CMAT Newsletter: July 2007

Wolfgang M. Hartmann

July 2007

Contents

1 General Remarks

Lots of debugging and bug fixing has happened during the last few months for a stable downloadable CMAT release 4 in July 2007. The language was extended to multidimensional arrays, some tensor operations, and data lists. Data lists are indexed arrays of objects like scalars, vectors, matrices, and tensors. Note, that dense and sparse multidimensional arrays (tensors) are tested, but not so much the mixed data type, like string data. The very important `loc()` function, which returns index locations of specific entries in a vector, matrix, or tensor.

1.1 New Functions

The following new functions are implemented:

const the tensor (multidimensional) extension of the `cons` function,

covshr computes two forms of shrinkage covariance matrix by Ledoit & Wolf (2003),

dim returns the size of dimensions of a data object like vector, matrix, tensor, list

loc returns index locations for entries satisfying specified properties

mat2ten moves a list of matrices to a tensor

randt the tensor (multidimensional) extension of the `rand` function

ten2mat moves a tensor to a list of matrices

ten2vec moves a tensor to a single vector

vec2ten moves a single vector to a tensor

tenperm returns tensor with permuted dimension

tentvec returns product of tensor with vector

tentmat returns product of tensor with vector

tentten returns product of tensor with vector

1.2 Fixed Bugs

Some rather incredible bugs were fixed again:

1. Bug when multiplying diagonal times symmetric matrix was fixed. Sorry, this was a bad one.
2. Scalar product of sparse left vector with dense right vector was fixed. Also a very bad bug.

2 Modifications of Features

2.1 Two Types of Index Processing in Matrices

The following situations may occur for accessing entries of a matrix `a[m,n]`:

1. there are two scalar indices, e.g. i and j: `a[i,j]` refers to only one entry in row i and column j of matrix `a`
2. there is one scalar i and one k vector v index: e.g. `a[i,v]` refers to k entries in row i and the k columns specified by the entries of v
3. there are two vector indices, e.g. u and v, where u has k entries and v has l entries: `a[u,v]` then refers to $k * l$ entries in in the rows specified by entries of u and the columns specified by the entries of v
4. there is only one index: this must be either
   - one 2-vector index v: `a[v]` refers to one entry in the row specified by $v[1]$ and the column specified by $v[2]$ or
   - one 2-column matrix m with k rows: `a[m]` refers to k entries in the rows specified by the first column of m and the columns specified by the second column of m

Here are a few examples:

1. Use of `loc` function to obtain 2-column index matrix:

```
a = [ 2.  3.  4.  5.  6.,
      4.  4.  5.  6.  7.,
      0.  3.  6.  7.  8.,
      0.  0.  2.  8.  9.,
      0.  0.  0.  1. 10. ];
ind = loc(a);
print "Ind=", ind;
```

```
Ind=
    |   1   2
------------
  1 |   1   1
  2 |   1   2
  3 |   1   3
  4 |   1   4
  5 |   1   5
  6 |   2   1
  7 |   2   2
  8 |   2   3
  9 |   2   4
 10 |   2   5
 11 |   3   2
 12 |   3   3
 13 |   3   4
 14 |   3   5
 15 |   4   3
 16 |   4   4
 17 |   4   5
 18 |   5   4
 19 |   5   5
```

Note, that `c` is a vector and `d` is the same as `a`:

```
c = a[ind];
print "C=", c;
d[ind] = c;
print "d=", d;
```

```
C=
   |          1
---------------
 1 |     2.0000
 2 |     3.0000
 3 |     4.0000
 4 |     5.0000
 5 |     6.0000
 6 |     4.0000
 7 |     4.0000
 8 |     5.0000
 9 |     6.0000
10 |     7.0000
11 |     3.0000
12 |     6.0000
13 |     7.0000
14 |     8.0000
15 |     2.0000
16 |     8.0000
17 |     9.0000
18 |    1.00000
19 |    10.0000
```

```
d=
   |          1          2          3          4          5
-------------------------------------------------------------
 1 |     2.0000     3.0000     4.0000     5.0000     6.0000
 2 |     4.0000     4.0000     5.0000     6.0000     7.0000
 3 |    0.00000     3.0000     6.0000     7.0000     8.0000
 4 |    0.00000    0.00000     2.0000     8.0000     9.0000
 5 |    0.00000    0.00000    0.00000    1.00000    10.0000
```

2. Difference between resetting obejcts and reusing them:

   This copies objects after freeing old content:

```
a = [ 1 2 3 ,                    x,y,z=
      2 3 1 ,                     S |   1   2   3
      3 1 2 ];                   ----------------
b = [ 1 2 3 ,                     1 |   1
      0 2 3 ,                     2 |   2   3
      0 0 3 ];                    3 |   3   1   2
c = [ 1 0 0 ,
      2 3 0 ,                     U |   1   2   3
      1 2 3 ];                   ----------------
                                  1 |   1   2   3
                                  2 |   0   2   3
x = y = z = cons(5,5);            3 |   0   0   3
x = a;
y = b;                            L |   1   2   3
z = c;                           ----------------
print "x,y,z=", x,y,z;            1 |   1   0   0
                                  2 |   2   3   0
                                  3 |   1   2   3
```

This copies objects inside old content:

```
x = y = z = cons(5,5);
x[1,1] = a;
y[1,1] = b;
z[1,1] = c;
print "x,y,z=", x,y,z;
```

```
x,y,z=

 S |   1   2   3   4   5
-----------------------
 1 |   1
 2 |   2   3
 3 |   3   1   2
 4 |   0   0   0   0
 5 |   0   0   0   0   0


 U |   1   2   3   4   5
-----------------------
 1 |   1   2   3   0   0
 2 |   0   2   3   0   0
 3 |   0   0   3   0   0
 4 |   0   0   0   0   0
 5 |   0   0   0   0   0

 L |   1   2   3   4   5
-----------------------
 1 |   1   0   0   0   0
 2 |   2   3   0   0   0
 3 |   1   2   3   0   0
 4 |   0   0   0   0   0
 5 |   0   0   0   0   0
```

Special properties are no longer valid:

```
x = y = z = cons(5,5);
x[2,1] = a;
y[2,1] = b;
z[2,1] = c;
print "x,y,z=", x,y,z;
```

```
x,y,z=
   |   1   2   3   4   5
-------------------------
 1 |   0   0   0   0   0
 2 |   1   2   3   0   0
 3 |   2   3   1   0   0
 4 |   3   1   2   0   0
 5 |   0   0   0   0   0


   |   1   2   3   4   5
-------------------------
 1 |   0   0   0   0   0
 2 |   1   2   3   0   0
 3 |   0   2   3   0   0
 4 |   0   0   3   0   0
 5 |   0   0   0   0   0

 L |   1   2   3   4   5
-------------------------
 1 |   0   0   0   0   0
 2 |   1   0   0   0   0
 3 |   2   3   0   0   0
 4 |   1   2   3   0   0
 5 |   0   0   0   0   0
```

3. Use of 2-column index matrix on left side:

   Loading with 2 column index matrix: (Note, matrix is extended to 6 columns)

```
a = [ 1 2 3 ,
      2 3 1 ,
      3 1 2 ];
b = [ 1 2 3 ,
      0 2 3 ,
      0 0 3 ];
c = [ 1 0 0 ,
      2 3 0 ,
      1 2 3 ];
d = [ 3. 5. 7. ];
x = y = z = cons(5,5);
ind = [ 1 2 ,
        3 4 ,
        5 6 ];
x[ind] = a;
y[ind] = b;
z[ind] = c;
print "x,y,z=", x,y,z;
```

```
x,y,z=
 U |   1   2   3   4   5   6
---------------------------
 1 |   0   1   0   0   0   0
 2 |   0   0   0   0   0   0
 3 |   0   0   0   2   0   0
 4 |   0   0   0   0   0   0
 5 |   0   0   0   0   0   3

 U |   1   2   3   4   5   6
---------------------------
 1 |   0   1   0   0   0   0
 2 |   0   0   0   0   0   0
 3 |   0   0   0   2   0   0
 4 |   0   0   0   0   0   0
 5 |   0   0   0   0   0   3

 U |   1   2   3   4   5   6
---------------------------
 1 |   0   1   0   0   0   0
 2 |   0   0   0   0   0   0
 3 |   0   0   0   0   0   0
 4 |   0   0   0   0   0   0
 5 |   0   0   0   0   0   0
```

```
print "Duplicate setting at same location";
d = [ 3. 5. 7. ];
x = cons(5,5);
ind = [ 1 2 ,
        3 4 ,
        3 4 ];
x[ind] = d;
print "x=", x;
```

```
Duplicate setting at same location

x=
 U |         1         2         3         4         5
---------------------------------------------------------
 1 |   0.00000    3.0000   0.00000   0.00000   0.00000
 2 |         0   0.00000   0.00000   0.00000   0.00000
 3 |         0         0   0.00000    7.0000   0.00000
 4 |         0         0         0   0.00000   0.00000
 5 |         0         0         0         0   0.00000
```

Note that all indices must be of type integer.

2.2 Extension of Language and Functions for Tensors

All subscript processing was extended to multidimensional arrays (tensors). This also includes all subscript reduction operations. Also simple (unitary and binary) arithmetic operations were extended to processing tensors.
The following functions were extended to processing tensors:

attrib

lstmem

lststk

lstvar

The following input illustrates the `attrib` function:

```
/* set the seed */
srand(1);
/* create random atens[4,4,4] */
nind  = cons(3,1,4);
atens = randt(nind,"duni");
```

```
  print "Constant Tensor=", atens;
  att = attrib(atens);
  att = attrib(atens,"nop");
  print "Attributes=",att;


Attributes=

                              att
                              ***

                        Sparse Matrix att

                 1          2           3
 1 Object Name     name       atens
 2 Object Type     otyp      tensor
 3 Data Type       dtyp        real
 4 Storage Type    styp   dens_full
 5 N Dimension     ndim      3.0000
 6 Dimension Names dnam      0.0000
 7 Variable Names  vnam      0.0000
 8 Size in Bytes   size    708.0000
 9 String Length   slen      0.0000
10 Number Strings  nstr      0.0000
11 Number MissVals nmis      0.0000
12 Sum Dimension   nvar     12.0000
13 Number Entries  nent     64.0000
14 Number NonzeroV nzer     64.0000
15 Smallest Value  vmin      0.0227
16 Largest Value   vmax      0.9768
17 Absolute Min    amin      0.0227
18 Absolute Max    amax      0.9768
19 Frobenius Norm  nrm2     21.0685
```

2.3 Extension to Various Functions

2.3.1 Extensions to `max()` and `min()` Function

The `max()` and `min()` functions have been implemented to accept two input arguments, like those standard functions in C. Now, they have been extended to accept one input argument and can now return the maximum or minimum value of a vector or matrix argument. The same and more general results were already available using the subscript reduction operators <> and ><.

```
  q = [ 0. 1. 1. 3. 5. -3. ];
  print "Min=", qmin = min(q);
```

```
  print "Max=", qmax = max(q);


Min=-3.0000
Max= 5.0000
```

The `max()` and `min()` functions can also be applied to tensors:

```
  /* set the seed */
  srand(1);
  /* create random atens[4,4,4] */
  nind  = cons(3,1,4);
  atens = randt(nind,"duni");
  print "Atens=", atens;

  amin = min(atens);
  print "amin=", amin;
  amax = max(atens);
  print "amax=", amax;
```

2.3.2 Extensions to `scad()` Function

1. For $nvar > Nobs$ the SCAD algorithm is a two stage algorithm where
   (a) the first stage reduces the number of variables from $n = nvar$ to $N = Nobs$ and
   (b) the second stage reduces the number of variables from $n = nvar$ to $p = nsel$ satisfying the error fraction criterion or the `minsel` crterion.

   As in the R program by Benner et al. the first stage utilized a trick referred to Hastie & Tibshirani (2004) to analyze the smaller $N \times N$ matrix $\mathbf{V}$ of the singular value decomposition of the $N \times n$ data matrix $\mathbf{X}$

   $$\mathbf{X} = \mathbf{UDV}^T = \mathbf{RV}^T.$$

   The first stage of the Benner et al. SCAD algorithm uses $\mathbf{R}$ instead of $\mathbf{X}$ and it is not entirely clear which amount of information is being lost by imposing the svd trick in the intialization phase.

   A better alternative to this svd trick was implemented by using the Sherman, Morrison, Woodbury formula to solve the large $n \times n$ linear system. This algorithm is now the default, however, the svd version can still be executed when specifying the *"nosmw"* option.

2. When $nvar > Nobs$ and the SMW version is used, computing sentitivity values (asymptotic standard errors) makes not much sense. A minimum percentage of the remaining variables can be zeroed out by specifying the `"perzer"` option.

