CMAT Newsletter: July 2009

Wolfgang M. Hartmann

July 2009

Contents

1 General Remarks

The new function `stand` is similar to a macro in SAS (written by Warren Sarle) and is related to the `univar` function which was extended by some more location and scale measures.
The old function `sample` was renamed to `sampmd` because it implemented a specific sampling algorithm with a specific sampling criterion (maximum distances). The general name `sample` is now used for a new function which implements equal and unequal probability sampling with or without replacement. This new function is similar to that in R, however, will produce different results due to the use of different random generators.
The new functions `combn`, `combn2`, `dmnom`, `hcube`, `nsimplex`, `rsimplex`, and `rmult` are similar to those in the `combinat` package of R. The function `rmult` is similar to `mrand` with the `"mnom"` option.
The new functions `affvsn` and `affrma` belong to the most used functions in R for the normalization of microarray data.
The new functions `loess` and `lowess` are implementations of Cleveland's algorithms which were first developed in Fortran for S and are also functions in R.
Due to a review for JSS on a nice time series package written in Matlab (Kugiumtzis, 2009) I added a number of time series functions. Some of those are similar in the results to those in TISEAN (Hegger, Kantz & Schreiber, 1999), however the code had to be completely new written and implemented for CMAT.

1.1 New Functions

affrma RMA (Robust Multichip Average) by Bolstad et.al (2003) for the normalization of microarray data

angle returns the angle of a complex number $z = x + i * y$

arima a simple ARIMA algorithm

armcov modified covariance method of linear prediction

burg estimates coefficients of moving average (MA) whitening filter using the lattice-filter method by Burg (1968)

combn all combinations of x taken m at a time

combn2 all combinations of x taken 2 at a time

dmnon density of multinomial

em1cl clustering univariate observations via mixtures of unimodal normal mixtures (Bartolucci, 2005)

hcube generate all points on hypercube lattice

kdtcrt create kD tree from data set

kdtnea find nearest neighbors on kD tree

kdtrng find neighbors in range of kD tree

ldp linear distance programming (Lawson and Hanson, 1974)

loess robust local polynomial regression fitting (Cleveland, Grosse, & Shyu, 1992)

lowess univariate (robust) locally weighted regression (Cleveland, 1979)

mempsd calculates power spectrum of autoregressive filter (by FFT or polynomial method)

nsimplex number of points on a (p, n) simplex number p-part compositions of n

polyfit least squares fit of polynomial model to data

polyval score data for polynomial model

pppd percentage points of Pearson distribution

pwelch estimates the power spectrum of a time series signal by the periodogram method (FFT) using Welch (1967)

rmult generate random samples from (n, p) multinomial distributions

sample performs equal and unequal probability sampling with or without replacement

sortp partial sorting for quantiles

stand is a function for columnwise centering and scaling of a data matrix wrt. location and scale measures

tslocfor forecasting a zero or first order local model

tsloctst error testing a zero or first order local model

tsmeas for a variety of time series measurements

tstrans for a variety of time series data transformations and filter

x11 a simple version of SAS PROC X11 for the seasonal adjustment of time series data

xsimplex generate all points on (p, n) simplex

1.2 Fixed Bugs

A number of bugs were fixed with the `mrand` function. Now the function is well tested for multinomial random numbers, for n dimensional random numbers inside and on the surface of spheres and simplexes. The n dimensional random vector v is inside or on the surface of a sphere with radius r when

$$\sqrt{\sum_i^n v_i^2} < r \quad \text{or} \quad \sqrt{\sum_i^n v_i^2} = r$$

The n dimensional random vector v is inside or on the surface of a simplex

$$\sum_i^n v_i < 1 \quad \text{or} \quad \sum_i^n v_i = 1$$

1. Multinomial Distribution:

```
   srand(1);
   m = 8; p = [ .2 .8 ];
   z11 = mrand("mnom",m,p);
   print "Multinomial M(8,2,p): z11=",z11;
```

```
Multinomial M(8,2,p): z11=
   |         1
--------------
 1 |    2.0000
 2 |    6.0000
```

```
   /* note: sum(p) == 1 */
   m = 8; p = [ .5 .5 ];
   z12 = mrand("mnom",m,p);
   print "Multinomial M(8,2,p): z12=",z12;
```

```
Multinomial M(8,2,p): z12=
   |         1
--------------
 1 |    4.0000
 2 |    4.0000
```

```
   /* note: sum(p) == 1 */
```

```
   m = 8; p = [ .8 .2 ];
   z13 = mrand("mnom",m,p);
   print "Multinomial M(8,2,p): z13=",z13;


Multinomial M(8,2,p): z13=
   |         1
--------------
 1 |    4.0000
 2 |    4.0000


   srand(1);
   m = 4; p = [ .2 .2 .2 .2 .2 ];
   z31 = mrand("mnom",m,p);
   print "Multinomial M(4,5,p): z31=",z31;


Multinomial M(4,5,p): z31=
   |         1
--------------
 1 |   1.00000
 2 |   1.00000
 3 |    2.0000
 4 |   0.00000
 5 |   0.00000


   m = 4; p = [ .1 .1 .1 .1 .6 ];
   z32 = mrand("mnom",m,p);
   print "Multinomial M(4,5,p): z32=",z32;


Multinomial M(4,5,p): z32=
   |         1
--------------
 1 |   0.00000
 2 |   0.00000
 3 |   0.00000
 4 |   0.00000
 5 |    4.0000
```

2. Inside and on Sphere with radius r:

```
   print "Inside sphere with radius r and dimension n";
   n = 5; radius = 10.;
   z4 = mrand("unis",n,radius);
   print "Random number in circle with radius:",z4;
   print "Radius=", rad1 = sqrt(z4[**]);
```

```
Inside sphere with radius r and dimension n

Random number in circle with radius:
   |         1
--------------
 1 |    4.4318
 2 |    3.5083
 3 |  -0.29795
 4 |    3.2946
 5 |    3.7376


Radius= 7.5407
```

```
   print "On outside of sphere with radius r and dimension n";
   n = 5; radius = 10.;
   z5 = mrand("unos",n,radius);
   print "Random number in circle with radius:",z5;
   print "radius=", rad1 = sqrt(z5[**]);
```

```
On outside of sphere with radius r and dimension n

Random number in circle with radius:
   |         1
--------------
 1 |    8.1525
 2 |    2.2828
 3 |    2.4728
 4 |  -0.60541
 5 |   -4.6737


radius= 10.000
```

3. Inside and on Simplex:

```
   print "Inside simplex with dimension n";
   n = 5;
   z6 = mrand("unie",n);
   print "Random number inside n dimensional simplex:",z6;
   print "Simplex=", sim1 = z6[+];
```

```
Inside simplex with dimension n

Random number inside n dimensional simplex:
   |         1
--------------
 1 |   0.25578
 2 |   0.02813
 3 |   0.06671
 4 |   0.08579
 5 |   0.07387


Simplex= 0.5103
```

```
   print "On surface of simplex with dimension n";
   n = 5;
   z7 = mrand("unoe",n);
   print "Random number on surface n dimensional simplex:",z7;
   print "Simplex=", sim2 = z7[+];
```

```
On surface of simplex with dimension n

Random number on surface n dimensional simplex:
   |         1
--------------
 1 |   0.00276
 2 |   0.12124
 3 |   0.38600
 4 |   0.20393
 5 |   0.28607


Simplex= 1.0000
```

2 Modifications of Features

2.1 Modifications of the `sample` Function

The old function `sample` was renamed to `sampmd` because it implemented a specific sampling algorithm with a specific sampling criterion (maximum distances). The general name `sample` is now used for a new function which implements equal and unequal probability sampling with or without replacement. This new function is similar to that in R, however, will produce different results due to the use of different random generators.

3 Extension to the Language

Objects of the form of kD trees were added to the CMAT language. At this time you can only create trees (function `kdtcrt()`), obtain nearest neighbor nodes (function `kdtnea()`) of points y, obtain all tree nodes within of a ball of specified radius for points y (function `kdtrng()`), and perform the general functions of printing and freeing.

4 Extensions to Various Functions

4.1 Extensions to `toeplitz` Function

c = toeplitz(a) | c = toeplitz(a,"cre")

y = toeplitz(a,"dur")

x = toeplitz(a,"lev",b)

s = toeplitz(a,"tre")

Purpose: The `toeplitz` function implements four algorithms for Toeplitz matrices :

- By default or specifying `sopt = "cre"`: it computes a Toeplitz matrix from an input vector a block Toeplitz matrix from an $nr \times nc$ input matrix. One dimension of the input argument must be a (int) multiple of the other dimension, i.e. either $nr = k*nc$ or $nc = k*nr$.

  nr = k * nc Be $\mathbf{A}$ of form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \mathbf{A}_3 \\ \vdots \\ \mathbf{A}_n \end{bmatrix}$$

  then

$$\mathbf{B} = \left[\begin{array}{c|c|c|c|c} \mathbf{A}_1 & \mathbf{A}_2^T & \mathbf{A}_3^T & \cdots & \mathbf{A}_n^T \\ \mathbf{A}_2 & \mathbf{A}_1 & \mathbf{A}_2^T & \cdots & \mathbf{A}_{n-1}^T \\ \mathbf{A}_3 & \mathbf{A}_2 & \mathbf{A}_1 & \cdots & \mathbf{A}_{n-2}^T \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \mathbf{A}_n & \mathbf{A}_{n-1} & \mathbf{A}_{n-2} & \cdots & \mathbf{A}_1 \end{array}\right]$$

  nc = k * nr Be $\mathbf{A}$ of form

$$\mathbf{A} = [\mathbf{A}_1|\mathbf{A}_2|\mathbf{A}_3|\cdots|\mathbf{A}_n]$$

  then

$$\mathbf{B} = \left[\begin{array}{c|c|c|c|c} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \cdots & \mathbf{A}_n \\ \mathbf{A}_2^T & \mathbf{A}_1 & \mathbf{A}_2 & \cdots & \mathbf{A}_{n-1} \\ \mathbf{A}_3^T & \mathbf{A}_2^T & \mathbf{A}_1 & \cdots & \mathbf{A}_{n-2} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \mathbf{A}_n^T & \mathbf{A}_{n-1}^T & \mathbf{A}_{n-2}^T & \cdots & \mathbf{A}_1 \end{array}\right]$$

- Specifying `sopt ="dur"`: For a given $n \times n$ Toeplitz matrix r in vector form it solves the Yule-Walker equations (Golub and Van Loan, 1989, p.185) using the algorithm by Durbin. The algorithm takes $2n^2$ flops, whereas standard methods like Cholesky would need $O(n^3)$ flops.

- Specifying `sopt ="lev"`: For a given Toeplitz matrix in vector form r and a right hand side vector b it efficiently solves the linear equations using the algorithm by Levinson (Golub and Van Loan, 1989, p.186). The algorithm takes $4n^2$ flops, whereas standard methods like Cholesky would need $O(n^3)$ flops.
- Specifying `sopt ="tre"`: For a given $n \times n$ Toeplitz matrix r in vector form it computes the inverse of that matrix using the algorithm by Trench (Golub and Van Loan, 1989, p.190). The result should be identical to that of the `inv` function on the same Toeplitz matrix, however, the Trench algorithm works on the vector form of the matrix and should be much faster than the `inv` function. The algorithm takes $13n^2/4$ flops, whereas standard methods like Cholesky would need $O(n^3)$ flops.

Input: sopt="cre" The argument `a` must be a either a numeric vector or matrix of (nonmissing) values.

sopt ="dur","lev","tre" : must be an $n+1$ vector r with $r_1 = 1$ specifying the Toeplitz matrix.

Output: sopt="cre" The output is the $\max(nr, nc) \times \max(nr, nc)$ square matrix $\mathbf{B}$ which is symmetric when submatrix $\mathbf{A}_1$ is symmetric.

sopt ="dur" : returns n vector y with the solution of the Yule-Walker equations $y = T^{-1}r$.

sopt ="lev" : returns n vector x with the solution of the linear system $x = T^{-1}b$

sopt ="tre" : returns the $n \times n$ matrix T^{-1}

Restrictions:

1. Missing values and string data are not permitted.
2. For `sopt="cre"` one dimension of the input argument must be a (int) multiple of the other dimension, i.e. either $nr = k * nc$ or $nc = k * nr$.
3. For `sopt ="dur"`, `sopt ="lev"`, `sopt ="tre"`: the first entry of the input vector must be 1. For singular Toeplitz matrix a missing value is returned.

Relationships:

Examples:

1. Example from SAS/IML Manual for `sopt="cre"`:

```
/* input row vector */
vec = [ 1:5 ];
toe = toeplitz(vec);
print "Toeplitz1=",toe;
```

```
Toeplitz1=
 S |         1         2         3         4         5
```

```
-------------------------------------------------------
 1 |   1.00000
 2 |   2.00000   1.00000
 3 |   3.00000   2.00000   1.00000
 4 |   4.00000   3.00000   2.00000   1.00000
 5 |   5.00000   4.00000   3.00000   2.00000   1.00000


   /* input column vector */
   vec = [ 1:5 ]‘;
   toe = toeplitz(vec);
   print "Toeplitz2=",toe;


Toeplitz2=
 S |         1         2         3         4         5
-------------------------------------------------------
 1 |   1.00000
 2 |   2.00000   1.00000
 3 |   3.00000   2.00000   1.00000
 4 |   4.00000   3.00000   2.00000   1.00000
 5 |   5.00000   4.00000   3.00000   2.00000   1.00000


   /*--- [2] Matrix ---*/
   mat = [ 1 2,  3 4 , 5 6 , 7 8 ];
   toe = toeplitz(mat);
   print "Toeplitz3=",toe;


Toeplitz3=
   |         1         2         3         4
-------------------------------------------
 1 |   1.00000   2.00000   5.00000   7.00000
 2 |   3.00000   4.00000   6.00000   8.00000
 3 |   5.00000   6.00000   1.00000   2.00000
 4 |   7.00000   8.00000   3.00000   4.00000


   mat = [ 1 2 3 4, 5 6 7 8 ];
   toe = toeplitz(mat);
   print "Toeplitz4=",toe;


Toeplitz4=
   |         1         2         3         4
-------------------------------------------
 1 |   1.00000   2.00000   3.00000   4.00000
```

```
 2 |   5.00000   6.00000   7.00000   8.00000
 3 |   3.00000   7.00000   1.00000   2.00000
 4 |   4.00000   8.00000   5.00000   6.00000
```

2. Example by Golub & VanLoan (1989) for `sopt="dur"`:

```
/* x1= -75/140 = -.5357, x2= 12/140 = 0.08571, x3= -5/140 = -0.03571  */
print "Durbin algorithm: solving the Yule-Walker equations";
toep = [ 1. .5 .2 .1 ];
x1 = toeplitz(toep,"dur");
print "Durbin X1=",x1;
```

```
Durbin X1=
   |         1
--------------
 1 |  -0.53571
 2 |   0.08571
 3 |  -0.03571
```

3. Example by Golub & VanLoan (1989) for `sopt="lev"`:

```
/* x1= 355/56 = 6.3393; x2= -376/56 = -6.7143, x3= 285/56 = 5.0893 */
print "Levinson algorithm: solving linear system with RHS b";
toep = [ 1. .5 .2 .1 ];
rhs  = [ 4. -1. 3. ];
x2 = toeplitz(toep,"lev",rhs);
print "Levinson X2=",x2;
```

```
Levinson X2=
   |         1
--------------
 1 |    6.3393
 2 |   -6.7143
 3 |    5.0893
```

4. Example by Golub & VanLoan (1989) for `sopt="tre"`:

```
print "Trench algorithm: invert Toeplitz matrix";
toep = [ 1. .5 .2 .1 ];
/* options debug="toep*=3"; */
tinv = toeplitz(toep,"tre");
print "Trench Tinv=",tinv;
```

```
Trench Tinv=
 S |         1         2         3
-----------------------------------
 1 |    1.3393
 2 |  -0.71429    1.7143
 3 |   0.08929  -0.71429    1.3393
```

Test correctness:

```
toep = [ 1. .5 .2 ,
         .5 1. .5 ,
         .2 .5 1. ];
print "Inverse=", tinv = inv(toep);
```

```
Inverse=
 S |         1         2         3
-----------------------------------
 1 |    1.3393
 2 |  -0.71429    1.7143
 3 |   0.08929  -0.71429    1.3393
```

4.2 Extensions to `univar` Function

The `univar` function

<c,ase,conf> = univar(a<,sopt<,optn>>)

permitted the specification of the following options

`sopt="min"|"max"|"rng"|"ari"|"med"|"var"|"std"| "mad"|"sma"|"s_n"|"q_n"|`
`"ske"|"kur"|"qu1"|"qu3"|"iqr"|"qua"`

for the computation of univariate measures:

"min" minimum over all values in each column j

"max" maximum over all values in each column j

"rng" range (maximum - minimum)

"ari" arithmetic mean (default)

$$\mu_j = \frac{1}{m}\sum_i^m a_{ij}$$

"med" median

"var" variance

$$\sigma_j^2 = \frac{1}{m-1}\sum_i^m (a_{ij} - \mu_j)^2$$

"std" standard deviation

$$\sigma_j = \sqrt{\frac{1}{m-1}\sum_i^m (a_{ij} - \mu_j)^2}$$

"mad" median absolute deviation (see [**?**])

$$mad_j = med_{i=1}^m |a_{ij} - med_{i=1}^m(a_{ij})|$$

"sma" scaled median absolute deviation (see [**?**])

$$sma_j = 1.4826\, mad_j$$

The scaled version is more consistent with the Gaussian distribution.

"s_n" scale coefficient S_n (see [**?**]) The S_n is a more efficient alternative to MAD:

$$S_n = c_n * med_i\ med_{j \neq i}\ |x_i - x_j|$$

where the outer median is a low median (order statistic of rank $\left[\frac{n+1}{2}\right]$) and the inner median is a high median (order statistic of rank $\left[\frac{n}{2} + 1\right]$), and where c_n is a scalar depending on sample size n.

"q_n" scale coefficient Q_n (see [**?**]) The Q_n is another efficient alternative to MAD. It is based on the kth order statistic of the $\left(\begin{array}{c} n \\ 2 \end{array} \right)$ inter-point distances:

$$Q_n = d_n * \{|x_i - x_j|; \quad i < j\}_{(k)} \quad \text{with} \quad k \approx \left(\begin{array}{c} n \\ 2 \end{array} \right)/4$$

where d_n is a scalar similar to, but different from c_n.

"ske" skewness

$$\gamma_{1(j)} = \frac{m}{(m-1)(m-2)} \frac{\sum_i^m (a_{ij} - \mu_j)^3}{\sigma_j^3}$$

"kur" kurtosis

$$\gamma_{2(j)} = \frac{m(m+1)}{(m-1)(m-2)(m-3)} \frac{\sum_i^m (a_{ij} - \mu_j)^4}{\sigma_j^4} - \frac{3(m-1)^2}{(m-2)(m-3)}$$

"qu1" first quartile

"qu3" third quartile

"iqr" interquartile range

"qua" all three quartiles

In addition we implemented the following options
`sopt="ust"|"loo"|"lpm"|"fsp"|"biw"|"bis"| "hub"|"hus"|"wav"|"was"|"msp"|"mss"`
for the computation of the univariate measures:

"ust" uncorrected standard deviation,

"loo" maximum of absolute values (L_{oo} norm)

"lpm" L_p norm (location) for $p \geq 1$

"fsp" fourth-spread (Hoaglin, 1983)

"biw" Tukey's biweight location (Mosteller & Tukey, 1977) for $c > 0$

"bis" Tukey's biweight scale (Iglewicz, 1983) for $c > 0$

"hub" Huber's location (Goodall, 1983) for $k > 0$

"hus" Huber's scale (Iglewicz, 1983) for $k > 0$

"wav" Andrew's wave location (Goodall, 1983) for $c > 0$

"was" Andrew's wave scale (Iglewicz, 1983) for $c > 0$

"msp" Minimum spacing location (Sarle, 1995) for $0 < p < 1$.

"mss" Minimum spacing scale (Sarle, 1995) for $0 < p < 1$.

The following new options can be used:

Option Name	Column 2	Meaning
`"init"`	int	initial values (=0: robust, =1: classic) for `"biw","bis","hub","hus","wav","was"`
`"maxit"`	int	integer value speciying the mximum number of iterations for `"biw"`, `"hub"`, and `"wav"`
`"plpm"`	real	parameter $p \geq 1$ for `"lpm"` measure
`"pbiw"`	real	parameter $c > 0$ for `"biw"` and `"bis"`
`"phub"`	real	parameter $k > 0$ for `"hub"` and `"hus"`
`"pspa"`	real	parameter $0 < p < 1$ for `"msp"` and `"mss"`
`"pwav"`	real	parameter $c > 0$ for `"wav"` and `"was"`
`"vers"`	int	integer value specifying the version of the algorithm for `"biw"`, `"hub"`, and `"wav"`

Some Examples with Heart Data by Hawkins:

1. Heart Data: Minimum Spacing

```
a= [ 1 42.8 40.0 37,
     2 63.5 93.5 50,
     3 37.5 35.5 34,
     4 39.5 30.0 36,
     5 45.5 52.0 43,
```

```
  6 38.5 17.0 28,
  7 43.0 38.5 37,
  8 22.5  8.5 20,
  9 37.0 33.0 34,
 10 23.5  9.5 30,
 11 33.0 21.0 38,
 12 58.0 79.0 47 ];
aa = a[,2:4];


optn = [ "pspa"  .2 ];
sopt = [ "msp" "mss" ];
c4 = univar(aa,sopt,optn);
print "\n Minimum Space Location and Scale for p=.2:\n",c4;
```

```
 Minimum Space Location and Scale for p=.2:

           |     Var_1     Var_2     Var_3
-------------------------------------------
  MidMinSp |    37.750    37.750    36.500
  MinSpacg |    1.5000    4.5000   1.00000
```

2. Heart Data: Optimization for Tukey's Biweight

```
optn = [ "pbiw"  6.0 ];
sopt = [ "biw" "bis" ];
c4 = univar(aa,sopt,optn);
print "\n Opt: Tukeys Biweight Location and Scale for c=6:\n",c4;
```

```
 Opt: Tukeys Biweight Location and Scale for c=6:

           |     Var_1     Var_2     Var_3
-------------------------------------------
  Biwgt_M  |    39.919    36.916    36.218
  Biwgt_A  |    11.576    24.794    7.9058
```

3. Heart Data: SAS One-step for Tukey's Biweight

```
optn = [ "phub"  1.5 ];
sopt = [ "hub" "hus" ];
c5 = univar(aa,sopt,optn);
```

```
SAS: Tukeys Biweight Location and Scale for c=6:

          |     Var_1     Var_2     Var_3
----------------------------------------
  Biwgt_M |    38.547    33.758    36.340
  Biwgt_A |    11.655    24.501    8.5679
```

4.3 Extensions to `glim` Function

<gof,parm,sterr,conf,cov,typ1,typ3,yhat,roc> =

= glim(data,model<,optn<,class<,xini<,con>>>>)

- The Hosmer-Lemeshow goodness-of-fit test was added. The test can be specified with the `"hlt"` option and can be computed only for binary response y.
- The `"predpr"` option with a string input starting with `"i"`, `"c"`, or `"x"` specifies the observationwise computation and the output (in form of additional columns in the `yhat` return object) of individual, cumulative, or (approximative) cross validated predicted probabilities for the logistic model with binary or multilevel ordinal response. Until now, the cross validated predicted probabilities can only be computed for binary response.
- The `"pdres"` option specifies the observationwise output of Pearson and deviance residuals as additional columns into the `yhat` return object. This option is valid only for the logistic model with binary response.

4.4 Extensions to `cluster` Function

The `cluster` function

< weights,add1,add2 > = cluster(x,"sopt"<,optn<,scal>>)

was extended for better usability. For the input of similarity or dissimilarity data we permit lower triangular or symmetric $n \times n$ matrices.
We added Rousseeuw's plots (see Kaufman & Rousseeuw, 1990) to all six clustering methods,

Pam Silhouette plot

Clara Silhouette plot

Fanny Silhouette plot

Agnes Banner plot

Diana Banner plot

Mona Banner plot

Since the *banner* output from *Agnes* and *Diana* of the hierarchical structure was too difficult to use, we have added a third ouput argument to the results which shows the cluster membership:

- if the `"nclus"` option is specified: the third return argument is a m vector of integers $k = mem[j]$ defining the memberhip of observation j to cluster k
- if the `"nclus"` option is not specified: the third return argument is a $m \times m$ matrix of integers $k = mem[i, j]$ defining the cluster memberhip of the entire hierarchical tree. That means, the first row shows the assignment to m clusters and the last row shows the assignment to just one cluster.

Note, that an $m \times m$ matrix of cluster membership may need a lot of memory. The `"metric"` option was changed and the `"scale"` option was added:

Option Name	**Column 2**	**Meaning**
`"metr"`	string	type of dissimilarity measure:
	`"l1met"`	City block (L_1) metric for interval data (default if argument `scal` is not specified).
	`"l2dis"`	Euclidean distance (L_2) for interval data.
	`"l2squ"`	squared Euclidean distance (L_2^2) for interval data.
	`"gower"`	Gower dissimilarity measure suitable for six different scale types of variables (columns) in `x` (default if argument `scal` is specified).
`"scale"`	string	scaling of columns of raw (interval) data
	`"no"`	no scaling (default)
	`"st"`	zero mean and unit variance scaling
	`"or"`	original scaling by Rousseeeuw

The return arguments for *Agnes* and *Diana* are now:

Agnes returns the following three arguments:

1. m vector of the ordering of objects
2. m vector of the ordering of dissimilarities (distances)
3. either m vector or $m \times m$ matrix defining the cluster membership of the observations

Diana returns the following three arguments:

1. m vector of the ordering of objects
2. m vector of the ordering of dissimilarities (distances)
3. either m vector or $m \times m$ matrix defining the cluster membership of the observations

4.5 Extensions to `lls` Function

The `lls` function solves also linear least squares problems with boundary constraints and especially those with nonnegativity constraints. For nonnegative least squares problems two algorithms are now available:

- by Adlers (1998) which can be used for large sparse $m \times n$ matrices $\mathbf{A}$ with $rank(A) = n <= m$; this algorithm is used for `sopt = "gel"`;
- by Lawson and Hanson (1974, 1995) which is always using dense storage for $\mathbf{A}$ but can also be used for rankdeficient matrices and $m < n$; this algorithm is used for `sopt = "qrd"`.

4.6 Extensions to `affvsn` Function

This function was introduced in the last newsletter from December 2008, but has changed so much that we provide an entirely new documentation for it.

< gof,parms,dnew,mu > = affvsn(data,optn<,ref>)

Purpose: The `vsn` function implements the Huber et al. (2003) algorithm for "Variance Stabilizing Normalization" of the columns of a matrix of microarray data. The implementation is very similar to that of the Bioconductor function in R. The algorithms outer cycle is a fast version of LTS (Least Trimmed Squares, see Rousseeuw & Leroy, 1987) for the robust estimation of a nonlinear model predicting the values of a new data set with normalized columns. Each iteration selects a new subset of rows, its size is defined by the user specification of the quantile. The innermost part of the algorithm consists in the estimation of the parameters of a nonlinear model by means of an optimization algorithm. If there are no row strata specified, the algorithm computes $npar = 2 * n$ optimal parameter estimates. When there are n_s row strata specified the number of estimates increases to $npar = 2 * n_s * n$ model parameters. Test computations show that the results of the optimizations may differ considerably depending on the initial values. The model obviously does not restrict the parameter estimates to a unique solution.

Input: **data** this should be a $N \times n$ matrix of microarray data where the rows correspond to features (genes) and the columns to samples which need to be normalized. The data may obtain a column for an ID variable specifying a strata. The column number of the ID variable must be specified by the `optn` argument.

optn The `optn` argument is specified in form of a two column matrix where the first column defines the option as string value (in quotes) and the second column can be used for a numeric or string specification of the option. See table below for content.

ref not yet implemented.

Option	Second Column	Meaning
"agtol"	real	absolute gradient criterion for terminating the optimization; default is 2.e-4
"gtol"	real	relative gradient criterion for terminating the optimization; default is 1.e-6
"idvar"	int	column number containing ID variable; missing value means no ID variable which is default
"ltsit"	int	maximum number of LTS iterations; valid only for quant ¡ 1; default is 10
"ltseps"	real	for terminating LTS; default is 1.e-4
"maxit"	int	maximum number of iterations for optimization; default is 1000
"nsamp"		size $n_r \leq N$ of a reduced data set, needed only when N is too large for the computer resources. default is $n_r = 0$
"pdat"	int	print input and fitted data sets
"popt"	int	print optimization histories 0 is default, i.e. no printed optimization history
"print"	int	amount of printed output =0 specifies no printed output, default is 1
"quant"	real	quantile determining the subsample size for fast LTS algorithm, which must be in $(0, 1]$, default is quant=.9 for quant=1. no LTS subsampling is done and algorithm terminates with estimates for the complete data set of N rows.
"seed"	int	seed value for random generator default is time of day

Output: **gof** a vector of scalar results, see below for content;

parms this is an $n \times 2 * n_s$ matrix of the estimated optimal model parameters;

dnew this is the optimally transformed $N \times n$ data set;

mu this is an N vector with the offset value for each obervation.

Restrictions: 1. The input data must be numeric and cannot contain any string or complex data.

