Functions Reference v.1.0.0

[**File: import.m**](#_x9p6f0x51kmm)[**2**](#_x9p6f0x51kmm)

[import](#_cp6ma8rm3pf2) [2](#_cp6ma8rm3pf2)

[**File: payoff.m**](#_gpmtcz4rar38)[**3**](#_gpmtcz4rar38)

[Cx](#_fepx2hmkizbz) [3](#_fepx2hmkizbz)

[payoff](#_sgoel7gkpie2) [3](#_sgoel7gkpie2)

[payoffDM](#_d1phr68acut5) [4](#_d1phr68acut5)

[CpayoffMatrix](#_d8vb92qtnfus) [4](#_d8vb92qtnfus)

[Ctotalpayoff](#_osky0e79q71v) [4](#_osky0e79q71v)

[**File: matching.m**](#_cvc0dkixvle0)[**5**](#_cvc0dkixvle0)

[generateAssignmentMatrix](#_i82z3ctshmre) [5](#_i82z3ctshmre)

[CmatchMatrix](#_ms1hxzth03pr) [6](#_ms1hxzth03pr)

[Cmates](#_ei0a11staaxf) [6](#_ei0a11staaxf)

[Cmate](#_v3qqamqso7y0) [7](#_v3qqamqso7y0)

[quotas](#_qqcvat7916cu) [7](#_qqcvat7916cu)

[**File: inequalities.m**](#_sh3sqs7fq266)[**7**](#_sh3sqs7fq266)

[Cineqmembers](#_wy6stkxdmy92) [8](#_wy6stkxdmy92)

[Cinequalities](#_wqnwe8rykkj0) [8](#_wqnwe8rykkj0)

[**File: dataArray.m**](#_7ozwa1x3snxc)[**9**](#_7ozwa1x3snxc)

[CdataArray](#_7nbh7zsc22z0) [9](#_7nbh7zsc22z0)

[**File: objective.m**](#_unuthl1xspbg)[**10**](#_unuthl1xspbg)

[coefficient1](#_68i3pkpr4lh6) [10](#_68i3pkpr4lh6)

[objective](#_n0dh4rl5nz5f) [10](#_n0dh4rl5nz5f)

[**File: maximize.m**](#_te7kgdjcsdp5)[**11**](#_te7kgdjcsdp5)

[optimize](#_bb2js28jduky) [11](#_bb2js28jduky)

[maximize](#_d9mlx4e0zeff) [12](#_d9mlx4e0zeff)

[**File: confidence.m**](#_mspdsg6bb9nk)[**13**](#_mspdsg6bb9nk)

[generateRandomSubsample](#_qc7ho5v9n1mx) [13](#_qc7ho5v9n1mx)

[pointIdentifiedCR](#_gqoxq5y93e8s) [14](#_gqoxq5y93e8s)

# 

# 

The full code contains more functions than those exposed in this document. A more detailed reference will be added in the next document revision. The curious reader can explore these inside the .m files.

The library contains several files that group together related functionality

FIles: {"import.m", "export.m", "payoff.m", "modifydata.m", "matching.m", "inequalities.m", "dataArray.m", "objective.m", "PSO.m", "maximize.m", "confidence.m"}

## File: import.m

*{import (There are many more for various cases that involve speed increase)}*

### import

Need global variables: -

Creates global variables: -

Using functions: -

**Description**

{noM,no,u,noAttr}=import[filename,"stream"] to load an upstream.

{noM,no,d,noAttr}=import[filename,"stream"] to load a downstream.

imports a file .xls or .xlsx or a tab delimited file .dat that includes data corresponding to an upstream (u) or downstream (d).

Datafiles consist of rows in the form {noM,no,attr1,attr2,...,attrn,noAttr}.

In the case of .xls or .xlsx files multiple sheets are joined - for example if each market resides in its own excel sheet.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

{header,noM,noU,noD,noAttr,distanceMatrices,matchMatrix,mate}=import[filename\_,"precomp",printflag\_:False]

imports a file .xls or .xlsx or a tab delimited file .dat that includes precomputed matched data (distances between same attributes are already computed).

Datafiles consists of rows in the form {m,u,d,u1,d1,u2,d2,...,un,dn,matched (0 or 1)}.

