Data Objects in CMAT©

Wolfgang M. Hartmann

December 2015

Contents

1 Hierarchy of Data Objects

1. scalars: data types: string, (long) int, (long) real, complex;

2. vectors: one index (dimensionality=1) :
   with entries of single and multiple data types;
   sparse and dense storage;

3. matrices: two indices (dimensionality=2) :
   with entries of single and multiple data types;
   dense and sparse storage;
   diagonal dense and sparse storage;
   compact triangular storage: for symmetric, lower, and upper triangular;

4. tensors: more than two indices (dimensionality > 2) :
   sparse and dense storage;
   with entries of single and multiple data types;

5. lists: are vectors of scalars, vectors, matrices, tensors, lists, and structs;
   entries are referred to by list name with a single index in brackets;

6. structs: consist of scalars, vectors, matrices, tensors, lists, structs;
   entries are referred to by a compound name, i.e. struct name dot entry name.

2 Data Types

Currently the following data types are implemented in CMAT:

int data are stored in long int's, i.e. on a 32 bit processor PC they are stored using 4 bytes.

real data are stored in long float or double, i.e. on a 32 bit processor PC they are stored using 8 bytes.

complex data are stored in two doubles, i.e. on a 32 bit processor PC they are stored using two times 8 bytes.

string data are stored bytewise as null terminated strings.

For an object like vector, matrix, or tensor, all numerical values are stored with the most dominant data type, i.e. if there is only one complex value in the matrix all other values are stored in complex form too. (Note, the runtime option `RELZERO=` can be used to control whenever the imaginary part of a complex number is treated as nonzero.) Planned for the future is the storage of binary

data. In the following, objects containing both numeric and string data are called *mixed data type* objects. Note, the strings for row, column, or general dimension names do not count as string data of the object.

3 Storage Forms of Objects

3.1 Storage Forms of Vectors

Row or column vectors of length n are stored in three forms:

- dense and uniform numeric data type storage form of all n entries,
- sparse and uniform data type storage form which takes the index of each nonzero as additional information,
- mixed data type storage form which takes additional information for each nonzero entry (`index`, `type`, `addr`), where `index` is the integer for row or column index, `type` is an integer the data type, and `addr` is the starting address of the entry.

If a vector contains all string data with *nearly* the same length, it is stored in dense rather than sparse form. The runtime option `RELZERO=` is used for deciding whether a value is treated as nonzero or zero.

3.2 Storage Forms of Matrices

Matrices with m rows and n columns are stored in a number of forms:

- dense and uniform numeric data type storage form:
  - as dense diagonal matrix when all off diagonal entries are zero.
  - as compact symmetric or lower or upper triangular matrix when $m = n$.
  - as general $m \times n$ matrices.
- sparse and uniform data type storage form, takes additional information for each nonzero entry (`rind`, `cind`):
  - as sparse diagonal matrix when all off diagonal entries and some or all diagonal are zero.
  - as symmetric matrix when $m = n$ and $a_{ij} = a_{ji}$.
  - as general $m \times n$ matrices.

- mixed data type storage form, takes additional information for each nonzero entry (`rind`, `cind`, `type`, `addr`):
  - as sparse diagonal matrix when all off diagonal entries and some or all diagonal are zero.
  - as symmetric matrix when $m = n$ and $a_{ij} = a_{ji}$.
  - as general $m \times n$ matrices.

Whereas the dense storage form needs only to store the value for each entry, there are two sparse storage forms, which need additional information for each enyry of the form $(rind, cind)$ for uniform data type and $(rind, cind, type, addr)$ for mixed data type, where `rind` is the integer for row index, `cind` is the integer for column index, `type` is an integer the data type, and `addr` is the starting address of the entry.

The runtime option `RELZERO=` is used for deciding whether a value is treated as nonzero or zero.

If matrix **A** has only numeric values, the decision between dense and sparse storage form is made using the setting of the runtime option `SPRANGE=`, which is by default set to .5 and means for uniform data type that if the matrix contains more than 50 % zeros it is stored and treated as sparse.

