Functions Reference v.1.0.0

[**Introduction**](#_lhr0998klmto) **2**

[**Global variables**](#_xqy6xjns8iff) **2**

[**File: import.m**](#_x9p6f0x51kmm) **3**

[import](#_cp6ma8rm3pf2) 3

[**File: payoff.m**](#_gpmtcz4rar38) **4**

[Cx](#_fepx2hmkizbz) 4

[payoff](#_sgoel7gkpie2) 4

[payoffDM](#_d1phr68acut5) 5

[CpayoffMatrix](#_d8vb92qtnfus) 5

[Ctotalpayoff](#_osky0e79q71v) 6

[**File: matching.m**](#_cvc0dkixvle0) **6**

[generateAssignmentMatrix](#_i82z3ctshmre) 6

[CmatchMatrix](#_ms1hxzth03pr) 7

[Cmates](#_ei0a11staaxf) 8

[Cmate](#_v3qqamqso7y0) 9

[quotas](#_qqcvat7916cu) 10

[**File: inequalities.m**](#_sh3sqs7fq266) **11**

[Cineqmembers](#_wy6stkxdmy92) 11

[Cinequalities](#_wqnwe8rykkj0) 12

[**File: dataArray.m**](#_7ozwa1x3snxc) **12**

[CdataArray](#_7nbh7zsc22z0) 12

[**File: objective.m**](#_unuthl1xspbg) **14**

[Coefficient1 (global variable)](#_68i3pkpr4lh6) 14

[objective](#_n0dh4rl5nz5f) 14

[**File: maximize.m**](#_te7kgdjcsdp5) **15**

[optimize](#_bb2js28jduky) 15

[maximize](#_d9mlx4e0zeff) 16

[**File: confidence.m**](#_mspdsg6bb9nk) **17**

[generateRandomSubsample](#_qc7ho5v9n1mx) 17

[pointIdentifiedCR](#_gqoxq5y93e8s) 17

# 

## Introduction

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"}

## Global variables

For efficiency reasons some global variables have been defined:

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Name** | **Default Value** | **Description** |
| mse | MSEresources | "Speed" | Switch to “Memory” model when working with big data |
|  | objectivecounter | 0 | Increments everytime the objective function is called |
|  | notebookslist | ... | List of files that are loaded when mse.m is loaded |
|  | directory | Null | The path of the library. This must be set before loading the library. |
| objective | coefficient1 | 1 | To normalize the payoff function, we set the first coefficient to \  either 1 or -1 coefficient1=1 (default) or coefficient=-1 |
| maximize | permuteinvariant | True | I |

## File: import.m

*Attention: There are many more import-related functions for various situations that involve speed increase or better memory management. Those are {readHeader, readYourCSV, importCSV, readTable,*

### import

Needs global variables: MSEresources

Creates global variables: -

Using functions: -

**Description**

{noM,noU,u,noAttr}=import[filename,"stream"] to import data on upstream (u) agents

{noM,noD,d,noAttr}=import[filename,"stream"] to import data on downstream (d) agents

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

Input datafiles created by the user should consist of rows in the form {market index, agent index, attributes of the agents (u) or (d)}

noM- number of markets

no- number of agents

u, respectively d, refers to the vector of individual attributes of that agent. This vector will be used to calculate distanceMatrices and the payoff function.

noAttr-number of individual attributes characterizing an agent

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 the precomputed data where the “distance attributes” between each pair of (u,d) agents have been already calculated.

noM- number of markets

noU- number of upstream agents

noD- number of downstream agents

noAttr- number of distance attributes characterizing a pair (u,d)

distanceMatrices is a matrix of noU x noD rows and noAttr columns which contains the “distance attributes” (i.e. multiplications of individual attributes or pair-specific attributes) between any pairs of upstream-downstream agents, matched or non-matched.

Datafiles consists of rows in the form {m,u,d,distance1, distance2, …, distance(noAttr), match (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**

*Please check in*

*examples/separate\_streams\_abstract.nb*

*examples/precomputed\_abstract.nb*

## File: payoff.m

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

### Cx

Needs global variables: -

Creates global variables: x1,x2,x3,... (just creates, it does not set 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

Needs global variables: noAttr

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 when the streams of u and d agents are imported separately (i.e. not in the precomputed format). 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:*

*(+-1)u[m,1]\*d[m,1]+u[m,2]\*d[m,2]\*x1+u[m,3]\*d[m,3]\*x2 where x1 and x2 are the (unknown) coefficients of the payoff function.*

### payoffDM

Needs global variables: noAttr, distanceMatrices

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:*

*distanceMatrix = ….*

*Output:*

*payoff[1,2,3] =...*

*payoff[2,1,1] = ...*

### 

### CpayoffMatrix

Needs 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.

payoff is used when input data consist of separate u and d streams

payoffDM is used for precomputed data

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

**Examples**

*Input:*

*Output:*

### Ctotalpayoff

Needs global variables: -

Creates global variables: totalpayoff

Using functions: -

**Description**

Ctotalpayoff[payoffobject,mates] calculates the total payoff (i.e. the sum of payoffs) across all markets for the specific mates arrangement. This function accepts as a first argument the “payffobject”, which can either be the name of the payoff function or the payoffMatrix (in case we have already calculated all pair payoffs).