3. The `"minsel"` option was added to specify a minimum number of variables that is returned. The algorithm will try to return the specified number of variable coefficients even if the specified error fraction would prevent maintaining the *samll* coefficients. This value is only valid for the second stage of the algorithm. The specified integer value must be smaller than the number of observations.

2.3.3 Extensions to `smsvm()` Function

1. For zeroing out small entries in the θ estimates the `"abszer"` and `"relzer"` option were added. The value for the `"relzer"` option refers to the maximum value of $|\theta|$, i.e. all coefficients θ_j which are smaller than $relzer * max_{j=1}^{n}|\theta_j|$ are set to zero. Defaults are 1.e-6 for both options. Now, the returned p vector θ may be sparse.

2.3.4 Extensions to `svm()` Function

1. In the past scaling was only applied to predictor variables. Now, when there is an interval response y for SV regression and the predictors are scaled, the response y also is scaled. It is recommended, however, that if the user wants to analyse scaled variables, he should do so before calling the function. That will save some computing resources.

2. For the two methods developed by Fung and Mangasarian (2002, 2003), L1FS and L2FS, for linear kernel now two alternative algorithms are implemented depending on the shape of the data matrix **A**. The original algorithms were developed for small number N (number of rows) of observations but large number n (number of columns) of variables. Using the Sherman, Morrison, and Woodbury theorem (Golub & Van Loan 1989, p. 51), the linear system can now also be solved for a symteric $(n+1)\times(n+1)$ coefficient matrix when the number N of observations is much larger than the number of variables. Of course this is only a modification for computational efficiency, and the results are the same for both algorithm versions, even if they would be applied to the less favorite situation.

3. The L1FS method (Fung and Mangasarian, 2003) depends on the settings of three parameters $(\alpha, \delta, \epsilon)$, in addition to the common regularization parameter C. Currently the implementation permits a grid search for determing the best parameter pair (α, δ). In the past the selection of the best solution was always based only on the training data set. Now, if an input test data set is specified and the `"tunsel"` is neither specified or specified to `"test"`, the selection criterion is the number misclassification based on the input test data set. For tied misclassifications of the test set the number of misclassifications of the training set is used conditionally. This should yield much better, especially since the L1FS method is mostly used for feature selection. (The old results can still can still be obtained by specifying the "tunsel" option to `"train"`.)

```
print "Heart" data: train: Nobs=210, test: Nobs=60; nvar=14";
data = rspfile("..\\tdata\\heart_210.dat");
test = rspfile("..\\tdata\\heart_60.dat");
modl = "1 = 2:14";
class = 1;

/*--- L1FS: with direct Cholesky: without line search ---*/
optn = [ "print"           2 ,
         "pplan"             ,
         "meth"       "l1fs" ,
         "peneps"       .001 ,
         "c"              1. ,
         "kern"       "line" ];
< alfa,sres,vres,yptr > = svm(data,modl,optn,class,.,.,.,test);


                     *****************
                     Model Information
                     *****************

                Number Valid Observations  210
                Observations Test Data      60
                Response Variable          Y[1]
                N Independend Variables     13
                Support Vector C Classification
                Model Without Linear Constraint
                Estimation Method          L1F
                Without Linesearch
                Kernel Function         Linear
                Use Unscaled Predictor


                   ***********************
                   Class Level Information
                   ***********************

 Class  Level    Value

 Y[ 1]      2       -1       1
         Memory needed for Kernel matrix: 0.169029 Mb
    Traindata stored incore: Mem need: 0.021 Mem spec: 2 Mb


            ************************************
            Number of Observations for Class Levels
```

```
**************************************

Variable       Value    Nobs  Proportion
Y[ 1]             -1     117   55.714286
                   1      93   44.285714



**************************************
Number of Observations for Class Levels
**************************************

Variable       Value    Nobs  Proportion
Y[ 1]             -1      33   55.000000
                   1      27   45.000000


        ---Start Training Cycle: Technique=  L1FS---

    Delta      Alpha  Criterion    MisTrn    MisTst Nzero Niter
1.00e-005  0.0010000 -0.0910620        32        11    12    26
1.00e-004  0.0010000 -0.0910573        31        11    12     9
0.0010000  0.0010000 -0.0909080        30        11    10     6
0.0100000  0.0010000 -0.0902644        31        11    11    28
1.00e-004  0.0100000 -0.0788475        29        12    12    24
0.0010000  0.0100000 -0.0789188        30        11    12     9
0.0100000  0.0100000 -0.0782817        30        11    12    18
0.0010000  0.1000000 -0.0734461        28        11    12     8
0.0100000  0.1000000 -0.0729414        29        12    12    13
0.0010000  1.0000000 -0.0726166        31        11    12     9
0.0100000  1.0000000 -0.0721732        30        11    12    15
0.0010000  10.000000 -0.0724986        30        11    12    10
0.0100000  10.000000 -0.0719793        29        11    12    13
0.0010000  100.00000 -0.0724890        30        11    12    10
0.0100000  100.00000 -0.0719699        29        11    12    13
0.0010000  1000.0000 -0.0724881        30        11    12    10
0.0100000  1000.0000 -0.0719689        29        11    12    13


       Misclassification (Test) for (alpha,delta) (C=1)
       ********************************************

                   Dense Matrix (7 by 5)

        |     d1e-005    d0.0001     d0.001      d0.01       d0.1
       ------------------------------------------------------------
```

a0.001	11.000000	11.000000	11.000000	11.000000	.
a0.01	.	12.000000	11.000000	11.000000	.
a0.1	.	.	11.000000	12.000000	.
a1	.	.	11.000000	11.000000	.
a10	.	.	11.000000	11.000000	.
a100	.	.	11.000000	11.000000	.
a1000	.	.	11.000000	11.000000	.

SSE (Test) for (alpha,delta) (C=1)
****

Dense Matrix (7 by 5)

	d1e-005	d0.0001	d0.001	d0.01	d0.1
a0.001	38.763909	38.965273	38.614912	38.785754	.
a0.01	.	56.948388	56.838485	56.397757	.
a0.1	.	.	94.735406	94.749821	.
a1	.	.	99.211167	101.64249	.
a10	.	.	102.77536	102.17875	.
a100	.	.	102.88414	102.30214	.
a1000	.	.	102.89504	102.31446	.

Misclassification (Training) for (alpha,delta) (C=1)
****

Dense Matrix (7 by 5)

	d1e-005	d0.0001	d0.001	d0.01	d0.1
a0.001	32.000000	31.000000	30.000000	31.000000	.
a0.01	.	29.000000	30.000000	30.000000	.
a0.1	.	.	28.000000	29.000000	.
a1	.	.	31.000000	30.000000	.
a10	.	.	30.000000	29.000000	.
a100	.	.	30.000000	29.000000	.
a1000	.	.	30.000000	29.000000	.

SSE (Training) for (alpha,delta) (C=1)
****

Dense Matrix (7 by 5)

	d1e-005	d0.0001	d0.001	d0.01	d0.1
a0.001	108.35788	108.72552	109.29293	108.60979	.
a0.01	.	167.61718	170.14322	165.21032	.
a0.1	.	.	303.31159	285.35631	.
a1	.	.	312.63559	313.78602	.
a10	.	.	308.85144	310.83635	.
a100	.	.	309.07152	311.21173	.
a1000	.	.	309.09356	311.24914	.

C=1 : Solution for delta=0.001 alpha=0.1 selected based on Test fit.

Linear Separating Plane (w*x = 1.08059)

Dense Row Vector (ncol=14)

R	1	2	3	4	5
	0.0000000	0.3016042	0.7149163	0.4295983	0.8585302

R	6	7	8	9	10
	-0.2174930	0.1876488	-0.7865211	0.2926560	0.7228086

R	11	12	13	14
	0.1363463	0.9297882	0.5695462	1.0805885

Largest 12 Plane Coefficients (Sorted)

1	12	X13	0.929788178
2	5	X6	0.858530184
3	8	X9	-0.786521133
4	10	X11	0.722808613
5	3	X4	0.714916323
6	13	X14	0.569546221
7	4	X5	0.429598302
8	2	X3	0.301604225
9	9	X10	0.292656017
10	6	X7	-0.217493025
11	7	X8	0.187648770
12	11	X12	0.136346281

```
Sparsity: 12 Nonzeros 1 Zeros (C=1)
```

4. As an alternative to the direct Cholesky decomposition of an $N \times N$ kernel matrix the conjugate gradient algorithm was implemented for the "L1FS" method. For linear kernel that saves core storage of an $N \times N$ kernel matrix when the $N \times n$ data matrix can be stored incore. The L1 estimation method obtains sparse parameter vectors and has better test validation than the L2 methods.

   For the NIR Spectra data set we have a training set with $N = 21$ and $n = 268$. The interval variable 269 is transformed into categorical respose by using the quartiles.

```
  options NOECHO;
#include "..\\tdata\\nir.dat"
  options ECHO;
  nrtrn = nrow(xtrn); nc = ncol(xtrn);
  nrtst = nrow(xtst);
  print "nrtrn,nctrn=",nrtrn,nc;

  sopt = [ "qu1" "med" "qu3" ];
  mom = univar(ytrn‘,sopt);
  /* oder: [1] median, [2]: quart1, [3]: quart3 */
  print mom;


  /* print "Moments of y=", mom; */
  for (j = 1; j <= nrtrn; j++)
  y[j] = (ytrn[j] <= mom[2]) ? 1
       : (ytrn[j] <= mom[1]) ? 2
       : (ytrn[j] <= mom[3]) ? 3 : 4;
  print "ytrn=", y;
  cdat = xtrn -> y; /* attrib(cdat); */


  free y;
  for (j = 1; j <= nrtst; j++)
  y[j] = (ytst[j] <= mom[2]) ? 1
       : (ytst[j] <= mom[1]) ? 2
       : (ytst[j] <= mom[3]) ? 3 : 4;
  print "ytst=", y;
  test = xtst -> y; /* attrib(test); */


  /* print cdat[ , 200:269 ]; */
```

```
/* model: */
xind = [ 1:268 ]; yind = 269;
modl = "269 = 1:268";
class = 269;
```

The following are the quartiles of variable 269:

```
           |         1
----------------------
  Median   |    20.760
  Quart_1  |   0.00000
  Quart_3  |    50.595
```

The following are the values of the categorical response variable:

```
     y = [ 4  4  4  4  4  4  3  3  3  3
           2  2  2  2  2  1  1  1  1  1  1 ];
```

We specify conjugate gradient solver with no more than 100 iterations:

```
modl = "269 = 1:268";
optn = [ "print"           2 ,
         "pplan"             ,
         "meth"       "l1fs" ,
         "icg"               ,
         "cgit"         100 ,
         "lines"             ,
         "c"            100. ,
         "peneps"         .1 ,
         "kern"       "line" ];
alpha = svm(cdat,modl,optn,class);
```

Here, the last column shows the average number of conjugate gradient iterations which becomes smaller for larger values of the α parameter. Note, Mangasarian & Thompson (2006) fix $\alpha = 1$, $\delta = .001$, and $\epsilon = .0001$, different from Fung & Mangsarian (2003), which recommend cross validation for (α, δ) and recommend the larger penalty parameter $\epsilon = .1$ for increased sparsity.

```
            ****************************************
            ---Fitting Response Category  1 vs. Rest---
```

```
        ****************************************

 Niter      Delta      Alpha  Criterion  Misclass    AvCgIt
    14  1.00e-005  0.0010000 -18.582332         0     54.86
    18  1.00e-004  0.0010000 -18.582277         0     52.89
    23  0.0010000  0.0010000 -18.580621         0     29.17
    12  1.00e-005  0.0100000 -7.1960648         0     36.67
    11  1.00e-004  0.0100000 -7.1960435         0     37.73
    15  0.0010000  0.0100000 -7.1956540         0     31.20
    14  1.00e-005  0.1000000 -3.3223165         0     28.36
    15  1.00e-004  0.1000000 -3.3223165         0     27.47
    13  0.0010000  0.1000000 -3.3222107         0     25.54
    50  0.0100000  0.1000000 -3.3217396         0     17.18
    24  1.00e-005  1.0000000 -1.6613663         5     24.96
    17  1.00e-004  1.0000000 -1.6613279         5     23.71
    15  0.0010000  1.0000000 -1.6612623         5     22.20
    42  0.0100000  1.0000000 -1.6608582         5     14.76
    25  1.00e-005  10.000000 -1.2787268         5     20.80
    21  1.00e-004  10.000000 -1.2787268         5     17.19
    20  0.0010000  10.000000 -1.2787268         5     17.00
    28  0.0100000  10.000000 -1.2782794         5     12.86
    33  1.00e-005  100.00000 -1.2322646         5     17.61
    47  1.00e-004  100.00000 -1.2322646         5     16.06
    35  0.0010000  100.00000 -1.2322646         5     16.34
    29  0.0100000  100.00000 -1.2322646         5     16.14


        Misclassification for (alpha,delta) (C=100)
        ****************************************

                    Dense Matrix (7 by 5)

       |     d1e-005    d0.0001     d0.001      d0.01       d0.1
     ------------------------------------------------------------
a0.001 |   0.0000000  0.0000000  0.0000000     .          .
 a0.01 |   0.0000000  0.0000000  0.0000000     .          .
  a0.1 |   0.0000000  0.0000000  0.0000000  0.0000000     .
    a1 |   5.0000000  5.0000000  5.0000000  5.0000000     .
   a10 |   5.0000000  5.0000000  5.0000000  5.0000000     .
  a100 |   5.0000000  5.0000000  5.0000000  5.0000000     .
 a1000 |      .          .          .          .          .

  C=100 : Solution for delta=1e-005 alpha=0.001 selected.
```