2. Currently the input data must not have any missing values.

Relationships: affarms(), affrma()

Examples: 1. Example 1: Subsample of first 100 observations of Affybatch Data:

```
  print "\n *** Test AFFVSN Function: AffyBatch\n";

#include "..\\tdata\\affybatch.dat"

  affb2 = affb0[1:100,];
  free affb0;
  print "AFFB2=",affb2[1:10,];


AFFB2=

   |       20A       20B       10A
-----------------------------------
 1 |    987.30    603.50    841.80
 2 |    127.30    202.00    118.00
 3 |    1048.8    668.00    958.00
 4 |    127.00    164.80    109.00
 5 |    1050.8    560.00    872.00
 6 |    130.50    99.000    105.00
 7 |    975.30    613.50    964.00
 8 |    143.30    161.00    117.00
 9 |    1046.5    647.00    1061.3
10 |    152.00    232.30    123.00


  optn = [ "print"       3 ,
           "popt"        1 ,
           "quant"      .9 ,
           "nsamp"       0 ,
           "ltsit"       7 ,
           "ltseps"  1.e-4 ,
           "maxit"    1000 ,
           "agtol"   1.e-3 ,
           "gtol"    2.e-4 ];
  < gof,parms,dnew,mu > = affvsn(affb2,optn);
   print "GOF=",gof;
   print "Parms=",parms;


        ************************************************
        VSN: Column Normalization by Variance Stabilization
        ************************************************

    Number of Rows of Data. . . . . .       100
```

```
Number of Columns of Data . . . . . .        3
Number of Estimates . . . . . .              6
LTS Iterations. . . . . .                    7
LTS Tolerance . . . . . . . . . . . . . . . . 0.000100000
LTS Quantile. . . . . . . . . . . . . . . . . 0.900000000
```

Name	Mean	Std Dev	Skewness	Kurtosis
Col_1	529.2770000	400.294769	0.03555787	-2.01056548
Col_2	349.2160000	220.212221	0.02998792	-1.95555250
Col_3	544.9800000	443.067630	0.05430788	-1.98246485

****
LTS Iteration History
****

Iter	Nselect	Nchanged	MaxChange
0	100	0	.
1	90	0	0.00000000

Offset and Slope Parameters
****

Dense Matrix (3 by 2)

	Offset	Slope
20A	120.67587	-0.9874578
20B	85.126001	-0.4107769
10A	133.76772	-1.0886330

Common Offset: -0.195931

Transformed Data Set
****

Dense Matrix (100 by 3)

	Col_1	Col_2	Col_3
Row_001	10.128050	10.118748	9.9004655
Row_002	8.5891019	8.9712471	8.6346973
Row_003	10.194177	10.240599	10.029783

```
Row_004 |   8.5881425  8.7989034  8.6092785
Row_005 |   10.196277  10.030345  9.9352053
Row_006 |   8.5992966  8.4322128  8.5978359
Row_007 |   10.114787  10.138327  10.036156
Row_008 |   8.6393696  8.7800815  8.6318950
Row_009 |   10.191758  10.202046  10.135753
Row_010 |   8.6659850  9.0978430  8.6486277
.........................................
Row_090 |   8.6322506  8.7421965  8.5891940
Row_091 |   10.046225  10.162433  10.022633
Row_092 |   8.5906996  8.5421280  8.6206311
Row_093 |   9.9411850  10.038704  10.056152
Row_094 |   8.6291443  8.5362364  8.5949611
Row_095 |   10.048901  10.152839  10.130998
Row_096 |   8.6750500  8.6070829  8.5834039
Row_097 |   10.097697  10.141241  10.144333
Row_098 |   8.5992966  8.7396351  8.5929452
Row_099 |   10.107774  10.086455  10.239449
Row_100 |   8.5881425  8.7189785  8.6064264


                Vector of Means
                ***************

            Dense Column Vector (nrow=100)

C |     Row_001    Row_002    Row_003    Row_004    Row_005
      6.8296883  5.9165321  6.9029990  5.8706176  6.8330531

C |     Row_006    Row_007    Row_008    Row_009    Row_010
      5.7858274  6.8624985  5.8833303  6.9180166  5.9667643

.....................................................

C |     Row_091    Row_092    Row_093    Row_094    Row_095
      6.8491028  5.8145037  6.8039903  5.8160940  6.8725418

C |     Row_096    Row_097    Row_098    Row_099    Row_100
      5.8403992  6.8842173  5.8557270  6.8958640  5.8514920

        Scalar Standard Deviation 0.00154819
      Number of LTS Iterations (Optimizations) 2
```

2. Example 2: Subsample of first 1000 observations of Affybatch Data:

```
print "\n *** Test AFFVSN Function: AffyBatch\n";
```

```
#include "..\\tdata\\affybatch.dat"

  affb1 = affb0[1:1000,];
  free affb0;


  print "AFFVSN: Nonlinear Variance Stabilizing Normalization";
  optn = [ "print"       3 ,
           "popt"        1 ,
           "quant"      .9 ,
           "nsamp"       0 ,
           "ltsit"       7 ,
           "ltseps"  1.e-4 ,
           "maxit"    1000 ,
           "agtol"   1.e-3 ,
           "gtol"    2.e-4 ];
  < gof,parms,dnew,mu > = affvsn(affb1,optn);
   print "GOF=",gof;
   print "Parms=",parms;


        ************************************************
        VSN: Column Normalization by Variance Stabilization
        ************************************************

    Number of Rows of Data. . . . . .      1000
    Number of Columns of Data . . . . .       3
    Number of Estimates . . . . . .           6
    LTS Iterations. . . . . .                 7
    LTS Tolerance . . . . . . . . . . . 0.000100000
    LTS Tolerance . . . . . . . . . . . 0.900000000


     Name             Mean     Std Dev    Skewness    Kurtosis
     ---------------------------------------------------------
     20A       475.0929000  612.671626  9.59744669  172.035534
     20B       317.4280000  355.046372  8.31115949  141.642031
     10A       453.2591000  529.930154  5.56319882  77.0406243


                      *********************
                      LTS Iteration History
                      *********************

        Iter  Nselect Nchanged   MaxChange OptIter     MaxGrad
           0    1000        0           .      500  0.05888938
```

```
1      900        0  0.00000000      15  4.097e-004
2      900        8  0.07227556      13  8.178e-004
3      900        2  0.02125046      11  7.494e-004
```

```
Offset and Slope Parameters
***************************

   Dense Matrix (3 by 2)

     |      Offset      Slope
   --------------------------
20A  |  -3980.1250  7.3352222
20B  |  -35993.828  7.7534064
10A  |   30966.198  7.3217304

  Common Offset: 11.7771
```

```
Vector of Means
***************

Dense Column Vector (nrow=1000)

C |    Row_0001   Row_0002   Row_0003   Row_0004   Row_0005
      14.839506  13.158362  14.936533  13.061021  14.846195

C |    Row_0006   Row_0007   Row_0008   Row_0009   Row_0010
      12.866191  14.885219  13.113457  14.958351  13.280592

C |    Row_0011   Row_0012   Row_0013   Row_0014   Row_0015
      14.867910  13.001598  14.789585  13.019991  14.814624

C |    Row_0016   Row_0017   Row_0018   Row_0019   Row_0020
      13.020371  14.799427  12.929062  14.831646  12.964749
.....................................................
C |    Row_0996   Row_0997   Row_0998   Row_0999   Row_1000
      16.205735  14.066063  12.629797  12.669701  12.624955

    Scalar Standard Deviation 0.0102716
Number of LTS Iterations (Optimizations) 4
```

3. Example 3: Kidney Data without Strata: $N = 8704, n = 2$:

```
options NOECHO;
```

```
#include "..\\tdata\\kidney.dat"
  options ECHO;

  nr = nrow(kidney); nc = ncol(kidney);
  print "Kidney: nr=",nr," nc=",nc;
  cnam = [" green red "];
  kidney = cname(kidney,cnam);
  print "Kidney[8700:8705,]=", kidney[8700:8705,];


                    Mixed Type Matrix (6 by 2)

                         green      red
                      1  -2.4900 -45.6500
                      2 -58.9900 -45.0200
                      3 -43.8900 -14.8700
                      4 -44.3500  39.1400
                      5  47.8000  55.9500
                      6    green      red


  options NOECHO;
#include "..\\tdata\\kidney_names.txt"
  options ECHO;
  print nrn = nrow(kidnam);
  print "Kidnam[8700:8705]=", kidnam[8700:8705];

  nr = 8704;
  kidn2  = kidney[1:nr,];
  rownam = kidnam[1:nr];
  cnam = [" green red "];
  kidn2 = cname(kidn2,cnam);
  kidn2 = rname(kidn2,rownam);
  print "Kidn2[1:20,]=", kidn2[1:20,];
  print "Kidn2[8700:8704,]=", kidn2[8700:8704,];


Kidn2[1:20,]=

      |     green       red
---------------------------
   X1 |    815.32    937.02
   X2 |    671.66    765.03
   X3 |    713.93    713.59
   X4 |    703.97    656.70
   X5 |    493.59    472.10
   X6 |    477.92    453.13
```

```
   X7 |    346.55    385.47
   X8 |    377.34    421.86
   X9 |    132.80    121.77
  X10 |    148.65    130.41
  X11 |    433.26    376.85
  X12 |    415.11    358.41
  X13 |    421.98    468.08
  X14 |    458.12    469.82
  X15 |    400.18    318.15
  X16 |    478.13    406.62
  X17 |    194.67    242.30
  X18 |    190.65    242.15
  X19 |    64.270    98.360
  X20 |    79.680    121.45


Kidn2[8700:8704,]=
        |     green       red
------------------------------
  X8700 |   -2.4900   -45.650
  X8701 |   -58.990   -45.020
  X8702 |   -43.890   -14.870
  X8703 |   -44.350    39.140
  X8704 |    47.800    55.950


  print "AFFVSN: Nonlinear Variance Stabilizing Normalization";
  optn = [ "print"       2 ,
           "popt"        1 ,
           "quant"      .9 ,
           "nsamp"       0 ,
           "ltsit"       7 ,
           "ltseps"  1.e-4 ,
           "maxit"    1000 ,
           "agtol"   1.e-3 ];
 < gof,parms,dnew,mu > = affvsn(kidn2,optn);
   print "GOF=",gof;
   print "Parms=",parms;
   print "dnew[1:10,]=",dnew[1:10,];


        ************************************************
        VSN: Column Normalization by Variance Stabilization
        ************************************************

    Number of Rows of Data. . . . . .      8704
    Number of Columns of Data . . . . .       2
```

```
Number of Estimates . . . . . .        4
LTS Iterations. . . . . .              7
LTS Tolerance . . . . . . . . . . . . 0.000100000
LTS Tolerance . . . . . . . . . . . . 0.900000000
```

Name	Mean	Std Dev	Skewness	Kurtosis
green	441.5344704	1140.80087	13.1191007	261.632594
red	454.3927034	1228.97438	13.1470974	250.669953

****
LTS Iteration History
****

Iter	Nselect	Nchanged	MaxChange	OptIter	MaxGrad
0	8704	0	.	19	5.875e-005
1	7830	0	0.00000000	23	1.099e-004
2	7830	32	0.14811972	22	0.00284971
3	7830	12	0.02285049	23	0.00184308
4	7830	2	0.00640679	10	1.651e-004

Offset and Slope Parameters
****

Dense Matrix (2 by 2)

	Offset	Slope
green	-0.0810981	-5.9291557
red	-0.0646649	-5.9585695

Common Offset: -7.57518
Scalar Standard Deviation 0.00335029
Number of LTS Iterations (Optimizations) 5

The return arguments are:

GOF=

	1
Failure	0.00000
Time	5.0000
SigmaSqu	0.0034

```
  LTSNiter |    5.0000
  OptNiter |    97.000
    unused |   0.00000


Parms=
         |    Offset     Slope
-----------------------------
  green  |  -0.08110   -5.9292
    red  |  -0.06466   -5.9586


dnew[1:10,]=
       |     green       red
--------------------------
   X1  |    9.7139    9.8726
   X2  |    9.4563    9.6000
   X3  |    9.5363    9.5087
   X4  |    9.5178    9.4014
   X5  |    9.0704    8.9989
   X6  |    9.0321    8.9521
   X7  |    8.6779    8.7754
   X8  |    8.7665    8.8723
   X9  |    7.9633    7.9321
  X10  |    8.0216    7.9633
```

4. Example 4: Kidney Data with $n_s = 2$ Strata: $N = 8704, n = 2$:

```
print "*** VSN with Strata ***";
ind = [ 2:2:nr ];
strat = cons(nr,1,1); strat[2:2:nr] = 2;
kidn3 = kidn2 -> strat;
cnam2 = [" green red strata "];
kidn3 = cname(kidn3,cnam2);
kidn3 = rname(kidn3,rownam);
print "Kidn3[1:20,]=", kidn3[1:20,];
print "Kidn3[8700:8704,]=", kidn3[8700:8704,];


Kidn3[1:20,]=

       |     green       red    strata
-------------------------------------
   X1  |    815.32    937.02   1.00000
   X2  |    671.66    765.03    2.0000
```

```
  X3 |    713.93    713.59   1.00000
  X4 |    703.97    656.70    2.0000
  X5 |    493.59    472.10   1.00000
  X6 |    477.92    453.13    2.0000
  X7 |    346.55    385.47   1.00000
  X8 |    377.34    421.86    2.0000
  X9 |    132.80    121.77   1.00000
 X10 |    148.65    130.41    2.0000
 X11 |    433.26    376.85   1.00000
 X12 |    415.11    358.41    2.0000
 X13 |    421.98    468.08   1.00000
 X14 |    458.12    469.82    2.0000
 X15 |    400.18    318.15   1.00000
 X16 |    478.13    406.62    2.0000
 X17 |    194.67    242.30   1.00000
 X18 |    190.65    242.15    2.0000
 X19 |    64.270    98.360   1.00000
 X20 |    79.680    121.45    2.0000


Kidn3[8700:8704,]=
        |     green       red    strata
----------------------------------------
 X8700 |   -2.4900   -45.650    2.0000
 X8701 |   -58.990   -45.020   1.00000
 X8702 |   -43.890   -14.870    2.0000
 X8703 |   -44.350    39.140   1.00000
 X8704 |    47.800    55.950    2.0000


 print "AFFVSN: Nonlinear Variance Stabilizing Normalization";
 optn = [ "print"       2 ,
          "popt"        1 ,
          "idvar"       3 ,
          "quant"      .9 ,
          "nsamp"       0 ,
          "ltsit"       7 ,
          "ltseps"  1.e-4 ,
          "maxit"    1000 ,
          "gtol"    1.e-3 ,
          "agtol"   1.e-3 ];
  < gof,parms,dnew,mu > = affvsn(kidn3,optn);
  print "GOF=",gof;
  print "Parms=",parms;
  print "dnew[1:10,]=",dnew[1:10,];
```

```
************************************************
VSN: Column Normalization by Variance Stabilization
************************************************

Number of Rows of Data. . . . . .      8704
Number of Columns of Data . . . . . .     2
Number of Row Strata. . . . . .           2
Number of Estimates . . . . . .           8
LTS Iterations. . . . . .                 7
LTS Tolerance . . . . . . . . . . . . . . 0.000100000
LTS Tolerance . . . . . . . . . . . . . . 0.900000000
Column Number of ID (Strata) Variable . . . . . .  3

          Number of Observations for 2 Strata
          -----------------------------------

                N      Strata        Nobs
                1           1        4352
                2           2        4352

Name             Mean     Std Dev    Skewness    Kurtosis
-----------------------------------------------------------
green     441.5344704  1140.80087  13.1191007  261.632594
red       454.3927034  1228.97438  13.1470974  250.669953

                  **********************
                  LTS Iteration History
                  **********************

   Iter  Nselect Nchanged   MaxChange OptIter     MaxGrad
      0    8704        0           .      96  6.212e-004
      1    7830        0  0.00000000      80  0.00252857
      2    7830       34  0.14034433      58  7.489e-004
      3    7830       12  0.03002145     103  0.00191574
      4    7830        4  0.00663385      71  2.002e-004
      5    7830        6  0.00389352      66  8.310e-004
      6    7830        2  0.00379861      35  0.00206817

             Offset and Slope Parameters
             ***************************

                Dense Matrix (2 by 4)
```

```
             |   Off_Str_1  Slo_Str_1  Off_Str_2  Slo_Str_2
-------------+----------------------------------------------
       green |  -0.0835686 -5.9356935 -0.0887593 -5.9128079
         red |  -0.0687708 -5.9621056 -0.0707674 -5.9451227


Offset Values for 2 Strata
--------------------------

  1 :      -7.582     -7.554


           Scalar Standard Deviation 0.00338175
          Number of LTS Iterations (Optimizations) 7
```

5 New Developments

5.1 Function `affrma`

< gof,dnew > = affrma(data,optn<,ref>)

Purpose: The `affrma` function implements the Bioconductor RMA algorithm by Bolstad et.al (2003) for the normalization of microarray data. This algorithm is an alternative to the `affvsn` and `affarms` methods for obtaining column normalized microarray data. However, other than fitting the parameters of a Maximum Likelihood model, the RMA method applies robust Medianpolish (Tukey, 1977a, p. 179), see also the `mpolish` function in CMAT to the data.

Input: data this should be a $N \times n$ matrix of microarray data where the rows correspond to features (genes) and the columns to samples which need to be normalized. The data may obtain a column for an ID variable specifying a strata. The column number of the ID variable must be specified by the `optn` argument.

optn The `optn` argument is specified in form of a two column matrix where the first column defines the option as string value (in quotes) and the second column can be used for a numeric or string specification of the option. See table below for content.

ref not yet implemented.

Option	Second Column	Meaning
`"bgcor"`		perform background correction default is background correction
`"idvar"`	int	column number containing ID variable; default is no ID variable
`"medp"`	int	algorithm for mean or median polish default is =0: median polish, =1: mean polish
`"mpit"`	int	maximum number of median polish iterations; default is 10 for median and 1 for mean polish
`"mpeps"`	real	for terminating median polish; default is 1.e-2
`"nobg"`		do not perform background correction default is background correction
`"nsamp"`		size $n_r \leq N$ of a reduced data set, needed only when N is too large for the computer resources. default is $n_r = 0$
`"pdat"`	int	print input and fitted data sets
`"print"`	int	amount of printed output =0 specifies no printed output, default is 1

Output: gof a vector of scalar results, see below for content;

dnew this is the normalized $N \times n$ data set.

Restrictions: 1. The input data must be numeric and cannot contain any string or complex data.

2. Currently the input data must not have any missing values.

Relationships: affarms(), affvsn()

Examples: 1. :

5.2 Function `angle`

x = angle(z)

Purpose: The `angle` function returns the phase angles in radians of the entries of the complex valued object $z = x + yi$. The result $x = atan2(y, x)$ is in $[-\pi, +\pi]$. For magnitude $r = abs(z) = sqrt(x^2 + y^2)$ the statement `z = r * exp(1.i*x)` converts back to the original complex z.

Input: The argument z must be numeric scalar, vector, or matrix.

Output: A missing value is returned if z is string or missing value. If argument is vector or matrix, function is computed elementwise.

Restrictions:

Relationships: abs(), atan2()

Examples:

```
z1 = 5.; z2 = -6.i; z3 = -1.+3.i;


print "x1=", x1 = angle(z1);
print "x2=", x2 = angle(z2);
print "x3=", x3 = angle(z3);
```

5.3 Function `arima`

< gof,coef,resi,forec> = arima(xdat,pord<,optn>)

Purpose: The `arima` function implements a simple algorithm for the estimation of the AR and ARIMA model. This function is still worked on.

Input: **xdat** is either a N column vector or a $N \times n$ matrix of N time series observations with n time variables.

pord for an AR model this can be a scalar p for the order of the AR model; for an ARIMA model this can be an integer vector with no more than the following four entries:

1. order p_0 of the initial AR model
2. order of AR (p)
3. order of I
4. order of MA (q)

optn this is a vector of options which should be initialized with missing values for defaults:

1. (int) amount of printed output (default is no output);
2. (int) number ns of forecasting steps (default is zero);
3. (int) maximum number of iterations for ARIMA process (default is 50);
4. (real) termination tolerance for ARIMA process (default 1.e-3).

Output: **gof** a vector of scalar results:

1. indicator for problems in the algorithm
2. global sum-of-squares residual
3. value of optimal log likelihood
4. Akaikes AIC crterion

coef the estimated coefficients of the model

resi the n residuals, one for each data column

forec only if option [2] specifies a number ns of forecast steps returns the $ns \times n$ matrix of forecast values

Restrictions: Currently no missing values are permitted in the data.

Relationships: tsarmcov(), tsmeas(), tstrans(), tsloctst(), tslocfor()

Examples: 1. AR model $p = 5$ with random data:

```
xd1 = [ 0.9501  0.2311  0.6068  0.4860  0.8913
        0.7621  0.4565  0.0185  0.8214  0.4447
        0.6154  0.7919  0.9218  0.7382  0.1763
        0.4057  0.9355  0.9169  0.4103  0.8936 ];
```

```
print "AR Model: pole=5, step=2";
pord = 5;
optn = [ 2 2 50 .001 ];
< gof1,coef1,resi1,fore1 > = arima(xd1,pord,optn);
```

```
            ************************************
            ARIMA Model of Order (AR,I,MA)=(0,0,0)
            ************************************
 Number of Observations. . . . . .          20
 Number of Variables . . . . . .            1
 Order of Initial AR Model . . . . . .      5
 Number of Forecast Steps. . . . . .        2
 Maximum ARIMA Iterations. . . . . .        50
 Termination Tolerance . . . . . .       0.001

            Data Centered around Mean 0.623705000
               Average Forecast Error: 0.23693
             Log-Likelihood=0.42107 AIC=9.15786
```

Coefficients		Differences	
1	-0.108	1	0
2	-0.5727	2	0
3	-0.3269	3	0
4	-0.1527	4	0
5	-0.3725	5	0
		6	0.1445
		7	-0.1929
		8	-0.4839
		9	0.07141
		10	-0.4381
		11	-0.08622
		12	-0.0253
		13	0.05772
		14	0.2866
		15	-0.2773
		16	-0.08071
		17	0.1776
		18	0.1843
		19	-0.1001
		20	0.3167

AR Iterated Time Series (2 Steps)

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

                1-0.65683135
                2-0.19378859
```

2. ARIMA model $p = (5, 1, 0, 0)$ with random data:

```
print "ARIMA Model: p0=5, p=1, step=2";
pord = [ 5 1 0 0 ];
optn = [ 2 2 50 .001 ];
< gof2,coef2,resi2,fore2 > = arima(xd1,pord,optn);


            ************************************
            ARIMA Model of Order (AR,I,MA)=(1,0,0)
            ************************************
 Number of Observations. . . . . .          20
 Number of Variables . . . . . .            1
 Order of Initial AR Model . . . . . .      5
 Number of Forecast Steps. . . . . .        2
 Maximum ARIMA Iterations. . . . . .        50
 Termination Tolerance . . . . . .       0.001


           Data Centered around Mean 0.623705000


 ARIMA Convergence: Residuals and Differences in Coefficients
 ************************************************************

                 1     6 0.01855633 0.15171437
                 2     7 0.09741175 0.02797422
                 3     8 0.11384142 0.03358074
                 4     9 0.05526902 0.01372017
                 5    10 3.563e-004 9.006e-005

              Average Forecast Error: 0.274395
             Log-Likelihood=-2.51504 AIC=7.03009
```

Coefficients		Differences	
1	0.0535	1	0
		2	-0.4101
		3	0.004099
		4	-0.1368
		5	0.275
		6	0.1241
		7	-0.1746
		8	-0.5963
		9	0.2301
		10	-0.1896
		11	0.001272
		12	0.1686
		13	0.2891
		14	0.09855
		15	-0.4535
		16	-0.1941
		17	0.3235
		18	0.2765
		19	-0.2291
		20	0.2813

ARIMA Iterated Time Series (2 Steps)

****

1	0.12740847
2	-0.20435799

5.4 Function `armcov`

< coef,err > = armcov(xdat,ord<,optn>)

Purpose: The `armcov` function implements the modified covariance methid of linear prediction. It is based on the QR method by S. L. Marple (1991). It obviously gives the same results as the Matlab function `armcov()`. It may be applied to complex data.

Input: **xdat** is either a N column vector or a $N \times n$ matrix of N time series observations with n time variables.

ord this can be a scalar order p or an integer vector $p = (p_j)$ with a variety of m orders p_j.

optn this is either a scalar or a vector of options which should be initialized with missing values for defaults:

1. (int) amount of printed output (default is no output);

Output: For an N vector of time series data and an integer $ord \geq 1$ the `armcov` function returns:

coef an $ord + 1$ real vector of moving average filter coefficients

err the residual e.

For more general input:

coef the $n * m \times \max_j p_j + 1$ estimated sets of $p_j + 1$ coefficients of the $n * m$ models

err the $n \times m$ matrix of residuals, one for each data column and each order in p

Restrictions: Currently no missing values are permitted in the data. This function permits complex input data.

Relationships: tsarima(), tsmeas(), tstrans(), tsloctst(), tslocfor()

Examples: All of the examples are based on the following data:

```
xd  = [ 0.8147 0.9058 0.1270 0.9134 0.6324
        0.0975 0.2785 0.5469 0.9575 0.9649
        0.1576 0.9706 0.9572 0.4854 0.8003
        0.1419 0.4218 0.9157 0.7922 0.9595 ];
```

1. Fitting model $ord = 1$ with random data:

```
print "ARMCOV Model: ord=1";
ord = 1; optn = 2;
< coef1,err1 > = armcov(xd,ord,optn);
print "COEF1=",coef1;
print "ERR1=",err1;
```

```
COEF1=
   |         1         2
-------------------------
 1 |   1.00000  -0.77388

ERR1= 0.2010
```

2. Fitting model $ord = 2$ with random data:

```
print "ARMCOV Model: ord=2";
ord = 2; optn = 2;
< coef2,err2 > = armcov(xd,ord,optn);
print "COEF2=",coef2;
print "ERR2=",err2;
```

```
COEF2=
   |         1         2         3
-----------------------------------
 1 |   1.00000  -0.49138  -0.35201

ERR2= 0.1839
```

3. Fitting model $ord = 5$ with random data:

```
print "ARMCOV Model: ord=5";
ord = 5; optn = 2;
< coef5,err5 > = armcov(xd,ord,optn);
print "COEF5=",coef5;
print "ERR5=",err5;
```

```
COEF5=
   |         1         2         3         4         5         6
-----------------------------------------------------------------
 1 |   1.00000  -0.31404   0.16813  -0.42493   0.11809  -0.54182

ERR5= 0.09846
```

5.5 Function `burg`

< coef,err > = burg(xdat,ord<,optn>)

Purpose: This function is based on the `burg_filter` Matlab function by P. V. Lanspeary (2006): The `burg` function calculates coefficients of a moving average (MA) whitening filter using the lattice-filter of Burg (1968) and also described by Kay & Marple (1981). This filter reduces the data to white noise. Two other filters can be built from this whitening filter:

1. a moving average linear prediction filter by removing the first (zero-lag) coefficient and negating the others.
2. an autoregressive generating (AR) filter by using "residual" as the sole MA coefficient and using "coeffs" to supply the AR coefficients of an ARMA filter.

The power spectrum of the ARMA filter is an estimate of the maximum entropy power spectrum of the data. The filter is

$$x_n = e_n + \sum_{k=0}^{ord} coef_k * x_{n-k}$$

Input: **xdat** an N vector of time series data or a $N \times n$ matrix with n univariate time series. The first entry is the zero lag coefficient which is always one.

ord an integer or a vector of integers specifying the maximum order of the moving average.

optn this is either a scalar or a vector of options which should be initialized with missing values for defaults:

1. (int) amount of printed output (default is no output);
2. (int) method=0: run all orders starting from 1; method=1: FPE early termination; method=2: AIC early termination;

Output: For an N vector of time series data and an integer $ord \geq 1$ the `burg` function returns:

coef an $ord + 1$ real vector of moving average filter coefficients

resi the mean square residual, white noise of the filter as a real scalar

For more general input:

coef the $n * m \times \max_j p_j + 1$ estimated sets of $p_j + 1$ coefficients of the $n * m$ models

resi the $n \times m$ matrix of residuals, one for each data column and each order in p

Restrictions: 1. Currently neither missing, complex, nor string data are permitted.

Relationships: armcov(), mempsd(), pwelch()

Examples: 1. Run algorithm for all $p = 1, \ldots, 5$:

```
xd  = [ 0.8147 0.9058 0.1270 0.9134 0.6324
        0.0975 0.2785 0.5469 0.9575 0.9649
        0.1576 0.9706 0.9572 0.4854 0.8003
        0.1419 0.4218 0.9157 0.7922 0.9595 ];


print "BURG-Filter: all nord";
optn = 2; nord = 5;
< coef,resi > = burg(xd,nord,optn);
print "ALL: COEFF=",coef;
print "ALL: RESI=",resi;
```

```
          Burg Method for Estimating AR Coefficients
          ******************************************

     Order    FPE_Crit    AIC_Crit    Residual   Terminate

         2  0.24507333 -1.40847869  0.18114116
         3  0.21122725 -1.56028580  0.14081817
         4  0.23449151 -1.46116151  0.14069491   FPE   AIC
         5  0.18315819 -1.71644430  0.09862364


ALL: COEFF=
   |         1         2         3         4         5         6
---------------------------------------------------------------------
 1 |   1.00000  -0.30480   0.14011  -0.40040   0.14594  -0.54683


ALL: RESI= 0.09862
```

2. Run algorithm for $p = 1, \ldots, 5$ with FPE termination:

```
print "BURG-Filter: with FPE termination";
optn = [ 2 1 ]; nord = 5;
< coef,resi > = burg(xd,nord,optn);
print "FPE: COEFF=",coef;
print "FPE: RESI=",resi;
```

```
            Burg Method for Estimating AR Coefficients
            ******************************************

        Order    FPE_Crit    AIC_Crit    Residual   Terminate

            2  0.24507333 -1.40847869  0.18114116
            3  0.21122725 -1.56028580  0.14081817
            4  0.23449151 -1.46116151  0.14069491   FPE   AIC


FPE: COEFF=
   |          1          2          3          4          5          6
---------------------------------------------------------------------------
 1 |    1.00000   -0.33494   -0.11590   -0.47181    0.00000    0.00000


FPE: RESI= 0.1408
```

Note that specifying AIC termination would yield the same result.

5.6 Function `combn`

c = combn(x<,n>)

Purpose: The `combn` function generates all $nr = \begin{pmatrix} m \\ n \end{pmatrix}$ combinations of x taken n at a time.

Input: **x** this can be an integer scalar m or a numeric vector of m entries.

n this must be an integer scalar.

Output: This is a $nr \times n$ matrix containing all combinations of the entries of x taken n at a time.

Restrictions: 1. Currently no complex nor string data are permitted.

Relationships: combn2(), hcube(), xsimplex()

Examples: 1. $(x, 2)$:

```
x1 = [ 1 : 5 ]; n1 = 2;
c1 = combn(x1,n1); print "C1=", c1;
```

```
C1=
   |         1         2
-------------------------
 1 |   1.00000    2.0000
 2 |   1.00000    3.0000
 3 |   1.00000    4.0000
 4 |   1.00000    5.0000
 5 |    2.0000    3.0000
 6 |    2.0000    4.0000
 7 |    2.0000    5.0000
 8 |    3.0000    4.0000
 9 |    3.0000    5.0000
10 |    4.0000    5.0000
```

2. $(x, 3)$:

```
x2 = [ 1 : 4 ]; n2 = 3;
c2 = combn(x2,n2); print "C2=", c2;
```

```
C2=
   |         1         2         3
---------------------------------
```

```
  1 |   1.00000    2.0000    3.0000
  2 |   1.00000    2.0000    4.0000
  3 |   1.00000    3.0000    4.0000
  4 |    2.0000    3.0000    4.0000
```

5.7 Function `combn2`

c2 = combn2(x)

Purpose: The `combn2` function generates all $nr = \begin{pmatrix} m \\ 2 \end{pmatrix}$ combinations of x taken $n = 2$ at a time.

Input: The input argument x can be an integer scalar m or a numeric vector of m entries.

Output: This is a $nr \times 2$ matrix containing all combinations of the entries of x taken $n = 2$ at a time.

Restrictions: 1. Currently no complex nor string data are permitted.

Relationships: combn()

Examples:

