MV++ v. 1.5a

Matrix / Vector Class
Reference Guide

November 8, 1995

Roldan Pozo

Applied and Computational Mathematics Divison
National Institute of Standards and Technology

Contents

1 Overview **1**
1.1 About MV++ . . . 1
1.2 Basic Features . . . 1
1.3 MV++ Classes . . . 3
1.3.1 Templated class version . . . 3
1.3.2 Non-templated class version . . . 3
1.4 References . . . 4

2 Reference Manual **5**
MV_Vector . . . 5
MV_VecIndex . . . 10
MV_ColMat . . . 13

Chapter 1

Overview

1.1 About MV++

MV++ is a small, efficient, set of concrete vector and matrix classes specifically designed for high performance numerical computing.

The MV++ package includes interfaces to the computational kernels found in the Basic Linear Algebra Subprograms (BLAS), such as scalar updates, vector sums, and dot products. The idea is to utilize vendor-supplied, or optimized BLAS routines that are fine-tuned for particular platforms.

The various MV++ classes form the building blocks of larger user-level libraries such as SparseLib++[2] and LAPACK++[1]. The MV++ library was put together with the following principles in mind:

- a simple **numerical** vector and column-oriented dense matrix class.
- **minimal overhead** in constructing, assigning, and copying vectors and matrices
- **performance** competitive with optimized Fortran kernels
- data structure **compatibility** with Fortran libraries and subroutines
- concrete classes, **no complicated inheritance hierarchy**, but
- support generic element types through **templated** paramters
- support operations with contiguous **subvectors** and **submatrices** (e.g. zeroing out a section of a vector.)
- optional runtime support for **array-bounds checking**.

1.2 Basic Features

MV++ provides two basic classes: a numerical vector (1-d array), and column (Fortran) oriented dense matrix. Indexing is performed via the `operator()`, as in `A(i,j)`.

Subvectors and submatrices can be accessed through `MV_VecIndex` classes. In other words, if B is the vector

$$\{0.0,\ 0.1,\ 0.2,\ 0.3,\ 0.4,\ 0.5,\ 0.6\}$$

then `B(MV_VecIndex(1,5)) = 9.9` sets B to

$$B = \{0.0,\ 0.1,\ 9.9,\ 9.9,\ 9.9,\ 9.9,\ 9.9,\ 0.6\}$$

and is equivalent to

```
for (i=1; i<=5; B[i++] = 9.9 )
```

Other considerations:

- efficient indexing as fast as native C arrays
- supports only unit strides (for efficient indexing)
- copy-by-value semantics
- optional "share" semantics allows vectors to be constructed as "views", or "references" of an existing memory, using MV_Vector_::ref modifier in the constructor. (see note below.)
- vector views can assign and references sections of vector, but cannot modify their *size*.
- block-range indexing via MV_VecIndex class (e.g. A(I) = B;) (note for the above to work, A(I) must return a vector view.)
- optional range checking (compile switch)
- support for both [] and () style indexing ([] not available for matrices.)
- Unrolling of loops (depth=4) for copying and assigning vectors. Thus on some machines, it may be faster to execute A=scalar, than to manually assign a native C array using an explicit for loop:

  ```
  for (i=0; i<N; d[i++] = scalar);
  ```

- function code for the () and [] operators has been inlined into the class declaration, for compilers that refuse to inline otherwise.
- optinal share semnatics. To create a view of (or reference to) an existing MV++ matrix or vector, use

  ```
  MV_Vector_double A( &d[0], n, MV_Vector_::ref );
  ```

  This allows one to construct vectors as views of any contiguous C array. It will not release the memory space when the vector is destroyed or goes out of scope.

1.3 MV++ Classes

MV++ supports both templated and non-templated vector and matrix classes. Non-templated versions of the classes are useful when using C++ compilers that do not provide full support for template instantiations (several compilers have problems using templates in applications linked with multiple .o files), or when large template header files begin to seriously affect compilation.

1.3.1 Templated class version

Templated MV++ vectors are denoted as `MV_Vector<type>` in `./include/mvvtp.h`. Matrices are denoted as `MV_ColMat<type>` in `./include/mvmtp.h`.