The best L1 solution for category 1 has 53 nonzero coefficients.

```
*****************************************
---Fitting Response Category  2 vs. Rest---
*****************************************
```

Niter	Delta	Alpha	Criterion	Misclass	AvCgIt
8	1.00e-005	0.0010000	-77.142505	2	50.38
9	1.00e-004	0.0010000	-77.142485	2	50.56
19	0.0010000	0.0010000	-77.142192	2	39.32
83	0.0100000	0.0010000	-77.141716	2	21.06
12	1.00e-005	0.0100000	-65.639075	1	50.33
10	1.00e-004	0.0100000	-65.639074	1	50.00
13	0.0010000	0.0100000	-65.638259	1	42.31
60	0.0100000	0.0100000	-65.638042	1	22.07
12	1.00e-005	0.1000000	-41.992299	8	37.83
13	1.00e-004	0.1000000	-41.992237	8	39.23
12	0.0010000	0.1000000	-41.991156	8	37.00
21	1.00e-005	1.0000000	-24.702470	9	29.52
25	1.00e-004	1.0000000	-24.702469	9	31.12
11	0.0010000	1.0000000	-24.702287	9	30.82
65	0.0100000	1.0000000	-24.702388	10	24.37
19	1.00e-005	10.000000	-20.154229	9	34.00
23	1.00e-004	10.000000	-20.154229	9	35.48
11	0.0010000	10.000000	-20.154112	9	30.45
57	0.0100000	10.000000	-20.154159	9	24.53
18	1.00e-005	100.00000	-19.629549	9	30.11
28	1.00e-004	100.00000	-19.629548	9	29.93
11	0.0010000	100.00000	-19.629275	9	28.82
60	0.0100000	100.00000	-19.629485	9	24.62
30	1.00e-005	1000.0000	-19.576319	9	28.27
28	1.00e-004	1000.0000	-19.576318	9	28.32
30	0.0010000	1000.0000	-19.576035	9	27.53
77	0.0100000	1000.0000	-19.576255	9	25.96

```
Misclassification for (alpha,delta) (C=100)
*******************************************
```

Dense Matrix (7 by 5)

	d1e-005	d0.0001	d0.001	d0.01	d0.1
a0.001	2.0000000	2.0000000	2.0000000	2.0000000	.
a0.01	1.0000000	1.0000000	1.0000000	1.0000000	.
a0.1	8.0000000	8.0000000	8.0000000	.	.
a1	9.0000000	9.0000000	9.0000000	10.000000	.
a10	9.0000000	9.0000000	9.0000000	9.0000000	.

```
 a100 |   9.0000000  9.0000000  9.0000000  9.0000000     .
a1000 |   9.0000000  9.0000000  9.0000000  9.0000000     .

 C=100 : Solution for delta=1e-005 alpha=0.01 selected.
```

The best L1 solution for category 2 has 99 nonzero coefficients.

```
        ****************************************
        ---Fitting Response Category  3 vs. Rest---
        ****************************************

Niter      Delta      Alpha  Criterion  Misclass    AvCgIt
    8  1.00e-005  0.0010000 -50.995378         0     50.88
    9  1.00e-004  0.0010000 -50.995374         0     47.44
   19  0.0010000  0.0010000 -50.994930         0     34.74
   89  0.0100000  0.0010000 -50.994248         0     20.82
    9  1.00e-005  0.0100000 -38.920745         0     48.67
   11  1.00e-004  0.0100000 -38.920585         0     47.00
   14  0.0010000  0.0100000 -38.920060         0     42.79
   65  0.0100000  0.0100000 -38.919394         0     20.03
   14  1.00e-005  0.1000000 -25.596262         7     37.21
   13  1.00e-004  0.1000000 -25.596261         7     39.15
   11  0.0010000  0.1000000 -25.596182         7     35.27
   49  0.0100000  0.1000000 -25.595630         7     23.10
   16  1.00e-005  1.0000000 -18.437301         7     26.81
   17  1.00e-004  1.0000000 -18.437301         7     26.82
   11  0.0010000  1.0000000 -18.437297         7     31.36
   48  0.0100000  1.0000000 -18.436920         7     23.54
   20  1.00e-005  10.000000 -17.131584         7     27.40
   20  1.00e-004  10.000000 -17.131584         7     28.85
    9  0.0010000  10.000000 -17.131475         7     30.78
   56  0.0100000  10.000000 -17.131156         7     22.48
   28  1.00e-005  100.00000 -16.991181         7     28.46
   29  1.00e-004  100.00000 -16.991172         7     27.66
   27  0.0010000  100.00000 -16.991064         7     29.81
   64  0.0100000  100.00000 -16.990808         7     22.80
   40  1.00e-005  1000.0000 -16.976833         7     26.55
   48  1.00e-004  1000.0000 -16.976833         7     29.52
   40  0.0010000  1000.0000 -16.976739         7     26.80
   83  0.0100000  1000.0000 -16.976482         7     23.37


        Misclassification for (alpha,delta) (C=100)
        ****************************************
```

```
                    Dense Matrix (7 by 5)

        |     d1e-005    d0.0001     d0.001      d0.01       d0.1
     ------------------------------------------------------------
 a0.001 |   0.0000000  0.0000000  0.0000000  0.0000000     .
  a0.01 |   0.0000000  0.0000000  0.0000000  0.0000000     .
   a0.1 |   7.0000000  7.0000000  7.0000000  7.0000000     .
     a1 |   7.0000000  7.0000000  7.0000000  7.0000000     .
    a10 |   7.0000000  7.0000000  7.0000000  7.0000000     .
   a100 |   7.0000000  7.0000000  7.0000000  7.0000000     .
  a1000 |   7.0000000  7.0000000  7.0000000  7.0000000     .

  C=100 : Solution for delta=1e-005 alpha=0.001 selected.
```

The best L1 solution for category 3 has 85 nonzero coefficients.

```
         ****************************************
         ---Fitting Response Category  4 vs. Rest---
         ****************************************

 Niter      Delta      Alpha  Criterion  Misclass    AvCgIt
    10  1.00e-005  0.0010000 -20.175593         0     48.00
     9  1.00e-004  0.0010000 -20.175422         0     43.00
    13  0.0010000  0.0010000 -20.173754         0     30.62
     9  1.00e-005  0.0100000 -14.395198         0     40.89
     9  1.00e-004  0.0100000 -14.395198         0     42.67
     9  0.0010000  0.0100000 -14.395169         0     32.56
    72  0.0100000  0.0100000 -14.394509         0     14.83
    13  1.00e-005  0.1000000 -5.6381085         0     34.85
    13  1.00e-004  0.1000000 -5.6381083         0     34.92
    15  0.0010000  0.1000000 -5.6369844         0     30.73
    70  0.0100000  0.1000000 -5.6369954         0     15.70
    17  1.00e-005  1.0000000 -2.2621917         4     25.29
    20  1.00e-004  1.0000000 -2.2621145         4     23.45
    21  0.0010000  1.0000000 -2.2621785         4     20.00
    75  0.0100000  1.0000000 -2.2620689         4     12.76
    22  1.00e-005  10.000000 -1.5862177         4     19.59
    19  1.00e-004  10.000000 -1.5862100         4     22.79
    21  0.0010000  10.000000 -1.5861926         4     18.71
    49  0.0100000  10.000000 -1.5862082         4     13.29
    25  1.00e-005  100.00000 -1.5168558         4     17.88
    29  1.00e-004  100.00000 -1.5168554         4     20.52
    20  0.0010000  100.00000 -1.5166643         4     17.65
    52  0.0100000  100.00000 -1.5168438         4     12.31
    29  1.00e-005  1000.0000 -1.5099002         4     17.28
```

```
   33  1.00e-004  1000.0000 -1.5098959         4     17.09
   32  0.0010000  1000.0000 -1.5093277         4     16.16
   60  0.0100000  1000.0000 -1.5098934         4     12.67


      Misclassification for (alpha,delta) (C=100)
      ***************************************

                Dense Matrix (7 by 5)

       |     d1e-005    d0.0001     d0.001      d0.01       d0.1
     ---------------------------------------------------------
a0.001 |   0.0000000  0.0000000  0.0000000     .           .
 a0.01 |   0.0000000  0.0000000  0.0000000  0.0000000     .
  a0.1 |   0.0000000  0.0000000  0.0000000  0.0000000     .
    a1 |   4.0000000  4.0000000  4.0000000  4.0000000     .
   a10 |   4.0000000  4.0000000  4.0000000  4.0000000     .
  a100 |   4.0000000  4.0000000  4.0000000  4.0000000     .
 a1000 |   4.0000000  4.0000000  4.0000000  4.0000000     .

  C=100 : Solution for delta=1e-005 alpha=0.001 selected.
```

The best L1 solution for category 4 has 70 nonzero coefficients. For all four response categories we obtain the following result where an observation observed with category 3 was misclassified by the model with category 2.

```
                  Classification Table
                  --------------------

         |    Predicted
Observed |       1       2       3       4  TotSum  OffSum
---------|-----------------------------------------------
       1 |       6       0       0       0       6       0
       2 |       0       5       0       0       5       0
       3 |       0       1       3       0       4       1
       4 |       0       0       0       6       6       0
  TotSum |       6       6       3       6      21
  OffSum |       0       1       0       0               1


Regularization Parameter C . . . . . . . . . . . . . . . . 100
Kernel Function. . . . . . . . . . . . . . . . . .        Linear
Norm of Longest Vector . . . . . . . . . . . . . . . . 31.9706
Number Misclassifactions (Training Data) . . . . . . . . . . 1
Number Fast Runs through TrainData . . . . . . . . . .  133782
```

```
Number Slow Runs through TrainData . . . . . 18371
Total Number of Kernel Calls . . . . . 442
Time for Optimization. . . . . 105
Total Processing Time. . . . . 105


            Results for Response Categories
            ******************************

                 Dense Matrix (4 by 12)

  |   N_Sup_Vec  N_SV_Marg  OptimCrit    L1_Loss  Grad_Norm
-------------------------------------------------------------
1 |   21.000000  2.0000000 -18.582332     .       3.94e-006
2 |   21.000000  5.0000000 -65.639075     .       2.16e-006
3 |   21.000000  6.0000000 -50.995378     .       7.66e-007
4 |   21.000000  0.0000000 -20.175593     .       3.26e-007

  |   Beta(PCE)  SV_Radius       Bias   WGT_Norm     Margin
-------------------------------------------------------------
1 |   0.5381051  31.954952 -0.9585232  360.62296  0.1053182
2 |  -3.8833832  31.954952  8.1650325  1207.8136  0.0575480
3 |   7.4880735  31.954952 -14.849561  1053.5110  0.0616184
4 |  -12.086528  31.954952  24.196683  592.22746  0.0821837

  |      VC_Dim  Mis_Train
--------------------------
1 |   368239.93  0.0000000
2 |   1233322.4  1.0000000
3 |   1075761.0  0.0000000
4 |   604735.67  0.0000000
```

The output ends with a table of the 307 nonzero coefficients for all four categories which for space reasons is not shown here.

5. The new method "L1LIN" was implemented: this is similar to the Mangasarian & Thompson (2006) linear L1 chunking method which is based on both, the Fung & Mangasarian unconstrained L1 method and the Bradley & Mangasarian (1998) LP chunking algorithm. The method generates generaly sparse parameter vectors and is efficient for both, very large number of observations and large number of variables. When the $N \times n$ data matrix fits incore and conjugate gradient solver is used, only $O(N)$ or $O(n)$ and no $O(N^2)$ or $O(n^2)$ memory is needed. If a direct Cholesky solver is used the matrix of the linear system is of size $O(K^2)$, where K is the chunk size which is a proportion of the number of observations plus a set of additional observations satisfying some linear constraints.