```
c3 = combn2(x1); print "C3=", c3;
```

```
C3=
   |         1         2
-----------------------
 1 |   1.00000    2.0000
 2 |   1.00000    3.0000
 3 |   1.00000    4.0000
 4 |   1.00000    5.0000
 5 |    2.0000    3.0000
 6 |    2.0000    4.0000
 7 |    2.0000    5.0000
 8 |    3.0000    4.0000
 9 |    3.0000    5.0000
10 |    4.0000    5.0000
```

```
c4 = combn2(5);  print "C4=", c4;
```

```
C4=
   |         1         2
-----------------------
 1 |   1.00000    2.0000
 2 |   1.00000    3.0000
 3 |   1.00000    4.0000
 4 |   1.00000    5.0000
 5 |    2.0000    3.0000
 6 |    2.0000    4.0000
 7 |    2.0000    5.0000
 8 |    3.0000    4.0000
 9 |    3.0000    5.0000
10 |    4.0000    5.0000
```

5.8 Function `dmnom`

dens = dmnom(x,prob)

Purpose: The `dmnom` function computes the density of multinomial distributed points.

Input: **x** must be an integer vector of n entries;

prob should be a real vector with n probabilities in $(0, 1)$ adding up to one.

Output: Returns a real scalar with the density.

Restrictions: 1. The input data should have no missing values.

Relationships: rmult()

Examples: 1. Example in the R manual of `combinat`:

```
x5 = [ 1 1 4 4 ];  p5 = [ .2 .2 .3 .3 ];
c5 = dmnom(x5,p5); print "C5=", c5;
```

```
C5= 0.01653
```

5.9 Function `hcube`

z = hcube(x<,tran<,scal>>)

Purpose: The `hcube` function generates all points on a hypercube lattice. The hyper cube can be translated and scaled if the second or third arguments are specified. Note, that any scaling is done before translation.

Input: **x** must be an integer vector of n entries;

tran is an optional n vector for translation of the cube;

scal is an optional n vector for scaling of the cube.

Output: Returned is a $\prod_i = 1^n x_i \times n$ matrix.

Restrictions: 1.

Relationships: combn(), xsimplex()

Examples:

```
y1 = [ 2 3 ];
c6 = hcube(y1); print "C6=", c6;
```

```
C6=
   |         1         2
-----------------------
 1 |   1.00000   1.00000
 2 |   1.00000    2.0000
 3 |   1.00000    3.0000
 4 |    2.0000   1.00000
 5 |    2.0000    2.0000
 6 |    2.0000    3.0000
```

```
y2 = [ 3 4 ];
c7 = hcube(y2); print "C7=", c7;
```

```
C7=
   |         1         2
-----------------------
 1 |   1.00000   1.00000
 2 |   1.00000    2.0000
 3 |   1.00000    3.0000
 4 |   1.00000    4.0000
 5 |    2.0000   1.00000
 6 |    2.0000    2.0000
```

```
 7 |    2.0000    3.0000
 8 |    2.0000    4.0000
 9 |    3.0000   1.00000
10 |    3.0000    2.0000
11 |    3.0000    3.0000
12 |    3.0000    4.0000
```

```
  y3 = [ 2 3 4 ];
  c8 = hcube(y3); print "C8=", c8;
```

```
C8=
   |         1         2         3
-----------------------------------
 1 |   1.00000   1.00000   1.00000
 2 |   1.00000   1.00000    2.0000
 3 |   1.00000   1.00000    3.0000
 4 |   1.00000   1.00000    4.0000
 5 |   1.00000    2.0000   1.00000
 6 |   1.00000    2.0000    2.0000
 7 |   1.00000    2.0000    3.0000
 8 |   1.00000    2.0000    4.0000
 9 |   1.00000    3.0000   1.00000
10 |   1.00000    3.0000    2.0000
11 |   1.00000    3.0000    3.0000
12 |   1.00000    3.0000    4.0000
13 |    2.0000   1.00000   1.00000
14 |    2.0000   1.00000    2.0000
15 |    2.0000   1.00000    3.0000
16 |    2.0000   1.00000    4.0000
17 |    2.0000    2.0000   1.00000
18 |    2.0000    2.0000    2.0000
19 |    2.0000    2.0000    3.0000
20 |    2.0000    2.0000    4.0000
21 |    2.0000    3.0000   1.00000
22 |    2.0000    3.0000    2.0000
23 |    2.0000    3.0000    3.0000
24 |    2.0000    3.0000    4.0000
```

5.10 Function `kdtcrt`

tree = kdtcrt(x<,optn>)

Purpose: The `kdtcrt` function generates a k dimensional graph for a $N \times n$ data set **X** (see de Berg et.al., 2008). The resulting tree is available as a data object, i.e. can be printed, written to a `.dob` file, can be used for input into the `kdtnea()` and `kdtrng()` functions, and can be freed. Sparsity of the data set **X** is exploited, but **X** but must be numeric without strings or missing values. Three types of distance function can be specified with the options argument:

=0 the maximum or L_∞ distance,

=1 the City-Block or L_1 distance,

=2 the Euclidean or L_2 distance (is the default).

Input: **x** is a numerical $N \times n$ data matrix which must not contain any missing values.

optn is an optional argument for specifying the amount of printred output, the form of the distance function, and an index base.

The optional options argument `optn` must be a numeric vector with the following entries:

1 the amount of printed output (default is =0: no printed output)

2 the form of the distance function is specified with an integer:

- **0** the maximum or L_∞ distance,
- **1** the City-Block or L_1 distance,
- **2** the Euclidean or L_2 distance (is the default).

Output: The only output is a data object of type kD tree.

Restrictions:

1. The input data `x` must not contain any string, complex, or missing values.
2. Currently the data cannot be complex, however, it would be easy to extend the algorithm for that.

Relationships: kdtnea(), kdtrng()

Examples:

1. Simple small example:

```
a1 = [ 0   0   0 ,  1   1   1 ,
       2   2   2 ,  3   3   3 ,
       4   4   4 ,  5   5   5 ,
       6   6   6 ,  7   7   7 ];
tr1 = kdtcrt(a1);
print "Tree=",tr1;
```

```
*****************************************************
Depth First Traversal: k-D Tree tr1 (8 Nodes, 3 Dims)
*****************************************************

          Values of Node 1: Dim= 0 Index= 4
             4.0000000 4.0000000 4.0000000
                       Going Left
          Values of Node 2: Dim= 1 Index= 2
             2.0000000 2.0000000 2.0000000
                       Going Left
          Values of Node 3: Dim= 2 Index= 1
             1.0000000 1.0000000 1.0000000
                       Going Left
          Values of Node 4: Dim= 0 Index= 0
             0.0000000 0.0000000 0.0000000
            Next: Left and Right are Null
                 Next: Right is Null
                  Next: Going Right
          Values of Node 5: Dim= 2 Index= 3
             3.0000000 3.0000000 3.0000000
            Next: Left and Right are Null
                  Next: Going Right
          Values of Node 6: Dim= 1 Index= 6
             6.0000000 6.0000000 6.0000000
                       Going Left
          Values of Node 7: Dim= 2 Index= 5
             5.0000000 5.0000000 5.0000000
            Next: Left and Right are Null
                  Next: Going Right
          Values of Node 8: Dim= 2 Index= 7
             7.0000000 7.0000000 7.0000000
            Next: Left and Right are Null
```

2. EEG - Epileptics (see Guy Shechter, 2004) where the vector x has 3400 rows and the tree has the same number of nodes:

```
options NOECHO;
eegepi = [
%inc "..\\tdata\\EEGepilep.dat";
         ];
options ECHO;

tr2 = kdtcrt(eegepi);
print "Tree=",tr2;
```

```
*********************************************************
Depth First Traversal: k-D Tree tr2 (3400 Nodes, 1 Dims)
*********************************************************

          Values of Node 1: Dim= 0 Index= 669
                       Going Left
          Values of Node 2: Dim= 0 Index= 255
                       Going Left
          Values of Node 3: Dim= 0 Index= 1464
                       Going Left
        ....................................
        Values of Node 3398: Dim= 0 Index= 687
            Next: Left and Right are Null
                 Next: Right is Null
                  Next: Going Right
        Values of Node 3399: Dim= 0 Index= 960
                       Going Left
        Values of Node 3400: Dim= 0 Index= 279
            Next: Left and Right are Null
                 Next: Right is Null
```

5.11 Function `kdtnea`

< inds,dist,pnts > = kdtnea(tree,y<,optn>)

Purpose: The `kdtnea` function finds for each row of an $N_y \times n$ data set $\mathbf{X}$ the nearest neighbor in a k dimensional tree created by a former `kdtcrt()` call (see de Berg et.al., 2008).

Input: **tree** is a k dimensional tree created with a call of the `kdtcrt()` function on a $N \times n$ data set $\mathbf{X}$.

y is a numerical $N_y \times n$ data matrix which must not contain any missing values.

optn is an optional argument for specifying the amount of printed output, the form of the distance function, and an index base.

The optional options argument `optn` must be a numeric vector with the following entries:

1 the amount of printed output (default is =0: no printed output)

2 the form of the distance function is specified with an integer:

=0 the maximum or L_∞ distance,

=1 the City-Block or L_1 distance,

=2 the Euclidean or L_2 distance (is the default).

3 specifies an index base for the `inds` result, 1 is the default.

Note, even though the user can specify a different distance formula for determining the nearest neighbors than it was used for creating the tree, a warning will be printed since that could be a bug in the specification.

Output: **inds** is an N_y vector with the indices for the rows of the `x` data set which are nearest neighbors to the N_y rows of `y`.

dist is an N_y vector with the distances among the N_y rows of `y` and its nearest neighbors.

pnts is an $N_y \times n$ matrix with the nearest neighbor rows found.

Restrictions:

1. The input data `y` must not contain any string, complex, or missing values.
2. The number of columns n of the data in `y` must agree with the number of columns of the data `x` used in creating the `tree` object.

Relationships: kdtcrt(), kdtrng()

Examples:

1. Simple small example:

   First create the tree (for output see function `kdtcrt()`:

```
  a1 = [ 0   0   0 ,  1   1   1 ,
         2   2   2 ,  3   3   3 ,
         4   4   4 ,  5   5   5 ,
         6   6   6 ,  7   7   7 ];
  tr1 = kdtcrt(a1);
 print "Tree=",tr1;


  b1 = [ 2   3   4 ];
 < inds,dist,pnts > = kdtnea(tr1,b1);
 print "Inds=",inds;
 print "Dist=",dist;
 print "Pnts=",pnts;


Inds= 4
Dist= 1.4142
Pnts=
   |         1         2         3
-----------------------------------
 1 |    3.0000    3.0000    3.0000
```

2. EEG - Epileptics (see Guy Shechter, 2004) where the vector x has 3400 rows and the tree has the same number of nodes:
   First create the tree (for output see function `kdtcrt()`:

```
   options NOECHO;
   eegepi = [
   %inc "..\\tdata\\EEGepilep.dat";
            ];
   options ECHO;

   tr2 = kdtcrt(eegepi);


   b2 = [ 1000 2000 3000 ];
   < inds,dist,pnts > = kdtnea(tr2,b2);
   print "Inds=",inds;
   print "Dist=",dist;
   print "Pnts=",pnts;


Inds=
   |     1
----------
 1 |     3
 2 |   617
```

```
 3 |   280


Dist=
   |         1
--------------
 1 |    176.00
 2 |   0.00000
 3 |    67.000


Pnts=
   |         1
--------------
 1 |    1176.0
 2 |    2000.0
 3 |    2933.0
```

5.12 Function `kdtrng`

< nfnd,inds,dist,pnts > = kdtrng(tree,y,radius<,optn>)

Purpose: The `kdtrng` function finds for each row of an $N_y \times n$ data set $\mathbf{X}$ all rows in a k dimensional tree which are in a ball of *radius* from that observation y_i (see de Berg et.al., 2008). Assuming the algorithm finds $M = \sum_i^{N_y} nfnd[i]$ points in all N_y balls of the rows of `y`.

Input: **tree** is a k dimensional tree created with a call of the `kdtcrt()` function on a $N \times n$ data set $\mathbf{X}$.

y is a numerical $N_y \times n$ data matrix which must not contain any missing values.

radius is a real positive scalar determining the size of the search ball.

optn is an optional argument for specifying the amount of printred output, the form of the distance function, and an index base.

The optional options argument `optn` must be a numeric vector with the following entries:

1 the amount of printed output (default is =0: no printed output)

2 the form of the distance function is specified with an integer:

=0 the maximum or L_∞ distance,

=1 the City-Block or L_1 distance,

=2 the Euclidean or L_2 distance (is the default).

3 specifies an index base for the `inds` result, 1 is the default.

Note, even though the user can specify a different distance formula for determining the nearest neighbors than it was used for creating the tree, a warning will be printed since that could be a bug in the specification.

Output: **nfnd** is an N_y specifying the number of points found for each row of `y`. It sums to M.

inds is an M vector of the indices of the rows of `x` used in creating the k dimensional input tree.

dist is an M vector of the distances among the rows of `y` with the `nfnd[j]` rows of `x` inside the N_y balls.

pnts is an $M \times n$ matrix of all points of `x` found inside the N_y balls.

Restrictions:

1. The input data `y` must not contain any string, complex, or missing values.
2. The input radius must be a positive real scalar.
3. The number of columns n of the data in `y` must agree with the number of columns of the data `x` used in creating the `tree` object.

Relationships: kdtcrt(), kdtnea()

Examples: 1. Simple small example:

First create the tree (for output see function `kdtcrt()`:

```
   a1 = [ 0   0   0 ,  1   1   1 ,
          2   2   2 ,  3   3   3 ,
          4   4   4 ,  5   5   5 ,
          6   6   6 ,  7   7   7 ];

  tr1 = kdtcrt(a1);
  print "Tree=",tr1;


  b1 = [ 2   3   4 ];
  radius = 3.;
  < nfnd,inds,dist,pnts > = kdtrng(tr1,b1,radius);
  print "Nfnd=",nfnd;
  print "Inds=",inds;
  print "Dist=",dist;
  print "Pnts=",pnts;


Nfnd= 3


Inds=
   |   1
--------
 1 |   5
 2 |   3
 3 |   4


Dist=
   |         1
--------------
 1 |    2.2361
 2 |    2.2361
 3 |    1.4142


Pnts=
   |         1         2         3
-----------------------------------
 1 |    4.0000    4.0000    4.0000
 2 |    2.0000    2.0000    2.0000
 3 |    3.0000    3.0000    3.0000
```

2. EEG - Epileptics (see Guy Shechter, 2004) where the vector x has 3400 rows and the tree has the same number of nodes:
   First create the tree (for output see function `kdtcrt()`:

```
options NOECHO;
eegepi = [
%inc "..\\tdata\\EEGepilep.dat";
         ];
options ECHO;

tr2 = kdtcrt(eegepi);


radius = 3.; b3 = 2000. ;
< nfnd,inds,dist,pnts > = kdtrng(tr2,b3,radius);
print "Nfnd=",nfnd;
print "Inds=",inds;
print "Dist=",dist;
print "Pnts=",pnts;
```

```
Nfnd= 19
```

```
Inds=
   |      1
-----------
 1 |   2376
 2 |   1389
 3 |   2261
 4 |   1182
 5 |    639
 6 |   2872
 7 |    396
 8 |    681
 9 |    418
10 |   1379
11 |   2522
12 |    194
13 |   1777
14 |    728
15 |    876
16 |    704
17 |    617
18 |    976
19 |   1146
```

```
Dist=
   |         1
--------------
 1 |    3.0000
 2 |    3.0000
 3 |    3.0000
 4 |    3.0000
 5 |    2.0000
 6 |   0.00000
 7 |   1.00000
 8 |    3.0000
 9 |    3.0000
10 |    2.0000
11 |   1.00000
12 |   1.00000
13 |   0.00000
14 |   0.00000
15 |   0.00000
16 |   0.00000
17 |   0.00000
18 |    3.0000
19 |    3.0000
```

```
Pnts=
   |         1
--------------
 1 |    1997.0
 2 |    1997.0
 3 |    1997.0
 4 |    1997.0
 5 |    2002.0
 6 |    2000.0
 7 |    1999.0
 8 |    1997.0
 9 |    1997.0
10 |    1998.0
11 |    1999.0
12 |    1999.0
13 |    2000.0
14 |    2000.0
15 |    2000.0
16 |    2000.0
17 |    2000.0
18 |    2003.0
19 |    2003.0
```

5.13 Function `ldp`

<x,norm> = ldp(a,b<optn>)

Purpose: For a given $m \times n$ matrix **A** and each column b_j, $j = 1, \ldots, p$, of a given $m \times p$ matrix **B** the LDP function computes the columns x_j, $j = 1, \ldots, p$, of the $n \times p$ matrix **X** which minimize the following p problems by linear distance programming (LDP),

$$\min x_j^T x_j \quad \text{subject to} \mathbf{A} x_j \geq b_j \quad , \quad j = 1, \ldots, p$$

An algorithm by Lawson & Hanson (1974,1995) was implemented.

Input: **a** Numeric (real or complex) $m \times n$ coefficient matrix **A**.

b Numeric (real or complex) $m \times p$ matrix **B** containing p right hand sides in its columns.

Output and Algorithms: 1. $n \times p$ matrix **X** containing the p minimum length solutions x_j in its columns;

2. the Euclidean norm of all p solutions **X**.

An algorithm by Lawson & Hanson (1995) is used.

Restrictions: 1. Missing values are returned if **A** or **B** contains any string data or missing values.

Relationships: lls(), lsolve(), lse(), glm(), pinv()

Examples: The following is the input and output for the test program PROG6.FOR by Lawson & Hanson:

```
mg = 3; me = 4; n = 2;
T = [ 0.25 0.5 0.5 0.8 ];
W = [ 0.50 0.6 0.7 1.2 ];
E = T |> cons(1,4,1.); F = W;

G = cons(3,2,0.); H = cons(3,1,0.);
G[1,1] = G[2,2]= 1.; G[3,1]= G[3,2]= -1.;
H[3] = -1.;


print "Compute SVD of E[4,2]";
< S,U,V > = svd(E);
print "SVD left: U=",U;
print "SVD right: V=",V;
print "SVD values: S=",S;
```

```
SVD left: U=
   |         1         2         3         4
-------------------------------------------
 1 |   0.44398  -0.71175  -0.34944   0.41735
 2 |   0.49578  -0.07234  -0.22183  -0.83652
 3 |   0.49578  -0.07234   0.86248   0.07137
 4 |   0.55793   0.69495  -0.29120   0.34780


SVD right: V=
 S |         1         2
------------------------
 1 |   0.46711
 2 |   0.88420  -0.46711


SVD values: S=
 D |         1         2         3         4
-------------------------------------------
 1 |    2.2545         0         0         0
 2 |         0   0.34571         0         0


     FF = U' * F';
     print "FF=",FF;


FF=
   |         1
--------------
 1 |    1.5360
 2 |   0.38402
 3 |  -0.05353
 4 |   0.17408


     print "GENERALLY RANK DETERMINATION AND LEVENBERG-MARQUARDT\n",
           "STABILIZATION COULD BE INSERTED HERE.";
     print "DEFINE THE CONSTRAINT MATRIX FOR THE Z COORDINATE SYSTEM.";
     G2 = cons(mg,n);
     for (i = 1; i <= mg; i++)
     for (j = 1; j <= n; j++) {
       sm = G[i,] * V[,j];
       G2[i,j] = sm / S[j,j];
     }
     print "G2=",G2;
```

```
G2=
   |         1         2
-----------------------
 1 |   0.20719    2.5576
 2 |   0.39219   -1.3512
 3 |  -0.59937   -1.2065
```

```
     print "DEFINE CONSTRAINT RT SIDE FOR THE Z COORDINATE SYSTEM.";
     H2 = cons(mg,1);
     for (i = 1; i <= mg; i++) {
       sm = G2[i,] * FF[1:n];
       H2[i] = H[i] - sm;
     }
     print "H2=", H2;
```

```
H2=
   |         1
--------------
 1 |   -1.3004
 2 |  -0.08354
 3 |   0.38395
```

```
     print "SOLVE THE CONSTRAINED PROBLEM IN Z-COORDINATES.";
     < Z,norm > = ldp(G2,H2);
     print "Z=", Z; print "Norm=", norm;
```

```
Z=
   |         1
--------------
 1 |  -0.12681
 2 |  -0.25525
```

```
Norm= 0.2850
```

```
     print "TRANSFORM BACK FROM Z-COORDINATES TO X-COORDINATES.";
     ZZ = (Z + FF[1:n]) ./ dia2vec(S); /* print "ZZ=",ZZ;  */
     X  = E * ZZ;
     print "THE COEFICIENTS OF THE FITTED LINE F(T)=X(1)*T+X(2)";
     print "X=",X;
```

```
THE COEFICIENTS OF THE FITTED LINE F(T)=X(1)*T+X(2)

X=
   |         1
--------------
 1 |   0.62132
 2 |   0.37868
```

```
     res = norm*norm;
     for (i = n+1; i <= me; i++) res += FF[i] * FF[i];
     res = sqrt(res);
     print "The Residual norm is=",res;
```

```
The Residual norm is= 0.3382
```

```
     print "COMPUTE THE RESIDUALS.";
     F = W - X[1] * T - X[2];
     print "The Consecutive Residuals are F=",F;
```

```
The Consecutive Residuals are F=
   |         1         2         3         4
-------------------------------------------------
 1 |  -0.03401  -0.08934   0.01066   0.32426
```

5.14 Function `loess`

<gof,yhat,ytst> = loess(ytrn,xtrn,optn<,wgt<,para<,drsq<,xtst>>>>)

Purpose: The `loess` function performs robust univariate and multivariate robust locally weighted regression via the model $y = f(x_1, \ldots, x_p)$ The `loess` function is based on C and FORTRAN code by Cleveland, Grosse, & Shyu (1992) and is an extended version of Cleveland's (179) earlier `lowess` function.

Input: **ytrn** must be a numeric (real or integer) N vector of response values.

xtrn must be a $N \times p$ matrix of numeric values for p predictor variables.

optn two-column matrix specifying additional options, see table below.

wgt is an optional numeric (real or integer) N vector of positive weights $w = (w_i)$, default is $w_i = 1$.

para is an optional binary p vector where a zero value (default) indicates that the corresponding predictor j is locally fitted and a one value indicates that the corresponding predictor j is globally fitted.

drsq is an optional binary p vector where a value of one indicates that the polynomial fit degree is reduced from default 2 (square) to 1 (linear). This input is valid only when the degree is specified with its default value as 2 (square).

xtst is an optional numeric (real or integer) $M \times p$ matrix of test data for which predicted values and maybe standard error estimates are computed (and returned in `ytst`).

Options Matrix Argument: The option argument is specified in form of a two column matrix:

Option Name	Column 2	Meaning
"alpha"	real	probability for confidence intervals, default=.05
"cell"	real	controls accuracy of interpolation, def=.2
		maximum number points in a cell of kd-tree
"degr"	int	polynomial degree, must be 1 or 2, def=2
"enpt"	real	alternative to span, not used by default
"fami"	string	"gau" for Gaussian or "sym" for symmetric
"maxit"	int	maximum number of iterations
		default: =0: for gaussian, =4: for symmetric
"noase"		suppress computing standard estimates
		when xtst is input ASEs are default
"nono"		suppress normalization of predictors
"print"	int	amount of printed outpout (def=0: no output)
"surf"	string	"int" for interpolation or "dir" for direct
"stat"	string	"app" for approximate or "exa" for exact
		def="app", exact computation may be slow
"span"	real	controls degree of smoothing, def=.75
"trac"	string	"app" for approximate or "exa" for exact
		def="exa", for $N > 1000$ "app" is recommended

Output: **gof** is a vector of scalar returns with entries:

1. indicates failure
2. equivalent number of parameters
3. residual standard error
4. trace of L
5. Δ_1
6. Δ_2
7. $DF = \frac{\Delta_1^2}{\Delta_2}$

yhat $N \times 3$ matrix containing:

1. predicted values yhat for the training data (ytrn,xtrn)
2. residuals $ytrn - yhat$
3. robust weights

ytst $M \times 2$ or $M \times 4$ matrix containing:

1. predicted values ytst for the test data (ytrn,xtrn)
2. standard error estimates for ytst
3. lower confidence interval for ytst
4. upper confidence interval for ytst

Restrictions:

1. The data arguments xtrn, ytrn, and xtst must have real or integer data.
2. Observations (rows) in xtrn and ytrn with missing values are removed.

3. The current version assumes that the argument `xtst` must not have missing values.
4. The arguments `xtrn` and `xtst` must have corresponding columns. If an *interpolation* (and not *direct*) surface is specified the value of `xtst[i,j]` must be in the range of `xtrn[.,j]` to be scored with nonmissing values in `ytst[i]`.

Relationships: lowess()

Examples: 1. Car Data: Example in R: Gaussian, Surface=Interpolate

```
xy = [ 1      4    2,  2      4   10,  3      7    4,  4      7   22,
       5      8   16,  6      9   10,  7     10   18,  8     10   26,
       .....................................................
       45    23   54,  46    24   70,  47    24   92,  48    24   93,
       49    24  120,  50    25   85 ];


cnam = [" speed dist "];
xy = xy[,2:3];
xy = cname(xy,cnam);
print "XY=",xy;
ytrn = xy[,2]; xtrn = xy[,1]; test = [ 5:30 ];


print "Gaussian (iter=1): Surface=intpol";
optn = [ "print"       2 ,
         "fami"    "gau" ,
         "maxit"       1 ,
         "degr"        2 ,
         "cell"       .2 ,
         "surf"    "int" ,
         "stat"    "app" ,
         "trac"    "exa" ,
         "span"      .75 ];
< gof,yprd,ytst > = loess(ytrn,xtrn,optn,.,.,.,test);
print "Surf=Intpol: GOF=",gof;
print "Gaussian LOESS R Example:",yprd;
print "Test: Standard Errors:",ytst;


     ****************************************************
     LOESS Fit: Number Observations=50 Number Predictors=1
     ****************************************************

   Family=GAUSS Statistics=APPROX Surface=INTERP Trace=EXACT
         Span=0.75 Cell=0.2 Iterations=1 EnpTarg=  .
           Equivalent Number of Observations: 4.7714
```

```
        Residual Standard Error: 15.2923
Trace(Lhat)=5.23006 Delta1=44.3113 Delta2=43.9975


 ****************************************
 Robust Locally Weighted Regression (LOESS)
 ****************************************

 N           Y    YPredict    Residual   RobWeight
 1  2.00000000  5.89376691 -3.89376691  1.00000000
 2  10.0000000  5.89376691  4.10623309  1.00000000
 3  4.00000000  12.5679604 -8.56796042  1.00000000
 4  22.0000000  12.5679604  9.43203958  1.00000000
 5  16.0000000  15.3691829  0.63081706  1.00000000
 ..............................................
49  120.000000  85.1424670  34.8575330  1.00000000
50  85.0000000  95.3230955 -10.3230955  1.00000000


            ********************
            Scoring 26 Test Data
            ********************

 N    YPredict  StandError      CI_low      CI_upp
 1  7.81048863  7.81048863 -7.92426957  23.5452468
 2  10.0418077  10.0418077 -10.1880939  30.2717093
 3  12.5679604  12.5679604 -12.7510468  37.8869677
 4  15.3691829  15.3691829 -15.5930767  46.3314426
 5  18.4257115  18.4257115 -18.6941319  55.5455550
 ..............................................
20  85.1424670  85.1424670 -86.3827977  256.667732
21  95.3230955  95.3230955 -96.7117347  287.357926
22           .           .           .           .
23           .           .           .           .
24           .           .           .           .
25           .           .           .           .
26           .           .           .           .

