CMAT© Newsletter: July 2014

Wolfgang M. Hartmann

July 2014

Contents

1 General Remarks

A number of bugs were fixed.
Mike Powell's LINCOA Fortran code was linked into the `nlp()` function. It is an interpolation approach to linearly constrained nonlinear optimization similar to the BOBYQA, UOBYQA, and NEWUOA software Powell's. It is designed to work without specifying the derivatives of the objective function, and especially for inequality constraints. However, any linear equality constraint must be formulated as two inquality constraints, once with smaller than and once with larger than the right hand side with different signs. Here in CMAT, however, this is implicitly realized by the `nlp` interface and there is no need for the user to take care of that.
The Windows DLL of `lpsolve` version 5.5 was linked into CMAT and can now be called with the LP function. Using `lpsolve` the following problems can be solved:

- the common (linearly constrained) LP optimization problem including those with integer constraints,
- the specific linear transport problem (all estimates are integer),
- the specific linear assignment problem.

For all methods a sensitivity analysis and the duals of the optimal estimates can be computed. Note, that `lpsolve` by default imposes zero lower bounds on the etsimates.
In addition to solving the linear assignment problem with the `lpsolve` DLL we implemented a version of the `LAPJV` and `BOTJV` code by Jonker & Volgenant (1987) was implemented for the linear assignment problem (`LAPJV`) and the linear bottleneck assignment problem (`BOTJV`).
The code for the new `knapsack()` function is similar to some found on the internet and yields probably only approximate solutions. We may return to that function later when we have some appropriate LP interface for that.
The new `tsp()` function for solving the *traveling salesman problem* is partially based on some code found in the CRAN function. For the *Linkern* and *Concorde* method we obtained the code from Prof. W. J. Cook of the University of Waterloo, Canada, under the restriction that it can only be used for "academic research" purposes. Using the two algorithms for other (than academic research) purposes needs the specific agreement of the developers. Please contact Prof. William J. Cook, at `bico@uwaterloo.ca` if you want to use the CMAT function `tsp` with the `"conc"` option. The *Concorde* code requires an interface to an LP solver. Dr. John Forrest was so friendly to provide an interface between *Clp* (in COIN-OR) and *Concorde*. We thank Prof. William J. Cook for his generous agreement to embed his and his coworkers code into CMAT. All other algorithms for the `tsp()` function were completely new programmed.
The new `latlong()` function is based mainly on information from websites (e.g. E. Williams).

1.1 New Functions

boundbox find the coordinates and the volume of the smallest box surrounding a set of n points $X_{ij}, i = 1, \ldots, n, j = 1, 2$ in 2-dimensional space

hamilton find all or some of Hamiltonian circuits in a graph

knapsack (approximately) solving the one- and multidimensional knapsack problem

latlong computes some measures based on latitude and longitude data

locat1 solves the multifacility location problem with rectilinear distance (Cheung, 1980)

lpassign solving the linear assignment problem (LAP)

- by interfacing the DLL of `lpsolve` (Berkelaar et al., 2004),
- using the Hungarian method and the LAPV algorithm by Jonker & Volgenant (1987) can be used for solving the LAP
- the linear bottleneck assignment problem (LBAP) can be solved using an algorithm by Jonker & Volgenant (1987)

lptransp computes estimates of the linear transport problem by interfacing the DLL of `lpsolve`

maxempty find the coordinates and the volume of the largest box parallel to the axes of x and y and completely surrounded by some points of a specified set of n points $X_{ij}, i = 1, \ldots, n, j = 1, 2$ in 2-dimensional space (programmed based on an algorithm by Hans W. Borchers in CRAN).

mstgra minimum spanning tree with sparse or dense graph input. Note that the function `mstree()` was renamed into `mstdis()` for minimum spanning tree based on distances among n points.

splnet finds the shortest path between two nodes a and b in a network.

tsp implements several algorithms trying to solve the NP-hard traveling salesman problem. Note, some of these can only be used for "academic research".

1.2 Fixed Bugs

2 Modifications of Features

The default of the sample size for the `clara` algorithm ("Clustering LARge Applications") in function `cluster()` was changed. For untypically small applications where the data have less than 100 observations, and where the `pam` algorithm ("Partitioning Around Medoids") is preferred, the full sample size is used.

3 Extensions to the Language

4 Extensions to Various Functions

4.1 Extensions of `lp()` Function

The definition of the `lp()` call was extended to the following:

xr = lp("meth",c,lau<,optn<,lubc<,xint>>>)

< xr,lm,rp,duals,sens > = lp("meth",c,lau<,optn<,lubc<,xint>>>)

The list of input arguments changed to:

"meth" is a string argument specifying the optimization method. Currently only three methods are available:

- `"lps"`: The package `lpsolve`, version 5.5 is being used (Berkelaar et al., 2004). Integer constraints can be specified with `xind`.
- `"con"`: The dense continuation method by Madsen and Pinar (1993) also called *LPASL* is used. Integer constraints (`xind`) cannot be specified.
- `"pcx"`: The sparse interior point method by Wright (1997) which is also implemented with the `pcx` function (see Czyzyk, Mehrotra, Wagner, and Wright, 1997). Integer constraints (`xind`) cannot be specified.

c is the name of a n vector specifying the linear objective function $c^T x$ which is to be minimized or maximized.

lau is the name of a $n_{lc} \times n + 2$ or $n_{lc} \times n + 1$ matrix with lower bounds (first column), coefficients, and upper bounds (last column) for n_{lc} general linear constraints. The specification of missing values for lower or upper bounds means that the corresponding side constraint is not imposed. If `lau` contains only $n+1$ columns, where n is defined by the dimension of the first argument `c`, then no upper bounds are imposed. This argument may be specified as a missing value if only boundary constraints are imposed with the last input argument.

optn (optional) specifies some of a set of options. Almost all options require a 2-column options matrix, where the first column contains a specific string and the second column contains a value specifying the users choice. In addition to the already documented options we added `"dual"` and `"sens"` for specifying the printed output of duals and sensitivities even when the last two return arguments are not required.

lubc (optional) is the name of a $n \times 2$ matrix with lower (first column) and upper (second column) simple boundary constraints.

xint (optional) is the name of a k vector specifying those indices of x which are subjected to integer constraints.

And there are two more output arguments when the `"lps"` method is specified:

duals Let nc be the number of linear and boundary constraints, the `duals` returns the $3 \times (n + nc)$ matrix of

1. duals for etsimates and constraints in its first row
2. the lower and upper bounds ("from to") of the sensitivities of the duals in rows two and three.

sens Returns the $2 \times n$ matrix of the lower and upper bounds of the sensitivities ("from to") of the estimates.

Some examples:

1. Computing duals and sensitivity:

```
print "Example from CRAN: Obtain sensitivities";
c = [ 1. 9. 3. ];
cnam = [" eins neun drei "];
c = cname(c,cnam);
lau = [ .  1. 2. 3.  9. ,
        .  3. 2. 2. 15. ];
cnam = [ "LC_1 LC_2 "];
lau = cname(lau,cnam);


print "crit=40.5, x= [ 0. 4.5 0. ]";
print "Default Lower Bounds for LPSOLV are 0.";
optn = [ "max"          ,
         "sens"         ,
         "print"      6 ];
< xr,lm,rc,sens,duals > = lp("lps",c,lau,optn);
print "LPSOLVE: Xsolution=", xr;
print "Lagrange Multipliers=", lm;
print "GOF=", rc;
print "SENS=", sens;
print "DUALS=", duals;


                    ********************
                    Optimization Results
                    ********************

                    Parameter Estimates
                    -------------------
```

Parameter	Estimate	Lower BC	Upper BC	Active BC
1 eins	0	0	.	
2 neun	4.5000000	0	.	
3 drei	0	0	.	

Value of Objective Function = 40.5

Active Constraints 1 Solve Time 0.01

First Order Lagrange Multipliers

Constraint	Lag. Mult.
Linear IC [1]	2.000000000

Linear Constraints Evaluated at Solution

```
[ 1]ACT   -1.00000 * eins -  2.00000 * neun -  3.00000 * drei
                                            +  9.00000 =           0
[ 2]      -3.00000 * eins -  2.00000 * neun -  2.00000 * drei
                                            +  15.0000 =  6.000000000
```

Sensitivities of Estimates

Dense Matrix (2 by 3)

	eins	neun	drei
Sens_from	-1.00e+030	2.0000000	-1.00e+030
Sens_to	4.5000000	1.00e+030	13.500000

Duals of Constraints and Estimates

Dense Matrix (3 by 5)

	LIC_1	LIC_2	eins	neun	drei

```
    Duals |  -4.5000000          0 -3.5000000          0 -10.500000
Dual_from |  -15.000000 -1.00e+030 -1.00e+030 -1.00e+030 -6.0000000
  Dual_to |           0  1.00e+030  3.0000000  1.00e+030  3.0000000
```

2. Specifying and solving an integer problem:

```
print "Example from CRAN: Integer constraints";
c = [ 1. 9. 3. ];
cnam = [" eins neun drei "];
c = cname(c,cnam);
lau = [ .  1. 2. 3.  9. ,
        .  3. 2. 2. 15. ];
cnam = [ "LC_1 LC_2 "];
lau = cname(lau,cnam);


print "Results is crit=37 xr=[ 1 4 0 ]";
xint = [ 1:3 ];
optn = [ "max"          ,
         "print"      6 ];
< xr,lm,rc > = lp("lps",c,lau,optn,.,xint);
print "LPSOLVE: Xsolution=", xr;
print "Lagrange Multipliers=", lm;
print "GOF=", rc;


                    ********************
                     Optimization Results
                     ********************

                     Parameter Estimates
                     -------------------

     Parameter   Estimate   Lower BC   Upper BC   Active BC

    1 eins      1.0000000          0     .
    2 neun      4.0000000          0     .
    3 drei              0          0     .

         Value of Objective Function =           37
  Active Constraints        1    Solve Time             0.02


                First Order Lagrange Multipliers
```

```
-------------------------------

 Constraint            Lag. Mult.

Linear IC  [1]        2.000000000


            Linear Constraints Evaluated at Solution
            ----------------------------------------

[ 1]ACT   -1.00000 * eins -  2.00000 * neun -  3.00000 * drei
                                          +  9.00000 =           0
[ 2]      -3.00000 * eins -  2.00000 * neun -  2.00000 * drei
                                          +  15.0000 =  4.000000000
```

4.2 Extensions of `nlp()` Function

Professor Mike Powell's optimization algorithm LINCOA was added to the list of available algorithms of the `nlp()` function. With the help of the *Intel Parallel Studio XE 2013* for Fortran/Fortran90 compilation an easy interface with the calling C code in `cmat` was implemented which calls directly Prof. Powell's Fortran code. Not much had to be added to the options of `nlp()` since most were already made available for the UOBYQA (NEWUOA) and BOBYQA techniques:

- the `"tech"` option now permits `"lincoa"` (and the shorter alias `"lnc"`)
- the number of interpolation points can be specified with the `"intpoi"` option inside the range $[n+1, (n+1)(n+2)/2]$ and is otherwise set by default to $2n+1$, the same as for BOBYQA
- the `"rhobeg"` (and as an alias the `"instep"`) option can be used for specifying `RHOBEG`, the size of the initial trust region, which is by default = .5, the same as for BOBYQA
- the `"rhoend"` (and as an alias the `"absxtol"`) option can be used for specifying `RHOEND`, the size of the final trust region, which is by default = $1.e-6$, the same as for BOBYQA
- the maximum number of function values for LINCOA is set by default to 3000, the same as for BOBYQA, but of course, you may just use the `"maxit"` option for setting a larger limit.

In my way of thinking of the methods UOBYQA, NEWUOA, BOBYQA, and LINCOA I differ slightly with Prof. Powell: when I am talking of iterations I mean all computations which is performed for a constant value of ρ, the size of the trust region. There for the number of changes of ρ when stepping from `"RHOBEG"` to `"RHOEND"` is for me the number of iterations, which is visible in the printout of the iteration history.
The new `"rhobeg"` and `"rhoend"` option names were added for an easier understanding of COBYLA, UOBYQA, NEWUOA, BOBYQA, and LINCOA. The old alias names of `nstep"` and `"absxtol"` can still be used.
Some extensive testing was done with about 60 examples. They are all converging to the correct optimum ansd if there are only boundary constraints specified sometimes with much less function calls than is needed with the older BOBYQA algorithm. At this point there is no reference available I could cite.
Two examples illustrate the usefulness of LINCOA:

1. Example coming with the Fortran code of LINCOA:

   For eplanation Mike Powell writes: "Calculate the tetrahedron of least volume that encloses the points (XP(J),YP(J),ZP(J)), J=1,2,...,NP. Our method requires the origin to be strictly inside the convex hull of these points. There are twelve variables that define the four faces of each

tetrahedron that is considered. Each face has the form ALPHA*X + BETA*Y + GAMMA*Z=1, the variables X(3K-2), X(3K-1) and X(3K) being the values of ALPHA, BETA and GAMMA for the K-th face, K=1,2,3,4. Let the set T contain all points in three dimensions that can be reached from the origin without crossing a face. Because the volume of T may be infinite, the objective function is the smaller of FMAX and the volume of T, where FMAX is set to an upper bound on the final volume initially. There are 4*NP linear constraints on the variables, namely that each of the given points (XP(J),YP(J),ZP(J)) shall be in T. Let XS = min XP(J), YS = min YP(J), ZS = min ZP(J) and SS = max XP(J)+YP(J)+ZP(J), where J runs from 1 to NP. The initial values of the variables are X(1)=1/XS, X(5)=1/YS, X(9)=1/ZS, X(2)=X(3)=X(4)=X(6)=X(7)=X(8)=0 and X(10)=X(11)=X(12)=1/SS, which satisfy the linear constraints, and which provide the bound FMAX= (SS-XS-YS-ZS)**3/6. Other details of the test calculation are given below, including the choice of the data points (XP(J),YP(J),ZP(J)), J=1,2,...,NP. The smaller final value of the objective function in the case NPT=35 shows that the problem has local minima."

```
n = 12; np = 50;
pi = macon("pi");
xp = yp = zp = cons(np);


/*--- Set the data points  ---*/
sx = sy = sz = 0.;
for (i = 1; i <= np; i++) {
  theta = (i - 1.) * pi / (np - 1.);
  xp[i] = cos(theta) * cos(2. * theta);
  sx += xp[i];
  yp[i] = sin(theta) * cos(2. * theta);
  sy += yp[i];
  zp[i] = sin(2. * theta);
  sz += zp[i];
}
sx /= np; sy /= np; sz /= np;
for (i = 1; i <= np; i++) {
  xp[i] -= sx; yp[i] -= sy; zp[i] -= sz;
}
/* print "XYZ=", xyz = xp -> yp -> zp; */


/*--- Set the linear constraints ---*/
nlc = 4 * np;
blc = cons(nlc,1,1.);
alc = cons(nlc,n,0.);
```

```
  k = 1;
  for (j = 1; j <= np; j++)
  for (i = 1; i <= 4; i++, k++) {
    alc[k,3*i-2] = xp[j];
    alc[k,3*i-1] = yp[j];
    alc[k,3*i  ] = zp[j];
  }
  mis = cons(nlc,1,.);
  lc = mis -> alc -> blc; /* print "LC=", lc; */


  /* Set the initial vector of variables. The JCASE=1,6 loop gives six
     different choices of NPT when LINCOA is called. */
  sx = sy = sz = ss = 0.;
  for (i = 1; i <= np; i++) {
    sx = min(sx,xp[i]);
    sy = min(sy,yp[i]);
    sz = min(sz,zp[i]);
    tt = xp[i] + yp[i] + zp[i];
    ss = max(ss,tt);
  }
  fmax = pow(ss - sx - sy - sz,3.) / 6.;
  print "Fmax=", fmax," sx,...=",sx,sy,sz,ss;
```

```
Fmax= 16.521 sx,...=-1.0000-0.7893-0.9995 1.8393
```

The following is the objective function:

```
  function ftetrah(x) global(fmax) {
   n = ncol(x); /* print "X=",x; */
   crit = fmax;
   v12 = x[1]*x[ 5] - x[ 4]*x[2];
   v13 = x[1]*x[ 8] - x[ 7]*x[2];
   v14 = x[1]*x[11] - x[10]*x[2];
   v23 = x[4]*x[ 8] - x[ 7]*x[5];
   v24 = x[4]*x[11] - x[10]*x[5];
   v34 = x[7]*x[11] - x[10]*x[8];

   del1 = v23*x[12] - v24*x[9] + v34*x[6];
   /* print "del1=",del1; */
   if (del1 > 0.) {
     del2 = -v34*x[3] - v13*x[12] + v14*x[9];
     /* print "del2=",del2; */
```

```
    if (del2 > 0.) {
      del3 = -v14*x[6] + v24*x[3] + v12*x[12];
      /* print "del3=",del3; */
      if (del3 > 0.) {
        del4 = -v12*x[9] + v13*x[6] - v23*x[3];
        if (del4 > 0.) {
          t1 = del1 + del2 + del3 + del4;
          t2 = del1 * del2 * del3 * del4;
          tt = pow(t1,3.) / t2;
          crit = min(tt / 6.,fmax);
 } } } }
 return(crit);
}
```

