vector<Scalar> > col(const Index j) const;
    ...
};
} // namespace TSFCore
```

Actually, the non-const version of *col(...)* is the only pure virtual function in *MultiVector* and therefore the only function that must be overridden in order to create a concrete (but suboptimal) *MultiVector* subclass. All of the other virtual functions in *MultiVector* have default implementations based on this function and *Vector::applyOp(...)*.

The following example function copies the most recent update vectors *s* and *y* into the multi-vectors *S\_store* and *Y\_store* and increments the counter *m* for a compact LBFGS implementation.

```
template<class Scalar>
void TSFCore::update_S_Y( const Vector<Scalar>& s, const Vector<Scalar>& y
                        ,MultiVector<Scalar>* S_store, MultiVector<Scalar>* Y_store, int* m )
{
    const int m_max = S_store->domain()->dim(); // Get the maximum number of updates allowed
    if(*m < m_max) {
        ++(*m); // Increment the number of updates
        assign(S_store->col(*m).get(),s); // Copy in s into S(:,m)
        assign(Y_store->col(*m).get(),y); // Copy in y into Y(:,m)
    }
    else {
        // We must drop the oldest pair (s,y) and copy in the newest pair
        ...
    }
}
```

Note that the *MultiVector* object that *col(...)* is called on is not guaranteed to be updated until the returned *Vector* object is destroyed when the `RefCountPtr<>` object returned from *col(...)* goes out of scope. The use in the above function guarantees that this happens after each call to the *assign(...)* function.

## 5.5.2 *MultiVector* sub-views

In addition to being able to access the columns of a *MultiVector* object one column at a time, a client can also create const and non-const *MultiVector* views of the columns using one of the *subView(...)* functions shown below.

```
namespace TSFCore {
template<class Scalar>
class MultiVector : virtual public LinearOp<Scalar> {
public:
...
virtual Teuchos::RefCountPtr<MultiVector<Scalar> >      subView(const Range1D& col_rng);
virtual Teuchos::RefCountPtr<const MultiVector<Scalar> > subView(const Range1D& col_rng) const;
virtual Teuchos::RefCountPtr<MultiVector<Scalar> >      subView(const int numCols
, const int cols[]);
virtual Teuchos::RefCountPtr<const MultiVector<Scalar> > subView(const int numCols
, const int cols[]) const;
...
};
} // namespace TSFCore
```

The ability to extract a *MultiVector* sub-view of a contiguous set of columns of a *MultiVector* object, which is supported by the first two functions, is required in order to implement certain types of numerical methods. For example, the implementation of the compact LBFGS method described above in Section 5.1 requires this functionality. The following example function shows how the contiguous *subView(...)* function is used in an LBFGS implementation where *MultiVector* storage objects *S\_store* and *Y\_store* are used to create *MultiVector* view objects *S* and *Y* for only the number of updates currently stored. These sub-view objects are used in later example code.

```
template<class Scalar>
Teuchos::RefCountPtr<const TSFCore::MultiVector<Scalar> >
TSFCore::get_updated( const MultiVector<Scalar>& Store, int m )
{
return Store.subView(Range1D(1,m));
}
```

The second form of the *subView(...)* function takes a list of (possibly unsorted but unique) column indexes *cols[]* and returns a *MultiVector* view object of those columns. This functionality is very useful in the development of some types of ANAs (e.g. block Krylov iterative linear equation solvers).

Note that both forms of the *subView(...)* function have (suboptimal) default implementations based on the *MultiVectorCols* utility subclass. This *MultiVectorCols* class, coincidentally, is also used to provide a general (but suboptimal) implementation of *MultiVector* just given an implementation of *Vector*. This utility subclass is also used to provide default implementations for many of the *MultiVector*-related functions which includes the default implementation of the *VectorSpace::createMembers(numMembers)* function.

### 5.5.3 *MultiVector* support for *applyOp(...)*

RTOp operators can be applied to the columns of a *MultiVector* object one column at a time using the *col(...)* function. However, a potentially more efficient approach is to allow the *MultiVector* object to apply the RTOp operator itself. This is supported by the *applyOp(...)* functions on *MultiVector*. The *applyOp(...)* functions are not called directly (they are protected) but instead are called by non-member (friend) functions *applyOp(...)* which then invoke the member functions. This approach allows a more natural way to invoke a reduction/transformation operation in line with the mathematical description in [10].

There are two versions of *MultiVector::applyOp(...)*: one that returns a list of reduction objects (one for each column of the multi-vector) and another that uses two RTOp operators to reduce all of the reduction objects over each column into single reduction object which is returned. Both versions of the *MultiVector::applyOp(...)* have default implementations that are based on *MultiVector::col(...)* and *Vector::applyOp(...)*.

Below, two example operations, which are defined in the header *TSFCoreMultiVectorStdOps.hpp*, are shown that are needed by various ANAs.

The first example is the update operator  $\alpha U + V \rightarrow V$  and is implemented in the following function.

```
template<class Scalar>
void TSFCore::update( Scalar alpha, const MultiVector<Scalar>& U, MultiVector<Scalar>* V )
{
    TEST_FOR_EXCEPTION(V==NULL, std::logic_error, "axpy(...), Error!"); // Validate input
    RTOpPack::TOpAxy<Scalar> axpy_op(alpha); // Create (init) op object
    const MultiVector<Scalar>* multi_vecs[] = { &U }; // Set up non-mutable mv args
    MultiVector<Scalar>* targ_multi_vecs[] = { V }; // Set up mutable mv args
    applyOp<Scalar>(axpy_op, 1, multi_vecs, 1, targ_multi_vecs, NULL); // Invoke the transformation operator
}
```

In the above call to *applyOp(...)*, a NULL pointer is passed in for the array of reduction objects which is allowed since this RTOp operator does not perform a reduction.

The second example is a column-wise dot product operation and is implemented in the following function.

```
template<class Scalar>
void TSFCore::dot( const MultiVector<Scalar>& V1, const MultiVector<Scalar>& V2, Scalar dot[] )
{
    const int m = V1.domain()->dim(); // Get the num cols
    RTOpPack::ROpDot<Scalar> dot_op; // Create op object
    std::vector<Teuchos::RefCountPtr<RTOpPack::ReductTarget> >
    rcp_dot_targs(m); // Array of reduct objects
    std::vector<RTOpPack::ReductTarget*>
    dot_targs(m); // Array of reduct object ptrs
    for( int kc = 0; kc < m; ++kc ) { // For each column:
```

```

    rcp_dot_targs[kc] = dot_op.reduct_obj_create();           // Create reduct object
    dot_targs[kc] = &rcp_dot_targs[kc];                     // Set raw pointer
}
const MultiVector<Scalar>* multi_vecs[] = { &V1, &V2 };   // Set up non-mutable mv args
applyOp(dot_op,2,multi_vecs,0,NULL,&dot_targs[0]);        // Invoke the reduction operator
for( int kc = 0; kc < m; ++kc ) {                          // For each column:
    dot[kc] = dot_op(*dot_targs[kc]);                       // Extract dot product val
}
}
}

```

Note that the above reduction operation will be performed with a single global reduction when performed on a distributed-memory parallel computer (using MPI). Without the concept of a *MultiVector* or support for the *applyOp( . . . )* function, this type of multi-vector reduction operation would require  $m$  separate global reductions, where  $m$  is the number of columns in the multi-vector. The presence of this function is critical for a near-optimal implementation with respect to minimizing communication in a distributed memory program.

### 5.5.4 *Vector* and *MultiVector* correspondence

The interface class *LinearOp* takes the perspective that most subclasses will naturally prefer to implement the *Vector* version of the function *apply( . . . )* and let the default implementation of the *MultiVector* version of this function deal with *MultiVector* objects. There are many cases where there is no way to provide more specialized implementations of these operations for multi-vectors. For example, while the BLAS and LAPACK are designed from the ground up to be more efficient with multiple right-hand-side vectors, many implementations of sparse direct linear solvers unfortunately only support the solution of single linear systems (e.g. the Harwell solvers such as MA47 and MA48 [43]). This realization provides the motivation for choosing the *Vector* versions of these functions as the default functions for subclasses to override. With that said, if a *LinearOp* subclass can provide an optimized implementation of the *MultiVector* version of the *apply( . . . )* function, does such a subclass also have to provide a completely independent implementation of the *Vector* version of this function? The answer is no since *Vector* is derived from *MultiVector*. Therefore, providing a vector implementation of the *apply( . . . )* function is trivial and looks like:

```

namespace TSFCore {
template<class Scalar>
class MyLinearOp : public LinearOp<Scalar> {
public:
    ...
    void apply( ETransp M_trans, const MultiVector<Scalar> &X, MultiVector<Scalar> *Y, Scalar alpha
               ,Scalar beta ) const
    {
        // Optimized implementation for multi-vectors
        ...
    }
    void apply( ETransp M_trans, const Vector<Scalar> &x, Vector<Scalar> *y, Scalar alpha
               ,Scalar beta ) const

```