[note] file tlowess.inp, line 64: The test data have 5
          observation(s) which are outside the range of the
          training data and are not scored with SURF=INTERPOLATE.
          You may use SURF=DIRECT.
```

2. Car Data: Example in R: Gaussian, Surface=Direct

```
print "Gaussian (iter=1): Surface=direct";
optn = [ "print"       2 ,
         "fami"    "gau" ,
         "maxit"       1 ,
         "degr"        2 ,
         "cell"       .2 ,
         "surf"    "dir" ,
         "stat"    "app" ,
         "trac"    "exa" ,
         "span"      .75 ];
< gof,yprd,ytst > = loess(ytrn,xtrn,optn,.,.,.,test);
print "Surf=Direct: GOF=",gof;
print "Gaussian LOESS R Example:",yprd;
print "Test: Standard Errors:",ytst;
```

```
     ****************************************************
     LOESS Fit: Number Observations=50 Number Predictors=1
     ****************************************************

   Family=GAUSS Statistics=APPROX Surface=DIRECT Trace=EXACT
        Span=0.75 Cell=0.2 Iterations=1 EnpTarg=  .
         Equivalent Number of Observations: 4.83393
              Residual Standard Error: 15.3109
      Trace(Lhat)=5.30078 Delta1=44.2324 Delta2=43.9162


          ****************************************
          Robust Locally Weighted Regression (LOESS)
          ****************************************

         N           Y    YPredict    Residual   RobWeight
         1  2.00000000  5.88705675 -3.88705675  1.00000000
         2  10.0000000  5.88705675  4.11294325  1.00000000
         3  4.00000000  12.4424238 -8.44242380  1.00000000
         4  22.0000000  12.4424238  9.55757620  1.00000000
         5  16.0000000  15.2810822  0.71891778  1.00000000
         .............................................
        49  120.000000  85.0533637  34.9466363  1.00000000
        50  85.0000000  95.3005225 -10.3005225  1.00000000


                    ********************
                    Scoring 26 Test Data
                    ********************

         N    YPredict  StandError      CI_low      CI_upp
```

```
 1  7.74100583  7.56599144 -7.50192464  22.9839363
 2  9.92659562  5.95909685 -2.07898220  21.9321734
 3  12.4424238  5.01201307  2.34490156  22.5399460
 4  15.2810822  4.55001335  6.11433419  24.4478303
 5  18.4257115  4.32159630  9.71914708  27.1322759
 .............................................
20  85.0533637  5.95248042  73.0611157  97.0456116
21  95.3005225  8.30690066  78.5649089  112.036136
22  106.974661  11.6019114  83.6007079  130.348614
23  120.092581  15.7924801  88.2760401  151.909122
24  134.665851  20.8646595  92.6305726  176.701129
25  150.698545  26.8238272  96.6575465  204.739544
26  168.190283  33.6839994  100.328343  236.052224
```

3. Car Data: Example in R: Symmetric, Surface=Interpolate

```
print "Symmetric (iter=3): Surface=intpol";
optn = [ "print"       2 ,
         "fami"    "sym" ,
         "maxit"       3 ,
         "degr"        2 ,
         "cell"       .2 ,
         "surf"    "int" ,
         "stat"    "app" ,
         "trac"    "exa" ,
         "span"      .75 ];
< gof,yprd,ytst > = loess(ytrn,xtrn,optn,.,.,.,test);
print "Symmetric(iter=3): Surf=Intpol: GOF=",gof;
print "Symmetric LOESS R Example:",yprd;
print "Test: Standard Errors:",ytst;
```

```
     ***************************************************
     LOESS Fit: Number Observations=50 Number Predictors=1
     ***************************************************

   Family=SYMM Statistics=APPROX Surface=INTERP Trace=EXACT
         Span=0.75 Cell=0.2 Iterations=3 EnpTarg=  .
           Equivalent Number of Observations: 4.7714
               Residual Standard Error: 14.1409
       Trace(Lhat)=5.23006 Delta1=44.3113 Delta2=43.9975
```

4. Car Data: Example in R: Symmetric, Surface=Direct

```
print "Symmetric (iter=3): Surface=direct";
optn = [ "print"       2 ,
         "fami"    "sym" ,
         "maxit"       3 ,
         "degr"        2 ,
         "cell"       .2 ,
         "surf"    "dir" ,
         "stat"    "app" ,
         "trac"    "exa" ,
         "span"      .75 ];
< gof,yprd,ytst > = loess(ytrn,xtrn,optn,.,.,.,test);
print "Symmetric(iter=3): Surf=Intpol: GOF=",gof;
print "Symmetric LOESS R Example:",yprd;
print "Test: Standard Errors:",ytst;
```

```
     ****************************************************
     LOESS Fit: Number Observations=50 Number Predictors=1
     ****************************************************

   Family=SYMM Statistics=APPROX Surface=DIRECT Trace=EXACT
         Span=0.75 Cell=0.2 Iterations=3 EnpTarg=  .
          Equivalent Number of Observations: 4.83393
               Residual Standard Error: 13.6457
       Trace(Lhat)=5.30078 Delta1=44.2324 Delta2=43.9162
```

5. Ethanol Data: Example by Cleveland, Grosse, & Shyu, p.20,23:

```
  options NOECHO;
#include "..\\tdata\\ethanol.dat"
  options ECHO;

  cnam = [" idnum NOx C E "];
  print "Ethanol=", ethanol;
  ytrn = ethanol[,2];
  xtrn = ethanol[,3:4];
  nr = nrow(xtrn);


  print "Cleveland, Grosse, & Shyu: p.20 and 23";
  print "Gaussian (iter=1): Surface=intpol: span=.5";
  optn = [ "print"       2 ,
           "fami"    "gau" ,
           "maxit"       1 ,
           "degr"        2 ,
           "span"       .5 ,
```

```
         "cell"       .2 ,
         "surf"    "int" ,
         "stat"    "app" ,
         "trac"    "exa" ];
< gof,yprd > = loess(ytrn,xtrn,optn);
print "Surf=Intpol: GOF=",gof;
print "Gaussian LOESS YPRD:",yprd;


     ***************************************************
     LOESS Fit: Number Observations=88 Number Predictors=2
     ***************************************************

   Family=GAUSS Statistics=APPROX Surface=INTERP Trace=EXACT
        Span=0.5 Cell=0.2 Iterations=1 EnpTarg=  .
         Equivalent Number of Observations: 12.9973
              Residual Standard Error: 0.259937
       Trace(Lhat)=15.4429 Delta1=70.1115 Delta2=68.5141


          ****************************************
          Robust Locally Weighted Regression (LOESS)
          ****************************************

         N           Y    YPredict    Residual   RobWeight
         1  3.74100000  3.78137430 -0.04037430  1.00000000
         2  2.29500000  2.61328006 -0.31828006  1.00000000
         3  1.49800000  1.54865307 -0.05065307  1.00000000
         4  2.88100000  2.76723888  0.11376112  1.00000000
         5  0.76000000  0.78715142 -0.02715142  1.00000000
         .............................................
        85  0.67800000  0.54348383  0.13451617  1.00000000
        86  0.37000000  0.45069057 -0.08069057  1.00000000
        87  0.53000000  0.47012674  0.05987326  1.00000000
        88  1.90000000  2.12710153 -0.22710153  1.00000000
```

6. Ethanol Data: Example by Cleveland, Grosse, & Shyu, p.29:

```
para = [ 1 0 ];
drsq = [ 1 0 ];

print "Test data, see page 32";
xtst = [  7.5  0.6,  9.0  0.8, 12.0  1.0,
         15.0  0.8, 18.0  0.6 ];


print "Cleveland, Grosse, & Shyu: p.29 and p.32";
```

```
print "Gaussian (iter=1): Surface=intpol: span=.5";
optn = [ "print"       2 ,
         "fami"    "gau" ,
         "maxit"       1 ,
         "degr"        2 ,
         "span"       .5 ,
         "cell"       .2 ,
         "surf"    "int" ,
         "stat"    "app" ,
         "trac"    "exa" ];
< gof,yprd,ytst > = loess(ytrn,xtrn,optn,.,para,drsq,xtst);
print "Surf=Intpol: GOF=",gof;
print "Gaussian LOESS YPRD:",yprd;
print "Test: Standard Errors:",ytst;
```

```
     ****************************************************
     LOESS Fit: Number Observations=88 Number Predictors=2
     ****************************************************

   Family=GAUSS Statistics=APPROX Surface=INTERP Trace=EXACT
        Span=0.5 Cell=0.2 Iterations=1 EnpTarg=  .
         Equivalent Number of Observations: 9.16731
              Residual Standard Error: 0.1842
      Trace(Lhat)=11.1047 Delta1=74.9579 Delta2=73.5893


          ****************************************
          Robust Locally Weighted Regression (LOESS)
          ****************************************

         N           Y    YPredict    Residual   RobWeight
         1  3.74100000  3.84537686 -0.10437686  1.00000000
         2  2.29500000  2.34556181 -0.05056181  1.00000000
         3  1.49800000  1.38206009  0.11593991  1.00000000
         4  2.88100000  2.84452085  0.03647915  1.00000000
         5  0.76000000  0.74574101  0.01425899  1.00000000
         .............................................
        85  0.67800000  0.49333696  0.18466304  1.00000000
        86  0.37000000  0.53182105 -0.16182105  1.00000000
        87  0.53000000  0.51556651  0.01443349  1.00000000
        88  1.90000000  1.63136308  0.26863692  1.00000000


                   ******************
                   Scoring 5 Test Data
                   ******************
```

N	YPredict	StandError	CI_low	CI_upp
1	0.28158249	0.12004258	0.04251479	0.52065018
2	2.59714110	0.04598603	2.50555881	2.68872340
3	3.06671776	0.04490255	2.97729326	3.15614227
4	3.25557778	0.06058134	3.13492858	3.37622698
5	1.06377877	0.08236972	0.89973748	1.22782005

5.15 Function `lowess`

<xyp,res> = lowess(xy<,optn>)

Purpose: The `lowess` function performs univariate robust locally weighted regression. For m points $(x, y)_i$ it computes predicted values $\hat{y}_i$, residuals $r_i = y_i - \hat{y}_i$, and robust weights w_i. This is an older and univariate version of the more general `loess()` function.

Input: **xy** must be a $m \times 2$ input data matrix where the first column contains the x values and the second column contains the y values of m points.

optn must be a numeric vector specifying some additional options, see below for details. For defaults it should be initialized with missing values.

Content of the Options Vector:

1. determines amount of printed output (default=0: no printed output)
2. maximum number of iterations (default=3); with nonrobust solution for zero iterations
3. f is the same as span in `loess()`, default: $f = \frac{2}{3}$
4. δ should be zero for small m,
   default: $\delta = \begin{cases} 0 & \text{for } m \leq 1000 \\ .01 * (\max x_i - \min x_i) & \text{otherwise} \end{cases}$
5. ϵ: terminate iteration when relative y change is smaller than ϵ, default $\epsilon = 1.e-4$

Output: **xyp** is a $m \times 3$ matrix containing the x and y input data in the first two columns and the predicted $\hat{y}$ values in column 3.

res is a $m \times 2$ matrix containing the robust weights w_i in column 1 and the residuals $r_i = y_i - \hat{y}_i$ in columns 2.

Restrictions: 1. Observations with missing values in the x or y column are moved to the end with predicted values and residuals set to missing values.

Relationships: loess()

Examples: 1. Example 1 by Cleveland:

```
xy = [ 1  2   3  4   5  10#6  8  10  12  14  50 ,
       18  2  15  6  10  4  16  11  7  3  14  17
       20  12  9  13  1  8  5  19 ];
xy = xy'; nr = nrow(xy);
cnam = [" x y "];
xy = cname(xy,cnam);
```

```
print "Example 1 by Cleveland";
optn = cons(5,1,.);
optn = [   1 ,   /* ipri */
           0 ,   /* maxit */
         .25 ,   /* f */
          0. ];  /* delt */
< oxy,res > = lowess(xy,optn);
print "Cleveland 1:",oxy;
print "Res=",res;
```

For zero iterations the solution is nonrobust and all robust weights are set to zero. The results match those reported by Cleveland:

```
          ****************************************
          Robust Locally Weighted Regression (LOWESS)
          ****************************************
```

N	X	Y	YPredict	Residual	RobWeight
1	1.00000000	18.0000000	13.6588201	4.34117989	0.00000000
2	2.00000000	2.00000000	11.1445697	-9.14456972	0.00000000
3	3.00000000	15.0000000	8.70116861	6.29883139	0.00000000
4	4.00000000	6.00000000	9.72203673	-3.72203673	0.00000000
5	5.00000000	10.0000000	10.0000000	0.00000000	0.00000000
6	6.00000000	4.00000000	11.3000000	-7.30000000	0.00000000
7	6.00000000	16.0000000	11.3000000	4.70000000	0.00000000
8	6.00000000	11.0000000	11.3000000	-0.30000000	0.00000000
9	6.00000000	7.00000000	11.3000000	-4.30000000	0.00000000
10	6.00000000	3.00000000	11.3000000	-8.30000000	0.00000000
11	6.00000000	14.0000000	11.3000000	2.70000000	0.00000000
12	6.00000000	17.0000000	11.3000000	5.70000000	0.00000000
13	6.00000000	20.0000000	11.3000000	8.70000000	0.00000000
14	6.00000000	12.0000000	11.3000000	0.70000000	0.00000000
15	6.00000000	9.00000000	11.3000000	-2.30000000	0.00000000
16	8.00000000	13.0000000	13.0000000	0.00000000	0.00000000
17	10.0000000	1.00000000	6.43989983	-5.43989983	0.00000000
18	12.0000000	8.00000000	5.59591371	2.40408629	0.00000000
19	14.0000000	5.00000000	5.45566698	-0.45566698	0.00000000
20	50.0000000	19.0000000	18.9981657	0.00183425	0.00000000

2. Example 2 by Cleveland:

```
print "Example 2 by Cleveland";
optn = cons(5,1,.);
optn = [   1 ,   /* ipri */
           0 ,   /* maxit */
```

```
         .25 ,   /* f */
          3. ];  /* delt */
< oxy,res > = lowess(xy,optn);
print "Cleveland 2:",oxy;
print "Res=",res;
```

```
          ****************************************
          Robust Locally Weighted Regression (LOWESS)
          ****************************************
```

N	X	Y	YPredict	Residual	RobWeight
1	1.00000000	18.0000000	13.6588201	4.34117989	0.00000000
2	2.00000000	2.00000000	12.3465590	-10.3465590	0.00000000
3	3.00000000	15.0000000	11.0342979	3.96570215	0.00000000
4	4.00000000	6.00000000	9.72203673	-3.72203673	0.00000000
5	5.00000000	10.0000000	10.5110184	-0.51101836	0.00000000
6	6.00000000	4.00000000	11.3000000	-7.30000000	0.00000000
7	6.00000000	16.0000000	11.3000000	4.70000000	0.00000000
8	6.00000000	11.0000000	11.3000000	-0.30000000	0.00000000
9	6.00000000	7.00000000	11.3000000	-4.30000000	0.00000000
10	6.00000000	3.00000000	11.3000000	-8.30000000	0.00000000
11	6.00000000	14.0000000	11.3000000	2.70000000	0.00000000
12	6.00000000	17.0000000	11.3000000	5.70000000	0.00000000
13	6.00000000	20.0000000	11.3000000	8.70000000	0.00000000
14	6.00000000	12.0000000	11.3000000	0.70000000	0.00000000
15	6.00000000	9.00000000	11.3000000	-2.30000000	0.00000000
16	8.00000000	13.0000000	13.0000000	0.00000000	0.00000000
17	10.0000000	1.00000000	6.43989983	-5.43989983	0.00000000
18	12.0000000	8.00000000	5.59591371	2.40408629	0.00000000
19	14.0000000	5.00000000	5.45566698	-0.45566698	0.00000000
20	50.0000000	19.0000000	18.9981657	0.00183425	0.00000000

3. Example 3 by Cleveland:

```
print "Example 3 by Cleveland";
optn = cons(5,1,.);
optn = [   1 ,   /* ipri */
           2 ,   /* maxit */
         .25 ,   /* f */
          0. ];  /* delt */
< oxy,res > = lowess(xy,optn);
print "Cleveland 3:",oxy;
print "Res=",res;
```

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

```
        Robust Locally Weighted Regression (LOWESS)
        ***************************************

 N           X           Y    YPredict    Residual   RobWeight
 1  1.00000000  18.0000000  14.8111660  3.18883397  0.94655395
 2  2.00000000  2.00000000  12.1154428 -10.1154428  0.63969543
 3  3.00000000  15.0000000  8.98377035  6.01622965  0.84638072
 4  4.00000000  6.00000000  9.67566603 -3.67566603  0.94386234
 5  5.00000000  10.0000000  10.0000000  0.00000000  1.00000000
 6  6.00000000  4.00000000  11.3460020 -7.34600199  0.78698212
 7  6.00000000  16.0000000  11.3460020  4.65399801  0.91053932
 8  6.00000000  11.0000000  11.3460020 -0.34600199  0.99953913
 9  6.00000000  7.00000000  11.3460020 -4.34600199  0.92275382
10  6.00000000  3.00000000  11.3460020 -8.34600199  0.72970174
11  6.00000000  14.0000000  11.3460020  2.65399801  0.97030789
12  6.00000000  17.0000000  11.3460020  5.65399801  0.86957704
13  6.00000000  20.0000000  11.3460020  8.65399801  0.70926168
14  6.00000000  12.0000000  11.3460020  0.65399801  0.99812242
15  6.00000000  9.00000000  11.3460020 -2.34600199  0.97730212
16  8.00000000  13.0000000  13.0000000  0.00000000  1.00000000
17  10.0000000  1.00000000  6.73445307 -5.73445307  0.87002440
18  12.0000000  8.00000000  5.74365955  2.25634045  0.97799401
19  14.0000000  5.00000000  5.41466631 -0.41466631  0.99925210
20  50.0000000  19.0000000  18.9981051  0.00189486  1.00000000
```

4. Car Data Example in R:

```
print "Car Data: Example in R";
xy = [ 1      4    2,  2      4   10,  3      7    4,  4      7   22,
       5      8   16,  6      9   10,  7     10   18,  8     10   26,
       9     10   34,  10    11   17,  11    11   28,  12    12   14,
       13    12   20,  14    12   24,  15    12   28,  16    13   26,
       17    13   34,  18    13   34,  19    13   46,  20    14   26,
       21    14   36,  22    14   60,  23    14   80,  24    15   20,
       25    15   26,  26    15   54,  27    16   32,  28    16   40,
       29    17   32,  30    17   40,  31    17   50,  32    18   42,
       33    18   56,  34    18   76,  35    18   84,  36    19   36,
       37    19   46,  38    19   68,  39    20   32,  40    20   48,
       41    20   52,  42    20   56,  43    20   64,  44    22   66,
       45    23   54,  46    24   70,  47    24   92,  48    24   93,
       49    24  120,  50    25   85 ];


 cnam = [" speed dist "];
 xy = xy[,2:3];
 xy = cname(xy,cnam); /* print "XY=",xy; */
```

```
optn = cons(5,1,.);
optn = [   1 ,   /* ipri */
           3 ,   /* maxit */
        .666 ,   /* f */
           . ];  /* delt */
< oxy,res > = lowess(xy,optn);
print "R Example:",oxy;
print "Res=",res;
```

```
          ****************************************
          Robust Locally Weighted Regression (LOWESS)
          ****************************************
```

N	X	Y	YPredict	Residual	RobWeight
1	4.00000000	2.00000000	4.96545928	-2.96545928	0.99268268
2	4.00000000	10.0000000	4.96545928	5.03454072	0.97703389
3	7.00000000	4.00000000	13.1244950	-9.12449504	0.92813284
4	7.00000000	22.0000000	13.1244950	8.87550496	0.93149621
5	8.00000000	16.0000000	15.8586334	0.14136662	0.99998479
6	9.00000000	10.0000000	18.5796905	-8.57969051	0.93561391
7	10.0000000	18.0000000	21.2803126	-3.28031258	0.99019598
8	10.0000000	26.0000000	21.2803126	4.71968742	0.98091853
9	10.0000000	34.0000000	21.2803126	12.7196874	0.86356976
10	11.0000000	17.0000000	24.1292771	-7.12927715	0.95453590
...					
41	20.0000000	52.0000000	56.4912241	-4.49122409	0.97904621
42	20.0000000	56.0000000	56.4912241	-0.49122409	0.99930662
43	20.0000000	64.0000000	56.4912241	7.50877591	0.95586277
44	22.0000000	66.0000000	67.5858242	-1.58582423	0.99630596
45	23.0000000	54.0000000	73.0796953	-19.0796953	0.69066036
46	24.0000000	70.0000000	78.6431636	-8.64316355	0.92721896
47	24.0000000	92.0000000	78.6431636	13.3568364	0.86003749
48	24.0000000	93.0000000	78.6431636	14.3568364	0.83836390
49	24.0000000	120.000000	78.6431636	41.3568364	0.07004493
50	25.0000000	85.0000000	84.3286981	0.67130190	0.99998975

5.16 Function `mempsd`

xyp = mempsd(coef,resi<,optn>)

Purpose: This function is based on the `mem_psd` Matlab function by P. V. Lanspeary (2006): The `mempsd` function calculates the power spectrum of the autoregressive filter

$$x_n = e_n + \sum_{k=0}^{ord} coef_k * x_{n-k}$$

where x_n is the filter output and e_n is the white noise input. For method=1, causes the spectrum to be calculated by FFT. Otherwise, the spectrum is calculated as a polynomial. Note that it is more computationally efficient to use the FFT method if length of the filter is not much smaller than the number of frequency values. The spectrum is scaled so that spectral energy between zero frequency and the Nyquist frequency is the same as the time-domain energy (i.e. mean square of the signal).

Input: The outputs of the `burg` function may be used for inputs of `mempsd` function.

coef must be a $p + 1$ vector of AR coefficients

resi must be a real scalar, the moving-average coefficient of the AR filter

optn this is either a scalar or a vector of options which should be initialized with missing values for defaults:

1. (int) amount of printed output (default is no output);
2. (int) method=1: FFT algorithm; method=2: polynomial algorithm, default=1;
3. (int) number of windows/frequencies, default=256;
4. (real) sampling frequency in Hertz, default=1.;

Output: The only return argument `xyp` is a $K \times 2$ matrix of the (x, y) values of the power spectrum periodogram.

Restrictions: 1. Currently neither missing, complex, nor string data are permitted.

Relationships: armcov(), burg(), pwelch()

Examples: 1. First run `burg` algorithm for AR coefficients:

```
xd  = [ 0.8147 0.9058 0.1270 0.9134 0.6324
        0.0975 0.2785 0.5469 0.9575 0.9649
        0.1576 0.9706 0.9572 0.4854 0.8003
        0.1419 0.4218 0.9157 0.7922 0.9595 ];
```

```
print "BURG-Filter: all nord";
optn = 2; nord = 3;
< coef,resi > = burg(xd,nord,optn);
print "ALL: COEFF=",coef;
print "ALL: RESI=",resi;
```

```
          Burg Method for Estimating AR Coefficients
          ******************************************

      Order    FPE_Crit    AIC_Crit    Residual   Terminate

          2  0.24507333 -1.40847869  0.18114116
          3  0.21122725 -1.56028580  0.14081817
```

```
ALL: COEFF=
   |         1         2         3         4
---------------------------------------------
 1 |   1.00000  -0.33494  -0.11590  -0.47181
```

```
ALL: RESI= 0.1408
```

2. Use FFT method `optn[2]=1`:

```
imet = 1; nwin = 20;
optn = [ 2 imet nwin ];
qsd = mempsd(coef,resi,optn);
print "FFT MEMPSD: psd=",qsd;
```

```
FFT MEMPSD: psd=
   |         1         2
-----------------------
 1 |   0.00000    47.075
 2 |    0.0250    2.5452
 3 |    0.0500   0.69972
 4 |    0.0750   0.34351
 5 |    0.1000   0.22086
 6 |   0.12500   0.16824
 7 |   0.15000   0.14549
 8 |   0.17500   0.14001
 9 |   0.20000   0.14868
10 |   0.22500   0.17375
11 |   0.25000   0.22282
12 |   0.27500   0.30772
```

```
13 |   0.30000   0.42296
14 |   0.32500   0.48068
15 |   0.35000   0.39747
16 |   0.37500   0.27546
17 |   0.40000   0.19116
18 |   0.42500   0.14256
19 |   0.45000   0.11589
20 |   0.47500   0.10256
21 |   0.50000    0.0985
```

3. Use Polynomial method `optn[2]=2`:

```
imet = 2; nwin = 20;
optn = [ 2 imet nwin ];
qsd = mempsd(coef,resi,optn);
print "POLY MEMPSD: psd=",qsd;
```

```
POLY MEMPSD: psd=
   |         1         2
-------------------------
 1 |   0.00000    47.075
 2 |    0.0250    2.5452
 3 |    0.0500   0.69972
 4 |    0.0750   0.34351
 5 |    0.1000   0.22086
 6 |   0.12500   0.16824
 7 |   0.15000   0.14549
 8 |   0.17500   0.14001
 9 |   0.20000   0.14868
10 |   0.22500   0.17375
11 |   0.25000   0.22282
12 |   0.27500   0.30772
13 |   0.30000   0.42296
14 |   0.32500   0.48068
15 |   0.35000   0.39747
16 |   0.37500   0.27546
17 |   0.40000   0.19116
18 |   0.42500   0.14256
19 |   0.45000   0.11589
20 |   0.47500   0.10256
21 |   0.50000    0.0985
```

5.17 Function `nsimplex`

n = nsimplex(p,n)

Purpose: The `nsimplex` function computes the number of points on a (p, n) simplex lattice; that is the number of p-part compositions of n.

Input: **p** should be integer scalar of integer vector with m entries

n should be integer scalar of integer vector with m entries

Output: Returns integer scalar or vector with m entries.

Restrictions: 1. Input data should not contain missing values.

Relationships: xsimplex()

Examples:

```
n0 = nsimplex(3,5);  print "N0=", n0;

N0= 21

  n1 = nsimplex(2,3);  print "N1=", n1;

N1= 4

  n2 = nsimplex(2,5);  print "N2=", n2;

N2= 6

  n3 = nsimplex(5,2);  print "N3=", n3;

N3= 15
```

5.18 Function `polyfit`

< coef,sse > = polyfit(x,y,d<,opt>)

Purpose: The `polyfit` function estimates the $d+1$ coefficients of polynomials

$$y = c_0 + c_1 x + \ldots + c_d x^d$$

of degree d. The fit criterion is unweighted least squares.

Input: **x** must be a N vector or a $N \times n_x$ matrix of x coordinate data.

y must be a N vector or a $N \times n_y$ matrix of y coordinate data.

d specifies the degree of the polynomial.

opt is an optional vector of options with the folowing meaning:

1. specifies the amount of printed output, default is `opt[1]=0`, which means no printed output.
2. a nonzero value specifies the fit of Legendre polynomials, default is `opt[2]=0`.
3. is valid only for $n_y = n_x > 1$ for pairwise treatment of columns of `x` and `y`, default is `opt[3]=0`.

Output: **coef**

- for `opt[3]=0`: `coef` is a $n_y * n_x \times d + 1$ matrix of the coefficients of the $n_y * n_x$ polynomials of for all pairs $(x_i, y_j), i = 1, \ldots, n_x, j = 1, \ldots, n_y$ of data columns.
- for `opt[3]=1`: `coef` is a $nx \times d + 1$ matrix of the coefficients of the n_x polynomials for all pairs $(x_i, y_i), i = 1, \ldots, n_x$ of data columns.

sse

- for `opt[3]=0`: `sse` is a $n_y \times n_x$ matrix of the sum of squares errors for the $n_x \times n_y$ data column pairs (x_i, y_j).
- for `opt[3]=1`: `sse` is a nx vector of the sum of squares errors for the n_x data column pairs (x_i, y_i).

Restrictions:

1. The input arguments `x` and `y` must have the same number of rows.
2. The input arguments `x` and `y` must contain nonmissing numeric data.
3. For `opt[3]=1`, the input arguments `x` and `y` must the same number of columns.

Relationships: polynom(), polyval()

Examples: 1. Example in Matlab 4 Manual, p. 531:

```
x = [ 0.:.1:2.5 ];
y = erf(x);
< coef,sse > = polyfit(x,y,6,2);
print "COEF=",coef;
print "SSE=",sse;
```

```
          Coefficients of Polynomial(s) with Degree 6

 4.41e-004 + 1.1064460*x + 0.1471041*x^2 - 0.7434628*x^3
+ 0.4217362*x^4 - 0.0982996*x^5 + 0.0084194*x^6

              Sum of Squares Error: 2.04571e-006
```

2. Process multiple columns of x and y:

```
print "Process all nx x ny columns of(x,y)";
optn = [ 2, 0, 0 ];
x = [ 0.:.1:2.5 ,
      .5:.1:3.0 ,
      1.:.1:3.5 ]';
y = erf(x);
< coef,sse > = polyfit(x,y,6,optn);
print "COEF=",coef;
print "SSE=",sse;
```

```
          Coefficients of Polynomial(s) with Degree 6

Y_1 X_1   4.41e-004 + 1.1064460*x + 0.1471041*x^2 - 0.7434628*x^3
        + 0.4217362*x^4 - 0.0982996*x^5 + 0.0084194*x^6

Y_1 X_2  -0.3935111 + 0.1585795*x + 2.0256702*x^2 - 1.8537326*x^3
        + 0.6990578*x^4 - 0.1235577*x^5 + 0.0084194*x^6

Y_1 X_3   0.3130172 - 3.6471094*x + 6.0171960*x^2 - 3.5817907*x^3
        + 1.0395246*x^4 - 0.1488158*x^5 + 0.0084194*x^6

Y_2 X_1   0.5202980 + 0.8862026*x - 0.4760669*x^2 - 0.0959656*x^3
        + 0.1955513*x^4 - 0.0727097*x^5 + 0.0090158*x^6

Y_2 X_2  -0.0151893 + 1.1681074*x + 0.0605480*x^2 - 0.6913821*x^3
        + 0.4111349*x^4 - 0.0997571*x^5 + 0.0090158*x^6

Y_2 X_3  -0.4687291 + 0.3505909*x + 1.8474722*x^2 - 1.7855842*x^3
```

```
         + 0.6943369*x^4 - 0.1268045*x^5 + 0.0090158*x^6

Y_3 X_1   0.8425695 + 0.4217127*x - 0.4601787*x^2 + 0.2544388*x^3
         - 0.0714750*x^4 + 0.0084799*x^5 - 1.33e-004*x^6

Y_3 X_2   0.4801294 + 1.1111329*x - 0.9597737*x^2 + 0.4189203*x^3
         - 0.0931723*x^4 + 0.0088779*x^5 - 1.33e-004*x^6

Y_3 X_3  -0.3738484 + 2.4344823*x - 1.7391345*x^2 + 0.6277915*x^3
         - 0.1158646*x^4 + 0.0092759*x^5 - 1.33e-004*x^6


              Sum of Squares Error for 9 Polynomials

                     X_1          X_2          X_3
           Y_1   2.0457e-006  2.0457e-006  2.0457e-006
           Y_2   8.0874e-007  8.0874e-007  8.0874e-007
           Y_3   1.8203e-007  1.8203e-007  1.8203e-007
```

3. Process column pairs of x and y:

```
print "Process pairwise columns of(x,y)";
optn = [ 2, 0, 1 ];
x = [ 0.:.1:2.5 ,
      .5:.1:3.0 ,
      1.:.1:3.5 ]';
y = erf(x);
< coef,sse > = polyfit(x,y,6,optn);
print "COEF=",coef;
print "SSE=",sse;
```

```
           Coefficients of Polynomial(s) with Degree 6

X_1   4.41e-004 + 1.1064460*x + 0.1471041*x^2 - 0.7434628*x^3
     + 0.4217362*x^4 - 0.0982996*x^5 + 0.0084194*x^6


X_2  -0.0151893 + 1.1681074*x + 0.0605480*x^2 - 0.6913821*x^3
     + 0.4111349*x^4 - 0.0997571*x^5 + 0.0090158*x^6


X_3  -0.3738484 + 2.4344823*x - 1.7391345*x^2 + 0.6277915*x^3
     - 0.1158646*x^4 + 0.0092759*x^5 - 1.33e-004*x^6


              Sum of Squares Error for 3 Polynomials
```

```
                1
X_1   2.0457e-006
X_2   8.0874e-007
X_3   1.8203e-007
```

5.19 Function `polyval`

y = polyval(coef,x,<,opt>)

Purpose: The `polyval` function evaluates polynomials

$$y = c_0 + c_1 x + \ldots + c_d x^d$$

of degree d with $d+1$ coefficient vector `coef` for $N \times n$ data `x`.

Input: **coef** is a $d+1$ vector of coefficients of a polynomial of degree d.

x specifies a $N \times n$ matrix of x data.

opt is an optional vector of options with the folowing meaning:

1. specifies the amount of printed output, default is `opt[1]=0`, which means no printed output.
2. a nonzero value specifies the evaluation of Legendre polynomials, default is `opt[2]=0`.

Output: The result is a $N \times n$ matrix of y data.

Restrictions: 1. The input argument `coef` and `x` must contain nonmissing numeric data.

Relationships: polynom(), polyfit()

Examples: 1. Example in Matlab 4 Manual, p. 531:

```
x = [ 0.:.1:2.5 ];
y = erf(x);
< coef,sse > = polyfit(x,y,6,2);
print "COEF=",coef;
print "SSE=",sse;
```

For the output see the document for the `polyfit` function.