We must specify a starting point for the optimization:

```
x0 = cons(1,n,0.);
x0[1] = 1. / sx; x0[5] = 1. / sy; x0[9] = 1. / sz;
x0[10] = x0[11] = x0[12] = 1. / ss;
crit = ftetrah(x0); print "F(x0)=",crit;
```

```
F(x0)= 16.521
```

```
npt = 15;
mopt = [ "tech"   "lincoa" ,
         "intpoi"      npt ,
         "rhobeg"      1.0 ,
         "rhoend"    1.e-6 ,
         "maxfun"    10000 ,
         "print"         4 ];
< xr,rp > = nlp(ftetrah,x0,mopt,.,lc);
crit = ftetrah(xr); print "F(xr)=",crit;
```

```
                        *****************
                        Optimization Start
                        *****************

                       Parameter Estimates
                       -------------------

                     Parameter      Estimate
```

```
 1 X_1     -1.00000000
 2 X_2               0
 3 X_3               0
 4 X_4               0
 5 X_5     -1.26701554
 6 X_6               0
 7 X_7               0
 8 X_8               0
 9 X_9     -1.00051405
10 X_10     0.54369049
11 X_11     0.54369049
12 X_12     0.54369049

          Value of Objective Function =       16.521


                      Linear Constraints
                      ------------------

[  1]   -1.0000000 <=  -  1.00000 * X_1      -  0.20818 * X_2
                       -   9e-018 * X_3      ( 2.00000 )
[  2]   -1.0000000 <=  -  1.00000 * X_4      -  0.20818 * X_5
                       -   9e-018 * X_6      ( 1.26376 )
[  3]   -1.0000000 <=  -  1.00000 * X_7      -  0.20818 * X_8
                       -   9e-018 * X_9      ( 1.00000 )
[  4]   -1.0000000 <=  -  1.00000 * X_10     -  0.20818 * X_11
                       -   9e-018 * X_12     ( 0.34313 )
[  5]   -1.0000000 <=  -  0.98975 * X_1      -  0.27172 * X_2
                       -  0.12788 * X_3      ( 1.98975 )
.............................................................


[ 197]ACT-1.0000000 <=  +  1.00000 * X_1      -  0.20818 * X_2
                       +   2e-016 * X_3      (       0 )
[ 198]   -1.0000000 <=  +  1.00000 * X_4      -  0.20818 * X_5
                       +   2e-016 * X_6      ( 1.26376 )
[ 199]   -1.0000000 <=  +  1.00000 * X_7      -  0.20818 * X_8
                       +   2e-016 * X_9      ( 1.00000 )
[ 200]   -1.0000000 <=  +  1.00000 * X_10     -  0.20818 * X_11
                       +   2e-016 * X_12     ( 1.43051 )


          LINCOA Algorithm by M.J.D. Powell (2013)
               *** Termination Criteria ***
  Minimum Iterations  . . . . . .          0
```

```
Maximum Iterations  . . . . . .        1000
Maximum Function Calls. . . . . .      10000
ABSFCONV Function Criterion . . . . . .    0
FCONV Function Criterion  . . . . . .  2.22e-016
FCONV2 Function Criterion . . . . . .    1e-006
FSIZE Parameter . . . . . .              0
ABSXCONV Parameter Change Criterion . . . . . .    1e-006
XCONV Parameter Change Criterion  . . . . . .      1e-008
XSIZE Parameter . . . . . .              0
ABSCONV Function Criterion  . . . . . . -1.34e+154
             *** Other Control Parameters ***
Version of Algorithm. . . . . .          0
Initial Simplex Size (INSTEP) . . . . . .          1
Final Simplex Size (ABSXTOL). . . . . .      1e-006
Number Interpolation Points . . . . . .          15
```

We print an iteration history whenever the size `RHO` of the trust region is reduced:

```
          LINCOA Algorithm by M.J.D. Powell (2013)

      Iter      nfun     optcrit     difcrit         rho
         1        17  12.7252293  3.79574440  0.10000000
         2        69  2.79438753  9.93084172  0.01000000
         3        82  2.78016283  0.01422471  0.00100000
         4       119  2.76133365  0.01882918  1.000e-004
         5       128  2.76131424  1.940e-005  1.000e-005
         6       137  2.76131265  1.590e-006  1.000e-006
         7       144  2.76131253  1.233e-007  1.000e-006

 Successful Termination After     7 Iterations
 Criterion       2.761312531        Max Grad Entry            0
 Max Const Viol. 2.2204e-016        N. Active Constraints    11
 N. Function Calls       145        Preproces. Time           0
 Time for Method           0        Effective Time            0


                    ********************
                    Optimization Results
                    ********************

                    Parameter Estimates
                    -------------------

                  Parameter      Estimate
```

```
 1 X_1     -1.23379112
 2 X_2     -1.23134771
 3 X_3     -0.88706165
 4 X_4      1.12988816
 5 X_5     -1.12828507
 6 X_6      1.08403365
 7 X_7      0.84313073
 8 X_8      0.75354025
 9 X_9     -0.84617569
10 X_10    -0.65858476
11 X_11     1.55680525
12 X_12     0.58534120

Value of Objective Function =      2.76131
```

The solution is a feasible point and the constraints 3, 14, 18, 44, 86, 105, 151, 189, 192, 193,and 196 are active at the solution:

```
             Linear Constraints Evaluated at Solution
             ----------------------------------------

 [  1]      -1.00000 * X_1      -  0.20818 * X_2
          -   9e-018 * X_3      +  1.00000 =  2.490128612
 [  2]      -1.00000 * X_4      -  0.20818 * X_5
          -   9e-018 * X_6      +  1.00000 =  0.104994121
 [  3]ACT   -1.00000 * X_7      -  0.20818 * X_8
          -   9e-018 * X_9      +  1.00000 =  7.9028e-018
 [  4]      -1.00000 * X_10     -  0.20818 * X_11
          -   9e-018 * X_12     +  1.00000 =  1.334494707
 [  5]      -0.98975 * X_1      -  0.27172 * X_2
          -  0.12788 * X_3      +  1.00000 =  2.669164996
.....................................................


 [ 196]ACT    0.98975 * X_10     -  0.27172 * X_11
           +  0.12788 * X_12     +  1.00000 = -9.7145e-017
 [ 197]       1.00000 * X_1      -  0.20818 * X_2
           +   2e-016 * X_3      +  1.00000 =  0.022546363
 [ 198]       1.00000 * X_4      -  0.20818 * X_5
           +   2e-016 * X_6      +  1.00000 =  2.364770444
 [ 199]       1.00000 * X_7      -  0.20818 * X_8
           +   2e-016 * X_9      +  1.00000 =  1.686261463
 [ 200]       1.00000 * X_10     -  0.20818 * X_11
           +   2e-016 * X_12     +  1.00000 =  0.017325184
```

2. Example coming with the Fortran code of BOBYQA:

   Since boundary constraints are special linear constraints, all examples which are run with BOBYQA can also be run by LINCOA.

```
print "\n *** Compare M. Powell's BOBYQA with LINCOA: ***\n";

function fsrecip(x) {
 n = ncol(x);
 crit = 0.;
 for (i = 4; i <= n; i+=2)
 for (j = 2; j <= i-2; j+=2) {
   t1 = x[i-1] - x[j-1]; t2 = x[i] - x[j];
   tt = t1 * t1 + t2 * t2;
   if (tt < 1.e-6) tt = 1.e-6;
   crit += 1. / sqrt(tt);
 }
 return(crit);
}


m = 5; n = 2 * m;
pi2 = 2. * macon("pi");
bc = cons(n,1,-1.) -> cons(n,1,1.);
x0 = cons(1,n,.);
for (j = k = 1; j <= m; j++, k+=2) {
 xin = (pi2 / (real)m) * j;
 x0[k] = cos(xin); x0[k+1] = sin(xin);
}
crit = fsrecip(x0); print "F(x0)=",crit;


F(x0)= 6.8819


npt = 2*n + 1;
mopt = [ "tech"   "lincoa" ,
         "intpoi"      npt ,
         "rhobeg"      1.0 ,
         "rhoend"    1.e-6 ,
         "maxfun"    10000 ,
         "print"         4 ];
< xr,rp > = nlp(fsrecip,x0,mopt,bc);
```

```
                  ******************
                  Optimization Start
                  ******************

                 Parameter Estimates
                 -------------------

   Parameter      Estimate   Lower BC   Upper BC

     1 X_1      0.30901699 -1.0000000  1.0000000
     2 X_2      0.95105652 -1.0000000  1.0000000
     3 X_3     -0.80901699 -1.0000000  1.0000000
     4 X_4      0.58778525 -1.0000000  1.0000000
     5 X_5     -0.80901699 -1.0000000  1.0000000
     6 X_6     -0.58778525 -1.0000000  1.0000000
     7 X_7      0.30901699 -1.0000000  1.0000000
     8 X_8     -0.95105652 -1.0000000  1.0000000
     9 X_9      1.00000000 -1.0000000  1.0000000
    10 X_10    -2.449e-016 -1.0000000  1.0000000

         Value of Objective Function =      6.88191


          LINCOA Algorithm by M.J.D. Powell (2013)
               *** Termination Criteria ***
Minimum Iterations  . . . . . .          0
Maximum Iterations  . . . . . .       1000
Maximum Function Calls. . . . . .     10000
ABSFCONV Function Criterion . . . . . .          0
FCONV Function Criterion  . . . . . .  2.22e-016
FCONV2 Function Criterion . . . . . .     1e-006
FSIZE Parameter . . . . . .          0
ABSXCONV Parameter Change Criterion . . . . . .     1e-006
XCONV Parameter Change Criterion  . . . . . .     1e-008
XSIZE Parameter . . . . . .          0
ABSCONV Function Criterion  . . . . . . -1.34e+154
              *** Other Control Parameters ***
Version of Algorithm. . . . . .          0
Initial Simplex Size (INSTEP) . . . . . .          1
Final Simplex Size (ABSXTOL). . . . . .     1e-006
Number Interpolation Points . . . . . .         21


          LINCOA Algorithm by M.J.D. Powell (2013)
```

```
      Iter      nfun     optcrit     difcrit         rho
         1        23  5.69374981  1.18815979  0.10000000
         2        40  5.60154297  0.09220684  0.01000000
         3        46  5.60154297           0  0.00100000
         4        54  5.60153418  8.785e-006  1.000e-004
         5        60  5.60153397  2.112e-007  1.000e-005
         6        64  5.60153397           0  1.000e-006
         7        69  5.60153397           0  1.000e-006

Successful Termination After     7 Iterations
Criterion       5.601533972        Max Grad Entry           0
Max Const Viol.           0        N. Active Constraints    9
N. Function Calls        70        N. Gradient Calls        1
Preproces. Time           0        Time for Method          0
Effective Time            0


                    ********************
                    Optimization Results
                    ********************

                    Parameter Estimates
                    -------------------

          Parameter      Estimate   Active BC

            1 X_1      1.00000000  Upper BC
            2 X_2      1.00000000  Upper BC
            3 X_3     -1.00000000  Lower BC
            4 X_4      1.00000000  Upper BC
            5 X_5     -1.00000000  Lower BC
            6 X_6     -1.00000000  Lower BC
            7 X_7      1.00000000  Upper BC
            8 X_8     -1.00000000  Lower BC
            9 X_9      1.00000000  Upper BC
           10 X_10    -1.236e-009

       Value of Objective Function =      5.60153
```

The same example can be run with the `BOBYQA` algorithm:

```
mopt = [ "tech"   "bobyqa" ,
         "intpoi"      npt ,
         "rhobeg"       .1 ,
         "rhoend"    1.e-6 ,
```

```
        "maxfun"     5000 ,
        "print"        4 ];
< xr,rp > = nlp(fsrecip,x0,mopt,bc);


                     ******************
                     Optimization Start
                     ******************

                     Parameter Estimates
                     -------------------

   Parameter      Estimate   Lower BC   Upper BC

     1 X_1      0.30901699 -1.0000000  1.0000000
     2 X_2      0.95105652 -1.0000000  1.0000000
     3 X_3     -0.80901699 -1.0000000  1.0000000
     4 X_4      0.58778525 -1.0000000  1.0000000
     5 X_5     -0.80901699 -1.0000000  1.0000000
     6 X_6     -0.58778525 -1.0000000  1.0000000
     7 X_7      0.30901699 -1.0000000  1.0000000
     8 X_8     -0.95105652 -1.0000000  1.0000000
     9 X_9      1.00000000 -1.0000000  1.0000000
    10 X_10    -2.449e-016 -1.0000000  1.0000000

         Value of Objective Function =      6.88191


          BOBYQA Algorithm by M.J.D. Powell (2008)
               *** Termination Criteria ***
Minimum Iterations  . . . . . .          0
Maximum Iterations  . . . . . .       1000
Maximum Function Calls. . . . . .       5000
ABSFCONV Function Criterion . . . . . .          0
FCONV Function Criterion  . . . . . .  2.22e-016
FCONV2 Function Criterion . . . . . .     1e-006
FSIZE Parameter . . . . . .          0
ABSXCONV Parameter Change Criterion . . . . . .     1e-006
XCONV Parameter Change Criterion  . . . . . .     1e-008
XSIZE Parameter . . . . . .          0
ABSCONV Function Criterion  . . . . . . -1.34e+154
             *** Other Control Parameters ***
Version of Algorithm. . . . . .          0
Initial Simplex Size (INSTEP) . . . . . .        0.1
Final Simplex Size (ABSXTOL). . . . . .     1e-006
Number Interpolation Points . . . . . .         21
```

```
        BOBYQA Algorithm by M.J.D. Powell (2008)


      Iter      nfun     optcrit     difcrit         rho
         1        44  5.60889786  1.27301174  0.01000000
         2        59  5.60156025  0.00733762  0.00100000
         3        73  5.60153398  2.627e-005  1.000e-004
         4        79  5.60153397  6.357e-009  1.000e-005
         5        94  5.60153397           0  1.000e-006
         6       106  5.60153397  1.902e-011  1.000e-006

Successful Termination After     6 Iterations
Criterion       5.601533972        Max Grad Entry  1.338420835
Max Const Viol.           0        Max Grad LagF.  1.3362e-007
N. Active Constraints     9
N. Function Calls       107        N. Gradient Calls         1
Preproces. Time           0        Time for Method           0
Effective Time            0


                   ********************
                   Optimization Results
                   ********************

                   Parameter Estimates
                   -------------------

          Parameter      Estimate   Active BC

            1 X_1      1.00000000  Upper BC
            2 X_2      1.00000000  Upper BC
            3 X_3     -1.00000000  Lower BC
            4 X_4      1.00000000  Upper BC
            5 X_5     -1.00000000  Lower BC
            6 X_6     -1.00000000  Lower BC
            7 X_7      1.00000000  Upper BC
            8 X_8     -1.00000000  Lower BC
            9 X_9      1.00000000  Upper BC
           10 X_10     3.401e-008

        Value of Objective Function =      5.60153
```

5 New Developments

5.1 The `boundbox()` Function

< vol,box > = boundbox(xy)

Purpose: The function `boundbox` finds the coordinates and the volume of the smallest rectangular box surrounding a set of n points X_{ij}, $i = 1, \ldots, n, j = 1, 2$ in 2-dimensional space.

Input: The only input is an $n \times 2$ matrix of the (x, y) coordinates of n points.

Output: **vol** a scalar which is the area of the box

box a 4×2 matrix of the four corner points of the recangular box

Restrictions: 1. The first input argument must not contain any missing or complex values or string data.

2. The first input argument must have two columns.

Relationships: maxepmty()

Examples: 1. Small example of $n = 100$ points:

```
xy = x‘-> y‘;
< vol,box > = boundbox(xy);
print "Vol=",vol;
print "Box=",box;
```

```
Vol= 1.4537

Box=
           |      Xdim      Ydim
-------------------------------
   UppLeft |   0.49421    1.3314
  UppRight |    1.3987   0.46976
   LowLeft |  -0.30839   0.48881
  LowRight |   0.59616  -0.37281
```

5.2 The `hamilton()` Function

< gof,circ > = hamilton(adja<,optn>)

< gof,circ > = hamilton(indx,vert<,optn>)

Purpose: Assuming the graph has n edges with path connections (a_{ij} where a_{ij} is 1 if there is a path from edge i to edge j and otherwise $a_{ij} = 0$. For given directed and undirected graph data this function tries to find all or some of the Hamiltonian circuits (if one exists at all). Note, that for very large graph data the complete set of circuits may not be found since the computational problem is NP hard.