```

{
  // Call MultiVector version
  apply(alpha,M_trans,static_cast<const MultiVector<Scalar>&>(x),static_cast<MultiVector<Scalar>*>(y),beta);
}
...
};
} // namespace TSFCore

```

Above, the use of `static_cast<>` to `MultiVector` is necessary to insure that the `MultiVector` version of `apply(...)` is called and not the `Vector` version (which it would be otherwise and would setup an infinite call recursion).

### 5.5.5 *MultiVector* acting as a *LinearOp*

The last issues to discuss with regard to *MultiVector* relate to where it fits in the class hierarchy. The decision adopted for TSFCore was to make *MultiVector* specialize *LinearOp*. In other words, a *MultiVector* object can also act as a *LinearOp* object.

As an example where this is needed, consider using the LBFGS inverse matrix  $H$  shown in Figure 3 as a linear operator which acts on multi-vector arguments  $U \in \mathbf{R}^{n \times p}$  and  $V \in \mathbf{R}^{n \times p}$  in an operation of the form

$$\begin{aligned}
U &= \alpha B^{-1}V \\
&= \alpha HV \\
&= \alpha gV + \alpha \begin{bmatrix} S & \gamma Y \end{bmatrix} \begin{bmatrix} Q_{ss} & Q_{sy} \\ Q_{sy}^T & Q_{yy} \end{bmatrix} \begin{bmatrix} S^T \\ \gamma Y^T \end{bmatrix} V
\end{aligned}$$

where the matrices  $Q_{ss}$ ,  $Q_{ys}$  and  $Q_{yy}$  are stored as small *MultiVector* objects. A multi-vector solve using the inverse  $H = B^{-1}$  might be used, for instance, in an active-set optimization algorithm where  $V \in \mathbf{R}^{n \times p}$  represents the  $p$  gradient vectors of the active constraints. This is an important operation, for instance, in the formation of a Schur complement of the KKT system in the QP subproblem of an reduced-space SQP method [5]. This multi-vector operation using  $H$  can be performed with the following atomic operations

$$\begin{aligned}
T_1 &= S^T V \\
T_2 &= Y^T V \\
T_3 &= Q_{ss} T_1 + \gamma Q_{sy} T_2 \\
T_4 &= Q_{sy}^T T_1 + \gamma Q_{yy} T_2 \\
U &= \alpha \gamma V + \alpha S T_3 + \alpha \gamma Y T_4
\end{aligned}$$

where  $T_1, T_2, T_3$  and  $T_4$  are all temporary *MultiVector* objects of dimension  $m \times p$ . The following function shows how the above operations are performed in order to implement the overall multi-vector operation.

```

template<class Scalar>
void TSFCore::LBFGS_solve(
    int m, Scalar g, const MultiVector<Scalar>& S_store, const MultiVector<Scalar>& Y_store
    ,const MultiVector<Scalar>& Q_ss, const MultiVector<Scalar>& Q_sy, const MultiVector<Scalar>& Q_yy
    ,const MultiVector<Scalar>& V, MultiVector<Scalar>* U, Scalar alpha = 1.0, Scalar beta = 0.0
)
{
    // validate input
    ...
    const int p = V.domain()->dim(); // Get number of columns in V and U
    Teuchos::RefCountPtr<const MultiVector<Scalar> >
        S = get_updated(S_store,m), // Get view of only stored columns in S_store
        Y = get_updated(Y_store,m); // Get view of only stored columns in Y_store
    Teuchos::RefCountPtr<MultiVector<Scalar> >
        T_1 = S->domain()->createMembers(p), // Create the temporary multi-vectors
        T_2 = Y->domain()->createMembers(p), // ...
        T_3 = S->domain()->createMembers(p), // ...
        T_4 = Y->domain()->createMembers(p); // ...
    S->apply(TRANS,V,T_1->get()); // T_1 = S'*V
    Y->apply(TRANS,V,T_2->get()); // T_2 = Y'*V
    Q_ss.apply(NOTRANS,*T_1,T_3->get()); // T_3 = Q_ss*T_1
    Q_sy.apply(NOTRANS,*T_2,T_3->get(),gamma,1.0); // T_3 += gamma*Q_sy*T_2
    Q_sy.apply(TRANS,*T_1,T_4->get()); // T_4 = Q_sy'*T_1
    Q_yy.apply(NOTRANS,*T_2,T_4->get(),gamma,1.0); // T_4 += gamma*Q_yy*T_2
    S->apply(NOTRANS,*T_3,U,alpha); // U = alpha*S*T_3
    Y->apply(NOTRANS,*T_4,U,alpha*gamma,1.0); // U += alpha*gamma*Y*T_4
    axpy(alpha*g,V,U); // U += alpha*g*V
}

```

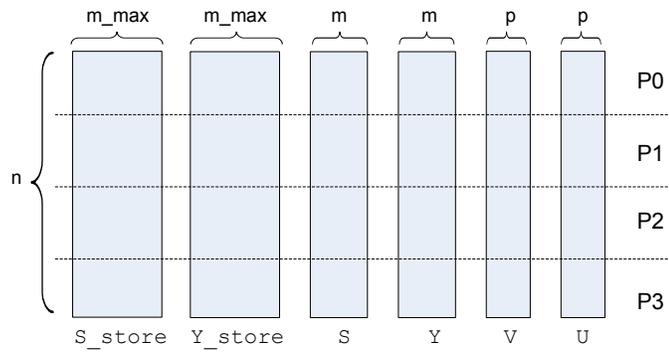
Consider the use of the above function in an SPMD environment where the ANA runs in duplicate and in parallel on each processor. Here, the elements for the multi-vector objects *S\_store* (and view *S*), *Y\_store* (and view *Y*), *V* and *U* are distributed across many different processors as Figure 5 shows. The case shown<sup>4</sup> in Figure 5 is for the situation where  $p < m$ . In SPMD mode, all of the elements in the multi-vector objects *Q\_ss*, *Q\_sy*, *Q\_yy*, *T\_1*, *T\_2*, *T\_3* and *T\_4* are stored locally and in duplicate (i.e. locally replicated) on each processor as shown<sup>5</sup> in Figure 5.b. Now let us consider the performance of this set of operations in this context. Note that there are principally three different types of operations with multi-vectors that are performed through the *MultiVector::apply(...)* function.

The first type of operation performed by *MultiVector::apply(...)* is the parallel/parallel matrix-matrix products performed in the lines

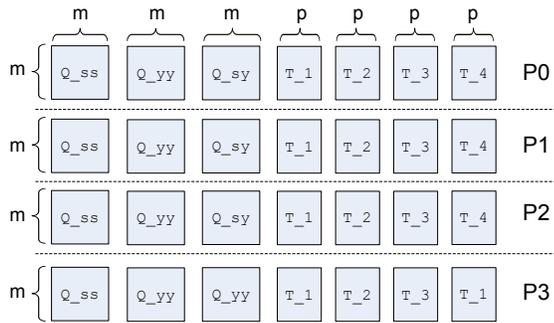
---

<sup>4</sup>The aspect ratio of the number of rows to number of columns in Figure 5 is exaggerated in that in a realistic case the number of rows usually numbers in the tens to hundreds of thousands while the number of columns usually number in only the tens. This was done for illustrative purposes. If the true aspect ratio were shown in Figure 5 then all of these multi-vectors would appear to be just vertical lines and would not show a distinction between different multi-vectors.

<sup>5</sup>The same unrealistic aspect ratio shown in Figure 5 is the same as shown in Figure 5.a, again for illustrative purposes.

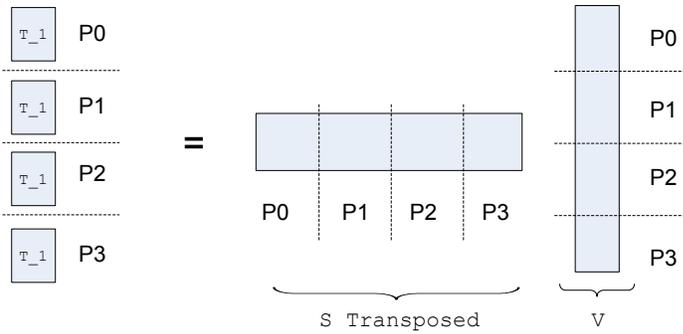


a) Disturbed-memory multi-vectors



b) Locally replicated multi-vectors

**Figure 5.** Carton of disturbed and locally replicated multi-vectors which are used in the example compact LBFGS sub-ANA when run in SPMD mode on four processors. The process boundaries are shown as dotted lines. The numbers for rows and columns of each multi-vector are also shown.



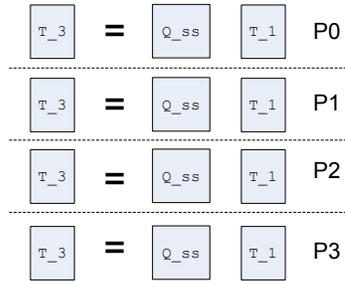
**Figure 6.** Carton of distributed-memory matrix-matrix product (i.e. block dot product)  $T_1 = S^T V$  run in SPMD mode on four processors. This operation first performs local matrix-matrix multiplication with the entries of  $S^T$  and  $V$  on each processor using level-3 BLAS and then a global reduction summation operation is performed (using MPI) to produce  $T_1$  which is returned to all of the processors.