If a matrix contains all string data with *nearly* the same length, it is stored in dense rather than sparse form. Note, the runtime option `SYMCRIT=` can be used as a fuzzy range for the symmetry of two entries.

Names can be assigned for each of the m rows and n columns of vectors and matrices using the `rname` and `cname` functions.

3.3 Storage Forms of Tensors

Tensors are stored in three forms:

- dense and uniform numeric data type storage form: this assumes that all entries have the same numeric data type, i.e. int, real, or complex; in that case the entries are accessed using the Horner scheme: For a 3-dimensional tensor `a` with dimensions (m, n, p), the entry a_{ijk} with $0 \leq i < m$, $0 \leq j < n$, $0 \leq k < p$, the index location is computed by $(i * n + j) * p + k$.
- sparse and uniform data type storage form: each nonzero entry is stored together with its index information as a set of integers;
- mixed data type storage form: each nonzero entry is stored together with its index information, it's data type, and it's (starting) address, the latter as a set of integers.

For the sparse and mixed data type storage the tensor entries together with the additional information are sorted w.r.t. to ascending index locations for faster access.

If tensor **A** has only numeric values, the decision between dense and sparse storage form is made like for matrices using the setting of the runtime option `SPRANGE=`, which is by default set to .5 and means if the tensor contains more than 50 % zeros it is stored and treated as sparse. As in vectors and matrices, names can be assigned for each of the dimensions using the `dname` function.

4 Data Lists

4.1 Syntax for Lists

Before using a list the name must be defined as a list name by the keyword statements:

$$\texttt{list}\ list\_name[irange] <, list\_name2[irange2], \ldots >;$$

whereas the *list_name* specifies the name of the list and the *irange* must be an integer defining the upper index range, i.e. the number of list entries. Note, when the *list_name* was used in CMAT code before this definition, the former object is free'd by default and its content is lost, even if the name was already defined as a list.

After a name was defined as a list, the list entries can be defined in the usual vector notation:

$$list\_name[index] = name\_of\_data\_object$$

.

List entries can be either scalars, vectors, matrices, tensors, structs, or lists (and so permitting the functionality of more than one-dimensional lists). Note, currently only one index can be used, i.e. lists are vectors of objects. Undefined list entries can be referred to as missing values. List entries can be used in arithmetic operations, as function arguments, and in `return();` statements of user functions.

4.2 Simple Example for Using Lists

Simple Example: List of three vectors:

```
list blist[3];
blist[1] = [ 10. 10. ];
blist[2] = [ 20. 20. 20. ];
```

```
  blist[3] = [ 30. 30. 30. 30. ];
  print "Blist=", blist;


Blist=
                      *************************
                      List blist with 3 Entries
                      *************************


                               blist[1]
                               ********

                      Dense Row Vector (ncol=2)

                      R |           1          2
                            10.000000  10.000000


                               blist[2]
                               ********

                      Dense Row Vector (ncol=3)

                  R |           1          2          3
                         20.000000  20.000000  20.000000


                               blist[3]
                               ********

                      Dense Row Vector (ncol=4)

            R |           1          2          3          4
                  30.000000  30.000000  30.000000  30.000000
```

The technical report: *Tensor and List Operations in CMAT* illustrates connections between list and tensor operations.

4.3 Data Lists and Structs

Lists and structs can be members of lists and structs:

```
struct str1;
list lst1;

str1.a = -15; str1.b = 27;
lst1[1] = str1.a; lst1[2] = str1.b;
lst1[1] += lst1[1]; lst1[2] += lst1[2];
print "List1=", lst1;
```

```
List1=

       *************************
       lst1 (List with 2 Entries)
       *************************

            lst1[1]:     -30
            lst1[2]:      54
```

```
struct str1;
list lst1;

lst1[1] = [ 1 2 3 ]; lst1[2] = [ 3 2 1 ];
str1.a = lst1[1]; str1.b = lst1[2];
print "Struct1=", str1;
```

```
Struct1=

       *************************
       Struct str1 with 2 Entries
       *************************

      Struct str1 Entry[1]=str1.b:
      ****************************

        Dense Row Vector str1.b

        R |       1      2      3
                  3      2      1


      Struct str1 Entry[2]=str1.a:
      ****************************

        Dense Row Vector str1.a

        R |       1      2      3
                  1      2      3
```