Input: There are two forms of input data specifying the graph:

binary $n \times n$ matrix of $(0, 1)$ adjacency data $\mathbf{A} = (a_{ij})$, where a_{ij} is 1 if there is a path from edge i to edge j and otherwise $a_{ij} = 0$.

- sparse `(indx,vert)` input form: the $n+1$ vector `indx` gives the index bounds for the origin and end of the path (defining the i), and the vector `vert` lists the indices of the target edge (defining j). Note, there must be `indx[1]=0` and `indx[n+1] = m`, and the entries must be monoton increasing integers.

a numeric vector of options:

[**1**.] amount of printed output, default=0

[**2**] method (algorithm), default=0

[**3**] maximum number of Hamiltonian cicuits to be found, if ¡ 0: find all possible, default=-1

[**4**] an integer upper bound for the number of backtrackings, if ¡ 0: find exact solution, default=-1

Output: **gof** lists some scalar results:

1. error return code
2. the number of Hamiltonian cicuits found
3. the number of backtrackings
4. computing time in seconds

circ a matrix or row vector containing one Hamiltonian circuits in each row (if there exists any).

Restrictions: 1. Since the problem is NP-hard, the algorithms are not always able to find all circuits, especially for large graphs.

2. All the input data is integer. String, complex, and missing values are not permitted.

Relationships: tsp(), vrp(), mstgra()

Examples: 1. Example of TOMS 595, $n = 6$:

The following two graph specifications are almost equivalent, except for the order of the target vertices:

```
indx = [ 0    3    5    8   11   13   16 ];
vert = [ 3    5    6    6    3    6    1    4    5    1
         2    2    3    4    2    5 ];


adja2 = [ 0 0 1 0 1 1 ,  /* 3 5 6 */
          0 0 1 0 0 1 ,  /* 6 3  */
          1 0 0 1 0 1 ,  /* 6 1 4 */
          1 1 0 0 1 0 ,  /* 5 1 2 */
          0 1 1 0 0 0 ,  /* 2 3 */
          0 1 0 1 1 0 ]; /* 4 2 5 */
```

Find all Hamiltonian circuits:

```
print "EXACT PROCEDURE TO FIND (AND PRINT) ALL THE HAMILTONIAN CIRCUITS.";
optn = [ 2,    /* ipri */
         0,    /* imet */
        -1,    /* maxc: max number circuits */
        -1 ];  /* nbck: number backtracking */
< gof,circ > = hamilton(indx,vert,optn);
```

```
                Number Hamiltonian Circuits: 3
                  Number of Backtrackings: 7
                 Total Processing Time: 0.006


                      Hamiltonian Cicuits
                      *******************

                     Dense Matrix (3 by 6)

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

We obtain the same result when we run the second data specification.

```
< gof,circ > = hamilton(adja2,optn);
```

```
print "HEURISTIC PROCEDURE TO FIND (AND PRINT) A SINGLE HAMILTONIAN CIRCUIT";
print "IF ONE EXISTS, WITHOUT PERFORMING MORE THAN 4 BACKTRACKINGS.";
optn = [ 2,    /* ipri */
         0,    /* imet */
         1,    /* maxc: max number circuits */
         4 ];  /* nbck: number backtracking */
< gof,circ > = hamilton(indx,vert,optn);


               Number Hamiltonian Circuits: 1
                 Number of Backtrackings: 3
                Total Processing Time: 0.005


                    Hamiltonian Cicuits
                    *******************

                  Dense Row Vector (ncol=6)

          R |       1      2      3      4      5      6
                    2      3      6      4      1      5
```

Obviously due to the slightly other order of vertices, this data specifications finds another cicuit:

```
< gof,circ > = hamilton(adja2,optn);


               Number Hamiltonian Circuits: 1
                 Number of Backtrackings: 0
                Total Processing Time: 0.005


                    Hamiltonian Cicuits
                    *******************

                  Dense Row Vector (ncol=6)

          R |       1      2      3      4      5      6
                    2      3      1      6      4      5
```

2. Example with $n = 8$:

```
adja = [  0 1 0 1 0 1 0 0 ,
          1 0 1 0 0 0 1 0 ,
          0 1 0 1 0 0 0 1 ,
          1 0 1 0 1 0 0 0 ,
```

```
          0 0 0 1 0 1 0 1 ,
          1 0 0 0 1 0 1 0 ,
          0 1 0 0 0 1 0 1 ,
          0 0 1 0 1 0 1 0 ];

indx2 = [ 0 3 6 9 12 15 18 21 24 ];
vert2 = [ 2 4 6  1 3 7  2 4 8  1 3 5
          4 6 8  1 5 7  2 6 8  3 5 7 ];

print "EXACT PROCEDURE TO FIND (AND PRINT) ALL THE HAMILTONIAN CIRCUITS.";
optn = [ 2,    /* ipri */
         0,    /* imet */
        -1,    /* maxc: max number circuits */
        -1 ];  /* nbck: number backtracking */
< gof, circ > = hamilton(adja,optn);

               Number Hamiltonian Circuits: 12
                 Number of Backtrackings: 21
                Total Processing Time: 0.006

                     Hamiltonian Cicuits
                     *******************

                    Dense Matrix (12 by 8)

      |       1      2      3      4      5      6      7      8
    ----------------------------------------------------------------
    1 |       1      2      3      4      5      8      7      6
    2 |       1      2      3      8      7      6      5      4
    3 |       1      2      7      6      5      8      3      4
    4 |       1      2      7      8      3      4      5      6
    5 |       1      4      3      2      7      8      5      6
    6 |       1      4      3      8      5      6      7      2
    7 |       1      4      5      6      7      8      3      2
    8 |       1      4      5      8      3      2      7      6
    9 |       1      6      5      4      3      8      7      2
   10 |       1      6      5      8      7      2      3      4
   11 |       1      6      7      2      3      8      5      4
   12 |       1      6      7      8      5      4      3      2
```

We obtain the same result when we run the second data specification.

```
< gof,circ > = hamilton(indx2,vert2,optn);
```

3. Example with $n = 8$:

```
adja = [  0 1 0 0 1 1 0 0 ,
          1 0 1 0 0 0 0 1 ,
          0 1 0 1 1 0 0 0 ,
          0 0 1 0 1 0 0 0 ,
          1 0 1 1 0 0 0 0 ,
          1 0 0 0 0 0 1 1 ,
          0 0 0 0 0 1 0 1 ,
          0 1 0 0 0 1 1 0 ];


indx2 = [ 0 3 6 9 11 14 17 19 22 ];
vert2 = [ 2 5 6  1 3 8  2 4 5  3 5
          1 3 4  1 7 8  6 8  2 6 7 ];


print "EXACT PROCEDURE TO FIND (AND PRINT) ALL THE HAMILTONIAN CIRCUITS.";
optn = [ 2,    /* ipri */
         0,    /* imet */
        -1,    /* maxc: max number circuits */
        -1 ];  /* nbck: number backtracking */
< gof, circ > = hamilton(adja,optn);
print "GOF=",gof;
print "CIRC=",circ;
```

```
               Number Hamiltonian Circuits: 2
                 Number of Backtrackings: 7
                Total Processing Time: 0.006


                     Hamiltonian Cicuits
                     *******************

                    Dense Matrix (2 by 8)

     |       1      2      3      4      5      6      7      8
   ----------------------------------------------------------------
   1 |       1      5      4      3      2      8      7      6
   2 |       1      6      7      8      2      3      4      5
```

We obtain the same result when we run the second data specification.

```
< gof,circ > = hamilton(indx2,vert2,optn);
```

5.3 The `knapsack()` Function

< gof,xind > = knapsack(prof,wgt,cap<,optn>)

Purpose: For given n vectors of positive integer profits and weights and $k >= 1$ positive integer capacities, function `knapsack()` solves either the one-dimensional ($k = 1$),

$$\max \sum_i^n p_i * x_i \quad s.t. \quad \sum_i^n w_i * x_i \leq cap,$$

where $x_i \in 0, 1, i = 1, \ldots, n$, or the multidimensional ($k > 1$) knapsack problem,

$$\max \sum_i^n (p_i * \sum_j^k x_{ji}) \quad s.t. \quad \sum_i^n w_i * x_{ij} \leq cap_j, \quad j = 1, \ldots, k,$$

where $x_{ij} \in \{0, 1\}, i = 1, \ldots, n, j = 1, \ldots, k$.

Two different algorithms are implemented:

1. only for the one dimensional case: full enumeration takes an $n \times (cap + 1)$ matrix for work space.
2. an algorithm which can be used for both, the one- and multi dimensional case which is using backtracking for improving the result. For the onedimensional application that is using less memory but more computer time than the specific onedimensional algorithm.

The algorithms have been reprogrammed after code found in the R package `adagio` by Hans Werner Borchers.

Input: **prof** n vector of positive integers for profits

wgt n vector of positive integers for weights

cap positive int scalar or k vector of positive scalars for capacities

optn a numeric vector of options:

- **[1**] amount of printed output
- **[2**] an integer upper bound for the number of backtrackings,
- **[3**] set different from zero if the second algorithm must be applied to the onedimensional application.

Output: **gof** lists some scalar results:

1. error return code
2. the value v^* of the objective function:
   - one-dimensional case: $\sum_i^n p_i * x_i$

- multi-dimensional case: $\sum_i^n (p_i * \sum_j^k x_{ji})$

3. computing time

xind 1. one-dimensional case: a binary n vector

2. multi-dimensional case: the values of j indicating the $x_{ij} = 1$

Restrictions: 1. Both n vectors of profits and weights must contain positive integers and capacity scalar or k vector must contain positive intgeres not smaller than the weights.

2. The first three input arguments must not contain any complex or missing values or strings.

Relationships: lp(), lptransp(), lpassign()

Examples: 1. One-dimensional Example:

```
print "xind = [ 1 2 3 4 6] and prof=280";
prof = [ 15 100 90 60 40 15 10  1 ];
wgts = [  2  20 20 30 40 30 60 10 ];
< gof, xind > = knapsack(prof,wgts,102);
print "Gof=",gof;
print "Xind=",xind;
```

```
Crit= 280

Xind=
   |   1
--------
 1 |   1
 2 |   1
 3 |   1
 4 |   1
 5 |   0
 6 |   1
 7 |   0
 8 |   0
```

2. Two-dimensional Example:

```
print "xind = [ 2 6 ], [1 4]  and prof=345";
prof = [ 110 150 70 80 30 5 ];
wgts = [  40  60 30 40 20 5 ];
cap = [ 65 85 ];
< gof, xind > = knapsack(prof,wgts,cap);
print "Gof=",gof;
print "Xind=",xind;
```

```
Crit= 345

Xind=
   |   1
--------
 1 |   2
 2 |   1
 3 |   0
 4 |   2
 5 |   0
 6 |   1
```

5.4 The `latlong()` Function

res = latlong(task,pnt1<,pnt2<,optn>>)

Purpose: For one or two data points in (latidude,longitude) triple format the `latlong()` function computes:

task	description	points
"l2dd"	convert from (deg,min,sec) format into decimal degree	pnt1
"dd2l"	convert from decimal degree into (deg,min,sec) format	pnt1
"dist"	compute distance between point 1 and point 2	pnt1, pnt2
"bear"	compute initial bearing between point 1 and point 2	pnt1, pnt2
"p1be"	compute point in specified distance	pnt1, (bear,dist)
"midp"	compute midpoint between point 1 and point 2	pnt1, pnt2
"intp"	compute n equidistant points between point 1 and 2	pnt1, pnt2

Both, latitude and longitude are angles and can be expressed either in tripel (deg,min,sec) or decimal degree (DD) form. The latitude with angle 0 describes the equator, positive values represent angles toward the North pole (N) and negative values represent angles toward the South pole (S). Locations with the same value of Latitude are circles parallel to the equator. The longitude with angle 0 crosses Greenwich near London, positive values represent angles toward East (E) and negative values represent angles toward West (W). All circles with the same value of longitude have the same radius, not so for the Latitudes which are becoming smaller as larger the angle.

In the US Latitudes W(est) are referred to with positive numbers. Here positive Latitude degrees always refer to E(ast).

Input: **task** must be a single 4-char string specifying the task, see table above for all valid choices

pnt1 must be either a nc1=2 or nc1=6 vector of (latitude, longitude) decimal degree or triple (deg,min,sec) specification or a $nr1 \times nc1$ matrix with two or six columns

pnt2 "l2dd","dd2l" must be missing value,

"p1be" must be either a nc2=2 or nc2=4 vector of decimal degree or triple (deg,min,sec) specification of bearing (course angle) and distance (in km, US Miles, or nautical miles, depending on option setting) or a $nr2 \times nc2$ matrix with two or four columns

otherwise must be either a nc2=2 or nc2=6 vector of (latitude, longitude) decimal degree or triple (deg,min,sec) specification or a $nr2 \times nc2$ matrix with two or six columns

optn must be a scalar or vector specifying some options:

1. only for task= "intp": the number M of equidistant points between the specified points 1 and 2 (for uneven M it also contains the midpoint)
2. only for task="dist": =0: return distance in km (default), =1: in US Miles, =2: in nautical Miles

Output: only one object is returned:

- for task="l2dd": conversion from tripel to decimal degree form:
  1. if $N = max(nr1, nr2) = 1$: for one input point specified as the triple(deg,min,sec) returns a real scalar with the corresponding decimal degrees
  2. if $N = max(nr1, nr2) > 1$: for N input points specified as the triple(deg,min,sec) returns a N vector with the the corresponding decimal degrees
- for task="dd2l": conversion from decimal degree to tripel form:
  1. if $N = max(nr1, nr2) = 1$: for one input point specified as a real scalar in decimal degrees returns the 3 vector with the corresponding (deg,min,sec) triple form
  2. if $N = max(nr1, nr2) > 1$: for N input points specified as vector of N real values in decimal degrees returns the $N \times 3$ matrix in corresponding (deg,min,sec) triple form
- for task="dist": compute the distance (see `optn[2]` between all pairs of points:
- if $N = max(nr1, nr2) = 1$: for one pair of input points returns a real scalar with the distance between the two points
- if $N = max(nr1, nr2) > 1$: for N pairs of input points returns a N vector with the N distances between all N pairs of points
- for task="bear": compute the initial bearing:
  1. if $N = max(nr1, nr2) = 1$: for one pair of input points returns a 3-vector of triple form of the bearing between the two points
  2. if $N = max(nr1, nr2) > 1$: for N pairs of input points returns a $N \times 3$ matrix with the bearings between all N pairs of points
- for task="p1be": for speified starting point, bearing and distance compute the (lat,long) angles of point at the end:
  1. if $N = max(nr1, nr2) = 1$: for one pair of input points returns a 6-vector of the triple forms of (latitude,longitude) of the coordinates of the end points
  2. if $N = max(nr1, nr2) > 1$: for N pairs of input points returns a $N \times 6$ matrix of the triple forms of (latitude,longitude) of the coordinates of all N end points
- for task="midp": compute the midpoint (half-way point) between all pairs of points:

1. if $N = max(nr1, nr2) = 1$: for one pair of input points returns a 6-vector of the triple forms of (latitude,longitude) of the midpoint coordinates between the two points
2. if $N = max(nr1, nr2) > 1$: for N pairs of input points returns a $N \times 6$ matrix of the triple forms of (latitude,longitude) of the midpoint coordinates between all N pairs of points

- for task="intp": computes M (see `optn[1]`) equidistant intermediate points on a path between all pairs of points:
  1. if $N = max(nr1, nr2) = 1$: for one pair of input points returns a $M \times 6$ matrix of the triple forms of (latitude,longitude) of the coordinates of intermediate points between the two points
  2. if $N = max(nr1, nr2) > 1$: for N pairs of input points returns a $N \times M \times 6$ tensor of the triple forms of (latitude,longitude) of the coordinates of intermediate points between all N pairs of points

Restrictions: 1. A missing value is returned if the input data contain any string or complex data.

2. The input data points must be (latidude,longitude) format i.e. each with valid (deg,min,sec) tripels.

Relationships: deg2rad(), rad2deg(), convert()

Examples: 1. Convert Location of the United States Capital:
Convert from decimal to tripel form:

```
print "Location of the United States Capital";
dlat1 =  38.889722; dlng1 = -77.008889;
print lat1 = latlong("dd2l",dlat1);
print lng1 = latlong("dd2l",dlng1);
```

```
   |   Degrees   Minutes   Seconds
-----------------------------------
 1 |    38.000    53.000    22.999
```

```
   |   Degrees   Minutes   Seconds
-----------------------------------
 1 |   -77.000   0.00000    32.000
```

Convert from tripel to decimal form:

```
lat2 = [ 38 53 23 ]; lng2 = [ -77 00 32 ];
print dlat2 = latlong("l2dd",lat2);
print dlng2 = latlong("l2dd",lng2);
```