3 New Developments

3.1 Function t = const($nsiz <, val >$)

t = const(nsiz<,val>)

Purpose: The function t = const($nsiz <, val >$) is a tensor extension of the matrix function m = cons($nr <, nc, val, type >$). Assuming the tensor should have d dimensions the input vector nind should specify d integers as the upper index ranges of each dimension. The upper ranges of the d dimensions should only be the assumed lower bounds, using statements the ranges can still be increased later, by just defining tensor entries outside the specified ranges.

Input: **nsiz** should be a vector of d int values, defining the sizes of the d dimensions of the tensor.

val is the value with which the tensor entries are intialized. The argument is optional and the default is zero.

Output: The tensor t with the specified size containing entries with the specified value.

Restrictions: 1. The first argument cannot contain any missing values.

2.

Relationships: randt()

Examples:

```
nsiz  = cons(3,1,4);    /* cons(nr,nc,val) */
atens = const(nsiz,.9);
print "Constant Tensor=", atens;
```

```
*****************************
Tensor atens with 3 Dimensions
*****************************


                atens_1
                *******

            Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |  0.9000000  0.9000000  0.9000000  0.9000000
2 |  0.9000000  0.9000000  0.9000000  0.9000000
3 |  0.9000000  0.9000000  0.9000000  0.9000000
4 |  0.9000000  0.9000000  0.9000000  0.9000000
```

```
                    atens_2
                    *******

             Dense Matrix (4 by 4)

  |          1          2          3          4
--------------------------------------------------
1 |  0.9000000  0.9000000  0.9000000  0.9000000
2 |  0.9000000  0.9000000  0.9000000  0.9000000
3 |  0.9000000  0.9000000  0.9000000  0.9000000
4 |  0.9000000  0.9000000  0.9000000  0.9000000


                    atens_3
                    *******

             Dense Matrix (4 by 4)

  |          1          2          3          4
--------------------------------------------------
1 |  0.9000000  0.9000000  0.9000000  0.9000000
2 |  0.9000000  0.9000000  0.9000000  0.9000000
3 |  0.9000000  0.9000000  0.9000000  0.9000000
4 |  0.9000000  0.9000000  0.9000000  0.9000000


                    atens_4
                    *******

             Dense Matrix (4 by 4)

  |          1          2          3          4
--------------------------------------------------
1 |  0.9000000  0.9000000  0.9000000  0.9000000
2 |  0.9000000  0.9000000  0.9000000  0.9000000
3 |  0.9000000  0.9000000  0.9000000  0.9000000
4 |  0.9000000  0.9000000  0.9000000  0.9000000
```

3.2 Function `covshr`

< cov,dlta,fcov,scov,smu > = covshr(xdat,"smet",<,par<,rind>>)

Purpose: The function `covshr` computes two forms of robust covariance matrices by the method of shrinkage

1. Ledoit & Wolf, 2003 a: shrinkage method where the target matrix is based by Sharpe's single-index (single-factor) model. Here, an

additional vector of market returns of the index could be supplied. If the fourth argument is not specified it is computed as the mean (across variables/columns) of the mean corrected data.

2. Ledoit & Wolf, 2003 b: shrinkage method where the target matrix is based on the constant correlation model.
3. Shrinking toward a diagonal model.
4. Shrinking toward a 2-parameter model, where one parameter defines the constant diagonal and the other defines the constant off-diagonal entry.

The $n \times n$ shrinkage covariance matrix $\mathbf{C}$ is defined as

$$\mathbf{C} = \delta\mathbf{F} + (1 - \delta)\mathbf{S}$$

where $\mathbf{S}$ is the sample covariance matrix, $\mathbf{F}$ is a target covariance matrix, and $\delta \in [0, 1]$ is the shrinkage constant. Note, that both shrinkage methods are designed for typical applications, where the number (T) of observations is much smaller than the number (N) of variables (stocks).

The EM algorithm for estimating the mean vector and sample covariance matrix is the same as with the `emcov` matrix and may have problems to converge.

Input: xdat contains a $N \times n$ matrix of real data which may contain missing values. For financial data the N rows correspond to (T) measurements and the n columns to (N) securities.

par The second argument is a string sapecifying the shrinkage method:

"corr" average correlation method

"mark" single-factor market model

"diag" diagonal method

"twop" 2 parameter method (diagonal and offdiagonal).

par The third argument is an optional parameter vector `par` specifying some details of the implementation. Most entries are the same as for the `emcov` function. Currently only the first 6 entries of this vector are relevant:

par[1 =imet] specifies what is returned: $imet = 0$ covariance matrix and mean vector by substitution of the (pairwise) mean $imet = 1$ covariance matrix and mean vector by listwise deletion $imet = 2$ covariance matrix and mean vector by pairwise deletion $imet = 3$ EM estimation of covariance matrix and mean vector

par[2 =icor] for $icor = 0$ a covariance matrix, for $icor \neq 0$ a correlation matrix is returned; $icor = 0$ is the default;

par[3 =vardiv] specifies the variance divisor when a covariance matrix is returned

par[4 =ipri] specifies whether information about the iterative EM algorithm (iteration history) is printed. Since the algorithm may have convergence problems, this may show some of the problems.

par[5 =eps] specifies a criterion for the termination of the EM algorithm. The iteration process is terminated when the maximum change of any entry of the covariance matrix is smaller than $eps > 0$.

par[6 =imax] specifies the maximum number of iterations. Default is `imax=500`.

rind For the market model the fourth argument can be specified specifying a vector of the market returns of a single index, then shrinkage method 1 (Ledoit & Wolf, 2003a) is being used. The size of this vector must be the same as the number of rows (T) of the first argument.

Output: **cov** the shrinkage covariance matrix **C**

dlta the shrinkage constant δ

fcov the target covariance matrix **F**

scov the sample covariance matrix **S**

smu the sample mean vector μ

Restrictions: 1. The input data `xdat` may contain missing values but no string or complex data.

2. Missing values in the data should be *missing completely at random* (MCAR).

Relationships: emcov()

Examples: 1. Average correlation model:

```
  print "NIR Spectra data set: train: nr=21, test: nr=7: nvar=268";
  options NOECHO;
#include "..\\tdata\\nir.dat"
  options ECHO;
  nr = nrow(xtrn); nc = ncol(xtrn);
  print "nrtrn,nctrn=",nr,nc;


  < cov,dlta,fcov,scov,smu > = covshr(xtrn,"corr");


Delta= 0.1227
```

Only the upper left 20×20 part of the large covariance matrix is shown here;

Shrink Cov=

S	1	2	3	4	5
1	0.00355				
2	0.00544	0.01151			
3	0.00774	0.01599	0.02862		
4	0.00934	0.02031	0.03412	0.05128	
5	0.01010	0.02263	0.03892	0.05402	0.07100
6	0.01049	0.02363	0.04109	0.05784	0.06987
7	0.01076	0.02391	0.04169	0.05925	0.07270
8	0.01086	0.02350	0.04084	0.05851	0.07298
9	0.01089	0.02274	0.03923	0.05662	0.07189
10	0.01111	0.02230	0.03811	0.05536	0.07157
11	0.01155	0.02230	0.03769	0.05509	0.07248
12	0.01209	0.02251	0.03764	0.05532	0.07397
13	0.01269	0.02297	0.03805	0.05615	0.07602
14	0.01331	0.02368	0.03897	0.05762	0.07857
15	0.01386	0.02430	0.03975	0.05885	0.08073
16	0.01417	0.02426	0.03932	0.05839	0.08095
17	0.01419	0.02357	0.03778	0.05638	0.07929
18	0.01398	0.02275	0.03620	0.05427	0.07718
19	0.01367	0.02220	0.03536	0.05308	0.07568
20	0.01327	0.02188	0.03511	0.05265	0.07462

S	6	7	8	9	10
6	0.08565				
7	0.08207	0.09829			
8	0.08412	0.09320	0.11096		
9	0.08477	0.09627	0.10632	0.12789	
10	0.08631	0.10038	0.11353	0.12622	0.15536
11	0.08929	0.10612	0.12254	0.13880	0.15679
12	0.09291	0.11253	0.13225	0.15210	0.17391
13	0.09689	0.11898	0.14159	0.16457	0.18972
14	0.10099	0.12501	0.14981	0.17515	0.20286
15	0.10448	0.13016	0.15685	0.18424	0.21416
16	0.10600	0.13346	0.16230	0.19207	0.22453
17	0.10544	0.13454	0.16546	0.19757	0.23250
18	0.10381	0.13375	0.16580	0.19919	0.23544
19	0.10204	0.13172	0.16351	0.19662	0.23254
20	0.09996	0.12831	0.15854	0.18992	0.22395

S	11	12	13	14	15
11	0.19604				

```
12 |   0.19838   0.24748
13 |   0.21778   0.24675   0.30210
14 |   0.23369   0.26550   0.29434   0.35142
15 |   0.24740   0.28167   0.31274   0.33783   0.39785
16 |   0.26049   0.29753   0.33106   0.35810   0.38162
17 |   0.27106   0.31071   0.34654   0.37535   0.40044
18 |   0.27535   0.31634   0.35331   0.38294   0.40874
19 |   0.27205   0.31259   0.34912   0.37833   0.40371
20 |   0.26139   0.29980   0.33439   0.36201   0.38593

 S |        16        17        18        19        20
---------------------------------------------------------
16 |   0.44874
17 |   0.42627   0.49646
18 |   0.43547   0.45888   0.51952
19 |   0.43002   0.45314   0.46382   0.50706
20 |   0.41063   0.43231   0.44239   0.43738   0.46236
```

2. Market model (single index):

```
< cov,dlta,fcov,scov,smu > = covshr(xtrn,"mark");

Delta= 0.04803
```

3. Diagonal model:

```
< cov,dlta,fcov,scov,smu > = covshr(xtrn,"diag");

Delta= 0.06186
```

4. Two parameter model:

```
< cov,dlta,fcov,scov,smu > = covshr(xtrn,"twop");

Delta= 0.09662
```

3.3 Function `dim`

nd = dim(a<,ind>)

Purpose: The function `dim` returns either:

1. A vector of the size of the dimensions of a data object: For vectors and matrices it returns a 2-vector with the number of rows and columns. For tensors `dim` returns a vector with more than 2 integers.
2. An integer of the size of a specific dimension which is specified by the second input argument.

This function is similar to the `size` function in Matlab.

Input: 1. The first argument of function `dim` specifies a data object like scalar, vector, matrix, or tensor.

2. The second input argument is optional. If it is used it should be an integer within the scope of the dimensionality of the first argument.

Output: The function `dim` has only one return which is either an integer vector or an integer scalar or a missing value.

Restrictions: 1. There are no obvious restrictions for the input objct `a`.

2. Note, the second input argument depends on the specified index base, which is by default equal to one, but maybe changed to zero by the runtime option.

Relationships: nrow(), ncol()

Examples: 1. Vector and Matrix:

```
brv = [ 1 2 3 ];                          Dim of row vector=
dbr = dim(brv);                              |   1   2
print "Dim of row vector=",dbr;             ------------
                                             1 |   1   3
bcv = [ 1, 2, 3 ];
dbc = dim(bcv);                           Dim of col vector=
print "Dim of col vector=",dbc;              |   1   2
                                            ------------
bmat = [ 100. 200. 300.,                     1 |   3   1
         300. 200. 100. ];
db2 = dim(bmat);                          Dim of matrix=
print "Dim of matrix=",db2;                  |   1   2
                                            ------------
                                             1 |   2   3
```

2. Tensor: Use of `dim` without second input argument:

```
real B1[3,4,5];                           Dimension B1=
print "B1[3,4,5]=", B1;                      |   1   2   3
n1 = dim(B1);                               ---------------
print "Dimension B1=", n1;                   1 |   3   4   5
```

3. Tensor: Use of `dim` with second input argument:

```
real B2[3,4,5] = 99.99;                    Number columns of B2= 5
print "B2[3,4,5]=", B2;
n3 = dim(B2,3);
print "Number columns of B2=", n3;
```

3.4 Function `dimlabel`

b=dimlabel(a,lab) | lab=dimlabel(a)

Purpose: There are two versions of this function:

b=dimlabel(a,lab) two input arguments and no return: The returned object `b` is identical with input `a` except that a list of vectors `lab` with label strings is assigned to its dimensions. If the second argument is a missing value, any existing dimension labels of `a` are removed.

lab=dimlabel(a) one input argument and one result: Returns a missing value if the object `a` has no dimension labels assigned, otherwise it returns a list of vectors with dimension labels of the object `a`.