Typical use is illustrated by an example:

```
#include "mvmtp.h"

// ...

class MyObj {  // ...  };

MV_ColMat<MyObj> A(m, n);
MV_Vector<MyObj> B(n), C(n);
```

The requirements of `MyObj` is that it must be form algebraic field. That is, the elements form a closure under the operations `+`, `*`, `/`, and `-`.

1.3.2 Non-templated class version

Non-templated classes in MV++ have names such as `MV_Vector_double` and `MV_MatCol_int`. By default, the inital installation supports

- `MV_Vector_double`
- `MV_Vector_float`
- `MV_Vector_int`
- `MV_Vector_complex`
- `MV_ColMat_double`
- `MV_ColMat_float`
- `MV_ColMat_int`

- MV_ColMat_complex

In general, the class corresponding to the templated equivalent of XX<t> is denoted as XX_t.

Type specific versions the MV++ classes are generated from the same "base" file via an editor or simple preprocessor, such as sed. For example, mvvt.h defines a class of

```
class MV_Vector_$TYPE
{
   protected:
        $TYPE *p_;
        ...
}
```

By creating a copy of this file in which every occurence of "$TYPE" is changed to the name of a user-specific class, one can create MV++ vectors out of any numerical object representing an algebraic ring. The simple command,

```
sed '1,$s/\$TYPE/MyObject/g' mvv.h > mvv_MyObject.h
```

will create an MV++ vector of MyObjects. Similarly, one can also create the accompanying mvv_MyObject.cc file from the mvv.cc base in the /src directory. [1]

1.4 References

[1] J. Dongarra, R. Pozo, D. Walker, "LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra," Proceedings of Supercomputing '93, IEEE Press, 1993, pp. 162-171.

[2] J. Dongarra, A. Lumsdaine, R. Pozo, K. Remington, "A Sparse Matrix Library in C++ for High Performance Architectures," Proceedings of the Object Oriented Numerics Conference, 1994, pp. 214-218.

[1] Use a preprocessor that can also change subtring expressions, since the $TYPE expression occurs as part of the class name.

Chapter 2

Reference Manual

Name

MV_Vector

Description

One-dimensional vector storage class with minimal overhead. It is one step above a C array; it utilizes copy-by-value semantics, provides for unit-stride referencing and indexing using *(start, end)* pairs.

- deep-copy (optimized)
- only a container class, no mathemtical operations defined yet.
- unit stride (elements are in contiguous memory locations)
- fixed 0-based offset
- $A(i)$ declared `inline` for efficiency

Major differences between original LAPACK++ vector class and MV++

- templated
- copy-by-value, rather than share-semantics
- much faster A(i) indexing, since indices always have unit stride.
- only *one* owner of data, but maybe various references, so no reference-counting scheme is used.

Declaration

#include < mvvtp.h >

MV_Vector<TYPE>()

Construct a null vector of zero length.

MV_Vector<TYPE>(int n = 0)

Construct a vector of length n, ($n \geq 0$). A vector of length zero is perfectly legal and usually termed a *null* vector. MV_Vector elements are UNITIALIZED.

MV_Vector<TYPE>(int n, double s)

Construct a vector of length n and initalize all elements to the scalar value s.

MV_Vector<TYPE>(double * x, int n)

Construct a n-length vector as a new **copy** of an existing C/C++ array.

MV_Vector<TYPE>(double * d, int n, MV_Vector_::ref)

Construct a n-length vector as a **view** (share semantics) of an existing C/C++ array. Further changes to elments of `d` will be reflected in `MV_Vector<TYPE>`. Data space `d` will **not** be destroyed when calling `MV_Vector<TYPE>`.

MV_Vector<TYPE>(const MV_Vector<TYPE>& V)

Create a new n-length vector as a copy of an existing `MV_Vector<TYPE>`.

~MV_Vector<TYPE>()

Reclaim vector memory space if this the only structure using it.

Assignments

MV_Vector<TYPE>& operator=(const MV_Vector<TYPE>& V)

If `*this` is a reference then **inject()** left-hand side (V) into existing memory (both sides must conform). Otherwise, `*this` owns its data space, so delete it and create a new copy of `V`. (If conformant with `V`, then just copy in place.) Return reference to `*this` view.

MV_Vector<TYPE>& operator=(double s)

Set elements of left-hand size to the scalar value s.