```
S->apply(TRANS,V,T_1->get());
Y->apply(TRANS,V,T_2->get());
```

where the results are stored in the local multi-vectors  $T_1$  and  $T_2$ . The operation  $T_1 = S^T V$  is shown in Figure 6 for the SPMD mode on four processors. This type of operation is also known as a block dot product [???]. These two operations only require a single global reduction each, independent of the number of updates  $m$  represented in  $S$  and  $Y$  or columns  $p$  in  $V$ . Note that if there was no concept of a multi-vector and these matrix-matrix products had to be performed one set of vectors at a time, then these two parallel matrix-matrix products would require a whopping  $2mp$  global reductions. For  $m = 40$  and  $p = 20$  this would result in  $2mp = 2(40)(20) = 1600$  global reductions! Clearly this many global reductions would destroy the parallel scalability of the overall ANA any many cases. It is in this type of operation that the concept of a *MultiVector* is most critical for near-optimal performance in parallel programs. In addition to minimizing communication overhead, the *MultiVector* implementation can utilize level-3 BLAS to perform the local processor matrix-matrix multiplications yielding near-optimal cache performance on most systems.

The second type of operation performed by *MultiVector::apply(...)* is the local/local matrix-matrix products of small local *MultiVector* objects in the lines

```
Q_ss.apply(NOTRANS,*T_1,T_3->get());
Q_sy.apply(NOTRANS,*T_2,T_3->get(),g,1.0);
Q_sy.apply(TRANS,*T_1,T_4->get());
Q_yy.apply(NOTRANS,*T_2,T_4->get(),g,1.0);
```



**Figure 7.** Carton of local matrix-matrix product  $T_3 = Q_{ss}T_1$  involving locally replicated multi-vectors run in SPMD mode on four processors. In SPMD mode this operation involves no processor-to-processor communication at all.

where the operation  $T_3 = Q_{ss}T_1$  is shown, for example, in Figure 7. Note that these types of local computations classify as serial overhead and therefore it is critical that the cost of these operations be kept to a minimum or they could cripple the parallel scalability of the overall ANA. Each of these four matrix-matrix multiplications involve only one virtual function call and the matrix-matrix multiplication itself can be performed with level-3 BLAS, achieving the fastest possible flop rate attainable on most processors [19].

The third type of operation performed by `MultiVector::apply(...)` is local/parallel matrix-matrix multiplications performed in the lines

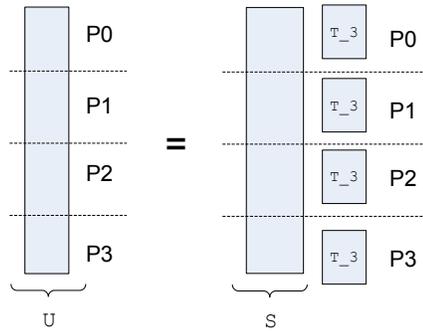
```
S->apply(NOTRANS,*T_3,U,alpha);
Y->apply(NOTRANS,*T_4,U,alpha*g,1.0);
```

where the operation  $U = ST_3$  is shown, for example, in Figure 8. This type of operation involves fully scalable work with no communication or synchronization required. Here, a vector-by-vector implementation will not be a bottleneck from a standpoint of global communication. However, this operation will utilize level-3 BLAS and yield near-optimal local cache performance where a vector-by-vector implementation would not.

The last type of operation performed in the above `LBFGS_solve(...)` function does not involve `MultiVector::apply(...)` and is shown in the line

```
axpy(alpha*g,V,U);
```

The implementation of this function uses an RTOp transformation operator with the `Multi-`



**Figure 8.** Carton of distributed-local matrix-matrix product  $U = ST_3$  involving both distributed and locally multi-vectors run in SPMD mode on four processors. To perform this operation, the local elements in the distributed multi-vector  $S$  are multiplied with the locally replicated multi-vector  $T_3$  and the result of the matrix-matrix product are set to the local elements of the distributed multi-vector  $U$ . In SPMD mode this operation requires no processor-to-processor communication at all.

`Vector::applyOp(...)` function. Note that this function only involves transformation operations (i.e. no communication) which are fully scalable.

### 5.5.6 Explicit access to *MultiVector* elements

Certain use cases require explicit access to the elements in *MultiVector* objects is required just as with *Vector* objects (see Section 5.3.2). The operations on *MultiVector* that allow explicit access to elements are shown below.

```
namespace TSFCore {
template<class Scalar>
class MultiVector : virtual public LinearOp<Scalar> {
public:
    ...
    virtual void getSubMultiVector( const Range1D &rowRng, const Range1D &colRng
        ,RTOPack::SubMultiVectorT<Scalar> *sub_mv ) const;
    virtual void freeSubMultiVector( RTOPack::SubMultiVectorT<Scalar>* sub_mv ) const;
    virtual void getSubMultiVector( const Range1D &rowRng, const Range1D &colRng
        ,RTOPack::MutableSubMultiVectorT<Scalar> *sub_mv );
    virtual void commitSubMultiVector( RTOPack::MutableSubMultiVectorT<Scalar>* sub_mv );
    ...
};
} // namespace TSFCore
```

The above *MultiVector* functions support two of the same use cases as the corresponding *Vector* function: (a) extracting a non-mutable view of a contiguous sub-matrix; and (b) extracting a mutable view of a contiguous sub-matrix and then committing the changes back again. Note that the third use case, explicitly setting the elements (without creating a view first), which is supported by *Vector*, is not supported by *MultiVector*.

The main difference between the explicit element access functions in *MultiVector* and *Vector* is that the *RTOp* classes *RTOpPack::SubMultiVectorT<>* and *RTOpPack::MutableSubMultiVectorT<>* are used instead of *RTOpPack::SubVectorT<>* and *RTOpPack::MutableSubVectorT<>*. The classes *RTOpPack::SubMultiVectorT<>* and *RTOpPack::MutableSubMultiVectorT<>* are used to represent column-oriented Fortran (i.e. BLAS) style dense rectangular matrices. Such a dense matrix *A* is represented by a pointer to the first element in the first column `Scalar *A`, the number of rows `Index A_nrows`, the number of columns `Index A_ncols` and the leading dimension between consecutive columns `A_ld` (see BLAS documentation).

The first use case (a) where a const view of sub-matrix is created and used requires calling the const functions `getSubMultiVector(...)` and `freeSubMultiVector(...)` respectively. The following example function shows how a sub-matrix of an input *MultiVector* *Y* can be copied in a column-wise Fortran(BLAS)-style matrix *A* in one atomic operation.

```
template<class Scalar>
void foo5(
    const MultiVector<Scalar> &Y, const Index firstRowOff, const Index firstColOff
    ,const Index A_nrows, const Index A_ncols, const Index A_ld, Scalar A[]
)
{
    RTOpPack::SubMultiVectorT<Scalar> sub_mv;           // Create (init) submultivector view object
    Y.getSubMultiVector(                               // Initialize the view object
        Range1D(firstRowOff+1,firstRowOff+A_nrows)
        Range1D(firstColOff+1,firstColOff+A_ncols)
        ,&sub_mv );
    for( Index j=0; j < sub_mv.numSubCols(); ++j ) { // Loop over columns
        for( Index i=0; i < sub_mv.subDim(); ++i ) { // Loop over elements in a column
            A[i+j*A_ld]                               // Copy elements
                =sub_mv.values()[i+j*sub_mv.leadingDim()];
        }
    }
    Y.freeSubMultiVector(&sub_mv);                     // Free the view of Y
}
```

In the above function, we could have used the overloaded operator function *RTOpPack::SubMultiVectorT<>::operator()(Index i, Index j)* to access the elements in the sub-view as `sub_mv(i+1, j+1)` (where the addition +1 is needed to convert from zero-based to one-based) but the above function shows how raw memory access is achieved. Note that the above function should only be expected to be efficient if `Y.range()->isInCore()==true` but should work in any case.

The second use case (b) involving the utilization of a non-const sub-matrix view of a *MultiVector* object requires the use of the non-const functions `getSubMultiVector(...)` and `commitSubMultiVector(...)`. The following function demonstrates this use case by showing how to add

elements in an input dense Fortran(BLAS) style matrix A to a sub-matrix in a *MultiVector* object Y.

```

template<class Scalar>
void foo6(
const Scalar A[], const Index A_nrows, const Index A_ncols, const Index A_ld
, const Index firstRowOff, const Index firstColOff, MultiVector<Scalar> *Y
)
{
  RTopPack::MutableSubMultiVectorT<Scalar> sub_mv; // Create (init) submultivector view object
  Y->getSubMultiVector( // Initialize the view object
    RangeD(firstRowOff+1,firstRowOff+A_nrows)
    RangeD(firstColOff+1,firstColOff+A_ncols)
    ,&sub_mv );
  for( Index j=0; j < sub_mv.numSubCols(); ++j ) { // Loop over columns
    for( Index i=0; i < sub_mv.subDim(); ++i ) { // Loop over elements in a column
      sub_mv.values()[i+j*sub_mv.leadingDim()] // Add to the elements
        += A[i+j*A_ld];
    }
  }
  Y->commitSubMultiVector(&sub_mv); // Commit changes back to Y
}

```

Note that in the above function that the underlying *MultiVector* object Y is not guaranteed to be modified until the function

```
Y->commitSubMultiVector(&sub_mv);
```

is called. Of course, no functions on the *MultiVector* object Y should be performed (except perhaps to access its *range()* and *domain()* spaces) should be called between the time that a non-const view is extracted and when it is committed back with a call to *commitSubMultiVector(...)*.

The types of explicit element access to multi-vectors described above is critical in block GMRES algorithms and in a compact limited memory sub-ANA first introduced in Section 5.1 as well as in a number of other ANAs.

## 5.6 Aliasing of *Vector* and *MultiVector* arguments

It has not been stated specifically yet but in all *Vector*, *MultiVector* and *LinearOp* functions where a *Vector* or *MultiVector* object may be modified, it is strictly forbidden for any of the mutable objects to alias any of the other objects of the same type in the same function. For example, code like the following is strictly forbidden.