Input: b=dimlabel(a,lab) The first argument is a tensor, matrix or vector `a`. The second argument is a list of vectors `lab` of strings which should be used as labels of the dimensions of `a`.

lab=dimlabel(a) The only input argument is a tensor, matrix, or vector `a`.

Output: b=dimlabel(a,lab) The returned object `b` is identical with the input object `a` except that the list of vectors `lab` of labels is assigned to its dimensions.

lab=dimlabel(a) Returns a missing value if the dimensions of the object `a` have no labels assigned, otherwise it returns the list of vectors of dimension labels of the object `a`.

Restrictions: 1. The second input argument must be a vector of strings that does not contain any numeric or missing values.

2. The number of strings in the second argument must match the number of columns of the first input argument.

Relationships: dimname(), rname(), cname(), rlabel(), clabel()

Examples: See the `dimname` function below.

3.5 Function `dimname`

b=dimname(a,nam)	nam=dimname(a)

Purpose: There are two versions of this function:

b=dimname(a,nam) two input arguments and no return: The returned object `b` is identical with input `a` except that the list of vectors `nam` of names is assigned to its dimensions. If the second argument is a missing value, any existing dimension names of `a` are removed.

nam=dimname(a) one input argument and one result: Returns a missing value if the dimensions of the object `a` have no names assigned, otherwise it returns the vector of dimension names of the object `a`.

Input: b=dimname(a,nam) The first argument is a tensor, matrix, or vector `a`. The second argument is a list of vectors `nam` of strings which should be used as names of the dimensios of `a`.

nam=dimname(a) The only input argument is a tensor, matrix, or vector `a`.

Output: b=name(a,nam) The returned object `b` is identical with input `a` except that the list of vectors `nam` of names is assigned to its dimensions.

nam=name(a) Returns a missing value if the dimensions of object `a` have no names assigned, otherwise it returns the list of vectors of dimension names of the object `a`.

Restrictions: 1. The second input argument must be a list of vectors of strings that does not contain any numeric or missing values.

2. The number of strings in the list of vectors in second argument must match the size of dimensions of the first input argument.

Relationships: dimlabel(), rname(), cname(), clabel(), rlabel().

Examples: 1. This easy examples illustrates how to create a list of variable names, how to attach it to dhe dimensions of a tensor, and how to move it from a tensor back into a list of string vectors.

```
srand(1);
nind  = [ 2 3 4 ];
atens = randt(nind,"duni");

list tnam[3];
tnam[1] = [ "time1" "time2" ];
tnam[2] = [ "row1" "row2" "row3" ];
tnam[3] = [ "col1" "col2" "col3" "col4" ];
print "List tnam=",tnam;
```

```
List tnam=
                        ************************
                        List tnam with 3 Entries
                        ************************


                                tnam[1]
                                *******

                           Dense Row Vector (ncol=2)

                              R |        1       2
                                      time1   time2


                                tnam[2]
                                *******

                           Dense Row Vector (ncol=3)

                            R |       1      2      3
                                    row1   row2   row3


                                tnam[3]
                                *******

                           Dense Row Vector (ncol=4)

                         R |       1      2      3      4
                                 col1   col2   col3   col4


  ctens = dimname(atens,tnam);
  print "Ctens=",ctens;


Ctens=
                       ******************************
                       Tensor ctens with 3 Dimensions
                       ******************************


                                ctens_1
                                *******

                            Dense Matrix (3 by 4)
```

```
          |        col1       col2       col3       col4
     -------------------------------------------------
row1 |   0.1849626  0.9700887  0.3998243  0.2593986
row2 |   0.9216026  0.9692773  0.5429792  0.5316917
row3 |   0.0497940  0.0665666  0.8193186  0.5238705


                        ctens_2
                        *******

                    Dense Matrix (3 by 4)

          |        col1       col2       col3       col4
     -------------------------------------------------
row1 |   0.8533943  0.0671846  0.9570239  0.2971940
row2 |   0.2726118  0.6899296  0.9767649  0.2265075
row3 |   0.6882366  0.4127639  0.5585541  0.2872256
```

```
/* get the list back: should be the same as tnam */
vnam = dimname(ctens);
print "Original names=",vnam;
```

```
Original names=
                      ************************
                      List vnam with 3 Entries
                      ************************


                             vnam[1]
                             *******

                    Dense Column Vector (nrow=2)

                        C |        1       2
                               time1   time2


                             vnam[2]
                             *******

                    Dense Column Vector (nrow=3)

                      C |       1      2      3
                             row1   row2   row3
```

```
                    vnam[3]
                    *******

          Dense Column Vector (nrow=4)

       C |       1      2      3      4
              col1   col2   col3   col4
```

3.6 Function `loc`

indx = loc(a<,crit<,val>>)

Purpose:

Input: The function `loc` finds all entries in input object `a` which satisfy a condition specified by the second and third input argument. The following conditions may be specified:

Criterion	Meaning
`"nonz"`	find locations of all nonzeros (includes locations of missing value) this is the default and must not be specified
`"zero"`	find locations of all zeros
`"miss"`	find locations of all missing values
`"nznm"`	find locations of all nonzeros and nonmissings
`"great"`	find locations of all entries in `a` which are larger than a specified third argument
`"small"`	find locations of all entries in `a` which are smaller than a specified third argument
`"equal"`	find locations of all entries in `a` which are equal to a specified third argument
`"unequ"`	find locations of all entries in `a` which are unequal to a specified third argument

The second and third argument must not be specified for the `"nonz"` condition. The third argument must not be specified for the `"nonz"`, `"zero"`, `"miss"`, `"nznm"` conditions. If the third argument is not be specified for the `"great"`, `"small"`, `"equal"`, `"nonequ"` conditions, a numeric value of zero is used as the default. If the third argument is specified as a string, the `"great"` and `"small"` conditions concern only the length of the string.

Output: The function `loc` returns with `indx` either

1. a misssing value if there are no entries in the object `a` which satify the specified condition or

2. a scalar index if there is only one entry in `a` which satisfies the condition or
3. a vector of K indices, when $K > 1$ entries of `a` satisfy the condition and the input object `a` is a vector, or
4. a $K \times 2$ matrix of row and column indices, when $K > 1$ entries of `a` satisfy the condition and the input object `a` is a matrix.

Restrictions: 1. There are no obvious restrictions for the input objct `a`.

2. Note, that the result depends on the specified index base, which is by default equal one, but maybe changed to zero by the `INDBASE=` runtime option.

Relationships:

Examples: 1. Some simple examples:

```
w = [ -1. 0. 0. 1. 2. 3. . ];
ind1 = loc(w);
print "ind1=", ind1;
```

```
ind1=
   |   1
--------
 1 |   1
 2 |   4
 3 |   5
 4 |   6
 5 |   7
```

```
ind2 = loc(w,"nznm");
print "ind2=", ind2;
```

```
ind2=
   |   1
--------
 1 |   1
 2 |   4
 3 |   5
 4 |   6
```

```
ind3 = loc(w,"zero");
print "ind3=", ind3;
```

```
ind3=
   |   1
--------
 1 |   2
 2 |   3
```

```
ind4 = loc(w,"miss",2.);
print "ind4=", ind4;
```

```
ind4=
   |   1
--------
 1 |   7
```

```
ind5 = loc(w,"great",2.);
print "ind5=", ind5;
```

```
ind5=
   |   1
--------
 1 |   6
```

```
str = [ "here" "there" "here" "there" ];
ind = loc(str,"equal","here");
print "ind=", ind;
```

```
ind=
   |   1
--------
 1 |   1
 2 |   3
```

```
  a = [ 2.  3.  4.  5.  6.,
        4.  4.  5.  6.  7.,
        0.  3.  6.  7.  8.,
        0.  0.  2.  8.  9.,
        0.  0.  0.  1. 10. ];
  ind = loc(a);
  print "Ind=", ind;
```

```
Ind=
   |   1   2
------------
 1 |   1   1
 2 |   1   2
 3 |   1   3
 4 |   1   4
 5 |   1   5
 6 |   2   1
 7 |   2   2
 8 |   2   3
 9 |   2   4
10 |   2   5
11 |   3   2
12 |   3   3
13 |   3   4
14 |   3   5
15 |   4   3
16 |   4   4
17 |   4   5
18 |   5   4
19 |   5   5
```

2. The `loc` function also applies to tensors:

```
/* set the seed */
srand(1);
/* create random atens[4,4,4] */
nind  = cons(3,1,4);
atens = btens = randt(nind,"duni");
print "Atens=", atens;

bind = loc(atens,"great",.5);
print "loc index vector=", bind;
btens[bind] = 9999.9;
print "Btens=", btens;
```

3.7 Function `mat2ten`

to = mat2ten(ml,nind<,ordr>)

Purpose: The function `mat2ten` can be used to move a list of matrices `ml` to a tensor `to`.

Input:
1. The first input argument refers to a list of K $m \times n$ matrices.
2. The seond input argument specifies the size of the tensor dimensions and should be a vector of integers. The product of these integers should equal the tensor size $K * m * n$.
3. The third (optional) argument specifies an order in which the tensor indices are processed. It should be a vector of integers.

Output: The only return argument is either a tensor, or a missing value, depending whether the processing was succesful.

Restrictions:
1. The second argument `nind` must be a vector of ints specifying the size of the dimension of the matrices and cannot contain missing values.
2. The (optional) third argument `ordr` must be a vector of ints specifying the order of dimension in the tensor and cannot contain missing values.
3. Note, the third input argument depends on the specified index base, which is by default equal to one, but maybe changed to zero by the runtime option.

Relationships: ten2mat()

Examples:

```
nind = [ 4 4 4 ];
dtens = mat2ten(matlst,nind);
print "Tensor made from matrix list=", dtens;
```

```
Tensor made from matrix list=

                    ****************************
                    Tensor dtens with 3 Dimensions
                    ****************************


                              dtens_1
                              *******

                          Dense Matrix (4 by 4)

                |          1          2          3          4
```

```
-----------------------------------------------
1 |   0.1615326  0.7821955  0.8198760  0.1816163
2 |   0.3839244  0.3459368  0.2220190  0.6055288
3 |   0.5259491  0.0853359  0.2895431  0.1075468
4 |   0.7749259  0.6610987  0.1972289  0.2294105


                  dtens_2
                  *******

            Dense Matrix (4 by 4)

  |           1          2          3          4
-----------------------------------------------
1 |   0.3936414  0.5776659  0.2598749  0.1643267
2 |   0.7545403  0.8493966  0.3872302  0.9103450
3 |   0.0151857  0.9133448  0.2087599  0.4406090
4 |   0.0063326  0.3148612  0.7310842  0.1049515


                  dtens_3
                  *******

            Dense Matrix (4 by 4)

  |           1          2          3          4
-----------------------------------------------
1 |   0.7236164  0.4824903  0.8112518  0.4846813
2 |   0.2917406  0.4730618  0.2494313  0.7311393
3 |   0.2807263  0.2316055  0.8921028  0.2311551
4 |   0.4825279  0.7883688  0.8275581  0.9260677


                  dtens_4
                  *******

            Dense Matrix (4 by 4)

  |           1          2          3          4
-----------------------------------------------
1 |   0.0076426  0.3004769  0.6560177  0.0625198
2 |   0.9401413  0.6350539  0.2925110  0.6809179
3 |   0.8969033  0.4506592  0.8169410  0.9702098
4 |   0.1582244  0.5267348  0.0188727  0.5607032
```

3.8 Function `t = randt(`$nsiz <, type, ... >$`)`

`t = randt(`$nsiz <, dist, ... >$`)`

Purpose: The function `t = randt(`$nsiz <, dist, ... >$`)` is a tensor extension of the matrix function `m = rand(`$< nr, nc, type, ... >$`)`. Assuming the tensor should have d dimensions the input vector `nsiz` should specify d integers as the upper index ranges of each dimension. The upper ranges of the d dimensions should only be the assumed lower bounds, using statements the ranges can still be increased later, by just defining tensor entries outside the specified ranges.