**Examples**

*Input:*

*Output:*

## 

## File: matching.m

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

### generateAssignmentMatrix

Needs 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. The optimal assignment is the one that maximizes the total payoff (i.e. the sum of all payoffs) in a market. 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 agent.

Notice that the quota is max number of matches per agent ( so in the data the real number of matches could be lower than the max).

**Examples**

*Input:*

*For 1-1 relationships (when quotaU=quotaD=1)*

*generateAssignmentMatrix[payoffMatrix,1,1]*

*For many to many relationships (when quotaU=n, quotaD=m)*

*generateAssignmentMatrix[payoffMatrix,n,m]*

*Output:*

*1-1*

*{*

*{0, 0, 0, 0, 0, 0, 0, 0, 1},*

*{0, 0, 0, 0, 0, 0, 1, 0, 0},*

*{1, 0, 0, 0, 0, 0, 0, 0, 0},*

*{0, 0, 0, 1, 0, 0, 0, 0, 0},*

*{0, 0, 1, 0, 0, 0, 0, 0, 0 }*

*}*

*3-2*

*{*

*{0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1},*

*{0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0},*

*{1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0},*

*{0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1},*

*{1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0}*

*}*

### CmatchMatrix

Needs 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', it creates the matchMatrix by running the generateAssignmentMatrix routine for all markets.

If p is set to 'True' , it does the same, only in parallel! ATTN: IT DOES NOT WORK WITH VARIABLE QUOTAS YET.

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

**Examples**

*Input:*

*Output:*

### Cmates

Needs global variables: -

Creates global variables: mates

Using functions: -

**Description**

Cmates[matchMatrix] simplifies the matchMatrix to a list of triples that define matches across all markets. It provides another way to express all the matching information that is, which upstream is matched with which downstream and in which market. The output consists of a list of lists of triples each of which has the following structure: {market\_index,upstream\_index,downstream\_index}.

Output example:{{{1,1,3},{1,3,1},{1,3,2}},{{2,1,1},{2,2,1},{2,2,3},{2,3,2}}}. In this example we have 2 lists, one per market and each inner list contains the triples. Note that in market 1, upstream agent 2 is not contributing which is fine.

This function is mainly used for the calculation of the total payoff - see Ctotalpayoff routine.

**Examples**

*Input:*

*Cmates[*

*{*

*{*

*{0, 0, 1}, {0, 0, 0}, {1, 1, 0}*

*},*

*{*

*{1, 0, 0, 0}, {1, 0, 1, 0}, {0, 1, 0, 0}*

*}*

*}*

*]*

*Output:*

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

*3}, {2, 3, 2}}}*

### Cmate

Needs global variables: -

Creates global variables: mate

Using functions: -

**Description**

Cmate[matchMatrix] simplifies the matchMatrix from the original code by Santiago & Fox, to a matrix format which consists of lists of pairs, one pair per market. Here, each pair has the following structure: {{{1},{2},...,{noU within this market}},{{downstreams that are matched with upstream1},{downstreams that are matched with upstream2},...,{downstreams that are matched with upstream noU}}.

Example : {{{{1},{2},{3}},{{3},{},{1,2}}},{{{1},{2},{3}},{{1},{1,3},{2}}}}. In this example, there are three upstream and three downstream agents in each market, indexed 1, 2, 3. In the first market, upstream 1 is matched with downstream 3, upstream 2 is not matched, and upstream 3 is matched with downstream 1 and 2. In the second market, upstream 1 is matched with downstream 1, upstream 2 with downstream 1 and 3, and upstream 3 with downstream 2.

The mate=Cmate[matchMatrix] is later fed into the Cineqmembers routine.