```

template<class Scalar>
void foo3( const LinearOp& M, ETransp M_trans, Vector<Scalar>* x )

```

```

{
  M.apply(M_trans,*x,x); // Error!!!!!!!!!!!!
}

```

Note that typically the above function would not even get to the numerics (where it would most likely compute the wrong results) because `M.range()->isCompatible(*M.domain())==false` in general. Instead, this operation must be implemented as follows.

```

template<class Scalar>
void foo4( const LinearOp& M, ETransp M_trans, Vector<Scalar>* x )
{
  Teuchos::RefCountPtr<Vector<Scalar> > x_tmp = x->clone(); // Create a copy
  M.apply(M_trans,*x_tmp,x); // Okay!
}

```

Allowing client code to pass in aliased arguments would greatly complicate the implementation of most `RTOp`, `MultiVector` and `LinearOp` subclasses and would introduce the possibility of many different types of bugs that would be extremely difficult to track down. This is an issue that is usually not well defined in most linear algebra interfaces but it is a very important issue. Allowing ANA developers to alias objects in these functions does not provide any new functionality and is considered to be only nonessential but convenient functionality and is therefore not included in TSFCore. In general, it is not possible to determine, from the abstract interfaces for the objects themselves, if objects alias each other. To perform this type of test would require special functions be added to the `Vector` and `MultiVector` interfaces and implementing these test functions would complicate the development of these types of subclasses greatly.

Note that aliasing of input data with output data is not strictly forbidden, and is allowed as long as this is built into the operation. For example, in the `LinearOp::apply(...)` function, the vector `y` both supplies data for the operation (if  $\beta \neq 0$ ) and stores the output for the operation as shown in (1). The same applies to several of the `RTOp`-based vector operations shown in Figure 4 (i.e. `Vp_S(...)`, `Vt_S(...)`, `Vp_S(...)`, `Vp_StV(...)` and `ele_wise_prod(...)`). Allowing vectors and multi-vectors to both supply data for an operation and store output from an operation is fine as long as the operation has been specifically designed to handle this as the above mentioned operations have.

In summary, do not alias output arguments with each other or with other input arguments in any of the TSFCore interface functions.

## 6 An Example Abstract Numerical Algorithm : An Iterative Linear Solver

In this section we describe how TSFCore can be directly used to build ANAs and while this is not the primary role TSFCore is designed for, this example shows that TSFCore provides all of the

needed functionality for near-optimally performing implementations. Code for a partial ANA in the form of a compact LBFGS method was described in Section 5.1. In this section, we will describe the implementation of a simple block BiCG [4] method. BiCG was chosen for this example because it requires adjoints and is fairly simple. Other types of block iterative linear solvers such as methods as CG, BiCGStab, GMRES and QMR [4] can be implemented in a similar manner.

The subclass `BiCGSolver` implements a simple block BiCG method. A listing for a single-vector version of the BiCG method is shown in Figure 9. This listing is identical to the listing in [4] except for the substitutions  $A = op(M)$ ,  $M = op(\tilde{M})$  and  $b = ay$  (where  $a$  is a scalar multiplier). The multi-vector version, as implemented using `TSFCore` in code, follows in a straightforward manner. This implementation does not take advantage of any potential linear dependence in the right-hand-side vectors in an attempt to accelerate the method such as is described in [???]. Such an enhanced multi-vector version could be implemented in a similar manner.

Figure 10 shows a partial listing for the `BiCGSolver::doIteration(...)` function (which implements a single iteration of the BiCG method) as implemented in the file `TSFCoreSolvers-BiCGSolver.hpp`. All of the functions called in the C++ code shown in Figure 10 have already been described except for the non-member functions `assign(...)` (lines 292, 293, 310 and 311) and `update(...)` (lines 315, 316 and 340–342) which are defined in the header `TSFCore-MultiVectorStdOps.hpp`. There are two assignment functions `assign(...)`: one that assigns a `MultiVector` object to a `Scalar`, and another that assigns one `MultiVector` object to another. Both of these functions are implemented through `MultiVector::applyOp(...)` and use already-defined `RTOp` operators. The two versions of the `update(...)` function used in this code, however, can not use `MultiVector::applyOp(...)` and instead are implemented column-by-column as, for instance

$$(\alpha_{(j)}\beta)U_{(:,j)} + V_{(:,j)} \rightarrow V_{(:,j)}, \text{ for } j = 1 \dots m$$

in the function

```
template<class Scalar>
void TSFCore::update( Scalar alpha[], Scalar beta, const MultiVector<Scalar>& U, MultiVector<Scalar>* V )
{
    ...
    const int m = U.domain()->dim();
    for( int j = 1; j <= m; ++j )
        Vp_StV( V->col(j).get(), alpha[j-1]*beta, *U.col(j) );
}
```

where the `Vp_StV(...)` function is the `axpy` operation for vectors and is declared in the header `TSFCoreVectorStdOpsDecl.hpp`. Note that when running the above BiCG method in an SPMD configuration (where the ANA runs in parallel and in duplicate in each process) this implementation of `update(...)` does not involve any communication or require any synchronization and therefore will not affect the performance of the algorithm for a communication point of view. However, when running in a master-slave configuration (where the ANA runs on the master and the linear algebra

```

Compute  $r^{(0)} = ay - op(M)x^{(0)}$  for the initial guess  $x^{(0)}$ .
Choose  $\tilde{r}^{(0)}$  (for example,  $\tilde{r}^{(0)} = \text{randomize}(-1, +1)$ ).
for  $i = 1, 2, \dots$ 
    solve  $op(\tilde{M})z^{(i-1)} = r^{(i-1)}$ 
    solve  $op(\tilde{M})^T \tilde{z}^{(i-1)} = \tilde{r}^{(i-1)}$ 
     $\rho_{i-1} = z^{(i-1)T} \tilde{r}^{(i-1)}$ 
    if  $\rho_{i-1} = 0$ , method fails
    if  $i = 1$ 
         $p^{(i)} = z^{(i-1)}$ 
         $\tilde{p}^{(i)} = \tilde{z}^{(i-1)}$ 
    else
         $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$ 
         $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$ 
         $\tilde{p}^{(i)} = \tilde{z}^{(i-1)} + \beta_{i-1} \tilde{p}^{(i-1)}$ 
    endif
     $q^{(i)} = op(M)p^{(i)}$ 
     $\tilde{q}^{(i)} = op(M)^T \tilde{p}^{(i)}$ 
     $\gamma_i = \tilde{p}^{(i)T} q^{(i)}$ 
     $\alpha_i = \rho_{i-1} / \gamma_i$ 
     $x^{(i)} = x^{(i-1)} + \alpha_{i-1} p^{(i)}$ 
     $r^{(i)} = r^{(i-1)} - \alpha_{i-1} q^{(i)}$ 
     $\tilde{r}^{(i)} = \tilde{r}^{(i-1)} - \alpha_{i-1} \tilde{q}^{(i)}$ 
    check convergence; continue if necessary
end

```

**Figure 9.** A single-vector version of the preconditioned bi-conjugate gradient method (BiCG).