Input: **nsiz** should be a vector of d int values, defining the sizes of the d dimensions of the tensor.

dist optional argument specifying the distribution. This argument and the following are the same as for the `rand()` function:

Distr.	Add. Arg.	Description
"icmp"	a, b	uniform RNG, very bad 16 bit version in Watcom C Compiler, int version
"iuni"	a, b	uniform with lower range a and upper range b, int version Mooore, RAND Corporation, see Fishman, p. 605
"iacm"	a, b	uniform random generator by Schrage (1979) in ACM, int version this is not a good choice
"ikis"	a, b	uniform random generator KISS by Marsaglia & Tsang, int version
"iecu"	a, b	Tausworthe uniform random generator by L'Ecuyer (1996), int version
"imwc"	a, b	multiply-with-carry RNG (Marsaglia, 2003), period 2^{128}, int version
"ix128"	a, b	XOR RNG (Marsaglia, 2003), period 2^{128}, int version
"iwow"	a, b	modified XOR RNG (Marsaglia, 2003), period $2^{192} - 2^{32}$, int version
"dcmp"	a, b	uniform with lower range a and upper range b very bad 16 bit version in Watcom C Compiler, real version
"duni"	a, b	uniform with lower range a and upper range b Mooore, RAND Corporation, see Fishman, p. 605, real version
"dacm"	a, b	uniform random generator by Schrage (1979) in ACM, real version this is not a good choice
"dkis"	a, b	uniform random generator KISS by Marsaglia & Tsang, real version
"decu"	a, b	Tausworthe uniform random generator by L'Ecuyer (1996), real version
"dmwc"	a, b	multiply-with-carry RNG (Marsaglia, 2003), period 2^{128}, real version
"dx128"	a, b	XOR RNG (Marsaglia, 2003), period 2^{128}, real version
"dwow"	a, b	modified XOR RNG (Marsaglia, 2003), period $2^{192} - 2^{32}$, real version

Distr.	Add. Arg.	Description
"beta"	α, β	Beta, $\mathcal{BE}(\alpha, \beta)$, with $\alpha > 0$ and $\beta >$
		Randlib version (Brown et al. 1997
"bet2"	α, β	Beta, $\mathcal{BE}(\alpha, \beta)$, with $\alpha > 0$ and $\beta >$
		version by Fishman (1996, [**?**])
"bino"	n, p	Binomial, $\mathcal{B}(n, p)$, with $n = 1, 2, \ldots$
		Randlib version (Brown et al. 1997
"bin2"	n, p	Binomial, $\mathcal{B}(n, p)$, with $n = 1, 2, \ldots$
		version by Fishman (1996, [**?**])
"cau1"	α, β	noncentral Cauchy, $\mathcal{C}(\alpha, \beta)$, with $-$
		version by Fishman p. 192 (1996, [
"cau2"	α, β	noncentral Cauchy, $\mathcal{C}(\alpha, \beta)$, with $-$
		version by Fishman p. 187 (1996, [
"chis"	*df*	chi square with $df > 0$
		Randlib version (Brown et al. 1997
"exex"		Double Exponential, $\mathcal{DE}(\lambda)$,
		version by Fishman p. 192 (1996, [
"expo"	λ	Exponential, $\mathcal{E}(\lambda)$, $\lambda > 0$
		Randlib version (Brown et al. 1997
"exp2"	λ	Exponential, $\mathcal{E}(\lambda)$, $\lambda > 0$
ziggurat method by Marsaglia & Tsang (2000, [**?**])		
"exp3"	λ	Exponential, $\mathcal{E}(\lambda)$, $\lambda > 0$
		version by Fishman p. 192 (1996, [
"exp4"	λ	Exponential, $\mathcal{E}(\lambda)$, $\lambda > 0$
		version by Fishman p. 189 (1996, [
"frch"	α	Fréchet, $\mathcal{FR}(\alpha)$, with $\alpha > 0$
		version by Zielinski(), p.106
"fsnd"	α, β	Snedecor's F, $\mathcal{F}(\alpha, \beta)$, with $\alpha > 0$
		Randlib version (Brown et al. 1997
"fsn2"	α, β	Snedecor's F, $\mathcal{F}(\alpha, \beta)$, with $\alpha > 0$
		version by Fishman p. 208 (1996, [
"gamm"	α, β	Gamma, $\mathcal{G}(\alpha, \beta)$, with $\alpha > 0$ and β
		Randlib version (Brown et al. 1997
"gam2"	α, β	Gamma, $\mathcal{G}(\alpha, \beta)$, with $\alpha > 0$ and β
		version by Fishman p. 193 (1996, [
"geom"	p	Geometric, $\mathcal{G}](p)$, with $0 < p < 1$
"hyge"	α, β, n	Hypergeometric, $\mathcal{H}(\alpha, \beta, n)$ with α
		$n \leq \alpha + \beta$
		version by Fishman p. 218-220 (19
"logn"	μ, σ	Lognormal, $\mathcal{LN}(\mu, \sigma)$, with mean μ
		$\sigma > 0$
		version by Fishman (1996, [**?**])
"ncch"	*df*,*nonc*	noncentral chi square with $df > 0$
		Randlib version (Brown et al. 1997
"ncfs"	α, β,*nonc*	noncentral F, $\mathcal{F}(\alpha, \beta, nonc)$, with α
		Randlib version (Brown et al. 1997
"negb"	r, p	negative Binomial, $\mathcal{NB}(r, p)$, with r
		Randlib version (Brown et al. 1997
"neg2"	r, p	negative Binomial, $\mathcal{NB}(r, p)$, with r
		version by Fishman p. 222 (1996, [
"norm"	μ, σ	Normal, $\mathcal{N}(\mu, \sigma)$, with mean μ and s
		0
		Randlib version (Brown et al. 1997
"nor2"	μ, σ	Normal, $\mathcal{N}(\mu, \sigma)$, with mean μ and s
		0
ziggurat method by Marsaglia & Tsang (2000, [**?**])		

The keywords `"iuni", "iacm", "ikis", "iecu"` generate corresponding uniform integer random numbers in $[0, MACLONG]$.

Output: The tensor `t` with the specified size containing random values as entries.

Restrictions: 1. The first argument cannot contain any missing values.

2.

Relationships: const()

Examples:

```
nind  = cons(3,1,4);    /* cons(nr,nc,val) */
btens = randt(nind,"duni");
print "Random Tensor=", btens;
```

```
*****************************
Tensor btens with 3 Dimensions
*****************************

                btens_1
                *******

            Dense Matrix (4 by 4)

  |          1         2         3         4
-------------------------------------------------
1 |   0.5227359  0.5820759  0.9495240  0.0589558
2 |   0.4789170  0.1752936  0.5748166  0.9191979
3 |   0.5654754  0.8211872  0.0233039  0.2533334
4 |   0.0391460  0.0291636  0.5537057  0.8072438


                btens_2
                *******

            Dense Matrix (4 by 4)

  |          1         2         3         4
-------------------------------------------------
1 |   0.3755272  0.1559292  0.6907556  0.2778315
2 |   0.3591513  0.6645153  0.0457034  0.0892478
3 |   0.3212697  0.5177460  0.6297670  0.0041646
4 |   0.9522087  0.6289332  0.9003093  0.0292697


                btens_3
                *******
```

```
                Dense Matrix (4 by 4)

  |          1          2          3          4
-----------------------------------------------
1 |  0.0436676  0.2648309  0.1362905  0.0320213
2 |  0.5454306  0.8316212  0.9688787  0.9546120
3 |  0.7059240  0.0738252  0.4689559  0.4186246
4 |  0.3660560  0.8317398  0.0331665  0.7558113


                      btens_4
                      *******

                Dense Matrix (4 by 4)

  |          1          2          3          4
-----------------------------------------------
1 |  0.9461433  0.4727213  0.9416930  0.5162474
2 |  0.7402075  0.6150453  0.5592833  0.2504971
3 |  0.5451838  0.7267708  0.6763260  0.4590319
4 |  0.1381715  0.7493522  0.1386659  0.7232725
```

3.9 Function `ten2mat`

ml = ten2mat(ti<,ordr>)

Purpose: The function `ten2mat` can be used to move a tensor to a list of matrices. When prining tensors, tensors are moved to matrix lists by default.

Input: 1. The first input argument should be a tensor `ti`.

2. The second (optional) argument specifies an order in which the tensor indices are processed.

Output: The only return argument is either a list of matrices or a missing value, depending whether the processing was succesful.

Restrictions:

Relationships: mat2ten()

1. The (optional) second argument `ordr` must be a vector of ints specifying the order of dimension in the tensor and cannot contain missing values.

2. Note, the second input argument depends on the specified index base, which is by default equal to one, but maybe changed to zero by the runtime option.

Examples:

```
nind  = cons(3,1,4);    /* cons(nr,nc,val) */
btens = randt(nind,"duni");
print "Random Tensor=", btens;
```

```
Random Tensor=

                ****************************
                Tensor btens with 3 Dimensions
                ****************************

                          btens_1
                          *******

                     Dense Matrix (4 by 4)

  |          1          2          3          4
--------------------------------------------------
1 |  0.1615326  0.7821955  0.8198760  0.1816163
2 |  0.3839244  0.3459368  0.2220190  0.6055288
3 |  0.5259491  0.0853359  0.2895431  0.1075468
4 |  0.7749259  0.6610987  0.1972289  0.2294105


                          btens_2
                          *******

                     Dense Matrix (4 by 4)

  |          1          2          3          4
--------------------------------------------------
1 |  0.3936414  0.5776659  0.2598749  0.1643267
2 |  0.7545403  0.8493966  0.3872302  0.9103450
3 |  0.0151857  0.9133448  0.2087599  0.4406090
4 |  0.0063326  0.3148612  0.7310842  0.1049515


                          btens_3
                          *******

                     Dense Matrix (4 by 4)

  |          1          2          3          4
--------------------------------------------------
1 |  0.7236164  0.4824903  0.8112518  0.4846813
```

```
2 |   0.2917406  0.4730618  0.2494313  0.7311393
3 |   0.2807263  0.2316055  0.8921028  0.2311551
4 |   0.4825279  0.7883688  0.8275581  0.9260677


                    btens_4
                    *******

               Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.0076426  0.3004769  0.6560177  0.0625198
2 |   0.9401413  0.6350539  0.2925110  0.6809179
3 |   0.8969033  0.4506592  0.8169410  0.9702098
4 |   0.1582244  0.5267348  0.0188727  0.5607032


  matlst = ten2mat(btens); /* lstmem(1); */
  print "List of Matrices=", matlst;


List of Matrices=

                  *************************
                  List matlst with 4 Entries
                  *************************


                        matlst[1]
                        *********

                  Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.1615326  0.7821955  0.8198760  0.1816163
2 |   0.3839244  0.3459368  0.2220190  0.6055288
3 |   0.5259491  0.0853359  0.2895431  0.1075468
4 |   0.7749259  0.6610987  0.1972289  0.2294105


                        matlst[2]
                        *********

                  Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
```

```
1 |   0.3936414  0.5776659  0.2598749  0.1643267
2 |   0.7545403  0.8493966  0.3872302  0.9103450
3 |   0.0151857  0.9133448  0.2087599  0.4406090
4 |   0.0063326  0.3148612  0.7310842  0.1049515


                matlst[3]
                *********

           Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.7236164  0.4824903  0.8112518  0.4846813
2 |   0.2917406  0.4730618  0.2494313  0.7311393
3 |   0.2807263  0.2316055  0.8921028  0.2311551
4 |   0.4825279  0.7883688  0.8275581  0.9260677


                matlst[4]
                *********

           Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.0076426  0.3004769  0.6560177  0.0625198
2 |   0.9401413  0.6350539  0.2925110  0.6809179
3 |   0.8969033  0.4506592  0.8169410  0.9702098
4 |   0.1582244  0.5267348  0.0188727  0.5607032
```

3.10 Function `ten2vec`

vec = ten2vec(tens<,ordr>)

Purpose: The `ten2vec` function moves all the entries of a tensor `tens` to a vector `vec`.

Input: **tens** Specifies an existing tensor whose entries should be moved to a vector.

ordr This is an optional permutation vector specifying the order of the dimensions how the entries of the tensor are moved into the vector.

Output: The single return is a vector containing all entries of the tensor. The order of the dimensions maybe specified by the `ordr` vector.

Restrictions: 1. Tensors with string data are not tested yet.

2. The size of the `ordr` vector must be the same as the dimensionality of the tensor, its entries must define a permutation. The integer values depend on the INDBASE option which is by default equal to one.