```
 38.890
-77.009
```

2. Compute distance, bearing, midpoint and intermediate points:

```
p1 = [ 50 03 59  -5 42 53 ];
p2 = [ 58 38 38  -3 04 12 ];


print "Distance in km=", dist = latlong("dist",p1,p2);
opt = [ . 1 ];
print "Distance in US Miles=", dist = latlong("dist",p1,p2,opt);
opt = [ . 2 ];
print "Distance in Nautical Miles=", dist = latlong("dist",p1,p2,opt);
```

```
Distance in km= 968.854
Distance in US Miles= 602.018
Distance in Nautical Miles= 523.139
```

```
print "Bearing =", bear = latlong("bear",p1,p2);
```

```
Bearing =
   |   Degrees   Minutes   Seconds
---------------------------------
 1 |    9.0000    7.0000    11.345
```

```
print "Midpoint=", midp = latlong("midp",p1,p2);
```

```
   |    LatDeg    LatMin    LatSec   LongDeg   LongMin   LongSec
-------------------------------------------------------------------
 1 |    50.000    19.000    24.240   -45.000    18.000    12.758
```

```
print "Intpoint=", midp = latlong("intp",p1,p2,9);
```

Here point 5 is the mid point:

```
Intpoint=
   |    LatDeg    LatMin    LatSec   LongDeg   LongMin   LongSec
-------------------------------------------------------------------
 1 |    51.000    11.000    54.262   -13.000    24.000    38.932
 2 |    51.000    48.000    14.690   -21.000    24.000    6.6827
 3 |    51.000    51.000    27.856   -29.000    30.000    36.880
```

```
4 |    51.000    21.000    25.299   -37.000    32.000    19.428
5 |    50.000    19.000    24.240   -45.000    18.000    12.758
6 |    48.000    47.000    52.439   -52.000    39.000    44.321
7 |    46.000    50.000    2.6083   -59.000    31.000    38.926
8 |    44.000    29.000    25.530   -65.000    51.000    52.078
9 |    41.000    49.000    28.709   -71.000    40.000    48.729
```

For multiple point input we obtain tensor:

```
p1 = [ 50 03 59   -5 42 53 ];
p2 = [ 58 38 38   -3 04 12 ,
       38 53 23  -77 00 32 ];
```

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

                         _1: Pt1_Pt2[1]
                         **************

                       Dense Matrix (10 by 6)

         |      LatDeg     LatMin     LatSec    LongDeg    LongMin
       ------------------------------------------------------------
IPNT_01 |   50.000000  50.000000  53.912886 -5.0000000  30.000000
IPNT_02 |   51.000000  37.000000  47.567973 -5.0000000  18.000000
IPNT_03 |   52.000000  24.000000  39.884888 -5.0000000  5.0000000
IPNT_04 |   53.000000  11.000000  30.776536 -4.0000000  52.000000
IPNT_05 |   53.000000  58.000000  20.148372 -4.0000000  38.000000
IPNT_06 |   54.000000  45.000000  7.8975864 -4.0000000  24.000000
IPNT_07 |   55.000000  31.000000  53.912186 -4.0000000  9.0000000
IPNT_08 |   56.000000  18.000000  38.069941 -3.0000000  54.000000
IPNT_09 |   57.000000  5.0000000  20.237191 -3.0000000  38.000000
IPNT_10 |   57.000000  52.000000  0.2674813 -3.0000000  21.000000

         |     LongSec
       ---------------
IPNT_01 |   57.169296
IPNT_02 |   36.936317
IPNT_03 |   50.753790
IPNT_04 |   36.946327
IPNT_05 |   53.696868
IPNT_06 |   39.031391
IPNT_07 |   50.801622
```

```
IPNT_08 |   26.665445
IPNT_09 |   24.064639
IPNT_10 |   40.199525
```

```
                        _2: Pt1_Pt2[2]
                        **************

                     Dense Matrix (10 by 6)

         |      LatDeg     LatMin     LatSec    LongDeg    LongMin
       ------------------------------------------------------------
IPNT_01 |   51.000000  7.0000000  0.1014184 -12.000000  41.000000
IPNT_02 |   51.000000  44.000000  4.4053303 -19.000000  56.000000
IPNT_03 |   51.000000  53.000000  54.352469 -27.000000  17.000000
IPNT_04 |   51.000000  36.000000  8.5485907 -34.000000  38.000000
IPNT_05 |   50.000000  51.000000  25.388122 -41.000000  49.000000
IPNT_06 |   49.000000  41.000000  16.620129 -48.000000  42.000000
IPNT_07 |   48.000000  7.0000000  53.225673 -55.000000  13.000000
IPNT_08 |   46.000000  13.000000  48.250963 -61.000000  18.000000
IPNT_09 |   44.000000  1.0000000  41.048440 -66.000000  57.000000
IPNT_10 |   41.000000  34.000000  5.5289763 -72.000000  11.000000

         |     LongSec
       ---------------
IPNT_01 |   47.807042
IPNT_02 |   4.4219888
IPNT_03 |   54.452608
IPNT_04 |   26.816315
IPNT_05 |   1.4275687
IPNT_06 |   19.272108
IPNT_07 |   7.3767720
IPNT_08 |   30.510076
IPNT_09 |   35.725106
IPNT_10 |   2.7318257
```

3. Airports: Los Angeles LAX, New York JFK (Ed Williams):

```
lax = [ 33 57 00   118 24 00 ];
jfk = [ 40 38 00    73 47 00 ];
opt = [ . 2 ];
print "Distance in Nautical Miles=", dist = latlong("dist",lax,jfk,opt);
print "Should be 2144 nm";
```

```
Distance in Nautical Miles= 2145.17
```

```
   print "Bearing =", bear = latlong("bear",lax,jfk);
   print "Should be 66 degrees";

Bearing =
   |   Degrees   Minutes   Seconds
---------------------------------
 1 |    65.000    53.000    31.800

   print "Go  100 nautical miles in 66.0 degree radial from LAX in direction JFK";
   bed = bear -> 100 ;
   opt = [ . 2 ];
   print "point =", pt = latlong("p1be",lax,bed,opt);

point =
   |    LatDeg    LatMin    LatSec   LongDeg   LongMin   LongSec
-------------------------------------------------------------------
 1 |    34.000    50.000    36.269    118.00    14.000    6.2701

   print "Intpoint=", midp = latlong("intp",lax,jfk,9);
   print "Midpoint is point 5:";

Intpoint=
   |    LatDeg    LatMin    LatSec   LongDeg   LongMin   LongSec
-------------------------------------------------------------------
 1 |    35.000    20.000    42.040    114.00    24.000    4.2136
 2 |    36.000    36.000    13.949    110.00    16.000    7.9478
 3 |    37.000    42.000    55.993    106.00   0.00000    27.736
 4 |    38.000    40.000    10.012    101.00    37.000    34.177
 5 |    39.000    27.000    20.706    97.000    8.0000    12.869
 6 |    40.000    3.0000    57.035    92.000    33.000    24.428
 7 |    40.000    29.000    33.609    87.000    54.000    23.358
 8 |    40.000    43.000    51.945    83.000    12.000    35.672
 9 |    40.000    46.000    41.447    78.000    29.000    35.365

   print "Midpoint=", midp = latlong("midp",lax,jfk);

Midpoint=
   |    LatDeg    LatMin    LatSec   LongDeg   LongMin   LongSec
-------------------------------------------------------------------
 1 |    39.000    27.000    20.706    97.000    8.0000    12.869
```

5.5 The `locat1()` Function

¡ gof,post ¿ = locat1(vwgt,fwgt<,optn>)

Purpose: The `locat1()` function solves the multifacility location problem with rectangular distance by the minimum cut approach. Assuming there are nf fixed points with given locations and nv points with variable locations.

Input: **vwgt** $nv \times nf$ weights of variable points (locations to be found)

fwgt $nv \times nf$ matrix of weights between variable (rows) and fixed (columns) points.

optn must be a scalar or vector specifying some options:

1. amount of printed output, default=0 means no printed output
2. method, default=0

Output: **gof**

post `post[i]` defines for each variable point i which of the fixed points is coinciding

Restrictions:

1. The first two input arguments must be integer. String, complex, or missing values are not permitted.
2. The input matrices `vwgt` and `fwgt` must have the same number of rows corresponding to the variable points.

Relationships: locatn(), lp()

Examples: The following are a few of the 18 test examples from ACM TOMS 558:

1. Test Problem 1:

```
vcst = [  0  3  2,
          3  0  1,
          2  1  0 ];
fcst = [ 3  2  3  1,
         1  2  1  2,
         9  1  2  2 ];
optn = 2;
< gof,post > = locat1(vcst,fcst,optn);
```

```
          Optimal Positions of the 3 Variable Points
           Coincide with the following Fixed Points

                    VarPoint  FixedPoint
                           1           2
```

```
                                    2           2
                                    3           1
```

2. Test Problem 2:

```
vcst = [ 0  1  3,
         1  0  3,
         3  3  0 ];
fcst = [ 9  1  1  2,
         2  1  3  2,
         2  2  4  1 ];
optn = 2;
< gof,post > = locat1(vcst,fcst,optn);


          Optimal Positions of the 3 Variable Points
           Coincide with the following Fixed Points

                       VarPoint  FixedPoint
                              1           1
                              2           2
                              3           2
                 Total Processing Time: 0.001
```

3. Test Problem 3:

```
vcst = [ 0 21 14  9  7,
        21  0 34 24 32,
        14 34  0 14 36,
         9 24 14  0  8,
         7 32 36  8  0 ];
fcst = [ 50 49 81 46 52 80 83 48  8 33 99  0 50 76 38 40 70 32 30 92,
         71 63 77 31 62  5  2 85 13 37 19 64 17 59 49 75 52 44 88 94,
         73 32 27 72  2  7 66 79  1 19 22 76 50 68 56 87 83 69 58 36,
         97 24 64 49 52  1 74  8 88 84 53 84 46 74 92 45 29 19 54 71,
         26 51 46 49 93 34 46 35 19 28 21 15 24 31 41 89 22 92 97 99 ];
optn = 2;
< gof,post > = locat1(vcst,fcst,optn);


          Optimal Positions of the 5 Variable Points
           Coincide with the following Fixed Points

                       VarPoint  FixedPoint
                              1          11
                              2          12
```

```
                                3          12
                                4          11
                                5          12
```

4. Test Problem 4:

```
vcst = [ 0 21 14  9  7,
        21  0 34 24 32,
        14 34  0 14 36,
         9 24 14  0  8,
         7 32 36  8  0 ];
fcst = [ 33  0 50 48 46 38 32 92 80 99 30  8 40 70 52 50 49 83 76 81,
         37 64 71 85 31 49 44 94  5 19 88 13 75 52 62 17 63  2 59 77,
         19 76 73 79 72 56 69 36  7 22 58  1 87 83  2 50 32 66 68 27,
         84 84 97  8 49 92 19 71  1 53 54 88 45 29 52 46 24 74 74 64,
         28 15 26 35 49 41 92 99 34 21 97 19 89 22 93 24 51 46 31 46 ];
optn = 2;
< gof,post > = locat1(vcst,fcst,optn);


          Optimal Positions of the 5 Variable Points
           Coincide with the following Fixed Points

                      VarPoint  FixedPoint
                             1          11
                             2          11
                             3          11
                             4          11
                             5          11
                   Total Processing Time: 0
```

5. Test Problem 5:

```
vcst = [ 0  2  2  2  2,
         2  0 20  1  0,
         2 20  0  0  0,
         2  1  0  0 40,
         2  0  0 40  0 ];
fcst = [ 10  0  0,
          4  1  4,
          4  1  4,
          4  5  4,
          4  5  4 ];
optn = 2;
< gof,post > = locat1(vcst,fcst,optn);
```

```
          Optimal Positions of the 5 Variable Points
           Coincide with the following Fixed Points

                       VarPoint  FixedPoint
                              1           1
                              2           2
                              3           2
                              4           2
                              5           2
                  Total Processing Time: 0
```

6. Test Problem 6:

```
vcst = [  0  1  1,
          1  0  1,
          1  1  0 ];
fcst = [  1  1  6  1  6,
          4  1  1  1  1,
          1  1  1  1  1 ];
optn = 2;
< gof,post > = locat1(vcst,fcst,optn);
```

```
          Optimal Positions of the 3 Variable Points
           Coincide with the following Fixed Points

                       VarPoint  FixedPoint
                              1           3
                              2           2
                              3           3
```

7. Test Problem 7:

```
vcst = [ 0  1  0,
         1  0  1,
         0  1  0 ];
fcst = [ 3  2  2  0  0,
         2  2  2  4  4,
         4  0  2  6  4 ];
optn = 2;
< gof,post > = locat1(vcst,fcst,optn);
```

```
          Optimal Positions of the 3 Variable Points
           Coincide with the following Fixed Points
```

```
                    VarPoint  FixedPoint
                           1           2
                           2           4
                           3           4
                Total Processing Time: 0
```

8. Test Problem 8:

```
vcst = [  0  1  0,
          1  0  1,
          0  1  0 ];
fcst = [  2  3  0  2  0,
          2  2  4  2  4,
          2  4  6  0  4 ];
optn = 2;
< gof,post > = locat1(vcst,fcst,optn);


          Optimal Positions of the 3 Variable Points
           Coincide with the following Fixed Points

                       VarPoint  FixedPoint
                              1           2
                              2           3
                              3           3
```

9. Test Problem 9:

```
vcst =  1;
fcst = [ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 ];
optn = 2;
< gof,post > = locat1(vcst,fcst,optn);


          Optimal Positions of the 1 Variable Points
           Coincide with the following Fixed Points

                       VarPoint  FixedPoint
                              1           8
```

10. Test Problem 10:

```
vcst = 1;
fcst = [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 ];
optn = 2;
 < gof,post > = locat1(vcst,fcst,optn);
```

```
          Optimal Positions of the 1 Variable Points
           Coincide with the following Fixed Points

                       VarPoint  FixedPoint
                              1          11
```

11. Test Problem 11:

```
vcst = 1;
fcst = [ 4  2 15  9  0 45  3  7  5  0  0  1 50  4  0 12 32  4 22  7 ];
optn = 2;
< gof,post > = locat1(vcst,fcst,optn);
```

```
          Optimal Positions of the 1 Variable Points
           Coincide with the following Fixed Points

                       VarPoint  FixedPoint
                              1          13
```

12. Test Problem 12:

```
vcst = [  0  1  1  1  1,
          1  0  1  1  1,
          1  1  0  1  1,
          1  1  1  0  1,
          1  1  1  1  0 ];
fcst = [  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1,
          1  1  1  1  1  1  1  1  1  1  1  1  1  1  1,
          1  1  1  1  1  1  1  1  1  1  1  1  1  1  1,
          1  1  1  1  1  1  1  1  1  1  1  1  1  1  1,
          1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 ];
```

```
          Optimal Positions of the 5 Variable Points
           Coincide with the following Fixed Points

                       VarPoint  FixedPoint
                              1           8
                              2           8
                              3           8
                              4           8
                              5           8
```

5.6 The `lpassign()` Function

< xr,rc,duals,sens > = lpassign(cost<,optn>)

Purpose: For a given $n \times n$ cost matrix the linear assignment problem is a a linearly constrained $0, 1$ optimization problem, computing the $n \times n$ parameter matrix x which minimizes the common linear assignment problem (LAP)

$$\min \sum_{i,j} x_{ij} * c_{ij},$$

and the linear bottleneck assignment problem (LBAP)

$$\min \max_{ij} \sum_{i,j} x_{ij} * c_{ij},$$

both subject to the constraint that each row and column may contain only one 1 and otherwise zeros. That means it is an integer problem with row and column constraints adding up to 1. For solving the LAP, either `lpsolve`, the *Hungarian method*, or the `LAPJV` algorithm by Jonker & Volgenant (1987) can be used. For solving the LBAP currently only a version of the `BOTJV` algorithm by Jonker & Volgenant (1987) can be used.

Input: cost must be a square $n \times n$ matrix of real values without any missing values

optn This argument must be 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.

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

Option Name	Column 2	Meaning
`"max"`		perform maximization (minimization is default)
`"nopr"`		no printed output (is default)
`"pres"`		perform presolution (is normally not needed)
`"print"`	int	amount of printed output (larger is more, default=0)
`"sens"`		print sensitivity analysis
`"vers"`	char	version of the algorithm
	LPS	solving LAP with `LPSOLVE` algorithm, this is the default
	HUN	solving LAP with the *Hungarian method*
	JOV	solving LAP with `LAPJV` algorithm
	BOT	solving LBAP with `BOTJV` algorithm

Output: xr $n \times n$ matrix of $0, 1$ values.

rc vector of scalar results:

duals
- for `lpsolve`: the $3 \times (n * n + 2 * n)$ matrix containing:
  1. the duals for estimates and row and column constraints in its first row
  2. the lower and upper bounds ($[from, to]$) of the sensitivities of the duals in rows two and three.
- for `lapjv`: the $n \times 2$ matrix of duals of row and column constraints
- for `botjv`: a missing value is returned

sens
- Returns the $2 \times n * n$ matrix of the lower and upper bounds of the sensitivities $[from, to]$ of the estimates x_{ij}.
- for `lapjv`: a missing value is returned
- for `botjv`: a missing value is returned

Restrictions:
1. The input cost matrix may not contain any missing value, string or complex data.
2. Note, that the `LAPJV` and `BOTJV` algorithms work only with rounded integer cost matrices and you may have to multiply the data by a factor large enough to make them distinct.