```
ymod = polyval(coef,x,2);
print "YMOD=",ymod;
sse = ssq(y - ymod);
print "SSE=",sse;
```

```
            Function Values Y for Polynomials
            *********************************

                Dense Row Vector (ncol=26)

R |          1          2          3          4          5
```

```
       4.41e-004  0.1118546  0.2223107  0.3287242  0.4287990

R |           6          7          8          9         10
       0.5209256  0.6040843  0.6777548  0.7418310  0.7965431

R |          11         12         13         14         15
       0.8423844  0.8800456  0.9103539  0.9342189  0.9525845

R |          16         17         18         19         20
       0.9663861  0.9765154  0.9837896  0.9889277  0.9925325

R |          21         22         23         24         25
       0.9950788  0.9969074  0.9982257  0.9991136  0.9995360

R |          26
       0.9993615
```

2. Process multiple columns of `x` and `y`:

```
print "Process all nx x ny columns of(x,y)";
optn = [ 2, 0, 0 ];
x = [ 0.:.1:2.5 ,
      .5:.1:3.0 ,
      1.:.1:3.5 ]';
y = erf(x);
< coef,sse > = polyfit(x,y,6,optn);
print "COEF=",coef;
print "SSE=",sse;
```

For the output see the document for the `polyfit` function.

```
ymod = polyval(coef[1,],x,2);
print "YMOD=",ymod;
sse = ssq(y - ymod);
print "SSE=",sse;
```

```
  Function Values Y for Polynomials
  *********************************

       Dense Matrix (26 by 3)

    |         X_1        X_2        X_3
  ------------------------------------
  1 |   4.41e-004  0.5209256  0.8423844
  2 |   0.1118546  0.6040843  0.8800456
```

3	0.2223107	0.6777548	0.9103539
4	0.3287242	0.7418310	0.9342189
5	0.4287990	0.7965431	0.9525845
6	0.5209256	0.8423844	0.9663861
7	0.6040843	0.8800456	0.9765154
8	0.6777548	0.9103539	0.9837896
9	0.7418310	0.9342189	0.9889277
10	0.7965431	0.9525845	0.9925325
11	0.8423844	0.9663861	0.9950788
12	0.8800456	0.9765154	0.9969074
13	0.9103539	0.9837896	0.9982257
14	0.9342189	0.9889277	0.9991136
15	0.9525845	0.9925325	0.9995360
16	0.9663861	0.9950788	0.9993615
17	0.9765154	0.9969074	0.9983867
18	0.9837896	0.9982257	0.9963665
19	0.9889277	0.9991136	0.9930511
20	0.9925325	0.9995360	0.9882287
21	0.9950788	0.9993615	0.9817739
22	0.9969074	0.9983867	0.9737027
23	0.9982257	0.9963665	0.9642335
24	0.9991136	0.9930511	0.9538538
25	0.9995360	0.9882287	0.9433933
26	0.9993615	0.9817739	0.9341030

5.20 Function pppd

pts = pppd(type,mu,std,skew,kurt—bnd<,pri>)

Purpose: The pppd function computes percentage points of Pearson distributions. It is based on the work of Pan(2009).

Input: **type** This should be an integer defining the type of specification of argument 5:

type=1 Kurtosis β_2 is specified.

type=2 Kurtosis is estimated from left bound.

type=3 Kurtosis is estimated from right bound.

mu specification of mean μ, can be scalar or a vector of n_1 numerical values;

std specification of standard deviation σ, can be scalar or a vector of n_2 numerical values;

skew specification of skewness $\sqrt{\beta_1}$, can be scalar or a vector of n_3 numerical values;

kurt|bnd depending on value of type input argument: either kurtosis (type=1), left bound (type=2), or right bound (type=3) is specified; can be scalar or a vector of n_4 numerical values;

pri this optional input argument should be an integer specifiying the amount of printed output

Output: The vector or matrix pts has $N = n_1 * n_2 * n_3 * n_4$ rows (referring to the product of input arguments) and 11 columns referring to the most commonly used percentiles: 1.0, 2.5, 5.0, 10.0, 25.0, 50.0, 75.0, 90.0, 95.0, 97.5, and 99.0 percent.

Restrictions: 1. No missing values are permitted at input.

2. The value for skew must be in $[-2, 2]$.

Relationships:

Examples: 1. Examples by Pan (2009):

```
print "First Example with Type=1: kurtosis is given";
p1 = pppd(1,0.,1.,1.3,4.2,1);
print "P1=",p1;
```

```
          Percentage Points for Pearson Curves
           Type 1: With the First Four Moments
        Mean=0 StdDev=1 Skewness=1.3 Kurtosis=4.2
```

```
         1.0%      2.5%      5.0%     10.0%     25.0%     50.0%
   -0.978913 -0.973849 -0.964978 -0.938281 -0.795986 -0.341188
            75.0%     90.0%     95.0%     97.5%     99.0%
          0.504230  1.498740  2.112209  2.609179  3.117556
```

```
mu = [ 0. .1 ];
p11 = pppd(1,mu,1.,1.3,4.2,1);
print "P11=",p11;
```

```
           Percentage Points for Pearson Curves
           ************************************

                 Dense Matrix (2 by 11)

    |    PP__1.0%   PP__2.5%   PP__5.0%   PP_10.0%   PP_25.0%
  ------------------------------------------------------------
MU1 |  -0.9789127 -0.9738491 -0.9649784 -0.9382810 -0.7959864
MU2 |  -0.8789127 -0.8738491 -0.8649784 -0.8382810 -0.6959864

    |    PP_50.0%   PP_75.0%   PP_90.0%   PP_95.0%   PP_97.5%
  ------------------------------------------------------------
MU1 |  -0.3411880  0.5042296  1.4987399  2.1122088  2.6091794
MU2 |  -0.2411880  0.6042296  1.5987399  2.2122088  2.7091794

    |    PP_99.0%
  ---------------
MU1 |   3.1175558
MU2 |   3.2175558
```

```
print "Second Example with Type=1: kurtosis is given";
p2 = pppd(1,0.08333,0.05,1.619,6.7905,1);
print "P2=",p2;
```

```
            Percentage Points for Pearson Curves
             Type 1: With the First Four Moments
    Mean=0.08333 StdDev=0.05 Skewness=1.619 Kurtosis=6.7905

         1.0%      2.5%      5.0%     10.0%     25.0%     50.0%
    0.025482  0.027304  0.029822  0.034236  0.046568  0.070089
            75.0%     90.0%     95.0%     97.5%     99.0%
          0.105887  0.150032  0.182180  0.213420  0.253057
```

```
print "Third Example with Type=2: left bound is given";
```

```
p3 = pppd(2,0.,1.,1.,-1.,1);
print "P3=",p3;
```

```
            Percentage Points for Pearson Curves
      Type 2: With First Three Moments and Left Boundary
            Mean=0 StdDev=1 Skewness=1 Boundary=-1
                     Estimated Kurtosis: 3

        1.0%      2.5%      5.0%     10.0%     25.0%     50.0%
   -0.999573 -0.997705 -0.993994 -0.975301 -0.843870 -0.347136
            75.0%     90.0%     95.0%     97.5%     99.0%
          0.611352  1.591630  2.085640  2.413299  2.676919
```

```
print "Fourth Example with Type=3: right bound is given";
p4 = pppd(3,0.,1.,.5,2.932345,1);
print "P4=",p4;
```

```
            Percentage Points for Pearson Curves
      Type 3: With First Three Moments and Right Boundary
         Mean=0 StdDev=1 Skewness=0.5 Boundary=2.93235
                     Estimated Kurtosis: 2.4

        1.0%      2.5%      5.0%     10.0%     25.0%     50.0%
   -1.500763 -1.450913 -1.365149 -1.223512 -0.827682 -0.141415
            75.0%     90.0%     95.0%     97.5%     99.0%
          0.707431  1.457671  1.848071  2.136932  2.397396
```

5.21 Function `pwelch`

xyp = pwelch(xdat<,optn>)

Purpose: This function is based on the `welch_psd` Matlab function by P. V. Lanspeary (2006): The `pwelch` function estimates the power spectrum of a time-series signal by the periodogram (FFT) method. The "signal" is divided into segments of length equal to the length of "window", and each segment is multiplied by "window" before (optional) zero-padding and calculating its FFT. The power spectrum is the mean square of the FFTs, scaled so that the area under the power spectrum is the same as the mean square of the signal. The function `pwelch` is very similar to the `Wiener1` function in TISEAN.

Input: **xdat** an N vector of time series data or a $N \times n$ matrix with n univariate time series. The first entry is the zero lag coefficient which is always one.

optn this is either a scalar or a vector of options which should be initialized with missing values for defaults:

1. (int) amount of printed output (default is no output);
2. (int) method: =0: similar to Lanspeary (2006); =1: similar to TISEAN; default is zero;
3. (int) only for method=0: length of FFT: default: next power of two larger than $\sqrt{N}$
4. only for method=0: (real) sampling frequency in Hertz: default = 1.;

Output: The only return argument `xyp` is a $K \times 2$ matrix of the (x, y) values of the power spectrum periodogram.

Restrictions: 1. Currently neither missing, complex, nor string data are permitted.

Relationships: armcov(), mempsd(), burg()

Examples:

```
xd  = [ 0.8147 0.9058 0.1270 0.9134 0.6324
  0.0975 0.2785 0.5469 0.9575 0.9649
  0.1576 0.9706 0.9572 0.4854 0.8003
  0.1419 0.4218 0.9157 0.7922 0.9595 ];
```

Default length of FFT is 8:

1.
```
print "PWELCH by LANSPEARY, 2006";
optn = [ 2  0 ];
xypnt = pwelch(xd,optn);
print "XYPNT=",xypnt;
```

```
XYPNT=
   |         1         2
------------------------
 1 |   0.00000   0.07829
 2 |   0.12500   0.10582
 3 |   0.25000   0.29042
 4 |   0.37500   0.43757
 5 |   0.50000   0.05866
```

2. Specified length of FFT is 20:

```
print "PWELCH by LANSPEARY";
optn = [ 2  0 20 ];
xypnt = pwelch(xd,optn);
print "XYPNT=",xypnt;
```

```
XYPNT=
   |         1         2
------------------------
 1 |   0.00000    2e-031
 2 |   0.05000   0.05527
 3 |   0.10000   0.43850
 4 |   0.15000   0.02235
 5 |   0.20000   0.36416
 6 |   0.25000   0.26770
 7 |   0.30000   0.12230
 8 |   0.35000   0.33969
 9 |   0.40000   0.33578
10 |   0.45000   0.07622
11 |   0.50000   0.09262
```

5.22 Function `rmult`

z = rmult(n,p)

Purpose: The `rmult` function generates random samples from (n, p) multinomial distributions.

Input: **n** a positive integer $n > 0$

p a probability vector with m entries in $(0, 1)$ totaling unity

Output: m vector of integers which sums up to n

Restrictions: 1. Neither n nor p must contain missing values.

Relationships: mrand(), dmnom()

Examples:

```
n1 = 100; p1 = [ .3 .1 .6 ];
r1 = rmult(n1,p1); print "R1=",r1;
```

```
R1=
   |    1    2    3
-------------------
 1 |   26    3   71
```

```
n2 = 20; p2 = [ .1 .1 .8 ];
r2 = rmult(n2,p2); print "R2=",r2;
```

```
R2=
   |    1    2    3
-------------------
 1 |    2    4   14
```

```
n3 = 10; p3 = [ .5 .2 .3 ];
r3 = rmult(n3,p3); print "R3=",r3;
```

```
R3=
   |   1   2   3
----------------
 1 |   6   3   1
```

5.23 Function `sample`

i = sample(nsmp,nobs|prob<,optn>)

Purpose: The `sample` function implements equal and uenequal probabilty sampling with or without replacement.

Input: **nsmp** the size of the sample drawn from the population

nobs|prob 1. an integer nobs specifies the size of the population for equal probability sampling

2. a real vector of probabilities `prob[nobs]` specifies the weights for unequal probability sampling. The probabilities should be in $(0, 1)$ and add up to unity. If the vector is not adding up to unity it is rescaled correspondingly.

optn an option vector that should be initialized with missing values for defaults:

1. `optn[1]=0`: specifies sampling without replacement; `optn[1]=1`: specifies sampling with replacement (default).
2. integer seed value for random generator (default is time of day).
3. kind of random generator: =0: KISS by Marsaglia (default), =1: L'Ecuyer, otherwise: RANUNI

There should be $nsmp \leq nobs$ for sampling without replacement.

Output: The only return argument is an $nsmp$ vector of integers in $[1, \ldots, nobs]$ identifying the sample drawn from the population. The sample is not sorted.

Restrictions: 1. The first two input arguments cannot contain missing values.

2. There should be $nsmp \leq nobs$ for sampling without replacement.

Relationships: sampmd()

Examples: 1. Equal Probability Sampling without replacement:

```
nobs = 100; nsmp = 20;
opt = [   0 ,    /* without rep */
        123 ];   /* seed */
s1 = sample(nsmp,nobs,opt); print "S1=",s1';
```

```
S1=
   |    1    2    3    4    5    6    7    8    9   10
---------------------------------------------------------
 1 |   99   45   83   30   76   89   41   94    8   73
```

```
   |   11   12   13   14   15   16   17   18   19   20
---------------------------------------------------------
 1 |   27   28   93   66   55   59   26   35   65   36
```

2. Equal Probability Sampling with replacement:

```
nobs = 100; nsmp = 20;
opt = [   1 ,    /* with rep */
        123 ];   /* seed */
s2 = sample(nsmp,nobs,opt); print "S2=",s2‘;
```

```
S2=
   |    1    2    3    4    5    6    7    8    9   10
---------------------------------------------------------
 1 |   99   45   84   31   80   94   43   45    9   81

   |   11   12   13   14   15   16   17   18   19   20
---------------------------------------------------------
 1 |   30   31   47   76   64   69   31   43   79   44
```

3. Unequal Probability Sampling without replacement:

```
nobs = 100; nsmp = 20;
t1 = .5; t2 = .5 / (nobs - 1.);
prob = cons(nobs,1,t2); prob[1] = t1;
opt = [   0 ,  /* without rep */
        123 ]; /* seed */
s3 = sample(nsmp,prob,opt); print "S3=",s3‘;
```

```
S3=
   |    1    2    3    4    5    6    7    8    9   10
---------------------------------------------------------
 1 |   98    1   84   31   78   93   43   44    9   79

   |   11   12   13   14   15   16   17   18   19   20
---------------------------------------------------------
 1 |   29   32   48   74   63   68   28   41   80   45
```

4. Unequal Probability Sampling with replacement:
   The following example shows how the first observation is preferred by the large probability $p_1 = .5$:

```
  nobs = 100; nsmp = 20;
  t1 = .5; t2 = .5 / (nobs - 1.);
  prob = cons(nobs,1,t2); prob[1] = t1;
  opt = [   1 ,  /* without rep */
          123 ]; /* seed */
  s4 = sample(nsmp,prob,opt); print "S4=",s4‘;
```

```
S4=
   |    1    2    3    4    5    6    7    8    9   10
---------------------------------------------------------
 1 |   98    1   69    1   59   87    1    1    1   61

   |   11   12   13   14   15   16   17   18   19   20
---------------------------------------------------------
 1 |    1    1    1   52   29   39    1    1   58    1
```

5.24 Function `sortp`

b = sortp(a,ind)

Purpose: The `sortp` function sorts partially the values of an m vector a or the n columns of an $m \times n$ matrix a wrt. an index `ind` with $1 \leq ind \leq m$. For simplicity lets assume that **a** is an m vector. Then partial sorting means, after sorting the entry $b[ind]$ is at the correctly sorted location, all values at the left, $b[i] < b[ind], i < ind$, and all values at the right, $b[j] > b[ind], j > ind$. That also means, that the $ind - 1$ values at the left of $b[ind]$ are not necessarily sorted among themslves as well as the $n - ind$ values at the right of $b[ind]$ are not necessarily sorted among themslves. Partial sorting is useful for obtaining quantiles, like median, quartiles or percentiles.

Input: **a** is either an m vector or an $m \times n$ matrix of int or real values. If `a` is a $m \times n$ matrix, it is processed columnwise.

ind this should be an int scalar of a valid index location in `a`, i.e. $1 \leq ind \leq n$.

Output: The vector (or matrix) `b` comtains all of the input `a`, however in a partially sorted order wrt. index `ind`.

Restrictions: 1. Missing values are shuffled to the end of the output object `b`.

Relationships: branks(), ranktie(), sortrow(), subscript operators: <, <:, < |, <!, >, >:, > |, >!

Examples: 1. :

```
print "Example by Cleveland";
xy = [ 1  2   3  4   5  10#6  8  10  12  14  50 ,
       18  2  15  6  10  4  16  11  7  3  14  17
       20  12  9  13  1  8  5  19 ];


y = xy[,2];
b = sortp(y,nr/2); print "B=",b;
med1 = b[nr/2];
med2 = univar(y,"med");
print "Med1=",med1," Med2=",med2;
```

The entry on index location 10 is at the correct place, since all entries before are smaller and all entries afterward are larger than 10. However the entries with indices 1 to 9 and those with indices 11 to 20 are not sorted among themselves:

```
Test SORTP

B=
   |    1
---------
 1 |    1
 2 |    2
 3 |    3
 4 |    6
 5 |    8
 6 |    4
 7 |    5
 8 |    7
 9 |    9
10 |   10
11 |   14
12 |   13
13 |   11
14 |   12
15 |   20
16 |   17
17 |   18
18 |   15
19 |   16
20 |   19

Med1= 10 Med2= 10.500
```

5.25 Function `stand`

<z,locscal> = stand(x<,sopt<,optn<,ls>>>)

sopt="mea"|"med"|"sum"|"euc"|"ust"|"std"|"ran"| "mid"|"iqr"|"mad"|
"loo"|"lpm"|"spa"|"biw"|"hub"|"wav"|"inp"|"rev"

Purpose: The `stand` function standardizes a $m \times n$ data matrix $\mathbf{X}$ columnwise, i.e.

$$z_{ij} = \frac{x_{ij} - l_j}{s_j} \quad , \quad i = 1, \ldots, m, \quad j = 1, \ldots, n$$

where l_j is the location and s_j the scale value of column j. Only the `"rev"erse` option performs the inverse operation

$$z_{ij} = x_{ij} s_j + l_j \quad , \quad i = 1, \ldots, m, \quad j = 1, \ldots, n$$

The `stand()` function is very much related to the `univar()` function and shares a good part of the code for computing the location and scale values. Therefore, some of the options of the two functions are the same.

The following location and scaling options are similar to those of PROC STDIZE in SAS/STAT and the STDIZ SAS macro (Sarle, 1995):

`sopt`	**Location**	**Scale**
mea(n)	mean	1
med(ian)	median	1
sum	0	sum
euc(len)	0	Euclidean length
ust(d)	0	Uncorrected Std. Deviation
std	mean	Standard Deviation
ran(ge)	minimum	range
mid(range)	midrange	range/2
max(abs)	0	max absolute value
iqr	median	interquartile range
mad	median	abs. deviation from median
biw(c)	biweight 1-step M-estim	biweight A-estimate
hub(er)(c)	Huber 1-step M-estimate	Huber A estimate
wav(e)(c)	Wave 1-step M-estimate	Wave A estimate
agk(p)	mean	AGK (ACECLUS) estimate
spa(cing)(p)	mid-minimum-spacing	minimum spacing
lpm(p)	*Lp* location	*Lp* scale (FASTCLUS)
inp	input	input
rev	reverse use of input	reverse use of input

Only the first three characters of the first table column are used for the `sopt` input argument. For the *Lp* metric the parameter $p \geq 1$ and the

location for $p = 1$ is the Median, for $p = 2$ is the mean, the scale for $p = 1$ is the MAD (*Median Absolute Deviation*), for $p = 2$ is the standard deviation `ust(d)`.

Input: **x** numeric (int or real) $m \times n$ input matrix.

sopt three character string argument, see first column of table above. The following are valid specifcations: `sopt="mea", "med", "sum", "euc", "ust", "std", "ran", "mid", "iqr", "mad", "loo", "lpm", "spa", "biw", "hub", "wav", "inp", "rev"`

optn two-column matrix specifying additional options and parameters c or p for the measures, see table below.

ls $2 \times n$ matrix specifying locations in its first row and scales in the second row for all n coluumns; valid only with the `"inp"` and the `"rev"` string options.

Options Matrix Argument: The option argument is specified in form of a two column matrix:

Option Name	Column 2	Meaning
`"nobs"`	real	number of observations N, by default the number of function values returned by `func()`
`"init"`	string	specifying initial value for `"biw"`, `"hub"`, and `"wav"` optimizations
	"rob"	robust Median and MAD values, this is default
	"cla"	classic arithmetic mean and standard deviation
`"maxit"`	int	integer value speciying the mximum number of iterations for `"biw"`, `"hub"`, and `"wav"`
`"print"`	int	integer value specifying the amount of printed output (def=0: no printed output)
`pbiw`	real	parameter $c > 0$ for Tukey's biweight, def: $c = 6$
`phub`	real	parameter $k > 0$ for Huber, def: $k = 1.5$
`plpm`	real	parameter $p \geq 1$ for Lp, def: $p = 1$
`pspa`	real	parameter $0 < p < 1$ for minimum spacing, def:
`pwav`	real	parameter $c > 0$ for Andrew's wave, def: $c = 1.5$
`"vardef"`	string	definition of variance divisor; string may be "n", "df", "wgt", or "wdf".
`"vardiv"`	real	variance divisor s
`"vers"`	int	integer value specifying the version of the algorithm for `"biw"`, `"hub"`, and `"wav"`

Output: **z** transformed $m \times n$ matrix. (Note, for zero location methods ("sum", "euc", "ustd", "max") the sparsity pattern of `x` is preserved in `z`.)

locscal $2 \times n$ matrix of the location in its first row and scale in its second row. Together with the `"rev"` string option this output could be used to re-transform `z` to `x` in a subsequent step.

Restrictions: 1. Missing values in `x` are preserved in `z`.

2. Each column must have enough nonmissing values for computing the location and scale values.

Relationships: univar(), quantile()

Examples: 1. Heart Data (Hawkins, 1994): Perform `mean` standardization and reverse

```
< c1,lsc > = stand(aa,"mea");
print "\n Mean Standardization: Data:\n",c1;
print "\n Mean Standardization: LocScal:\n",lsc;
< c1,lsc > = stand(c1,"rev",.,lsc);
print "\n Reverse Standardization: Data:\n",c1;
print "\n Reverse Standardization: LocScal:\n",lsc;
```

```
 Mean Standardization: Data:

   |         1         2         3
-----------------------------------
 1 |    2.4417    1.8750   0.83333
 2 |    23.142    55.375    13.833
 3 |   -2.8583   -2.6250   -2.1667
 4 |  -0.85833   -8.1250  -0.16667
 5 |    5.1417    13.875    6.8333
 6 |   -1.8583   -21.125   -8.1667
 7 |    2.6417   0.37500   0.83333
 8 |   -17.858   -29.625   -16.167
 9 |   -3.3583   -5.1250   -2.1667
10 |   -16.858   -28.625   -6.1667
11 |   -7.3583   -17.125    1.8333
12 |    17.642    40.875    10.833


 Mean Standardization: LocScal:

            |     Var_1     Var_2     Var_3
-----------------------------------------
   Ari_Mean |    40.358    38.125    36.167
  UnitScale |   1.00000   1.00000   1.00000


 Reverse Standardization: Data:

   |         1         2         3
-----------------------------------
 1 |    42.800    40.000    37.000
 2 |    63.500    93.500    50.000
```

```
 3 |    37.500    35.500    34.000
 4 |    39.500    30.000    36.000
 5 |    45.500    52.000    43.000
 6 |    38.500    17.000    28.000
 7 |    43.000    38.500    37.000
 8 |    22.500    8.5000    20.000
 9 |    37.000    33.000    34.000
10 |    23.500    9.5000    30.000
11 |    33.000    21.000    38.000
12 |    58.000    79.000    47.000
```

```
 Reverse Standardization: LocScal:

            |     Var_1     Var_2     Var_3
-------------------------------------------
   Ari_Mean |    40.358    38.125    36.167
  UnitScale |   1.00000   1.00000   1.00000
```

2. Heart Data (Hawkins, 1994): The common `std` standardization:

```
 STD Standardization: Data:

   |         1         2         3
-----------------------------------
 1 |   0.20457   0.07196   0.10190
 2 |    1.9389    2.1251    1.6915
 3 |  -0.23948  -0.10074  -0.26494
 4 |  -0.07191  -0.31181  -0.02038
 5 |   0.43078   0.53247   0.83558
 6 |  -0.15570  -0.81070  -0.99862
 7 |   0.22133   0.01439   0.10190
 8 |   -1.4962   -1.1369   -1.9769
 9 |  -0.28137  -0.19668  -0.26494
10 |   -1.4124   -1.0985  -0.75406
11 |  -0.61650  -0.65719   0.22418
12 |    1.4781    1.5686    1.3247
```

```
 STD Standardization: LocScal:

           |     Var_1     Var_2     Var_3
-----------------------------------------
  Ari_Mean |    40.358    38.125    36.167
  Std_Dev  |    11.936    26.058    8.1779
```

3. Heart Data (Hawkins, 1994): Tukey's biweight standardization (optimization):

```
optn = [ "pbiw"  6.0 ];
< c4,lsc > = stand(aa,"biw",optn);
print "\n Tukeys Biweight Standardization for c=6:\n",c4;
print "\n Biweight Standardization: LocScal:\n",lsc;
```

```
 Tukeys Biweight Standardization for c=6:

   |         1         2         3
-----------------------------------
 1 |   0.24891   0.12440   0.09897
 2 |    2.0371    2.2822    1.7433
 3 |  -0.20894  -0.05709  -0.28049
 4 |  -0.03617  -0.27892  -0.02752
 5 |   0.48216   0.60839   0.85790
 6 |  -0.12256  -0.80324   -1.0394
 7 |   0.26619   0.06390   0.09897
 8 |   -1.5048   -1.1461   -2.0513
 9 |  -0.25214  -0.15792  -0.28049
10 |   -1.4184   -1.1057  -0.78645
11 |  -0.59769  -0.64191   0.22546
12 |    1.5620    1.6974    1.3639


 Biweight Standardization: LocScal:

           |     Var_1     Var_2     Var_3
----------------------------------------
  Biwgt_M  |    39.919    36.916    36.218
  Biwgt_A  |    11.576    24.794    7.9058
```

4. Heart Data (Hawkins, 1994): Tukey's biweight standardization (SAS one-step):

```
optn = [ "pbiw"  6.0 ,
         "vers"    2 ];
< c4,lsc > = stand(aa,"biw",optn);
print "\n SAS: Tukeys Biweight Standardization for c=6:\n",c4;
print "\n Biweight Standardization: LocScal:\n",lsc;
```

```
 SAS: Tukeys Biweight Standardization for c=6:

   |         1         2         3
```

```
---------------------------------
 1 |   0.36490   0.25475   0.07702
 2 |    2.1410    2.4383    1.5943
 3 |  -0.08985   0.07108  -0.27312
 4 |   0.08176  -0.15340  -0.03969
 5 |   0.59656   0.74453   0.77731
 6 |  -0.00404  -0.68399  -0.97341
 7 |   0.38206   0.19353   0.07702
 8 |   -1.3769   -1.0309   -1.9071
 9 |  -0.13275  -0.03095  -0.27312
10 |   -1.2911  -0.99010  -0.73998
11 |  -0.47595  -0.52073   0.19374
12 |    1.6691    1.8465    1.2442


 Biweight Standardization: LocScal:

           |     Var_1     Var_2     Var_3
-----------------------------------------
  Biwgt_M  |    38.547    33.758    36.340
  Biwgt_A  |    11.655    24.501    8.5679
```

5. Heart Data (Hawkins, 1994): Huber's by optimization:

```
optn = [ "phub"  1.5 ];
< c5,lsc > = stand(aa,"hub",optn);
print "\n Opt: Huber Standardization for k=1.5:\n",c5;
print "\n Huber Standardization: LocScal:\n",lsc;
```

```
 Huber Standardization for k=1.5:

   |         1         2         3
---------------------------------
 1 |   0.25409   0.19594   0.07830
 2 |    1.8977    2.4564    1.6376
 3 |  -0.16674   0.00581  -0.28155
 4 |  -0.00794  -0.22658  -0.04165
 5 |   0.46847   0.70297   0.79799
 6 |  -0.08734  -0.77585   -1.0012
 7 |   0.26997   0.13257   0.07830
 8 |   -1.3578   -1.1350   -1.9608
 9 |  -0.20644  -0.09982  -0.28155
10 |   -1.2784   -1.0927  -0.76134
11 |  -0.52405  -0.60684   0.19825
12 |    1.4610    1.8438    1.2778
```

```
 Huber Standardization: LocScal:

           |     Var_1     Var_2     Var_3
-------------------------------------------
  Huber_M  |    39.600    35.363    36.347
  Huber_A  |    12.594    23.668    8.3369
```

6. Heart Data (Hawkins, 1994): Huber's by SAS one-step:

```
   optn = [ "phub"  1.5 ,
            "vers"    2 ];
   < c5,lsc > = stand(aa,"hub",optn);
   print "\n SAS: Huber Standardization for k=1.5:\n",c5;
   print "\n Huber Standardization: LocScal:\n",lsc;
```

```
 SAS: Huber Standardization for k=1.5:

   |         1         2         3
-----------------------------------
 1 |   0.38347   0.26954   0.11516
 2 |    2.8641    2.4462    1.8261
 3 |  -0.25165   0.08646  -0.27967
 4 |  -0.01198  -0.13731  -0.01645
 5 |   0.70703   0.75775   0.90482
 6 |  -0.13182  -0.66621   -1.0693
 7 |   0.40744   0.20851   0.11516
 8 |   -2.0492   -1.0120   -2.1222
 9 |  -0.31157  -0.01526  -0.27967
10 |   -1.9293  -0.97135  -0.80611
11 |  -0.79091  -0.50347   0.24677
12 |    2.2050    1.8562    1.4313


 Huber Standardization: LocScal:

           |     Var_1     Var_2     Var_3
-------------------------------------------
  Huber_M  |    39.600    33.375    36.125
  Huber_A  |    8.3448    24.579    7.5982
```

7. Heart Data (Hawkins, 1994): Minimum spacing (Sarle, 1995):

```
   optn = [ "pspa"  .2 ];
   < c2,lsc > = stand(aa,"spa",optn);
   print "\n Minimum Spacing Standardization\n",c2;
```

```
   print "\n Minimum Spacing Standardization: LocScal:\n",lsc;
```

```
 Minimum Spacing Standardization

   |         1         2         3
-----------------------------------
 1 |    3.3667   0.50000   0.50000
 2 |    17.167    12.389    13.500
 3 |  -0.16667  -0.50000   -2.5000
 4 |    1.1667   -1.7222  -0.50000
 5 |    5.1667    3.1667    6.5000
 6 |   0.50000   -4.6111   -8.5000
 7 |    3.5000   0.16667   0.50000
 8 |   -10.167   -6.5000   -16.500
 9 |  -0.50000   -1.0556   -2.5000
10 |   -9.5000   -6.2778   -6.5000
11 |   -3.1667   -3.7222    1.5000
12 |    13.500    9.1667    10.500
```