In the case of .xls or .xlsx files multiple sheets are joined - for example if each market resides in its own excel sheet.

**Examples**

*Input:*

*Output:*

## File: payoff.m

*{Cx, payoff, payoffDM, payoffMatrix, Ctotalpayoff}*

### Cx

Need global variables: -

Creates global variables: x1,x2,x3,... (just create, not setting any value at this point)

Using functions: -

**Description**

Cx[n] creates a list of n variables named x1,x2,...,xn.

**Examples**

*Input:*

*Cx[3]*

*Output:*

*{x1,x2,x3}*

### payoff

Need global variables: -

Creates global variables: -

Using functions: -

**Description**

payoff[m,i,j] returns the payoff of i-upstream and j-upstream in the m-market.

It is used in the case of importing streams (not precomputed). It is assumed that u , d , and noAttr have been already assigned

**Examples**

*Input:*

*payoff[2,4,10] (\*2nd market, 4th upstream, 10th downstream)*

*Output:*

*17.2+7.3\*x1+11.8\*x2*

### payoffDM

Need global variables: -

Creates global variables: -

Using functions: -

**Description**

payoffDM[m,i,j] returns the payoff of i-upstream and j-upstream in the m-market.

It is used in the case of precomputed data. It is assumed that noAttr and distanceMatrices have been already assigned.

**Examples**

*Input:*

*Output:*

### CpayoffMatrix

Need global variables: noM, noU, noD, noAttr

Creates global variables: payoffMatrix

Using functions: Cx

**Description**

CpayoffMatrix[payoff(or payoffDM),noM\_,noU\_:noU,noD\_:noD,parallel\_:False] calculates and

assigns the payoffMatrix. In case of separated streams payoff is used and in the case of precomputed data payoffDM is used.

CpayoffMatrix[solution\_] substitutes the solution to all payoffMatrix's entries.

**Examples**

*Input:*

*Output:*

### Ctotalpayoff

Need global variables: -

Creates global variables: -

Using functions: -

**Description**

Ctotalpayoff[payoffobject,mates] calculates the total payoff that a specific match defined by mates returns. If the payoffobject is the payoff function then the head equals Symbol and the corresponding routine is used. If the payoffobject is the payoffMatrix then the head equals List and the corresponding routine is used.

**Examples**

*Input:*

*Output:*

## 

## File: matching.m

*{generateAssignmentMatrix, CmatchMatrix, Cmates, Cmate, quotas}*

### generateAssignmentMatrix

Need global variables: -

Creates global variables: -

Using functions: -

**Description**

generateAssignmentMatrix[payoffs\_,quotaU\_:1,quotaD\_:1,options\_\_\_?OptionQ]

generates the optimal assignment of matches from the given matrix of payoffs for each match. In an assignment matrix, each entry (i,j) is 1 if i and j are matched and 0 otherwise. The quota can be a number (the same for all streams) or a list that sets a specific quota per stream.

**Examples**

*Input:*

*Output:*

### CmatchMatrix

Need global variables: -

Creates global variables: matchMatrix

Using functions: generateAssignmentMatrix

**Description**

CmatchMatrix[payoffMatrix\_,quotaU\_:1,quotaD\_:1,p\_:False]

calculates and creates/updates the global variable 'matchMatrix'.

if p is set to 'False' creates the matchMatrix by running generateAssignmentMatrix routine for all markets.

if p is set to 'True' does the same only in parallel! NOT WORKING WITH VARIABLE QUOTAS - yet.

if p is set to an integer from 1 to 'number of Markets' then the p'th element of the matchMatrix is calculated.

**Examples**

*Input:*

*Output:*

### Cmates

Need global variables: -

Creates global variables: -

Using functions: -

**Description**

Cmates[matchMatrix] simplifies matchMatrix to a list of triples that define matches across all markets. Example : {{{1,1,3},{1,3,1},{1,3,2}},{{2,1,1},{2,2,1},{2,2,3},{2,3,2}}}. This is mainly used for the calculation of the total payoff - see Ctotalpayoff routine.