List names can be arguments of user specified functions:

```
print " User Function with list argument";  This should be 15:  15
 function tff1(lst1) {
    a = lst1[1]; b = lst1[2];
    k = (a < b) ? -a : -b;
    a = a + b;
    return(k);
 }

 list lst1;
 lst1[1] = -15; lst1[2] = 27;
 c = tff1(lst1);
 print " This should be 15: ",c;
```

List names can be returned by user specified functions:

```
print "User Function returns with List";    This should be 15:  15
 function tff3(a,b) {
    list lst;
    k = (a < b) ? -a : -b;
    a = a + b;
    lst[1] = k; lst[2] = a; lst[3] = b;
    return(lst);
 }

 lst = tff3(-15,27);
 print "List result=", lst;
 print " This should be 15: ",lst[1];
```

5 Data Structs

5.1 Syntax for Structs

Before using a struct, the name must be declared as a struct_name by the following keyword statement:

`struct` *struct_name* <, *struct_name*_2, . . . >;

Note, when the *struct_name* was used in CMAT code before this definition, the former object is freed by default and its content is lost, even if the name was already defined as a struct. Structs which are entries of structs do not have to be defined by the `struct` declaration.

Struct entries can be either scalars, vectors, matrices, tensors, lists, or structs.

After a name was defined as a struct, a struct entry can be defined in the

notation:

$$struct\_name.entry\_name = name\_of\_data\_object$$

.

or

$$struct\_name.entry\_name[irange] = name\_of\_data\_object$$

.

Note, the *entry_name* and the *name_of_data_object* can be the same but must not necessarily be so. Struct entries can be used in arithmetic operations.

5.2 Simple Examples for Applying Structs

Here we assign a scalar `b` as an entry of struct `a`:

```
struct a;                              [1] Struct a=
a.b = 5;
print "[1] Struct a=", a;                     **************************
                                              Structure a with 1 Entries
                                              **************************

                                          Struct a Entry[1]=a.b:       5
```

This is how we access the entry `b` of struct `a`:

```
b = a.b;                               Entry b= 5
print "Entry b=",b;                    Struct Entry 2 * a.b= 10.000
a.b *= 2.;
print "Struct Entry 2 * a.b=",a.b;
```

Make matrix `c` to another entry of struct `a`:

```
c = [ 2. 1.,
      1. 2.];
a.c = c;
print "[2] Struct a=", a;
```

```
[2] Struct a=
      [1] Struct a with 2 Entries
      ***************************

         Struct a Entry[1]=a.c:
         **********************

        Dense Symmetric Matrix a.c

         S |           1          2
         --------------------------
         1 |   2.0000000
         2 |   1.0000000  2.0000000

      Struct a Entry[2]=a.b:  10.0000000
```

After assigning struct `d` to an entry, the struct `a` now has three entries:

```
struct d;
d.ent = [ 1 2 3 ];
a.d = d;
print "[3] Struct a=", a;
lstmem(1);
lststk(1);
```

```
[3] Struct a=
      [2] Struct a with 3 Entries
      ***************************

     [1] Struct a.d with 1 Entries
     *****************************

      Struct a.d Entry[1]=a.d.ent:
      ****************************

         Dense Row Vector d.ent

        R |       1      2      3
                  1      2      3

         Struct a Entry[2]=a.c:
         **********************

        Dense Symmetric Matrix a.c

         S |           1          2
         --------------------------
         1 |   2.0000000
         2 |   1.0000000  2.0000000

      Struct a Entry[3]=a.b:  10.0000000
```