```

00273 template<class Scalar>
00274 void BiCGSolver<Scalar>::doIteration(
00275     const LinearOp<Scalar> &M, ETransp opM_notrans, ETransp opM_trans, MultiVector<Scalar> *X, Scalar a
00276     ,const LinearOp<Scalar> *M_tilde_inv, ETransp opM_tilde_inv_notrans, ETransp opM_tilde_inv_trans
00277 ) const
00278 {
00285     const Index m = currNumSystems_;
00286     int j;
00287     if( M_tilde_inv ) {
00288         M_tilde_inv->apply( opM_tilde_inv_notrans, *R_, Z_.get() );
00289         M_tilde_inv->apply( opM_tilde_inv_trans, *R_tilde_, Z_tilde_.get() );
00290     }
00291     else {
00292         assign( Z_.get(), *R_ );
00293         assign( Z_tilde_.get(), *R_tilde_ );
00294     }
00299     dot( *Z_, *R_tilde_, &rho_[0] );
00303     for(j=0;j<m;++j) {
00304         TEST_FOR_EXCEPTION(
00305             rho_[j] == 0.0, Exceptions::SolverBreakdown
00306             , "BiCGSolver<Scalar>::solve(...): Error, rho["<<j<<"] = 0.0, the method has failed!"
00307             );
00308     }
00309     if( currIteration_ == 1 ) {
00310         assign( P_.get(), *Z_ );
00311         assign( P_tilde_.get(), *Z_tilde_ );
00312     }
00313     else {
00314         for(j=0;j<m;++j) beta_[j] = rho_[j]/rho_old_[j];
00315         update( *Z_, &beta_[0], 1.0, P_.get() );
00316         update( *Z_tilde_, &beta_[0], 1.0, P_tilde_.get() );
00317     }
00322     M.apply(opM_notrans, *P_, Q_.get() );
00323     M.apply(opM_trans, *P_tilde_, Q_tilde_.get() );
00328     dot( *P_tilde_, *Q_, &gamma_[0] );
00329     for(j=0;j<m;++j) alpha_[j] = rho_[j]/gamma_[j];
00334     for(j=0;j<m;++j) {
00335         TEST_FOR_EXCEPTION(
00336             alpha_[j] == 0.0 || RTOp_is_nan_inf(alpha_[j]), Exceptions::SolverBreakdown
00337             , "BiCGSolver<Scalar>::solve(...): Error, rho["<<j<<"] = 0.0, the method has failed!"
00338             );
00339     }
00340     update( &alpha_[0], +1.0, *P_, X );
00341     update( &alpha_[0], -1.0, *Q_, R_.get() );
00342     update( &alpha_[0], -1.0, *Q_tilde_, R_tilde_.get() );
00348 }

```

**Figure 10.** Implementation of an iteration of a multi-vector version of BiCG.

runs in the  $N_p$  slave process) every function invocation of a function on a nonlocal TSFCore object involves communication, including each call to `MultiVector::col(j)`. While the number of function invocations on TSFCore objects for all of the other operations shown in Figure 10 are independent of the number of right-hand-sides  $m$ , this is not true for the above implementation of the `update(...)` function. However, from a local cache performance point of view, note that this is a level-1 BLAS operation so there is no real performance motivation for providing a multi-vector version.

The reason that this operation is performed column-by-column is that it is not well supported by the functions `MultiVector::applyOp(...)` or `MultiVector::apply(...)`. The problem is that in the current design of RTOp and `MultiVector::applyOp(...)`, an RTOp operator object does not have any way to distinguish between different columns of a multi-vector in order to apply different values of  $\alpha_{(j)}$  for each column  $j$ . To allow this would require changing the design of RTOp to deal with multi-vectors directly instead of just individual vectors.

This operation could be implemented with the `MultiVector::apply(...)` function using a `MultiVector` object

$$A = \begin{bmatrix} \alpha_{(1)}\beta & & & \\ & \alpha_{(2)}\beta & & \\ & & \ddots & \\ & & & \alpha_{(m)}\beta \end{bmatrix}$$

and then performing

$$UA + V \rightarrow V.$$

But, since it would generally be assumed that the local multi-vector  $A$  is dense, this would likely cost  $O(nm^2)$  flops instead of the  $O(nm)$  flops of the actual update operation (where  $n$  is the global number of unknowns in each linear system).

To yield a near-optimal implementation in all computing environments, this type of update operation would have to be added directly to the `MultiVector` interface. However, it is not clear that this is justified since iterative linear solvers such as this BiCG method are likely to only run in SPMD mode.

With that said, assuming that the BiCG method shown in Figure 10 is run in SPMD mode, the entire algorithm only involves three global reductions per BiCG iteration – independent of the number of linear systems  $m$  that are being solved. These three global reductions include the two multi-vector dot products on lines 299 and 328 along with a multi-vector norm calculation for the convergence check which is performed in a calling function. The two preconditioner solves on lines 288–289 and the two multi-vector operator applications in lines 322–323 likely involve global communication also, so in general there will be a total of seven parallel synchronizations per BiCG iteration (or only five if no preconditioner is used) — independent of the number of linear systems being solved. Therefore, this implementation allows for near-optimal performance both in terms of

minimizing the number of global synchronizations and in local cache performance (because of the use of block operations with multi-vectors).

## 7 General Object-Oriented Software Design Concepts and Principles

In this section we discuss some of the basic C++ idioms and design patterns that have been used to construct the TSFCore C++ classes. The primary issues relate to modern approaches to general memory management for object-oriented programming in C++ and to object allocation verses initialization. There is also a short discussion of proper object-oriented design principles.

### 7.1 General C++ idioms used by TSFCore

TSFCore uses the following general C++ idioms:

- a) Objects are allocated through “abstract factory” interfaces that return `RefCountPtr`-wrapped objects.
- b) Details of object deallocation are handled through `RefCountPtr`.
- c) Objects can be optionally copied through `clone()` functions.
- d) Concrete subclasses use the “separate construction and initialization” idiom for maximum flexibility and reusability.
- e) Error handling is performed through the C++ exception handling mechanism and all exceptions thrown are derived from `std::exception()` and are thrown using the macro `TEST_FOR_EXCEPTION(...)`.

a) The basic design patterns used for memory management in TSFCore are the “abstract factory” and the “prototype” patterns as described in the well known “gang-of-four” book [23]. When combined with the C++ idiom of smart reference-counted pointers for automatic garbage collection (see [34, Items 28-29]) these design patterns become very powerful and greatly help C++ developers to dodge many of the pitfalls of dynamic memory allocation in C++. The basic memory management infrastructure is defined in a namespace called *Teuchos* which is external to TSFCore. By far the most important class in *Teuchos* (see [6]) is the templated smart reference-counted pointer class `RefCountPtr<>`. This templated class is very close to the templated class `shared_ptr<>` that is provided in the `boost` library [14]. The use of the class `RefCountPtr<>` is described very well in [9] and in the Doxygen documentation so it will not be described here. However, example C++ code that uses this class was shown in the above sections.

b) All memory management issues associated with abstract objects, which include instantiations of all of the classes shown in Figure 2, are handled using `RefCountPtr<>`. In this way, a client never needs to explicitly delete any of these objects. An object will be automatically deleted once all of the `RefCountPtr<>` objects that point to the object go out of scope. The functions `VectorSpace::createMember()` and `VectorSpace::createMembers(...)`, as well as many others that (may) have to allocate new objects, all return pointers to these objects embedded in `RefCountPtr<>` objects. Note that there are many types of C++ client code, such as functions and functions, that simply collaborate with preallocated objects for a short period of time and do not need to assume any responsibilities for memory management. In these cases, the reference or raw pointer to the underlying object can be extracted from the `RefCountPtr<>` object which is then passed on to C++ code that accepts only references or raw pointers. There are several examples of this type of usage in the code examples in the previous sections.

c) The “abstract factory” design pattern (as implemented by `VectorSpace` for instance) enabled with `RefCountPtr<>` effectively relieves clients from having to deal with how objects are created and destroyed but there is another type of memory management task that is also required in some use cases. To describe the problem, suppose that a C++ client has a handle to a `LinearOp` object (either through a smart or raw pointer) and that client wants to copy the object so that some other client will not modify the object before said client is finished with the current `LinearOp` object. This is a classical problem with the use of objects with *reference* (or *pointer*) semantics which does not occur with objects that use *value* semantics [42]. This use case requires the ability to “clone” an object which is the basis of the “prototype” design pattern. Every abstract interface shown in Figure 2 defines some type of `clone()` function which return `RefCountPtr<>` objects pointing to the cloned (or copied) object. In some cases the concrete subclass does not have to override the `clone()` function in order to achieve this functionality (i.e. `Vector` and `MultiVector`) while in other cases it does (i.e. `LinearOp`). In cases where a meaningful default implementation for the `clone()` function can not be provided, a default implementation returning a null `RefCountPtr<>` object is provided. The implication of this approach is that while the `clone()` function is a useful feature, it is considered an optional feature where subclasses are not required to provide an implementation. However, every good subclass implementation should provide an implementation of the `clone()` function since it makes the work of the client much easier in some use cases.

d) Another set of issues that are related to the memory management issues described above are issues concerning object allocation versus object initialization. Scott Myers [34] and others advocate the “object initialization on construction” style of developing subclasses on the basis that it makes the subclasses easier to write. However, this approach is not optimal for the reusability of a subclass in different use cases from the ones for which the subclass was originally designed. To maximize ease of use by clients and maximize reusability, another style of developing subclasses “independent object allocation and initialization” is to be preferred. This latter style of developing subclasses is the approach that is adopted by all of the TSFCore concrete subclasses. To support this, every concrete subclass has a default constructor (which constructs to an uninitialized state) and a set of `initialize(...)` functions that are used to actually initialize the object. In order to also support the “object initialization on construction” style (which is useful in many different cases)

there are also a corresponding set of constructors that call these `initialize(...)` functions using the same arguments. For an example of this style, see the concrete subclass `MultiVectorCols` in the Doxygen documentation.

e) Error handling in TSFCore uses built-in exception handling in C++. All exceptions thrown by TSFCore code are derived from `std::exception`. Exceptions are thrown using the macro `TEST_FOR_EXCEPTION(...)` which results in the `std::exception::what()` function containing an error message with the file name and line number from where the exception was thrown. This type of information is very helpful in debugging. In many cases, armed with just this information and a good programmer-developed error message, a bug can be found, diagnosed and fixed without even needing to run a debugger. The use of the macro `TEST_FOR_EXCEPTION(...)` was shown in several of the above example code snippets.

## 7.2 General object-oriented design concepts used by TSFCore

Finally, a few comments on proper object-oriented design are described here that used in the design of TSFCore. These concepts are:

- a) Every operation should be implementable by every concrete subclass.
- b) If a) is not possible, then all preconditions that need to be satisfied in order for an operation to be called successfully must be reasonably knowable by the client.
- c) If a postcondition for an operation can not be met then an exception should be thrown.
- e) An interface should include all operations needed for near-optimal performance in every reasonable use case.
- d) But, reasonable default implementations should be provided for as many operations in an interface as possible in order to allow for faster prototyping efforts.

a) It is generally accepted that object-oriented interfaces should be minimal and every function in an interface should be implementable by every concrete implementation [42, Section 24.4.3]. However, there are some cases where the goals of simplicity and strict conformance to this principle of ideal object-oriented design are at odds. Finding the proper balance of simplicity and strict object-oriented correctness requires knowledge, experience and taste. In all but one case, the TSFCore interfaces strictly conform to this ideal principle of object-oriented design. The one exception is the support of transposed (adjoint) operations. If an operation may not be supportable by an implementation then the interface should provide a way for the client to discern this without having to actually invoke the operation.

b) The principle in a) is related to another principle of proper object-oriented design that absolutely every interface and function in TSFCore adheres to and this is the principle that every function

should have its preconditions (see [22] for a decision of pre- and postconditions) clearly stated and the client should be able to check the preconditions before the function is called. Failure to use this principle makes the use of such software very difficult and results in a lot of unexpected runtime errors. If an operation can not be performed by an object because of the violation of a precondition, then a good way to handle this is for the function to throw an exception. However, proper object-oriented design does not require this since it is the responsibility of the client to ensure that preconditions are satisfied (see [22]). In practice, however, defensive programming practices (see [42]) dictate that clients should be considered to be unreliable and therefore all preconditions should be checked by every major function implementation (at least in a debug build) and if a precondition is found to be violated then an exception should be thrown which contains a detailed error message that describes the problem (i.e. as returned from `std::exception::what()`).

c) If the preconditions are met before the function is called and the function can not satisfy the postconditions for some reason then the function should throw an exception in general. This latter type of exception is the primary reason that exception handling was added to the C++ standard in the first place [41].

d) Another desirable principle of object-oriented design is that an interface should provide declarations for all important functions for which if specialized implementations for all of these functions were provided, then the resulting overall software implementation would be near-optimal with respect to storage and runtime efficiency. Again, knowledge, experience and taste are required in the selection of the appropriate set of functions. However, there is conflict between the goals of declaring many functions for the sake of near-optimal performance and the desire to keep the number of functions to a minimum to ease subclass development. This brings us to our next principle.

e) The approach that each TSFCore interface takes to the issue in d) is that the (nearly) full set of functions needed for a near-optimal implementation are declared in the interface but reasonable (suboptimal) default implementations are provided for as many of the functions as possible. Examples of the application of this principle are mentioned for every major TSFCore interface (for example, the default implementation of the *MultiVector* version of the function *LinearOp::apply(...)* which is based on the *Vector* version).

## 8 Nonessential but Convenient Functionality

While the basic TSFCore interfaces described in this paper provide all of the functionality required to be directly used in ANA development these interfaces lack much of the nonessential but convenient functionality that is very helpful in developing ANA code. This nonessential but convenient functionality can be built on top of the core functionality which is precisely the type of extra functionality that TSFCore/utilities and TSFExtended provide. In this section, several different examples of nonessential but convenient functionality are given along with references to where this functionality exists in TSF and *AbstractLinAlgPack*.

## 8.1 Sub-vector views as *Vector* objects

In Section 5.3.1, the use case where the sub-vectors of a *Vector* object are treated as logical vector was discussed. The example in that section got the job done but a better approach to providing access to sub-vectors is to create a sub-view decorator subclass (see the “decorator” pattern in [23]) that allows the creation of a *Vector* view object of a contiguous range of elements in another *Vector* object. Such a subclass is included in *AbstractLinAlgPack* (see *VectorSubView* and *VectorMutableSubView*) and is very useful for high-level ANA code. These “sub-view” subclasses can be easily implemented through the *Vector::applyOp(...)* function.

## 8.2 Composition of *Vector* and *LinearOp* objects

The ideal way to represent composite blocked or product vector objects, such as described in Section 5.3.1, is to create a composite blocked or product vector subclass. Interfaces and implementations of such product objects are provided in *TSFCore/utilites* (see the classes *ProductVectorSpace*, *ProductVector* and *ProductMultiVector*). These types of composite product *Vector* and *VectorSpace* subclasses are easy to develop because of the specification of *Vector::applyOp(...)*.

Note that a product vector such as

$$\tilde{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}$$

with  $N$  block vectors is distinctly different from a multi-vector

$$Y = [ y_1 \quad y_2 \quad \dots \quad y_N ]$$

with  $N$  columns. In the multi-vector  $Y$ , each of the column vectors  $y_j$  lie in the same vector space (i.e. the range space of the linear operator represented by the multi-vector) which may not be the case for the vector blocks  $x_j$  of  $\tilde{x}$  which may lie in distinctly different vector spaces  $\mathcal{X}_j$ . While it may seem that the mathematical differences between a multi-vector and a product vector are subtle, they are distinctly different from a software implication point of view. Multi-vectors are ment to represent tall, thin dense matrices such as for multiple right-hand-sides that are passed to a linear solver or for performing multiple linear operator applications (with the same linear operator) while product vectors and product vector spaces are ment to represent single vector objects which are composed of individual vector blocks such as would be used for the composite unknowns in an SFE method

or a multi-period design problem. For example, a product vector space would be able to create a product multi-vector such as

$$\tilde{Y} = \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_N \end{bmatrix}$$

where each constituent multi-vector  $Y_j$  may have a different range space but all must have the same domain space obviously. Such a product multi-vector is implemented by `ProductMulti-Vector`.

Similar generic composition subclasses can also exist for linear operators (see `TSFExtended`).

### 8.3 Matlab-like notation and handle classes for linear algebra using operator overloading

`TSFCore` contains abstractions for linear algebra objects. Mathematicians use a precise syntax to describe linear algebra operations. Matlab [33] has established a useful convention for mathematical linear algebra syntax using only ASC characters. C++ has operator overloading. When you put all of this together it seems obvious, at first glance, that operator overloading in C++ should be used to specify linear algebra operations like

$$y = Au + \gamma B^T v + \eta Cw$$

in C++ as