Relationships: vec2ten(), ten2mat(), mat2ten

Examples: 1. Dense Tensor:

```
srand(1);
nind  = [ 2 3 4 ];
atens = randt(nind,"duni");


print " * Dense Tensor *";
print "Atens=", atens;
```

```
Atens=
                  ****************************
                  Tensor atens with 3 Dimensions
                  ****************************


                            atens_1
                            *******

                       Dense Matrix (3 by 4)

              |          1          2          3          4
            ------------------------------------------------
            1 |  0.1849626  0.9700887  0.3998243  0.2593986
            2 |  0.9216026  0.9692773  0.5429792  0.5316917
            3 |  0.0497940  0.0665666  0.8193186  0.5238705


                            atens_2
                            *******

                       Dense Matrix (3 by 4)

              |          1          2          3          4
            ------------------------------------------------
            1 |  0.8533943  0.0671846  0.9570239  0.2971940
            2 |  0.2726118  0.6899296  0.9767649  0.2265075
            3 |  0.6882366  0.4127639  0.5585541  0.2872256
```

```
cvec = ten2vec(atens);
```

```
  print "cvec=", cvec;
```

```
cvec=
   |         1
--------------
 1 |   0.18496
 2 |   0.97009
 3 |   0.39982
 4 |   0.25940
 5 |   0.92160
 6 |   0.96928
 7 |   0.54298
 8 |   0.53169
 9 |   0.04979
10 |   0.06657
11 |   0.81932
12 |   0.52387
13 |   0.85339
14 |   0.06718
15 |   0.95702
16 |   0.29719
17 |   0.27261
18 |   0.68993
19 |   0.97676
20 |   0.22651
21 |   0.68824
22 |   0.41276
23 |   0.55855
24 |   0.28723
```

```
  dtens = vec2ten(cvec,nind);
  print "dtens should be equal to atens=", dtens;
```

```
                  ****************************
                  Tensor dtens with 3 Dimensions
                  ****************************


                           dtens_1
                           *******

                       Dense Matrix (3 by 4)

             |           1          2          3          4
```

```
-----------------------------------------------
1 |   0.1849626  0.9700887  0.3998243  0.2593986
2 |   0.9216026  0.9692773  0.5429792  0.5316917
3 |   0.0497940  0.0665666  0.8193186  0.5238705


                dtens_2
                *******

           Dense Matrix (3 by 4)

  |          1          2          3          4
-----------------------------------------------
1 |   0.8533943  0.0671846  0.9570239  0.2971940
2 |   0.2726118  0.6899296  0.9767649  0.2265075
3 |   0.6882366  0.4127639  0.5585541  0.2872256
```

2. Sparse Tensor:

```
print " * Sparse Tensor *";
btens = atens > .8;
print "Btens=",btens;
```

```
                ******************************
                Tensor btens with 3 Dimensions
                ******************************


                btens_1
                *******

                Sparse Matrix (3 by 4)

                  |       1      2      3      4
                ---------------------------------
                1 |       0      1      0      0
                2 |       1      1      0      0
                3 |       0      0      1      0


                btens_2
                *******

            Sparse Upper Triangular Matrix (3 by 4)

                U |       1      2      3      4
```

```
                 --------------------------------
                 1 |       1      0      1      0
                 2 |       0      0      1      0
                 3 |       0      0      0      0
```

```
cvec = ten2vec(btens);
print "cvec=", cvec;
```

```
                              cvec
                              ****

                    Sparse Column Vector cvec

      C |       2      5      6     11     13     15     19
                1      1      1      1      1      1      1
```

```
dtens = vec2ten(cvec,nind);
print "dtens should be equal to btens=", dtens;
```

```
                 ******************************
                 Tensor dtens with 3 Dimensions
                 ******************************


                           dtens_1
                           *******

                      Sparse Matrix (3 by 4)

                   |       1      2      3      4
                 --------------------------------
                 1 |       0      1      0      0
                 2 |       1      1      0      0
                 3 |       0      0      1      0


                           dtens_2
                           *******

              Sparse Upper Triangular Matrix (3 by 4)

                 U |       1      2      3      4
                 --------------------------------
                 1 |       1      0      1      0
```

```
2 |       0       0       1       0
3 |       0       0       0       0
```

3.11 Function `vec2ten`

tens = vec2ten(vec,dims)

Purpose: The `vec2ten` function moves all the entries of a a vector `vec` to tensor `tens`.

Input: **vec** Specifies the vector whose entries should be moved to the tensor.

dims Specifies the size of the dimensions and should be a vector of integers. The product of these integers should correspond to the size of the input vector `vec`.

Output: Output is a tensor with dimensions `dims` containing the values of the vector. If the vector has less entries than the tensor size `dims`, the remaining entries of the tensor are set to zero.

Restrictions: 1. Tensors with string data are not tested yet.

2. The size of the vector `vec` and the product of the integers in `dims` should be about the same. If there are less entries in the vector than the size of the tensor defined by `dims` the rest of the tensor is filled with zeros.

Relationships: ten2vec(), ten2mat(), mat2ten

Examples: 1. Dense Vector:

```
srand(1);
nind  = [ 2 3 4 ];
atens = randt(nind,"duni");


print " * Dense Tensor *";
print "Atens=", atens;
```

```
Atens=
                    ******************************
                    Tensor atens with 3 Dimensions
                    ******************************


                              atens_1
                              *******
```

```
                    Dense Matrix (3 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.1849626  0.9700887  0.3998243  0.2593986
2 |   0.9216026  0.9692773  0.5429792  0.5316917
3 |   0.0497940  0.0665666  0.8193186  0.5238705


                         atens_2
                         *******

                    Dense Matrix (3 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.8533943  0.0671846  0.9570239  0.2971940
2 |   0.2726118  0.6899296  0.9767649  0.2265075
3 |   0.6882366  0.4127639  0.5585541  0.2872256
```

```
cvec = ten2vec(atens);
print "cvec=", cvec;
```

```
cvec=
   |         1
--------------
 1 |   0.18496
 2 |   0.97009
 3 |   0.39982
 4 |   0.25940
 5 |   0.92160
 6 |   0.96928
 7 |   0.54298
 8 |   0.53169
 9 |   0.04979
10 |   0.06657
11 |   0.81932
12 |   0.52387
13 |   0.85339
14 |   0.06718
15 |   0.95702
16 |   0.29719
17 |   0.27261
18 |   0.68993
```

```
19 |   0.97676
20 |   0.22651
21 |   0.68824
22 |   0.41276
23 |   0.55855
24 |   0.28723
```

```
dtens = vec2ten(cvec,nind);
print "dtens should be equal to atens=", dtens;
```

```
                ******************************
                Tensor dtens with 3 Dimensions
                ******************************


                          dtens_1
                          *******

                     Dense Matrix (3 by 4)

              |           1          2          3          4
              ------------------------------------------------
              1 |   0.1849626  0.9700887  0.3998243  0.2593986
              2 |   0.9216026  0.9692773  0.5429792  0.5316917
              3 |   0.0497940  0.0665666  0.8193186  0.5238705


                          dtens_2
                          *******

                     Dense Matrix (3 by 4)

              |           1          2          3          4
              ------------------------------------------------
              1 |   0.8533943  0.0671846  0.9570239  0.2971940
              2 |   0.2726118  0.6899296  0.9767649  0.2265075
              3 |   0.6882366  0.4127639  0.5585541  0.2872256
```

2. Sparse Vector:

```
print " * Sparse Tensor *";
btens = atens > .8;
print "Btens=",btens;
```

```
*****************************
Tensor btens with 3 Dimensions
*****************************


                btens_1
                *******

            Sparse Matrix (3 by 4)

              |       1      2      3      4
            ---------------------------------
            1 |       0      1      0      0
            2 |       1      1      0      0
            3 |       0      0      1      0


                btens_2
                *******

        Sparse Upper Triangular Matrix (3 by 4)

            U |       1      2      3      4
            ---------------------------------
            1 |       1      0      1      0
            2 |       0      0      1      0
            3 |       0      0      0      0
```

```
cvec = ten2vec(btens);
print "cvec=", cvec;
```

```
                           cvec
                           ****

                Sparse Column Vector cvec

      C |       2      5      6     11     13     15     19
                1      1      1      1      1      1      1
```

```
dtens = vec2ten(cvec,nind);
print "dtens should be equal to btens=", dtens;
```

```
*****************************
Tensor dtens with 3 Dimensions
```

```
*****************************

           dtens_1
           *******

    Sparse Matrix (3 by 4)

  |       1      2      3      4
-------------------------------
1 |       0      1      0      0
2 |       1      1      0      0
3 |       0      0      1      0

           dtens_2
           *******

Sparse Upper Triangular Matrix (3 by 4)

U |       1      2      3      4
-------------------------------
1 |       1      0      1      0
2 |       0      0      1      0
3 |       0      0      0      0
```

3.12 Function `tenperm`

to = tenperm(ti,ordr)

Purpose: For an input tensor `ti` and a specified order `ordr` the function `tenperm` returns a tensor with permuted dimensions. The function is a generalization of the transpose ‘ operation for matrices and vectors.

Input: 1. The first argument `ti` of function `tenperm` specifies a tensor.

2. The second input argument must be a vector of K integers specifying the order of dimensions for the result tensor `to`.

Output: The function `tenperm` returns a tensor `to` with reorderd dimensions.

Restrictions: 1. There are no obvious restrictions for the input objct `a`.

2. Note, the second input argument depends on the specified index base, which is by default equal to one, but maybe changed to zero by the runtime option.

Relationships: ‘

Examples: 1. :

```
nind  = cons(3,1,4);    /* cons(nr,nc,val) */
btens = randt(nind,"duni");
print "Random Tensor=", btens;
```

```
*****************************
Tensor btens with 3 Dimensions
*****************************


                btens_1
                *******

            Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.5227359  0.5820759  0.9495240  0.0589558
2 |   0.4789170  0.1752936  0.5748166  0.9191979
3 |   0.5654754  0.8211872  0.0233039  0.2533334
4 |   0.0391460  0.0291636  0.5537057  0.8072438

                btens_2
                *******

            Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.3755272  0.1559292  0.6907556  0.2778315
2 |   0.3591513  0.6645153  0.0457034  0.0892478
3 |   0.3212697  0.5177460  0.6297670  0.0041646
4 |   0.9522087  0.6289332  0.9003093  0.0292697

                btens_3
                *******

            Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.0436676  0.2648309  0.1362905  0.0320213
2 |   0.5454306  0.8316212  0.9688787  0.9546120
3 |   0.7059240  0.0738252  0.4689559  0.4186246
4 |   0.3660560  0.8317398  0.0331665  0.7558113
```

```
                    btens_4
                    *******

                Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.9461433  0.4727213  0.9416930  0.5162474
2 |   0.7402075  0.6150453  0.5592833  0.2504971
3 |   0.5451838  0.7267708  0.6763260  0.4590319
4 |   0.1381715  0.7493522  0.1386659  0.7232725
```

```
ordr = [ 3 2 1 ];
ctens = tenperm(btens,ordr);
print "Permuted Indices=",ctens;
```

```
              ******************************
              Tensor ctens with 3 Dimensions
              ******************************

                    ctens_1
                    *******

                Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.5227359  0.3755272  0.0436676  0.9461433
2 |   0.4789170  0.3591513  0.5454306  0.7402075
3 |   0.5654754  0.3212697  0.7059240  0.5451838
4 |   0.0391460  0.9522087  0.3660560  0.1381715


                    ctens_2
                    *******

                Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.5820759  0.1559292  0.2648309  0.4727213
2 |   0.1752936  0.6645153  0.8316212  0.6150453
3 |   0.8211872  0.5177460  0.0738252  0.7267708
4 |   0.0291636  0.6289332  0.8317398  0.7493522


                    ctens_3
                    *******
```

```
                Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |  0.9495240  0.6907556  0.1362905  0.9416930
2 |  0.5748166  0.0457034  0.9688787  0.5592833
3 |  0.0233039  0.6297670  0.4689559  0.6763260
4 |  0.5537057  0.9003093  0.0331665  0.1386659


                     ctens_4
                     *******

                Dense Matrix (4 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |  0.0589558  0.2778315  0.0320213  0.5162474
2 |  0.9191979  0.0892478  0.9546120  0.2504971
3 |  0.2533334  0.0041646  0.4186246  0.4590319
4 |  0.8072438  0.0292697  0.7558113  0.7232725
```

3.13 Function `tentvec`

tens|scal|vec = tentvec(aten,bvec<,n>)

Purpose: The `tentvec` function implements three different ways to multiply a tensor `aten` with one or more vectors `bvec`. Assuming tensor`aten` has dimensionality d with sizes $(m_1, \ldots, m_d)$.

1. n mode multiplication of tensor `aten` with one vector `bvec`: Assuming `bvec` has size m.
   - optional third argument `n` is not specified: `n` is assumed equal to d and the vector size m must be the same as m_d
   - optional third argument `n` is specified: the integer `n` must refer to a dimension of `aten` and the vector size m must be the same as m_n

   The result of the multiplication is a tensor with dimension $d-1$.
2. multiplication of tensor `aten` with a list of d vectors `bvec`: Any specification of n is ignored. The sizes of the d vectors in the list must agree with the tensor sizes $(m_1, \ldots, m_d)$. The result of the multiplication is a scalar.
3. multiplication of tensor `aten` with a list of $d-1$ vectors `bvec`:
   - optional third argument `n` is not specified: `n` is assumed equal to d.

- optional third argument `n` is specified: the integer `n` must refer to a dimension of `aten`.