**Examples**

matchMatrix =

{

{{0, 0}, {0, 1}, {1, 1}},

{{0, 1, 0}, {0, 1, 0}},

{{1, 1, 0, 0}, {1, 0, 1, 1}, {1, 0, 1, 0}, {1, 1, 1, 1}}

}

Cmate[matchMatrix]

Out[10]= {

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

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

{{{1}, {2}, {3}, {4}}, {{1, 2}, {1, 3, 4}, {1, 3}, {1, 2, 3, 4}}}

}

Another example:

Cmate[

{

{

{0, 0, 1}, {0, 0, 0}, {1, 1, 0}

},

{

{1, 0, 0, 0}, {1, 0, 1, 0}, {0, 1, 0, 0}

}

}

]

Returns:

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

### quotas

Needs global variables: -

Creates global variables: -

Using functions: -

**Description**

quotas[matchMatrix] returns the list {quotaU,quotaD}. Quota is defined for each stream u and d.

**Examples**

*Input:*

*matchMatrix={{{0, 0, 0, 0, 0, 1, 0, 0, 1, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0,*

*0, 0, 0, 0, 0, 0, 0, 1, 0}, {1, 0, 0, 1, 1, 0, 0, 0, 0, 0}, {0, 0,*

*0, 1, 0, 0, 1, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 1, 0, 0}, {1, 0,*

*0, 0, 0, 0, 1, 0, 0, 0}, {0, 0, 0, 1, 1, 0, 0, 0, 0, 0}, {0, 0, 1,*

*0, 0, 0, 0, 0, 0, 1}, {0, 0, 0, 0, 0, 0, 0, 1, 0, 0}}, {{1, 1, 0,*

*1, 1, 1, 1, 0, 1, 1}, {1, 0, 0, 0, 0, 0, 1, 0, 0, 0}, {0, 0, 1,*

*0, 0, 0, 0, 0, 0, 1}, {1, 1, 0, 0, 0, 0, 0, 0, 0, 0}, {1, 0, 0, 0,*

*0, 0, 0, 1, 0, 0}, {1, 0, 0, 1, 0, 0, 0, 0, 0, 0}, {1, 0, 0, 0,*

*0, 0, 0, 0, 1, 0}, {1, 0, 1, 0, 0, 0, 0, 0, 0, 0}, {1, 0, 0, 0, 1,*

*0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 1, 0, 1, 0, 0}}, {{0, 0, 0, 0, 0,*

*0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0,*

*0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0,*

*0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0,*

*0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0,*

*0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 1}}};*

*quotas[matchMatrix]*

*Output:*

{

{{2, 0, 1, 3, 2, 1, 2, 2, 2, 1}, {8, 2, 2, 2, 2, 2, 2, 2, 2, 2}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 1}},

{{2, 0, 1, 3, 2, 1, 2, 2, 2, 1}, {8, 2, 2, 2, 2, 2, 2, 2, 2, 2}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 1}}

}

## File: inequalities.m

*{Cineqmembers, Cinequalities}*

### Cineqmembers

Needs global variables: MSEresources

Creates global variables: ineqmembers

Using functions: -

**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 defines also the way inequalities are formed. At this time, inequalities are created to follow the theoretical proofs done by J Fox. 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 careful because in the “Memory model”, ineqmembers are erased after being used for the dataArray calculation, in order to speed up the calculations.

**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

Needs global variables: MSEresources

Creates global variables: inequalities

Using functions: -

**Description**

Cinequalities[f,ineqmembers] applies 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

Needs global variables: ineqmembers, MSEresources

Creates global variables: groupsIDs, dataArray

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 (global variable)

**Description**

To normalize the payoff function, we set the first coefficient to either 1 or -1

coefficient1=1 (default) or coefficient=-1

### objective

Needs global variables: coefficient1, objectivecounter

Updates global variables: objectivecounter

Using functions: -

**Description**

objective[dataArray,x1,x2,...,xn] defines the objective function to maximize the number of satisfied inequalities. An inequality is satisfied when the left side is weakly greater than the right side (>=).

**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]*

(The symbol ` used here indicates that that the code does machine precision calculations).

## File: maximize.m

*{optimize, maximize}*

### optimize

Needs global variables: maxIterations (this is set to Automatic by default)

Updates 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 unknowns {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**

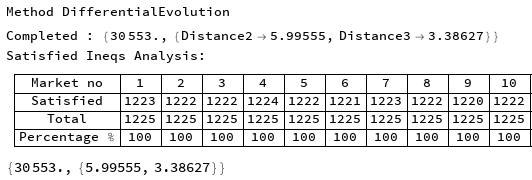
maximize[dataArray\_,noAttr\_,method\_:"DifferentialEvolution", permuteinvariant\_:False, printflag\_:False] is MSE specific and uses the optimize function. It uses the objective function (that counts the number of satisfied inequalities). It returns a list {max,{x1->value1, x2->value2, ...}} where max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

**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 the 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

Needs 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:

Check the example/precomputed\_numeric.nb