```
y = A*u + gamma*trans(B)*v + eta*w;
```

However, providing a near-optimal implementation (i.e. no unnecessary temporaries or multiple memory accesses) of operator overloading for linear algebra in C++ is nontrivial. While this type of syntax is desirable, it does not provide any new functionality and is only nonessential but convenient functionality and is therefore not included in the basic `TSFCore` interfaces. If such Matlab-like syntax is desired, it can be found in `TSFExtended` through the use of handle classes which include overloaded operators. However, an efficient operator overloading mechanism in C++ is hard to implement and is difficult for C++ novices to debug through. Operator overloading built on top of `TSFCore` must be bullet proof and provide unmatched exception handling so that users must never need to debug through this code.

Closely associated with operator overloading is the concept of handle classes [18]. Handles assume the same type of role as smart pointers except all of the function forwarding (which is performed automatically with the operator function `RefCountPtr<>::operator->()`) must be performed manually in handle class (which must be written and maintained for every function on every class by some developer). Handles make the implementation of linear algebra operations with operator overloading much easier. Handles are used extensively in TSFExtended. Since TSFCore does not implement operator overloading, handles classify as nonessential but convenient functionality and are therefore not included in TSFCore.

## 9 Summary

TSFCore provides the intersection of all of the functionality required by a variety of abstract numerical algorithms ranging from iterative linear solvers all the way up to optimizers. The foundation of TSFCore described here only covers vector spaces, vectors, multi-vectors and linear operators. While this is sufficient for most linear ANA algorithms (i.e. linear equation solvers and eigen value solvers) it is not sufficient for higher-level nonlinear algorithms. An extension of the basic TSFCore interfaces for nonlinear problems is described in [8].

By adopting TSFCore as a standard interface layer, interoperability between applications, linear algebra libraries and abstract numerical algorithms in advanced scientific computing environments becomes automatic to a large extent.

## References

- [1] David Abrahams. Generic programming techniques.
- [2] E. Anderson et al. *LAPACK User's Guide*. SIAM, 1995.
- [3] S. Balay, W. D. Gropp, L.C. McInnes, and B.F. Smith. PETSc 2.0. <http://www.mcs.anl.gov/petsc>.
- [4] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.
- [5] R. A. Bartlett. *Object Oriented Approaches to Large Scale NonLinear Programming For Process Systems Engineering*. Ph.D Thesis, Chemical Engineering Department, Carnegie Mellon University, Pittsburgh, 2001.
- [6] R. A. Bartlett. *An Introduction to Algorithm Development in MOOCHO*. Sandia National Labs, 2003.
- [7] R. A. Bartlett. *MOOCHO : Multifunctional Object-Oriented arCHitecture for Optimization, User's Guide*. Sandia National Labs, 2003.
- [8] R. A. Bartlett. TSFCore::Nonlin : An extension of TSFCore for the development of nonlinear abstract numerical algorithms and interfacing to nonlinear applications. Technical report SAND03-xxxx, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2003.
- [9] R. A. Bartlett. Teuchos::RefCountPtr : An intruduction to the trilinos smart reference-counted pointer class for (almost) automatic dynamic memory management in c++. Technical report SAND04-3268, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2004.
- [10] R. A. Bartlett, B. G. van Bloeman Waanders, and M. A. Heroux. Vector reduction/transformation operators for linear algebra interfaces to efficiently develop complex abstract numerical algorithms independently of data mapping, 2003. Submitted to *ACM TOMS*.
- [11] Steve Benson, Lois Curfman McInnes, and Jorge Moré. TAO : Toolkit for advanced optimization (web page).
- [12] L. S. Blackford, J. Choi, A. Cleary, E.D. 'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walder, and R.C. Whaley. *ScalLAPACK User's Guide*. SIAM, Philadelphia, PA, 1997.
- [13] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

- [14] BOOST. The BOOST library. <http://www.boost.org>.
- [15] R. H. Byrd, J. Nocedal, and R.B. Schnabel. Representations of quasi-newton matrices and their use in limited methods. *Math. Prog.*, 63:129–156, 1994.
- [16] George D. Byrne and Allan C. Hindmarsh. PVODE, an ODE solver for parallel computers. *Int. J. High Perf. Comput. Applic.*, 13:354–365, 1999.
- [17] Robert L. Clay, Kyran D. Mish, Ivan J. Otero, Lee M. Taylor, and Alan B. Williams. An annotated reference guide to the finite-element interface (FEI) specification : Version 1.0. Technical Report SAND99-8229, Sandia National Laboratories, 1999.
- [18] J. O. Coplien. *Advanced C++*. Addison-Wesley, 1992.
- [19] J. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.
- [20] H. C. Edwards. SIERRA framework version 3: Core services theory and design. Technical report SAND2002-3616, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2002.
- [21] The MPI Forum. *MPI: A Message Passing Interface Standard*. University of Tennessee, 1995. <http://www.mpi-forum.org/docs/docs.html>.
- [22] M. Fowler and K. Scott. *UML Distilled, second edition*. Addison-Wesley, 2000.
- [23] E. Gamma et al. *Design Patterns: Elements fo Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [24] M. S. Gockenbach, M. J. Petro, and W. W. Symes. C++ classes for for linking optimization with complex simulations. *ACM Transactions on Mathematical Software*, 1999.
- [25] M. A. Heroux. Epetra : Concrete C++ linear algebra classes for parallel linear algebra. <http://software.sandia.gov/Trilinos>.
- [26] Mike A. Heroux, Teri Barth, David Day, Rob Hoekstra, Rich Lehoucq, Kevin Long, Roger Pawlowski, Ray Tuminaro, and Alan Williams. Trilinos : object-oriented, high-performance parallel solver libraries for the solution of large-scale complex multi-physics engineering and scientific applications. <http://software.sandia.gov/Trilinos>.
- [27] S. A. Hutchinson, , J. N. Shadid, and R. S. Tuminaro. Aztec user’s guide: Version 1.0. Technical report, Sandia National Laboratories, Albuquerque, New Mexico 87185, 1995. SAND95-1559.
- [28] S. A. Hutchinson, L. V. Prevoost, J.N. Shadid, C. Tong, and R.S. Tuminaro. Aztec user’s guide: Version 2.0. Technical Report ANL-95/11–Revision 2.0.22, Sandia National Laboratories, 1998.
- [29] T. Kolda and R. Pawlowski. NOX: An object-oriented, nonlinear solver package. <http://software.sandia.gov/Trilinos>.

- [30] Sandia National Labs. ESI: Equation solver interface. <http://z.ca.sandia.gov/esi>, 2001.
- [31] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.
- [32] A. Lumsdanie and J. Siek. The matrix template library. <http://www.lsc.nd.edu/research/mtl/>, 1998.
- [33] Mathworks. Matlab<sup>©</sup>: The language of technical computing. <http://www.mathworks.com>.
- [34] S. Meyers. *More Effective C++*. Addison-Wesley, 1996.
- [35] J. Nocedal and S. Wright. *Numerical Optimization*. Springer, New York, 1999.
- [36] Department of Energy. ASCI: Advanced simulation and computing initiative. <http://www.sandia.gov/ASCI>.
- [37] M. F. Pellissetti and R. G. Ghanem. Iterative solution of systems of linear equations arising in the context of stochastic finite elements. *Advances in Engineering Software*, 54(189):211–230, 1990.
- [38] R. Pozo. TNT: Template numerical toolkit. <http://math.nist.gov/tnt>.
- [39] S. Roberts et al. Meschach++: Matrix computations in c++. <http://www.netlib.org/c/meschach/>, 1996.
- [40] Roque Wave Software. Math++: Object-oriented library for numeric computation. Technical Report 96-03, Roque Wave Software, Inc., 1996.
- [41] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, New York, 1994.
- [42] B. Stroustrup. *The C++ Programming Language, special edition*. Addison-Wesley, New York, 1997.
- [43] AEA Technology. *Harwell Subroutine Library: Release 12*. Harwell Laboratory, Oxfordshire, England, 1995.



## A TSFCore C++ class declarations