The sizes of the $d-1$ vectors in the list must agree with the tensor sizes $(m_1, \ldots, m_{n-1}, m_{n+1}, \ldots, m_d)$. The result of the multiplication is a vector with size m_n.

Input: **aten** The first input argument should be a tensor (multidimensional array with $d \geq 3$ dimensions.)

bvec The second input argument must be either a single vector, a list of d vectors or a list of $d-1$ vectors. There are restrictions to the size of the vector(s) as stated above.

n The optional third argument `n` must refer to one of the dimensions of tensor `aten`.

Output: The only result is either a scalar, vector or tensor depending on the form of the second input argument.

Restrictions:
1. All input arguments are numeric and cannot contain string data or missing values.
2. The dimension of the second input argument must be compatible with that of the first argument.
3. Note, that the value of third input argument depends on the specified index base, which is by default equal one, but maybe changed to zero by the `INDBASE=` runtime option.

Relationships: tenttmat(), tentten()

Examples: 1. Multiply tensor with a single vector:

```
srand(1);
nind  = [ 2 3 4 ];
atens = randt(nind,"duni");
print "Random Tensor Atens=", atens;


Random Tensor Atens=

                    ****************************
                    Tensor atens with 3 Dimensions
                    ****************************


                             atens_1
                             *******

                        Dense Matrix (3 by 4)
```

```
  |          1          2          3          4
-------------------------------------------------
1 |   0.1849626  0.9700887  0.3998243  0.2593986
2 |   0.9216026  0.9692773  0.5429792  0.5316917
3 |   0.0497940  0.0665666  0.8193186  0.5238705


                    atens_2
                    *******

                Dense Matrix (3 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   0.8533943  0.0671846  0.9570239  0.2971940
2 |   0.2726118  0.6899296  0.9767649  0.2265075
3 |   0.6882366  0.4127639  0.5585541  0.2872256
```

```
cvec = [ 10. 10. ];
cmat = tentvec(atens,cvec,1);
print "Tensor * Vector (ten1vec) Cmat=",cmat;
```

```
Tensor * Vector (ten1vec) Cmat=
   |         1         2         3         4
-----------------------------------------------
 1 |    10.384    10.373    13.568    5.5659
 2 |    11.942    16.592    15.197    7.5820
 3 |    7.3803    4.7933    13.779    8.1110
```

2. Multiply tensor with a list of d vectors:

```
list blist[3];
blist[1] = [ 10. 10. ];
blist[2] = [ 20. 20. 20. ];
blist[3] = [ 30. 30. 30. 30. ];
cscal = tentvec(atens,blist);
print "Tensor * VectorList (tennvec) Cscal=",cscal;
```

```
Tensor * VectorList (tennvec) Cscal= 75160.59
```

3. Multiply tensor with a list of $d-1$ vectors:

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

```
blist[2] = [ 30. 30. 30. 30. ];
cvec = tentvec(atens,blist,2);
print "Tensor * VectorList (tenmvec) Cvec=",cvec;
```

```
Tensor * VectorList (tenmvec) Cvec=
   |         1
--------------
 1 |    1196.7
 2 |    1539.4
 3 |    1021.9
```

3.14 Function `tentmat`

tens = tentmat(aten,bmat<,n>)

Purpose: The `tentmat` function implements three different ways to multiply a tensor with one or more matrices `bmat`. Assuming tensor`aten` has dimensionality d with sizes $(m_1, \ldots, m_d)$.

1. n mode multiplication of tensor `aten` with one matrix `bmat`: Assuming `bmat` has size $mr \times mc$.
   - optional third argument `n` is not specified: `n` is assumed equal to d and the row number mr must be the same as m_d
   - optional third argument `n` is specified: the integer `n` must refer to a dimension of `aten` and the row number mr must be the same as m_n

   The result of the multiplication is a tensor with dimension d where dimension n has now size m_c, like the number of columns of matrix `bmat`.
2. multiplication of tensor `aten` with a list of d matrices `bmat`: Any specification of n is ignored. The row numbers of the d matrices in the list must agree with the tensor sizes $(m_1, \ldots, m_d)$. The result of the multiplication is a tensor with dimensionality d and sizes like the columns numbers of the d matrices.
3. multiplication of tensor `aten` with a list of $d-1$ matrices `bmat`:
   - optional third argument `n` is not specified: `n` is assumed equal to d.
   - optional third argument `n` is specified: the integer `n` must refer to a dimension of `aten`.

   The row numbers of the $d-1$ matrices in the list must agree with the tensor sizes $(m_1, \ldots, m_{n-1}, m_{n+1}, \ldots, m_d)$. The result of the

multiplication is a tensor with d dimensions with sizes corresponding to the column sizes of the $d-1$ matrices except of that of dimension n which is of the same size m_n as that tensor `aten`.

Input: **aten** The first input argument should be a tensor (multidimensional array with $d \geq 3$ dimensions.)

bvec The second input argument must be either a single matrix, a list of d matrices or a list of $d-1$ matrices. There are restrictions to the size of the matrix(ces) as stated above.

n The optional third argument `n` must refer to one of the dimensions of tensor `aten`.

Output: The only result is either a tensor with dimension depending on the form of the second input argument.

Restrictions:

1. The input arguments cannot contain string data or missing values.
2. The dimension of the second input argument must be compatible with that of the first argument.
3. Note, that the value of third input argument depends on the specified index base, which is by default equal one, but maybe changed to zero by the `INDBASE=` runtime option.

Relationships: tenttvec(), tentten()

Examples: 1. Multiply tensor with a single matrix:

```
srand(1);
nind  = [ 2 3 4 ];
atens = randt(nind,"duni");
bmat = [ 10. 10. 10. ,
         10. 10. 10. ];
cten1 = tentmat(atens,bmat,1);
print "Tensor * Matrix (ten1mat) Cten1=",cten1;


Tensor * Matrix (ten1mat) Cten1=

                  ****************************
                  Tensor cten1 with 3 Dimensions
                  ****************************


                          cten1_1
                          *******

                     Dense Matrix (3 by 4)
```

```
  |          1          2          3          4
-------------------------------------------------
1 |   10.383569  10.372733  13.568482  5.5659261
2 |   11.942144  16.592070  15.197440  7.5819924
3 |   7.3803058  4.7933042  13.778727  8.1109613


                cten1_2
                *******

            Dense Matrix (3 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   10.383569  10.372733  13.568482  5.5659261
2 |   11.942144  16.592070  15.197440  7.5819924
3 |   7.3803058  4.7933042  13.778727  8.1109613


                cten1_3
                *******

            Dense Matrix (3 by 4)

  |          1          2          3          4
-------------------------------------------------
1 |   10.383569  10.372733  13.568482  5.5659261
2 |   11.942144  16.592070  15.197440  7.5819924
3 |   7.3803058  4.7933042  13.778727  8.1109613
```

2. Multiply tensor with a list of d matrices:

```
list blist[3];
blist[1] = [ 10. 10. 10. ,
             10. 10. 10. ];
blist[2] = [ 20. 20. 20. ,
             20. 20. 20. ,
             20. 20. 20. ];
blist[3] = [ 30. 30. 30. ,
             30. 30. 30. ,
             30. 30. 30. ,
             30. 30. 30. ];
ctens = tentmat(atens,blist);
print "Tensor * MatrixList (tennmat) Ctens=",ctens;
```

```
Tensor * MatrixList (tennmat) Ctens=

                  ****************************
                  Tensor ctens with 3 Dimensions
                  ****************************

                             ctens_1
                             *******

                   Dense Symmetric Matrix (3 by 3)

                  S |           1          2          3
                  ------------------------------------
                  1 |   75160.592
                  2 |   75160.592  75160.592
                  3 |   75160.592  75160.592  75160.592


                             ctens_2
                             *******

                   Dense Symmetric Matrix (3 by 3)

                  S |           1          2          3
                  ------------------------------------
                  1 |   75160.592
                  2 |   75160.592  75160.592
                  3 |   75160.592  75160.592  75160.592


                             ctens_3
                             *******

                   Dense Symmetric Matrix (3 by 3)

                  S |           1          2          3
                  ------------------------------------
                  1 |   75160.592
                  2 |   75160.592  75160.592
                  3 |   75160.592  75160.592  75160.592
```

3. Multiply tensor with a list of $d-1$ matrices:

```
list blist[2];
blist[1] = [ 10. 10. 10. ,
             10. 10. 10. ];
```

```
  blist[2] = [ 30. 30. 30. ,
               30. 30. 30. ,
               30. 30. 30. ,
               30. 30. 30. ];
  cmat = tentmat(atens,blist,2);
  print "Tensor * MatrixList (tenmmat) Cmat=",cmat;


Tensor * MatrixList (tenmmat) Cmat=

                  ****************************
                  Tensor cmat with 3 Dimensions
                  ****************************

                             cmat_1
                             ******

                      Dense Matrix (3 by 3)

                    |          1          2          3
                  ------------------------------------
                  1 |   1196.7213  1196.7213  1196.7213
                  2 |   1539.4094  1539.4094  1539.4094
                  3 |   1021.8989  1021.8989  1021.8989


                             cmat_2
                             ******

                      Dense Matrix (3 by 3)

                    |          1          2          3
                  ------------------------------------
                  1 |   1196.7213  1196.7213  1196.7213
                  2 |   1539.4094  1539.4094  1539.4094
                  3 |   1021.8989  1021.8989  1021.8989


                             cmat_3
                             ******

                      Dense Matrix (3 by 3)

                    |          1          2          3
                  ------------------------------------
                  1 |   1196.7213  1196.7213  1196.7213
                  2 |   1539.4094  1539.4094  1539.4094
```

```
3 |   1021.8989  1021.8989  1021.8989
```

3.15 Function tentten

tens—scal = tentten(aten,bten<,n>)

Purpose: The `tentten` function implements two different ways to multiply a tensor with a tensor.

1. Both tensors `aten` and `bten` have same dimension and size and optional third argument `n` is not be specified or specified equal to dimensionality. Result of multiplication is scalar which is the dot product of corresponding entries of `aten` and `bten`.
2. Both tensors `aten` and `bten` may have different size: If the third argument `n` is not specified, `n` is obtained as the number of first dimensions of `aten` and `bten` which have the same size. If `n` is specified at least the first n dimensions of `aten` and `bten` must have same size. Assuming ad and bd are the dimensionalities of `aten` and `bten`, then the sum of the dimensions with unequal size $(ad - n) + (bd - n)$ is the dimensionality of the result tensor, matrix, vector or scalar.

n The optional third argument `n` must refer to one of the dimensions of tensor `aten`.

Input: **aten** The first input argument should be a tensor (multidimensional array with $d \geq 3$ dimensions.)

aten The second input argument should be a tensor (multidimensional array with $d \geq 3$ dimensions.)

Output:

Restrictions:
1. The input arguments cannot contain string data or missing values.
2. The integer specified with the third input argument must refer to a dimension.
3. Note, that the value of third input argument depends on the specified index base, which is by default equal one, but maybe changed to zero by the `INDBASE=` runtime option.

Relationships: tentvec(), tenttmat()

Examples: 1. Tensors of equal size: Result is scalar:

```
srand(1);
nind  = [ 2 3 4 ];
```

```
atens = randt(nind,"duni");
btens = randt(nind,"duni");
cscal = tentten(atens,btens);
print "Tensor * Tensor (tennten) Cscal=",cscal;
```

```
                ******************************
                Tensor atens with 3 Dimensions
                ******************************


                           atens_1
                           *******

                       Dense Matrix (3 by 4)

              |           1          2          3          4
            ----------------------------------------------------
            1 |   0.1849626  0.9700887  0.3998243  0.2593986
            2 |   0.9216026  0.9692773  0.5429792  0.5316917
            3 |   0.0497940  0.0665666  0.8193186  0.5238705


                           atens_2
                           *******

                       Dense Matrix (3 by 4)

              |           1          2          3          4
            ----------------------------------------------------
            1 |   0.8533943  0.0671846  0.9570239  0.2971940
            2 |   0.2726118  0.6899296  0.9767649  0.2265075
            3 |   0.6882366  0.4127639  0.5585541  0.2872256

Tensor * Tensor (tennten) Cscal= 6.7656
```

2. Tensors of unequal size: Result has dimension of sum of unequal dimensions:

```
mind  = [ 2 3 3 ];
btens = randt(mind,"duni");
cmat = tentten(atens,btens,2);
print "Tensor * Tensor (tenmten) Cmat=",cmat;
```

```
Tensor * Tensor (tenmten) Cmat=
```

	1	2	3
1	10.705	8.1924	4.1397
2	11.444	8.7583	4.4257
3	15.331	11.733	5.9289
4	7.6608	5.8628	2.9626