Relationships: lp(), lptransp(), nlp()

Examples:
1. Example of CRAN function `lp.assign` using `lpsolve`:

```
print "Assignment problem from CRAN: crit=8";
cost = [ 2  7  7  2 ,
         7  7  3  2 ,
         7  2  8 10 ,
         1  9  8  2 ];
rnam = [" RC_1:RC_4 "];
cost = rname(cost,rnam);
cnam = [" CC_1:CC_4 "];
cost = cname(cost,cnam);
```

Here the default `lpsolve` algorithm is used:

```
optn = [ "sens"          ,
         "print"      6 ];
< xr,rc,sens,duals > = lpassign(cost,optn);
print "LPSOLVE: Xsolution=", xr;
print "GOF=", rc;
print "SENS=", sens;
print "DUALS=", duals;
```

Linear Assignment Problem: LPSOLVE

Input Cost Matrix

	CC_1	CC_2	CC_3	CC_4
RC_1	2.000000000	7.000000000	7.000000000	2.000000000
RC_2	7.000000000	7.000000000	3.000000000	2.000000000
RC_3	7.000000000	2.000000000	8.000000000	10.00000000
RC_4	1.000000000	9.000000000	8.000000000	2.000000000

Input Row and Column Constraints

****

N	Row Name	Row RHS	Col Name	Col RHS
1	RC_1 ==	1.00000000	CC_1 ==	1.00000000
2	RC_2 ==	1.00000000	CC_2 ==	1.00000000
3	RC_3 ==	1.00000000	CC_3 ==	1.00000000
4	RC_4 ==	1.00000000	CC_4 ==	1.00000000

Value of Objective Function = 8

Parameter Estimates

****

Dense Matrix (4 by 4)

	CC_1	CC_2	CC_3	CC_4
RC_1	0	0	0	1.0000000
RC_2	0	0	1.0000000	0
RC_3	0	1.0000000	0	0
RC_4	1.0000000	0	0	0

Sensitivities of Estimates

****

Dense Matrix (2 by 16)

	C_1_1	C_2_1	C_3_1	C_4_1	C_1_2
Sens_from	1.0000000	0	2.0000000	-1.00e+030	2.0000000
Sens_to	1.00e+030	1.00e+030	1.00e+030	2.0000000	1.00e+030

	C_2_2	C_3_2	C_4_2	C_1_3	C_2_3
Sens_from	0	-1.00e+030	1.0000000	5.0000000	-1.00e+030
Sens_to	1.00e+030	5.0000000	1.00e+030	1.00e+030	5.0000000

	C_3_3	C_4_3	C_1_4	C_2_4	C_3_4
Sens_from	5.0000000	4.0000000	-1.00e+030	0	2.0000000
Sens_to	1.00e+030	1.00e+030	3.0000000	1.00e+030	1.00e+030

	C_4_4
Sens_from	1.0000000
Sens_to	1.00e+030

Duals of Constraints and Estimates

****

Dense Matrix (3 by 24)

	RC_1	RC_2	RC_3	RC_4	CC_1
Duals	2.0000000	0	2.0000000	1.0000000	0
Dual_from	1.0000000	-1.00e+030	1.0000000	1.0000000	-1.00e+030
Dual_to	1.0000000	1.00e+030	1.0000000	1.0000000	1.00e+030

	CC_2	CC_3	CC_4	C_1_1	C_2_1
Duals	0	3.0000000	0	0	7.0000000
Dual_from	-1.00e+030	1.0000000	1.0000000	-1.00e+030	0
Dual_to	1.00e+030	1.0000000	1.0000000	1.00e+030	0

	C_3_1	C_4_1	C_1_2	C_2_2	C_3_2
Duals	5.0000000	0	5.0000000	7.0000000	0
Dual_from	0	-1.00e+030	0	0	-1.00e+030
Dual_to	0	1.00e+030	0	0	1.00e+030

	C_4_2	C_1_3	C_2_3	C_3_3	C_4_3
Duals	8.0000000	2.0000000	0	3.0000000	4.0000000
Dual_from	0	0	-1.00e+030	0	0
Dual_to	0	0	1.00e+030	0	0

```
          |       C_1_4      C_2_4      C_3_4      C_4_4
         ------------------------------------------------
    Duals |           0  2.0000000  8.0000000  1.0000000
Dual_from |  -1.00e+030          0          0          0
  Dual_to |   1.00e+030          0          0  1.0000000
```

2. Example of CRAN function with Hungarian method:

```
print "HUNGARIAN: Assignment problem from CRAN: crit=8";
cost = [ 2  7  7  2 ,
         7  7  3  2 ,
         7  2  8 10 ,
         1  9  8  2 ];
rnam = [" RC_1:RC_4 "];
cost = rname(cost,rnam);
cnam = [" CC_1:CC_4 "];
cost = cname(cost,cnam);

 print "Use LPSOLVE algorithm";
 optn = [ "vers"   "hun" ,
          "print"      6 ];
 < xr,rc > = lpassign(cost,optn);


          Linear Assignment Problem: Hungarian Method


                       Input Cost Matrix

               CC_1         CC_2         CC_3         CC_4
  RC_1   2.000000000  7.000000000  7.000000000  2.000000000
  RC_2   7.000000000  7.000000000  3.000000000  2.000000000
  RC_3   7.000000000  2.000000000  8.000000000  10.00000000
  RC_4   1.000000000  9.000000000  8.000000000  2.000000000


          Value of Objective Function =            8


                      Parameter Estimates
                      *******************

                    Sparse Matrix (4 by 4)

                   |    CC_1   CC_2   CC_3   CC_4
```

```
          |    CC_1   CC_2   CC_3   CC_4
     -------------------------------------
RC_1 |       0      0      0      1
RC_2 |       0      0      1      0
RC_3 |       0      1      0      0
RC_4 |       1      0      0      0
```

3. Example of CRAN function with `lapjv`:

```
print "Use Jonker & Volgenant LAPJV algorithm";
optn = [ "duals"        ,
         "vers"   "jov" ,
         "print"      6 ];
< xr,rc,duals > = lpassign(cost,optn);


      Linear Assignment Problem: Jonkers and Volgenant

                     Input Cost Matrix

               CC_1         CC_2         CC_3         CC_4
  RC_1   2.000000000  7.000000000  7.000000000  2.000000000
  RC_2   7.000000000  7.000000000  3.000000000  2.000000000
  RC_3   7.000000000  2.000000000  8.000000000  10.00000000
  RC_4   1.000000000  9.000000000  8.000000000  2.000000000


        Value of Objective Function =            8


                  Parameter Estimates
                  *******************

                 Sparse Matrix (4 by 4)

                |    CC_1   CC_2   CC_3   CC_4
               ---------------------------------
          RC_1 |       0      0      0      1
          RC_2 |       0      0      1      0
          RC_3 |       0      1      0      0
          RC_4 |       1      0      0      0


          Duals of Constraints and Estimates
          **********************************

                Dense Matrix (4 by 2)
```

```
          |  Duals_Row Duals_Col
     -----------------------
RC_1 |          0         1
RC_2 |          5        -2
RC_3 |          5        -3
RC_4 |          1         1
```

4. Solving the LBAP using `botjv`:

```
print "Use Jonker & Volgenant Bottleneck algorithm";
optn = [ "vers"   "bot" ,
         "print"      6 ];
< xr,rc,duals > = lpassign(cost,optn);
```

```
            Linear Bottleneck Assignment Problem

                      Input Cost Matrix

               CC_1         CC_2         CC_3         CC_4
  RC_1   2.000000000  7.000000000  7.000000000  2.000000000
  RC_2   7.000000000  7.000000000  3.000000000  2.000000000
  RC_3   7.000000000  2.000000000  8.000000000  10.00000000
  RC_4   1.000000000  9.000000000  8.000000000  2.000000000

         Value of Objective Function =            3

                      Parameter Estimates
                      *******************

                     Sparse Matrix (4 by 4)

          |    CC_1   CC_2   CC_3   CC_4
     ---------------------------------
RC_1 |       0      0      0      1
RC_2 |       0      0      1      0
RC_3 |       0      1      0      0
RC_4 |       1      0      0      0
```

5.7 The `lptransp()` Function

< xr,rc,duals,sens > = lptransp(cost,rrhs,crhs<,optn>)

Purpose: For a given $n \times m$ cost matrix the linear transport problem is a linearly constrained integer optimization problem, computing a $n \times m$ parameter matrix x with

$$\min \sum_{i,j} x_{ij} * c_{ij},$$

where each row of x sums up to `rrhs[n]` (supply) and each column of x sums up to `crhs[m]` (demand).

Input: **cost** must be a $n \times m$ matrix of real values without any missing values

rrhs an $n \times 2$ matrix of real values of the n lower (column 1) and upper (column 2) bounds for

$$\sum_{j} X_{ij}, \quad i = 1 \ldots, n.$$

crhs an $m \times 2$ matrix of real values of the m lower (column 1) and upper (column 2) bounds for

$$\sum_{i} X_{ij}, \quad j = 1 \ldots, m.$$

optn This argument must be 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.

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

Option Name	Column 2	Meaning
`"max"`		perform maximization (minimization is default)
`"nopr"`		no printed output (is default)
`"pres"`		perform presolution (is normally not needed)
`"print"`	int	amount of printed output (larger is more, default=0)
`"sens"`		print sensitivity analysis

Output: **xr** $n \times m$ matrix of integer estimates.

rc vector of scalar results:

duals the $3 \times (n * m + n + m)$ matrix of duals of the estimates x_{ij} and row and column constraints.

sens the $2 \times n * m$ matrix of sensitivity intervals $[from, to]$ of the estimates x_{ij}.

Restrictions:
1. The input cost matrix may not contain any missing value, string or complex data.
2. The dimension of second and third input arguments must be compatible with those of the first input argument.

Relationships: lp(), lpassign(), nlp()

Examples: 1. Example of CRAN function `lp.assign`:

```
cost = cons(8,5,10000.);
cost[4,1] = 0;
cost[1,5] = cost[2,5] = cost[3,5] = 0.;
cost[5,5] = cost[6,5] = cost[7,5] =  cost[8,5] = 0.;
cost[1,2] = cost[2,3] = cost[3,4] = 7.;
cost[1,3] = cost[2,4] = 7.7;
cost[5,1] = cost[7,3] = 8.;
cost[1,4] = 8.4; cost[6,2] = 9.; cost[8,4] = 10.;
cost[4,2] = .7; cost[4,3] = 1.4; cost[4,4] = 2.1;
rnam = [" RC_1:RC_8 "];
cost = rname(cost,rnam);
cnam = [" CC_1:CC_5 "];
cost = cname(cost,cnam); print "Cost=", cost;


rrhs = [ 200. 300. 350. 200. 100. 50.  100. 150. ];
rrhs = cons(8,1,.) -> rrhs‘;
crhs = [ 250 100 400 500 200 ];
crhs = crhs‘ -> cons(5,1,.);

 optn = [ "sens"         ,
          "print"      6 ];
 < xr,rc,sens,duals > = lptransp(cost,rrhs,crhs,optn);
```

```
Cost=
       |       CC_1       CC_2       CC_3       CC_4       CC_5
-------------------------------------------------------------------
  RC_1 |   10000.00    7.00000    7.70000    8.40000   0.000000
  RC_2 |   10000.00   10000.00    7.00000    7.70000   0.000000
  RC_3 |   10000.00   10000.00   10000.00    7.00000   0.000000
  RC_4 |   0.000000   0.700000    1.40000    2.10000   10000.00
  RC_5 |    8.00000   10000.00   10000.00   10000.00   0.000000
  RC_6 |   10000.00    9.00000   10000.00   10000.00   0.000000
  RC_7 |   10000.00   10000.00    8.00000   10000.00   0.000000
  RC_8 |   10000.00   10000.00   10000.00   10.00000   0.000000
```

Input Cost Matrix

	CC_1	CC_2	CC_3	CC_4
RC_1	10000.00000	7.000000000	7.700000000	8.400000000
RC_2	10000.00000	10000.00000	7.000000000	7.700000000
RC_3	10000.00000	10000.00000	10000.00000	7.000000000
RC_4	0	0.700000000	1.400000000	2.100000000
RC_5	8.000000000	10000.00000	10000.00000	10000.00000
RC_6	10000.00000	9.000000000	10000.00000	10000.00000
RC_7	10000.00000	10000.00000	8.000000000	10000.00000
RC_8	10000.00000	10000.00000	10000.00000	10.00000000

Input Cost Matrix

	CC_5
RC_1	0
RC_2	0
RC_3	0
RC_4	10000.00000
RC_5	0
RC_6	0
RC_7	0
RC_8	0

Input Row and Column Constraints

****

N	Row Name		Row RHS	Col Name		Col RHS
1	RC_1	<=	200.000000	CC_1	>=	250.000000
2	RC_2	<=	300.000000	CC_2	>=	100.000000
3	RC_3	<=	350.000000	CC_3	>=	400.000000
4	RC_4	<=	200.000000	CC_4	>=	500.000000
5	RC_5	<=	100.000000	CC_5	>=	200.000000
6	RC_6	<=	50.0000000			
7	RC_7	<=	100.000000			
8	RC_8	<=	150.000000			

Value of Objective Function = 7790

Parameter Estimates

****

Dense Matrix (8 by 5)

	CC_1	CC_2	CC_3	CC_4	CC_5
RC_1	0	100.00000	100.00000	0	0
RC_2	0	0	200.00000	100.00000	0
RC_3	0	0	0	350.00000	0
RC_4	200.00000	0	0	0	0
RC_5	50.000000	0	0	0	50.000000
RC_6	0	0	0	0	50.000000
RC_7	0	0	100.00000	0	0
RC_8	0	0	0	50.000000	100.00000

Sensitivities of Estimates

Dense Matrix (2 by 40)

	C_1_1	C_2_1	C_3_1	C_4_1	C_5_1
Sens_from	6.4000000	5.7000000	5.0000000	-1.00e+030	7.9000000
Sens_to	1.00e+030	1.00e+030	1.00e+030	0.1000000	10000.000

	C_6_1	C_7_1	C_8_1	C_1_2	C_2_2
Sens_from	8.0000000	6.7000000	8.0000000	-1.6000000	6.3000000
Sens_to	1.00e+030	1.00e+030	1.00e+030	7.1000000	1.00e+030

.....