MV_Vector<TYPE>& inject(const MV_Vector<TYPE>& V)

Copy elements of `V` into the memory space referenced by the left-hand side, without first releasing it. The effect is that if other vectors share memory with left-hand side, they too will be affected. Note that the length of `V` must be same be the same as that of the left-hand side vector.

MV_Vector<TYPE>& copy(MV_Vector<TYPE>& V)

Release left-hand side and copy elements of `V`. Unlike `MV_Vector<TYPE>::inject()` it does not require conformity, and previous references of left-hand side are unaffected.

int newsize(int n)

Resize to a *new* vector of length n. The element values are UNINITALIZED, even if n is less than the current vector length.

Access Functions

TYPE& operator ()(int i)

Return ith element of vector, with zero-based offset. Optional runtime bounds checking ($0 \leq i \leq n$) set by compile time macro `MV_VECTOR_BOUNDS_CHECK`.

TYPE& operator [ ](int i)

Identical to `MV_Vector<TYPE>::operator()` above. Included mainly for compatiblity to C/C++ `[]` syntax.

TYPE& operator ( )(const MV_VecIndex &I)

Returns a reference (view) of this vector specified by the ranges in `MV_VecIndex`.

For example, the following statements

```
MV_Vector<int> A(20), B(30);
MV_VecIndex I(0,3), J(7,10);

A(I) = B(J);
```

assign the first four elements of A to the values of `B(7)` through `B(10)`.

Information Functions

int size()

Return the length, n, of the vector.

int null()

Shorthand to test if zero-length vector. Identical to `(size()==0)`.

`int ref()`

return 1 if vector is a view (reference) to another vector or C array, zero otherwise.

Macros

`MV_VECTOR_BOUNDS_CHECK`

Optional compile time macro, to perform bounds checking ($0 \leq i \leq n$) in a n-length vector. The default is NOT to perform this check (this is consistent with C/C++); however, it is **highly recommended**, particularly during inital phases of development, testing, and debugging. There is a performance penalty for this, (essentially a boolean test at each element reference) so it can turned off for production runs, where performance may be critical.

I/O Functions

`ostream& operator ≪(ostream& s, const MV_Vector<TYPE>& V)`

Print vector, one element per line.

See also MV_ColMat

Name

MV_VecIndex : a contiguous subrange of MV_Vector elements.

Description

`MV_VecIndex` is an integer pair denoting the start and ending indices of a vector view. Only supports unit strides, so there is no *increment* argument. As an example, if B is the vector

$$\{0.0,\ 0.1,\ 0.2,\ 0.3,\ 0.4,\ 0.5,\ 0.6\}$$

then `B(MV_VecIndex(1,5)) = 9.9` sets B to

$$B = \{0.0,\ 0.1,\ 9.9,\ 9.9,\ 9.9,\ 9.9,\ 9.9,\ 0.6\}$$

and is equivalent to

```
for (i=1; i<=5; B[i++] = 9.9 )
```

Declaration

#include <mvvind.h>

MV_VecIndex()

Construct a null index (start=end=-1). Used to denote the complete vector range. For example,

```
B(MV_VecIndex()) = A;
```

will set all elements of B to those of A. This is equivalent to writing

```
B(MV_VecIndex(0,B.size()-1) = A;
```

or

```
B() = A;
```

or

```
for (i=0; i<N; i++)
  B(i) = A(i);
```

MV_VecIndex(unsigned int i, unsigned int j)

Construct an index range starting from position `i` through `j`. Conditions: $0 \leq i \leq j$.

MV_VecIndex(unsigned int i)

Construct an index range consisting of a single position, `i`. Used mainly for converting integers into MV_VecIndex's.

int start()

Returns the first index of the index range, (0 if MV_VecIndex is set to `all()`).

int end()

Returns the end of index range, or -1 if MV_VecIndex is set to `all`.

int length()

Returns the number of elements in index range, or -1 if MV_VecIndex is set to `all()`.

MV_VecIndex& operator+=(int i)

moves index range up by `i` elements. For example,

```
MV_VecIndex I(1:10);
I+= 3;
```

reset `I` to be $(4:13)$.

MV_VecIndex operator+(int i)

creates new index whose range is moved up by `i` elements. For example,

```
MV_VecIndex I(1:10);
MV_VecIndex K = I+3;
```

sets `K` to be $(4:13)$.