**Examples**

*Input:*

*Output:*

### Cmate

Need global variables: -

Creates global variables: -

Using functions: -

**Description**

Cmate[matchMatrix] simplifies matchMatrix to pairs of lists, one pair per market. The first part is the normal numbering and the second is the corespondance. Example : {{{{1},{2},{3}},{{3},{},{1,2}}},{{{1},{2},{3}},{{1},{1,3},{2}}}}. This is the prevailing way to express mates. This is feeded to the Cineqmembers routine.

**Examples**

*Input:*

*Output:*

### quotas

Need global variables: -

Creates global variables: -

Using functions: -

**Description**

quotas[matchMatrix] returns the list {quotaU,quotaD}. Each quota is defined for each stream.

**Examples**

*Input:*

*Output:*

## File: inequalities.m

*{Cineqmembers, Cinequalities}*

### Cineqmembers

**Description**

Cineqmembers[mate] generates all the members required to form the inequalities for many to many relationships defined by the mate. The produced list of lists of triples define also the way inequalities are formed. At this time there is only one way we have to create inequalities. CAUTION: ineqmembers is the largest object so it consumes a lot of memory. This is why we use MSEresources="Memory" when it is needed. Be carefull because then ineqmembers are erased after used for the dataArray calculation (only in the memory model).

**Examples**

*Input:*

Cineqmembers[ {{{{1}, {2}, {3}}, {{3}, {}, {1, 2}}}, {{{1}, {2}, {3}}, {{1}, {1,3}, {2}}}} ]

*Output:*

{{{{{1, 1, 3}}, {{1, 1, 3}, {1, 3, 1}, {1, 3, 2}}, {{1, 3, 1}, {1, 3, 2}}}, {{{1, 2, 3}}, {{1, 1, 1}, {1, 1, 2}, {1, 3, 3}}, {{1, 2, 1}, {1, 2, 2}}}}, {{{{2, 1, 1}, {2, 2, 1}, {2, 2, 3}}, {{2, 1, 1}, {2, 3, 2}}, {{2, 2, 1}, {2, 2, 3}, {2, 3, 2}}}, {{{2, 1, 1}, {2, 1, 3}, {2, 2, 1}}, {{2, 1, 2}, {2, 3, 1}}, {{2, 2, 2}, {2, 3, 1}, {2, 3, 3}}}}}

### Cinequalities

**Description**

Cinequalities[f,ineqmembers] apply properly the f function to ineqmembers to create inequalities. This routine is called by CdataArray internally where as a function it uses payoffMatrix[[##]]&.

**Examples**

*Input:*

Cinequalities[ {{{{{1, 1, 3}}, {{1, 1, 3}, {1, 3, 1}, {1, 3, 2}}, {{1, 3, 1}, {1, 3, 2}}}, {{{1, 2, 3}}, {{1, 1, 1}, {1, 1, 2}, {1, 3, 3}}, {{1, 2, 1}, {1, 2, 2}}}}, {{{{2, 1, 1}, {2, 2, 1}, {2, 2, 3}}, {{2, 1, 1}, {2, 3, 2}}, {{2, 2, 1}, {2, 2, 3}, {2, 3, 2}}}, {{{2, 1, 1}, {2, 1, 3}, {2, 2, 1}}, {{2, 1, 2}, {2, 3, 1}}, {{2, 2, 2}, {2, 3, 1}, {2, 3, 3}}}}} ]

*Output:*

{{f[1, 1, 3] - f[1, 2, 3], -f[1, 1, 1] - f[1, 1, 2] + f[1, 1, 3] + f[1, 3, 1] + f[1, 3, 2] - f[1, 3, 3], -f[1, 2, 1] - f[1, 2, 2] + f[1, 3, 1] + f[1, 3, 2]}, {-f[2, 1, 3] + f[2, 2, 3], f[2, 1, 1] - f[2, 1, 2] - f[2, 3, 1] + f[2, 3, 2], f[2, 2, 1] - f[2, 2, 2] + f[2, 2, 3] - f[2, 3, 1] + f[2, 3, 2] - f[2, 3, 3]}}

## File: dataArray.m

*{CdataArray}*

### CdataArray

Need global variables: ineqmembers

Creates global variables: groupsIDs

Using functions: Cinequalities

**Description**

CdataArray[payoffMatrix,xlist] creates the dataArray. It works either using the "Speed" model or the "Memory" model. It uses ineqmembers and Cinequalities internally and for the memory model it erases ineqmembers after use.