	C_7_4	C_8_4	C_1_5	C_2_5	C_3_5
Sens_from	8.7000000	8.7000000	-1.6000000	-2.3000000	-3.0000000
Sens_to	1.00e+030	10.100000	1.00e+030	1.00e+030	1.00e+030

	C_4_5	C_5_5	C_6_5	C_7_5	C_8_5
Sens_from	-8.0000000	-9992.0000	-1.00e+030	-1.3000000	0
Sens_to	1.00e+030	0	0	1.00e+030	1.3000000

Duals of Constraints and Estimates

Dense Matrix (3 by 53)

```
          |        RC_1       RC_2       RC_3       RC_4       RC_5
----------------------------------------------------------------------
    Duals |  -1.6000000 -2.3000000 -3.0000000 -8.0000000          0
Dual_from |   200.00000  300.00000  350.00000  200.00000  100.00000
  Dual_to |   250.00000  350.00000  400.00000  250.00000  200.00000

          |        RC_6       RC_7       RC_8       CC_1       CC_2
----------------------------------------------------------------------
    Duals |           0 -1.3000000          0  8.0000000  8.6000000
Dual_from |   50.000000  100.00000 -1.00e+030  200.00000  50.000000
  Dual_to |   150.00000  150.00000  1.00e+030  250.00000  100.00000

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

          |       C_1_5      C_2_5      C_3_5      C_4_5      C_5_5
----------------------------------------------------------------------
    Duals |   1.6000000  2.3000000  3.0000000  10008.000          0
Dual_from |  -50.000000 -50.000000 -50.000000 -50.000000 -1.00e+030
  Dual_to |   100.00000  100.00000  100.00000  50.000000  1.00e+030

          |       C_6_5      C_7_5      C_8_5
-------------------------------------------
    Duals |           0  1.3000000          0
Dual_from |  -1.00e+030 -50.000000 -1.00e+030
  Dual_to |   1.00e+030  100.00000  1.00e+030
```

2. Example of PROC LP in SAS OR:

```
cost = [  20  58 121  70 193  60  74 213 218  54 ,
          58  20  92  94 174 118  71 185 173  57 ,
         121  92  20  87  83 172 163  94 102 149 ,
          70  94  87  20 137  96 142 154 189 122 ,
         193 174  83 137  20 223 245  34  95 230 ,
          60 118 172  96 233  20 109 259 273  92 ,
          74  71 163 142 245 109  20 257 240  20 ,
         213 185  94 164  34 259 257  20  67 244 ,
         218 173 102 189  95 273 240  67  20 232 ,
          54  59 149 122 230  92  20 244 232  20 ];

rnam = [" Atlanta Chicago Denver Houston Los_Ange Miami New_York San_Fran Seattle
cost = rname(cost,rnam);
cost = cname(cost,rnam); print "Cost=", cost;


rrhs = [ 10 150 90 27 80 26 80  25  7 15 ];
```

```
rrhs = cons(10,1,.) -> rrhs‘;
crhs = [ 50  75 89  8 27 39 64 100 50  8 ];
crhs =  crhs‘ -> cons(10,1,.);

 optn = [ "sens"         ,
          "print"      6 ];
 < xr,rc > = lptransp(cost,rrhs,crhs,optn);
```

Input Cost Matrix

	Atlanta	Chicago	Denver	Houston
Atlanta	20.00000000	58.00000000	121.0000000	70.00000000
Chicago	58.00000000	20.00000000	92.00000000	94.00000000
Denver	121.0000000	92.00000000	20.00000000	87.00000000
Houston	70.00000000	94.00000000	87.00000000	20.00000000
Los_Ange	193.0000000	174.0000000	83.00000000	137.0000000
Miami	60.00000000	118.0000000	172.0000000	96.00000000
New_York	74.00000000	71.00000000	163.0000000	142.0000000
San_Fran	213.0000000	185.0000000	94.00000000	164.0000000
Seattle	218.0000000	173.0000000	102.0000000	189.0000000
Washingt	54.00000000	59.00000000	149.0000000	122.0000000

Input Cost Matrix

	Los_Ange	Miami	New_York	San_Fran
Atlanta	193.0000000	60.00000000	74.00000000	213.0000000
Chicago	174.0000000	118.0000000	71.00000000	185.0000000
Denver	83.00000000	172.0000000	163.0000000	94.00000000
Houston	137.0000000	96.00000000	142.0000000	154.0000000
Los_Ange	20.00000000	223.0000000	245.0000000	34.00000000
Miami	233.0000000	20.00000000	109.0000000	259.0000000
New_York	245.0000000	109.0000000	20.00000000	257.0000000
San_Fran	34.00000000	259.0000000	257.0000000	20.00000000
Seattle	95.00000000	273.0000000	240.0000000	67.00000000
Washingt	230.0000000	92.00000000	20.00000000	244.0000000

Input Cost Matrix

	Seattle	Washingt
Atlanta	218.0000000	54.00000000
Chicago	173.0000000	57.00000000
Denver	102.0000000	149.0000000
Houston	189.0000000	122.0000000
Los_Ange	95.00000000	230.0000000

Miami	273.0000000	92.00000000
New_York	240.0000000	20.00000000
San_Fran	67.00000000	244.0000000
Seattle	20.00000000	232.0000000
Washingt	232.0000000	20.00000000

Input Row and Column Constraints

****

N	Row Name	Row RHS	Col Name	Col RHS
1	Atlanta <=	10.0000000	Atlanta >=	50.0000000
2	Chicago <=	150.000000	Chicago >=	75.0000000
3	Denver <=	90.0000000	Denver >=	89.0000000
4	Houston <=	27.0000000	Houston >=	8.00000000
5	Los_Ange <=	80.0000000	Los_Ange >=	27.0000000
6	Miami <=	26.0000000	Miami >=	39.0000000
7	New_York <=	80.0000000	New_York >=	64.0000000
8	San_Fran <=	25.0000000	San_Fran >=	100.000000
9	Seattle <=	7.00000000	Seattle >=	50.0000000
10	Washingt <=	15.0000000	Washingt >=	8.00000000

Value of Objective Function = 22928

Parameter Estimates

****

Dense Matrix (10 by 10)

	Atlanta	Chicago	Denver	Houston	Los_Ange
Atlanta	10.000000	0	0	0	0
Chicago	30.000000	75.000000	2.0000000	0	0
Denver	0	0	87.000000	0	0
Houston	0	0	0	8.0000000	19.000000
Los_Ange	0	0	0	0	8.0000000
Miami	0	0	0	0	0
New_York	0	0	0	0	0
San_Fran	0	0	0	0	0
Seattle	0	0	0	0	0
Washingt	10.000000	0	0	0	0

	Miami	New_York	San_Fran	Seattle	Washingt

Atlanta	0	0	0	0	0
Chicago	0	0	0	43.000000	0
Denver	0	0	3.0000000	0	0
Houston	0	0	0	0	0
Los_Ange	0	0	72.000000	0	0
Miami	26.000000	0	0	0	0
New_York	8.0000000	64.000000	0	0	8.0000000
San_Fran	0	0	25.000000	0	0
Seattle	0	0	0	7.0000000	0
Washingt	5.0000000	0	0	0	0

Sensitivities of Estimates

****

Dense Matrix (2 by 100)

	C_01_01	C_02_01	C_03_01	C_04_01	C_05_01
Sens_from	-1.00e+030	20.000000	-14.000000	43.000000	-74.000000
Sens_to	22.000000	71.000000	1.00e+030	1.00e+030	1.00e+030

	C_06_01	C_07_01	C_08_01	C_09_01	C_10_01
Sens_from	-18.000000	71.000000	-88.000000	-95.000000	52.000000
Sens_to	1.00e+030	1.00e+030	1.00e+030	1.00e+030	57.000000

.....

	C_01_10	C_02_10	C_03_10	C_04_10	C_05_10
Sens_from	-31.000000	7.0000000	-65.000000	-8.0000000	-125.00000
Sens_to	1.00e+030	1.00e+030	1.00e+030	1.00e+030	1.00e+030

	C_06_10	C_07_10	C_08_10	C_09_10	C_10_10
Sens_from	-69.000000	3.55e-015	-139.00000	-146.00000	3.0000000
Sens_to	1.00e+030	37.000000	1.00e+030	1.00e+030	1.00e+030

Duals of Constraints and Estimates

****

Dense Matrix (3 by 120)

	Atlanta	Chicago	Denver	Houston	Los_Ange

Duals	-51.000000	-13.000000	-85.000000	-28.000000	-145.00000
Dual_from	10.000000	150.00000	90.000000	27.000000	80.000000
Dual_to	18.000000	158.00000	92.000000	29.000000	82.000000

	Miami	New_York	San_Fran	Seattle	Washingt
Duals	-89.000000	0	-159.00000	-166.00000	-17.000000
Dual_from	26.000000	-1.00e+030	25.000000	7.0000000	15.000000
Dual_to	34.000000	1.00e+030	27.000000	15.000000	23.000000

	C_01_10	C_02_10	C_03_10	C_04_10	C_05_10
Duals	85.000000	50.000000	214.00000	130.00000	355.00000
Dual_from	-8.0000000	-8.0000000	-2.0000000	-2.0000000	-2.0000000
Dual_to	5.0000000	5.0000000	5.0000000	5.0000000	5.0000000

	C_06_10	C_07_10	C_08_10	C_09_10	C_10_10
Duals	161.00000	0	383.00000	378.00000	17.000000
Dual_from	-8.0000000	-1.00e+030	-2.0000000	-8.0000000	-8.0000000
Dual_to	8.0000000	1.00e+030	5.0000000	5.0000000	5.0000000

5.8 The `maxempty()` Function

< vol,box > = maxempty(xy<, *xybc* >)

Purpose: The `maxempty()` function finds the coordinates and the volume of the largest box parallel to the axes of x and y and completely surrounded by some points of a specified set of n points $X_{ij}, i = 1, \ldots, n, j = 1, 2$ in 2-dimensional space. The function is programmed based on an algorithm by Hans W. Borchers in CRAN.

Input: The first input is an $n \times 2$ matrix of the (x, y) coordinates of n points. The second input si a 2×2 matrix specifying lower and upper bounds for x (column 1) and y (column 2).

Output: **vol** a scalar which is the area of the box

box a 2×2 matrix of the lower left and upper right corner points of the rectangular box (parallel to (x, y) axes).

Restrictions:
1. The first input argument must not contain any missing or complex values or string data.
2. The first input argument must have two columns.

Relationships: boundbox()

Examples: 1. Small example of $n = 100$ points:

```
< vol,box > = maxempty(xy);
print "Vol=",vol;
print "Box=",box;
```

```
Vol= 0.08239

Box=
           |      Xdim      Ydim
-------------------------------
   LowLeft |   0.70237   0.17973
  UppRight |   0.81758   0.89484
```

5.9 The `mstgra()` Function

The function `mstgra()` generates a minimum spanning tree based on the sparse or dense input of a graph. Note that the function `mstree()` was renamed into `mstdis()` for minimum spanning tree based on distances among n points.

< gof,pred > = mstgra(adja,cost<,optn>)

< gof,pred > = mstgra(indx,vert,cost<,optn>)

Purpose: Assuming the input graph has n edges with path connections (a_{ij} where a_{ij} is 1 if there is a path from edge i to edge j and otherwise $a_{ij} = 0$. For given (undirected) graph data this function computes a minimum spanning tree with Prim's (1957) algorithm.

Input: There are two forms of input data specifying the graph:

binary $n \times n$ matrix of $(0, 1)$ adjacency data $\mathbf{A} = (a_{ij})$, where a_{ij} is 1 if there is a path from edge i to edge j and otherwise $a_{ij} = 0$.

- sparse `(indx,vert)` input form: the $n+1$ vector `indx` gives the index bounds for the origin and end of the path (defining the i), and the m vector `vert` lists the indices of the target edge (defining j). Note, there must be `indx[1]=0` and `indx[n+1] = m`, and the entries must be monoton increasing integers.

a numeric m vector of (small) integer costs c of the edges.

2. a numeric vector of options:

[**1**] amount of printed output, default=0

[**2**] method (algorithm), currently unused, default=0

[**3**] maximum edge cost, default $max(c_i)$

Output: **gof** lists some scalar results:

1. error return code
2. total cost of all edges of the tree
3. computing time in seconds

pred is a vector `pred[i]` containing the node number of the predecessor of node i.

Restrictions: 1. All the input data is integer. String, complex, and missing values are not permitted.

2. The dimensions of input data must be compatible.

Relationships: mstdis(), hamilton()

Examples: 1. Example of TOMS 613, $n = 16, m = 84$:

```
 indx = [  0   4   9  14  17  22  30  38  43  48
          56  64  69  72  77  82  84 ];
 vert = [ 2   5   6   1   3   5   6   7   2   4   6   7   8   3   7
          8   1   2   6   9  10   1   2   3   5   7   9  10  11   2
          3   4   6   8  10  11  12   3   4   7  11  12   5   6  10
         13  14   5   6   7   9  11  13  14  15   6   7   8  10  12
         14  15  16   7   8  11  15  16   9  10  14   9  10  11  13
         15  10  11  12  14  16  11  12  15 ];
 cost = [ 9  12   6   9   6   3   3   4   6   7   3  10   2   7   3
          9  12   3   6  10   5   6   3   3   6   7   4   1   5   4
         10   3   7  12   6   2   5   2   9  12  10   7  10   4   5
          3   3   5   1   6   5   4   8   2   3   5   2  10   4   3
          6   7   3   5   7   3   4   6   3   8   6   3   2   6   6
          1   3   7   4   1  10   3   6  10 ];


  optn = [  3 ,   /* ipri */
            0 ,   /* method */
           15 ];  /* maxc */
  < gof,pred > = mstgra(indx,vert,cost,optn);
  print "GOF =",gof;
  print "PRED=",pred;


GOF =
               |         1
-------------------------
  Return_Code |   0.00000
   Total_Cost |    42.000
   Total_Time |    0.0010


PRED=
 L |   Predecessor   Vertex_Cost
--------------------------------
 1 |             0             0
 2 |             6             3
 3 |             6             3
 4 |             7             3
 5 |             2             3
 6 |             1             6
 7 |             2             4
 8 |             3             2
 9 |            14             3
10 |             6             1
11 |             7             2
12 |            11             3
```

```
13 |             9             3
14 |            10             2
15 |            14             1
16 |            11             3
```

2. Example of TOMS 613, $n = 8, m = 26$:

```
indx = [ 0   5   9  12  15  19  21  24  26 ];
vert = [ 2   3   7   5   3   1   5   4   6   1   2   2   5   8   1
         2   4   8   3   7   6   1   8   7   5   4 ];
cost = [ 4   4   3   1   1   4   3   3   3   4   1   3   4   2   1
         3   4   1   3   4   4   3   2   2   1   2 ];


 optn = [  3 ,   /* ipri */
           0 ,   /* method */
          10 ];  /* maxc */
 < gof,pred > = mstgra(indx,vert,cost,optn);
 print "GOF =",gof;
 print "PRED=",pred;


GOF =
              |         1
--------------------------
  Return_Code |   0.00000
   Total_Cost |    13.000
   Total_Time |   0.00000


PRED=
 L |   Predecessor   Vertex_Cost
-------------------------------
 1 |             0             0
 2 |             5             3
 3 |             2             1
 4 |             8             2
 5 |             1             1
 6 |             3             3
 7 |             8             2
 8 |             5             1
```

5.10 The `splnet()` Function

< gof,path,len > = splnet(adja,cost,brack<,optn>)

< gof,path,len > = splnet(indx,vert,cost,brack<,optn>)

Purpose: Assuming the input graph has n edges with path connections (a_{ij} where a_{ij} is 1 if there is a path from edge i to edge j and otherwise $a_{ij} = 0$. For given graph data this function computes the shortest path between a start node a and an end node b using an algorithm by U. Pape (1980).

Input: There are two forms of input data specifying the graph:

binary $n \times n$ matrix of $(0, 1)$ adjacency data $\mathbf{A} = (a_{ij})$, where a_{ij} is 1 if there is a path from edge i to edge j and otherwise $a_{ij} = 0$.

- sparse `(indx,vert)` input form: the $n+1$ vector `indx` gives the index bounds for the origin and end of the path (defining the i), and the m vector `vert` lists the indices of the target edge (defining j). Note, there must be `indx[1]=0` and `indx[n+1] = m`, and the entries must be monoton increasing integers.

`cost` (distances) of the node vertices must be compatible with the form of the first input argument and is

1. either an $n \times n$ matrix of costs (distances) whenever the first input argument is a binary adjacency matrix
- or a numeric m vector of integer costs (distances) c among the edges, whenever the first and second input arguments are vectors

2. `brack` is a 2-vector ($K = 1$) or an $K \times 2$ matrix with the numbers of starting and end nodes for which the shortest path should be computed.
3. a numeric vector of options:

   [**1**] amount of printed output, default=0

   [**2**] method (algorithm), currently unused, default=0

Output: **gof** lists some scalar results:

1. error return code
2. computing time in seconds

path for each starting node it lists the ordered path of the nearest neighbor nodes

len is a vector or matrix which has K rows corresponding to the **ordered** rows of `brack` and contains the length of path between starting and end node, as well as the list of nodes inbetween.

Restrictions: 1. All the input data is integer. String, complex, and missing values are not permitted.