`MV_VecIndex operator-(int i)`

creates new index whose range is moved down by `i` elements,

`MV_VecIndex& operator-=(int i)`

moves index range down by `i` elements.

Name

MV_ColMat : column oriented (Fortran) templated dense matrix.

Description

A two-dimensional version of `MV_Vector<TYPE>`. Storage is column oriented, compatible as an argument to Fortran libraries.

- uses (deep) copy semantics
- indexing via `A(i,j)` where i, j are either integers or `MV_VecIndex` indices.
- supports only contiguous submatrices
- utilizes `Vector<TYPE>` container class
- has basic BLAS++ math functionality
- optimized to avoid memory copies when returning temporary `MV_ColMat<TYPE>` results by value from functions

Declaration

#include <mvvtp.h>

MV_ColMat<TYPE>()

Construct a null $0x0$ matrix.

MV_ColMat<TYPE>(int m, int n)

Construct a column-major matrix of size $m \times n$, $(m, n \geq 0)$. Matrix elements are UNITIALIZED.

MV_ColMat<TYPE>(int m, int n, const TYPE& s)

Construct a column-major matrix of size $m \times n$, $(m, n \geq 0)$, and initalize matrix elments to the scalar s.

MV_ColMat<TYPE>(TYPE * v, int m, int n)

Construct a $m \times n$ matrix by copying the values from a one-dimensional C/C++ array of length mn.

MV_ColMat<TYPE>(TYPE * v, int m, int n, MV_ColMat_::ref)

Construct a $m \times n$ column-oriented matrix as a *view* of exisiting C array (length $m \times n$).

MV_ColMat<TYPE>(const MV_ColMat<TYPE>& V)

Create a new n-length vector from an existing one by copying.

int newsize(int m, int n)

Resize to a *new* matrix of size $m \times n$. The element values are UNINITALIZED, even if resizing to a smaller matrix.

~MV_ColMat<TYPE>()

Destroy matrix and reclaim vector memory space if this the only structure using it.

Assignments

MV_ColMat<TYPE>& operator=(const MV_ColMat<TYPE>& M)

Release left-hand side (reclaiming memory space if possible) and construct a new copy of `V`. Return reference to new copy.

MV_ColMat<TYPE>& operator=(const TYPE& s)

Set elements of left-hand size to the scalar value s. No new matrix is created, so that if there other matrices that may reference this memory space will also be affected.

Access Functions

TYPE& operator ( )(int i, int j)

Return (i, j)th element of vector, with zero-based offset. Optional runtime bounds checking $(0 \leq i \leq m)$, $(0 \leq j \leq n)$, set by compile time macro `MV_MATRIX_BOUNDS_CHECK`.

TYPE& operator ( )(MV_VecIndex I, MV_VecIndex J)

Return submatrix view specified by indices `I` and `J`. (See `MV_VecIndex` class.) These indices specify start, increment, and ending offsets, similar to triplet notation of Matlab or Fortran 90. For example, if `B` is a 10×10 matrix, `I` is `(0:1)` and `J` is `(3:4)`, then `B(I,J)` denotes the 2×2 matrix

$$\begin{bmatrix} b_{0,3} & b_{0,4} \\ b_{0,4} & b_{1,4} \end{bmatrix}$$

.

Information Functions

int size(int d)

Return the length, n, of the dth dimension, i.e. for an $M \times N$ matrix `size(0)` returns M and `size(1)` returns N.

int lda()

Return the leading dimension of this matrix ($>= M$).

int ref()

Returns 1 if this matirix is a view to another C/C++ array.

Macros

MV_MATRIX_BOUNDS_CHECK

Compile time macro, either defined or undefined to perform bounds checking on matmrix indexing operations. The default is NOT to perform this check (this is consistent with C/C++); however, it is **highly recommended** to utilize this check – particularly during inital phases of development, testing, and debugging. There is a performance penalty for this, (essentially a boolean test at each element reference) so it can turned off for production runs, where performance may be critical.

I/O Functions

friend ostream& operator *ll*(ostream& s, const MV_ColMat<TYPE>& V)

Print matrix (one row perline), with elements seperated by white space.

See also MV_Vector<TYPE>, MV_VecIndex