```
 Minimum Spacing Standardization: LocScal:

           |     Var_1     Var_2     Var_3
---------------------------------------------
  MidMinSp |    37.750    37.750    36.500
  MinSpacg |    1.5000    4.5000   1.00000
```

5.26 Function `tslocfor`

forec = tslocfor(xdat,order<,optn>)

Purpose: The `tslocfor` function implements algorithm for forecasting time series data with a first or zeroth order local model by iterating an artificial trajectory. This function is similar to the `lfo-run` and `lzo-run` functions in the TISEAN package.

Input: **xdat** $N \times n$ matrix of univariate ($n = 1$) or multivariate ($n > 1$) time series data

order must be zero or one specifying the order of the model

optn must be a scalar or vector of options with missing values for default settings:

1. ipri: amount of printed output (default=0: no printed output)
2. delay for the embedding (def=1)
3. emb: embedding dimension (def=2)
4. F: length of prediction (def=1000)
5. eps0: neighborhood size to start with (def=0.001)
6. epsf: factor for increasing neighborhood seize (def=1.2)
7. minn: minimum number of neighbors for fit (def=30) must be larger than number of parameters to estimate
8. only for order=0: Q: dynamical noise (def=0)
9. only for order=0: seed for random generator (only valid when adding noise $Q > 0$.)

Note, for obtaining a nonsingular $X'X$ matrix for estimating the first order model `mnn` must be larger than the number of parameters `n * emb` of the model.

Output: The output is a $F \times 3 + n$ matrix with columns containing:

1. the row number: for first order model the forecast is checked whether it is in the range of the data; a negative row number indicates the forecast escapes the data region,
2. the number of neighborhood points on which the forecast model is based (must be ¿= minn)
3. the neighborhood size epsilon used for estimating the model
4. the n forecast values

Restrictions:

Relationships: tsloctst(), tstrans(), armcov(), arima(), tsmeas()

Examples: 1. Small example ($N = 20, n = 1$) for first order estimation:

```
xd1 = [ 0.9501  0.2311  0.6068  0.4860  0.8913
        0.7621  0.4565  0.0185  0.8214  0.4447
        0.6154  0.7919  0.9218  0.7382  0.1763
        0.4057  0.9355  0.9169  0.4103  0.8936 ];


optn = [ 2 ,   /* -V: ipri */
         1 ,   /* -d: dely */
         2 ,   /* -m: emb  */
         5 ,   /* -L: flen */
         . ,   /* -r: eps0 */
       1.2 ,   /* -f: epsf */
         5 ];  /* -k: minn */


print "lforun: order=1: minn=5, flen=5";
forec = tslocfor(xd1,1,optn);


            Forecasts for First Order Local Model
            *************************************

               N  Pnts     Epsilon  Forecast Values
               1     5  0.34182189  0.80539115
               2     5  0.34182189  0.39119369
               3     6  0.23737631  0.58589184
               4     5  0.28485158  0.78739464
               5     5  0.23737631  0.81363317
```

2. Small example ($N = 20, n = 1$) for zero order estimation:

```
optn = [ 2 ,   /* -V: ipri */
         1 ,   /* -d: dely */
         2 ,   /* -m: emb  */
         5 ,   /* -L: flen */
         . ,   /* -r: eps0 */
       1.2 ,   /* -f: epsf */
         5 ];  /* -k: minn */


print "lzorun: order=0: minn=5, flen=5";
forec = tslocfor(xd1,0,optn);


            Forecasts for Zero Order Local Model
            ************************************
```

N	Pnts	Epsilon	Forecast Values
1	5	0.34182189	0.77574000
2	5	0.35440094	0.48364000
3	5	0.28352075	0.60512000
4	5	0.30620241	0.62190000
5	5	0.24496193	0.64778000

5.27 Function `tsloctst`

< relerr,inderr > = tsloctst(xdat,order<,optn>)

Purpose: The `tsloctst` function implements algorithms for the error estimation of first or zeroth order local models. This function is similar to

- the `lfo-tst` and `lzo-tst`: if `minn` is specified
- the `lfo-ar` and `lzo-gm`: if `minn` is missing (not specified)

functions in the TISEAN package (the "f" and "z" corresponds to first and zeroth order models).

Input: **xdat** $N \times n$ matrix of univariate ($n = 1$) or multivariate ($n > 1$) time series data

order must be zero or one specifying the order of the model

optn must be a scalar or vector of options with missing values for default settings:

1. ipri: amount of printed output (default=0: no printed output)
2. delay for the embedding (def=1)
3. emb: embedding dimension (def=2)
4. c: for many points the errors should be computed (def=all)
5. eps0: neighborhood size to start with (def=0.001, relates to $[0, 1]$ rescaled data)
6. epsf: factor for increasing neighborhood seize (def=1.2)
7. minn: only for `lfo-tst` and `lzo-tst`: minimum number of neighbors for fit (def=30) (must be larger than number of parameters to estimate)
8. eps1: only for `lfo-ar` and `lzo-gm`: neighborhood size to end with (def=1., relates to $[0, 1]$ rescaled data)
9. step: steps to be forecasted (def=1)
10. caus: width of causality window
11. only for `lzo-tst`: temporal distance between reference points (def=1)

Note, for obtaining a nonsingular $X'X$ matrix for estimating the first order model `mnn` must be larger than the number of parameters `n * emb` of the model.

Output:

- if `minn` is specified as positive integer:

relerr is either a n vector forst first order models or an $step \times n$ matrix of relative errors for zeroth order models

inderr is a $K \times n$ matrix of individual errors, where $K = c - h$, $c = N - step$, $h = (emb - 1) * delay$.

- if `minn` is missing (not specified): this is a $L \times n + 4$ matrix with the following columns:
  1. neighborhood size of data
  2. relative forecast error (error / variance of data)
  3. average number of neighbors found per point
  4. variance of the fraction of points for which neighbors were found
  5. n columns with the fraction of points for which neighbors were found

Restrictions:

Relationships: tslocfor(), tstrans(), arima(), armcov(), tsmeas()

Examples: All examples refer to the same data $N = 20, n = 1$:

```
xd1 = [ 0.9501  0.2311  0.6068  0.4860  0.8913
        0.7621  0.4565  0.0185  0.8214  0.4447
        0.6154  0.7919  0.9218  0.7382  0.1763
        0.4057  0.9355  0.9169  0.4103  0.8936 ];
```

1. First order estimation and minn is specified as positive integer:

```
optn = [ 2 ,   /* -V: ipri */
         1 ,   /* -d: dely */
         2 ,   /* -m: emb  */
         . ,   /* -n: clen */
         . ,   /* -r: eps0 */
       1.2 ,   /* -f: epsf */
         3 ,   /* -k: minn */
         . ,   /* -R: eps1 */
         2 ,   /* -s: step */
         4 ];  /* -C: caus */


print "lfotst: with specified minn: order=1";
< relerr,inderr > = tsloctst(xd1,1,optn);


   Relative Forecast Errors for 1 Component(s)  1.41383548

        Individual Forecast Errors for 1 Component(s)
        *********************************************

                        2  0.16895331
                        3 -0.09195844
                        4  0.12657959
```

```
 5  0.00638964
 6  0.52549582
 7 -0.10316451
 8  0.61940683
 9  0.23495255
10 -0.31616744
11 -0.18841008
12 -0.38033374
13  0.33142825
14 -0.33758721
15 -0.41656854
16 -0.29172437
17 -0.05155954
18 -1.01550691
```

2. Zeroth order estimation and minn is specified as positive integer:

```
optn = [ 2 ,   /* -V: ipri */
         1 ,   /* -d: dely */
         2 ,   /* -m: emb  */
         . ,   /* -n: clen */
         . ,   /* -r: eps0 */
       1.2 ,   /* -f: epsf */
         3 ,   /* -k: minn */
         . ,   /* -R: eps1 */
         2 ,   /* -s: step */
         4 ];  /* -C: caus */


print "lzotst: with specified minn: order=0";
< relerr,inderr > = tsloctst(xd1,0,optn);


         Relative Forecast Errors for 1 Component(s)
         *******************************************

                        1  0.78949995
                        2  1.09827969

        Individual Forecast Errors for 1 Component(s)
        *********************************************

                        2  0.36360000
                        3 -0.07326667
                        4 -0.00348000
                        5  0.10515000
```

```
 6  0.47336667
 7 -0.21233333
 8  0.33042500
 9  0.12410000
10 -0.24137500
11 -0.15216000
12 -0.32635000
13  0.27990000
14  0.19277500
15 -0.27428000
16 -0.18798333
17  0.34165000
18 -0.69343333
[
```

3. First order estimation and minn is missing (not specified):

```
optn = [ 2 ,   /* -V: ipri */
         1 ,   /* -d: dely */
         2 ,   /* -m: emb  */
         . ,   /* -n: clen */
         . ,   /* -r: eps0 */
       1.2 ,   /* -f: epsf */
         . ,   /* -k: minn */
         . ,   /* -R: eps1 */
         2 ,   /* -s: step */
         4 ];  /* -C: caus */


print "lfoar: with default eps1: order=1";
< relerr,inderr > = tsloctst(xd1,1,optn);


      Neighborhood Error Information for 1 Component(s)
      *************************************************

      NbhSize  RelFoError  PointFract AvNumbNeigb  RelFoError
   0.45855544  0.63669543  0.29411765  6.40000000  0.63669543
   0.45855544  0.63669543  0.29411765  6.40000000  0.63669543
   0.55026652  1.22777656  0.64705882  7.00000000  1.22777656
   0.55026652  1.22777656  0.64705882  7.00000000  1.22777656
   0.66031983  1.19856695  0.76470588  7.69230769  1.19856695
   0.66031983  1.19856695  0.76470588  7.69230769  1.19856695
   0.79238379  1.01807948  0.88235294  8.66666667  1.01807948
   0.79238379  1.01807948  0.88235294  8.66666667  1.01807948
   0.95086055  1.02711244  1.00000000  8.94117647  1.02711244
   0.95086055  1.02711244  1.00000000  8.94117647  1.02711244
```

4. Zeroth order estimation and minn is missing (not specified):

```
optn = [ 2 ,   /* -V: ipri */
         1 ,   /* -d: dely */
         2 ,   /* -m: emb  */
         . ,   /* -n: clen */
         . ,   /* -r: eps0 */
       1.2 ,   /* -f: epsf */
         . ,   /* -k: minn */
         . ,   /* -R: eps1 */
         2 ,   /* -s: step */
         4 ];  /* -C: caus */


print "lzogm: with default eps1: order=0";
< relerr,inderr > = tsloctst(xd1,0,optn);


      Neighborhood Error Information for 1 Component(s)
      *************************************************

      NbhSize  RelFoError  PointFract AvNumbNeigb  RelFoError
   0.45855544  0.92977982  0.29411765  6.40000000  0.92977982
   0.45855544  0.92977982  0.29411765  6.40000000  0.92977982
   0.55026652  0.93673877  0.64705882  7.00000000  0.93673877
   0.55026652  0.93673877  0.64705882  7.00000000  0.93673877
   0.66031983  0.98582010  0.76470588  7.69230769  0.98582010
   0.66031983  0.98582010  0.76470588  7.69230769  0.98582010
   0.79238379  0.98200687  0.88235294  8.66666667  0.98200687
   0.79238379  0.98200687  0.88235294  8.66666667  0.98200687
   0.95086055  0.98360353  1.00000000  8.94117647  0.98360353
   0.95086055  0.98360353  1.00000000  8.94117647  0.98360353
```

5.28 Function `tsmeas`

acv = tsmeas(xdat,"alcdis",nbins<,optn>)

acv = tsmeas(xdat,"alcpro",nbins<,optn>)

<mse,nmse,nrmse,cc> = tsmeas(xdat,"arfit",embed<,optn>)

<mse,nmse,nrmse,cc> = tsmeas(xdat,"arprd",embed<,optn>)

< bic,cumbic > = tsmeas(xdat,"bicorr",delay<,optn>)

cordim = tsmeas(xdat,"cordim",delay,embed,nrads<,optn>)

< cdim,csum,cent > = tsmeas(xdat,"cordi2"<,optn>)

corent = tsmeas(xdat,"corent",delay,embed,radius<,optn>)

corrad = tsmeas(xdat,"corrad",delay,embed,corsum<,optn>)

corsum = tsmeas(xdat,"corsum",delay,embed,radius<,optn>)

detfl = tsmeas(xdat,"detfl"<,optn>)

<ebnd,psv,freq> = tsmeas(xdat,"eband",band<,optn>)

<ym,ymn,tima,tmimi,dmima> = tsmeas(xdat,"exfea",filtord,nsamp<,optn>)

< xmin,xmax > = tsmeas(xdat,"extrem"<,optn>)

falsnn = tsmeas(xdat,"falsnn",delay,embed<,optn>)

falsnn = tsmeas(xdat,"falsn2"<,optn>)

<fsle,ulen,npnt> = tsmeas(xdat,"fslexp"<,optn>)

<mob,comp> = tsmeas(xdat,"hjoer"<,optn>)

hurst = tsmeas(xdat,"hurst"<,optn>)

<cor,cum,dec,zer> = tsmeas(xdat,"kenda",delay<,optn>)

< mse,nmse,nrmse,cc > = tsmeas(xdat,"larfit",delay,embed,nnei<,optn>)

< mse,nmse,nrmse,cc > = tsmeas(xdat,"larprd",delay,embed,nnei<,optn>)

lyap = tsmeas(xdat,"lyapk"<,optn>)

lyap = tsmeas(xdat,"lyapr"<,optn>)

<medf,psv,freq> = tsmeas(xdat,"medfr"<,optn>)

< mut,cummut,minmut > = tsmeas(xdat,"mutdis",delay,nbins<,optn>)

< mut,cummut,minmut > = tsmeas(xdat,"mutpro",delay,nbins<,optn>)

rmut = tsmeas(xdat,"mutual"<,optn>)

<cor,cum,dec,zer> = tsmeas(xdat,"pears",delay<,optn>)

renyi = tsmeas(xdat,"renent"<,optn>)

<cor,cum,dec,zer> = tsmeas(xdat,"spear",delay<,optn>)

corr = tsmeas(xdat,"xcorr1"<,optn>)

corr = tsmeas(xdat,"xcorr2"<,optn>)

Purpose: The `tsmeas` function implements a number of linear and nonlinear measures for time series data.

Input: **xdat** is either a N column vector or a $N \times n$ matrix of N time series observations with n time variables.

sopt specifies the kind of measurement which is computed:

"alcdis" computes the algorithmic complexity on the time series based on partitions of equal distance of the time series; an integer or a vector of integers `nbins` with the number of bins must be specified as additional argument.

"alcpro" computes the algorithmic complexity on the time series based on partitions of the time series with same probability;an integer or a vector of integers `nbins` with the number of bins must be specified as additional argument.

"arfit" computes statistical errors of fit at lead times; an integer or a vector of integers `embed` with the embeddings must be specified as additional argument;

"arprd" computes statistical errors of prediction at lead times; an integer or a vector of integers `embed` with the embeddings must be specified as additional argument.

"bicorr" computes the bicorrelation on the time series for given delays and the cumulative bicorrelation for each given lag; an integer or a vector of integers `delay` with the delays must be specified as additional argument.

"cordim" computes the correlation dimension for a given time series, for a range of delays, a range of embedding dimensions, and for a range of upper/lower ratio of scaling window; additional arguments are:

1. integer scalar or a vector of integers `delay` with the delays;
2. integer scalar or a vector of integers `embed` with the embeddings;
3. integer scalar or a vector of integers `nrads` with the number of radii.

"cordi2" computes the correlation sum, entropy, and dimension for a given time series, for a number of components and maximal embedding dimension, for a given Theiler window, a minimal and maximal length scale, and a specified number of epsilon values (similar to `d2` function in TISEAN).

"corent" computes the approximate entropy that measures the so-called "pattern similarity" in a given time series; additional arguments are:

1. integer scalar or a vector of integers `delay` with the delays;
2. integer scalar or a vector of integers `embed` with the embeddings;
3. real scalar or a vector of reals `radius` with the radii.

"corrad" computes the radii regarding a set of given correlation sums on a given time series and state space reconstructions with delays and embedding dimensions; additional arguments are:

1. integer scalar or a vector of integers `delay` with the delays;
2. integer scalar or a vector of integers `embed` with the embeddings;
3. real scalar or a vector of reals `corsum` with given correlation sum.

"corsum" computes the correlation on a given time series for a number of radius and state space reconstructions with delays and embedding dimensions; additional arguments are:

1. integer scalar or a vector of integers `delay` with the delays;
2. integer scalar or a vector of integers `embed` with the embeddings;
3. real scalar or a vector of reals `radius` with the radii.

"detfl" computes the detrended fluctuation analysis.

"eband" computes the energy in the frequency band; either a 2-vector or a $b \times 2$ matrix of lower (first column) and upper (second column) band frequency `band` must be specified as an additional argument.

"exfea" finds the local extreme values for a given time series computed for a window; the time series can be first filtered by a forward-backward moving average filter; additional arguments are:

1. integer scalar or a vector of integers `filtord` with the filter order;
2. integer scalar or a vector of integers `nsamp` with the number of samples to form sliding window;

"extrem" determines maxima and/or minima of a component of a time series (corresponds to Poincaré section at the zeros of the derivative); quadratic interpolation is applied; the minimal time between two extrema can be specified (similar to `extrema` function in TISEAN).

"falsnn" computes the percentage of false nearest neighbors for a range of delays; additional arguments are:

1. integer scalar or a vector of integers `delay` with the delays;

2. integer scalar or a vector of integers `embed` with the embeddings;

"falsn2" computes the fraction of false nearest neighbors of all data points and iterates neighbors one step in future; the number of components and the maximum embedding dimension, the delay, a ratio factor, and the Theiler window can be specified (similar to `false-nearest` function in TISEAN).

"fslexp" estimates the finite size Lyapunov exponent for a specified embedding dimension, a delay, a window around the reference point which should be omitted, and the minimum length scale for the neighborhood search (similar to `fsle` function in TISEAN).

"hjoer" computes the Hjoerth parameters mobility and complexity.

"hurst" computes the Hurst exponent.

"kenda" computes Kendall autocorrelation, the cumulative Kendall autocorrelation, the decorrelation time, and the time the Kendall autocorrelation falls to zero; an integer or a vector of integers `delay` with the delays must be specified as additional argument.

"larfit" makes in-sample direct predictions with a local model on a given time series; the state space reconstruction is done with the method of delays and the parameters are the embedding dimension and the delay time; additional arguments are:

1. integer scalar or a vector of integers `delay` with the delays;
2. integer scalar or a vector of integers `embed` with the embeddings;
3. integer scalar or a vector of integers `nnei` with the number of neighbors;
4. real scalar or a vector of reals `ptim` with the prediction times.

"larprd" makes direct predictions with a local model on a test set given by the last part of the given time series (as the 'fraction' porportion of the time series); a number of input parameters determine the local model that are allowed to vary (components of the input vectors); the state space reconstruction is done with the method of delays and the parameters are the embedding dimension and the delay time; additional arguments are:

1. integer scalar or a vector of integers `delay` with the delays;
2. integer scalar or a vector of integers `embed` with the embeddings;
3. integer scalar or a vector of integers `nnei` with the number of neighbors;
4. real scalar or a vector of reals `ptim` with the prediction times.

"lyapk" estimates the largest Lyapunov exponent of a given time series using the algorithm by Kantz (1994)

"lyapr" estimates the largest Lyapunov exponent of a given time series using the algorithm by Rosenstein et al. (1993)

"medfr" computes the median frequency in a frequency range.

"mutdis" computes the minimum of mutual information computed for given delays; additional arguments are:

1. integer scalar or a vector of integers `delay` with the delays;
2. integer scalar or a vector of integers `nbins` with the number of bins.

"mutpro" computes the minimum of mutual information computed for given delays; additional arguments are:

1. integer scalar or a vector of integers `delay` with the delays;
2. integer scalar or a vector of integers `nbins` with the number of bins.

"mutual" estimates the time delayed mutual information of the data using fixed mesh boxes; the number of boxes and the maximum time delay can be specified (similar to `mutual` function in TISEAN).

"pears" computes Pearson autocorrelation, the cumulative Pearson autocorrelation, the decorrelation time, and the time the Pearson autocorrelation falls to zero; an integer or a vector of integers `delay` with the delays must be specified as additional argument.

"renent" computes Renyi entropy of Qth order using a partition of the phase space (instead of Grassberger-Procaccia); the number of components and the maximum embedding dimension, the order of entropy, the delay, a minimum and maximum length scale, and the number of epsilon values can be specified (similar to `boxcount` function in TISEAN).

"spear" computes Spearman autocorrelation, the cumulative Spearman autocorrelation, the decorrelation time, and the time the Spearman autocorrelation falls to zero; an integer or a vector of integers `delay` with the delays must be specified as additional argument.

"xcorr1" computes the autocorrelation (also covariance or unscaled scalar product) of a data set (similar to `cor` function in TISEAN and the `xcorr` function in Matlab).

"xcorr2" computes the cross correlations between two time series (similar to `xcor` function in TISEAN and the `xcorr` function in Matlab).

optn

1. ipri: amount of printed output (default=0: no printed output)
2. `"arfit","arprd"`: int number of predictions (default: tmax=1)
   `"falsnn","cordim","corsum","corrad","corent"`: int size of Theiler window (default: theil=0)
   `"eband","medfr"`: real left cutoff (default: lftcut=0)
   `"exfea"`: int seed for random number (default: time of day);
   `"larfit","larprd"`: specifies one of two algorithms:

iter=0: linear direct fit / prediction;
iter=1: linear iterative fit / prediction;
(default: iter=0)
"lyapk","lyapr": delay (def=1)

3. "arprd": real fraction for test set (no default for fract)
   "falsnn": real escape factor (default: esc=10.)
   "cordim": int number of radii, resolution (no default for resol)
   "larfit","larprd": truncation parameter Q (no default for iq)
   "medfr": real right cutoff (default: rgtcut=.5)
   "lyapk": maximum embedding dimension (default=2)
   "lyapr": embedding (default=2)
4. "falsnn": int seed for random number (default: time of day);
   "larprd": real fraction for test set (no default for fract)
   "lyapk": number steps (default=50)
   "lyapr": number steps (default=10)
5. "lyapk": epsilon count (default=5)
   "lyapr": theiler window (default=0)
6. "eps0","lyapr": starting epsilon (default=1.e-3)
7. "lyapk": upper range for epsilon (default=1.e-2)
8. "lyapk": minimum embedding dimension (default=2)
9. "lyapk": theiler window (default=0)

Output:

Restrictions:

Relationships: arima(), armcov(), tslocfor(), tsloctst(), tstrans()

Examples: The following data sets are used in the examples below:

```
 xd = [ 0.8147 0.9058 0.1270 0.9134 0.6324
        0.0975 0.2785 0.5469 0.9575 0.9649
        0.1576 0.9706 0.9572 0.4854 0.8003
        0.1419 0.4218 0.9157 0.7922 0.9595 ];


xd1 = [ 0.4387 0.3816 0.7655 0.7952 0.1869 0.4898 0.4456 0.6463 0.7094 0.7547
        0.2760 0.6797 0.6551 0.1626 0.1190 0.4984 0.9597 0.3404 0.5853 0.2238
        0.7513 0.2551 0.5060 0.6991 0.8909 0.9593 0.5472 0.1386 0.1493 0.2575
        0.8407 0.2543 0.8143 0.2435 0.9293 0.3500 0.1966 0.2511 0.6160 0.4733
        0.3517 0.8308 0.5853 0.5497 0.9172 0.2858 0.7572 0.7537 0.3804 0.5678
        0.0759 0.0540 0.5308 0.7792 0.9340 0.1299 0.5688 0.4694 0.0119 0.3371
        0.1622 0.7943 0.3112 0.5285 0.1656 0.6020 0.2630 0.6541 0.6892 0.7482
        0.4505 0.0838 0.2290 0.9133 0.1524 0.8258 0.5383 0.9961 0.0782 0.4427
        0.1067 0.9619 0.0046 0.7749 0.8173 0.8687 0.0844 0.3998 0.2599 0.8001
        0.4314 0.9106 0.1818 0.2638 0.1455 0.1361 0.8693 0.5797 0.5499 0.1450 ];
```

```
xd2 = [ 0.8147 0.9058 0.1270 0.9134 0.6324
        0.0975 0.2785 0.5469 0.9575 0.9649
        0.1576 0.9706 0.9572 0.4854 0.8003
        0.1419 0.4218 0.9157 0.7922 0.9595
        0.6557 0.0357 0.8491 0.9340 0.6787
        0.7577 0.7431 0.3922 0.6555 0.1712
        0.7060 0.0318 0.2769 0.0462 0.0971
        0.8235 0.6948 0.3171 0.9502 0.0344 ];


xd3 = [ 0.9501  0.2311  0.6068  0.4860  0.8913
        0.7621  0.4565  0.0185  0.8214  0.4447
        0.6154  0.7919  0.9218  0.7382  0.1763
        0.4057  0.9355  0.9169  0.4103  0.8936 ];


xd9 = [ 0.7386 0.5860 0.2467 0.6664 0.0835 0.6260 0.6609 0.7298 0.8908 0.9823
        0.7690 0.5814 0.9283 0.5801 0.0170 0.1209 0.8627 0.4843 0.8449 0.2094 ];
```

"alcdis" algorithmic complexity:

```
optn = [ 2 ]; bv = 2;
alcdis = tsmeas(xd1,"alcdis",bv,optn);
print "ALCDIS=",alcdis;


 ALCDIS= 1.9267
```

"alcpro" algorithmic complexity:

```
optn = [ 2 ]; bv = 2;
alcpro = tsmeas(xd1,"alcpro",bv,optn);
print "ALCPRO=",alcpro;


 ALCPRO= 1.9267
```

"arfit" statistical errors of fit at lead times:

```
tmax = 1;
optn = [ 2 tmax ];
m = 5;
< mse,nmse,nrmse,cc > = tsmeas(xd,"arfit",m,optn);
print "MSE=",mse;
print "NMSE=",nmse;
print "NRMSE=",nrmse;
```

```
MSE= 0.09812
NMSE= 0.8439
NRMSE= 0.9187
CC= 0.3788
```

"arprd" statistical errors of prediction at lead times:

```
tmax = 1; fract = .2;
optn = [ 2 tmax fract ];
m = 5;
< mse,nmse,nrmse,cc > = tsmeas(xd,"arprd",m,optn);
print "MSE=",mse;
print "NMSE=",nmse;
print "NRMSE=",nrmse;
```

```
MSE= 0.08674
NMSE= 1.4550
NRMSE= 1.2062
CC= 0.2882
```

"bicorr" bicorrelation on the time series:

```
optn = [ 2 ]; tau = 5;
< bic,cumbic > = tsmeas(xd,"bicorr",tau,optn);
print "BIC=",bic;
print "Cumbic=",cumbic;
```

```
BIC=-0.2491
Cumbic= 1.2789
```

"cordim" correlation dimension:

```
theil = 0; resol = 10;
optn = [ 2 theil resol ];
tau = 1; mv = 3; sv = 2;
cordim = tsmeas(xd1,"cordim",tau,mv,sv,optn);
print "CORDIM=",cordim;
```

```
CORDIM= 2.3448
```

"cordi2" correlation sum, entropy, and dimension:

```
   dely = 1; theil = 0; emb = 10;
   emin = emax = .; howo = 10; maxf = 0;
   optn = [ 2 dely theil emb emin emax howo maxf ];
   < cdim,csum,cent > = tsmeas(xd3,"cordi2",optn);
   print "CORDIM2=",cdim;
   print "CORSUM2=",csum;
   print "CORENT2=",cent;
```

CORDIM2=

	1	2	3	4	5
1	0.48174	0.97353	1.2305	0.72911	0.90309
2	0.87961	2.2446	2.0969	.	.
3	1.3848	2.4049	.	.	.
4	1.6928	2.6252	.	.	.
5	2.2211	.	.	.	.
6	2.8866	.	.	.	.
7	3.4149	.	.	.	.
8	4.3180	.	.	.	.
9	5.2211	.	.	.	.
10	.	.	.	.	.

	6	7	8	9
1	0.90309	.	.	.
2	.	.	.	.
3	.	.	.	.
4	.	.	.	.
5	.	.	.	.
6	.	.	.	.
7	.	.	.	.
8	.	.	.	.
9	.	.	.	.
10	.	.	.	.

CORSUM2=

	1	2	3	4	5
1	1.00000	0.69091	0.32727	0.12727	0.07273
2	1.00000	0.50909	0.09091	0.01818	0.00000
3	1.00000	0.34545	0.05455	0.00000	0.00000
4	1.00000	0.27273	0.03636	0.00000	0.00000
5	1.00000	0.18182	0.00000	0.00000	0.00000
6	1.00000	0.10909	0.00000	0.00000	0.00000

7	1.00000	0.07273	0.00000	0.00000	0.00000
8	1.00000	0.03636	0.00000	0.00000	0.00000
9	1.00000	0.01818	0.00000	0.00000	0.00000
10	1.00000	0.00000	0.00000	0.00000	0.00000

	6	7	8	9	10
1	0.03636	0.01818	0.00000	0.00000	0.00000
2	0.00000	0.00000	0.00000	0.00000	0.00000
3	0.00000	0.00000	0.00000	0.00000	0.00000
4	0.00000	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000	0.00000
9	0.00000	0.00000	0.00000	0.00000	0.00000
10	0.00000	0.00000	0.00000	0.00000	0.00000

CORENT2=

	1	2	3	4	5
1	0.00000	0.36975	1.1170	2.0614	2.6210
2	0.00000	0.30538	1.2809	1.9459	.
3	0.00000	0.38777	0.51083	.	.
4	0.00000	0.23639	0.40547	.	.
5	0.00000	0.40547	.	.	.
6	0.00000	0.51083	.	.	.
7	0.00000	0.40547	.	.	.
8	0.00000	0.69315	.	.	.
9	0.00000	0.69315	.	.	.
10	0.00000	.	.	.	.

	6	7	8	9	10
1	3.3142	4.0073	.	.	.
2	.	.	.	.	.
3	.	.	.	.	.
4	.	.	.	.	.
5	.	.	.	.	.
6	.	.	.	.	.
7	.	.	.	.	.
8	.	.	.	.	.
9	.	.	.	.	.
10	.	.	.	.	.

"corent" approximate entropy:

```
theil = 0;
optn = [ 2 theil ]; tau = 5; mv = 3; rv = [ .2 .4 ];
corent = tsmeas(xd1,"corent",tau,mv,rv,optn);
print "CORENT=",corent;
```

```
CORENT=
   |         1
--------------
 1 |   0.45626
 2 |   0.04279
```

"corrad" radii for given correlation sum:

```
theil = 0;
optn = [ 2 theil ]; tau = 5; mv = 3; rv = [ .2 .4 ];
corrad = tsmeas(xd9,"corrad",tau,mv,rv,optn);
print "CORRAD=",corrad;
```

```
CORRAD=
   |         1
--------------
 1 |   0.50383
 2 |   0.58902
```

"corsum" correlation sum:

```
/* Minbo's data: corsumT =  0 0.2000 */
theil = 0;
optn = [ 2 theil ]; tau = 5; mv = 3; rv = [ .2 .4 ];
corsum = tsmeas(xd9,"corsum",tau,mv,rv,optn);
print "CORSUM=",corsum;
```

```
CORSUM=
   |         1
--------------
 1 |   0.00000
 2 |   0.20000
```

"detfl" detrended fluctuation analysis:

```
optn = [ 2 ];
detfl = tsmeas(xd,"detfl",optn);
print "DETFL=",detfl;
```

```
DETFL= 0.2635
```

"eband" energy in the frequency band:

"exfea" local extreme values:

"extrem" maxima and/or minima:

```
dely = 1; job = 0; delt = 0.;
optn = [ 2 job delt ];
< xmin,xmax > = tsmeas(xd3,"extrem",optn);
print "Xmin=",xmin;
print "Xmax=",xmax;
```

```
Xmin=
   |         1         2
-----------------------
 1 |   0.21764    1.1568
 2 |    .         .
 3 |   0.46677    1.5728
 4 |    .         .
 5 |    .         .
 6 |    .         .
 7 |   0.00509    4.1234
 8 |    .         .
 9 |   0.43501    2.3352
10 |    .         .
11 |    .         .
12 |    .         .
13 |    .         .
14 |   0.15884    5.0219
15 |    .         .
16 |    .         .
17 |    .         .
18 |   0.41023    3.8017
```

```
Xmax=
   |         1         2
------------------------
 1 |    .         .
 2 |   0.62316    2.2567
 3 |    .         .
 4 |   0.90913    2.0016
 5 |    .         .
 6 |    .         .
 7 |    .         .
 8 |   0.84065    3.9224
 9 |    .         .
10 |    .         .
11 |    .         .
12 |   0.92295    3.7337
13 |    .         .
14 |    .         .
15 |    .         .
16 |   0.99507    4.5517
17 |    .         .
18 |    .         .
```

"falsnn" percentage of false nearest neighbors:

```
   theil = 0; esc = 3; seed = 12345;
   optn = [ 2 theil esc seed ];
   tau = 5; mv = 3;
   falsnn = tsmeas(xd2,"falsnn",tau,mv,optn);
   print "FALSNN=",falsnn;


FALSNN= 0.2800


   theil = 0; esc = 1; seed = 12345;
   optn = [ 2 theil esc seed ];
   tau = 1; mv = [ 1 2 ];
   falsnn = tsmeas(xd3,"falsnn",tau,mv,optn);
   print "FALSNN=",falsnn;


FALSNN=
   |         1         2
------------------------
 1 |   1.00000   1.00000
```

"falsn2" fraction of false nearest neighbors:

```
dely = 1; theil = 0;
minemb = 1; maxemb = 2; esc = 1.;
optn = [ 2 dely theil minemb maxemb esc ];
falsnn = tsmeas(xd3,"falsn2",optn);
print "FALSNN2=",falsnn;
```

```
FALSNN2=
   |         1         2         3         4
---------------------------------------------
 1 |   1.00000   0.94444   0.03116   0.04604
 2 |    2.0000   0.76471   0.13002   0.14472
```

"fslexp" finite size Lyapunov exponent:

```
dely = 1; mind = 0; emb = 2; eps0 = .;
optn = [ 2 dely mind emb eps0 ];
< fsle,ulen,pnts > = tsmeas(xd3,"fslexp",optn);
print "FSLE=",fsle;
print "ULEN=",ulen;
print "PNTS=",pnts;
```

```
FSLE=
   |         1         2
------------------------
 1 |   0.21039   0.56755
ULEN=
   |         1         2
------------------------
 1 |   0.09966   0.14095
PNTS=
   |         1         2
------------------------
 1 |   1.00000    2.0000
```

"hjoer" Hjoerth parameters mobility and complexity:

```
optn = [ 2 ];
<  mob,comp > = tsmeas(xd,"hjoer",optn);
print "MOB=",mob;
print "COMP=",comp;
```

```
MOB= 0.6942
COMP= 1.4910
```

"hurst" Hurst exponent:

```
optn = [ 2 ];
hurst = tsmeas(xd,"hurst",optn);
print "HURST=",hurst;
```

```
HURST= 0.3767
```

"kenda" Kendall autocorrelation:

```
optn = [ 2 ]; tau = 5;
 < cor2,cum2,dec2,zer2 > = tsmeas(xd,"kenda",tau,optn);
print "Cor2=",cor2;
print "Cum2=",cum2;
print "Dec2=",dec2;
print "Zer2=",zer2;
```

```
Cor2=-0.009524
Cum2= 0.4275
Dec2= 1.0000
Zer2= 2.0000
```

"larfit" in-sample direct predictions with a local model:

```
print "LARFIT is using KdTree: iter=0: NONiterated version";
iter = 0; iq = 5;
optn = [ 2 iter iq ];
tau = 5; mv = 3; nn = [ 8 12 ]; tv = 18;
< mse,nmse,nrmse,cc > = tsmeas(xd1,"larfit",tau,mv,nn,tv,optn);
print "MSE=",mse;
print "NMSE=",nmse;
print "NRMSE=",nrmse;
print "CC=",cc;
```

```
MSE=
   |         1         2
-----------------------
 1 |   0.15715   0.11849
NMSE=
   |         1         2
```

```
------------------------
 1 |    1.8162    1.3694
NRMSE=
   |         1         2
------------------------
 1 |    1.3477    1.1702
CC=
   |         1         2
------------------------
 1 |  -0.10454  -0.12461
```

```
print "LARFIT is using KdTree: iter=1: user iterated version";
iter = 1; iq = 5;
optn = [ 2 iter iq ];
tau = 5; mv = 3; nn = [ 8 12 ]; tv = 18;
< mse,nmse,nrmse,cc > = tsmeas(xd1,"larfit",tau,mv,nn,tv,optn);
print "MSE=",mse;
print "NMSE=",nmse;
print "NRMSE=",nrmse;
print "CC=",cc;
```

```
MSE=
   |            1            2
-----------------------------
 1 |   4.491e+013   9.859e-002
NMSE=
   |            1            2
-----------------------------
 1 |   5.190e+014   1.139e+000
NRMSE=
   |           1           2
---------------------------
 1 |    22781435    1.067419
CC=
   |         1         2
------------------------
 1 |   0.03821  -0.00940
```

"larprd" direct predictions with a local model:

```
print "LARPRD is using KdTree: iter=0: NONiterated version";
iter = 0; iq = 5 ; fract = .2;
optn = [ 2 iter iq fract ];
tau = 5; mv = 3; nn = [ 8 12 ]; tv = 18;
```

```
   < mse,nmse,nrmse,cc > = tsmeas(xd1,"larprd",tau,mv,nn,tv,optn);
   print "MSE=",mse;
   print "NMSE=",nmse;
   print "NRMSE=",nrmse;
   print "CC=",cc;


MSE=
   |         1         2
------------------------
 1 |   0.05093   0.03846
NMSE=
   |         1         2
------------------------
 1 |   0.86367   0.65218
NRMSE=
   |         1         2
------------------------
 1 |   0.92934   0.80758
CC=
   |         1         2
------------------------
 1 |   0.39351   0.43221


   print "LARPRD is using KdTree: iter=1: user iterated version";
   iter = 1; iq = 5 ; fract = .2;
   optn = [ 2 iter iq fract ];
   tau = 5; mv = 3; nn = [ 8 12 ]; tv = 18;
 /* options debug="tsmeas*=3 locar*=3"; */
   < mse,nmse,nrmse,cc > = tsmeas(xd1,"larprd",tau,mv,nn,tv,optn);
   print "MSE=",mse;
   print "NMSE=",nmse;
   print "NRMSE=",nrmse;
   print "CC=",cc;


MSE=
   |         1         2
------------------------
 1 |   0.13538   0.04587
NMSE=
   |         1         2
------------------------
 1 |    2.2959   0.77791
NRMSE=
   |         1         2
```

```
------------------------
 1 |    1.5152   0.88199
CC=
   |         1         2
------------------------
 1 |  -0.87078  -0.09230
```

"lyapk" largest Lyapunov exponent, algorithm by Kantz (1994):

```
/* [1]: ipri(0), [2]: dely(1), [3]: maemb(2),
   [4]: maxit(50), [5]: cnteps(5), [6]: eps0(1.e-3),
   [7]: eps1(1.e-2), [8]: miemb(2), [9]: nref(),
   [10]: mind (0) */
dely = 1; maemb = miemb = 2;
maxit = 5; cnteps = 5;
eps0 = 1.e-3; eps1 = 1.07342;
optn = [ 2 dely maemb maxit cnteps eps0 eps1 miemb ];
lyap = tsmeas(xd3,"lyapk",optn);
print "LYAPK=",lyap;
```

Dim	Ceps	Embed	Iter	Count	LyapExp	Epsilon
1	0	1	0	0	0.00000000	0.0010000
1	0	1	1	0	0.00000000	0.0010000
1	0	1	2	0	0.00000000	0.0010000
1	0	1	3	0	0.00000000	0.0010000
1	0	1	4	0	0.00000000	0.0010000
1	0	1	5	0	0.00000000	0.0010000
1	1	1	0	0	0.00000000	0.0057239
1	1	1	1	0	0.00000000	0.0057239
1	1	1	2	0	0.00000000	0.0057239
1	1	1	3	0	0.00000000	0.0057239
1	1	1	4	0	0.00000000	0.0057239
1	1	1	5	0	0.00000000	0.0057239
1	2	1	0	0	0.00000000	0.0327631
1	2	1	1	0	0.00000000	0.0327631
1	2	1	2	0	0.00000000	0.0327631
1	2	1	3	0	0.00000000	0.0327631
1	2	1	4	0	0.00000000	0.0327631
1	2	1	5	0	0.00000000	0.0327631
1	3	1	0	7	-2.34352158	0.1875328
1	3	1	1	7	-0.90348339	0.1875328
1	3	1	2	7	-0.59614605	0.1875328
1	3	1	3	7	-1.17141611	0.1875328
1	3	1	4	7	-0.62791995	0.1875328
1	3	1	5	7	-0.45494269	0.1875328

```
1     4     1     0      14 -0.56893325  1.0734200
1     4     1     1      14 -0.56636360  1.0734200
1     4     1     2      14 -0.55418162  1.0734200
1     4     1     3      14 -0.50597255  1.0734200
1     4     1     4      14 -0.49699453  1.0734200
1     4     1     5      14 -0.48799001  1.0734200
```

"lyapr" largest Lyapunov exponent, algorithm by Rosenstein et al. (1993):

```
/* [1]: ipri(0), [2]: dely(1), [3]: emb(2),
   [4]: step(10), [5]: mind(0), [6]: eps0(1.e-3) */
dely = 1; emb = 2; step = 10; mind = 0; eps0 = 1.e-3;
optn = [ 2 dely emb step mind eps0 ];
lyap = tsmeas(xd3,"lyapr",optn);
print "LYAPR=",lyap;
```

```
Dim   Step   Count     LyapExp    Epsilon
  1      1       9 -1.48876060  0.5394078
  1      2       9 -0.86934920  0.5394078
  1      3       9 -0.69852292  0.5394078
  1      4       9 -0.98014137  0.5394078
  1      5       9 -0.85177456  0.5394078
  1      6       9 -0.42783544  0.5394078
  1      7       9 -0.42220083  0.5394078
  1      8       9 -0.61480084  0.5394078
  1      9       9 -0.70789218  0.5394078
  1     10       9 -0.70259573  0.5394078
  1     11       9 -1.29667240  0.5394078
```

"medfr" median frequency in a frequency range:

"mutdis" minimum of mutual information:

```
optn = [ 2 ]; tau = 5; bv = 5;
< mut,cummut,minmut > = tsmeas(xd,"mutdis",tau,bv,optn);
print "Mut=",mut;
print "Cummut=",cummut;
print "Minmut=",minmut;
```

```
Mut= 0.5287
Cummut= 3.8408
```

```
Minmut= 2.0000
```

"mutpro" minimum of mutual information:

```
optn = [ 2 ]; tau = 5; bv = 5;
< mut,cummut,minmut > = tsmeas(xd,"mutpro",tau,bv,optn);
print "Mut=",mut;
print "Cummut=",cummut;
print "Minmut=",minmut;
```

```
Mut= 0.5471
Cummut= 2.9272
Minmut= 1.0000
```

"mutual" time delayed mutual information:

```
dely = 2; nbox = 5;
 optn = [ 2 dely nbox ];
 rmut = tsmeas(xd3,"mutual",optn);
 print "Rmutual=",rmut;
```

```
Rmutual=
   |         1
--------------
 1 |    1.4150
 2 |   0.39923
 3 |   0.37183
```

"pears" Pearson autocorrelation:

```
optn = [ 2 ];
tau = 5;
< cor1,cum1,dec1,zer1 > = tsmeas(xd,"pears",tau,optn);
print "Cor1=",cor1;
print "Cum1=",cum1;
print "Dec1=",dec1;
print "Zer1=",zer1;
```

```
Cor1= 0.1646
Cum1= 0.7401
Dec1= 1.0000
Zer1= 1.0000
```

"renent" Renyi entropy of Qth order:

```
dely = 1; ecnt = 5; minemb = 1; maxemb = 10;
fact = 2.; emin = emax = .;
optn = [ 2 dely ecnt minemb maxemb fact emin emax ];
renyi = tsmeas(xd3,"renent",optn);
print "Renyi=",renyi;
```

```
   |         1         2         3         4         5
---------------------------------------------------------
 1 |   1.00000   1.00000   0.93160   0.00000   0.00000
 2 |   1.00000   1.00000   0.16566    1.4285    1.4285
 3 |   1.00000   1.00000    0.0295    2.0877    2.0877
 4 |   1.00000   1.00000    0.0052    2.3979    2.3979
 5 |   1.00000   1.00000    0.0009    2.3979    2.3979
.........................................................
45 |   1.00000    9.0000    0.0009    2.3979   0.00000
46 |   1.00000   10.0000   0.93160   0.00000   0.00000
47 |   1.00000   10.0000   0.16566    2.3979   0.00000
48 |   1.00000   10.0000    0.0295    2.3979   0.00000
49 |   1.00000   10.0000    0.0052    2.3979   0.00000
50 |   1.00000   10.0000    0.0009    2.3979   0.00000
```

"spear" Spearman autocorrelation:

```
optn = [ 2 ]; tau = 5;
< cor3,cum3,dec3,zer3 > = tsmeas(xd,"spear",tau,optn);
print "Cor3=",cor3;
print "Cum3=",cum3;
print "Dec3=",dec3;
print "Zer3=",zer3;
```

```
Cor3= 0.03929
Cum3= 0.6488
Dec3= 1.0000
Zer3= 1.0000
```

"xcorr1" autocorrelation:

```
tau = 5; ist = 0;
optn = [ 2 tau ist ];
corr = tsmeas(xd3,"xcorr1",optn);
print "CORRs=",corr;
```

```
CORRs=
   |         1         2         3         4         5         6
---------------------------------------------------------------------
 1 |   1.00000  -0.11253  -0.28765  -0.27560   0.17776  -0.05680
```

"xcorr2" cross correlations:

5.29 Function `tstrans`

ydat = tstrans(xdat,"sopt"<,optn>)

Purpose: The `tstrans` function implements a number of algorithms for generating data transformations, filter and surrogates.

Standardize time series data:

"gaus" Gaussian: takes a time series and changes its marginal cumulative function to Gaussian

"unif" uniform: takes a time series and changes its marginal cumulative function to Uniform in [0,1]

"line" linear: takes a time series and transform it linearly to the interval [0,1] (min is 0 and max is 1).

"norm" normalized: takes a time series 'xV' and transform it linearly to a time series with zero mean and unit standard deviation.

Data filters:

"l121" implements a simple (iterated) 1-2-1 filter

"notc" implements a notch filter

"sago" implements Savitzky-Golay filter

Surrogate Methods:

"perm" creates K surrogates of random permuted data.

"four" Fourier Transform : generates a given number of K Fourier Transform surrogates for the given time series

"aaft" Amplitude Adjusted Fourier Transform: generates K AAFT-surrogate time series and stores them columnwise in a matrix; the AAFT surrogates are supposed to have the same amplitude distribution (marginal cdf) and autocorrelation as the given time series. See: Theiler et al (1992).

"iaaf" Iterated Amplitude Adjusted Fourier: generates K IAAFT-surrogate time series and stores them columnwise in the matrix.; these surrogates are supposed to have the same amplitude distribution (marginal cdf) and autocorrelation as the given time series. See: Schreiber & Schmitz (1996)

"stap" statistically transformed AR process (not done yet): generates K stochastic surrogates for a given time series as statically transformed realisations of a Gaussian (autoregressive) process; the generated surrogate data have the same amplitude distribution (marginal cdf) and autocorrelation as the original time series. See: Kugiumtzis (2002b)

"amrb" Autoregressive Model Residual Bootstrap (not done yet): generates K surrogate data for the given time series using an estimated AR model and bootstrapping the residuals of the model; the order of AR is found by AIC calculated for p=1,...,arm.

Input: xdat is either a N column vector or a $N \times n$ matrix of N time series observations with n time variables

sopt specifies the kind of data transformation which is computed:

1. Transformations and Filters:
   "gaus" Gaussian
   "unif" uniform
   "line" linear [0,1]
   "norm" (mu,stdev) normalized
   "l121" simple (iterated) 1-2-1 filter
   "notc" notch filter
   "sago" Savitzky-Golay filter
2. Surrogate Methods:
   "perm" random permuted surrogates
   "four" Fourier Transform
   "aaft" Amplitude Adjusted Fourier Transform
   "iaaf" Iterated Amplitude Adjusted Fourier
   "stap" STAtistically transformed AutoRegressive process
   "amrb" AMR Bootstrap (not done yet)

optn
1. ipri: for all methods;
2. nsur: for FOUR, AAFT, IAAF, STAP; f: for NOTCH; niter: for L121; degree: for SAGO;
3. seed: for PERM, FOUR, AAFT, IAAF; w: for NOTCH; nder: for SAGO;
4. ideg: for STAP only; s: for NOTCH; b: for SAGO;
5. narm: for STAP; f: for SAGO;

Output:
1. Transformations and Filters: Returned is the transformed data set with the same dimension $N \times n$ as the input data. That means, the return object is a $N \times n$ vector or matrix.
2. Surrogate Methods: Returned are `optn[2]=nsur` horicontally stacked data sets with the same dimension $N \times n$ as the input data. That means, the return object is a $N \times n * nsur$ matrix.

Restrictions:

Relationships: tsmeas(), arima(), armcov()

Examples: The same data set is used for all of the following transformations:

```
xd1 = [ 0.9501  0.2311  0.6068  0.4860  0.8913
        0.7621  0.4565  0.0185  0.8214  0.4447
        0.6154  0.7919  0.9218  0.7382  0.1763
        0.4057  0.9355  0.9169  0.4103  0.8936 ];
```

"gaus" Gaussian

```
optn = [ 2 ];
trans11 = tstrans(xd1,"gaus",optn);
```

"unif" uniform

```
optn = [ 2 ];
trans21 = tstrans(xd1,"unif",optn);
```

"line" linear [0,1]

```
optn = [ 2 ];
trans31 = tstrans(xd1,"line",optn);
```

"norm" (mu,stdev) normalized

```
optn = [ 2 ];
trans41 = tstrans(xd1,"norm",optn);
```

```
res = trans11 -> trans21 -> trans31 -> trans41;
print "Gauss, Uniform, Linear and Normal Transformions",res;
```

```
Gauss, Uniform, Linear and Normal Transformions
   |         1         2         3         4
-------------------------------------------------
 1 |    1.8367   0.96688   1.00000    1.1556
 2 |   -1.1197   0.13141   0.22821   -1.3901
 3 |  -0.18585   0.42628   0.63149  -0.05985
 4 |  -0.31301   0.37714   0.50182  -0.48756
 5 |   0.58627   0.72115   0.93688   0.94745
 6 |   0.18585   0.57372   0.79820   0.49001
 7 |  -0.44546   0.32799   0.47016  -0.59201
 8 |   -1.8367   0.03312   0.00000   -2.1428
 9 |   0.44546   0.67201   0.86185   0.69996
10 |  -0.58627   0.27885   0.45749  -0.63379
11 |  -0.06163   0.47543   0.64073  -0.02940
12 |   0.31301   0.62286   0.83018   0.59552
```

```
 13 |    1.1197   0.86859   0.96962    1.0554
 14 |   0.06163   0.52457   0.77254   0.40538
 15 |   -1.3900   0.08227   0.16939   -1.5841
 16 |  -0.91324   0.18056   0.41563  -0.77187
 17 |    1.3900   0.91773   0.98433    1.1040
 18 |   0.91324   0.81944   0.96436    1.0381
 19 |  -0.73982   0.22970   0.42057  -0.75559
 20 |   0.73982   0.77030   0.93935   0.95560
```

"l121" simple (iterated) 1-2-1 filter

```
optn = [ 2 ];
trans11 = tstrans(xd1,"l121",optn);
```

"notc" notch filter

```
f = 2.; w = .01; h = 1.;
optn = [ 2 f w h ];
trans21 = tstrans(xd1,"notc",optn);
```

"sago" Savitzky-Golay filter

```
ndeg = 2; nord = 0; nb = nf = 2;
optn = [ 2 ndeg nord nb nf ];
trans41 = tstrans(xd1,"sago",optn);
print "Trans41=",trans41;


Res = trans11 -> trans21 -> trans41;
print "L121, Notch, and Savitzky-Golay Filters", Res;
```

```
L121, Notch, and Savitzky-Golay Filters
   |         1         2         3
-----------------------------------
 1 |   0.59060   0.95010   0.95010
 2 |   0.50477   0.23110   0.23110
 3 |   0.48268   0.60680   0.38276
 4 |   0.61752   0.48600   0.66456
 5 |   0.75768   0.89130   0.76970
 6 |   0.71800   0.76210   0.78902
 7 |   0.42340   0.45650   0.34256
 8 |   0.32872   0.01850   0.34368
 9 |   0.52650   0.82140   0.46590
10 |   0.58155   0.44470   0.63915
11 |   0.61685   0.61540   0.57347
```

```
 12 |   0.78025   0.79190   0.81029
 13 |   0.84342   0.92180   0.90448
 14 |   0.64363   0.73820   0.63239
 15 |   0.37412   0.17630   0.31863
 16 |   0.48080   0.40570   0.43638
 17 |   0.79840   0.93550   0.85757
 18 |   0.79490   0.91690   0.79540
 19 |   0.65777   0.41030   0.41030
 20 |   0.65195   0.89360   0.89360
```

"perm" random permuted surrogates

```
optn = [ 2 2 12345 ];
resam11 = tstrans(xd1,"perm",optn);
```

"four" Fourier Transform

```
optn = [ 2 2 12345 ];
resam21 = tstrans(xd1,"four",optn);
```

"aaft" Amplitude Adjusted Fourier Transform

```
optn = [ 2 2 12345 ];
resam31 = tstrans(xd1,"aaft",optn);
```

"iaaf" Iterated Amplitude Adjusted Fourier

```
optn = [ 2 2 12345 ];
resam41 = tstrans(xd1,"iaaf",optn);


res = resam11 -> resam21 -> resam31 -> resam41;
print "Perm, Fourier, AAFT, IAAFT Surrogates",res;
```

```
Perm, Fourier, AAFT, IAAFT Surrogates
   |         1         2         3         4
-------------------------------------------
 1 |   0.01850   0.17630   0.89825   0.56905
 2 |   0.76210   0.48600   0.53910   0.27286
 3 |   0.40570   0.92180   0.53919   0.73729
 4 |   0.79190   0.44470   0.92130   0.57834
 5 |   0.93550   0.73820   0.38723   0.76122
 6 |   0.82140   0.41030   0.32700   0.58026
 7 |   0.95010   0.76210   0.36810    1.3598
```

8	0.93550	0.45650	0.77869	0.59685
9	0.45650	0.82140	1.0966	0.35428
10	0.73820	0.95010	0.37686	0.48752
11	0.23110	0.89360	0.41540	0.46925
12	0.61540	0.17630	0.84556	0.77450
13	0.48600	0.44470	0.73711	0.73566
14	0.60680	0.91690	0.28699	0.40510
15	0.60680	0.61540	1.2149	0.69173
16	0.01850	0.40570	0.82427	0.82169
17	0.89130	0.41030	0.73983	0.06651
18	0.89360	0.89130	0.25827	0.38302
19	0.91690	0.79190	0.38875	1.0406
20	0.23110	0.92180	0.53065	0.78857

	5	6	7	8
1	0.48600	0.93550	0.48600	0.17630
2	0.17630	0.60680	0.23110	0.60680
3	0.93550	0.44470	0.91690	0.95010
4	0.23110	0.91690	0.93550	0.89130
5	0.45650	0.95010	0.41030	0.40570
6	0.95010	0.61540	0.17630	0.89360
7	0.79190	0.45650	0.89130	0.61540
8	0.01850	0.73820	0.82140	0.01850
9	0.41030	0.89130	0.92180	0.45650
10	0.61540	0.48600	0.73820	0.44470
11	0.89130	0.40570	0.44470	0.93550
12	0.73820	0.92180	0.89360	0.82140
13	0.60680	0.89360	0.40570	0.48600
14	0.82140	0.01850	0.45650	0.23110
15	0.91690	0.41030	0.76210	0.92180
16	0.40570	0.23110	0.95010	0.41030
17	0.76210	0.79190	0.61540	0.73820
18	0.89360	0.76210	0.60680	0.79190
19	0.92180	0.82140	0.01850	0.91690
20	0.44470	0.17630	0.79190	0.76210

5.30 Function x11

< gof,newdata > = x11(data<,optn<,inidate>>)

Purpose: The x11 function seasonally adjusts univariate time series data. The implementation is based on Maiti (2009) and obtains results which are slightly different from the SAS/ETS PROC. This X11 implementation is only for the very basic seasonal adjustment and works for monthly and quarterly data. It is recommended to have at least seven years of data and will not work for less than five years of data.

Input: data the input data set is either a numeric vector or a matrix which may have the following columns:

- numeric time series variable
- string or integer date variable:

  integer monthly date this should be in the form ddmmyyyy or ddmmyy, where $dd \in 1, \ldots, 31$ and $mm \in 1, \ldots, 12$

  string monthly date this should be in the form "ddMMMyyyy" or "ddMMMyy", where $dd \in 1, \ldots, 31$ and $mm \in JAN, FEB, MAR, APR, MAY, JUN, JU$

  integer quarterly date this should be in the form yyyyq or yyq, where $q \in 1, 2, 3, 4$

  string quarterly date this should be in the form "yyyyQq" or "yyQq", where $q \in 1, 2, 3, 4$

  The day inputs are not relevant for the algorithm.
- string or integer ID variable for performing the algorithm for each value of the ID variable

The options argument specifies which columns of the input data relate to the time series, date or ID variable. The data must not include the date or the ID variable, however must be at least a vector for the the time series variable.

optn If specified this vector should be intialized to missing.

1. int specifying the amount of printed output (=0: no printed output)
2. int column number of time series variable (only needed if input data has more than one column)
3. int column number of date variable (default is no date variable)
4. int column number of ID variable (default is no ID variable)
5. =0: monthly data input (is default), =1: quarterly input data
6. defines the table returned as the "adjusted" series (=3: fully adjusted D11 table, this is default; =1: C1 table after B iteration; =2: D1 table after C iteration)

inidate If the data set does not contain a date variable this numeric or string date value specifies the start date of the time series. In integer coding this should be either in the form ddmmyyyy or ddmmyy, where $dd \in 1, \ldots, 31$ and $mm \in 1, \ldots, 12$ In string coding this should be either in the form "ddMMMyyyy" or "ddMMMyy", where $dd \in 1, \ldots, 31$ and $mm \in JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC$

Output: gof

newdata this is the same as the input data set `data` except that it contains one or more additional columns with the adjusted time series values (tables C1, D1, and D11).

Restrictions: 1. The input data columns for time series, date and ID variable should not have missing values.

2.

Relationships:

Examples: 1. First Example by Maiti (2009): Monthly Data

```
print "\n *** First Example by Maiti\n";
print "\n *** Like SAS Example but with 01jan1990\n";
print "Monthly data";
/* format date monyy7.; */
sales1= [ 112 118 132 129 121 135 148 148 136 119 104 118
          115 126 141 135 125 149 170 170 158 133 114 140
          145 150 178 163 172 178 199 199 184 162 146 166
          171 180 193 181 183 218 230 242 209 191 172 194
          196 196 236 235 229 243 264 272 237 211 180 201
          204 188 235 227 234 264 302 293 259 229 203 229
          242 233 267 269 270 315 364 347 312 274 237 278
          284 277 317 313 318 374 413 405 355 306 271 306
          315 301 356 348 355 422 465 467 404 347 305 336
          340 318 362 348 363 435 491 505 404 359 310 337
          360 342 406 396 420 472 548 559 463 407 362 405
          417 391 419 461 472 535 622 606 508 461 390 432 ];


strdat = "01JAN1990";
optn = [ 2 ,   /* ipri */
         . ];
< gof,newx > = x11(sales1,optn,strdat);
print "NEWX=",newx;


             Data set with Adjusted Time Series
             **********************************
```