2. The dimensions of input data must be compatible.

Relationships: mstgra(), mstdis(),

Examples: 1. Example of TOMS 562, $n = 20, m = 60$:

```
indx = [ 0   5  11  15  19  24  27  30  32  37
        40  42  44  46  50  52  53  55  57  59  60 ];
vert = [ 2   4   3   5   6  10   7   6   1  14
        15   6   1  11   8   7  12   1   9   1
         9   8  13  18  14   2   3  16   2   4
         3   5   4   5  18   1  12  15  14   2
         3  19   4   7   5  17  10  19   6   2
        10   2   7  13  20   9   5  14  11  17 ];
cost = [ 8   8   8   9  14   9   9  10   8  14
        14   8   8   8  10   9   8   8   9   9
         9   8   9  14   9  10   8   8   9   9
        10   8   9   9   9  14  14   8   8   9
         8  20   8  14   9   9   8  10   9  14
         8  14   8   9   8   9  14  10  20   8 ];


optn = 3;
brack1 = [ 20 # 1, 1:20 ]‘;
< gof,dist,len > = splnet(indx,vert,cost,brack1,optn);
print "GOF=",gof;
print "Dist1=", dist;
print "Length=", len;


Distances from Node 1
---------------------

  1 :         0        8        8        8        9
  6 :        14       17       17       17       17
 11 :        16       16       18       22       22
 16 :        25       27       23       32       35

Path from Node 1 to Node 1 Length=0
-----------------------------------

  1 :         1

Path from Node 1 to Node 2 Length=8
-----------------------------------

  1 :         1        2
```

```
Path from Node 1 to Node 3 Length=8
-----------------------------------

  1 :         1        3

Path from Node 1 to Node 4 Length=8
-----------------------------------

  1 :         1        4

Path from Node 1 to Node 5 Length=9
-----------------------------------

  1 :         1        5
.................................................


Path from Node 1 to Node 11 Length=16
-------------------------------------

  1 :         1        3       11

Path from Node 1 to Node 12 Length=16
-------------------------------------

  1 :         1        4       12

Path from Node 1 to Node 13 Length=18
-------------------------------------

  1 :         1        5       13
.................................................


Path from Node 1 to Node 19 Length=32
-------------------------------------

  1 :         1        2       14       19

Path from Node 1 to Node 20 Length=35
-------------------------------------

  1 :         1        5       13       17       20
```

```
  optn = 3;
  brack1 = [ 20 # 1, 1:20 ]‘;
  brack2 = [ 6 # 3, 20 15 10 5 6 9 ]‘;
  brack3 = [ 11 2 ];
  brack4 = [ 5 # 20, 16 8 4 2 1 ]‘;
  brack = brack1 |> brack2 |> brack3 |> brack4;
  < gof,dist,len > = splnet(indx,vert,cost,brack,optn);
  print "GOF=",gof;
  print "Dist=", dist;
  print "Length=", len;
```

GOF=

	1
Return_Code	0.00000
Total_Time	0.00100

Dist=

	Start_Node	Node_1	Node_2	Node_3
1	1	0	8	8
2	3	8	16	0
3	11	16	24	8
4	20	35	43	43

	Node_4	Node_5	Node_6	Node_7
1	8	9	14	17
2	16	17	8	25
3	24	25	16	33
4	43	26	49	52

	Node_8	Node_9	Node_10	Node_11
1	17	17	17	16
2	10	25	25	8
3	18	33	33	0
4	34	35	52	51

	Node_12	Node_13	Node_14	Node_15
1	16	18	22	22
2	24	26	17	30

3	32	34	25	38
4	49	17	57	57

	Node_16	Node_17	Node_18	Node_19
1	25	27	23	32
2	33	35	31	27
3	41	43	39	20
4	60	8	40	67

	Node_20
1	35
2	43
3	51
4	0

Length=

	Start_Node	End_Node	Path_Length	Node_1
1	1	1	0	1
2	1	2	8	1
3	1	3	8	1
4	1	4	8	1
5	1	5	9	1
6	1	6	14	1
7	1	7	17	1
8	1	8	17	1
9	1	9	17	1
10	1	10	17	1
11	1	11	16	1
12	1	12	16	1
13	1	13	18	1
14	1	14	22	1
15	1	15	22	1
16	1	16	25	1
17	1	17	27	1
18	1	18	23	1
19	1	19	32	1
20	1	20	35	1
21	3	20	43	3
22	3	15	30	3
23	3	10	25	3
24	3	5	17	3

25	3	6	8	3
26	3	9	25	3
27	11	2	24	11
28	20	16	60	20
29	20	8	34	20
30	20	4	43	20
31	20	2	43	20
32	20	1	35	20

	Node_2	Node_3	Node_4	Node_5
1	0	0	0	0
2	2	0	0	0
3	3	0	0	0
4	4	0	0	0
5	5	0	0	0
6	6	0	0	0
7	2	7	0	0
8	5	8	0	0
9	4	9	0	0
10	2	10	0	0
11	3	11	0	0
12	4	12	0	0
13	5	13	0	0
14	2	14	0	0
15	2	15	0	0
16	2	7	16	0
17	5	13	17	0
18	5	18	0	0
19	2	14	19	0
20	5	13	17	20
21	1	5	13	17
22	1	2	15	0
23	6	14	10	0
24	1	5	0	0
25	6	0	0	0
26	1	4	9	0
27	3	1	2	0
28	17	13	5	1
29	17	13	5	8
30	17	13	5	1
31	17	13	5	1
32	17	13	5	1

	Node_6	Node_7	Node_8

1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	0	0	0
13	0	0	0
14	0	0	0
15	0	0	0
16	0	0	0
17	0	0	0
18	0	0	0
19	0	0	0
20	0	0	0
21	20	0	0
22	0	0	0
23	0	0	0
24	0	0	0
25	0	0	0
26	0	0	0
27	0	0	0
28	2	7	16
29	0	0	0
30	4	0	0
31	2	0	0
32	0	0	0

5.11 The `tsp()` Function

< length,tour,dist > = tsp(meth,norm,data<,optn<,intour>>)

Purpose: The `tsp()` function is trying to find the optimal order of n cities passed through at a travel with minimum length of way (or minimum cost). For a long time it is known that the problem is NP-hard, i.e. cannot be solved in polynomial time. Both, the symmetric and the The results can be printed and returned in various forms:

- the cities of the tour in integer or string form
- the ordering of the cities in the output can be either in route form or in input form (only with integers representing the stop number).

Some of the algorithms can deal with unsymmetric square matrices. However, any unsymmetric problem can be reformulated as a symetric by including the same number n of dummy cities into the tour and solving the problem in $2n$ dimensions, see Jonkers & Wolgenant (1983). This can be done implicitly by specifying the `"maksym"` option.

Using *Concorde* algorithm for other than **academic research** purposes needs the specific agreement of the developers. Please contact Prof. William B. Cook, of the University of Waterloo in Canada, at `bico@uwaterloo.ca` if you want to use the CMAT function `tsp()` with the `"conc"` option for anything else than **academic research**. The *Concorde* code requires an interface to an LP solver. Dr. John Forrest was so friendly to provide an interface between *Clp* (in COIN-OR) and *Concorde*.

Note, that for very large graph data not the shortest route must be found since the computational problem is NP hard.

Input: meth is either a string scalar or a q vector of strings specifying the algorithm(s) which should be used. The following methods are available:

"conc" symmetric *Concorde* algorithm (Appelgate et al., 2000, 2006): can be used for academic research only

"lker" symmetric Lin-Kernighan algorithm (Lin & Kernighan, 1965; Appelgate et al., 2003) can be used for academic research only

"lkex" symmetric and asymmetric extended Lin-Kernighan algorithm based on work by Keld Helsgaun (2000)

"nins" nearest neighbor-insertion (similar to CRAN) (Rosenkrantz, Stearns,& Lewis, 1977),

"fins" farthest neighbor-insertion (similar to CRAN) (Rosenkrantz, Stearns ,& Lewis, 1977),

"cins" cheapest neighbor-insertion (similar to CRAN) (Rosenkrantz, Stearns,& Lewis, 1977),

"ains" random neighbor-insertion (similar to CRAN) (Rosenkrantz, Stearns,& Lewis, 1977),

"nnei" nearest neighbor attachment (similar to CRAN) (Rosenkrantz, Stearns,& Lewis, 1977),

"rnne" repetitive nearest neighbor attachment (similar to CRAN) (Rosenkrantz, Stearns,& Lewis, 1977),

"twoo" two-stage optimization (similar to CRAN) (Croes, 1958; Lin, 1965)

"brab" branch-and-bound optimization by Carpaneto et al. (1995) (note, for some applications that method seems to have some serious problems)

norm must be a string scalar defining the formula for computing distances; one of the following choices can be made

spec	**formula**	dimensions
"matr"	data is distance matrix (default)	any
"eucl"	Euclidean distances	2 and 3
"city"	$L1$ norm (City block) distances	2 and 3
"maxn"	Maximum norm ($L_{\inf}$) distances	2 and 3
"geog"	Geographic norm distances	2
"geom"	Geometric norm distances	2
"crys"	Crystal norm distances	3

data there are two ways to specify the data:

- only for `norm="matr"`: must be numeric square matrix, preferrably, but not necessarily symmetric, some algorithms will work with unsymmetric but square distance matrices
- all other norms: must be a numeric $n \times p$ matrix of coordinates of n points in p dimensions, currently restricted to $p = 2$ and $p = 3$.

optn This argument must be 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.

intour must be a numeric n vector of a starting tour, the values must be a permutation of $1, \ldots, n$.

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

Option Name	Column 2	Meaning
"inpord"		output in the input order of the cities
"makes"		make unsymmetric problem symmetric
"nopr"		no printed output (is default)
"pdis"		print scaled (integer) distance matrix
"prec"	int	power of 10 for scaling distances
"print"	int	amount of printed output (larger is more, default=0)
"repl"	int	number of replications for ... algorithm
"retstr"		output of tour in string form
"seed"	int	seed for random generator
"start"	int	start city of the tour

Output: **length** is either a 2 vector or $2 \times q$ matrix containing the length of tour(s) in its first column and the computer time for each of the q methods

tour is a K *timesn* vector (for $K = 1$) or matrix of either int or string data showing the order of cities on the route

dist is a K *timesn* vector (for $K = 1$) or matrix of real values of the distances from the corresponding cities of the `tour` output to the next stop.

Restrictions: 1. The distance matrix may not contain any string, complex, or missing data.

2. If a starting tour `intour` is specified, its dimension must be compatible with that of the distance matrix.

Relationships: lp(), lpassign(), lptrans()

Examples: 1. Distances between ten German cities:

```
print "Ingwer Borg: MDS Book: Ten German cities";
dst = [    0,
         279   0,
         120 171  0,
         278 105 158  0,
         226  99 106  53  0,
         178 120  58 102  49  0,
         152 191  59 142 101  73  0,
         124 205 125 250 199 155 184  0,
         135 153  78 189 138  95 136  61  0,
          74 206  63 216 163 114 116  76  64  0 ];
 cnam = [" Basel Berlin Frankfurt Hamburg Hannover "] ->
        [" Kassel Koeln Muenchen Nuernberg Stuttgart "];
 dst = cname(dst,cnam);
```

```
meth = [" nins fins cins ains nnei rnne twoo "];
optn = [ "start"      1 ,
         "seed"   12345 ,
         "print"      4 ];
< len,tour,dist > = tsp(meth,"matr",dst,optn);
print "Length=", len;
print "Tour=", tour;
print "Distances=",dist;
```

```
         Route of the Travel: Method=NINS Length=880
                    (Output in Tour Order)
         ****************************************

              1     Basel     1  120.000000
              2 Frankfurt     3  59.0000000
              3     Koeln     7  142.000000
              4   Hamburg     4  105.000000
              5    Berlin     2  99.0000000
              6  Hannover     5  49.0000000
              7    Kassel     6  95.0000000
              8 Nuernberg     9  61.0000000
              9  Muenchen     8  76.0000000
             10 Stuttgart    10  74.0000000


         Route of the Travel: Method=FINS Length=823
                    (Output in Tour Order)
         ****************************************

              1     Basel     1  120.000000
              2 Frankfurt     3  59.0000000
              3     Koeln     7  73.0000000
              4    Kassel     6  49.0000000
              5  Hannover     5  53.0000000
              6   Hamburg     4  105.000000
              7    Berlin     2  153.000000
              8 Nuernberg     9  61.0000000
              9  Muenchen     8  76.0000000
             10 Stuttgart    10  74.0000000


         Route of the Travel: Method=CINS Length=880
                    (Output in Tour Order)
         ****************************************

              1     Basel     1  120.000000
```

```
     2 Frankfurt     3  59.0000000
     3     Koeln     7  142.000000
     4   Hamburg     4  105.000000
     5    Berlin     2  99.0000000
     6  Hannover     5  49.0000000
     7    Kassel     6  95.0000000
     8 Nuernberg     9  61.0000000
     9  Muenchen     8  76.0000000
    10 Stuttgart    10  74.0000000


Route of the Travel: Method=AINS Length=814
           (Output in Tour Order)
*******************************************

     1     Basel     1  124.000000
     2  Muenchen     8  61.0000000
     3 Nuernberg     9  153.000000
     4    Berlin     2  105.000000
     5   Hamburg     4  53.0000000
     6  Hannover     5  49.0000000
     7    Kassel     6  73.0000000
     8     Koeln     7  59.0000000
     9 Frankfurt     3  63.0000000
    10 Stuttgart    10  74.0000000


Route of the Travel: Method=NNEI Length=1120
           (Output in Tour Order)
*******************************************

     1     Basel     1  279.000000
     2    Berlin     2  205.000000
     3  Muenchen     8  61.0000000
     4 Nuernberg     9  136.000000
     5     Koeln     7  142.000000
     6   Hamburg     4  53.0000000
     7  Hannover     5  49.0000000
     8    Kassel     6  58.0000000
     9 Frankfurt     3  63.0000000
    10 Stuttgart    10  74.0000000


Route of the Travel: Method=RNNE Length=968
           (Output in Tour Order)
*******************************************
```

```
 1     Basel     1  279.000000
 2    Berlin     2  105.000000
 3   Hamburg     4  53.0000000
 4  Hannover     5  49.0000000
 5    Kassel     6  58.0000000
 6 Frankfurt     3  59.0000000
 7     Koeln     7  116.000000
 8 Stuttgart    10  64.0000000
 9 Nuernberg     9  61.0000000
10  Muenchen     8  124.000000


Route of the Travel: Method=TWOO Length=814
          (Output in Tour Order)
*****************************************

 1     Basel     1  74.0000000
 2 Stuttgart    10  63.0000000
 3 Frankfurt     3  59.0000000
 4     Koeln     7  73.0000000
 5    Kassel     6  49.0000000
 6  Hannover     5  53.0000000
 7   Hamburg     4  105.000000
 8    Berlin     2  153.000000
 9 Nuernberg     9  61.0000000
10  Muenchen     8  124.000000
```

Only Ains and 2-Opt find the optimal solution:

```
Length=
  LOW |  TourLength    CompTime
------------------------------
 NINS |      880.00           0
 FINS |      823.00     0.00000
 CINS |      880.00     0.00000
 AINS |      814.00     0.00000
 NNEI |      1120.0     0.00000
 RNNE |      968.00     0.00000
 TWOO |      814.00     0.00000
```

2. Distances between ten US cities (for data see SAS PROC LP):

```
  cost = [   0 ,                                    /* Atlanta */
            58   0 ,                                /* Chicago */
           121  92   0 ,                            /* Denver */
            70  94  87   0 ,                        /* Houston */
```

```
          193 174  83 137   0 ,                      /* Los Angeles */
           60 118 172  96 233   0 ,                  /* Miami */
           74  71 163 142 245 109   0 ,              /* New York */
          213 185  94 164  34 259 257   0 ,          /* San Francisco */
          218 173 102 189  95 273 240  67   0 ,      /* Seattle */
           54  59 149 122 230  92  20 244 232   0 ]; /* Washington */


rnam = [" Atlanta Chicago Denver Houston Los_Angeles "] ->
       [" Miami New_York San_Fran Seattle Washington "];
cost = rname(cost,rnam);
cost = cname(cost,rnam); /* print "Cost=", cost; */


optn = [ "start"      1 ,
         "seed"   12345 ,
         "print"      4 ];
< len,tour,dist > = tsp("conc","matr",cost,optn);
```

The code of *Concorde* prints a few messages, which, however, may be ignored. This output appears to show up only for small n applications. It is not recommended to run the *Concorde* method for applications with $n \leq 20$.

```
  LP Value  1: 660.500000  (0.00 seconds)
  LP Value  2: 733.000000  (0.00 seconds)
WARNING: No dual change in basis finding code
Did not find a basic optimal solution: rval=2
Fractional matching routine failed
Warning: restarting running timer: Miscellaneous
No warmstart, stumbling on anyway
```

With *Linkern* and *Concorde* we obtain identical results:

```
          Route of the Travel: Method=CONC Length=733
                    (Output in Tour Order)
          ******************************************

               1     Atlanta     1  60.0000000
               2       Miami     6  96.0000000
               3     Houston     4  137.000000
               4 Los_Angeles     5  34.0000000
               5    San_Fran     8  67.0000000
               6     Seattle     9  102.000000
               7      Denver     3  92.0000000
               8     Chicago     2  71.0000000
```

```
  9    New_York     7  20.0000000
 10  Washington    10  54.0000000


      Time for Initialization: 0
    Total Processing Time: 0.131
```

Even some of the heuristic methods obtain the optimal result:

```
  LOW |  TourLength    CompTime
------------------------------
 NINS |      745.00           0
 FINS |      733.00     0.00000
 CINS |      745.00     0.00000
 AINS |      733.00     0.00000
 NNEI |      848.00     0.00000
 RNNE |      781.00     0.00000
 TWOO |      733.00     0.00000
```