**Examples**

*Input:*

*noAttr=3*

*ineqmembers={{{{{1, 2, 2}}, {{1, 3, 1}, {1, 3, 2}}, {{1, 2, 2}, {1, 3, 1}, {1, 3, 2}}}, {{{1, 1, 2}}, {{1, 1, 1}, {1, 1, 2}}, {{1, 2, 1}, {1, 2, 2}, {1, 3, 2}}}}, {{{{2, 1, 2}, {2, 2, 2}}}, {{{2, 1, 2}, {2, 2, 2}}}}, {{{{3, 1, 1}, {3, 1, 2}, {3, 2, 1}, {3, 2, 3}, {3, 2, 4}}, {{3, 1, 1}, {3, 1, 2}, {3, 3, 1}, {3, 3, 3}}, {{3, 1, 1}, {3, 1, 2}, {3, 4, 1}, {3, 4, 2}, {3, 4, 3}, {3, 4, 4}}, {{3, 2, 1}, {3, 2, 3}, {3, 2, 4}, {3, 3, 1}, {3, 3, 3}}, {{3, 2, 1}, {3, 2, 3}, {3, 2, 4}, {3, 4, 1}, {3, 4, 2}, {3, 4, 3}, {3, 4, 4}}, {{3, 3, 1}, {3, 3, 3}, {3, 4, 1}, {3, 4, 2}, {3, 4, 3}, {3, 4, 4}}}, {{{3, 1, 1}, {3, 1, 3}, {3, 1, 4}, {3, 2, 1}, {3, 2, 2}}, {{3, 1, 1}, {3, 1, 3}, {3, 3, 1}, {3, 3, 2}}, {{3, 1, 1}, {3, 1, 2}, {3, 1, 3}, {3, 1, 4}, {3, 4, 1}, {3, 4, 2}}, {{3, 2, 1}, {3, 2, 3}, {3, 3, 1}, {3, 3, 3}, {3, 3, 4}}, {{3, 2, 1}, {3, 2, 2}, {3, 2, 3}, {3, 2, 4}, {3, 4, 1}, {3, 4, 3}, {3, 4, 4}}, {{3, 3, 1}, {3, 3, 2}, {3, 3, 3}, {3, 3, 4}, {3, 4, 1}, {3, 4, 3}}}}}*

*payoffMatrix={{{d1111 + d1112 x1 + d1113 x2, d1121 + d1122 x1 + d1123 x2}, {d1211 + d1212 x1 + d1213 x2, d1221 + d1222 x1 + d1223 x2}, {d1311 + d1312 x1 + d1313 x2,*

*d1321 + d1322 x1 + d1323 x2}}, {{d2111 + d2112 x1 + d2113 x2, d2121 + d2122 x1 + d2123 x2, d2131 + d2132 x1 + d2133 x2}, {d2211 + d2212 x1 + d2213 x2, d2221 + d2222 x1 + d2223 x2, d2231 + d2232 x1 + d2233 x2}}, {{d3111 + d3112 x1 + d3113 x2, d3121 + d3122 x1 + d3123 x2, d3131 + d3132 x1 + d3133 x2, d3141 + d3142 x1 + d3143 x2}, {d3211 + d3212 x1 + d3213 x2, d3221 + d3222 x1 + d3223 x2, d3231 + d3232 x1 + d3233 x2,*

*d3241 + d3242 x1 + d3243 x2}, {d3311 + d3312 x1 + d3313 x2, d3321 + d3322 x1 + d3323 x2, d3331 + d3332 x1 + d3333 x2, d3341 + d3342 x1 + d3343 x2}, {d3411 + d3412 x1 + d3413 x2, d3421 + d3422 x1 + d3423 x2, d3431 + d3432 x1 + d3433 x2, d3441 + d3442 x1 + d3443 x2}}}*

CdataArray[payoffMatrix, Cx[noAttr - 1]]