```
                    Dense Matrix (144 by 4)

              |    Original        C1t        D1t       D11t
        ----------------------------------------------------
     19900101 |   112.00000  112.19294  112.19294  123.16170
     19900201 |   118.00000  118.00000  132.35108  142.66306
     19900301 |   132.00000  132.00000  135.06235  128.07081
     19900401 |   129.00000  113.63715  90.303968  93.694362
     19900501 |   121.00000  121.03189  121.03189  123.24595
     19900601 |   135.00000  135.00000  135.00000  124.27295
     19900701 |   148.00000  148.00000  148.00000  123.54395
     19900801 |   148.00000  148.87422  148.01206  123.92608
     19900901 |   136.00000  136.00000  136.00000  128.71661
     19901001 |   119.00000  119.00000  119.00000  128.77265
     19901101 |   104.00000  104.00000  104.00000  129.29629
     19901201 |   118.00000  118.00000  118.00000  130.25042
     ....................................................
     20001201 |   405.00000  405.00000  405.00000  455.19497
     20010101 |   417.00000  417.00000  417.00000  457.11270
     20010201 |   391.00000  391.00000  391.00000  452.85182
     20010301 |   419.00000  444.00490  439.43309  444.76893
     20010401 |   461.00000  461.00000  461.00000  479.02462
     20010501 |   472.00000  472.00000  472.00000  481.02088
     20010601 |   535.00000  535.00000  535.00000  471.46586
     20010701 |   622.00000  622.00000  622.00000  489.16113
     20010801 |   606.00000  606.00000  606.00000  501.87770
     20010901 |   508.00000  508.00000  508.00000  477.18732
     20011001 |   461.00000  461.00000  461.00000  498.27687
     20011101 |   390.00000  390.00000  390.00000  484.87545
     20011201 |   432.00000  432.00000  432.00000  485.54130
```

2. Second Example by Maiti (2009): Quarterly Data

```
print "\n *** Second Example by Maiti\n";
print "\n *** Quarterly Data starting with 01jan1991\n";
quart = [ "1991Q1"  6.59, "1991Q2"  6.01, "1991Q3"  6.51, "1991Q4"  6.18,
          "1992Q1"  5.52, "1992Q2"  5.59, "1992Q3"  5.84, "1992Q4"  6.33,
          "1993Q1"  6.52, "1993Q2"  7.35, "1993Q3"  9.24, "1993Q4" 10.08,
          "1994Q1"  9.91, "1994Q2" 11.15, "1994Q3" 12.40, "1994Q4" 11.64,
          "1995Q1"  9.94, "1995Q2"  8.16, "1995Q3"  8.22, "1995Q4"  8.29,
          "1996Q1"  7.54, "1996Q2"  7.44, "1996Q3"  7.80, "1996Q4"  7.28,
          "1997Q1"  5.52, "1997Q2"  5.59, "1997Q3"  5.84, "1997Q4"  6.33 ];


 optn = [ 2 ,   /* ipri */
          2 ,   /* column time series var */
```

```
          1 ,   /* column for date var */
          . ,   /* column ID variable */
          1 ];  /* quarterly data */
< gof,newx > = x11(quart,optn);
```

```
            Data set with Adjusted Time Series
            **********************************

               Mixed Type Matrix (28 by 5)

     Date Original      C1t      D1t     D11t
 1 1991Q1   6.5900   6.4135   6.4665   6.8897
 2 1991Q2   6.0100   6.0100   6.0100   6.2696
 3 1991Q3   6.5100   6.5059   6.5059   6.2413
 4 1991Q4   6.1800   6.1800   6.1800   5.9183
 5 1992Q1   5.5200   5.5200   5.5200   5.8813
 6 1992Q2   5.5900   5.5900   5.5900   5.8314
 7 1992Q3   5.8400   5.8400   5.8400   5.6025
 8 1992Q4   6.3300   6.4368   6.4216   6.1497
 9 1993Q1   6.5200   6.5200   6.5200   6.9468
10 1993Q2   7.3500   7.5155   7.5155   7.8401
11 1993Q3   9.2400   9.2400   9.2400   8.8642
12 1993Q4  10.0800  10.0800  10.0800   9.6531
13 1994Q1   9.9100   9.9545   9.9545  10.6061
14 1994Q2  11.1500  11.1500  11.1500  11.6316
15 1994Q3  12.4000  12.4000  12.4000  11.8957
16 1994Q4  11.6400  11.6400  11.6400  11.1471
17 1995Q1   9.9400   9.2652   9.2652   9.8716
18 1995Q2   8.1600   8.1600   8.1600   8.5125
19 1995Q3   8.2200   8.2200   8.2200   7.8857
20 1995Q4   8.2900   8.2900   8.2900   7.9389
21 1996Q1   7.5400   7.5400   7.5400   8.0335
22 1996Q2   7.4400   7.4400   7.4400   7.7614
23 1996Q3   7.8000   7.8000   7.8000   7.4828
24 1996Q4   7.2800   6.8063   6.8063   6.5181
25 1997Q1   5.5200   5.5200   5.5200   5.8813
26 1997Q2   5.5900   5.5900   5.5900   5.8314
27 1997Q3   5.8400   5.8400   5.8400   5.6025
28 1997Q4   6.3300   6.3300   6.3300   6.0619
```

3. Modified Second Example by Maiti (2009): Without Time Column

```
print "\n *** Modified Second Example by Maiti\n";
print "\n *** Quarterly Data starting with 01jan1991\n";
quart = [ 6.59,  6.01,  6.51,   6.18,
```

```
          5.52,  5.59,  5.84,   6.33,
          6.52,  7.35,  9.24,  10.08,
          9.91, 11.15, 12.40,  11.64,
          9.94,  8.16,  8.22,   8.29,
          7.54,  7.44,  7.80,   7.28,
          5.52,  5.59,  5.84,   6.33 ];

strdat = "01JAN1991";
optn = [ 2 ,   /* ipri */
         1 ,   /* column time series var */
         . ,   /* column for date var */
         . ,   /* column ID variable */
         1 ];  /* quarterly data */
< gof,newx > = x11(quart,optn,strdat);
```

The results are the same as for the other specification.

4. Third Example by Maiti (2009): Monthly Data with ID

```
print "\n *** Third Example Newly Submitted\n";
print "\n *** With ID variable and starting with 01jan1990\n";
print "Monthly data";
/* format date monyy7.; */
sales1= [ 112 118 132 129 121 135 148 148 136 119 104 118
          115 126 141 135 125 149 170 170 158 133 114 140
          145 150 178 163 172 178 199 199 184 162 146 166
          171 180 193 181 183 218 230 242 209 191 172 194
          196 196 236 235 229 243 264 272 237 211 180 201
          204 188 235 227 234 264 302 293 259 229 203 229
          242 233 267 269 270 315 364 347 312 274 237 278
          284 277 317 313 318 374 413 405 355 306 271 306
          315 301 356 348 355 422 465 467 404 347 305 336
          340 318 362 348 363 435 491 505 404 359 310 337
          360 342 406 396 420 472 548 559 463 407 362 405
          417 391 419 461 472 535 622 606 508 461 390 432 ];
```

Generate ID variable:

```
n = ncol(sales1);
sales2 = sales1 / 10.;
sales = (sales1 -> sales2)‘;
s1 = cons(n,1,"LA");
s2 = cons(n,1,"CA");
sales = sales -> (s1 |> s2);
print "Sales=",sales;
```

```
strdat = "01JAN1990";
optn = [ 2 ,   /* ipri */
         1 ,   /* column time series var */
         . ,   /* column for date var */
         2 ];  /* column ID variable */
< gof,newx > = x11(sales,optn,strdat);
print "NEWX=",newx;
```

Data set with Adjusted Time Series
****

Mixed Type Matrix (288 by 5)

	Original	ID	C1t	D1t	D11t
19900101	112.0000	LA	112.1929	112.1929	123.1617
19900201	118.0000	LA	118.0000	132.3511	142.6631
19900301	132.0000	LA	132.0000	135.0624	128.0708
19900401	129.0000	LA	113.6372	90.3040	93.6944
19900501	121.0000	LA	121.0319	121.0319	123.2460
19900601	135.0000	LA	135.0000	135.0000	124.2730
19900701	148.0000	LA	148.0000	148.0000	123.5440
19900801	148.0000	LA	148.8742	148.0121	123.9261
19900901	136.0000	LA	136.0000	136.0000	128.7166
19901001	119.0000	LA	119.0000	119.0000	128.7726
19901101	104.0000	LA	104.0000	104.0000	129.2963
19901201	118.0000	LA	118.0000	118.0000	130.2504
	.....				
20010101	417.0000	LA	417.0000	417.0000	457.1127
20010201	391.0000	LA	391.0000	391.0000	452.8518
20010301	419.0000	LA	444.0049	439.4331	444.7689
20010401	461.0000	LA	461.0000	461.0000	479.0246
20010501	472.0000	LA	472.0000	472.0000	481.0209
20010601	535.0000	LA	535.0000	535.0000	471.4659
20010701	622.0000	LA	622.0000	622.0000	489.1611
20010801	606.0000	LA	606.0000	606.0000	501.8777
20010901	508.0000	LA	508.0000	508.0000	477.1873
20011001	461.0000	LA	461.0000	461.0000	498.2769
20011101	390.0000	LA	390.0000	390.0000	484.8755
20011201	432.0000	LA	432.0000	432.0000	485.5413
19900101	11.2000	CA	11.2193	11.2193	12.3162
19900201	11.8000	CA	11.8000	13.2351	14.2663
19900301	13.2000	CA	13.2000	13.5062	12.8071
19900401	12.9000	CA	11.3637	9.0304	9.3694
19900501	12.1000	CA	12.1032	12.1032	12.3246

```
19900601   13.5000  CA   13.5000   13.5000   12.4273
19900701   14.8000  CA   14.8000   14.8000   12.3544
19900801   14.8000  CA   14.8874   14.8012   12.3926
19900901   13.6000  CA   13.6000   13.6000   12.8717
19901001   11.9000  CA   11.9000   11.9000   12.8773
19901101   10.4000  CA   10.4000   10.4000   12.9296
19901201   11.8000  CA   11.8000   11.8000   13.0250
          ..............................

20010101   41.7000  CA   41.7000   41.7000   45.7113
20010201   39.1000  CA   39.1000   39.1000   45.2852
20010301   41.9000  CA   44.4005   43.9433   44.4769
20010401   46.1000  CA   46.1000   46.1000   47.9025
20010501   47.2000  CA   47.2000   47.2000   48.1021
20010601   53.5000  CA   53.5000   53.5000   47.1466
20010701   62.2000  CA   62.2000   62.2000   48.9161
20010801   60.6000  CA   60.6000   60.6000   50.1878
20010901   50.8000  CA   50.8000   50.8000   47.7187
20011001   46.1000  CA   46.1000   46.1000   49.8277
20011101   39.0000  CA   39.0000   39.0000   48.4875
20011201   43.2000  CA   43.2000   43.2000   48.5541
```

5. Fourth Example by Maiti (2009): Quarterly Data with ID

```
print "\n *** Fourth Example Newly Submitted\n";
print "\n *** Quarterly Data with ID starting 01jan1991\n";
quart1 = [ 6.59,  6.01,  6.51,   6.18,
           5.52,  5.59,  5.84,   6.33,
           6.52,  7.35,  9.24,  10.08,
           9.91, 11.15, 12.40,  11.64,
           9.94,  8.16,  8.22,   8.29,
           7.54,  7.44,  7.80,   7.28,
           5.52,  5.59,  5.84,   6.33 ];
```

Create ID column:

```
n = nrow(quart1);
quart2 = quart1 / 2.;
quart = quart1 |> quart2;
s1 = cons(n,1,"A");
s2 = cons(n,1,"B");
quart = quart -> (s1 |> s2);
print "Sales=",quart;
```

```
strdat = "01JAN1991";
optn = [ 2 ,   /* ipri */
         1 ,   /* column time series var */
         . ,   /* column for date var */
         2 ,   /* column ID variable */
         1 ];  /* quarterly data */
< gof,newx > = x11(quart,optn,strdat);
```

Data set with Adjusted Time Series
****

Mixed Type Matrix (56 by 5)

	Original	ID	C1t	D1t	D11t
199103	6.5900	A	6.4135	6.4665	6.8897
199106	6.0100	A	6.0100	6.0100	6.2696
199109	6.5100	A	6.5059	6.5059	6.2413
199112	6.1800	A	6.1800	6.1800	5.9183
199203	5.5200	A	5.5200	5.5200	5.8813
199206	5.5900	A	5.5900	5.5900	5.8314
199209	5.8400	A	5.8400	5.8400	5.6025
199212	6.3300	A	6.4368	6.4216	6.1497
199303	6.5200	A	6.5200	6.5200	6.9468
199306	7.3500	A	7.5155	7.5155	7.8401
199309	9.2400	A	9.2400	9.2400	8.8642
199312	10.0800	A	10.0800	10.0800	9.6531
.....					
199603	3.7700	B	3.7700	3.7700	4.0168
199606	3.7200	B	3.7200	3.7200	3.8807
199609	3.9000	B	3.9000	3.9000	3.7414
199612	3.6400	B	3.4032	3.4032	3.2590
199703	2.7600	B	2.7600	2.7600	2.9406
199706	2.7950	B	2.7950	2.7950	2.9157
199709	2.9200	B	2.9200	2.9200	2.8012
199712	3.1650	B	3.1650	3.1650	3.0310

5.31 Function xsimplex

x = xsimplex(p,n)

Purpose: The xsimplex function generates all points on a (p, n) simplex lattice; that are all p-part compositions of n.

Input: **p** should be integer scalar

n should be integer scalar

Output: Returns a matrix where each row contains a p dimensional vector of nonnegative integers that sum to n.

Restrictions: 1. The input data must not contain any missing values.

Relationships: combn(), hcube(), nsimplex()

Examples:

```
z0 = xsimplex(3,4);  print "Z0=", z0;
```

```
Z0=
 L |   1   2   3
---------------
 1 |   4   0   0
 2 |   3   1   0
 3 |   3   0   1
 4 |   2   2   0
 5 |   2   1   1
 6 |   2   0   2
 7 |   1   3   0
 8 |   1   2   1
 9 |   1   1   2
10 |   1   0   3
11 |   0   4   0
12 |   0   3   1
13 |   0   2   2
14 |   0   1   3
15 |   0   0   4
```

```
z1 = xsimplex(2,3);  print "Z1=", z1;
```

```
Z1=
 L |   1   2
-----------
```

```
 1 |   3   0
 2 |   2   1
 3 |   1   2
 4 |   0   3
```

```
  z2 = xsimplex(2,5);  print "Z2=", z2;
```

```
Z2=
 L |   1   2
-----------
 1 |   5   0
 2 |   4   1
 3 |   3   2
 4 |   2   3
 5 |   1   4
 6 |   0   5
```

```
  z3 = xsimplex(5,2);  print "Z3=", z3;
```

```
Z3=
 L |   1   2   3   4   5
-------------------------
 1 |   2   0   0   0   0
 2 |   1   1   0   0   0
 3 |   1   0   1   0   0
 4 |   1   0   0   1   0
 5 |   1   0   0   0   1
 6 |   0   2   0   0   0
 7 |   0   1   1   0   0
 8 |   0   1   0   1   0
 9 |   0   1   0   0   1
10 |   0   0   2   0   0
11 |   0   0   1   1   0
12 |   0   0   1   0   1
13 |   0   0   0   2   0
14 |   0   0   0   1   1
15 |   0   0   0   0   2
```

6 Illustration

6.1 Evaluating Logistic and Proportional Odds Model Fit

For continuous predictors and bivariate and multilevel ordinal response the Hosmer-Lemeshow χ^2 goodness-of-fit test for the logistic model can be biased and of low power (Xie, Pendergast, & Clarke, 2008, Xie & Bian, 2009). A new algorithm was proposed which decomposes the data set into clusters wrt. of similarity of observations measured by the predictors.
The data set `JSS450.dat` consisting of `nrow=997` rows (observations) and six columns containing five continuous predictor variables and an ordinal three level response `case` is available in the `tdata` directory:

```
options NOECHO;
%inc "..\\tdata\\JSS450.dat";
options ECHO;
```

We want to extract nclus=10 clusters from nrow=997 observations of the data:

```
nrow = nrow(data1); ncol = ncol(data1);
nclus = 10; ny = 3; nvar = 5;
print "DATA1: nr=",nrow," nc=",ncol," nclus=",nclus," ny=",ny;
vnam1 = [" TXASSIGN AGE BLDLOSS NIHSSPRE TSAHTOIND CASE "];
print "Data1=",data1[1:10,];
cnam1 = [" Cluster Case IP1 IP2 IP3 "];
print "Clus1=",clus1[1:10,];
cmem1 = clus1[,1];
```

```
Data1=
   |    TXASSIGN         AGE     BLDLOSS
-----------------------------------------
 1 |           2          48         200
 2 |           2          27         300
 3 |           1          53         400
 4 |           1          71         300
 5 |           2          28         500
 6 |           1          39         400
 7 |           2          59         500
 8 |           1          44         500
 9 |           2          20         500
10 |           2          41         500

   |    NIHSSPRE   TSAHTOIND        CASE
-----------------------------------------
```

```
 1 |          0          7          1
 2 |          0          1          1
 3 |          2          3          1
 4 |          6          1          1
 5 |          1          2          1
 6 |          0          2          1
 7 |         -1          0          1
 8 |          0          1          1
 9 |          0          2          1
10 |          0          0          1
```

These rae the clusters when SAS/PROC Cluster is applied to the standardized data with Ward's hierarchical method:

```
Clus1=
   |   Cluster      Case       IP1       IP2       IP3
-------------------------------------------------------
 1 |    6.0000   1.00000   0.71603   0.18989   0.09409
 2 |    3.0000   1.00000   0.80768   0.13362   0.05869
 3 |   1.00000   1.00000   0.62560   0.23891   0.13549
 4 |    8.0000   1.00000   0.36499   0.32201   0.31301
 5 |    3.0000   1.00000   0.73901   0.17634   0.08465
 6 |   1.00000   1.00000   0.77528   0.15417   0.07055
 7 |    4.0000   1.00000   0.67545   0.21279   0.11177
 8 |   1.00000   1.00000   0.74214   0.17446   0.08340
 9 |    3.0000   1.00000   0.80505   0.13532   0.05963
10 |    3.0000   1.00000   0.72356   0.18549   0.09095
```

For the cluster analysis the data `data2` should not contain the response variable:

```
vnam2 = [" TXASSIGN AGE BLDLOSS NIHSSPRE TSAHTOIND "];
data2 = data1[,1:5];
data2 = cname(data2,vnam2);
```

For standardization we simply use `univariate()` to compute the mean and standard deviation of the columns of `data2`:

```
print "Mean=0, Var=1: standardize";
sopt = [" ari std "];
must  = univar(data2,sopt); print must;
xmu = cons(nrow,1,1.) * must[1,]; /* print xmu[1:10,]; */
xst = cons(nrow,1,1.) * must[2,];
xdat = (data2 - xmu) ./ xst;      /* print xdat[1:10,]; */
mus2 = univar(xdat,sopt); print "Test stand:", mus2;
```

```
           |    TXASSIGN         AGE     BLDLOSS
-------------------------------------------------
  Ari_Mean |      1.5005      51.707      422.14
  Std_Dev  |     0.50025      12.539      391.11

           |    NIHSSPRE   TSAHTOIND
-----------------------------------
  Ari_Mean |      1.1143      3.2016
  Std_Dev  |      2.5084      2.9460
```

The eigenvalues of the covariance matrix of the standardized data are:

```
  print "COV and DIST Matrices";
  cov = bivar(xdat,"cov");
  eval = dia2vec(eig(cov));
  print "Eigval=",eval;
```

```
Eigval=
   |         1
--------------
 1 |   0.80143
 2 |   0.87394
 3 |    1.0105
 4 |    1.0824
 5 |    1.2318
```

```
  optn = [ 1 ,  /* iscl: mean=0, var=1 */
           0 ]; /* ipri */
  dist = dist(xdat,"l2",optn);
  print dsub = dist[1:10,1:10];
```

```
 S |         1         2         3         4         5
-------------------------------------------------------
 1 |   0.00000
 2 |    2.6492   0.00000
 3 |    2.6260    3.0753   0.00000
 4 |    4.1588    4.6937    2.2649   0.00000
 5 |    2.4843   0.73621    2.8828    4.4840   0.00000
 6 |    2.7664    2.2567    1.4134    3.5235    2.2338
 7 |    2.6763    2.6549    2.5995    3.6161    2.6849
 8 |    2.9722    2.4689    1.2950    3.2588    2.4287
 9 |    2.9078   0.82968    3.4262    5.1612   0.75232
10 |    2.5585    1.2741    2.5787    3.9772    1.3018
```

```
 S |         6         7         8         9        10
-------------------------------------------------------
 6 |   0.00000
 7 |    2.6880   0.00000
 8 |   0.58275    2.3877   0.00000
 9 |    2.5214    3.2084    2.7883   0.00000
10 |    2.1325    1.4899    2.0417    1.8072   0.00000
```

The cmem2 return argument contains the cluster structure:

```
print "[1] Ward Cluster Analysis for 10 Clusters";
optn = [ "data"     "raw" ,
         "meth"    "ward" ,
         "scale"     "st" ,
         "metr"   "l2dis" ,
         "print"        2 ,
         "ncl"         10 ];
< part,prop,cmem2 > = cluster(data2,"agnes",optn);
```

The cluster membership of computing Ward's algorithm by SAS PROC CLUSTER and CMAT's `cluster()` function are very similar (of course cluster numbers are different):

```
/* print "Part=", part;
   print "Prop=", prop; */
print "My cluster membership is close to that of SAS";
print "Maybe just one tied union is different";
nc1 = nc2 = cons(10,1,0.);
for (i = 1; i <= nrow; i++) {
  nc1[cmem1[i]] += 1;
  nc2[cmem2[i]] += 1;
}
print "Distribution of Cluster Membership",nc1 -> nc2;
```

```
Distribution of cluster membership
   |     1     2
---------------
 1 |   235    97
 2 |   171   157
 3 |   153   235
 4 |   160    65
 5 |    71   156
 6 |    97   171
```

```
 7 |    27    13
 8 |    65    71
 9 |    13    27
10 |     5     5
```

For computiung the multilevel ordinal logistic regression we need to use `data1` where column 6 contains the response variable:

```
print "[2] Get yhat[nr] of logistic model";
clas = 6;
modl = "6 = 1:5 ";
optn = [ "print"            5 ,
         "popt"             3 ,
         "link"       "logit" ,
         "dist"       "binom" ,
         "predpr"         "i" ,  /* ind. predprobs */
         "notype1"            ,
         "notype3"            ,
         "yord"               ,
         "tech"      "trureg" ];
< gof,parm,sterr,conf,cov,typ1,typ3,yhat > = glim(data1,modl,optn,clas);
print "Type1,3=",typ1,typ3;
print "Yhat=",yhat[1:10,];
```

The following are the first ten rows of the `yhat` output:

```
Yhat=
   |     Y_obs   Y_frequ     Y_hat     Resid     Y_low     Y_upp
-------------------------------------------------------------------
 1 |   0.00000   1.00000   0.71602  -0.71602   0.65865   0.76715
 2 |   0.00000   1.00000   0.80768  -0.80768   0.74906   0.85526
 3 |   0.00000   1.00000   0.62560  -0.62560   0.57806   0.67083
 4 |   0.00000   1.00000   0.36497  -0.36497   0.28010   0.45916
 5 |   0.00000   1.00000   0.73901  -0.73901   0.67188   0.79656
 6 |   0.00000   1.00000   0.77528  -0.77528   0.72815   0.81630
 7 |   0.00000   1.00000   0.67544  -0.67544   0.61075   0.73406
 8 |   0.00000   1.00000   0.74214  -0.74214   0.69351   0.78544
 9 |   0.00000   1.00000   0.80505  -0.80505   0.73540   0.85986
10 |   0.00000   1.00000   0.72355  -0.72355   0.66771   0.77320


   |   AStdErr      IP_1      IP_2      IP_3    _INTO_
-----------------------------------------------------
 1 |   0.13649   0.71602   0.18990   0.09408   1.00000
 2 |   0.17420   0.80768   0.13363   0.05869   1.00000
```

```
 3 |   0.10131   0.62560   0.23892   0.13548   1.00000
 4 |   0.19905   0.36497   0.32203   0.31300   1.00000
 5 |   0.16537   0.73901   0.17634   0.08465   1.00000
 6 |   0.12914   0.77528   0.15417   0.07054   1.00000
 7 |   0.14411   0.67544   0.21280   0.11176   1.00000
 8 |   0.12272   0.74214   0.17447   0.08340   1.00000
 9 |   0.20202   0.80505   0.13532   0.05963   1.00000
10 |   0.13486   0.72355   0.18550   0.09095   1.00000
```

The IP values are the same as obtained with executing SAS PROC LOGISTIC. However, since the cluster results between SAS PROC CLUSTER and CMAT `cluster()` function are slightly different we will here use the SAS cluster results to obtain the same goodness-of-fit measures as the authors of JSS450 did:

```
  print "Compute tab1=fab0[nclus,ny] and tab1=cab0[ny*nclus,ny+2]";
  print "cab0: col=1: Observed , col=2: Expected";
  pab0 = cons(nclus,ny);
  fab0 = cons(nclus,ny);
  cab0 = cons(nclus*ny,ny+1);
  for (ir = 1; ir <= nrow; ir++) {
    ic = clus1[ir,1]; iy = clus1[ir,2];
    fab0[ic,iy] += 1;
    pab0[ic,1] += yhat[ir, 8];
    pab0[ic,2] += yhat[ir, 9];
    pab0[ic,3] += yhat[ir,10];
    k = (iy-1) * nclus + ic;
    cab0[k,1] += 1;
    cab0[k,2] += yhat[ir,8];
    cab0[k,3] += yhat[ir,9];
    cab0[k,4] += yhat[ir,10];
  }
  cnam = [" Obs CP1 CP2 CP3 "];
  cab0 = cname(cab0,cnam);
  print "Observed: CAB0=",cab0;
  print "Expected: PAB0=",pab0;
```

```
Observed: CAB0=
   |       Obs       CP1       CP2       CP3
-------------------------------------------------
 1 |    173.00    131.31    28.254    13.440
 2 |   100.000    63.179    23.305    13.516
 3 |    108.00    79.145    19.295    9.5602
 4 |    110.00    67.788    26.439    15.773
 5 |    50.000    32.062    11.395    6.5428
```

```
 6 |    62.000    42.627    12.626    6.7473
 7 |    13.000    5.8569    3.7986    3.3445
 8 |    23.000    9.7760    6.5493    6.6748
 9 |    2.0000   0.18852   0.37354    1.4379
10 |   1.00000   0.27154   0.31584   0.41262
11 |    52.000    38.433    9.1063    4.4608
12 |    39.000    24.167    9.3292    5.5043
13 |    36.000    25.225    7.0430    3.7316
14 |    26.000    16.046    6.2594    3.6942
15 |    13.000    8.3404    2.9626    1.6970
16 |    25.000    15.633    5.8912    3.4759
17 |    5.0000    1.7608    1.5933    1.6459
18 |    13.000    4.1817    3.8710    4.9472
19 |    3.0000   0.29118   0.55921    2.1496
20 |   1.00000   0.16075   0.26171   0.57754
21 |   10.0000    7.2126    1.8578   0.92960
22 |    32.000    19.202    7.8113    4.9862
23 |    9.0000    6.2901    1.7852   0.92473
24 |    24.000    14.207    6.0209    3.7717
25 |    8.0000    4.6537    1.9726    1.3737
26 |   10.0000    6.1409    2.4169    1.4423
27 |    9.0000    3.6544    2.6952    2.6504
28 |    29.000    9.6132    8.5548    10.832
29 |    8.0000   0.69526    1.3538    5.9510
30 |    3.0000   0.62592   0.67970    1.6944


Expected: PAB0=
   |         1         2         3
---------------------------------
 1 |    176.95    39.218    18.830
 2 |    106.55    40.445    24.007
 3 |    110.66    28.123    14.217
 4 |    98.042    38.720    23.239
 5 |    45.056    16.330    9.6135
 6 |    64.401    20.934    11.665
 7 |    11.272    8.0871    7.6408
 8 |    23.571    18.975    22.454
 9 |    1.1750    2.2865    9.5385
10 |    1.0582    1.2573    2.6845


  /* Compute tab2[]: with Expected, Pearson, and Deviance */
  cab1 = cons(ny*nclus,4);
  k = 1;
  for (iy = 1; iy <= ny; iy++)
```

```
  for (ic = 1; ic <= nclus; ic++, k++) {
    cab1[k,1] = obs = cab0[k,1];
    cab1[k,2] = exp = pab0[ic,iy];
    cab1[k,3] = (obs - exp)**2 / exp;
    cab1[k,4] = 2.*obs * log(obs /exp);
  }
  cnam = [" Obs Exp Pearson Deviance "];
  cab1 = cname(cab1,cnam);
  print "CAB1=",cab1;
```

```
CAB1=
   |        Obs        Exp    Pearson   Deviance
-------------------------------------------------
 1 |     173.00     176.95      0.088    -7.8153
 2 |    100.000     106.55    0.40242    -12.685
 3 |     108.00     110.66      0.064    -5.2562
 4 |     110.00     98.042     1.4585     25.319
 5 |     50.000     45.056    0.54241     10.411
 6 |     62.000     64.401      0.090    -4.7111
 7 |     13.000     11.272    0.26485     3.7079
 8 |     23.000     23.571      0.014    -1.1279
 9 |     2.0000     1.1750    0.57932     2.1276
10 |    1.00000     1.0582      0.003   -0.11315
11 |     52.000     39.218     4.1660     29.339
12 |     39.000     40.445      0.052    -2.8384
13 |     36.000     28.123     2.2062     17.779
14 |     26.000     38.720     4.1785    -20.709
15 |     13.000     16.330    0.67908    -5.9295
16 |     25.000     20.934    0.78987     8.8758
17 |     5.0000     8.0871     1.1784    -4.8083
18 |     13.000     18.975     1.8815    -9.8327
19 |     3.0000     2.2865    0.22265     1.6296
20 |    1.00000     1.2573      0.053   -0.45786
21 |    10.0000     18.830     4.1406    -12.657
22 |     32.000     24.007     2.6615     18.394
23 |     9.0000     14.217     1.9141    -8.2292
24 |     24.000     23.239      0.025     1.5475
25 |     8.0000     9.6135    0.27081    -2.9397
26 |    10.0000     11.665    0.23778    -3.0810
27 |     9.0000     7.6408    0.24180     2.9471
28 |     29.000     22.454     1.9084     14.838
29 |     8.0000     9.5385    0.24816    -2.8144
30 |     3.0000     2.6845      0.037    0.66661
```

```
  pear = cab1[+,3]; devi = cab1[+,4];
```

```
df = (nclus-1.)*(ny-1.) - nvar / 2.;
ppear = cdf("chis",pear,df);
print "Pearson: Chisqu=",pear," df=",df," prob=",ppear;
pdevi = cdf("chis",devi,df);
print "Deviance: GOF=",devi," df=",df," prob=",pdevi;
```

Using the SAS PROC CLUSTER results we obtain for the goodness-of-fit measures:

```
Pearson: Chisqu= 30.598 df= 15.500 prob= 0.01232
Deviance: GOF= 31.576 df= 15.500 prob= 0.009170
```

Using the `cmat` results of the `cluster` function we obtain the following goodness-of-fit:

```
Pearson: Chisqu= 29.289 df= 15.500 prob= 0.01812
Deviance: GOF= 30.130 df= 15.500 prob= 0.01416
```