```
namespace TSFCore {

using RangePack::RangelD;
template<class Scalar> class VectorSpaceFactory;
template<class Scalar> class VectorSpace;
template<class Scalar> class Vector;
template<class Scalar> class OpBase;
template<class Scalar> class LinearOp;
template<class Scalar> class MultiVector;
template<class Scalar>

class VectorSpaceFactory {
public:
    virtual Teuchos::RefCountPtr< const VectorSpace<Scalar> > createVecSpc(int dim) const = 0;
};

template<class Scalar>
class VectorSpace {
public:
    virtual ~VectorSpace() {}
    virtual Index dim() const = 0;
    virtual bool isCompatible( const VectorSpace<Scalar>& vecSpc ) const = 0;
    virtual Teuchos::RefCountPtr< Vector<Scalar> > createMember() const = 0;
    virtual bool isInCore() const;
    virtual Teuchos::RefCountPtr< MultiVector<Scalar> > createMembers(int numMembers) const;
    virtual Teuchos::RefCountPtr<Vector<Scalar> > createMemberView(
        const RTOpPack::MutableSubVectorT<Scalar> &raw_v ) const;
    virtual Teuchos::RefCountPtr<const Vector<Scalar> > createMemberView(
        const RTOpPack::SubVectorT<Scalar> &raw_v ) const;
    virtual Teuchos::RefCountPtr<MultiVector<Scalar> > createMembersView(
        const RTOpPack::MutableSubMultiVectorT<Scalar> &raw_mv ) const;
    virtual Teuchos::RefCountPtr<const MultiVector<Scalar> > createMembersView(
        const RTOpPack::SubMultiVectorT<Scalar> &raw_mv ) const;
    virtual Teuchos::RefCountPtr< const VectorSpaceFactory<Scalar> > smallVecSpcFcty() const;
    virtual Scalar scalarProd( const Vector<Scalar>& x, const Vector<Scalar>& y ) const;
    virtual void scalarProds( const MultiVector<Scalar>& X, const MultiVector<Scalar>& Y
        ,Scalar scalar_prods[] ) const;
    virtual Teuchos::RefCountPtr< const VectorSpace<Scalar> > clone() const;
};

template<class Scalar>
class OpBase {
public:
    virtual ~OpBase();
    virtual Teuchos::RefCountPtr< const VectorSpace<Scalar> > domain() const = 0;
    virtual Teuchos::RefCountPtr< const VectorSpace<Scalar> > range() const = 0;
    virtual bool opSupported(ETransp M_trans) const;
};

template<class Scalar>
class LinearOp : virtual public OpBase<Scalar> {
public:
    virtual void apply( const ETransp M_trans, const Vector<Scalar> &x
        ,Vector<Scalar> *y ,const Scalar alpha=1.0 ,const Scalar beta=0.0 ) const = 0;
    virtual Teuchos::RefCountPtr<const LinearOp<Scalar> > clone() const;
    virtual void apply( const ETransp M_trans ,const MultiVector<Scalar> &X
        ,MultiVector<Scalar> *Y ,const Scalar alpha=1.0 ,const Scalar beta=0.0 ) const;
};
```