Here we assign a vector `a.d.f` as a second entry of substruct `a.d`:

```
a.d.f = [ 3 2 1 ];
print "[4] Struct a=", a;
```

```
[4] Struct a=
      [3] Struct a with 3 Entries
      ***************************

     [1] Struct a.d with 2 Entries
     *****************************

       Struct a.d Entry[1]=a.d.f:
       **************************

         Dense Row Vector a.d.f

        R |       1      2      3
                  3      2      1


     Struct a.d Entry[2]=a.d.ent:
     ****************************

        Dense Row Vector d.ent

        R |       1      2      3
                  1      2      3


       Struct a Entry[2]=a.c:
       **********************

      Dense Symmetric Matrix a.c

       S |           1          2
       --------------------------
       1 |   2.0000000
       2 |   1.0000000  2.0000000


   Struct a Entry[3]=a.b:  10.0000000
```

```
print "[5] Struct a=", a;
```

```
[5] Struct a=
      [3] Struct a with 4 Entries
      ***************************

     [1] Struct a.d with 2 Entries
     ****************************

       Struct a.d Entry[1]=a.d.f:
       *************************

         Dense Row Vector a.d.f

        R |       1      2      3
                  3      2      1


       Struct a.d Entry[2]=a.d.ent:
       ****************************

         Dense Row Vector d.ent

        R |       1      2      3
                  1      2      3


        Struct a Entry[2]=a.cb:
        ***********************

       Dense Symmetric Matrix a.cb

        S |           1          2
        --------------------------
        1 |   20.000000
        2 |   10.000000  20.000000


         Struct a Entry[3]=a.c:
         **********************

       Dense Symmetric Matrix a.c

        S |           1          2
        --------------------------
        1 |   2.0000000
        2 |   1.0000000  2.0000000


     Struct a Entry[4]=a.b:  10.0000000
```

The following show operations with struct entries:

```
cb =  a.c * a.b;
print "cb", cb;
a.cb = a.c * a.b;
print "Entry a.cb=", a.cb;
```

```
cb
 S |         1         2
-------------------------
 1 |    20.000
 2 |   10.0000    20.000

Entry a.cb=
 S |         1         2
-------------------------
 1 |    20.000
 2 |   10.0000    20.000
```

```
free a.d;
print "a after freeing a.d=", a;
```

```
a after freeing a.d=
      [4] Struct a with 3 Entries
      ***************************


        Struct a Entry[1]=a.cb:
        ***********************

      Dense Symmetric Matrix a.cb

        S |           1          2
        --------------------------
        1 |   20.000000
        2 |   10.000000  20.000000



         Struct a Entry[2]=a.c:
         **********************

       Dense Symmetric Matrix a.c

        S |           1          2
        --------------------------
        1 |   2.0000000
        2 |   1.0000000  2.0000000


     Struct a Entry[3]=a.b:  10.0000000
```

5.3 Structs and User Specified Functions

Structs can be arguments of user specified functions:

```
print " User Function with struct argument";This should be 15:  15
 function tff2(str1) {
    a = str1.a; b = str1.b;
    k = (a < b) ? -a : -b;
    a = a + b;
    return(k);
 }

 struct str1;
 str1.a = -15; str1.b = 27;

 c = tff2(str1);
 print " This should be 15: ",c;
```

Structs can be returned by user specified functions:

```
print "User Function returns with Struct";  This should be 15:  15
 function tff4(a,b) {
    struct str;
    k = (a < b) ? -a : -b;
    a = a + b;
    str.k = k; str.a = a; str.b = b;
    return(str);
 }

 str = tff4(-15,27);
 lstvar(1);
 lstmem(1);
 print "Struct result=", str;
 print " This should be 15: ",str.k;
```

6 Additional Remarks

7 The Bibliography

References

[1] Hartmann, W. (2007), "Tensor and List Operations in CMAT", Technical Report CMAT, Heidelberg.

[2] Hartmann, W. (2006), "CMAT - Tutorial", Technical Report CMAT, Heidelberg.

[3] Hartmann, W. (2006), "CMAT - User Manual", Technical Report CMAT, Heidelberg.