*Output:*

{{-d1121 + d1221, -d1122 + d1222, -d1123 + d1223}, {-d1111 - d1121 + d1311 + d1321, -d1112 - d1122 + d1312 + d1322, -d1113 - d1123 + d1313 + d1323}, {-d1211 + d1311, -d1212 + d1312, -d1213 + d1313}, {0, 0, 0}, {d3121 - d3131 - d3141 - d3221 + d3231 + d3241, d3122 - d3132 - d3142 - d3222 + d3232 + d3242, d3123 - d3133 - d3143 - d3223 + d3233 + d3243}, {d3121 - d3131 - d3321 + d3331, d3122 - d3132 - d3322 + d3332,

d3123 - d3133 - d3323 + d3333}, {-d3131 - d3141 + d3431 + d3441, -d3132 - d3142 + d3432 + d3442, -d3133 - d3143 + d3433 + d3443}, {d3241 - d3341, d3242 - d3342,

d3243 - d3343}, {-d3221 + d3421, -d3222 + d3422, -d3223 + d3423}, {-d3321 - d3341 + d3421 + d3441, -d3322 - d3342 + d3422 + d3442, -d3323 - d3343 + d3423 + d3443}}

groupIDs=={{1}, {1}, {1}, {2}, {3}, {3}, {3}, {3}, {3}, {3}}

## File: objective.m

*{coefficient1, objective}*

### coefficient1

**Description**

Set coefficient1=1 (default) or coefficient=-1

### objective

Need global variables: coefficient1, objectivecounter

Creates global variables: objectivecounter

Using functions: -

**Description**

objective[dataArray,x1,x2,...,xn] defines the objective function to maximize, as the number of satisfied inequalities. For a specific x-vector value we get a list of numbers. The number of positives is the outcome.

It uses Mathematica’s UnitStep function that returns 1 for equalities too of course in the Machine precision sense which often is impossible to achieve due to complex calculations that rarely put the 2 inequality’s part equal to each other.

Note: In the same file a function named objectiveV is also defined. V is for Verbose and this function printouts more info when running than objective which is optimized for maximum performance

**Examples**

*Input:*

*coefficient1 = -1;*

*dataArray= {{a11, a12, a13}, {a21, a22, a23}, {a31, a32, a33}, {a41, a42, a43}, {a51, a52, a53}, {a61, a62, a63}}*

objective[dataArray,2,3]

*Output:*

*UnitStep[-1.` a11 + 2 a12 + 3 a13] + UnitStep[-1.` a21 + 2 a22 + 3 a23] +*

*UnitStep[-1.` a31 + 2 a32 + 3 a33] + UnitStep[-1.` a41 + 2 a42 + 3 a43] +*

*UnitStep[-1.` a51 + 2 a52 + 3 a53] + UnitStep[-1.` a61 + 2 a62 + 3 a63]*

## File: maximize.m

*{optimize, maximize}*

### optimize

Need global variables: maxIterations

Creates global variables: objectivecounter

Using functions: PSO

**Description**

optimize[f,x,method] is a wrapper that involves several separated optimization methods. f must be defined as a pure function in the sense f=func[Sequence@@#]&. x is a list of uknowns {x1,x2,...,xn}. Implemented methods for now include Mathematica's, DifferentialEvolution, SimulatedAnnealing, RandomSearch, NelderMead. ParticleSwarmOptimization is an external method (PSO).

Each method carries its own parameters. Default parameters can be changed as in the following examples:

example1:

optimize["parameters"] = { {"ParticleSwarmOptimization", 32, 0, 10, 100, 8}};

example2:

method -> {"DifferentialEvolution", "CrossProbability" -> 0.5, "ScalingFactor" -> 0.6, "RandomSeed" -> 0, "SearchPoints" -> 200}

**Examples**

*Input:*

*optimize["methods"]*

*Output:*

*{"Automatic", "DifferentialEvolution", "NelderMead", "SimulatedAnnealing", "RandomSearch", "ParticleSwarmOptimization"}*