```

template<class Scalar>
class MultiVector : virtual public LinearOp<Scalar> {
public:
    virtual Teuchos::RefCountPtr<const Vector<Scalar> > col(Index j) const;
    virtual Teuchos::RefCountPtr<Vector<Scalar> > col(Index j) = 0;
    virtual Teuchos::RefCountPtr<const MultiVector<Scalar> > clone_mv() const;
    virtual Teuchos::RefCountPtr<MultiVector<Scalar> > clone_mv();
    virtual Teuchos::RefCountPtr<const MultiVector<Scalar> > subView( const RangeID& col_rng ) const;
    virtual Teuchos::RefCountPtr<MultiVector<Scalar> > subView( const RangeID& col_rng );
    virtual Teuchos::RefCountPtr<const MultiVector<Scalar> > subView( const int numCols, const int cols[] ) const;
    virtual Teuchos::RefCountPtr<MultiVector<Scalar> > subView( const int numCols, const int cols[] );
    virtual void applyOp( const RTopPack::RTOPT<Scalar> &primary_op, const size_t num_multi_vecs
        ,const MultiVector<Scalar>* multi_vecs[], const size_t num_targ_multi_vecs
        ,MultiVector<Scalar>* targ_multi_vecs[], RTopPack::ReductTarget* reduct_objs[], const Index primary_first_ele
        ,const Index primary_sub_dim,const Index primary_global_offset, const Index secondary_first_ele
        ,const Index secondary_sub_dim ) const;
    virtual void applyOp( const RTopPack::RTOPT<Scalar> &primary_op, const RTopPack::RTOPT<Scalar> &secondary_op
        ,const size_t num_multi_vecs, const MultiVector<Scalar>* multi_vecs[], const size_t num_targ_multi_vecs
        ,MultiVector<Scalar>* targ_multi_vecs[], RTopPack::ReductTarget* reduct_obj, const Index primary_first_ele
        ,const Index primary_sub_dim, const Index primary_global_offset, const Index secondary_first_ele
        ,const Index secondary_sub_dim ) const;
    virtual void getSubMultiVector( const RangeID &rowRng, const RangeID &colRng
        ,RTopPack::SubMultiVectorT<Scalar> *sub_mv ) const;
    virtual void freeSubMultiVector( RTopPack::SubMultiVectorT<Scalar>* sub_mv ) const;
    virtual void getSubMultiVector( const RangeID &rowRng, const RangeID &colRng
        ,RTopPack::MutableSubMultiVectorT<Scalar> *sub_mv );
    virtual void commitSubMultiVector( RTopPack::MutableSubMultiVectorT<Scalar>* sub_mv );
    void apply( const ETransp M_trans, const Vector<Scalar> &x, Vector<Scalar> *y, const Scalar alpha
        ,const Scalar beta ) const;
    Teuchos::RefCountPtr<const LinearOp<Scalar> > clone() const;
};

```

```

template<class Scalar>
class Vector : virtual public MultiVector<Scalar> {
public:
    virtual ~Vector() {}
    virtual Teuchos::RefCountPtr< const VectorSpace<Scalar> > space() const = 0;
    virtual void applyOp( const RTopPack::RTOPT<Scalar> &op, const size_t num_vecs
        ,const Vector<Scalar>* vecs[], const size_t num_targ_vecs ,Vector<Scalar>* targ_vecs[]
        ,RTopPack::ReductTarget *reduct_obj ,const Index first_ele ,const Index sub_dim
        ,const Index global_offset ) const = 0;
    virtual void getSubVector( const RangeID& rng, RTopPack::SubVectorT<Scalar>* sub_vec ) const;
    virtual void freeSubVector( RTopPack::SubVectorT<Scalar>* sub_vec ) const;
    virtual void getSubVector( const RangeID& rng, RTopPack::MutableSubVectorT<Scalar>* sub_vec );
    virtual void commitSubVector( RTopPack::MutableSubVectorT<Scalar>* sub_vec );
    virtual void setSubVector( const RTopPack::SparseSubVectorT<Scalar>& sub_vec );
    // Multi-vector overrides
    ...
};

```

```

template<class Scalar>
void applyOp( const RTopPack::RTOPT<Scalar> &op, const size_t num_vecs
    ,const Vector<Scalar>* vecs[], const size_t num_targ_vecs ,Vector<Scalar>* targ_vecs[]
    ,RTopPack::ReductTarget *reduct_obj ,const Index first_ele=1 ,const Index sub_dim=0
    ,const Index global_offset=0 );

```

```

template<class Scalar>
void applyOp( const RTopPack::RTOPT<Scalar> &primary_op, const size_t num_multi_vecs
    ,const MultiVector<Scalar>* multi_vecs[], const size_t num_targ_multi_vecs
    ,MultiVector<Scalar>* targ_multi_vecs[], RTopPack::ReductTarget* reduct_objs[], const Index primary_first_ele=1

```

```
,const Index primary_sub_dim=1, const Index primary_global_offset=0, const Index secondary_first_ele=1
,const Index secondary_sub_dim=0 ) const;

template<class Scalar>
void applyOp( const RTOpPack::RTOpT<Scalar> &primary_op, const RTOpPack::RTOpT<Scalar> &secondary_op
, const size_t num_multi_vecs, const MultiVector<Scalar>* multi_vecs[], const size_t num_targ_multi_vecs
,MultiVector<Scalar>* targ_multi_vecs[], RTOpPack::ReductTarget *reduct_obj, const Index primary_first_ele=1
,const Index primary_sub_dim=0, const Index primary_global_offset=0, const Index secondary_first_ele=1
,const Index secondary_sub_dim=0 ) const;

} // namespace TSFCore
```

## DISTRIBUTION:

- |  |   |
|--|---|
| 1 Omar Ghattas<br>Carnegie Mellon University<br>5000 Forms Ave.<br>Pittsburgh, PA 15213  | 1 George Biros<br>Department of Applied Mechanics and<br>Mechanical Engineering<br>University of Pennsylvania<br>220 Towne Building 220 S. 33rd St.<br>Philadelphia, PA 19104-6315, USA |
| 1 Larry Biegler<br>Department Chemical Engineering<br>Carnegie Mellon University<br>5000 Forms Ave.<br>Pittsburgh, PA 15213                                    | 1 MS 0847<br>Bill Camp, 9200  |
| 1 Carl Laird<br>Department Chemical Engineering<br>Carnegie Mellon University<br>5000 Forms Ave.<br>Pittsburgh, PA 15213                                       | 1 MS 0847<br>Sudip Dosanjh, 9233  |
| 1 Matthias Heinkenschloss<br>Department of Computational and Applied<br>Mathematics<br>MS 134 Rice University<br>6100 S. Main Street<br>Houston, TX 77005-1892 | 1 MS 0847<br>Scott Mitchell, 9211   |
| 1 Bill Symes<br>Department of Computational and Applied<br>Mathematics<br>MS 134 Rice University<br>6100 S. Main Street<br>Houston, TX 77005-1892              | 1 MS 1110<br>Roscoe Bartlett, 9211  |
| 1 Tony Padula<br>Department of Computational and Applied<br>Mathematics<br>MS 134 Rice University<br>6100 S. Main Street<br>Houston, TX 77005-1892             | 1 MS 1110<br>Scott Collis, 9211   |
| 1 Mark Gockenbach<br>Department of Mathematical Sciences<br>Michigan Technological University<br>1400 Townsend Drive<br>Houghton, Michigan 49931-1295, U.S.A.  | 1 MS 0847<br>Bart van Bloemen Waanders,<br>9211   |
|  | 1 MS 0847<br>Mike Eldred, 9211  |
|  | 1 MS 0819<br>Tim Trucano, 9211  |
|  | 1 MS 0847<br>Tony Giunta, 9211  |
|  | 1 MS 0947<br>Laura Swiler, 9211   |
|  | 1 MS 9217<br>Mark Adams, 9214   |
|  | 1 MS 1110<br>Pavel Bochev, 9214   |
|  | 1 MS 1110<br>Todd Coffey, 9214  |

1 MS 1110  
David Day, 9214

1 MS 1110  
John Delaurentis, 9214

1 MS 1110  
Michael Heroux, 9214

1 MS 1110  
Ulrich Hetmaniuk, 9214

1 MS 9217  
Jonathan Hu, 9214

1 MS 1110  
Richard Lehoucq, 9214

1 MS 1110  
Louis Romero, 9214

1 MS 1110  
David Ropp, 9214

1 MS 1110  
Heidi Thornquist, 9214

1 MS 9217  
Raymond Tuminaro, 9214

1 MS 1110  
James Willenbring, 9214

1 MS 1110  
David Womble, 9214

1 MS 9217  
Steve Thomas, 8962

1 MS 9217  
Paul Boggs, 8962

1 MS 9217  
Kevin Long, 8962

1 MS 9217  
Patricia Hough, 8962

1 MS 9217  
Tamara Kolda, 8962

1 MS 9217  
Monica Martinez-Canales, 8962

1 MS 9217  
Pamela Williams, 8962

1 MS 9217  
Victoria Howle, 8962

1 MS 1110  
William Hart, 9215

1 MS 0847  
Steve Wojtkiewicz, 9124

1 MS 0316  
Eric Keiter, 9233

1 MS 0316  
Scott Hutchinson, 9233

1 MS 0316  
Curt Ober, 9233

1 MS 0316  
Tom Smith, 9233

1 MS 9143  
Carter Edwards, 0827

1 MS 9143  
James Stewart, 0826

1 MS 0819  
Ricard Drake, 9231

1 MS 0316  
Robert Hoekstra, 9233

1 MS 0316  
Roger Pawlowski, 9233

1 MS 1110  
Eric Phipps, 9233

1 MS 1110  
Andrew Salinger, 9233

1 MS 0826  
Alan Williams, 8961

1 Kendall Stanley

1 MS 9018  
Central Technical Files, 8945-1

2 MS 0899  
Technical Library, 9610

2 MS 0612  
Review & Approval Desk, 4916