3. Distances between $n = 50$ US cities (see CRAN function `tsp`):

```
   print "CRAN: Distances among 50 Cities of the USA";
   options NOECHO;
%inc "..\\tdata\\usca50.dat";
   options ECHO;


   meth = [" conc lker lkex "];
   optn = [ "start"      1 ,
            "seed"   12345 ,
            "print"      4 ];
   < gof,tour,dist > = tsp(meth,"matr",usca50,optn);
```

```
      |  TourLength    CompTime
------------------------------
 CONC |     14497.0    0.533000
 LINK |     14497.0    0.105000
 LKEX |     14497.0    1.0e-003
```

Again, we obtain the same result for *Concorde* and *Linkern*:

```
          Route of the Travel: Method=CONC Length=14497
                      (Output in Tour Order)
          ******************************************
```

```
 1       Abilene, TX     1  226.000000
 2      Amarillo, TX     7  272.000000
 3   Albuquerque, NM     4  697.000000
 4   Bakersfield, CA    17  249.000000
 5      Berkeley, CA    26  162.000000
 6   Carson City, NV    48  358.000000
 7         Boise, ID    33  244.000000
 8         Butte, MT    44  194.000000
 9      Billings, MT    27  446.000000
10       Calgary, AB    45  409.000000
.....................................
40        Canton, OH    47  173.000000
41       Ashland, KY    11  231.000000
42 Bowling Green, KY    35  236.000000
43     Asheville, NC    10  150.000000
44       Augusta, GA    14  140.000000
45       Atlanta, GA    12  139.000000
46    Birmingham, AL    30  248.000000
47        Biloxi, MS    28  135.000000
48   Baton Rouge, LA    20  177.000000
49      Beaumont, TX    23  217.000000
50        Austin, TX    16  190.000000


   Time for Initialization: 0.001
    Total Processing Time: 0.551
```

For the heuristic methods we obtain tours with the following lengths:

```
      |  TourLength    CompTime
------------------------------
 NINS |     17711.0    1.0e-003
 FINS |     15070.0    0.000000
 CINS |     15713.0        0.00
 AINS |     16481.0        0.00
 NNEI |     19223.0    0.000000
 RNNE |     16350.0        0.02
 TWOO |     14811.0        0.00
```

4. Distances between $n = 312$ US cities (see CRAN function `tsp`):

```
   options NOECHO;
%inc "..\\tdata\\usca312.dat";
   options ECHO;
```

```
meth = [" conc lker lkex "];
optn = [ "start"      1 ,
         "seed"   12345 ,
         "print"      4 ];
< len,tour,dist > = tsp(meth,"matr",usca312,optn);
```

Again, we obtain the same result for *Concorde* and *Linkern*:

```
          Route of the Travel: Method=CONC Length=38508
                     (Output in Tour Order)
          ********************************************

              1          Abilene, TX     1  141.000000
              2         Ft Worth, TX    97  30.0000000
              3           Dallas, TX    69  87.0000000
              4             Waco, TX   292  95.0000000
              5           Austin, TX    16  73.0000000
              6      San Antonio, TX   238  146.000000
              7           Laredo, TX   143  130.000000
              8   Corpus Christi, TX    68  183.000000
              9          Houston, TX   121  46.0000000
             10        Galveston, TX   103  66.0000000
             ......................................
            300      Saint Louis, MO   232  117.000000
            301         Columbia, MO    63  130.000000
            302      Springfield, MO   264  67.0000000
            303           Joplin, MO   130  117.000000
            304         Ft Smith, AR    95  103.000000
            305            Tulsa, OK   287  97.0000000
            306    Oklahoma City, OK   188  67.0000000
            307             Enid, OK    85  94.0000000
            308          Wichita, KS   300  80.0000000
            309           Salina, KS   236  150.000000
            310       Dodge City, KS    77  201.000000
            311         Amarillo, TX     7  113.000000
            312           Lubbock, TX   156  145.000000
```

However, the computation time differs remarkably betwen *Concorde* and *Linkern*:

```
      |  TourLength    CompTime
------------------------------
 CONC |     38918.0     1155.79
 LINK |     38508.0     1.58000
```

```
 LKEX |     38508.0        0.00
```

For the heuristic methods we obtain tours with the following lengths:

```
      |  TourLength   CompTime
-------------------------------
 NINS |   45194.000  0.1070000
 FINS |   42348.000  0.1070000
 CINS |   45334.000  0.3960000
 AINS |   41773.000  0.0050000
 NNEI |   77591.000  0.1030000
 RNNE |   72088.000  32.653000
 TWOO |   41219.000  0.6240000
```

5. Twodimensional Coordinates of $n = 2392$ Points (Padberg & Rinaldi):

```
   options NOECHO;
%inc "..\\tdata\\pr2392.dat";
   options ECHO;
```

Note, for saving computer time, we specify the optimal length of the tour with $d = 378032$:

```
   optn = [ "start"          1 ,
            "seed"       12345 ,
            "movetyp"        5 ,
            "patcha"         2 ,
            "patchc"         3 ,
            "optim"     378032 ,
            "runs"          10 ,
            "print"          2 ];
   < gof,tour,dist > = tsp("lkex","eucl",pr2392,optn);
   print "Length=", gof;
```

The `CONC` and `LKER` methods take more than oine day to accomplish. However, we can compare the result of `LKEX` with that of some of the heuristic methods:

```
Length=
      |  TourLength    CompTime
------------------------------
 NINS |    470923.0     227.620
 FINS |    429580.3     242.284
 CINS |    460046.0     668.477
```

```
AINS |    436547.4     1.13900
NNEI |    756374.1     232.269
RNNE |      takes too long
TWOO |    428431.6     2735.81
LKEX |    378062.8     206.545
LKER |      takes too long
CONC |      takes too long
```

6. $n = 100$ Points equally distributed on a circle with radius $r = 10$:
   This following data are not suitable for methods using integer distances, like *Linkern* and *Concorde* since the radius of 10. would be too small to generate distances greater 0 for points which are next to each other on the circumference of the circle. For using int distances, either the radius is increased or the data matrix must be scaled. Please note also, for an input distance matrix, the `"prec"` option could be used for scaling before it is casted to integer values.

```
n = 100; srand(12345);
print "Points on a Circle:", n;

 rad = 10.; xy = cons(n,2,.);
 pi2 = 2. * macon("pi");
 th = theta = pi2 / n;
 for (i = 1; i <= n; i++) {
   xy[i,1] = rad * sin(th);
   xy[i,2] = rad * cos(th);
   th += theta;
 }
 print xy[1:10,];
```

   Now randomly permute the points:

```
/* Permute data and names */
ind = randperm(n); /* print ind; */
strt = loc(ind,"eq",1); print strt,ind[strt];
xy2 = xy[ind,];
rnam = [ "Point_1":n ]; /* print rnam; */
rnam = rnam[ind];
xy2 = rname(xy2,rnam);
```

   Try the heuristic methods:

```
meth = [" nins fins cins ains nnei rnne twoo "];
optn = [ "start"   strt ,
         "repl"      10 ,
```

```
          "seed"   12345 ,
          "print"      4 ];
   < len,tour,dist > = tsp(meth,"eucl",xy2,optn);
   print "Length=", len;
   print "Tour=", tour;
   print "Distances=",dist;
```

All methods obtain the perfect solution since each two neighbour points have the distance $d = 0.62821518$, the optimal distance should be $n * d$:

```
      |  TourLength    CompTime
------------------------------
 NINS |      6282.2        0.06
 FINS |      6282.2        0.06
 CINS |      6282.2     0.15300
 AINS |      6282.2        0.01
 NNEI |      6282.2        0.05
 RNNE |      6282.2      5.1400
 TWOO |      6282.2     0.51900
```

```
Tour=
           |  NINS  FINS  CINS  AINS  NNEI  RNNE  TWOO
-----------------------------------------------------
  Stop_001 |    86    86    86    86    86    86    86
  Stop_002 |    34    34    34    34    34    34    34
  Stop_003 |    99    99    99    99    99    99    99
  Stop_004 |    29    29    29    29    29    29    29
  Stop_005 |    69    69    69    69    69    69    69
  .....................................................
  Stop_095 |    68    68    68    68    68    68    68
  Stop_096 |    43    43    43    43    43    43    43
  Stop_097 |    65    65    65    65    65    65    65
  Stop_098 |    38    38    38    38    38    38    38
  Stop_099 |    84    84    84    84    84    84    84
  Stop_100 |     6     6     6     6     6     6     6
```

```
Distances=
           |      NINS      FINS      CINS      AINS
-----------------------------------------------------
  Stop_001 |   0.62822   0.62822   0.62822   0.62822
  Stop_002 |   0.62822   0.62822   0.62822   0.62822
  Stop_003 |   0.62822   0.62822   0.62822   0.62822
  Stop_004 |   0.62822   0.62822   0.62822   0.62822
```

Stop_005	0.62822	0.62822	0.62822	0.62822
...				
Stop_095	0.62822	0.62822	0.62822	0.62822
Stop_096	0.62822	0.62822	0.62822	0.62822
Stop_097	0.62822	0.62822	0.62822	0.62822
Stop_098	0.62822	0.62822	0.62822	0.62822
Stop_099	0.62822	0.62822	0.62822	0.62822
Stop_100	0.62822	0.62822	0.62822	0.62822

	NNEI	RNNE	TWOO
Stop_001	0.62822	0.62822	0.62822
Stop_002	0.62822	0.62822	0.62822
Stop_003	0.62822	0.62822	0.62822
Stop_004	0.62822	0.62822	0.62822
Stop_005	0.62822	0.62822	0.62822
...			
Stop_095	0.62822	0.62822	0.62822
Stop_096	0.62822	0.62822	0.62822
Stop_097	0.62822	0.62822	0.62822
Stop_098	0.62822	0.62822	0.62822
Stop_099	0.62822	0.62822	0.62822
Stop_100	0.62822	0.62822	0.62822

6 Illustrations

6.1 Comparing Computer Time for `lpassign()` Algorithms

CMAT compiled in *Debug* mode takes in seconds:

Size	LPS	HUN	JOV	BOT
100	3.52	0.002	0.001	0.001
200	46.73	0.008	0.004	0.002
300	342.66	0.024	0.010	0.006
400	8335.85	0.051	0.024	0.009
500	.	0.107	0.028	0.014
1000	.	0.657	0.136	0.063
1500	.	2.027	0.453	0.128
2000	.	4.639	0.830	0.222
2500	.	7.720	1.594	0.369
3000	.	12.988	2.756	0.595
3500	.	21.314	3.839	0.801
4000	.	31.108	5.226	1.110
4500	.	44.079	7.535	1.429
5000	.	59.170	8.090	1.700

CMAT compiled in *Release* mode:

Size	LPS	HUN	JOV	BOT
100	0.655	0	0	0
200	7.878	0	0	0
300	30.461	0.015	0	0
400	799.565	0.015	0.016	0
500	.	0.047	0	0
1000	.	0.249	0.032	0.031
1500	.	0.780	0.109	0.047
2000	.	1.903	0.230	0.094
2500	.	3.729	0.375	0.125
3000	.	5.881	0.561	0.234
3500	.	9.080	1.029	0.296
4000	.	12.963	1.185	0.374
4500	.	18.267	1.747	0.484
5000	.	24.931	1.935	0.608

7 The Bibliography

References

[1] Appelgate, D., Bixby, R., Chvatal, V. & Cook, W.(2006), *Concorde TSP Solver*, `http://www.tsp.gatech.edu/concorde`.

[2] Appelgate, D., Bixby, R., Chvatal, V. & Cook, W.(2000), "TSP cuts which do not conform to the template paradigm" in M. Junger & D. Naddef (eds.): *Computational Combinatorial Optimization, Optimal or Provably Near-Optimal Solutions*, *Lecture Notes in Computer Science*, Vol. 2241, pp. 261-304, London: Springer Verlag.

[3] Appelgate, D., Cook, W. & Rohe, A.(2003), "Chained Lin-Kernighan for large traveling salesman problems", *INFORMS Journal on Computing*, **15**, 82-92.

[4] Berkelaar, M., Eikland, K., & Notebaert, P. (2004), *lp_solve (alternatively lpsolve)*, Open source (Mixed-Integer) Linear Programming system, Version 5.5.

[5] Borchers, H. W. (2013), Package *adagio*, in CRAN.

[6] Carpaneto, G., Dell'Amico, M., & Toth, P. (1995), "A branch-and-bound algorithm for large scale asymmetric traveling salesman problems", Algorithm 750, *Transactions on Mathematical Software*, **21**, 410-415.

[7] Cheung, T. Y. (1980), "Multifacility location problem with rectilinear distance by the minimum cut approach", *ACM Trans. Math. Software*, **6**, 387-390.

[8] Croes, G. A. (1958), "A method for solving traveling-salesman problems", *Operations Research*, **6**, 791-812.

[9] Czyzyk, J., Mehrotra, S., Wagner, M. & Wright, S.J. (1997) "PCx User Guide (Version 1.1)", Technical Report OTC 96/01, Office of Computational and Technology Research, US Dept. of Energy.

[10] Du, D.-Z. & Pardalos, P. M. (1993), *Network Optimization Problems - Algorithms, Applications, and Complexity*, Singapore: World Scientific.

[11] Ecker, J. G. & Kupferschid, M. (1988), *Introduction to Operations Research*, (Reprint of 1991), Malabar, FL: Krieger Publishing Company.

[12] Fredman, M.L., Johnson, D.S., McGeoch, L.A. & Ostheimer, G. (1995), "Data structures for Traveling Salesmen", *J. Algorithms*, **18**, 432-479.

[13] Hahsler, M. & Hornik, K. (2013), "TSP - Infrastructure for the traveling salesperson problem", Technical Report, CRAN.

[14] Haymond, R.E., Jarvis, J.P. & Shier, D.R. (2013), "Algorithm 613: Minimum spanning tree for moderate integer weights", *ACM TOMS*, **10**, 108-110.

[15] Helsgaun, K. (2000), "An effective implementation of the Lin-Kernighan Traveling Salesman Heuristic", *European Journal of Operational Research*, **126**, 106-130.

[16] Helsgaun, K. (2006), "An effective implementation of K-opt moves for the Lin-Kernighan TSP Heuristic", *Datalogiske Skrifter*, **109**, Roskilde University, Denmark.

[17] Jonker, R. & Volgenant, A. (1983), "Transforming asymmetric into symmetric traveling salesman problems", *Operations Reseatrch Letters*, **2**, 161-163.

[18] Jonker, R. & Volgenant, A. (1987), "A shortest augmenting path algorithm for dense and sparse linear assignment problems", *Computing*, **38**, 325-340.

[19] Kellerer, H., Pferschy, U., & Pisinger, D. (2004), *Knapsack Problems*, Berlin, Heidelberg: Springer Verlag.

[20] Lin, S. (1965), "Computer solutions of the traveling-salesman problem", *Bell Systems Technology*, **44**, 2245-2269.

[21] Lin, S. & Kernighan, B. (1973), "An effective heuristic algorithm for the traveling-salesman problem", *Operations Research*, **21**, 498-516.

[22] Martello, S. (1983), "Algorithm 595: An enumerative algorithm for finding Hamiltonian circuits in a directed graph", *ACM TOMS*, **9**, 131-138.

[23] Martello, S. & Toth, P. (1990), *Knapsack Problems: Algorithms and Computer Implementations*, New York: John Wiley & Sons.

[24] Nemhauser, G. L. & Wolsey, L. A. (1988), *Integer and Combinatorial Optimization*, New York: John Wiley & Sons.

[25] Pape, U. (1980), "Algorithm 562: Shortest Path Lengths", *ACM TOMS*, **6**, 450-455.

[26] Powell, J.M.D. (2003), "On the use of quadratic models in unconstrained minimization without derivatives", Report DAMTP 2003/NA03, University of Cambridge.

[27] Powell, J.M.D. (2014), "On fast trust region methods for quadratic models with linear constraints", Report DAMTP 2014/NA02, University of Cambridge.

[28] Prim, R.C. (1957), "Shortest connection networks and some generalizations", *Bell System Tech. Journal*, **36**, 1389-1401.

[29] Reinelt, G. (1991), "TSPLIB - A Traveling Salesman Problem Library", *ORSA J. Comput.*, **3-4**, 376-385.

[30] Rosenkrantz, D. J., Stearns, R.E. & Lewis, P. M. (1977), "An analysis of several heuristics for the traveling salesman problem", *SIAM Journal on Computing*, **6**, 563-581.

[31] Sasieni, M., Yaspan, A., & Friedman, L. (1968), *Methoden und Probleme der Unternehmensforschung*, Berlin: Verlag Die Wirtschaft

[32] Simonetti, N. (1998), "Subroutines for dynamic program for the Traveling Salesman Problem", `www.contrib.andrew.cmu.edu/ neils/tsp/index.html`

[33] Volgenant, A. & van den Hout, W. B. (1990), "TSP1 and TSP2 - Symmetric traveling salesman problem for personal computers", Technical Report with Borland Pascal implementation.

[34] Williams, E. (2014), "Aviation Formulary V1.46", `http://williams.best.vwh.net/avform.htm`