*Input:*

*Just a simple function example*

*f=Sin[Total[#]]&;*

*optimize[f, {x1, x2, x3}]*

*Output:*

*{1., {x1 -> -3.68221, x2 -> -1.29326, x3 -> 6.54627}}*

Input:

f = objective[dataArray, Sequence @@ #] &

optimize[f, {x1, x2}, #] & /@ {"Automatic", "DifferentialEvolution", "SimulatedAnnealing", "RandomSearch", "NelderMead", "ParticleSwarmOptimization"} }

Output:

For each method we get a list of entries with this structure: {max, {x1->value1 … , x2->value2, … }}

### maximize

Need global variables: header

Creates global variables: -

Using functions: Cx, optimize, objective, objectiveV

**Description**

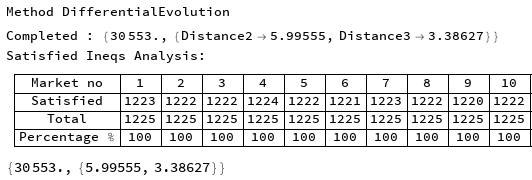
maximize[dataArray\_,noAttr\_,method\_:"DifferentialEvolution", permuteinvariant\_:False, printflag\_:False] is MSE specific and uses the optimize function. It uses objective function (that counts the number of satisfied inequalities). It returns a list {max,{x1->value1, x2->value2, ...}} where max is the maximum number of satisfied inequalities found and the solution {x1,x2,...}

**Example**

Input: (This utilizes a datafile where attributes follow specific distributions)

maximize[dataArray, noAttr, "DifferentialEvolution", False, True]

Output: (Showing just the first 10 markets)



## File: confidence.m

*{generateRandomSubsample, pointIdentifiedCR}*

*Dependencies: Needs["Combinatorica`"](\*For RandomKSubset\*)*

### generateRandomSubsample

**Description**

generateRandomSubsample[ssSize,groupIDs,dataArray] generates a subsample of a given size from a data array.

Parameters:

ssSize - Size of the subsample generated,in terms of number of distinct entities that will be represented in the subsample (ie,nests or coalitions).

groupIDs - A data map that the routine will use to examine the rows of the data array for possible inclusion into the subsample.

dataArray - A data array structure suitable for passing into the objective function.

**examples:**

input:

RandomKSubset[{1, 2, 3, 4, 5, 6, 7, 8}, 4]

output:

{2,5,6,7}

### pointIdentifiedCR

Need global variables: header

Creates global variables: nest, subNormalization, fullNormalization, nextRandomSubsample, coalitions, ssEstimate, ssEstimates

Using functions: generateRandomSubsample, maximize

**Description**

pointIdentifiedCR[ssSize,numSubsamples,pointEstimate,args,groupIDs,dataArray,method,

permuteinvariant,options]

generates a confidence region estimate using subsampling.

Parameters:

ssSize - The size of each subsample to be estimated.

numSubsamples -The number of subsamples to use in estimating the confidence region.

pointEstimate - The point estimate to build the confidence region around (typically the output

of pairwiseMSE).

objFunc - The objective function used in pairwiseMSE.

args - A list of unique symbols used in pairwiseMSE.

groupIDs - A data map used to generate the subsamples.

dataArray - The dataArray parameter used in pairwiseMSE.

options - An optional parameter specifying options. Available options are:

progressUpdate - How often to print progress (0 to disable).Default=0.

confidenceLevel - The confidence level of the region.Default=.95.

asymptotics - Type of asymptotics to use (nests or coalitions).Default=nests.

subsampleMonitor - An expression to evaluate for each subsample.Default=Null.

symmetric - True or False.If True,the confidence region will be

symmetric.Default=False.";

**Example**

Input: (This utilizes a datafile where attributes follow specific distributions)

ssSize = 3; numSubsamples = 50; alpha = 0.05;

(Print["Starting pointIdentified process where alpha = ", alpha];

pointIdentified =

pointIdentifiedCR[ssSize, numSubsamples, sol[[2]], Cx[noAttr - 1],

groupIDs, dataArray, method, permuteinvariant,

confidenceLevel -> (1 - alpha), asymptotics -> nests,

progressUpdate -> 10];

{pointIdentified,

ListPlot@Flatten[Map[

MapIndexed[Flatten@{#2, #1} &, #] &,

pointIdentified[[-1]]

], 1]

}) // AbsoluteTiming

Output: