The Matlab Game Theory Toolbox *MatTuGames* Version 0.4: An Introduction, Basics, and Examples

Holger I. MEINHARDT *†

October 2, 2013

The game theoretical Matlab toolbox *MatTuGames* provides about 160 functions for modeling, and calculating some solutions as well as properties of cooperative games with transferable utilities. This toolbox is a partial port of the *MATHEMATICA*® package *TuGames*. In contrast to existing Matlab toolboxes to investigate TU games, which are written in a C/C++ programming style with the consequence that these functions are executed relatively slowly, we heavily relied on vectorized constructs in order to write more efficient Matlab functions. In particular, the toolbox provides functions to compute the (pre-)kernel, (pre-)nucleolus, and anti (pre-)kernel as well as game values like the Banzhaf, Myerson, Owen, position, Shapley, solidarity, and coalition solidarity value and much more. In addition, we will discuss how one can use Matlab's Parallel Computing Toolbox in connection with this toolbox to benefit from a gain in performance by launching supplementary Matlab workers. We close this manual by providing some information how to call our *MATHEMATICA*® package *TuGames* within a running Matlab session.

Keywords: Matlab Toolbox, Transferable Utility Game. Solution Concepts, Game Properties

2000 Mathematics Subject Classifications: 90C20, 90C25, 91A12
JEL Classifications: C71

*We are very thankful to Paul Weber helping us to overcome a problem with the Parallel Computing Toolbox on the HP XC3000. In addition, we also owe to Edric Ellis from the MathWorks, Inc. a debt of gratitude to extend the computational limits of our toolbox in parallel mode for Version 0.2.

†Holger I. Meinhardt, Institute of Operations Research, Karlsruhe Institute of Technology (KIT), Englerstr. 11, Building: 11.40, D-76128 Karlsruhe. E-mail: Holger.Meinhardt@wiwi.uni-karlsruhe.de

CONTENTS

1. Introduction

The purpose of this manual is to present some basic illustration how one can use functions provided by the game theoretical Matlab toolbox *MatTuGames*. We will neither discuss all function nor do we give a thorough discussion of all options available that can be enabled from a specific function. In order to get some further information with what kind of additional output/input arguments a function can be called for, it is recommended to invoke in the Command Window of Matlab:

```
help function_name
```

this will return a short user-guide and a list of output and input arguments. In order to get some general information about the toolbox, one has to execute help MatTug, however, typing ReleaseNote returns some information about the additions and changes w.r.t. the previous release. To verify the basic installation execute getting_started in the doc folder.

2. How to use the Toolbox

2.1. Some Preliminaries

In the sequel, we will rely on the usual convention to represent the player set, coalitions and values of a cooperative game by the symbols N, S and v respectively, in order to do some computation within our computational environment. Following this convention, we denote by n the number of players. To do some computation, we need to assign values to the player set, coalitions and worth vector to fully describe a TU game. It is completely sufficient to specify the number of players by a particular integer value to derive the set of players and coalitions, respectively. Internally, Matlab performs its operation on the basis of double-precision binary floating point arithmetic. Consequently, the integer value specifying the number of persons involved should not be larger than 52, since arrays larger than

```
>> 1:2^52;
```

cannot be handled by Matlab, and therefore, this value constitutes an internal Matlab restriction. To make the forthcoming illustration at least somehow interesting, we set this value to 4, hence

```
>> n=4;
```

then we can derive the player set by:

```
>> N=1:n

N =

  1  2  3  4
```

where the players are named by integers. Now, each coalition S can be characterized uniquely by the sum $\sum_{i\in S} 2^{i-1}$, which is the sum of integers associated to players belonging to a particular coalition. For example, the coalition $\{3, 4\}$ is represented by $2^{3-1} + 2^{4-1} = 4 + 8 = 12$, whereas, coalition $\{2, 3, 4\}$ is given by $2^{2-1} + 2^{3-1} + 2^{4-1} = 2 + 4 + 8 = 14$. Thus, the power set without the empty set is specified by:

```
>> P=1:2^n-1

P =

   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
```

Each of these integer characterization of a coalition has an unique n-digit binary representation. If the i-th component of this vector is assigned with the value 1, then player i belongs to the coalition, however, if the i-th component is assigned with the value 0, then player i is not a member of that coalition. The empty coalition is specified by a sequence of zeros, and the grand coalition by a sequence of ones, respectively. Such an n-vector must be read from right to left. Resuming the examples from above, getting:

```
>> dec2bin(12,4)

ans =

1100

>> dec2bin(14,4)

ans =

1110
```

In case that we want to sort the power set according to the size of coalitions where the smallest coalitions should coming first, and within a block of coalitions with equal size we want to apply a lexicographical order, then we should issue the command from below to derive such a coalition order:

```
>> [co bins]=sortsets(P)

co =

   1   2   4   8   3   5   9   6  10  12   7  11  13  14  15


bins =

0001
0010
0100
1000
0011
0101
1001
0110
1010
1100
0111
1011
1101
1110
1111
```

It is also possible to derive from the set $N = \{1, 2, 3, 4\}$ a corresponding power set representation without the empty. This can be obtained with the command *PowerSet()*. We invoke this command with at most one input argument, that is a set representation, which must be coded in vector format [1, 2, 3, 4], as can be seen next

```
>> pws=PowerSet([1 2 3 4])

pws =

 Columns 1 through 10

  [4]  [3]  [1x2 double]  [2]  [1x2 double]  [1x2 double]  [1x3 double]  [1]  [1x2 double]  [1x2 double]
```

```
 Columns 11 through 15

  [1x3 double]  [1x2 double]  [1x3 double]  [1x3 double]  [1x4 double]
```

This operation has returned all subsets of the set $\{1, 2, 3, 4\}$. Any set/coalition S coded in a vector format of length $|S| \geq 2$ can be chosen to derive the corresponding power set.

In order to retrieve the contents of a cell argument, that is a specific coalition, it is enough to type

```
>> pws{13}

ans =

   1   2   4
```

when we want to single out element 13, for instance. However, to retrieve the complete contents of all cells one can just execute

```
>> for k=1:15, pws{k}, end

ans =

   4


ans =

   3


ans =

   3   4


ans =

   2


ans =

   2   4


ans =

   2   3


ans =

   2   3   4


ans =

   1


ans =
```

```
   1   4

ans =

   1   3

ans =

   1   3   4

ans =

   1   2

ans =

   1   2   4

ans =

   1   2   3

ans =

   1   2   3   4
```

From this example, we see that the ordering is not given by its generic power set representation which would be $\{\{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, N\}$. In contrast, this ordering is satisfied within the output argument *bins* of the sorting process by the *sortsets()* command.

Of course, we are also able to compute from a collection of coalitions given by its generic power set representation its unique integer representation. For instance, consider the following collection of coalitions given by $clm = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{3, 4\}\}$. We observe that these coalitions have equal size, then we can input the collection of coalitions *clm* in matrix form, i.e.,

```
>> clm=[1 2; 1 3; 1 4; 3 4]

clm =

   1   2
   1   3
   1   4
   3   4
```

after that, we call the function *clToMatlab()* to get its unique integer representation, hence

```
sS=clToMatlab(clm)

sS =

   3   5   9   12
```

In order to check that the representation is correct, we call the function *sortsets()* to observe that with this function we can also sort sub-collections of power sets, thus

```
>> [coS binS]=sortsets(sS,4)

coS =

   3   5   9   12


binS =

0011
0101
1001
1100
```

For sorting a collection of coalitions of unequal size, we have to code it in cell rather than in matrix form. To make this point more precise, consider now the collection

$$clm2 = \{\{1\}, \{2\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{3, 4\}, \{1, 2, 4\}, \{2, 3, 4\}\},$$

this collection must be represented by

```
>> clm2={[1], [2], [1 2], [1 3], [1 4], [3 4],[1 2 3], [2 3 4]}

clm2 =

  [1]   [2]   [1x2 double]   [1x2 double]   [1x2 double]   [1x2 double]   [1x3 double]   [1x3 double]
```

Again, to get its unique integer representation, calling

```
>> sS2=clToMatlab(clm2)

sS2 =

   1   2   3   5   7   9   12   14
```

To verify this result, we invoke again

```
 >> [coS2 binS2]=sortsets(sS2,4)

coS2 =

   1   2   3   5   9   12   7   14


binS2 =

0001
0010
0011
0101
1001
1100
0111
1110
```

Finally, we want to see whether we can even supply an unique coalition, and indeed, let us choose for this purpose the coalition $clm3 = \{\{2, 3, 4\}\}$, represented under Matlab as

```
>> clm3=[2 3 4]

clm3 =

   2   3   4
```

which gives

```
>> sS3=clToMatlab(clm3)

sS3 =

  14
```

2.2. Defining a Game: Two basic Examples

A Bankruptcy Game

After we have understood in which order the power set is represented under Matlab, we can now assign the worth to each coalition. We can do that while specifying the correspond worth vector by a list of numbers, or we can rely on some toolbox functions to define a TU game. Of course, the toolbox can also handle cost games, but it is recommended to convert them into savings games, this can be achieved by issuing the function *savings_game()*. Let us treat now a bankruptcy situation $\langle E, \vec{d}\rangle$, the estate is given by $E = 250$, and the claims vector of the claimants by $\vec{d} = \{60, 90, 150, 210\}$, thus when we type

```
>> E=250;
>> d=[60 90 150 210];
```

we have specified a bankruptcy situation under Matlab. These values are sufficient to do some computation within our Matlab environment. In a first step, we might be interested in a division rule for that bankruptcy problem. A function that provides a division rule for a bankruptcy problem is called *Talmudic_Rule()*, which is a generalized contested garment principle, and that will coincides with the nucleolus of the corresponding bankruptcy game as we will see below.

```
>> tlm_rl=Talmudic_Rule(E,d)

tlm_rl =

  30.0000  45.0000  75.0000 100.0000
```

Obtaining from a bankruptcy situation $\langle E, \vec{d}\rangle$ its corresponding modest bankruptcy game, we make use of the following definition:

$$v_{E,\vec{d}}(S) := \max\{0, E - d(N\backslash S)\} \quad \forall\, S \subseteq N,$$

where $d(N\backslash S) = \sum_{k\in N\backslash S} d_k$.

From the data that define a bankruptcy situation, we can generate the corresponding modest bankruptcy game while issuing:

```
>> bv=bankruptcy_game(E,d)

bv =

   0   0   0   0   0   0  40   0  10  40 100 100 160 190 250
```

As noted from above, this Matlab toolbox is a port of our *MATHEMATICA*® package *TuGames*[1], we are also interested in to transfer the values into the *MATHEMATICA*® order, this will be realized by executing:

```
>> bw=gameToMama(bv)

bw =

   0   0   0   0   0   0  10   0  40 100  40 100 160 190 250
```

Notice, that the values of the coalitions of game *bw* are arranged according to the order obtained by the function *sortsets()*. It should be obvious, that the game *bw* describes a totally different game under Matlab than game *bv*, and they should not be confound.

To reverse the game operation, the function *gameToMatlab()* is available.

```
>> bv1=gameToMatlab(bw)
bv1 =

   0   0   0   0   0   0  40   0  10  40 100 100 160 190 250


>> bv

bv =

   0   0   0   0   0   0  40   0  10  40 100 100 160 190 250
```

Besides that, we are also able to compute a game basis, and the corresponding Harsanyi dividends or unanimity coordinates. Thus, the game basis of a four-person game is obtained by:

```
>> gb=game_basis(4)

gb =

   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0
   1   1   1   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
   1   0   0   1   1   0   0   0   0   0   0   0   0   0   0
   0   1   0   1   0   1   0   0   0   0   0   0   0   0   0
   1   1   1   1   1   1   1   0   0   0   0   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   1   1   0   0   0   0   0   0
   0   1   0   0   0   0   0   1   0   1   0   0   0   0   0
   1   1   1   0   0   0   0   1   1   1   1   0   0   0   0
   0   0   0   1   0   0   0   1   0   0   0   1   0   0   0
   1   0   0   1   1   0   0   1   1   0   0   1   1   0   0
   0   1   0   1   0   1   0   1   0   1   0   1   0   1   0
   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

and the Harsanyi dividends through:

```
>> hd=harsanyi_dividends(bv)

hd =

   0   0   0   0   0   0  40   0  10  40  50 100  50  50 -90
```

[1]The corresponding computer program can be downloaded from http://library.wolfram.com/infocenter/MathSource/5709/.

From these unanimity coordinates we can reconstruct our original game while issuing:

```
>> bv1=getgame(hd)

bv1 =

   0   0   0   0   0   0  40   0  10  40 100 100 160 190 250


>> bv==bv1
ans =

  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
```

A dual game of the corresponding bankruptcy game bv can be computed in two different ways. The first approach relies on the computation of the greedy bankruptcy game derived from the bankruptcy situation $\langle E, \vec{d} \rangle$. A greedy bankruptcy game is defined as follows

$$\tilde{v}_{E,\vec{d}}(S) := \min\{E, d(S)\} \quad \forall\ S \subseteq N.$$

This task will be accomplished while calling the function *greedy_bankruptcy()*, which requires the same input parameter as the function *bankruptcy_game()* as can be seen below

```
>> gv=greedy_bankruptcy(E,d)

gv =

  60  90 150 150 210 240 250 210 250 250 250 250 250 250 250
```

But, it is also possible to derive the dual directly from its primal game bv. This approach can be followed with the function *dual_game()*. This function needs just one input parameter, namely the data array of the game under consideration, in this case, the modest bankruptcy game bv.

```
>> dv=dual_game(bv)

dv =

  60  90 150 150 210 240 250 210 250 250 250 250 250 250 250
```

That both dual game representations are identical is verified by the following Matlab routine

```
>> all(gv==dv)

ans =

   1
```

A more general Four Person Example

In our next example let us consider a TU game v_2 that is based on the following coalition order of a power set of four person

$$clm_2 := \{\{1\}, \{2\}, \{1, 2\}, \{3\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}, \{4\}, \{1, 4\}, \{2, 4\}, \{1, 2, 4\}, \{3, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}\}.$$

We want now to show that this coalition order coincides with the unique integer representation of coalitions under Matlab, that is to say, the above coalition order is represented under Matlab as

$$sS_2 := \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\},$$

as some simple and tedious calculation reveals to us. In addition, assume that the valuation of the coalitions is given by the following vector:

$$vlm_2 = \{0, 0, 1, 0, 1/2, 1, 5/4, 0, 1, 0, 1, 1, 1, 1, 2\},$$

In a first step we need to code the corresponding power set. For doing so, some effort is required to code with the keyboard the above power set clm_2 in cell form under Matlab, thus we type

```
>> clm_2={[1], [2], [1 2], [3], [1 3], [2 3], [1 2 3], [4], [1 4], [2 4], [1 2 4], [3 4], [1 3 4], [2 3 4], [1 2 3 4]}

clm_2 =

 Columns 1 through 10

  [1]  [2]  [1x2 double]  [3]  [1x2 double]  [1x2 double]  [1x3 double]  [4]  [1x2 double]  [1x2 double]

 Columns 11 through 15

  [1x3 double]  [1x2 double]  [1x3 double]  [1x3 double]  [1x4 double]
```

By the above code, we have got a set representation of the power set *clm_2*. This is not the correct format to invoke some computation. We need to transcribe this format in the unique integer representation of coalitions. To get such a representation we refer to the function to *clToMatlab()* to get the correct coalition format. We simply call

```
>> sS_2=clToMatlab(clm_2)

sS_2 =

   1   2   3   4   5   6   7   8   9   10   11   12   13   14   15
```

We observe that we got the expected result. In the second step, we have to supply the valuation vector of coalitions, thus, we type

```
>> vlm_2=[0 0 1 0 1/2 1 5/4 0 1 0 1 1 1 1 2]

vlm_2 =

 Columns 1 through 14

     0     0   1.0000     0   0.5000   1.0000   1.2500     0   1.0000     0   1.0000   1.0000   1.0000   1.0000

 Column 15

  2.0000
```

Of course, the output argument *vlm_2* contains the complete game information. This data array can be used to verify that the function *vclToMatlab()* replicates exactly the game representation *vlm_2*. In order to do that, we call

```
>> v_2=vclToMatlab(clm_2,vlm_2)

v_2 =

 Columns 1 through 14
```

```
     0     0   1.0000     0   0.5000   1.0000   1.2500     0   1.0000     0   1.0000   1.0000   1.0000   1.0000

 Column 15

   2.0000
```

to observe in the next step that this is really the case, whence, we have

```
>> v_2==vlm_2

ans =

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

However, let us now supply the above power set in its generic coalition order, that is,

$$clm_{2a} = \{\{1\}, \{2\}, \{3\}, \{4\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\},$$

this power set characterization must be again coded in cell input, thus

```
>> clm_2a={[1], [2], [3], [4], [1 2], [1 3], [1 4], [2 3], [2 4], [3 4], [1 2 3], [1 2 4], [1 3 4], [2 3 4], [1 2 3 4]}

clm_2a =

 Columns 1 through 10

   [1]   [2]   [3]   [4]   [1x2 double]   [1x2 double]   [1x2 double]   [1x2 double]   [1x2 double]   [1x2 double]

 Columns 11 through 15

   [1x3 double]   [1x3 double]   [1x3 double]   [1x3 double]   [1x4 double]
```

This order must be identical to that provided by the function *sortsets()*. But then, we need to adjust the above valuation vector while changing the order of the coalitional values, this is done while typing

```
>> vlm_2a=[0 0 0 0 1  1/2  1  1  0  1  5/4  1   1   1   2 ]

vlm_2a =

 Columns 1 through 14

     0     0     0     0   1.0000   0.5000   1.0000   1.0000     0   1.0000   1.2500   1.0000   1.0000   1.0000

 Column 15

   2.0000
```

Notice, that this order of coalitional values is identical to that which we would obtain while calling the function *gameToMama()*, i.e.

$$vlm_{2a} = \{0, 0, 0, 0, 1, 1/2, 1, 1, 0, 1, 5/4, 1, 1, 1, 2\},$$

is replicated by invoking

```
>> w_2a=gameToMama(v_2)

w_2a =
```

```
 Columns 1 through 14

     0     0     0     0  1.0000  0.5000  1.0000  1.0000     0  1.0000  1.2500  1.0000  1.0000  1.0000

 Column 15

   2.0000

>> w_2a==vlm_2a

ans =

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

A note on caution is required here. One should be aware that the input parameter *clm_2a* and *vlm_2a* does not replicate in connection with function *vclToMatlab()* game *w_2a*. To observe this, call

```
>> [v_2a sS_2a]=vclToMatlab(clm_2a,vlm_2a)

v_2a =

 Columns 1 through 14

     0     0  1.0000     0  0.5000  1.0000  1.2500     0  1.0000     0  1.0000  1.0000  1.0000  1.0000

 Column 15

   2.0000

>> sS_2a =

   1   2   4   8   3   5   9   6  10  12   7  11  13  14  15
```

and recognize that when we call *vclToMatlab()* with the input parameter *clm_2a* and *vlm_2a*, then we get for the first *varagout* argument the game representation of *v_2* and not that of *w_2a* as can bee seen next

```
>> v_2a==v_2

ans =

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

However, for the second *varagout* argument the return vector is not the unique integer representation *sS_2* rather than the integer representation obtained by the generic power set characterization.

```
>> dec2bin(sS_2a,4)

ans =

0001
0010
0100
1000
0011
0101
1001
0110
1010
1100
0111
```

```
1011
1101
1110
1111
```

The above procedure, however, has the disadvantage that we have to key in a lot of data to define a game. Let us now discuss an alternative approach that requires much less effort from the keyboard. To this end, we refer to the toolbox command *PowerSet()* to derive all coalitions. Therefore, we execute

```
>> pws=PowerSet([1 2 3 4])

pws =

 Columns 1 through 10

  [4]  [3]  [1x2 double]  [2]  [1x2 double]  [1x2 double]  [1x3 double]  [1]  [1x2 double]  [1x2 double]

 Columns 11 through 15

  [1x3 double]  [1x2 double]  [1x3 double]  [1x3 double]  [1x4 double]
```

to obtain the complete set of coalitions without the empty set. Notice, that this ordering is completely different from the power set representation as described by *clm_2*. This requires that the vector of coalitional values of the game must be coded consistently to get the same game characterization as under *vlm_2*. Hence, the vector of coalitional values which is consistent with the power set characterization *pws* is given by

```
>> vlm_3=[0 0 1 0 0 1 1 0 1 1/2 1 1 1 5/4 2]

vlm3 =

 Columns 1 through 14

     0     0   1.0000     0     0   1.0000   1.0000     0   1.0000   0.5000   1.0000   1.0000   1.0000   1.2500

 Column 15

  2.0000
```

We first check that the two coalitional vectors are different. This is accomplished by

```
>> vlm_2==vlm_3

ans =

   1   1   1   1   0   1   0   1   1   0   1   1   1   0   1
```

Using the command *vclToMatlab()*, we attain the same characterization as under *vlm_2*. To see this, let us compute the game representation while supplying the power set *pws* and the related vector of coalitional values *vlm_3*. Then we get

```
>> v_3=vclToMatlab(pws,vlm_3)

v_3 =

 Columns 1 through 14
```

```
    0     0   1.0000     0   0.5000   1.0000   1.2500     0   1.0000     0   1.0000   1.0000   1.0000   1.0000

 Column 15

   2.0000
```

and finally checking that this representation is identical with the game characterization under an unique integer representation of coalitions. And indeed, we have

```
>> vlm_2==v_3

ans =

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

2.3. Defining a Game: A more comprehensive Example

In order to demonstrate that we are neither restricted to bankruptcy games nor to a specific number of players[2] to generate a transferable utility game, we want now discuss how we can obtain from a collection of coalitions and its valuation a TU game under Matlab. For this purpose, assume that we have given a seven person game $\langle N = \{1, 2, 3, 4, 5, 6, 7\},\ v_3\rangle$, where the game is enlisted by Table 2.1

Table 2.1: Seven Person Game

S	$v_3(S)$	S	$v_3(S)$
$\{1, 3, 4\}$	5	$\{1, 2, 5, 6\}$	10
$\{1, 6, 7\}$	4	$\{2, 3, 6, 7\}$	12
$\{2, 4, 7\}$	4	$\{4, 5, 6, 7\}$	15
$\{2, 5, 7\}$	6	N	25
otherwise	0		

In a first step, we have to supply the set of coalitions. Here, we dispense from the generic coalition order while coding the following arbitrary order

```
>> clm_3={[1 2 5 6] [1 3 4], [1 6 7], [2 3 6 7], [2 4 7], [2 5 7], [4 5 6 7], [1 2 3 4 5 6 7]}

clm_3 =

  [1x4 double]  [1x3 double]  [1x3 double]  [1x4 double]  [1x3 double]  [1x3 double]  [1x4 double]  [1x7 double]
```

Notice that for a correct representation of the game we have to add the grand coalition N to the set of coalitions in contrast to the example given in the Subsection 2.1. The grand coalition is needed to get the correct number of players involved in the game, and therefore the correct characterization of the game.

The vector of valuations must reflect the order chosen for the coalitions by *clm_3*, thus

```
>> vlm_3=[10 5 4 12 4 6 15 25]

vlm_3 =

  10   5   4   12   4   6   15   25
```

[2]Recall that due to the double-precision binary floating point arithmetic of Matlab, the maximum number of players is restricted to 52.

After having provided the set of coalitions and the valuation of coalitions, we are able to compute the seven person game while using the function *vclToMatlab()*. This function has at most two output arguments. The first output variable contains the game information and the second returns the unique integer representation of coalitions having a positive coalitional value in an unsorted order.

```
>> tic;[v_3 sS_3]=vclToMatlab(clm_3,vlm_3);toc
Elapsed time is 0.037729 seconds.
```

In less than a second the game is computed. The complete game is displayed when typing

```
>> v_3

v_3 =

 Columns 1 through 23

   0   0   0   0   0   0   0   0   0   0   0   0   5   0   0   0   0   0   0   0   0   0   0

 Columns 24 through 46

   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0

 Columns 47 through 69

   0   0   0   0  10   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0

 Columns 70 through 92

   0   0   0   0   4   0   0   0   0   0   0   0   6   0   0   0   0   0   0   0   0   0   0

 Columns 93 through 115

   0   0   0   0   4   0   0   0   0  12   0   0   0   0   0   0   0   0   0   0   0   0   0

 Columns 116 through 127

   0   0   0   0  15   0   0   0   0   0   0  25
```

By the result given above, it should be obvious that all coalitions which got no valuation different from zero are assigned with a zero value by default.

The game has the correct size and some coalitions have been allocated with a pre-defined value. Nevertheless, we need to verify that the information contained in the variable *sS_3* coincides with the information hold in variable *clm_3*, and that the game is represented correctly, for this purpose we invoke

```
>> sS_3

sS_3 =

  51   13   97  102   74   82  120  127


>> dec2bin(sS_3,7)
ans =

0110011
0001101
1100001
1100110
1001010
```

```
1010010
1111000
1111111
```

or let us again use the function *clToMatlab()* to compute directly the unique integer representation of coalitions, hence,

```
>> tic;sS_3a=clToMatlab(clm_3);toc
Elapsed time is 0.018966 seconds.
```

This returns

```
>> sS_3a

sS_3a =

  13   51   74   82   97  102  120  127
```

We realize that the order of the coalition is sorted, therefore a little more effort is needed to compare the variable *sS_3* with *sS_3a*. This is done while typing

```
>> all(sS_3a==sort(sS_3))

ans =

   1
```

Both result are identical as expected.

In order to be sure that the correct coalitions have been allocated with its coalition value, we call

```
>> find(v_1>0)

ans =

  13   51   74   82   97  102  120  127
```

which confirms a correct allocation of values.

```
>> v_3(S(v_3>0))

ans =

   5   10    4    6    4   12   15   25
```

Nevertheless, the reader should compare this result with the variable *vlm_3* to figure out whether she/he has understood the representation.

2.4. Defining a Game: An Assignment Game

In this section we want to discuss how we can define an odd-numbered assignment game of 5-players. The example, we want to study is taken from Curiel (1997). The story of the example runs as follows: there are three sellers named Vladimir, Wanda and Xavier who want to sell a house. Whereas the two buyers named Yolanda and Zarik want to buy one. Vladmir values his house at $100,000, Wanda values her house at $150,000 and Xavier values his house at $200,000. We code the sellers valuation by the following vector

```
>> svl=[100  150  200]

svl =

  100  150  200
```

Vladimir's house is worth \$80,000 to Yolanda and \$150,000 to Zarik, Wanda's house is worth \$200,000 to Yolanda and \$120,000 to Zarik, and Xavier's house is worth \$220,000 to Yolanda and \$230,000 to Zarik. This valuation can be represented by a buyer's matrix, whereas each row reflects the worth of the buyers for a specific house, and the columns reflects the worth of each house for a specific buyer. This matrix is given by

```
>> bvl=[80  150; 200  120;220  230]

bvl =

   80  150
  200  120
  220  230
```

In order to get an assignment game, we need a profit matrix to reflect the surplus that can be divided between the sellers and buyers. For computing a profit matrix the command *profit_matrix()* requires two input arguments, namely the valuation vector/matrix of buyers and the valuation vector/matrix of sellers. Return values are: a symmetric profit, buyers, and sellers matrix. Hence, an asymmetric assignment problem is transformed into a symmetric one. This can be seen by the column of zeros into the profit and buyers matrix.

```
>> [prfm buy_mat sel_mat]=profit_matrix(bvl,svl)

prfm =

    0   50    0
   50    0    0
   20   30    0


buy_mat =

   80  150    0
  200  120    0
  220  230    0

sel_mat =

  100  100  100
  150  150  150
  200  200  200
```

To compute an assignment game we need to specify the sellers and buyers set. Since the assignment problem is symmetric, it is just enough to specify the sellers vector to get a well defined assignment game.

```
>> sl_vec = 1:3

sl_vec =

    1    2    3
```

The function *assignment_game()* computes from an assignment problem represented by a sellers vector and a surplus/profit matrix the corresponding symmetric assignment game.

```
>> [v_a1 prof_mat]=assignment_game(sl_vec,prfm)

v_a1 =

 Columns 1 through 23

   0   0   0   0   0   0   0   0   0  50  50  20  20  50  50   0  50   0  50  30  50  30  50

 Columns 24 through 46

   0  50  50 100  30  70  80 100   0   0   0   0   0   0   0   0   0   0  50  50  20  20  50

 Columns 47 through 63

  50   0  50   0  50  30  50  30  50   0  50  50 100  30  70  80 100


prof_mat =

   0  50   0
  50   0   0
  20  30   0
```

To get from a symmetric assignment game its asymmetric game, it is sufficient to truncate the array of the coalitional valuations at the position of irrelevant coalitions, which starts here at position 32 due to the unique integer characterization of coalitions, since a 5-person game has at most $2^5 - 1 = 31$ coalitions. Getting all 31 relevant coalitions we simply type

```
>> v_a=v_a1(1:31)

v_a =

 Columns 1 through 23

   0   0   0   0   0   0   0   0   0  50  50  20  20  50  50   0  50   0  50  30  50  30  50

 Columns 24 through 31

   0  50  50 100  30  70  80 100
```

To verify that the game representation is identical with that given by Curiel (1997), we must transform the game with the *gameToMama()* command

```
>> w_a=gameToMama(v_a)

w_a =

 Columns 1 through 23

   0   0   0   0   0   0   0   0  50   0  50   0  20  30   0   0  50  50  20  50  50  50  30

 Columns 24 through 31

  50  30  50  50 100  70  80 100
```

Now, we can verify if the computed assignment game coincides with that given by Curiel (1997).

Table 2.2: Assignment Game

S	$v(S)$	S	$v(S)$	S	$v(S)$
{1, 4}	0	{1, 2, 5}	50	{3, 4, 5}	30
{1, 5}	50	{1, 3, 4}	20	{1, 2, 3, 4}	50
{2, 4}	50	{1, 3, 5}	50	{1, 2, 3, 5}	50
{2, 5}	0	{1, 4, 5}	50	{1, 2, 4, 5}	100
{3, 4}	20	{2, 3, 4}	50	{1, 3, 4, 5}	70
{3, 5}	30	{2, 3, 5}	30	{2, 3, 4, 5}	80
{1, 2, 4}	50	{2, 4, 5}	50	N	100

2.5. Some Solution Concepts

Of course, we can compute some game values, like the (pre-)nucleolus, Shapley value, Banzhaf value, and the τ-value. The toolbox *MatTuGames* provides two functions to compute the Shapley value. The first function calculates the Shapley value based on the potential of a TU game. In contrast, the second function uses the marginal worth vectors, this function is called *ShapleyValueM()*, which is only useful up to 10-person games. More efficient is the first function, called *ShapleyValue()*. For example, on a **HP XC3000** super-computer,[3] the average computing time of the Shapley value for a 20-person game was about 22 seconds, and for 21-person game, the computation time was about 44 seconds. Resume again our four-person bankruptcy game *bv* example, then we get:

```
>> sh_bv=ShapleyValue(bv)

sh_v =

  29.1667  44.1667  74.1667 102.5000

>> shM_bv=ShapleyValueM(bv)

shM_bv =

  29.1667  44.1667  74.1667 102.5000
```

We can also compute the weighted Shapley value of a TU game with the command *weightedShapley()*. The computation of the weighted Shapley value requires that that the command *weightedShapley()* is supplied with two input arguments, namely the TU game under consideration and a vector of weights of length n. To demonstrate its use let us first define a weights vector of ones, that is each player has equal weight. We define

```
 >> wghs1=ones(1,4)

wghs1 =

   1   1   1   1
```

[3]The HP XC3000 consists of 356 compute nodes, each compute node contains two Quad-Core Intel Xeon processors running on a clock speed of 2.53 GHz. The main memory of the nodes ranges from 24 GB up to 144 GB. Each node has a theoretical peak performance of 81 GFLOPS giving an overall theoretical peak performance of 30.8 TFLOPS.

In this case, the result of the *weightedShapley()* computation must be identical with the computation of the *ShapleyValue()*, and indeed

```
>> wsh1_bv=weightedShapley(bv,wghs1)

wsh1_bv =

  29.1667  44.1667  74.1667 102.5000
```

In the next step, we introduce a weights vector with unequal weights. This is done while coding

```
>> wghs=[0.4 0.1 0.2 0.3]

wghs =

   0.4000   0.1000   0.2000   0.3000
```

We observe that now the result of a weighted Shapley value computation is different from the Shapely value of the bankruptcy game.

```
>> wsh_bv=weightedShapley(bv,wghs)

wsh_bv =

  39.7937  21.2976  61.2063 127.7024
```

Checking efficiency, we attain

```
>> sum(wsh_bv)

ans =

  250
```

In addition, we realize that the fractions distributed to the players by the weighted Shapley value is different from the pre-selected weights, since we have

```
>> wsh_bv/sum(wsh_bv)

ans =

   0.1592   0.0852   0.2448   0.5108
```

The τ-value, which compromises between a disagreement vector and an utopia payoff of the game *bv*, is given for our example by:

```
>> tau_bv=TauValue(bv)

tau_bv =

  29.4118  44.1176  73.5294 102.9412
```

In order to compute the reduced generating polytope of the convex cone of the all non-negative zero-normalized games to classify n-person TU games Ostmann (1984), it is useful to evaluate the quotas of a TU game, we can do that with the command *quotas()*. This function requires just one input argument, namely the game that should be studied.

```
>> qt_bv=quotas(bv)

qt_bv =

 -26.6667   3.3333  63.3333 123.3333
```

The toolbox can also compute the pre-kernel of a game. The function *PreKernel()* is an implementation of Algorithm 8.1.1 of Meinhardt (2013) to seek for a pre-kernel element. Notice, that this function is not an one-to-one implementation, since we had to compromise between computational efficiency and a literal implementation of the algorithm. According to this compromise, we observed an average computation time for a pre-kernel element for 15-players between 0.35 and 1.1 seconds depending on the class of the game. For games with more than 15-players, this function has some precision problems and the results should always be verified with the function *PrekernelQ()*. Nevertheless, we were able to found pre-kernel elements for 22-person games after 108 seconds. In all cases, the average was taken by considering 1000 randomly generated bankruptcy (modest and greedy) and weighted majority games. For our example game, we should be able to reproduce the result obtained by the function *Talmudic_Rule()*, and indeed, we get:

```
>> prk_bv=PreKernel(bv)

prk_bv =

  30.0000  45.0000  75.0000 100.0000
```

It is also possible to issue the function *PreKernel()* with different starting points rather than applying its default vector, that is, in case for our example: $[bv(15), bv(15), bv(15), bv(15)]/4 = [62.5, 62.5, 62.5, 62.5]$. Next, we have chosen six different starting vectors to illustrate this point, all of them converge to the single-valued pre-kernel.

```
>> idm=eye(4)*bv(15)

idm =

  250    0    0    0
    0  250    0    0
    0    0  250    0
    0    0    0  250

>> prk01_bv=PreKernel(bv,idm(1,:))

prk01_bv =

  30.0000  45.0000  75.0000 100.0000

>> prk02_bv=PreKernel(bv,idm(2,:))

prk02_bv =

  30.0000  45.0000  75.0000 100.0000

>> prk03_bv=PreKernel(bv,idm(3,:))

prk03_bv =

  30.0000  45.0000  75.0000 100.0000
```

```
>> prk04_bv=PreKernel(bv,idm(4,:))

prk04_bv =

  30.0000  45.0000  75.0000 100.0000

>> prk05_bv=PreKernel(bv,tau_bv)

prk05_bv =

  30.0000  45.0000  75.0000 100.0000

>> prk06_v=PreKernel(bv,sh_bv)

prk06_bv =

  30.0000  45.0000  75.0000 100.0000
```

If you have a license of the optimization toolbox, then the pre-nucleolus and the nucleolus can be computed with the function *PreNucl()* and *nucl()* respectively.

```
>> prn_bv=PreNucl(bv)
Optimization terminated.
Optimization terminated.
Optimization terminated.
Optimization terminated.

ans =

   30   45   75  100

>> nuc_bv=nucl(bv)
Optimization terminated.
Optimization terminated.
Optimization terminated.
Optimization terminated.

ans =

   30   45   75  100
```

Both solution coincides with the (pre-)kernel, since convex games are zero-monotonic games.

In case that you do not have a license of the optimization toolbox, we recommend to use the pre-nucleolus function *prenucl()* written by Jean Derks[4]. This function can be executed by

```
>> pn_bv=prenucl(bv)

pn_bv =

   30   45   75  100
```

Of course, we can also compute the kernel of the game with

```
>> kr_bv=Kernel(bv)

kr_bv =

  30.0000  45.0000  75.0000 100.0000
```

[4]See http://www.personeel.unimaas.nl/Jean-Derks/downlds/CGinMatlab20050731.zip.

We can check the pre-kernel solution evaluated by calling:

```
>> [prkQ smat e]=PrekernelQ(bv,prk_bv)

prkQ =

   1


smat =

     0 -30.0000 -30.0000 -30.0000
 -30.0000      0 -45.0000 -45.0000
 -30.0000 -45.0000      0 -75.0000
 -30.0000 -45.0000 -75.0000      0


e =

 Columns 1 through 12

 -30.0000 -45.0000 -75.0000 -75.0000 -105.0000 -120.0000 -110.0000 -100.0000 -120.0000 -105.0000 -75.0000 -75.0000

 Columns 13 through 15

 -45.0000 -30.0000   0.0000
```

whereby the returned integer 1 by the variable *prkQ* indicates the correctness of the solution found. The last array e has returned the values of the excess function $e(\cdot)$. The returned matrix *smat* is the matrix of maximum surpluses from which we can immediately see that the balancedness property is satisfied by issuing:

```
>> rats(smat-smat')

ans =

     0       *       *       *
     *       0       *       *
     *       *       0       *
     *       *       *       0
```

It was a commonly held belief of game theorists that the pre-nucleolus of the modest bankruptcy game is identical to the pre-nucleolus of its dual game, the greedy bankruptcy game. Following Funaki and Meinhardt (2006), we provide a counter-example to establish that this belief is false. However, the example demonstrates that the pre-kernel and the pre-nucleolus coincide with the anti pre-kernel and the anti pre-nucleolus of its associated greedy bankruptcy game, respectively.

Computing a pre-kernel element of the dual game or the greedy bankruptcy game *dv* gives us

```
>> dv_prk=PreKernel(dv)

dv_prk =

  25.0000  40.0000  80.0000 105.0000
```

and obtaining for the pre-nucleolus while issuing the *prenucl()* function from Jean Derks

```
>> pn_dv=prenucl(dv)
```

```
pn_dv =

  25   40   80  105
```

This result is identical to the anti pre-kernel of the modest bankruptcy game *bv*. Thus, the pre-kernel element of the dual and the anti pre-kernel coincides, since

```
>> aprk_bv=Anti_PreKernel(bv)

aprk_bv =

  25.0000  40.0000  80.0000 105.0000
```

To compare this result with our previous pre-kernel computation of the modest bankruptcy game *bv*, we observe that this payoff vector is not identical with that of the pre-kernel. Hence, the pre-kernel of game *bv* and the pre-kernel of the dual game *dv* are different.

```
>> prk_bv

prk_bv =

  30.0000  45.0000  75.0000 100.0000
```

However, the computed pre-kernel element of game *bv* coincides with the computed anti pre-kernel of the greedy bankruptcy (dual) game *dv*, as can be seen next

```
>> aprk_dv=Anti_PreKernel(dv)

aprk_dv =

  30.0000  45.0000  75.0000 100.0000
```

which is the expected result.

Finally, we want to demonstrate that the toolbox also allows to compute the least square pre-nucleolus as well as the least square nucleolus of a TU game. These solution concepts have been introduced by Ruiz et al. (1996) and based on the idea of minimizing the variance of the resulting excess of coalitions instead of minimizing the maximum complaint as under the (pre-)nucleolus solution. First, we evaluate the least square pre-nucleolus

```
 >> lspn_bv=LS_PreNucl(bv)

lspn_bv =

  29.3750  44.3750  74.3750 101.8750
```

and then the least square nucleolus

```
>> lsnuc_bv=LS_Nucl(bv)

lsnuc_bv =

  29.3750  44.3750  74.3750 101.8750
```

Both results are identical, since the analyzed game is convex. The reader should also compare these results with the pre-nucleolus and nucleolus of the game.

2.6. Fairness and Related Values with a Coalition Structure

The Shapley value allocates the benefit in accordance with the productivity of players measured in terms of its mean contributions to various coalitions. However, it can be questioned on the grounds of group solidarity with weaker members that the rewards to players will only be allocated w.r.t. a performance principle to distribute the gain of mutual cooperation. A value that reflects some kind of social sympathy with weaker members is the so-called solidarity value which distributes the benefit of mutual cooperation according to the expected average marginal contribution of a member to a coalition (cf. Nowak and Radzik (1994)). Allocating the benefit w.r.t. the solidarity value allow null players to receive positive payoffs. In this respect, productive players, that is players who contribute more that the expected average marginal contribution, abstain from their surplus over the expected average to assign it to null players.

In order to observe the differences between these two values let us return to the assignment game of Subsection 2.4. While observing the payoffs of the Shapley value we realize that all players get a positive reward, there is no null player. However, the variance in the payoffs is relatively large indicating an uneven distribution of the benefit which reflects the differences in their productivity.

```
>> sh_v_a=ShapleyValue(v_a)

sh_v_a =

  16.0000  19.3333   7.6667  27.6667  29.3333
```

In contrast, the solidarity value which reflects some kind of social behavior reveals a more egalitarian distribution of the benefit. It is not completely egalitarian, nevertheless, the variance in the distribution has diminished. In comparison to the Shapley value, the first three players, that is, the sellers of the houses have benefited on the expense of the buyers when a greater degree of solidarity between the players is considered. Especially, the third player is the over-proportional beneficiary when agents exhibit a greater degree of social sympathy and concern.

```
>> sl_v_a=SolidarityValue(v_a)

sl_v_a =

  19.0556  19.7639  17.2222  21.7361  22.2222
```

The solidarity value exhibits some social cohesion with weaker agents of the grand coalition. However, it can be challenged that all agents who participate at a negotiation process to split the surplus of mutual cooperation will accord the same kind of regard to each other. In some cases, there might be between a subgroup of agents a greater degree of cohesion according to some common interest and characteristics. Then some coalitions are very likely to assemble, whereas others are quite implausible due to reverse interests and relations. Therefore, it can be questioned that the subgroup of buyers in the assignment game example who benefits most when the total surplus is allocated in accordance with the Shapley value want to share their surplus with the subgroup of sellers. In this example, the buyers are in a stronger position due to a buyer's market, and therefore, they are able to acquire some part of the producer's surplus for their own purpose. Is it reasonable to assume under these conditions that the buyers will convey some share of their negotiation gain back to the sellers or even some part of their consumer's surplus to get a greater egalitarian outcome as it was proposed by the solidarity value? A more realistic appraisal should take into account that sellers and buyers can form some prior unions to enforce better their interest.

The simplest case of a priori unions, we can imagine, is a partition of the player set N. Let us consider two cases. In the first case, we assume that two à priori unions will assemble, that is, two sellers and one

buyer as well as one seller and one buyer form à priori unions to split the surplus of mutual cooperation. However, in the second case, we suppose that all sellers assemble a union as well as all buyers. The first partition of the player set is coded in Matlab by

```
>> P={[1 3 5],[2 4]}

P =

  [1x3 double]  [1x2 double]
```

Remind that these coalitions must be transcribed into the unique integer representation of sets to be usable under Matlab. This is done with the function *clToMatlab()* as can be seen next

```
>> pm=clToMatlab(P)

pm =

  10  21
```

Similar, for the latter case, which partition the player set into sellers and buyers, we execute

```
>> P2={[1 2 3],[4 5]}

P2 =

  [1x3 double]  [1x2 double]

>> pm2=clToMatlab(P2)

pm2 =

   7  24
```

In the literature, several distribution rules with a coalition structure are discussed. Here, we start with the Aumann-Drèze value (cf. Aumman and Drèze (1974)). In this context, players within a union have found an à priori agreement to cooperate with each other, and finally the players within each subgroup play a subgame in order to split the gain of cooperation in compliance with the Shapley value. Due to Table 2.2, we observe that coalition $\{2, 4\}$ as well as coalition $\{1, 3, 5\}$ obtains a value of 50. In the former case, the corresponding subgame is given by

```
>> sG1=SubGame(v_a,10)

sG1 =

   0   0  50
```

with a Shapley value of

```
>> sh_sg1=ShapleyValue(sG1)

sh_sg1 =

  25  25

>> st_g1=StandardSolution(sG1)

st_g1 =

  25  25
```

The latter subgame is given by

```
>> sG2=SubGame(v_a,21)

sG2 =

   0   0   0   0  50  30  50
```

with a resultant Shapley value of

```
>> sh_sg2=ShapleyValue(sG2)

sh_sg2 =

  15   5  30
```

These payoffs should be compared with the Aumann-Drèze value, which assigns the following payoffs to the players

```
>> ad_vl1=ADvalue(v_a,pm)

ad_vl1 =

  15  25   5  25  30
```

However, if the sellers and buyers assemble separated à priori unions, the payoff assigned to each player is zero by the Aumann-Drèze value. As the attentive reader may want to check by herself/himself, the resultant subgames are null games.

```
>> ad_vl2=ADvalue(v_a,pm2)

ad_vl2 =

   0   0   0   0   0
```

Thus, in order to get a positive payoff an à priori union must contain a seller and a buyer. Furthermore, we realize that in contrast to the first example the payoffs distributed to the players are not efficient. In the context of the Aumann-Drèze value the unions are assembled to get their value and not to improve their bargaining situation in order to obtain a higher share from the maximum amount the grand coalition can make available through mutual cooperation. If one imagine the situation that agents form prior unions to be in a better position during the bargain of how to divide the proceeds of cooperation, then agents may interact on two levels. On the first level unions negotiate among themselves about their share of the grand coalition. However, on the second level, that is within each union, members negotiate among themselves to divide the share of the union. This includes the possibility that members threaten their partners to leave the union in order to form with outsiders a new coalition. This kind of threat must be taken into account by the partners of the union. A value which reflects this kind of two level bargaining is the Owen value. To be more precise, on the first level the unions play a quotient game and divide among themselves the proceeds by the set of principles of distributive arbitration which specifies the Shapley value. On the second level, the payoff gathered by the union is as well internally distributed to its members in compliance with the Shapley value (cf. Owen (1977, 1995)). Therefore, in the first example of à priori unions, the Owen value distributes

```
>> ow_vl1=OwenValue(v_a,pm)

ow_vl1 =

  15.8333  20.0000  10.8333  30.0000  23.3333
```

whereas for the second prior union example, the Owen value allocates

```
>> ow_vl2=OwenValue(v_a,pm2)

ow_vl2 =

  19.1667  21.6667   9.1667  25.0000  25.0000

>> sum(ow_vl2)

ans =

  100
```

In contrast to the Aumann-Drèze value, the players get now an efficient and positive outcome. Again the payoffs reflect in both cases the buyer's market while distributing the highest reward to the buyers. However, the Owen value distributes on both levels the proceeds in accordance with the productive principle that rules the Shapley value. In Calvo and Gutiérrez (2012) a different approach was pursued, which is an amalgamation of the productive principle intrinsic in the Shapley value and of the social sympathy principle intrinsic in the solidarity value. This approach was led by the idea that partners who have agreed upon to form an à priori union exhibit a greater degree of social sympathy and concern with members than with outsiders. Under these circumstance a union will be assembled to use their productive power to get the highest share as possible for the union. This implies that the first level game (the quotient game) is played according to the productive principle with the consequence that the Shapley value determines the share of each union. However, on the second level when members decide to distribute the share of the union among themselves a disproportionate distribution of the proceeds is not acceptable between members, thus the share is distributed on the second level in compliance with the solidarity value. In comparison to the Owen value, the coalition solidarity value (cf. Calvo and Gutiérrez (2012)) provides in both cases a more egalitarian outcome within unions, as we can observe from the following evaluation

```
>> csl_v1a=CoalitionSolidarity(v_a,pm)

csl_v1a =

  16.3889  22.5000  14.7222  27.5000  18.8889

>> csl2_v1a=CoalitionSolidarity(v_a,pm2)

csl2_v1a =

  17.7778  18.6111  13.6111  25.0000  25.0000
```

These payoffs should also be compared with the payoffs obtained when unions are led on the first level by a social behavior while distributing the proceeds of mutual cooperation in the grand coalition by a quotient game in accordance with the solidarity value. On the second level, however, that is, when the share of a union should be divided between members, members are led by a productive principle while allocating the share among themselves according to the Shapley value (cf. Calvo and Gutiérrez (2012, 2013)). For the considered assignment game, we observe that the distribution within unions become more unequal in comparison to the coalition solidarity value.

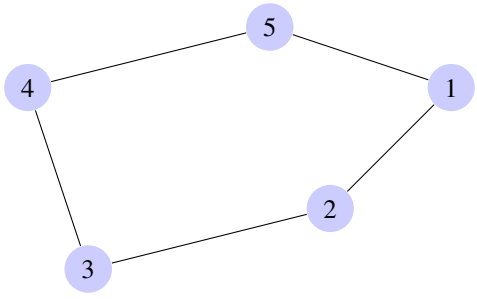
```
>> slsh_v1a=SolidarityShapleyValue(v_a,pm)

slsh_v1a =

  15.8333  20.0000  10.8333  30.0000  23.3333

>> slsh2_v1a=SolidarityShapleyValue(v_a,pm2)

slsh2_v1a =

  19.1667  21.6667   9.1667  25.0000  25.0000
```

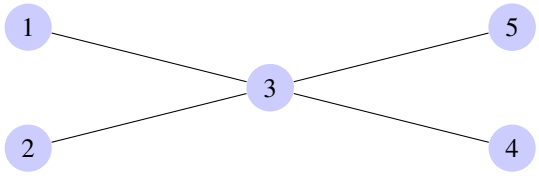
To complete this discussion, consider the case that on both levels the corresponding games are played w.r.t. a social cohesion (cf. Calvo and Gutiérrez (2012, 2013)). We observe that the payoffs coincide with the payoffs obtained by the coalition solidarity value.

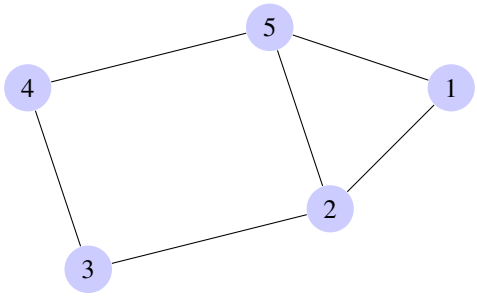
```
>> asl_v1a=apu_SolidarityValue(v_a,pm)

asl_v1a =

  16.3889  22.5000  14.7222  27.5000  18.8889

>> asl2_v1a=apu_SolidarityValue(v_a,pm2)

asl2_v1a =

  17.7778  18.6111  13.6111  25.0000  25.0000
```

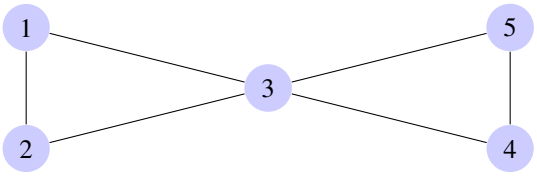
So far, we have examined coalition structures where no player has a distinguished or pivotal position in a bargaining situation. Nevertheless, we can imagine situations where some players possess a central position within a negotiation, since they are chief negotiators, for instance. Myerson 1977 suggested to treat such a situation as an undirected graph in which the nodes correspond to the players and the arcs represent communication links between pairs of players (cf. also with Myerson (1991)). We would suggest that under these circumstances these type of players will be rewarded with an extra payoff to reflect their central position.

Let us now present some undirected graphs which represent communication structures à la Myerson.









We code the left lower graph by

```
>> L={[1 2],[2 3],[2 5],[3 4],[4 5],[1 5]};
```

The right lower graph, however, can be coded under Matlab by

```
>> L2={[1 2],[1 3],[2 3],[3 4],[3 5],[4 5]}

L2 =

  [1x2 double]  [1x2 double]  [1x2 double]  [1x2 double]  [1x2 double]  [1x2 double]
```

Notice, that these communication links represent coalitions which must be again transcribed into its unique integer representation. This is done by

```
>> lm2=clToMatlab(L2)

lm2 =

   3   5   6   12   20   24
```

The Myerson value was introduced by Myerson (1977) and can be computed in the following way: First one finds a partition for all coalitions w.r.t. a communication structure. In a second step the values of all components of each partition are added up to obtain a quotient game. From the quotient game the Shapley value is determined.

The Myerson value w.r.t. the communication structure *lm2* can be computed with the function *MyersonValue()*. This function requires at least two input arguments, that is, a transferable utility game and a coalition/communication structure, which is *lm2* in the current example. A third optional input argument specifies the type of communication structure. Below, the string argument *'cs'* is provided for indicating that a communication structure à la Myerson is used. This is also its default value.

```
>> my_vl=MyersonValue(v_a,lm2)

my_vl =

  7.6667  11.0000  41.0000  19.3333  21.0000

>> my1_vl=MyersonValue(v_a,lm2,'cs')

my1_vl =

  7.6667  11.0000  41.0000  19.3333  21.0000
```

As indicated by the communication structure *lm2* player 3 possesses a pivotal position. We observe by the above evaluation that the Myerson value reimburse this by an extra payment in comparison to the Owen, solidarity or coalition solidarity value.

Myerson pointed out in 1980 that his model needs a generalization towards more restricted cooperation structures that cannot be modeled by a graph. In this line of research the introduction of union stable system by Algaba et al. (2000, 2001) must be classified. There it was assumed that if two feasible coalitions have common members, they can be considered as intermediaries or mediators between these coalitions to ensure mutual cooperation among them. They called a communication situation with these properties a union stable system. For this purpose consider the following communication system which is union stable.

```
>> F2={[1 2],[1 3],[2 3],[1 2 3],[3 4],[1 3 4],[2 3 4],[1 2 3 4],[3 5],[1 3 5],[2 3 5],[1 2 3 5],[4 5],[3 4 5],[1 3 4 5],...
  [2 3 4 5],[1 2 3 4 5]}

F2 =

 Columns 1 through 8

  [1x2 double]  [1x2 double]  [1x2 double]  [1x3 double]  [1x2 double]  [1x3 double]  [1x3 double]  [1x4 double]

 Columns 9 through 16

  [1x2 double]  [1x3 double]  [1x3 double]  [1x4 double]  [1x2 double]  [1x3 double]  [1x4 double]  [1x4 double]

 Column 17

  [1x5 double]
```

To observe this, we have to convert the above coalition structure into its unique integer representation.

```
>> fm2=clToMatlab(F2)

fm2 =

   3   5   6   7  12  13  14  15  20  21  22  23  24  28  29  30  31
```

Then we can verify that the above system is indeed union stable.

```
>> usQ=union_stableQ(fm2)

usQ =

   1
```

Notice that within the union stable system *fm2* player 3 also holds a central position. We shall see below that the Myerson value allocates the highest payoff to the pivotal player as well when a union stable system is assumed.

It is also possible to construct a union stable system from a communication situation à la Myerson. Consider the above communication structure *lm2*. When applying the command *genUnionStable()* on the structure *lm2*, we get

```
>> us2=genUnionStable(lm2)

us2 =

   3   5   6   7  12  13  14  15  20  21  22  23  24  28  29  30  31
```

which is union stable as we can see next

```
>> usQ2=union_stableQ(us2)

usQ2 =

   1
```

Comparing the coalition structure *fm2* with *us2* we realize that both systems coincide

```
>> all(fm2==us2)

ans =

   1
```

However, the coalition structure *lm2* is not union stable

```
>> usQl=union_stableQ(lm2)

usQl =

   0
```

After having introduced the concept of a union stable system, we are now in the position to compute the Myerson value w.r.t. a union stable system. Again, at least two input arguments are needed. In this cases, it will be internally checked that the provided coalition structure is indeed union stable. If the supplied system is not union stable, it is internally assumed that the communication structure is à la Myerson. As a third optional argument a string *'us'* can be supplied. Permissible strings are therefore: *'cs','us'*, and the empty string *''* which means *'cs'*.

```
>> my2_vl=MyersonValue(v_a,fm2)

my2_vl =

  7.6667  11.0000  41.0000  19.3333  21.0000

>> my2_vl=MyersonValue(v_a,fm2,'us')

my2_vl =

  7.6667  11.0000  41.0000  19.3333  21.0000
```

We recommend, however, to supply the command with a string argument to indicate the type of coalition structure. If the indicated type is not correct a warning will be displayed on the screen.

```
>> my_vl2=MyersonValue(v_a,fm2,'cs')
Warning: Coalition structure is not a communication structure, assuming union stable.
> In MyersonValue at 71
Warning: Wrong coalition structure will result in an incorrect result!
> In MyersonValue at 72

my_vl2 =

  7.6667  11.0000  41.0000  19.3333  21.0000
```

By the above example, we notice that the Myerson value focuses on the role of players (node) to establish communication within various coalitions. However, one can also consider a dual approach while focusing on the communicative strength of an arc measured by the outcome of the Shapley value of a dual game defined on the arcs of the communication graph, the so-called arc-game. There, it is supposed that each player has a veto power of the use of communication links (arcs) at which he is an endpoint. According to the fact that players on a communication link have equal power, it is well justified that the worth of an arc should be divided equally among them. The payoff players receive by this kind of distributive arbitration is called position value (cf. Borm et al. (1992)).

In Algaba et al. (2000, 2004) union stable systems are also applied to the position value. This value was introduced by Meessen (1988) and a first axiomatization was given by Borm et al. (1992). Similar to the function that computes the Myerson value, the command which evaluates the position value requires at least two arguments, that is, a TU game and a coalition structure. Again, a third optional input argument specifies the communication structure. In the example below the string *'us'* is used to indicate a union stable communication situation. This also its default value.

```
>> ps_vl=PositionValue(v_a,fm2)

ps_vl =

  8.0833   9.0833  42.2500  19.6667  20.9167

>> ps_vl=PositionValue(v_a,fm2,'us')

ps_vl =

  8.0833   9.0833  42.2500  19.6667  20.9167

>> sum(ps_vl)

ans =

 100.0000
```

There is no unique approach available to model a communication situation when a graph modeling is not possible. Instead of referring on a union stable system, it is also possible to rely on a hypergraph communication situation introduced by van den Nouveland et al. (1992). As worked out in Algaba et al. (2004) both have some close relation. Given a hypergraph system, a union stable system is associated. In a hypergraph system communication is only possible by a conference, whereas a conference must be a set from the power set $2^n - 1$, which must have at least two members. The following system is a hypergraph

```
>> H2={[1 2],[1 3],[2 3],[1 2 3],[3 4],[3 5],[4 5],[3 4 5]}

H2 =

  [1x2 double]  [1x2 double]  [1x2 double]  [1x3 double]  [1x2 double]  [1x2 double]  [1x2 double]  [1x3 double]

>> hm2=clToMatlab(H2)

hm2 =

   3   5   6   7  12  20  24  28
```

as can be checked while calling the function *hypergraphQ()*

```
>> hypergraphQ(hm2,5)
```

```
ans =

   1
```

The position value w.r.t. a hypergraph system is given by

```
>> ps2_vl=PositionValue(v_a,hm2,'hs')

ps2_vl =

  28.4921  29.2063  55.0000  31.9841  35.3175

>> sum(ps2_vl)

ans =

  180
```

Here, we invoked the function with three input arguments. The last input argument indicates the hypergraph system. Thus, permissible string arguments are the strings *'us', 'hs'*, and the empty set to indicate the default, which is *'us'*. Notice that the position value is neither fair nor stable, which can be seen that the sum of the payoffs adds to 180 instead of 100, the value the grand coalition can make available. The payoffs can also add up to less than 100, this depends on the hypergraph system. Again, we recommend to use the command with a string argument.

```
>> ps2_vl=PositionValue(v_a,hm2)
Warning: Coalition structure is not union stable, checking hypergraph.
> In PositionValue at 47
Warning: Wrong coalition structure will result in an incorrect result!
> In PositionValue at 48
Warning: Coalition structure is a hypergraph!. Assuming hypergraph.
> In PositionValue at 54

ps2_vl =

  28.4921  29.2063  55.0000  31.9841  35.3175
```

As suggested above, it is assumed that the provided coalition structure is union stable. However, it will be always verified that the system is union stable, if not, it will be checked on a hypergraph system as we can observe by the warnings printed on the screen. If the system is neither of them, a hypergraph will be constructed.

2.7. Replication of a Pre-Kernel Element

By Theorem 7.6.1 of Meinhardt (2013) it is also possible to replicate a certain pre-kernel point as a pre-kernel point for some related games within a particular variation on the parameter basis of the default game, which is in our case the game *bv*. The Matlab function that computes for us the linear independent games that replicate the pre-kernel element $prk\_bv = [30, 45, 75, 100]$ is called *replicate_prk()*. This function does not only compute the set of games that replicate our pre-kernel solution, it also checks whether the solution satisfies the pre-kernel properties for the derived games. It checks, in addition, whether the equivalence classes have changed or not by the parameter change. For our default game *bv* the scaling factor μ can be varied within the range of $(-42.5, 43.6)$ for all 11-linear independent games without affecting the pre-kernel solution. For our example, we set μ to 5. In order to compute all games that replicate a specific

pre-kernel point one has to supply the function *replicate_prk()* with the default game, here *bv*, a pre-kernel element of game *bv*, the scaling factor μ and the option to specify that the largest or smallest most effective coalitions should be treated. We issue the function *replicate_prk()* by:

```
>> RepSol=replicate_prk(bv,prk_bv,5,1)

RepSol =

     V_PrkQ: [1 1 1 1 1 1 1 1 1 1 1]
      V_SPC: [11x15 double]
       SBCQ: [1 1 1 1 1 1 1 1 1 1 1]
        SBC: [1x1 struct]
      Mat_W: [7x15 double]
    P_Basis: [11x15 double]
  VarP_Basis: [11x15 double]
```

The return variables are structure elements. Thus, to retrieve the contents of one of the above structure elements, one has to issue for instance:

```
>> RepSol.V_PrkQ

ans =

   1   1   1   1   1   1   1   1   1   1   1
```

The first field variable *V_PrkQ* yields a sequence of ones and/or zeros. A zero in column k indicates that the game v_k does not possess *prk_bv* as pre-kernel element. However, a number one in column k means that the pre-kernel properties are also satisfied at *prk_bv* for game v_k. For our example, a sequence of ones is returned by *V_PrkQ* meaning that all 11-linear independent games which are derived from *bv* due to the relationship obtained through Equation 7.5.1 in Meinhardt (2013) replicate *prk_bv* as a pre-kernel solution. The second field variable *V_SPC* yields the list of all 11 linear independent games which were derived from game *bv*. According to its size, this output is suppressed. The third field variable *SBCQ* yields again a sequence of ones and/or zeros indicates whether the equivalence class that contains the element *prk_bv* has changed or has remained invariant against a change in the basis of the game parameter. The fourth output variable *SBC* contains the information about the equivalence classes for all linear independent games. It returns a field that contains the information about the set of most effective coalitions that constitute the equivalence classes containing *prk_bv*. The fifth field variable *Mat_W* returns the matrix W of Equation 7.5.1. Furthermore, the field variable *P_Basis* contains the information about the scaled up parameter basis by the scaling factor $\mu = 5$. Finally, the field variable *VarP_Basis* returns the amplitude of the change in the game values with respect to the default game *bv*.

The rank condition of the game matrix *V_SPC* can be easily checked out by executing:

```
>> rank(RepSol.V_SPC)

ans =

  11
```

The game matrix in rational numbers can be obtained while issuing:

```
>> rats(RepSol.V_SPC)

ans =

  -79/33     31/29     23/32     65/36     1/335     22/19     7204/175     -1/17     341/38     1565/39     3665/36     5547/55     4671/29     6191/33     250
```

```
-135/49    -51/20    -263/52    -105/47    -113/26     -77/18     872/25      11/43     331/40    1227/32    4595/47    4024/41    3149/20    9175/49     250
 23/38     -59/24       9/19       87/46      57/65     -13/90    1557/38     -2/41      89/9     1477/38    4687/46    1916/19    3781/24    7243/38     250
     0          0       3/56      -1/46     -59/49     -118/61     747/19      65/21     163/14    5083/121   4599/46   10160/99     160        190        250
     0          0     -59/42      39/68     -71/37      -1/10      409/11     -21/22    1062/85    2775/71    6839/68    3487/35     160        190        250
     0          0     -11/21       3/14      41/36     -697/279    886/23     -23/21     409/50   15427/363   1403/14    7247/73     160        190        250
     0          0    -117/59      17/21      14/33      -69/104   1783/49      -9/62     289/32    1583/40    2117/21    2937/29     160        190        250
     0          0      78/43     -20/27     -45/23      -8/5       668/17     -20/17     422/51     800/21    2680/27     919/9      160        190        250
     0          0      16/45      -9/62     -163/61      4/17      929/25      -7/31      73/8     1165/29    6191/62    5253/53     160        190        250
     0          0      21/17     -54/107     22/57      -119/55   19365/503     -15/41     520/53    1119/29   10646/107   2864/29     160        190        250
     0          0      -2/9        1/11     -25/76      -25/76    1491/41       7/12      853/80    3253/80    1101/11    2116/21     160        190        250
```

The information about the cell contents of the variable *SBC* can be addressed by issuing the following command:

```
>> RepSol.SBC

ans =

   Eq1: [4x4 double]
   Eq2: [4x4 double]
   Eq3: [4x4 double]
   Eq4: [4x4 double]
   Eq5: [4x4 double]
   Eq6: [4x4 double]
   Eq7: [4x4 double]
   Eq8: [4x4 double]
   Eq9: [4x4 double]
  Eq10: [4x4 double]
  Eq11: [4x4 double]
  Eq12: [4x4 double]
```

The last field variable *Eq12* contains the set of coalitions which forms the equivalence class containing *prk_bv* for the default game $v_0 = bv$. To get the information which coalitions constitute the equivalence class that contains *prk_bv* as a pre-kernel element of game v_2, one needs to issue:

```
>> RepSol.SBC.Eq2

ans =

   1   1   1   1
  14   1   2   2
  14  13   1   4
  14  13  11   1
```

Note that the diagonal of ones has no meaning, they should not confound with coalition $\{1\}$. These values are needed to construct the matrix, and refer to the player pairs $(i, i) \in N \times N$. However, the most effective coalition for the pair $(2, 1) \in N \times N$ is coalition 14, or to put it differently $\{2, 3, 4\}$ by using its generic representation.

2.8. Replication of the Shapley Value

Similar to a replication of a pre-kernel element it is also possible to replicate the Shapley value by a spanning system of linear independent games within the game space. This was first recognized by Yokote et al. (2013) while using a linear basis approach to the Shapley value. Unfortunately, the proof they present for their Theorem 6 is not correct, and must be modified in an appropriated manner to get a spanning system which replicates the Shapley value within the game space. The implementation of this Theorem in the function *replicate_Shapley()* considers the necessary adjustment to calculate the linear manifold in the game space which replicates the Shapley value of a default game. We shall observe that this linear

manifold has a larger dimension than $2^n - 1 - n$ which implies that the spanning system induced by the linear basis is only a linear sub-manifold in the null space of the Shapley value.

Again we rely on the four-person bankruptcy game example of the previous sections. For this purpose let us recall the Shapley value of the game *bv*, which is

```
>> sh_bv=ShapleyValue(bv)

sh_bv =

  29.1667  44.1667  74.1667 102.5000
```

In order to compute all games that replicate the Shapley value of a default game one has to supply the function *replicate_prk()* with the default game, here *bv*, and the options to specify the scaling factor $\mu \in \mathbb{R}\backslash\{0\}$, and a tolerance value, where its default value is $10^8 * eps$. We issue the function *replicate_Shapley()* without a tolerance value but with a scaling factor 10, as can be seen next:

```
>> RepShap=replicate_Shapley(bv,10)

RepShap =

    shQ: 1
   shQv: [1 1 1 1 1 1 1 1 1 1 1 1]
    shm: [12x4 double]
     vm: [12x15 double]
```

We observe that the return value is a structure element with four components, whereas the first field variable *shQ* indicates that all generated games have the same Shapley value given by the logical value 1, hence true, otherwise 0 (false). The second field variable *shQv* yields a sequence of ones and/or zeros. A zero in column k indicates that the game v_k does not possess *sh_bv* as the Shapley value. However, a number one in column k means that the Shapley value *sh_bv* is also satisfied at sh_v_k for game v_k. Notice that the last entity in the sequence must be always true, since this value is related to the default game. This also holds true for the remaining field variables. The third field variable returns a matrix of computed Shapley values from the list of related games. The first eleven rows belong to the derived games and the last row to the default game *bv*. The last field variable *vm* yields the list of all 11 linear independent games which were derived from game *bv* and the default game *bv* itself at the last position.

Retrieving the contents of the first and second component of the structure element is accomplished by issuing

```
>> RepShap.shQ

ans =

   1

>> RepShap.shQv

ans =

   1   1   1   1   1   1   1   1   1   1   1   1

>> length(RepShap.shQv)

ans =

  12
```

In this example, all derived games have the same Shapley value as indicated by the sequences of ones. To inspect the computed list of Shapley values derived from the related games can be done through

```
>> RepShap.shm

ans =

  29.1667  44.1667  74.1667 102.5000
  29.1667  44.1667  74.1667 102.5000
  29.1667  44.1667  74.1667 102.5000
  29.1667  44.1667  74.1667 102.5000
  29.1667  44.1667  74.1667 102.5000
  29.1667  44.1667  74.1667 102.5000
  29.1667  44.1667  74.1667 102.5000
  29.1667  44.1667  74.1667 102.5000
  29.1667  44.1667  74.1667 102.5000
  29.1667  44.1667  74.1667 102.5000
  29.1667  44.1667  74.1667 102.5000
```

We recognize that all computed values coincide with the Shapley value of the default game *bv*. However, it is still unclear how the derived games look like and if they are linear independent. To get an idea we have to retrieve the structure component *RepShap.vm* in order to obtain the list of derived games.

```
>> RepShap.vm

ans =

   10   10    0    0   10   10   40    0   20   50  100  100  170  200  250
   10    0   10   10    0   10   40    0   20   40  110  110  160  200  250
    0   10   10   10   10    0   40    0   10   50  110  110  170  190  250
   10   10    0   10    0    0   40    0   20   50  100  110  160  190  250
   10    0   10    0   10    0   50   10   10   50  100  110  160  200  250
    0   10   10    0    0   10   50   10   20   40  100  110  170  190  250
   10   10    0    0   10   10   40   10   10   40  100  110  160  190  250
    0    0    0   10   10   10   50   10   20   50  110  100  160  190  250
   10    0   10   10    0   10   40   10   10   50  100  100  160  190  250
    0   10   10   10   10    0   40   10   20   40  100  100  160  190  250
   10   10    0   10    0    0   40   10   10   40  100  100  160  190  250
    0    0    0    0    0    0   40    0   10   40  100  100  160  190  250
```

To verify that all derived games are linear independent can be accomplished with the *rank* command of Matlab.

```
>> rank(RepShap.vm)

ans =

   12
```

Hence, all games are linear independent. Thus, from these games we can specify the linear manifold of games in the game space which have an identical Shapley value as our original game. Moreover, we observe that the dimension of the null space of the Shapley value is larger than $2^n - 1 - n = 2^4 - 1 - 4 = 11$. This gives clear indication that the spanning system induced by the linear basis of a TU game is a submanifold in the null space of the Shapley value. Thus, it is not possible to characterize the whole null space of the Shapley value from the linear basis of a TU game.

2.9. SOME GAME PROPERTIES

Besides the computation of solutions, we can also check some game properties with the toolbox. For instance, to check whether the game is k-convex, we have to type the command:

```
>> [ck kcvq]=k_convexQ(bv)

ck =

   3   4


kcvq =

   0   0   1   1
```

The results returned say that the game is 3 as well as 4-convex, but not 1 and 2-convex, that's why, the zeros appear in the last row. The property of 4-convexity means that the game is also convex, which can be verified through:

```
>> convex_gameQ(bv)

ans =

   1
```

Since, that game is convex, it should, in addition, be average-convex as well as semi-convex that can be checked out by issuing:

```
>> average_convexQ(bv)

ans =

   1

>> semi_convexQ(bv)

ans =

   1
```

Of course, then the game should also be super-additive. A weak form can be checked by:

```
>> [wsaq]=weakly_super_additiveQ(bv)

wsaq =

   1
```

whereas super-additivity can be verified by:

```
>> [saq subgC]=super_additiveQ(bv)

saq =

   1


subgC =
```

```
 Columns 1 through 9

  [0]  [0]  [1x3 double]  [0]  [1x3 double]  [1x3 double]  [1x7 double]  [0]  [1x3 double]

 Columns 10 through 15

  [1x3 double]  [1x7 double]  [1x3 double]  [1x7 double]  [1x7 double]  []
```

In contrast to the weak form, the strong version checks whether all sub-games are super-additive. The list of computed sub-games is returned by the cell variable *subgC*. In order to retrieve the data of the sub-games, one has simply to type, for instance:

```
>> subgC{1,14}

ans =

   0   0   0   0  40  100  190
```

This has returned the 14-th sub-game in the cell of sub-games *subgC*, which is according to its 7 coalitional values a three-person game.

A convex game should have an non-empty core, which can be checked with

```
>> [crq_bv m_x]=coreQ(bv)
Optimization terminated successfully.

crq_bv =

   1


m_x =

  31.1639  45.8805  77.8603  95.0953
```

when the Matlab optimization toolbox is installed on your system.

The last result, is an imputation that minimizes an objective function over the core polytope, thus this imputation should be a core allocation that will be verified by:

```
>> [crq bS]=belongToCoreQ(bv,m_x)

crq =

   1


bS =

  Empty matrix: 1-by-0
```

The variable *bs* indicates the set of blocking coalitions which is an empty set here, since the allocation *m_x* belongs to the core of game *bv*.

It should be noted that all functions which check a property can be additional assigned with a tolerance value *tol* to circumvent numerical problems. To get information about the default value of *tol* issue *help function_name*.

2.10. External Libraries and Graphical Features

In case that the *cdd-library* of Komei Fukuda[5] is installed, then it is possible to enumerate all core vertices. Passing the option "*float*" in the command means that the internal computation will be performed by relying on floating point operations, whereby the option "*gmp*" will call the *GNU Multiple Precision Arithmetic Library* to give the result in rational numbers. Of course, the first method is faster than the second, whereas the second has a higher precision than the first.

```
>> crv=CoreVertices(bv,'float')
Optimization terminated successfully.
size = 16 x 5
Number Type = real
(Initially added rows ) = 7 11 13 14 15
(Iter, Row, #Total, #Curr, #Feas)=   6   12     7    6    0
(Iter, Row, #Total, #Curr, #Feas)=   7   10     8    6    0
(Iter, Row, #Total, #Curr, #Feas)=   8    9    11    8    0
(Iter, Row, #Total, #Curr, #Feas)=   9    8    20   15    3
(Iter, Row, #Total, #Curr, #Feas)=  10    4    27   17    4
(Iter, Row, #Total, #Curr, #Feas)=  11    2    34   17    7
(Iter, Row, #Total, #Curr, #Feas)=  12    6    34   17    7
(Iter, Row, #Total, #Curr, #Feas)=  13    1    41   17   12
(Iter, Row, #Total, #Curr, #Feas)=  14    5    41   17   12
(Iter, Row, #Total, #Curr, #Feas)=  15    3    41   17   12
(Iter, Row, #Total, #Curr, #Feas)=  16   17    41   16   12
(Iter, Row, #Total, #Curr, #Feas)=  17   16    41   12   12

crv =

   0    0  150  100
   0    0   40  210
   0   90  150   10
  60    0  150   40
  10   90  150    0
  60   40  150    0
  60   90  100    0
  60   90    0  100
  40    0    0  210
  60    0    0  190
   0   90    0  160
   0   40    0  210
```

To plot the core of a game, the *gmp*-method should be the preferred method due to precision problems of floating point arithmetic. For instance, using the option *"float"* instead of *"gmp"* in the function *CorePlot()* from below could result into a failure even when the core exists. In the example next, we plot the core of game *bv* by using the *gmp*-method. In addition, to the core, we want to draw the imputation set indicated by the integer 1 in *CorePlot()*, otherwise zero; and we want to know how the Shapley value, the pre-nucleolus, and the pre-kernel are located with respect to the core solution. In that case one has to pass the option "*all*" to *CorePlot()*. If one is only interested in the Shapley value, one needs just to enable the option "*shap*", however, for the pre-nucleolus one has to set the "*prn*" option, and finally for the pre-kernel solution, the option value has to be set to "*prk*". In case that none of these solutions should be drawn, the option value must be set to "*none*". In either case, the function *CorePlot()* allows to simply omit this option. In contrast to other functions with a fixed input order, this function allows even an arbitrary input order.

```
>> CorePlot(bv,'gmp',1,'all')
```

[5] See http://www.ifor.math.ethz.ch/ fukuda/.

```
Optimization terminated successfully.
Optimization terminated successfully.
size = 16 x 5
Number Type = rational
(Initially added rows ) = 7 11 13 14 15
(Iter, Row, #Total, #Curr, #Feas)=   6   12     7    6    0
(Iter, Row, #Total, #Curr, #Feas)=   7   10     8    6    0
(Iter, Row, #Total, #Curr, #Feas)=   8    9    11    8    0
(Iter, Row, #Total, #Curr, #Feas)=   9    8    20   15    3
(Iter, Row, #Total, #Curr, #Feas)=  10    4    27   17    4
(Iter, Row, #Total, #Curr, #Feas)=  11    2    34   17    7
(Iter, Row, #Total, #Curr, #Feas)=  12    6    34   17    7
(Iter, Row, #Total, #Curr, #Feas)=  13    1    41   17   12
(Iter, Row, #Total, #Curr, #Feas)=  14    5    41   17   12
(Iter, Row, #Total, #Curr, #Feas)=  15    3    41   17   12
(Iter, Row, #Total, #Curr, #Feas)=  16   16    41   12   12
size = 6 x 5
Number Type = real
(Initially added rows ) = 1 2 3 4 5
(Iter, Row, #Total, #Curr, #Feas)=   6    6     5    4    4
```

The core of game *bv* is shown by Figure 1. This figure gives at least some basic understanding about the shape of the core. But, the result is not really satisfying. Producing core graphics, which are more sophisticated, we refer the interested user to our *MATHEMATICA*® package *TuGames*[6]. The graphic shows, in addition, the Shapely value as a yellow filled circle, and the pre-nucleolus depicted as a cyan filled triangle. Both solutions are very close to each other, that's why, they overlap in the figure from below.

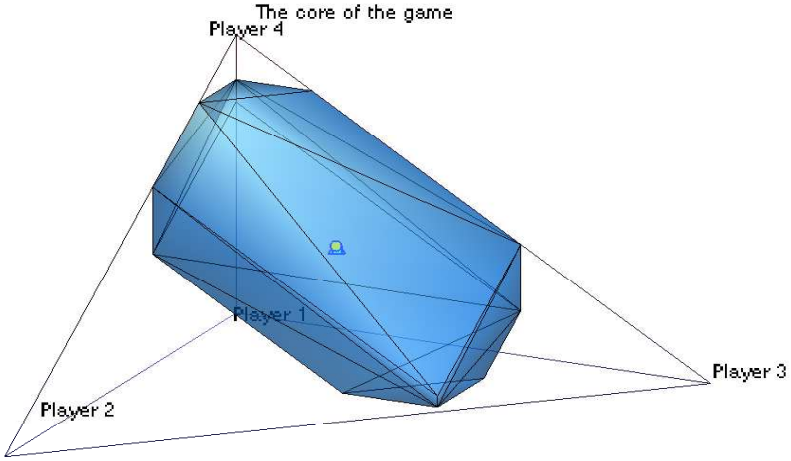


Figure 1: The core, the Shapley value, and the pre-nucleolus of game *v*

The drawback from the above procedure is that we access only indirectly the *cdd-library*. In order to have a direct access to this library, the Matlab interface for the CDD solver – *Cddmex* – must be

[6]See http://members.wolfram.com/jeffb/visualization/gametheory.shtml.

installed on your system[7], then one can use the counterparts of the foregoing introduced functions, called *CddCoreVertices()* and *CddCorePlot()* to directly access the *cdd-library*. Especially, for large games, this speeds up again computation. Apart from those functions, there are also some additional functions that make use of *Cddmex*.

2.11. Third Party Solvers

The MATLAB toolbox *MatTuGames* provides also some interfaces to third party solvers to compute the pre-nucleolus, nucleolus, pre-kernel and kernel of a TU game. These solvers can be used as an alternative to the Matlab optimization toolbox. According to our experience some of these solvers have a higher numerical stability and are faster than the solvers provided by the Matlab optimization toolbox. Nevertheless, when doing floating point arithmetic it is always useful to rely on several solvers from different vendors to verify the correctness of a result. In this subsection, we shortly want to introduce some interfaces for various solvers that can be used with our toolbox. The usage of the interfaces differs in no aspect from the core functions of the toolbox. To start with the discussion, let us compute a kernel element of the above assignment game. In a first step, we use the function *Kernel()* that relies on the quadratic optimization solver of the Matlab optimization toolbox.

```
>> kr_v=Kernel(v_a)

ans =

  10.0000  15.0000   0.0000  35.0000  40.0000

>> kernelQ(v_a,kr_v)

ans =

   1
```

We observe that this result is correct. Notice, that a correct kernel payoff can only be expected for zero-monotonic games. For this class of games, the proposed Algorithm 8.7.1 in Meinhardt (2013) allows a singling out of kernel elements, since this algorithm is simply a modified pre-kernel search process restricted on the imputation set instead of searching on the whole payoff space. Thus, if one has found a pre-kernel element for zero-monotonic game one has also found a kernel element. However, Algorithm 8.7.2 for non-zero-monotonic games does not guarantee a convergence to a kernel element due to a minimum function value of the underlying objective function which can be greater than zero. Nevertheless, quadratic optimization solvers can be applied and are very useful to find kernel elements also for the latter class of games. But then one should always apply the function *kernelQ()* to verify the result.

According to the fact that we can impose a quadratic programming approach to determine a kernel or a pre-kernel element, we can alternatively use some third party solvers which provide a Matlab interface. For instance, if one has a license of the GUROBI, MOSEK or CPLEX vendor respectively, it is also possible to use the corresponding interface to compute a kernel element by making use of the quadratic optimization solver from one of these vendors. Thus, we get for the GUROBI interface to compute a kernel point

```
>> gkr_v = gurobi_kernel(v_a)

gkr_v =
```

[7] See http://control.ee.ethz.ch/ hybrid/cdd.php.

```
  10.0000  15.0000   0.0000  35.0000  40.0000
>> tol=10^8*eps
tol =
  2.2204e-08
>> kernelQ(v_a,gkr_v,tol)
ans =
   1
```

and here the associated MOSEK interface

```
>> mkr_v = msk_kernel(v_a)
mkr_v =
  10.0000  15.0000   0.0000  35.0000  40.0000

>> kernelQ(v_a,mkr_v,tol)
ans =
   1
```

This solution is also obtained with the quadratic programming solver provided by CPLEX, as we can see next

```
>> ckr_v = cplex_kernel(v_a)
Presolve time =   0.02 sec.
Total real time on 8 threads =   0.03 sec.
ckr_v =
  10.0000  15.0000   0.0000  35.0000  40.0000
```

Hence, we observe that these third party solvers return the correct solution within the numerical tolerance.

The first table provides a list of third party solvers for which we have written an interface. From the listed URLs the user can apply for the corresponding mex-files or download the source files.

	Vendor's URL
CDDMEX	http://control.ee.ethz.ch/ hybrid/cdd.php
CLP	https://projects.coin-or.org/Clp
CPLEX	http://www-01.ibm.com/software/websphere/ilog/
CVX	http://cvxr.com/cvx/download/
GLPK	http://www.gnu.org/software/glpk/
GUROBI	http://www.gurobi.com
HSL	http://www.hsl.rl.ac.uk/
IPOPT	https://projects.coin-or.org/Ipopt
MOSEK	http://www.mosek.com
OASES	http://www.kuleuven.be/optec/software/qpOASES
QPBB	http://dollar.biz.uiowa.edu/ sburer/pmwiki/pmwiki.php
QPC	http://sigpromu.org/quadprog/index.html

The second table provides some information about the solution concepts that can be computed while using a specific third party solver and the solver that will be used within the interfaces.

Table 2.3: Third Party Solvers

Vendor	Pre-Kernel	Kernel	Pre-Nucleolus	Nucleolus	LP[a]	QP[b]	LSQ[c]	SLS[d]
CDDMEX	Y	N	Y	Y	Y	N	N	N
CLP	N	Y	N	N	N	Y	N	N
CPLEX	Y	Y	Y	Y	Y	Y	N	N
CVX	Y	Y	N	N	Y	Y	Y	N
GLPK	Y	Y	Y	Y	Y	Y	N	N
GUROBI	Y	Y	Y	Y	Y	Y	N	N
HSL	Y	N	N	N	N	N	N	Y
IPOPT	Y	Y	N	N	N	N	Y	N
MOSEK	Y	Y	Y	Y	Y	Y	Y	N
OASES	Y	Y	N	N	N	Y	N	N
QPBB	N	Y	N	N	N	Y	N	N
QPC	Y	Y	N	N	N	Y	N	N

[a] Linear Programming Solver
[b] Quadratic Programming Solver
[c] Least Square Solver
[d] Sparse Linear System

2.12. Some Consistency Properties

Moreover, with our toolbox, we are able to check if a solution vector satisfies several consistency properties.[8] For instance, the pre-nucleolus should fulfill the Davis and Maschler reduced game property; and the Shapley value should met the Hart and Mas-Colell reduced game property, denoted as DM-RGP or HMS-RGP respectively. But before we discuss this issue, let us compute the Davis and Maschler reduced games w.r.t. the pre-nucleolus. In a second step the Hart and Mas-Colell reduced games w.r.t. the Shapley value will be computed. Starting with the Davis and Maschler reduced games w.r.t. the pre-nucleolus, we get:

```
>> v_dm=DM_Reduced_game(bv,pn_bv)

v_dm =

 Columns 1 through 9

  [30]  [45]  [1x3 double]  [75]  [1x3 double]  [1x3 double]  [1x7 double]  [100]  [1x3 double]
  [ 1]  [ 2]  [1x3 double]  [ 4]  [1x3 double]  [1x3 double]  [1x7 double]  [ 8]  [1x3 double]

 Columns 10 through 14

  [1x3 double]  [1x7 double]  [1x3 double]  [1x7 double]  [1x7 double]
  [1x3 double]  [1x7 double]  [1x3 double]  [1x7 double]  [1x7 double]
```

[8] One option to get full operation of the consistency functions is to use an adjusted and optimized set of files of the Derks toolbox. This set of files have been adjusted by us to conduct a proper and more rapid consistency investigation. It also fixes a problem with closed loops under certain game classes. This set of files can be made available upon request. Alternatively, one can rely on the PreNucl() function, but this requires a license of Matlab's optimization toolbox. The advantage of the adjusted set of files in comparison to the optimization toolbox is its speed. It speeds up the computation of a consistency analysis on a proposed solution up to a factor 5. According to our experience, the reliability and numerical stability also seems to be better.

We can retrieve the contents of cells by typing:

```
>> v_dm{1,12}

ans =

   0   25  175

>> v_dm{2,12}

ans =

   4    8   12
```

This first result indicates the 12-th DM reduced game w.r.t. the pre-nucleolus, which has been formed by the coalitions given by the second return sequence.

Furthermore, the Hart and Mas-Colell reduced games w.r.t. the Shapley value, the corresponding Shapley values of all resulting sub-games, as well as the coalitions that constitute the reduced games will be computed, when we pass the command from below to Matlab:

```
>> v_t=HMS_Reduced_game(bv,sh_bv,'SHAP')

v_t =

 Columns 1 through 8

  [ 29.1667]  [ 44.1667]  [1x3 double]  [ 74.1667]  [1x3 double]  [1x3 double]  [1x7 double]  [102.5000]
  {1x1 cell}  {1x1 cell}  {1x3 cell }  {1x1 cell}  {1x3 cell }  {1x3 cell }  {1x7 cell }  {1x1 cell}
  [    1]  [    2]  [1x3 double]  [    4]  [1x3 double]  [1x3 double]  [1x7 double]  [    8]

 Columns 9 through 14

  [1x3 double]  [1x3 double]  [1x7 double]  [1x3 double]  [1x7 double]  [1x7 double]
  {1x3 cell }  {1x3 cell }  {1x7 cell }  {1x3 cell }  {1x7 cell }  {1x7 cell }
  [1x3 double]  [1x3 double]  [1x7 double]  [1x3 double]  [1x7 double]  [1x7 double]
```

Analogously, the cell contents can be retrieved by typing:

```
>> v_t{1,12}

ans =

  13.3333  41.6667 176.6667

>> celldisp(v_t{2,12})

ans{1} =

  13.3333  13.3333  13.3333


ans{2} =

  21.6667  36.6667  41.6667


ans{3} =

  29.1667  44.1667  74.1667 102.5000
```

```
>> v_t{3,12}

ans =

   4   8   12
```

After we have discussed the functions that computes the DM reduced games as well as the HMS reduced games w.r.t. to a particular solution, we are now in the position to address the issue whether the pre-nucleolus (Shapley value) satisfies DM-RGP (HMS-RGP). First, we check DM-RGP w.r.t. the pre-nucleolus and then w.r.t. the Shapley value. In a second step, we check HMS-RGP w.r.t. the Shapley value and then w.r.t. the pre-nucleolus. In the former case, we are able to verify that the pre-nucleolus satisfies DM-RGP. Calling the function from below, then we observe an affirmation:

```
>> [RGP RGPC]=Reduced_game_propertyQ(bv,pn_bv,'PRN')

RGP =

  rgpQ: 1
  rgpq: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]


RGPC =

  'vS'  {2x14 cell}  'impVec'  {1x15 cell}
```

By the above variables a short summary of the computation is returned. The first output variable is a field variable that contains the information on the first structure element *rgpQ* whether DM-RGP is satisfied (1) or not (0). The second structure element *rgpq* yields a sequence of ones and/or zeros. It contains a precise list of reduced games for which the restriction of *pn_bv* on S is a solution of the reduced game under consideration (1) or not (0). Only in case that the sequence holds a list of ones, DM-RGP is confirmed. However, the second output variable is not a field variable, it is rather a cell variable. This output variable has four sub-cells. The first and third sub-cells are just a wild-cards of names for the second and fourth sub-cell respectively. The second sub-cell contains the information about all reduced games on S at *pn_bv*, and the fourth sub-cell contains the information related to the restriction of *pn_bv* on S, for all $S \subset N$. Therefore, only the contents of the second and fourth sub-cell are usable to go along with some further calculations. If one is interested in a more detailed summary, one has to invoke the following commands:

```
>> RGPC{1}

ans =

vS


>> RGPC{2}

ans =

 Columns 1 through 10

  [30]  [45]  [1x3 double]  [75]  [1x3 double]  [1x3 double]  [1x7 double]  [100]  [1x3 double]  [1x3 double]
```

```
  [ 1]  [ 2]  [1x3 double]  [ 4]  [1x3 double]  [1x3 double]  [1x7 double]  [ 8]  [1x3 double]  [1x3 double]

 Columns 11 through 14

  [1x7 double]  [1x3 double]  [1x7 double]  [1x7 double]
  [1x7 double]  [1x3 double]  [1x7 double]  [1x7 double]


>> RGPC{3}

ans =

impVec


>> RGPC{4}

ans =

 Columns 1 through 10

  [30]  [45]  [1x2 double]  [75]  [1x2 double]  [1x2 double]  [1x3 double]  [100]  [1x2 double]  [1x2 double]

 Columns 11 through 15

  [1x3 double]  [1x2 double]  [1x3 double]  [1x3 double]  []
```

Full information about the contents of all cell variables can be obtained while issuing again the command *celldisp()*. In the course of our discussion we will refrain from displaying any complete set or detailed summary of results obtained from the calculation.

Of course, the Shapley value should not satisfy this kind of consistency property. This statement can be affirmed when we issue the same function as above with the same option parameter w.r.t. the Shapley value rather than w.r.t. the pre-nucleolus. To check this we execute:

```
>> [RGP2 RGPC2]=Reduced_game_propertyQ(bv,sh_bv,'PRN')

RGP2 =

  rgpQ: 0
  rgpq: [1 1 0 1 0 0 0 1 0 0 0 0 0 0 0]


RGPC2 =

  'vS'  {2x14 cell}  'impVec'  {1x15 cell}
```

Of course, the Shapely value must satisfy the HMS-RGP as we can see by:

```
>> [RGP3 RGPC3]=Reduced_game_propertyQ(bv,sh_bv,'SHAP')

RGP3 =

  rgpQ: 1
  rgpq: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]


RGPC3 =

  'vS'  {3x14 cell}  'impVec'  {1x15 cell}
```

But the pre-nucleolus should not met the HMS-RGP, and indeed, we obtain the expected result as can be seen next.

```
>> [RGP4 RGPC4]=Reduced_game_propertyQ(bv,pn_bv,'HMS_PN')

RGP4 =

  rgpQ: 0
  rgpq: [1 1 1 1 0 1 0 1 0 0 0 0 0 1 1]


RGPC4 =

  'vS'  {3x14 cell}  'impVec'  {1x15 cell}
```

2.13. Simple Game

In the foregoing discussion we focused on a bankruptcy game to explain the use of several functions to compute solution concepts and properties of TU games. To observe that the functions provided by our toolbox follow an universal approach, we want now demonstrate amongst others how one can analyze simple games. In addition to the foregoing game class, the investigation of simple games requires some further tools exclusively designated to study this class of games. Be remembered that an n-person game is referred to simple, if it satisfies the following properties:

- $v(S) \in \{0, 1\}$ for all $S \subset N$ and $v(N) = 1$.
- $v(S) \leq v(T)$ for all $S \subset T \subseteq N$.

Notice, that a coalition S is called winning if $v(S) = 1$, otherwise losing. Moreover, a coalition is minimal winning if S is winning and no proper sub-coalition T of S is winning. In the literature, simple games are also called voting games.

To start our example, assume that the collection of coalitions shown below are winning coalitions

$$\{\{1, 2, 5, 6\}, \{1, 3, 4\}, \{1, 6, 7\}, \{2, 3, 6, 7\}, \{2, 4, 7\}, \{2, 5, 7\}, \{4, 5, 6, 7\}\}.$$

Recall the discussion of Subsection 2.1 to notice that this set of winning coalitions must be converted into the unique integer representation of coalitions. In contrast to the foregoing example of 2.3, we need not to enlist the grand coalition. Now, let us code the collection as a cell. Whence,

```
>> clm={[1 2 5 6], [1 3 4], [1 6 7], [2 3 6 7], [2 4 7], [2 5 7],[4 5 6 7]}

clm =

  [1x4 double]  [1x3 double]  [1x3 double]  [1x4 double]  [1x3 double]  [1x3 double]  [1x4 double]
```

Then we can make use of the function *clToMatlab()* to transcribe the representation above into a Matlab format.

```
>> sS=clToMatlab(clm)

sS =

  13  51  74  82  97  102  120
```

This result is the coalition representation in integer characterization. These coalitions represent a set of minimal winning coalitions from which we shall derive a simple game. In a first step, we have to compute from the set of minimal winning coalitions the whole set of winning coalitions. This can be

accomplished by the function *winning_coalitions()*. This function needs as input parameter a set of predefined coalitions, in our case, the collection of minimal winning coalitions *sS* and the number of players involved, which we set to $n = 7$. Then we just have to invoke

```
>> wS=winning_coalitions(sS,7);

>> wS

wS =

 Columns 1 through 23

  13   15   29   31   45   47   51   55   59   61   63   74   75   77   78   79   82   83   86   87   90   91   93

 Columns 24 through 44

  94   95   97   99  101  102  103  105  106  107  109  110  111  113  114  115  117  118  119  120  121

 Columns 45 through 50

 122  123  124  125  126  127

>> length(wS)

ans =

  50
```

to compute the complete set of winning coalitions. Now, we are in the position to construct a simple game in the second and final step. The function *simple_game()* computes for us the simple game while needing as input parameter the set of all winning coalitions *wS* and the number of players involved, which we have set to seven. Thus, we call

```
>> tic;sv=simple_game(wS,7);toc
Elapsed time is 0.000306 seconds.
```

By its definition a simple game has to be monotonic. The monotonicity property can be verified with the function *monotone_gameQ()*. This function has two *varargout* arguments. The first *varagout* argument returns a logical value to indicate whether the game is monotonic, which is grasped by the value one for indicating a true statement or by a zero value, otherwise. However, the second *varagout* argument returns a logical vector to indicate whether the marginal contribution of a player k is increasing 1 or not 0.

```
>> tic;[mq_sv A_sv]=monotone_gameQ(sv);toc
Elapsed time is 0.051266 seconds.
>> mq_sv

mq_sv =

   1

>> A_sv'

ans =

   1   1   1   1   1   1   1
```

The game is monotonic and the marginal contributions are increasing. Moreover, the game is even zero-monotonic as shown by the computation below with the function *zero_monotonicQ()*

```
>> tic;zmq_sv=zero_monotonicQ(sv);toc
Elapsed time is 0.109490 seconds.
>> zmq_sv

zmq_sv =

   1
```

This result establishes that the kernel and the pre-kernel of the game *sv* must be identical.

To get a measurement of the individual voting power of players, our toolbox provides a function that computes the Banzhaf-Coleman index. This function is referred to *banzhaf()* and needs just the game parameter to compute the Banzhaf power index.

```
>> bzf_sv=banzhaf(sv)

bzf_sv =

  0.1724   0.1552   0.0862   0.1379   0.0862   0.1207   0.2414
```

Now let us compare the Banzhaf power index with the Shapley-Shubik power index to figure out its differences

```
>> sh_sv=ShapleyValue(sv)

sh_sv =

  0.1833   0.1500   0.0833   0.1333   0.0833   0.1167   0.2500
```

Notice that the restriction of the Shapley value to the class of simple games is referred to the Shapley-Shubik power index. For completeness, we also want to enlist the result of a pre-kernel computation, giving next

```
>> prk_sv=PreKernel(sv)

prk_sv =

  0.2500   0.1250   0.1250   0.1250   0.1250   0.0000   0.2500

>> prkQ1_sv=PrekernelQ(sv,prk_sv)

prkQ1_sv =

   1
```

as well as the pre-nucleolus

```
>> pn_sv=prenucl(sv)

pn_sv =

  0.2857   0.1429   0.0714   0.1429   0.0714   0.0000   0.2857

>> prkQ2_sv=PrekernelQ(sv,pn_sv)

prkQ2_sv =

   1
```

indicating that the pre-kernel is not an unique element.

Finally, let us see whether this simple game has some veto players. The set of veto players is defined formally as $J^v := \{i \in N \,|\, v(N\backslash\{i\}) = 0\}$. The function which computes for us the set of veto players is called *veto_players()*, which needs only one input argument, namely the data array of the simple game.

```
>> [vp_sv lvp_sv]=veto_players(sv)

vp_sv =

   0   0   0   0   0   0   0


lvp_sv =

   0   0   0   0   0   0   0
```

This function has two *varargout* arguments. The first output argument returns the list/vector of veto players. An integer value $0 < k \leq n$ at position k indicates that player k is a veto player, a zero at position k indicates that player k is not a veto player. In this example, we do not have any veto player shown by the vector of zeros. The second output argument gives the same information but in logical format. The veto player property is false, that is, the zero value for each player is given for the game. A note on caution is required here. The user must be aware that this function does not impose a consistency check, meaning that the data array of the game is not tested of being a simple game.

That this computation is correct can be seen by the result given below, that indicates an empty core, otherwise the core has to be non-empty.

```
>> coreQ(sv)
Optimization terminated.

ans =

   0
```

This implies that we are not able to compute the τ-value of the game, as can be seen next

```
>> tau_sv=TauValue(sv)
Error using TauValue (line 59)
Game is not quasi-balanced! No Tau-Value computed. Sorry!
```

2.14. Weighted Majority Game: The UN Security Council

Let us now discuss a special subclass of simple games, where a council of n members has to pass a bill. To capture a wide range of possible voting games one is not restricted to the general election rule one man one vote. The number of votes needs not to be tied to the number of players, like for a stockholders' meeting. The total number of votes are denoted as $w(N) \in \mathbb{N}$. For passing the bill at least $0 < th \leq w(N)$ votes are needed. A simple game is referred to a weighted majority game, if there exists a quota/threshold $th > 0$ and weights $w_k \geq 0$ for all $k \in N$ such that for all $S \subseteq N$ it holds either $v(S) = 1$ if $w(S) \geq th$ or $v(S) = 0$ otherwise. Such a game is generically represented as $[th; w_1, \ldots, w_n]$.

In the course of our discussion, we rely on a real life example, that is, we want to study the corresponding weighted majority game for the UN security council. Remember that the UN security council consists

of 5 permanent members (China, France, Russia, United Kingdom and USA) and 10 non-permanent members, which are elected by the UN General Assembly for a two-years term. The 5 permanent members have veto power. This structure of the UN security council can be characterized as a weighted majority game by

$$[39; 7, 7, 7, 7, 7, \underbrace{1, \ldots, 1}_{10 \text{ times}}].$$

We code this voting scheme in Matlab through

```
>> w_vec=[7 7 7 7 7 1 1 1 1 1 1 1 1 1 1]

w_vec =

   7   7   7   7   7   1   1   1   1   1   1   1   1   1   1

>> th=39

th =

  39
```

The corresponding weighted majority game is computed while invoking

```
>> wmg=weighted_majority(th,w_vec);
```

In a first step we want to figure out if the pre-kernel solution of the UN security council game *wmg* is unique or not. For this purpose, we check whether the weighed majority game *wmg* possesses a homogeneous representation, since due to Sudhölter (1996) it is known that the pre-kernel of every homogeneous weighted majority game is star-shaped. The toolbox provides the function *homogeneous_representationQ* to check for us the homogeneous representation of a weighted majority game. To call this function one has to supply it with two input arguments, namely, the quota level th and the vector of weights $\{w_1, \ldots, w_n\}$. After the computation has carried out successfully, at most three *varargout* arguments are returned. The first argument returns a logical value to indicate whether the weighted majority game is homogeneous or not, that is, the argument contains the value one for a true statement, whereas the zero value represents a false statement. This argument is denoted here as *hrQ*. The second output argument contains the complete list of minimal winning coalitions, whereas the third output argument holds the information about the corresponding weighted majority game. These output arguments are called in this example *hr* and *wmg2*, respectively.

```
tic;[hrQ hr wmg2]=homogeneous_representationQ(th,w_vec);toc
Elapsed time is 2.562078 seconds.
```

We observe that the elapsed computation time is slightly more than 2.6 seconds to return the result. Checking out the contents of the output variables, we see below that the *hrQ* variable contains the value one, hence, the corresponding weighted majority game *wmg2* is homogeneous. We shall check later if this game coincides with the weighted majority game *wmg* that we obtained from our first calculation.

```
>> hrQ

hrQ =

   1
```

Moreover, we realize that the number of minimal winning coalitions is equal to 210. This vector of minimal winning coalitions is too large to be expressed here.

```
>> length(hr)

ans =

  210
```

In the next step, we verify that the data array that contains the game information has the correct size.

```
>> length(wmg2)

ans =

    32767
```

As we have observed from above, this is case, so we can verify in a further step whether the game *wmg* and *wmg2* are identical.

```
>> all(wmg2==wmg)

ans =

   1
```

The value one is returned by this operation, hence both variables contain the same game information.

After having worked out this, we make again a recurrence to the contents of the output variable *hr*. For doing so, we have to compute the set of minimal winning coalitions. We do this while executing

```
>> tic;[mW WCl]=minimal_winning(th,w_vec);toc
Elapsed time is 2.211715 seconds.
>> length(mW)

ans =

  210

>> length(WCl)

ans =

  848
```

The function returns two *varargout* arguments. The first output variable, denoted by *mW*, contains the collection of all minimal winning coalitions, whereas the second *WCl* contains the complete set of winning coalitions. In the first case we got a number of 210 minimal winning coalitions, which could coincide with the previously computed number of minimal winning coalitions hold by the variable *hr*. The latter output variable *WCl* holds in total 848 winning coalitions.

Checking out that the variables *mW* and *hr* contain the same information can be seen below

```
>> all(mW==hr)

ans =

   1
```

Of course, we can also compute the whole set of winning coalitions directly while using the function *winning_coalitions()*.

```
>> tic;WCl2=winning_coalitions(mW,15);toc
Elapsed time is 1.115236 seconds.
>> length(WCl2)

ans =

  848
```

Again, we obtain the number 848 giving some evidence that the variables *WCl* and *WCl2* will coincide. And indeed, as we will observe by the next operation, both variables are identical and holding therefore the same set of information.

```
>> all(WCl==wCl2)

ans =

   1
```

Remember that from the set of winning coalitions a simple game can be generated, as it will be done below

```
>> tic;sv=simple_game(WCl,15);toc
Elapsed time is 0.003209 seconds.
```

Verifying that game *sv* is identical with game *wmg* as well as with game *wmg2*, we call

```
>> all(sv==wmg)

ans =

   1
```

Giving us the confirmation that in any case we have computed the same homogeneous weighted game.

To check out that we can indeed replicate the veto players of the UN security council game in order to figure out that the game is represented correctly, we invoke

```
 >> [vp_wmg lvp]=veto_players(wmg)

vp_wmg =

   1   2   3   4   5   0   0   0   0   0   0   0   0   0   0


lvp =

   1   1   1   1   1   0   0   0   0   0   0   0   0   0   0
```

Recall that the veto players in the voting scheme representation has been placed on the first fifth positions of the weights vector, hence the above computation reveals to us that the correct set of veto players has been returned.

Since, the UN security council game has veto players, the core of the game should be non-empty. The confirmation is attained while calling the *coreQ()* function

```
>> tic;[crQ xcr]=coreQ(wmg);toc
Optimization terminated.
Elapsed time is 61.701371 seconds.
>> crQ

crQ =

   1

>> xcr

xcr =

 Columns 1 through 14

  0.2000   0.2000   0.2000   0.2000   0.2000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000

 Column 15

  0.0000
```

This result is confirmed with the *belongToCoreQ()* command, as can be seen next

```
>> tic;[crq_wmg bS_wmg]=belongToCoreQ(wmg,xcr,'rat');toc
Elapsed time is 0.021335 seconds.
>> crq_wmg

crq_wmg =

   1

>> bS_wmg

bS_wmg =

  Empty matrix: 1-by-0
```

The complete set of core vertices is computed after slightly more than 1.5 minutes with the *CddCoreVertices()* function, which is the theoretical expected result.

```
>> tic;[vert_wmg crst_wmg]=CddCoreVertices(wmg);toc
Optimization terminated.
Elapsed time is 90.614037 seconds.
>> vert_wmg

vert_wmg =

   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   1   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   1   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   1   0   0   0   0   0   0   0   0   0   0
```

Finally, we want to focus on the individual voting power of players involved in the UN security council game given by some solution schemes. The Banzhaf power index gives

```
>> tic;bzf_wmg=banzhaf(wmg);toc
Elapsed time is 0.037587 seconds.
```

```
>> bzf_wmg

bzf_wmg =

 Columns 1 through 14

  0.1669   0.1669   0.1669   0.1669   0.1669   0.0165   0.0165   0.0165   0.0165   0.0165   0.0165   0.0165   0.0165   0.0165

 Column 15

   0.0165
```

whereas the Shapley-Shubik power index is given by

```
>> tic;sh_wmg=ShapleyValue(wmg);toc
Elapsed time is 0.731415 seconds.
>> sh_wmg

sh_wmg =

 Columns 1 through 14

  0.1963   0.1963   0.1963   0.1963   0.1963   0.0019   0.0019   0.0019   0.0019   0.0019   0.0019   0.0019   0.0019   0.0019

 Column 15

   0.0019
```

A pre-kernel element distributes the voting power only to the veto players, as can be seen next

```
>> tic;prk_wmg=PreKernel(wmg);toc
Elapsed time is 0.305283 seconds.

>> prk_wmg

prk_wmg =

 Columns 1 through 14

   0.2000   0.2000   0.2000   0.2000   0.2000        0        0        0        0        0        0        0        0        0

 Column 15

      0
```

Verifying that the above power index is a pre-kernel element can be seen by calling

```
>> prkQ_wmg=PrekernelQ(wmg,prk_wmg)

prkQ_wmg =

    1
```

However, the τ-value is a pre-kernel element, distributing the same power index as in the previous computation

```
>> tic;tv_wmg=TauValue(wmg);toc
Elapsed time is 0.135587 seconds.
>> tv_wmg
```

```
tv_wmg =

 Columns 1 through 14

  0.2000   0.2000   0.2000   0.2000   0.2000      0      0      0      0      0      0      0      0      0

 Column 15

      0
```

3. Parallel Computing

The functionality of our toolbox[9] is not just restricted to Matlab's serial computing capability, some of the functions already discussed also possess a parallel computing counterpart that would boost the computational performance related to game properties from 10-person games onward. However, an improvement in the computational performance concerning game solutions is not observable before 20-person games onward. For games having less than 10-person in the former case and less than 20-person in the latter case, an activation of Matlab's Parallel Computing Toolbox makes not much sense, since the gain in performance is due to the initializing process, communication overhead, and network traffic rather small or is even outweighed. The execution time for small problems can be twenty times slower due to the overhead in initializing the so-called Matlab *workers*. For these cases, one has even to expect that a computation with a serial function is in general terminated – due to Matlab's multithreading capability – before the Parallel Computing Toolbox has set up all its *workers*. Furthermore, most commands like *ShapleyVaule()* or *Talmudic_Rule()* possess no code that can be executed in parallel. Both possess a recursive structure, each iteration depends on the previous iteration or its start value, and cannot be run in parallel mode. However, the command *PreKernel()*, for instance, possesses a code structure that can be executed in parallel, and can therefore profit from running independent loop iterations simultaneously rather than serially. For example, to compute the maximum surpluses, and therefore the set of most effective coalitions of a player pair $\{i, j\}$ is independent from all other player pairs. As a consequence of this structure, a sweep needs in average 99 percent of the elapsed time to compute all maximum surpluses and to single out the set of most effective coalitions; only a small fraction of the elapsed computing time is assigned to solve a system of linear equations in order determine the pre-imputation for the next iteration step. Hence, executing these independent problems in parallel can significantly improve the performance. And indeed, depending on the game class, the number of *workers* launched, and the available physical memory, we observed an increase in performance between 35 and 88 percent.

Amdahl's Law

According to our observation made with Matlab's *Profiler*, the fraction of the code that can profit from a parallelization for the Pre-Kernel computation is at least about 95 percent, therefore, we can conclude due to Amdahl's Law that this function can heavily profit from a parallelization of the code. For seeing this, denote first the total number of Matlab *workers* (available CPU cores) launched as m. In addition, assume that p denotes the fraction of the code that can be parallelized and the remaining fraction $(1 - p)$ is not parallelizable, then due to Amdahl's Law we have the following formula to estimate the maximum speedup we might achieve while launching m Matlab *workers* in parallel

[9] This section is outdated and needs a complete revision. For the details see the Appendix A

$$Sp(m) = \frac{1}{(1-p) + \frac{p}{m}}$$

This formula let us predict that the best we can achieve is a 1.9 times speedup ratio when using 2 *workers*. However, lauchning four *workers* let us expect 3.4783 speedup ratio. For 8 and 12 *workers* we have already a 5.9259 times speedup ratio and 7.7419 times, respectively. In the limit, that is, when the number of Matlab workers tends to infinity, the maximum speedup ratio tends to $1/(1-p)$. Implying that we can never achieve in parallel a speedup ratio that is better than 20 times of the elapsed total computation time in serial.

3.1. Using the Parallel Computing Toolbox

In the following we suppose that the Parallel Computing Toolbox is available on your computer system. This can be checked out with the command *ver* from the command window. A parallel computing session will be started for instance by:

```
>> matlabpool open 8
Starting matlabpool using the 'MyConf1' configuration ... connected to 8 labs.
```

Here, we launched eight so-called *workers*. It depends on the Matlab license as well as on the computer architecture how many workers one can launch. In our case, we can access eight CPU cores on a compute node, therefore, we decided to launch in addition to the *scheduler* eight *workers*. Depending on the problem, it might be useful due to communication overhead and network traffic to reserve one core for the *scheduler*.

To see how the performance can be improved by running the complement of the command *PreKernel()* in parallel, we consider a 21-person bankruptcy game where we have in total to treat 2097151 different coalitions. All functions that can be run in parallel mode start with the prefix *p_* and can be found in the directory *pct_tugames*. To start with a bankruptcy situation, we need to specify 21-claims for 21-claimants, as well as to specify an estate value, this value will be set to (2/3) of the sum of claims. The claims vector is given next:

```
>> d3

d3 =

  25   37   33   22   28   70   75   83  101  143   56   69   44  128  136  150  187  54   98  174  181

>> length(d3)

ans =

  21
```

and the estate value by:

```
>> E3

E3 =

  1.2627e+03
```

The corresponding bankruptcy game is again derived with the function *bankruptcy_game()*, which has no parallel complement. To avoid that all coalitional values will be displayed on the screen, we will suppress this operation with a semicolon.

```
v3=bankruptcy_game(E3,d3);
```

Our reference situation will be a serial computation of the pre-kernel for game v_3. This computation lasts on a **HP XC3000** super-computer slightly more than 57.8 seconds.

```
>> tic;prk_v3=PreKernel(v3);toc
Elapsed time is 57.836607 seconds.
```

The pre-kernel is an unique imputation given by the vector below.

```
>> prk_v3

prk_v3 =

 Columns 1 through 12

  12.5000  18.5000  16.5000  11.0000  14.0000  35.0000  37.5152  45.5152  63.5152 105.5152  28.0000  34.5000

 Columns 13 through 21

  22.0000  90.5152  98.5152 112.5152 149.5152  27.0000  60.5152 136.5152 143.5152
```

Before we compare the above reference time in serial with the total computation time elapsed in parallel mode, let us remind Amdahl's Law to obtain the theoretical timing values for an ideal world without friction. In such an ideal world, – that is, a world without a costly initializing process, communication overhead, and network traffic – using four *workers* for the computation of a pre-kernel element for the above game can never be accomplished on a **HP XC3000** super-computer in less than 16.626 seconds. However, when using eight *workers* on this machine for the same problem, the computation process can never do better than 9.7588 seconds. Launching an infinite number of Matlab *workers*, Amdahl's Law predicts ceteris paribus that a pre-kernel computation process in parallel can be at best achieved in 2.8915 seconds.

After we have specified the reference time and clarified some theoretical predictions, we want now run the pre-kernel computation in parallel mode. By doing so, we issue the command *p_PreKernel()* with the same argument as above.

```
>> tic;p_prk_v3=p_PreKernel(v3);toc
Elapsed time is 22.003671 seconds.
```

The solution found in parallel is identical to the solution found in serial, as one can assure oneself while invoking:

```
>> p_prk_v3

p_prk_v3 =

 Columns 1 through 12

  12.5000  18.5000  16.5000  11.0000  14.0000  35.0000  37.5152  45.5152  63.5152 105.5152  28.0000  34.5000

 Columns 13 through 21

  22.0000  90.5152  98.5152 112.5152 149.5152  27.0000  60.5152 136.5152 143.5152
```

We have observed immediately that the elapsed time from running the computation in parallel is almost four-tenth (62 percent performance gain) of that elapsed while running the computation in serial. We have considerably benefit by executing the code in parallel. Launching some additional *workers* on a system having 12 or more CPU cores can probably improve the performance further. According to the one-dimensional structure of parallel computing under Matlab, we can not imagine that it is useful to launch more than 21 *workers* for this problem. The optimal number of *workers* might be far less than 21. Nevertheless, we observe that the performance is quite acceptable; the observed computation time is just 9/4 times of the theoretical predicted of 9.7588 seconds by Amdahl's Law for eight *workers*. But, of course, far beyond of its limit from 2.8915 seconds. The above timing would almost coincide with the predicted timing when only 70 percent of the code could be parallelized, that is, for $p = 0.7$. However, a profiler analysis shows that despite a total elapsed time of 22 seconds, there is at least a 10 percent communication overhead in the parallel computation process, which can still be reduced. We close this discussion by noticing that the timing can never be better than 5.5076 seconds when using 21 *workers* (CPU cores).

Now, it remains to verify that the solution found is a pre-kernel element. First, we increase the precision tolerance slightly from $10^6 \cdot eps$ to $10^7 \cdot eps$ to overcome numerical precision problems. This is done while setting:

```
>> tol=10^7*eps

tol =

  2.2204e-09
```

We will check now the validity of our result both by serial as well as by parallel computation. In a first step we issue the function *PrekernelQ()*.

```
>> tic;prkQ_v3=PrekernelQ(v3,p_prk_v3,tol);toc
Elapsed time is 12.065575 seconds.
```

The elapsed computation time is about 12 seconds. Executing this computation in parallel with the command *p_PrekernelQ()* more than bisects the elapsed time as can be seen next.

```
>> tic;p_prkQ_v3=p_PrekernelQ(v3,p_prk_v3,tol);toc
Elapsed time is 4.803003 seconds.
```

According to the timing from above, identical code segments on the relevant parts, the fact that the sweep to determine the pre-kernel of this game needed in total 5-iterations to complete, we can conclude with the remedy of the profiler toolbox that about 99 percent of a sweep to find a pre-kernel element has been spent to compute the maximum surpluses and the set of most effective coalitions; only the remaining 1 percent of the elapsed time has been assigned for setting up the system of linear equations and to solve it. Moreover, the latter command is not only more efficient, it reproduces also the same result as its serial counterpart.

```
>> prkQ_v3

prkQ_v3 =

   1

>> p_prkQ_v3
```

```
p_prkQ_v3 =

   1
```

According to the fact that the modest bankruptcy game v_3 was derived from a bankruptcy situation, we want to verify the results from above with the *Talmudic_Rule()*. In order to avoid that Matlab will complain that the recursion limit has been exceeded with the consequence that the computation will break down, we provide this function with the tolerance level from above. With this additional argument we make sure that the upcoming computation terminates properly. Remind that we have to supply this function with the parameter of the corresponding bankruptcy situation and not with the game parameter. Issuing

```
>> tic;tlm_rl3=Talmudic_Rule(E3,d3,tol);toc
Elapsed time is 0.186595 seconds.
>> tlm_rl3

tlm_rl3 =

 Columns 1 through 12

  12.5000  18.5000  16.5000  11.0000  14.0000  35.0000  37.5152  45.5152  63.5152 105.5152  28.0000  34.5000

 Columns 13 through 21

  22.0000  90.5152  98.5152 112.5152 149.5152  27.0000  60.5152 136.5152 143.5152
```

we realize that this function needs no parallel boost. Even in serial, the computational performance outranges the boldest expectation, and outspeeds any parallel computation in seeking the pre-nucleolus for a bankruptcy game. In principle, there would be no need to activate the parallel mode to determine the pre-nucleolus of a bankruptcy game nor to use the corresponding serial command *PreKernel()*.

We won't close this discussion without mentioning the Shapley value. Although, its computational process proceeds completely in serial, we could nevertheless compute it in less than 47 seconds. Still less than the time needed to complete a pre-kernel element computation in serial with 57.8 seconds, but more than twice the time required to complete a pre-kernel computation that proceeds in parallel with eight workers.

```
>> tic;sh_v3=ShapleyValue(v3);toc
Elapsed time is 46.954827 seconds.

>> sh_v3

sh_g4 =

 Columns 1 through 12

  16.4567  24.3900  21.7430  14.4769  18.4378  46.3291  49.6697  55.0241  67.1157  95.5868  36.9991  45.6616

 Columns 13 through 21

  29.0284  85.3746  90.8151 100.3706 125.8594  35.6691  65.0961 116.8622 121.7009
```

3.2. (Average-) Convexity and Consistency Reconsidered

Studying some game properties is computationally more intensive than computing game solutions. This let us expect that especially in these situation we profit most when launching a calculation in parallel rather

than in serial. We have alredy observed that the efficiency gain exceeds largely the 50 percent level while computing some game solutions in parallel. To show that we can also expect a large performance gain for the computation of some game properties by executing code in parallel, we firstly rely on the verification of the convexity property of bankruptcy games. For being a modest bankruptcy game, game v_3 must be convex. By verifying this, we issue for the serial mode:

```
>> tic;cQ_v3=convex_gameQ(v3);toc
Elapsed time is 86.492177 seconds.

>> cQ_v3

cQ_v3 =

   1
```

whereas for a computation in parallel, the command *p_convex_gameQ()* is executed through:

```
>> tic;p_cQ_v3=p_convex_gameQ(v3);toc
Elapsed time is 13.990107 seconds.

>> p_cQ_v3

p_cQ_v3 =

   1
```

Both functions return the expected result. More striking is the gain in performance in parallel compared to a serial computation. Apparently, the efficiency gain is about 84 percent. This gain is even higher for very intensive computation like the verification of the average convexity property or for one of the consistency property that lasts hours rather than seconds or minutes. Making this point more clear, let us treat a bankruptcy situation with 16-person to study average-convexity as well as the DM-reduced game property for the pre-nucleolus and the Shapley value. We refrained from listing serial computation, since the computation in serial lasts 25.61 minutes when verifying average-convexity and more than 50 minutes for verifying DM-RGP. Nevertheless, to make the gain in performance somehow visible, we let run in a first step the commands in parallel using four *workers* and in the next step while launching eight *workers*.

Before, we focus on these issues we will close the running parallel Matlab session. To quit a parallel Matlab session, we execute finally:

```
>> matlabpool close
Sending a stop signal to all the labs ... stopped.
```

Launching a new parallel computation with only four *workers*, we issue on the command window:

```
>> matlabpool open 4
Starting matlabpool using the 'MyConf1' configuration ... connected to 4 labs.
```

In the sequel, we enlist in the first place the results obtained while using four *workers*, however, in the second place the corresponding results are listed while invoking eight workers. Apparently, this is done, to compare immediately the derived results. For sake of convenience, we derived during a first run all results related to a four workers session and in the subsequent session all results which are derived by calling eight workers rather than switching consecutively between the desired number of workers. To start with, we introduce the following bankruptcy situation:

```
>> d4

d4 =

  25   37   33   22   28   70   75   83  101  143   56   69   44  128  136  150

>> E4

E4 =

  800
```

Remind that a bankruptcy game is convex, therefore it must also be average-convex. When we run the execution in parallel with four *workers*, the timing is about 7 minutes. But when we let run eight *workers* in parallel the elapsed time 3.72 minutes. Thus, the computation time is almost bisected as can be immediately checked out while inspecting the following lines:

```
>> tic;p_avQ_v4=p_average_convexQ(v4);toc
Elapsed time is 423.860037 seconds.

>> p_avQ_v4

p_avQ_v4 =

   1

>> tic;p_avQ_v4=p_average_convexQ(v4);toc
Elapsed time is 223.6023 seconds.

>> p_avQ_v4

p_avQ_v4 =

   1
```

Recall that the elapsed time to verify the average-convexity property in serial is about 25.61 minutes indicating an overall performance gain of about 85.45 percent for a parallel computation using eight workers compared to a serial computation. However, the efficiency gain for four workers is approximately 72.4 percent.

After we have treated the average-convexity, we want now turn our attention to DM-RGP. For this purpose, we must again compute the single-valued pre-kernel and the Shapley value of game v_4. Firstly, we compute the unique pre-kernel point by relying on serial as well as on parallel computation.

```
>> tic;prk_v4=PreKernel(v4);toc
Elapsed time is 1.873194 seconds.

>> prk_v4

prk_v4 =

 Columns 1 through 12

  12.5000  18.5000  16.5000  11.0000  14.0000  39.1667  44.1667  52.1667  70.1667 112.1667  28.0000  38.1667

 Columns 13 through 16

 22.0000  97.1667  105.1667 119.1667
```

```
>> tic;p_prk_v4=p_PreKernel(v4);toc
Elapsed time is 2.209448 seconds.

> p_prk_v4

p_prk_v4 =

 Columns 1 through 12

  12.5000  18.5000  16.5000  11.0000  14.0000  39.1667  44.1667  52.1667  70.1667 112.1667  28.0000  38.1667

 Columns 13 through 16

 22.0000  97.1667  105.1667 119.1667
```

By comparing the elapsed times, there is no advantage for switching on the parallel mode. Verifying this result with the *Talmudic_Rule()* command, getting:

```
>> tic;tlm_rl4=Talmudic_Rule(E4,d4);toc
Elapsed time is 0.127701 seconds.

>> tlm_rl4

tlm_rl4 =

 Columns 1 through 12

  12.5000  18.5000  16.5000  11.0000  14.0000  39.1667  44.1667  52.1667  70.1667 112.1667  28.0000  38.1667

 Columns 13 through 16

 22.0000  97.1667  105.1667 119.1667
```

In the next step, we execute the command *p_Reduced_game_propertyQ()* for checking DM-RGP for the vector *prk_v4* using four Matlab *workers*, we notice that the elapsed time is about 705 seconds or 11.75 minutes. A considerable increase in performance compared to the 50 minutes in serial computation.

```
>> tic;[RGP1_v4 RGPC1_v4]=p_Reduced_game_propertyQ(v4,prk_v4,'PRN');toc
Elapsed time is 705.036770 seconds.

>> RGP1_v4

RGP1_v4 =

  rgpQ: 1
  rgpq: [1x65535 double]
```

As expected, DM-RGP is confirmed. Launching four additional Matlab *workers* reproduces the result from above and almost bisects the elapsed time. Instead of 11.75 minutes, the computation was completed in 6 minutes. Now, the gain in efficiency is about 88 percent instead of 76.5 percent.

```
tic;[RGP1_v4 RGPC1_v4]=p_Reduced_game_propertyQ(v4,prk_v4,'PRN');toc
Elapsed time is 360.342181 seconds.

>> RGP1_v4
```

```
RGP1_v4 =

  rgpQ: 1
  rgpq: [1x65535 double]
```

Similar for the Shapley value, which is given by the vector:

```
>> tic;sh_v4=ShapleyValue(v4);toc
Elapsed time is 1.481795 seconds.
>> sh_v4

sh_v4 =

 Columns 1 through 12

  16.4075  24.3360  21.6892  14.4303  18.3861  46.3380  49.6979  55.0906  67.2997  96.2784  36.9689  45.6672

 Columns 13 through 16

  28.9793  85.8452  91.3983 101.1874
```

After 709 seconds the validity of DM-RGP was rejected with four *workers*, as can be seen next:

```
>> tic;[RGP2_v4 RGPC2_v4]=p_Reduced_game_propertyQ(v4,sh_v4,'PRN');toc
Elapsed time is 709.484578 seconds.
>> RGP2_v4

RGP2_v4 =

  rgpQ: 0
  rgpq: [1x65535 double]
```

Using eight *workers*, we get the same result in almost 6 minutes. Again, the improvement in performance is about 88 percent.

```
>> tic;[RGP2_v4 RGPC2_v4]=p_Reduced_game_propertyQ(v4,sh_v4,'PRN');toc
Elapsed time is 361.950656 seconds.

>> RGP2_v4

RGP2_v4 =

  rgpQ: 0
  rgpq: [1x65535 double]
```

3.3. Replication of a Pre-Kernel Element Reconsidered

In Subsection 2.7 we have discussed the command *replicate_prk()*, which allows a replication of a pre-kernel element. This command also has a parallel counterpart, which is called *p_replicate_prk()*, and which is, depending on the number of Matlab *workers*, at least three/four times faster than its serial function. Apart from any efficiency consideration, this function allows also to circumvent some problems related to physical memory requests for relative large games. For instance, the serial command *replicate_prk()* requires for game v_4, which is a 16-person game, at least 132 GB physical memory. In order to avoid such a huge memory request, one needs just to call function *p_replicate_prk()* with only one *nargout* argument, that is, it is enough to issue

```
>> RepSol=p_replicate_prk(v4,prk_v4,1,1)
```

In that case, the structure variable *RepSol* returns only a summary of the results, that is, a sequence of ones and/or zeros, contained in the output variable *V_PrkQ* indicating whether the derived game replicates the pre-kernel element, and the output variable *SBCQ* indicates whether the equivalence class has remained unchanged. However, when one wants restore the original behavior, one has to call the function with two *nargout* arguments, that is, issuing

```
>> [RepSol RepBMat]=p_replicate_prk(v4,prk_v4,1,1)
```

In that case, the structure variable *RepBMat* contains the matrices output. A note on caution is required here, based on the fact that the data output is considerably huge, which means that one needs at least 114.5 GB on available disk space to save the whole data set. Moreover, one should be aware that saving a variable that is larger than 2 GB requires to invoke the *-v7.3* option of the *save()* command, whereas the saving operation lasts minutes rather than seconds. However, writing on disk space is in general unproblematic, whereas reading huge files could cause some problems, if a certain threshold is passed. According to our experience, we encountered no problems to read mat-files up to a size of 6 GB, but failed to do so for larger mat-files.

Let us consider now a concrete example. For that purpose, resume game v_4 from the previous subsection, setting the scaling factor $\mu = 1$, and finally calling

```
>> tic;RepSol_v4=p_replicate_prk(v4,prk_v4,1,1);p_rpt=toc;
```

This computation lasts about 11516 seconds or to state it differently 3.1 hours to complete while calling eight Matlab *workers*, as can been seen next:

```
>> p_rpt

p_rpt =

  1.1516e+04

1.1516e+04/60

ans = 191.93
```

Furthermore, due to some memory release operations, we could reduce the memory request from 134 GB to 64 GB. As indicated by the foregoing discussion, only the summarized results are returned.

```
>> RepSol_v4

RepSol_v4 =

  V_PrkQ: [1x65519 double]
   SBCQ: [1x65519 double]
```

From these results, we can grasp the information that in total 65519 linear independent games have been constructed from the default game, and for each game the equivalence class has been determined that contains the payoff *prk_v4*. In the next step, observe, whether all of these games possess the payoff *prk_v4* as a pre-kernel element. And indeed, this is confirmed while executing

```
>> all(RepSol_v4.V_PrkQ)

ans =

   1
```

or while counting the number of all non-zero elements, that is, all ones in the output variable RepSol_v4.V_PrkQ, hence

```
>> nnz(RepSol_v4.V_PrkQ)

ans =

    65519
```

It is every time a correct conduct to close a parallel session after the work has been done. We quit it by issuing:

```
>> matlabpool close
Sending a stop signal to all the labs ... stopped.
```

3.4. Replication of the Shapley Value Reconsidered

Similar to the possibility of invoking a replication of a pre-kernel element in parallel, we implement a function that also allows this kind of computation for the Shapley value. By Subsection 2.8 we have learned that the command *replicate_Shapley()* allows the replication of the Shapley value by a spanning system of linear independent games within the game space. This command also has a parallel counterpart, which is called *p_replicate_Shapley()*. This function is designed to circumvent efficiency considerations and problems related to physical memory requests for relative large games. For instance, the serial command *replicate_Shapley()* demands for the game v_4, which is a 16-person game, at least 82 GB physical memory. With the parallel command *p_replicate_Shapley()* we can reduce this value to 50 GB. However, the behavior of this function is different from that of *p_replicate_prk()*. In order to reduce the memory request the latter requires a function call with one *nargout* arguments, the former, however, requires a function call with four *nargin* arguments, whereas the last argument must be a string, namely *'red'* which induces a reduced memory demand and a resume of results. Instead of returning two matrices with a size of more than 32 GB in total only the summary of the computation will be returned as a logical value and sequence of ones/zeros (cf. Subsection 2.8).

Even though the Shapley value computation is based on the potential method the elapsed time to compute it is up to the factor 15 slower than a pre-kernel evaluation. For this reason, we use 8 Matlab workers to speed up the computational process of the Shapley value replication.

```
>> matlabpool open MyProfile2 8;
Starting matlabpool using the 'MyProfile2' profile ... connected to 8 workers.
```

Recall that the Shapley value of the game v_4 is given by

```
>> sh_v4

sh_v4 =
```

```
 Columns 1 through 12

  16.4075  24.3360  21.6892  14.4303  18.3861  46.3380  49.6979  55.0906  67.2997  96.2784  36.9689  45.6672

 Columns 13 through 16

  28.9793  85.8452  91.3983 101.1874
```

Getting a replication of the Shapley value of the original game, the function *p_replicate_Shapley()* must be called with at most four *nargin* arguments. In order to avoid a huge memory request the last input argument must be set to *'red'*, otherwise *'full'*, or empty. Therefore, one issues for instance

```
>> tol=10^8*eps;
>> tic;RepShap=p_replicate_Shapley(v4,25,tol,'red');p_rpt16=toc;
```

Here, the first input argument is the default game, the second the scaling factor to derive all linear independent games that replicates the Shapley value under consideration, and the value *'tol'* specifies a tolerance value. This value is set to $10^8 * eps$, which is also its default value.

```
>> p_rpt16

p_rpt16 =

   1.2212e+04

>> p_rpt16/60

ans =

  203.5414
```

The total elapsed time to terminated this process of a Shapley value replication lasts about 204 minutes while using 8 Matlab workers.

In that case, the structure variable *RepShap* returns only a summary of the results, that is, the value true or false in the output variable *RepShap.shQ*, and a sequence of ones and/or zeros, contained in the output variable *RepShap.shQv* indicating whether the derived game replicates the Shapley value. The Shapley values and game space matrix are not returned in order to save physical memory and space on the hard disk.

```
>> RepShap

RepShap =

    shQ: 1
   shQv: [1x65520 logical]
```

The foregoing result states that the Shapley value of the default game is replicated for the all games contained in the null space of the Shapley value.

However, when one wants restore the original behavior, one has to call the function with at least one or at most four *nargin* arguments. In the latter case, the string argument must be set to *'full'*. The default behavior, however, is *'full'*. In this case, the complete list of results is returned. Here, an estimated data set of 32 GB is created that must be saved with the *-v7.3* option of the *save()* command to write it on the hard disk. Thus, to attain the complete data set we set the string argument to *'full'*, and issuing

```
>> RepShap=p_replicate_Shapley(v4,25,tol,'full');
```

to get the same result as above together with the resultant matrices. It is also enough to call the function with two argument as shown below

```
>> RepShap=p_replicate_Shapley(v4,25);
```

If one uses only one argument, that is, the original game, then all default values will be set. In this case the scaling factor μ is set to 1.

Finally, we close the *matlabpool* session with

```
>> matlabpool close
Sending a stop signal to all the labs ... stopped.
```

3.5. Weighted Majority Games: Game Solutions Reconsidered

Up to now we have mainly treated bankruptcy or assignment games. In the upcoming subsection, we will now turn our attention to weighted majority games instead, in order to demonstrate the universal approach of the toolbox. That is, not only bankruptcy or assignment games can be investigated but also any game class should be treatable. Of course, relying on an universal design has the drawback that a particular problem cannot be as efficiently treated as with a custom-build design, which is especially designed for to solve a particular class of problems. Following this universal philosophy, we refrained, for instance, to write a function that is based on Wolsey's algorithm to compute the nucleolus for simple games. Especially under the consideration that an implementation of this algorithm would have required a lot of effort. An effort to be out of all proportion to the gain we would obtain by computing only for simple games the nucleolus very efficiently. In our view, the implemented Algorithm 8.1.1 of Meinhardt (2013) is efficient enough to compute a pre-kernel element for any imaginable game class in a reasonable amount of time and not just the nucleolus for a particular game class (see Appendix A). As we will learn from the forthcoming example, the function *PreKernel()*, which implements Algorithm 8.1.1, computes a pre-kernel element for large weighted majority games in average 20 percent faster than for bankruptcy games. To illuminate this fact, we have chosen a 22-person weighted majority game as an appropriated example.

Recall that a simple game is a weighted majority game, if there exists a quota and positive weights such that a coalition is called a winning coalition whenever the sum of weights or votes pass the quota. The following vector enlists the votes or weights of all 22-person involved in a ballot.

```
>> wgs

wgs =

  48   45   41   38   31   28   25   25   21   18   16   16   13   10    8    8    6    5    4    2    1    1
```

In order to pass a bill, we set the quota to:

```
>> qt

qt =

  104
```

The function *weighted_majority()* computes the weighted majority game for us. According to the enormous number of coalitions involved, we suppressed the game representation on the screen by a semicolon.

```
sv=weighted_majority(qt,wgs);
```

The computation of a pre-kernel element in serial is approximately completed after 2.67 seconds. In comparing this timing with the elapsed average computation time of a pre-kernel element for 22-person weighted majority games as given in Appendix A, we realize that this value is relatively close to the reported average of 2.4 seconds.

```
>> tic;prk_sv=PreKernel(sv);toc
Elapsed time is 2.665380 seconds.
```

The evaluated pre-kernel element is given by:

```
>> prk_sv

prk_sv =

 Columns 1 through 14

  0.1171   0.1098   0.1000   0.0927   0.0756   0.0683   0.0610   0.0610   0.0512   0.0439   0.0390   0.0390   0.0317   0.0244

 Columns 15 through 22

  0.0195   0.0195   0.0146   0.0122   0.0098   0.0049   0.0024   0.0024
```

Invoking now four workers in parallel for the same task is now accomplished after 3.1 seconds indicating an overall performance loss of approximately 16 percent. Therefore, running a pre-kernel evaluation in serial with 8 threads is faster than running the same computation in parallel with four cores and one thread on each core. However, if we run this task in serial with only one thread, then the computation is considerably slower.

```
>> tic;p_prk_sv=p_PreKernel(sv);toc
Elapsed time is 3.095911 seconds.

>> all(abs(p_prk_sv-prk_sv<tol))

ans =

   1
```

Of course, the parallel mode reproduces the result obtained from serial computation.

Turning to the Shapley value, we observe that the computation in serial is terminated considerably slower than for a pre-kernel element computation in serial as well as in parallel using four workers. Thus, similar to convex games, the rather difficult task to compute a pre-kernel point can now be done faster than the determination of the Shapley value when performing its calculation in serial.

```
>> tic;sh_sv=ShapleyValue(sv);toc
Elapsed time is 79.985266 seconds.

>> sh_sv

sh_sv =
```

```
 Columns 1 through 14

  0.1226   0.1140   0.1030   0.0950   0.0755   0.0677   0.0603   0.0603   0.0502   0.0429   0.0376   0.0376   0.0305   0.0232

 Columns 15 through 22

  0.0182   0.0182   0.0136   0.0114   0.0092   0.0045   0.0022   0.0022
```

Is it possible to improve the performance while launching some additional workers? To see this, we call four additional workers for a pre-kernel element computation in parallel. By the evaluation below we observe that we do even worse. Now, the elapsed time is reduced to 3.3 seconds indicating a total efficiency loss of slightly more than 21 percent compared to a serial computation and of 6 percent to the parallel computation with four workers.

```
>> tic;p_prk_sv=p_PreKernel(sv);toc
Elapsed time is 3.284528 seconds.

>> all(abs(p_prk_sv-prk_sv<tol))

ans =

   1
```

The increase is due to the additional communication overhead in the parallel computation process compared to the situation with four workers. The problem is still too small to obtain an overall performance gain in parallel. Such a gain can only be observed for a pre-kernel computation with 23 players onward (cf. Appendix A). Nevertheless, for both modes we improved considerably its performance simply by optimizing the code basis. Compare this with the 130 seconds in serial and 43 seconds respectively for version 0.2, or compare the tables in Appendix A.

Apart from any computation of the pre-kernel or Shapley value, we can also compute the Banzhaf value in serial as well as in parallel. Calling eight workers in parallel gives us a performance gain of 76.5 percent for the computation of the Banzhaf value in comparison to the serial mode. Considerably more than the efficiency gain obtained by a pre-kernel point computation in parallel. Although, this value seemed very impressing at a first glance, a scrutiny reveals immediately that even in serial the computation of the Banzhaf value is very fast while needing not more than 6.5 seconds to complete.

```
>> tic;bzh_sv=banzhaf(sv);toc
Elapsed time is 6.401321 seconds.
>> tic;p_bzh_sv=p_banzhaf(sv);toc
Elapsed time is 1.564119 seconds.
```

Both calculations return the same result as revealed by the following lines:

```
>> bzh_sv

bzh_sv =

 Columns 1 through 14

  0.1049   0.1012   0.0953   0.0903   0.0766   0.0700   0.0632   0.0632   0.0538   0.0464   0.0415   0.0415   0.0338   0.0262

 Columns 15 through 22

  0.0210   0.0210   0.0158   0.0131   0.0105   0.0053   0.0026   0.0026
```

```
>> bzh_sv==p_bzh_sv

ans =

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

Now let us turn to the evaluation of some selected game values w.r.t. a coalition structure. All game values that can be computed in serial can also be computed in parallel (see Subsection 2.6). In the course of this discussion we focus on the parallel evaluation of the Aumann-Drèze, (weighted) Owen, and coalition solidarity value. To compute such a game value, we need to specify a partition of the player set N. We choose the following partition

```
P={[1 2 3],[4 5],[6 7],[8 9 10],[11 12],[13 14],[15 16],[17 18],[19 20],[21 22]};
```

Do not forget the transcription of the coalitions to its Matlab representation, which is accomplished through

```
>> pm=clToMatlab(P)

pm =

      7      24      96     896    3072   12288   49152  196608  786432 3145728
```

We start with the Aumman-Drèze value in serial as well in parallel. This evaluation is executed by

```
>> tic;ad_vl=ADvalue(sv,pm);toc
Elapsed time is 1.429700 seconds.
>> tic;p_ad_sv=p_ADvalue(sv,pm);toc
Elapsed time is 1.537764 seconds.
>> p_ad_sv

p_ad_sv =

 Columns 1 through 14

  0.3333   0.3333   0.3333        0        0        0        0        0        0        0        0        0        0        0

 Columns 15 through 22

       0        0        0        0        0        0        0        0
```

Hence, for the Aumann-Drèze value, we observe no real performance gain under a parallel evaluation. The timing is very close to the serial computation.

In contrast, the computation of the Owen value in parallel offers to us a performance gain of 55 percent in comparison to the serial mode. The reader certainly want to compare the payoffs of the Owen value with that of the Shapley value and the pre-kernel point.

```
>> tic;ov_sv=OwenValue(sv,pm);toc
Elapsed time is 12.961957 seconds.
>> tic;p_ov_sv=p_OwenValue(sv,pm);toc
Elapsed time is 5.784977 seconds.
>> p_ov_sv

p_ov_sv =

 Columns 1 through 14
```

```
  0.1424   0.1337   0.1231   0.0869   0.0683   0.0599   0.0567   0.0550   0.0473   0.0413   0.0359   0.0359   0.0244   0.0185

 Columns 15 through 22

  0.0147   0.0147   0.0123   0.0091   0.0093   0.0046   0.0030   0.0030
```

Next, we demonstrate how we can compute the weighted Owen value in serial as well as in parallel. There is no difference in the calling between the Owen and weighted Owen value. Both requires two input arguments, namely the TU game under consideration and a coalition structure. The difference between both functions is that the latter internally computes the weights from the supplied coalition structure which is applied to divide the proceeds of cooperation in accordance with the weighted Shapley value for the associated quotient game.

```
>> tic;wov_sv=weightedOwen(sv,pm);toc
Elapsed time is 14.205009 seconds.
>> tic;p_wov_sv=p_weightedOwen(sv,pm);toc
Elapsed time is 6.817728 seconds.
>> p_wov_sv

p_wov_sv =

 Columns 1 through 14

  0.1093   0.1005   0.0900   0.1045   0.0859   0.0677   0.0645   0.0567   0.0489   0.0430   0.0440   0.0440   0.0300   0.0240

 Columns 15 through 22

  0.0181   0.0181   0.0145   0.0114   0.0112   0.0064   0.0037   0.0037
```

The performance gain in parallel is similar to that of the computation of the Owen value. In this particular case, we realize an overall gain of 52 percent instead. The extra time is needed to compute the weights and the weighted Shapley value of the quotient game instead of the Shapley value.

We close this subsection while presenting the performance gain of the computation of the coalition solidarity value in parallel. Here, the evaluation speeds up by 61 percent.

```
>> tic;csl_sv=CoalitionSolidarity(sv,pm);toc
Elapsed time is 13.098777 seconds.
>> tic;p_csl_sv=p_CoalitionSolidarity(sv,pm);toc
Elapsed time is 5.067205 seconds.
>> p_csl_sv

p_csl_sv =

 Columns 1 through 14

  0.1366   0.1334   0.1293   0.0822   0.0729   0.0591   0.0575   0.0509   0.0476   0.0451   0.0359   0.0359   0.0229   0.0199

 Columns 15 through 22

  0.0147   0.0147   0.0115   0.0099   0.0081   0.0058   0.0030   0.0030
```

By the proceeding discussion we have learnt that a parallelization can speed up the evaluation of a solution. However, this depends crucially on the size of problem and of the initializing process, communication overhead, and network traffic. In the case of the pre-kernel evaluation, however, we saw that a simple code optimization in serial and parallel boosts the computation by an average factor of 30 (cf. Appendix A).

3.6. FOR-LOOP VERSUS PARFOR-LOOP

This subsection is devoted to the issue how we can use a *parfor-loop* in combination with a serial command to benefit nevertheless from a parallel computation. These results will be confronted to a simple *for-loop* while issuing a parallel command. Finally, we investigate the usefulness of a *parfor-loop* in combination with a parallel command and set these results in comparison to the former ones. As a by product, we will learn how we can profit from executing a command in parallel without going too deeply into the details of parallel computation. The reader should notice unless stated otherwise that all of the forthcoming results have been derived while having launched four Matlab *workers*.

As mentioned above, a parallel computation under Matlab has a one dimensional structure. This means that a *parfor-loop* cannot be nested or make a reference to a nested function. For instance, a simple *for-loop* allows a nested structure like this:

```
for ii=1:n
 for jj=1:m
  < for-loop body >
 end
end
```

Due to its one dimensionality parallel computing structure, a nested *parfor-loop* is not allowed.

```
parfor ii=1:n
 parfor jj=1:m
  < parfor-loop body >
 end
end
```

Hence, the above nested *parfor-loop* cannot be executed. Matlab would break down with an error message. One dimensionality means that, for instance, the following loop construct is admissible:

```
parfor ii=1:n
 for jj=1:m
  < for-loop body >
 end
end
```

Moreover, assume that *p_func()* is a parallel nested function that will be called from a *parfor-loop* body, then making no reference to a nested function means that this function will be called by means of a normal function handle, that is, function *p_func()* will be executed in serial and not in parallel, although this function contains code segments that can be run in parallel.

```
parfor ii=1:n
 p_funct(ii)
end
```

Thus, the above loop will be executed in parallel, but the function *p_func()* will be not. For doing so, one has to issue this function from outside of any *parfor-loop* construct. Calling this function as a stand-alone function will do the job, that is, when executing:

```
p_funct(1)
```

After we have understood these preliminaries, we devote our attention to a concrete example. This example shows how one can use the parallel mode to seek for the whole pre-kernel while relying on a grid of distinct starting points. Of course, this make not so much sense for bankruptcy games, since the pre-kernel coincides with the pre-nucleolus, therefore we have already determined the whole pre-kernel by the computation from above, which is unique and can serve therefore as a reference point. Apparently, the forthcoming example should simply be considered as illustrative. Furthermore, we want demonstrate the robustness of the pre-kernel functions, because finding a pre-kernel point for a bankruptcy game is due to the enormous number of equivalence classes involved not as easy as for weighted majority games, for instance. Hence, the probability of a failure is higher under a bankruptcy game than under a weighted majority game whenever the program does not apply a norm check (cf. Meinhardt (2013)). For efficiency as well as for memory management reasons, we have refrained from implementing a norm check, which would have required a perfect recall of all calculation performed during a sweep.

The grid points will be generated by the following simple matrix operation.

```
>> idm=eye(16)*v4(end);
```

Having constructed a grid of different starting points, we call for reference reasons, a serial *for-loop*. All iterations will be conducted in succession, that is, a new iteration with a distinct starting point starts at that moment as the previous iteration has been successfully terminated. Indicating that we need 16 sweeps to finish all iterations in serial.

```
>> tic; for k=1:16 prk_mat(k,:)=PreKernel(v4,idm(k,:));end; toc
Elapsed time is 27.312601 seconds.
>> prk_mat(1,:)

prk_mat(1,:) =

 Columns 1 through 12

  12.5000  18.5000  16.5000  11.0000  14.0000  39.1667  44.1667  52.1667  70.1667 112.1667  28.0000  38.1667


 Columns 13 through 16

  22.0000  97.1667  105.1667 119.1667
```

The elapsed time is about 27.3 seconds, and all 16 starting points converge to the unique pre-kernel point. Hence, the average computing time is about 1.7063 seconds.

In the next step, we call the serial command *PreKernel()* from inside of a *parfor-loop*. Again, all starting points converge. Moreover, the overall computation time is reduced from 27 seconds to 8.35 seconds whereas the average computing time increases from 1.7 to 2.09 seconds, since we have launched four *workers*, four iterations are allocated to each worker, hence, we need in total four sweeps to complete. Obviously, the increase in the average is due to communication overhead and network traffic.

```
 >> tic; parfor k=1:16 prk_mat_pf(k,:)=PreKernel(v4,idm(k,:));end; toc
Elapsed time is 8.358497 seconds.
>> all(all(prk_mat_pf==prk_mat))

ans =

   1
```

In the next step we investigate the call of the parallel nested function *p_PreKernel()* from inside of a *parfor-loop*.

```
>> tic; parfor k=1:16 p_prk_mat_pf(k,:)=p_PreKernel(v4,idm(k,:));end; toc
Elapsed time is 8.426119 seconds.

>> all(all(p_prk_mat_pf==prk_mat))

ans =

   1
```

The result is very close to the previous result, if we call the parallel nested function *p_PreKernel()* from inside of a *parfor-loop*. Remind the discussion from above. No reference is made, this function is called by means of a normal function handle, it runs in serial rather than in parallel. Thus, we notice that there is no efficiency gain when invoking a parallel nested function in parallel. Especially, for large problems we must expect an efficiency loss that is much higher than that we observed for this example.

We complete by considering the effect when calling the parallel function *p_PreKernel()* from inside of a serial *for-loop*.

```
>> tic; for k=1:16 p_prk_mat(k,:)=p_PreKernel(v4,idm(k,:));end; toc
Elapsed time is 34.421083 seconds.
>> all(all(p_prk_mat==prk_mat))

ans =

   1
```

We observe that not only the total elapsed time is highest with 34.4 seconds, this call imposes even the highest average computing time of 2.1513 seconds per iteration. Here, the communication overhead and network traffic outweighs the efficiency gain by distributing the iterations to four Matlab workers.

3.7. How many Matlab Workers are Optimal?

This question is not easy to answer, since it depends at least on the processor speed, the physical memory available, how computationally intensive the problem is, and how many data are transferred across the network.[10] But due to its one dimensional parallel computing structure and the problems involved in game theory, we guess that the optimal number of workers is less than n for games larger than $n \geq 10$ when investigating game properties, and $n \geq 20$ for the study of game solutions. Moreover, due to fact that we are restricted on our system up to eight Matlab *workers*, and due to our experience that let us suggest that the optimal number of workers must be larger than eight, we can only try to deduce the exact number. We are trying to approximate the optimal number *workers* from running a problem with one, four and eight workers while looking on the marginal rate of performance gain. As we will realize by the experiment from below, the marginal rate of performance gain is quickly decreasing, which let us suggests that for this experiment the optimal number of works must lie within the range of 12 and 18 workers. We think that this result should also be valid for the computation of game solutions when the efficiency is not as steep and the marginal rate of performance decreases almost lineally.

Again, we generate a grid of starting points to seek for a pre-kernel element. This experiment is based on the game v_3, which is a 21-person game. This game is large but not too large, and therefore has the appropriate size to investigate the above question in a reasonable time.

[10] For more details see the user guides of the Parallel Computing Toolbox and the Distributed Computing Server. For a quick overview, we recommend the site: http://www.mathworks.com/support/solutions/en/data/1-10ELAF/index.html?solution=1-10ELAF.

```
idm21=eye(21)*v3(end);
```

First, we execute eight times consecutively the serial *PreKernel()* command, each having a distinct starting points. We have already demonstrated that the single-valued pre-kernel can be computed with that function in 127 seconds, this let us suppose that a serial sweep needs at least 17 minutes to complete.

```
>> tic; for k=1:8 prk_v3_mat(k,:)=PreKernel(v3,idm21(k,:));end; toc
Elapsed time is 1068.360612 seconds.

>> 1068.360612/8

ans =

 133.5451
```

And indeed, the serial computation to seek for the pre-kernel point lasts about 18 minutes, whereas the average computing time is about 133.5 seconds, which is slightly more than the 127 seconds from above. All starting points converge to the pre-kernel point as can be checked out by the following matrix:

```
>> for k=1:8 eqQ(k,:)=abs(prk_v3_mat(k,:)-prk_v3)<tol; end
>> eqQ

eqQ =

   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   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   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

In the second step, we do the same calculation as above in parallel while launching in total four workers.

```
>> tic; parfor k=1:8 prk_v3_mat_pf(k,:)=PreKernel(v3,idm21(k,:));end; toc
Elapsed time is 343.884823 seconds

>> 343.884823/2

ans =

 171.9424
```

Now, the elapsed time is about 5.7 minutes, giving an overall performance gain of approximately 68 percent. Since, we have invoked four workers, two sweeps are needed to complete the eight iterations, this imposes an average computing time of about 172 seconds in comparison to the 133.5 seconds obtained by the serial run.

```
>> for k=1:8 eqQ(k,:)=abs(prk_v3_mat_pf(k,:)-prk_v3)<tol; end
>> all(all(eqQ))

ans =

   1
```

The check for computational correctness reveals that the solution matrix coincides with those derived by the first run.

In the next step, we devote our attention to the identical problem while launching eight workers. We have in total eight iterations and eight workers, which means that we need only one sweep in parallel to complete.

```
>> tic; parfor k=1:8 prk_v3_mat_pf(k,:)=PreKernel(v3,idm21(k,:));end; toc
Elapsed time is 228.157181 seconds.
```

The total and the average elapsed time is approximately 3.8 minutes. In reference to the serial computation from above, the overall gain in efficiency is about 78.6 percent, however, in comparison to the previous computation with four workers the efficiency gain is only about 10.7 basis points indicating a decreasing marginal rate of performance. Extrapolating these figures, it is certainly not unrealistic to assume that the optimal number of workers must be larger than eight and that this number is within the range of 12 and 18 workers.

```
>> for k=1:8 eqQ(k,:)=abs(prk_v3_mat_pf(k,:)-prk_v3)<tol; end
>> all(all(eqQ))

ans =

   1
```

The obligatory test of correctness reveals that the convergence is successfully established.

We close this section by demonstrating that all starting points from the grid will convergence to the single-valued pre-kernel. In contrast to an expected computing time of approximately 46.5 minutes in serial, this will be achieved in parallel calling eight workers within 9.7 minutes. This indicates a performance gain of approximately 79.1 percent. This figure confirms our first observation when only 8 iterations have been considered.

```
>> tic; parfor k=1:21 prk_v3_mat_pf(k,:)=PreKernel(v3,idm21(k,:));end; toc
Elapsed time is 582.189554 seconds.
```

The Timing would suggest that we need in total 2.5 sweeps to complete the 21 iterations. However, the computation indicates an average of about 194 seconds or 3.23 minutes per sweep rather than 228 seconds.

```
>> 582.189554/228.157181

ans =

  2.5517
```

Finally, the reader should assure oneself about the correctness of the performed computation while inspecting:

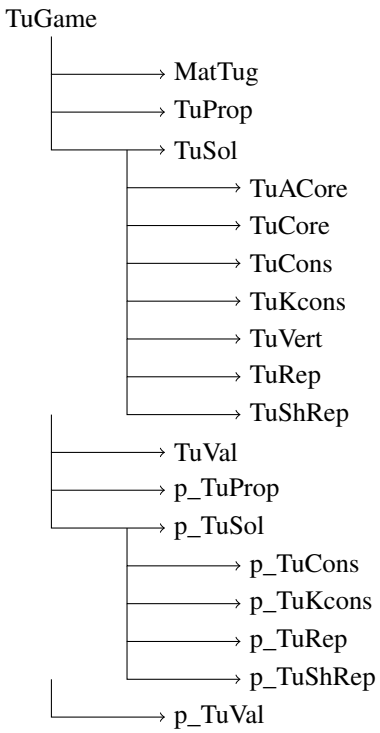
```
>> for k=1:21 eqQ(k,:)=abs(prk_v3_mat_pf(k,:)-prk_v3)<tol; end
>> all(all(eqQ))

ans =

   1
```

4. Class Object TuGame

The Matlab programming language also offers the possibility to the user to create object-oriented programs. Relying on object-oriented programming techniques allow us to encapsulate data and operations within a set of objects designed to ensure that the data are not changed unintentionally, and to perform a relatively complex list of tasks properly. For doing such kind of operation within our Matlab toolbox *MatTuGames*, we designed a class object TuGame with various subclasses. This class object is build up by some properties, methods, and objects to create encapsulated data through instances to itself. It describes a set of objects with some common characteristics that constitutes a TU game. A data array v containing the information of a TU game will be converted to a class object. It will be checked that the data array has the correct size and format. These information among others will be stored in some class instances which will be readout by overloading functions.

The next figure gives an overview of the various subclasses of the object class TuGame.

Figure 2: Class Object TuGame Hierarchy



We now give short synopsis of the available class objects and its functionality.

TuGame — stores some basic game data needed by overloading functions;

MatTug — provides some basic information of the toolbox commands;

TuProp — performs several computations in serial for retrieving and storing game properties;

TuSol — performs several computations in serial for retrieving and storing game solutions;

TuACore — performs several computations in serial for retrieving and storing relevant data to draw the anti core with overloading functions;

TuCore — performs several computations in serial for retrieving and storing relevant data to draw

the core with overloading functions;

TuCons — performs several computations in serial for retrieving and storing consistency properties;

TuKcons — performs several computations in serial for retrieving and storing k-consistency properties;

TuVert — performs several computations in serial for retrieving and storing core and imputation vertices;

TuRep — performs several computations in serial for retrieving and storing the game space that replicates a pre-kernel element of a default game;

TuShRep — performs several computations in serial for retrieving and storing the game space that replicates the Shapley value of a default game;

TuVal — performs several computations in serial for retrieving and storing game values;

p_TuProp — performs several computations in parallel for retrieving and storing game properties;

p_TuSol — performs several computations in parallel for retrieving and storing game solutions;

p_TuCons — performs several computations in parallel for retrieving and storing k-consistency properties;

p_TuKcons — performs several computations in parallel for retrieving and storing k-consistency properties;

p_TuRep — performs several computations in parallel for retrieving and storing the game space that replicates a pre-kernel element of a default game;

p_TuShRep — performs several computations in parallel for retrieving and storing the game space that replicates the Shapley value of a default game;

p_TuVal — performs several computations in parallel for retrieving and storing game values.

In the course of our discussion, we demonstrate by an even-numbered assignment game how one can use the class object TuGame to perform several computations for retrieving and modifying game data. Moreover, we demonstrate how the inheritance of class objects establishes a relationship between objects of different hierarchy levels.

In order to construct an even-numbered assignment game of 6-persons, we must specify in a first step the assignment problem. We determine the whole player set N, that is, sellers and buyers by setting $n = 6$. Since, the game is symmetric, we have at most three sellers, indicated by setting $sl = 3$.

```
>> n=6;
>> sl=3;
>> rand('state',135);
```

To derive an assignment game we need also a profit matrix (cf. with Subsection 2.4). This matrix can be easily generated while executing

```
>> pfm = magic(sl)

pfm =

   8   1   6
   3   5   7
   4   9   2
```

The number of sellers is used to determine the seller set, which is done by

```
>> slv = 1:sl

slv =

   1   2   3
```

Now, we have specified an assignment problem which shall be used to derive an even-numbered assignment game while calling the function *assignment_game()* with these two arguments

```
v=assignment_game(slv,pfm)

>> v

v =

 Columns 1 through 23

   0   0   0   0   0   0   0   0   8   3   8   4   8   4   8   0   1   5   5   9   9   9   9

 Columns 24 through 46

   0   8   5  13   9  17  12  17   0   6   7   7   2   6   7   7   0   8   7  15   4  10  11

 Columns 47 through 63

  15   0   6   7  11   9  15  16  16   0   8   7  15   9  17  16  24
```

The game data array *v* of a TU game is enough to generate the corresponding class object TuGame. The class object will be created with the method *TuGame()*. As a second input argument we also provide a game type by the string *'cr'* to indicate that the game has a non-empty core. Permissible game types are

- *'cv'* for convex/average-convex, and semi-convex game;
- *'cr'* game with non-empty core;
- *'sv'* simple game;
- *'acr'* game with non-empty anti core;
- *''* and the empty string, which is the default value.

As a third input argument the game format can be provided. Permissible arguments are

- *'mattug'* to indicate the unique integer representation of coalitions to perform computation under *MatTuGames*, which is its default value;
- *'mama'* to indicate the generic power set representation of coalitions to perform computation with MATHEMATICA®.

It should be noticed here that the creation of any class object requires at least one input argument, namely the data array of the TU game, the other two arguments are optional.

```
>> clv = TuGame(v,'cr')

clv =

 TuGame with properties:

   tuvalues: [1x63 double]
    tusize: 63
  tuplayers: 6
    tutype: 'cr'
```

```
    tuessQ: 1
   tuformat: 'mattug'
     tumv: 24
    tumnQ: 0
     tuSi: [62 61 59 55 47 31]
     tuvi: [0 0 0 0 0 0]
   tustpt: []
```

The class object *clv* created by the method *TuGame()* stores the following game properties

- *tuvalues* stores the information about the characteristic values of a TU game;
- *tusize* stores the length of the game array/vector;
- *tuplayers* stores the number of players involved;
- *tutype* stores the game type information ('cv','cr','sv','acr',' ');
- *tuessQ* stores the information whether the game is essential;
- *tuformat* stores the format how the game is represented;
- *tumv* stores the largest coalitional value;
- *tumnQ* stores the information whether a proper coalition has a higher value than the grand coalition;
- *tuSi* stores the coalitions having size of n-1;
- *tuvi* stores the values of singleton coalitions;
- *tustpt* stores a starting point for doing computation. Has lower priority than providing a second input argument as an alternative starting point to an overloading function. Thus, if a starting point is provided as second input argument to an overloading function the information stored in *tustpt* will not be used.

The class object properties are retrived by executing *clv.property_name*. For instance, to retrieve the property *tuvalues* one simply types

```
>> clv.tuvalues

ans =

 Columns 1 through 23

   0   0   0   0   0   0   0   0   8   3   8   4   8   4   8   0   1   5   5   9   9   9   9

 Columns 24 through 46

   0   8   5  13   9  17  12  17   0   6   7   7   2   6   7   7   0   8   7  15   4  10  11

 Columns 47 through 63

  15   0   6   7  11   9  15  16  16   0   8   7  15   9  17  16  24
```

It also possible to retrieve data of a class object with the *get* command of Matlab. To demonstrate its use we consider the following example

```
>> get(clv,'tusize')

ans =

    63
```

All functions that have as an input argument a data array of coalitional values are overloaded. However, every function that creates a TU game from a specific situation can not be overloaded. To see this consider the following functions that allow an overloading with the class object *clv*.

```
>> prn_v=PreNucl(clv)

prn_v =

   5.0000   3.0000   4.0000   3.0000   5.0000   4.0000

>> prk_v=PreKernel(clv)

prk_v =

   5.0000   3.0000   4.0000   3.0000   5.0000   4.0000

>> kr_v=Kernel(clv)
kr_v =

   5.0000   3.0000   4.0000   3.0000   5.0000   4.0000

>> sh_v=ShapleyValue(clv)

sh_v =

   4.1167   3.5167   4.3667   3.8333   4.5833   3.5833

>> nc_v=nucl(clv)

nc_v =

   5.0000   3.0000   4.0000   3.0000   5.0000   4.0000
```

However, the command *bankruptcy_game()* does not allow as an input argument the class object *clv*. This is due by the fact, that from a bankruptcy situation a bankruptcy game is created with this command.

```
>> bv=bankruptcy_game(clv)
Error using bankruptcy_game (line 27)
Not enough input arguments.
```

The advantages of using the class object TuGame as an input argument are versatile. For example, we need not to verify the game format when calling a method. This was already done while executing the method *TuGame()*. In case that the format is not correct, the function call interrupts with an error message, hence

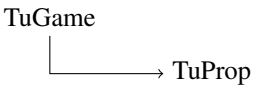
```
>> clv1=TuGame(kr_v)
Error using TuGame (line 115)
Game has not the correct size! Must be a 2^n-1 vector.
```

The class object properties can speed up the computational process, since several loop constructs need not to be applied within a method. Clearly, the disadvantage is that more data must be transferred.

4.1. Class Object TuProp

The class object TuProp stores relevant information related to some game properties. This class object is a subclass of the class object TuGame and inherits therefore all properties from it. The creation of this subclass object allows the user to perform standardized methods for verifying important game properties.

Figure 3: Class Object TuProp Hierarchy



If a class object clv_pr will be created by the method *TuProp()*, then various routines to verify game properties will be called immediately. This implies that the creation of the class object *clv_pr* for larger games can last a while. Especially, the verification of the average convexity or the non-emptiness of the core needs some time to complete.

A class object TuProp will be created in the same vein as the super-class object TuGame. At least the game data array *v* is required and at most the same two additional arguments can be provided as under the class object TuGame. Next, we creat the class object *clv_pr* while invoking the method *TuProp()*

```
>> clv_pr = TuProp(v,'cr')

clv_pr =

 TuProp with properties:

    cv_valid: 0
   acv_valid: 0
   scv_valid: 1
   kcv_valid: 0
   sad_valid: 1
  wsad_valid: 1
   mon_valid: 1
  zmon_valid: 1
    cr_valid: 1
   acr_valid: 0
    tuvalues: [1x63 double]
     tusize: 63
  tuplayers: 6
     tutype: 'cr'
     tuessQ: 1
   tuformat: 'mattug'
      tumv: 24
     tumnQ: 0
      tuSi: [62 61 59 55 47 31]
      tuvi: [0 0 0 0 0 0]
    tustpt: []
```

Apart from the properties inherited from the class object TuGame, the subclass object TuProp has the following additional properties for which we give a short explanation.

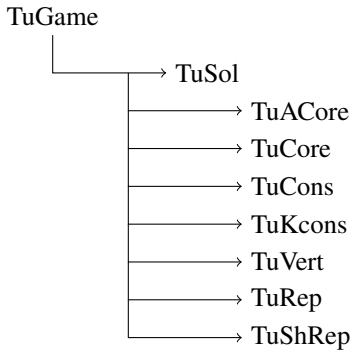
- *cv_valid* stores whether the game is convex (1) or not (0);
- *acv_valid* stores whether the game is average-convex (1) or not (0);
- *scv_valid* stores whether the game is semi-convex;
- *kcv_valid* stores whether the game is *k*-convex;
- *sad_valid* stores whether the game is super-additive;
- *wsad_valid* stores whether the game is weakly super-additive;
- *mon_valid* stores whether the game is monotone;
- *zmon_valid* stores whether the game is zero-monotone;
- *cr_valid* stores whether the game has a non-empty core (1) or not (0).

- *acr_valid* stores whether the game has a non-empty anti core (1) or not (0).

4.2. Class Objects TuSol and p_TuSol

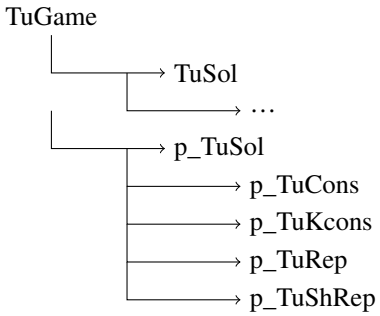
The class object TuSol is a subclass of the class object TuGame which implies that it inherits all properties and methods from its super-class. However, as we can see from the overview of the class object hierarchy given by the next Figure, the class object TuSol is also a super-class object for various other subclass objects. This means, the class object TuSol bequeaths besides its own properties also the properties which it has inherited by the class object TuGame to all subclass objects for which it constitutes a super-class object. The class object TuSol has the task to gather all solutions which are triggered by an event.

Figure 4: Class Object TuSol Hierarchy



There exists also a class object p_TuSol with the same attributes as the class object TuSol but with one crucial difference, the former is attributed for doing computation in parallel whereas the latter for performing the same operations in serial. Moreover, the class objects p_TuSol and TuSol are subclass objects of TuGame but they differ in their subclasses and their invoked methods. Both class objects resides on the same hierarchy level. As a consequence, no data can be exchanged among them.

Figure 5: Class Object p_TuSol Hierarchy



We create a class object TuSol in the similar way as we did it already for the super-class object TuGame. Thus, we execute the method *TuSol()* with at least one or at most three arguments to create a class object *clv_sol*.

```
>> clv_sol = TuSol(v,'cr')
```

```
clv_sol =

 TuSol with properties:

    tu_prk: []
    tu_prn: []
   tu_prk2: []
    tu_sh: []
   tu_tauv: []
    tu_bzf: []
   tu_aprk: []
  prk_valid: 0
  prn_valid: 0
 prk2_valid: 0
 aprk_valid: 0
   tuvalues: [1x63 double]
    tusize: 63
  tuplayers: 6
    tutype: 'cr'
    tuessQ: 1
  tuformat: 'mattug'
     tumv: 24
    tumnQ: 0
     tuSi: [62 61 59 55 47 31]
     tuvi: [0 0 0 0 0 0]
   tustpt: []
```

At this stage nothing has been computed. Just the properties which are inherited from its super-class object will be distributed to it. Let us first notice which additional properties can be assigned with data. These are:

- *tu_prk* stores a pre-kernel element;
- *tu_prk2* stores a second pre-kernel element instead of the pre-nucleolus, if it exits;
- *tu_prn* stores the pre-nucleolus for games with $n \leq 15$;
- *tu_sh stores* the Shapley value;
- *tu_tauv* stores the Tau value;
- *tu_bzf* stores the Banzhaf value;
- *tu_aprk* stores the anti pre-kernel;
- *prk_valid* stores 1 if *tu_prk* stores a pre-kernel element, otherwise 0;
- *prk2_valid* stores 1 if *tu_prk2* stores a pre-kernel element, otherwise 0.
- *prn_valid* stores 1 if *tu_prn* stores the pre-nucleolus, otherwise 0.

As we will learn in the sequel, this can be done either by distributing an already computed value or by triggering a computation. To learn more about the different procedures of value assignment let us compute in a first step the Shapely value of the game.

```
>> sh_v=ShapleyValue(v)

sh_v =

  4.1167   3.5167   4.3667   3.8333   4.5833   3.5833
```

The Shapley value can now be assigned to the class object TuSol through

```
>> clv_sol.tu_sh=sh_v
```

```
clv_sol =

 TuSol with properties:

     tu_prk: []
     tu_prn: []
    tu_prk2: []
     tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
    tu_tauv: []
     tu_bzf: []
    tu_aprk: []
   prk_valid: 0
   prn_valid: 0
  prk2_valid: 0
  aprk_valid: 0
    tuvalues: [1x63 double]
     tusize: 63
   tuplayers: 6
     tutype: 'cr'
     tuessQ: 1
    tuformat: 'mattug'
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

or one simply assigns the empty set to the property. This triggers an event to execute internally the computation of the Shapley value. This can be accomplished with

```
>> clv_sol.tu_sh=[]
```

or to use the built-in Matlab command *set()* to trigger the computation of the Shapley value. For instance, we call the command *set()* with the empty string.

```
>> set(clv_sol,'tu_sh','')
```

If one wants to avoid a reevaluation of a solution, since the solution was already computed and the purpose is just to copy a solution to a class object, then we can again rely on the *set* command to assign, for instance, the Shapley value stored in the variable *sh_v* to the class object *clv_sol*. Therefore, we call

```
>> set(clv_sol,'tu_sh',sh_v)
```

It is also possible to set the Shapley value to the property *clv_sol.tu_sh* while executing the overloading function *ShapleyValue()*.

```
>> clv_sol.tu_sh=ShapleyValue(slv_sol)
```

As we have already mentioned it in the introductory part, it is also possible to create encapsulated data through an instance to itself, that is, by the method *setShapley()*. This operation updates the class object *clv_sol* while writing the result of the Shapley value computation to the property element *clv_sol.tu_sh*. Therefore, we have to supply the method *setShapley()* with the input argument of the class object we want to modify and as its output argument the same class object, since the result of the computation should be written on this class object to update it, hence *clv_sol*. Thus, we have to execute

```
>> clv_sol=setShapley(clv_sol)

clv_sol =

 TuSol with properties:

    tu_prk: []
    tu_prn: []
   tu_prk2: []
    tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
   tu_tauv: []
    tu_bzf: []
   tu_aprk: []
  prk_valid: 1
  prn_valid: 0
 prk2_valid: 0
 aprk_valid: 0
  tuvalues: [1x63 double]
   tusize: 63
  tuplayers: 6
   tutype: 'cr'
   tuessQ: 1
  tuformat: 'mattug'
     tumv: 24
    tumnQ: 0
     tuSi: [62 61 59 55 47 31]
     tuvi: [0 0 0 0 0 0]
    tustpt: []
```

Similar, when calling the method *setPreKernel()* we can assign a pre-kernel element to the class object *clv_sol*.

```
>> clv_sol=setPreKernel(clv_sol)

clv_sol =

 TuSol with properties:

    tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_prn: []
   tu_prk2: []
    tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
   tu_tauv: []
    tu_bzf: []
   tu_aprk: []
  prk_valid: 1
  prn_valid: 0
 prk2_valid: 0
 aprk_valid: 0
  tuvalues: [1x63 double]
   tusize: 63
  tuplayers: 6
   tutype: 'cr'
   tuessQ: 1
  tuformat: 'mattug'
     tumv: 24
    tumnQ: 0
     tuSi: [62 61 59 55 47 31]
     tuvi: [0 0 0 0 0 0]
    tustpt: []
```

The pre-nucleolus can be assigned with the method *setPreNuc()*.

```
>> clv_sol=setPreNuc(clv_sol)

clv_sol =

 TuSol with properties:

     tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
     tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_prk2: []
     tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
    tu_tauv: []
     tu_bzf: []
    tu_aprk: []
  prk_valid: 1
  prn_valid: 1
 prk2_valid: 0
 aprk_valid: 0
   tuvalues: [1x63 double]
    tusize: 63
  tuplayers: 6
    tutype: 'cr'
    tuessQ: 1
   tuformat: 'mattug'
      tumv: 24
     tumnQ: 0
     tuSi: [62 61 59 55 47 31]
     tuvi: [0 0 0 0 0 0]
    tustpt: []
```

Other permissible methods of the class object TuSol for setting solutions are: *setTauValue()*, *setBanzhaf()*, and *setAntiPreKernel()*.

Nevertheless, it can be quite annoying to assign value by value to a class object. To automate the whole process, it exists the method *setAllSolutions()*. To observe how it works, let us create first a fresh class object TuSol. This is done by

```
>> clv_sol1=TuSol(v,'cr')

clv_sol1 =

 TuSol with properties:

     tu_prk: []
     tu_prn: []
    tu_prk2: []
     tu_sh: []
    tu_tauv: []
     tu_bzf: []
    tu_aprk: []
  prk_valid: 0
  prn_valid: 0
 prk2_valid: 0
 aprk_valid: 0
   tuvalues: [1x63 double]
    tusize: 63
  tuplayers: 6
    tutype: 'cr'
    tuessQ: 1
   tuformat: 'mattug'
      tumv: 24
     tumnQ: 0
     tuSi: [62 61 59 55 47 31]
     tuvi: [0 0 0 0 0 0]
    tustpt: []
```

Let us again remember how we have to use a method. As an input argument it is always required to supply the class object that should be updated, in this example the class object *clv_sol1*. As an output argument the same class object must be used (*clv_sol1*), since all results of the computational process must be written on this class object to make the updating effective. In this respect notice that the properties of the class objects are not write protected after all properties have been assigned with values. Selecting a wrong class object as an output argument has the negative side-effect that all previous computed values on the single out class object will be overwritten. In the worst case, correct results will be updated with results from a different game. This means for our example that we have to apply the class object *clv_sol1* as an output argument. After these clarifying comments let us now compute all solutions with the method *setAllSolutions()*. We call it through

```
>> tic;clv_sol1=setAllSolutions(clv_sol1);toc
Elapsed time is 2.103740 seconds.
```

then all relevant solutions of the game will be computed and assigned to the class object *clv_sol1* as we can see from below

```
>> clv_sol1

clv_sol1 =

 TuSol with properties:

     tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
     tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_prk2: []
     tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
    tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
     tu_bzf: []
    tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
   prk_valid: 1
   prn_valid: 1
  prk2_valid: 0
  aprk_valid: 1
    tuvalues: [1x63 double]
     tusize: 63
   tuplayers: 6
     tutype: 'cr'
     tuessQ: 1
    tuformat: 'mattug'
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

What is remarkable here, the Banzhaf value will not be computed, since the game type is indicated as *'cr'* and not as *'sv'*, that is, not as a simple game. Moreover, by default only one pre-kernel element will be computed. In order to find a second one, it is just enough to execute

```
>> slv_sol1.tu_prk2=[]

slv_sol1 =

 TuSol with properties:

     tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
```

```
    tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
   tu_prk2: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
   tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
    tu_bzf: []
   tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
  prk_valid: 1
  prn_valid: 1
 prk2_valid: 1
 aprk_valid: 1
  tuvalues: [1x63 double]
   tusize: 63
 tuplayers: 6
   tutype: 'cr'
   tuessQ: 1
  tuformat: 'mattug'
     tumv: 24
    tumnQ: 0
     tuSi: [62 61 59 55 47 31]
     tuvi: [0 0 0 0 0 0]
   tustpt: []
```

As we can see, the standard procedure was not successful to find a second pre-kernel element. Just the element already found was assigned to *tu_prk2*.

The operation performed inside a method related to the class object TuSol also triggers some validation procedure for the pre-kernel and pre-nucleolus solution. Thus, if we want to set the Shapley value to the property *tu_prk2* that should store an element of the pre-kernel, then the value will be assigned to the property while overwriting the old value but it will be indicated as a non valid pre-kernel element as we realize through *prk2_valid: 0*.

```
>> clv_sol1.tu_prk2=clv_sol1.tu_sh

clv_sol1 =

 TuSol with properties:

    tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
   tu_prk2: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
    tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
   tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
    tu_bzf: []
   tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
  prk_valid: 1
  prn_valid: 1
 prk2_valid: 0
 aprk_valid: 1
  tuvalues: [1x63 double]
   tusize: 63
 tuplayers: 6
   tutype: 'cr'
   tuessQ: 1
  tuformat: 'mattug'
     tumv: 24
    tumnQ: 0
     tuSi: [62 61 59 55 47 31]
     tuvi: [0 0 0 0 0 0]
   tustpt: []
```

Every computation related to the pre-nucleolus and pre-kernel triggered by an event will initiate a validation process. After the completion of the evaluation the user can immediately recognize if the pre-

nucleolus or a pre-kernel element is indicated as invalid. If so, this could mean that a wrong solution was found, but it can also mean that the solution was denoted as wrong within the pre-specified level of numerical tolerance. Increasing the tolerance level helps in this case to classify the solution as valid. However, the user bears full responsibility to verify that an indicated solution is correct or incorrect.

To summarize our discussion we first enlist all methods provided by the class object TuSol

- *TuSol* creates the class object TuSol;
- *setAllSolutions* sets all solutions listed below to the class object TuSol;
- *setPreKernel* sets a pre-kernel element to the class object TuSol;
- *setPreNuc* sets the pre-nucleolus to the class object TuSol;
- *setShapley* sets the Shapley value to the class object TuSol;
- *setTauValue* sets the Tau value to the class object TuSol;
- *setBanzhaf* sets the Banzhaf value to the class object TuSol;
- *setAntiPreKernel* sets an anti pre-kernel element to the class object TuSol;

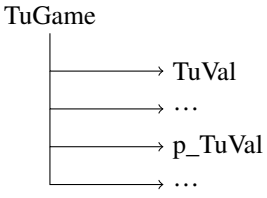
and finally all methods provided by the class object p_TuSol to start a parallel evaluation.

- *p_TuSol* creates the class object p_TuSol.
- *p_setAllSolutions* sets all solutions listed below to the class object p_TuSol.
- *p_setPreKernel* sets a pre-kernel element to the class object p_TuSol.
- *p_setPreNuc* sets the pre-nucleolus to the class object p_TuSol.
- *p_setShapley* sets the Shapley value to the class object p_TuSol.
- *p_setTauValue* sets the Tau value to the class object p_TuSol.
- *p_setBanzhaf* sets the Banzhaf value to the class object p_TuSol.
- *p_setAntiPreKernel* sets an anti pre-kernel element to the class object p_TuSol.

4.3. Class Objects TuVal and p_TuVal

The class object TuVal is a subclass of the class object TuGame which implies that it inherits all properties and methods from its super-class. The class object TuVal has the task to store some important types of coalition structures and to gather all game values w.r.t. these coalition structures. There exists also a class object p_TuVal with the same attributes as the class object TuVal but with one crucial difference, the former is attributed for doing computation in parallel whereas the latter for performing the same operations in serial. Moreover, the class objects p_TuVal and TuVal are subclass objects of TuGame but they differ in their invoked methods. Both class objects resides on the same hierarchy level, no data can be exchanged among them.

Figure 6: Class Object TuVal and p_TuVal Hierarchy



To illustrate the usage of the methods *TuVal()* and *p_TuVal()* to create the associated class objects, we

take an example from the literature rather than relying on the above introduced assignment game. This example was extensively discussed in Algaba et al. (2004) to illuminate some properties of the position value w.r.t. hypergraph systems. This should allow the reader to retrace our arguments and to verify the computation of the position value w.r.t. these hypergraph communication situations. Moreover, the usage of these class objects allow an immediate comparison with other game values.

In addition, we want demonstrate that there is no difference in its usage between both class objects, except that the names of the methods differ. To execute the methods from the latter the prefix p_ must be added to the method names introduced under the former. For that reason we start a parallel session with

```
Starting matlabpool using the 'MyProfile2' profile ... connected to 12 workers.
```

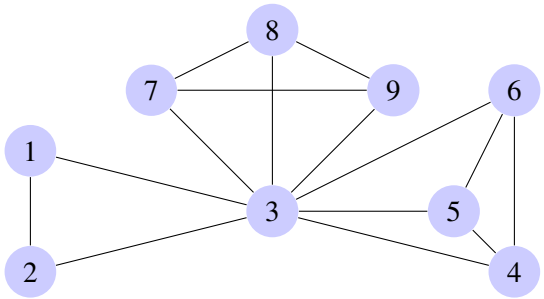
The cooperative game discussed by example 3.2 in Algaba et al. (2004) can be interpreted as a production economy of manufacturers of the same type where each manufacturer holds some units of raw material. With these raw materials a quantity of an identical product can be produced. The production function is quadratic and to keep the structure of the game simple the corresponding characteristic function agrees with the production function of the manufacturers.

This kind of cooperative production game can be derived with the function *production_game_sq()*. This command requires two input arguments, the first is the number of players/manufacturers involved and the second argument indicates how many manufacturers must merge their economic activities (units of raw material) to produce a positive quantity. In this particular example we study a 9-person cooperative production game with a required minimum firm size of 2-manufacturers to produce a positive quantity.

```
>> v_p=production_game_sq(9,2);
```

In a further step we have to specify the coalition structures from which several game values should be derived. These coalition structures must be assigned to the class objects *TuVal()* and *p_TuVal()* to be able to trigger the event to compute game values w.r.t. a coalition structure. In order to perform a successful computation with these class objects, it is required to supply at least a communication situation à la Myerson. From this communication structure all other missing coalition structures are derived. However, we recommend to supply all coalition structures in advance to get a direct understanding about the evaluated results. This is the followed procedure in the sequel of our discussion.

For doing so, we specify the different coalition structures in the order of its appearance within the class objects. Therefore, we start with the communication situation à la Myerson given by the graph below.



This communication situation is coded under Matlab by

```
>> L={[1 2],[1 3],[2 3],[3 4],[3 5],[3 6],[4 5],[4 6],[5 6],[3 7],[3 8],[3 9],[7 8],[7 9],[8 9]}

L =
```

```
 Columns 1 through 8

  [1x2 double]  [1x2 double]  [1x2 double]  [1x2 double]  [1x2 double]  [1x2 double]  [1x2 double]  [1x2 double]

 Columns 9 through 15

  [1x2 double]  [1x2 double]  [1x2 double]  [1x2 double]  [1x2 double]  [1x2 double]  [1x2 double]
```

Recall that these coalitions must be transcribed into its unique integer representation to be usable under Matlab. Hence, we convert them by the following well-known command

```
>> lm=clToMatlab(L)

lm =

   3   5   6  12  20  24  36  40  48  68 132 192 260 320 384
```

Some game values require a partition of the player set to specify a set of à priori unions. This is done while coding the partition of the 9-person set by

```
>> P={[1 2 5 6],[3 4 7],[8 9]}

P =

  [1x4 double]  [1x3 double]  [1x2 double]
```

which gives in its unique integer representation

```
>> pm=clToMatlab(P)

pm =

  51  76  384
```

For the remaining coalition structures, we rely on the representation given by Algaba et al. (2004, pp. 470-472). These authors consider two hypergraph communication situations. The first is given by

```
>> H1={[1 2 3 4],[3 4 7],[3 4 5 6],[1 2 3 4 5 6],[8 9]}

H1 =

  [1x4 double]  [1x3 double]  [1x4 double]  [1x6 double]  [1x2 double]
```

with its Matlab representation obtained by

```
>> hm1=clToMatlab(H1)

hm1 =

  15  60  63  76  384
```

Verifying that this system is indeed a hypergraph communication situation we invoke

```
>> hsQ1=hypergraphQ(hm1,9)

hsQ1 =

   1
```

The second hypergraph communication situation considered in the example is specified by

```
>> H2={[1 2 3 4],[3 4 7],[3 4 5 6],[8 9]}

H2 =

  [1x4 double]  [1x3 double]  [1x4 double]  [1x2 double]

>> hm2=clToMatlab(H2)

hm2 =

  15  60  76  384
```

Again, we can check that the specified system is a hypergraph while executing

```
>> hsQ2=hypergraphQ(hm2,9)

hsQ2 =

   1
```

The above authors introduce also a union stable coalition structure specified by

```
>> F1={[1],[2],[3],[4],[5],[6],[7],[8],[9],[1 2 3 4],[3 4 7],[3 4 5 6],[1 2 3 4 5 6],[8 9],[1 2 3 4 7],...
  [3 4 5 6 7],[1 2 3 4 5 6 7]}

F1 =

 Columns 1 through 14

  [1]  [2]  [3]  [4]  [5]  [6]  [7]  [8]  [9]  [1x4 double]  [1x3 double]  [1x4 double]  [1x6 double]  [1x2 double]

 Columns 15 through 17

  [1x5 double]  [1x5 double]  [1x7 double]
```

From this coding we receive its Matlab representation through

```
>> fm1=clToMatlab(F1)

fm1 =

   1   2   4   8  15  16  32  60  63  64  76  79  124  127  128  256  384
```

to be able to check that the introduced system is union stable. And indeed, this is confirmed with the parallel command *p_union_stableQ()*

```
>> usQ=p_union_stableQ(fm1)

usQ =

   1
```

To complete the preparatory part, we verify still two game properties. First, we want to study if the core is non-empty, which is the case as we can see next

```
>> CddCoreQ(v_p)

ans =

   1
```

Second, we study the convexity of the cooperative production game. Which can also be affirmed by retrieving the output from the following evaluation

```
>> convex_gameQ(v_p)

ans =

   1
```

Let us now create class objects TuVal and p_TuVal to realize the different handling. These class objects will be created in the same way as we did it already for the super-class object TuGame or the class object TuSol. Thus, we execute the method *TuVal()* with at least one or at most three arguments to create a class object *fclv*. Since, the game is convex, we supply the method which creates this class object with the optional argument that indicates its game type. Hence,

```
>> fclv=TuVal(v_p,'cv')

fclv =

 TuVal with properties:

    tu_sh: []
    tu_ad: []
    tu_ow: []
    tu_my: []
   tu_myus: []
   tu_psus: []
   tu_pshs: []
    tu_sl: []
   tu_csl: []
   tu_slsh: []
   tu_asl: []
    tu_cs: []
   tu_ptn: []
    tu_us: []
    tu_hs: []
  us_valid: 0
  hs_valid: 0
  tuvalues: [1x511 double]
   tusize: 511
  tuplayers: 9
   tutype: 'cv'
   tuessQ: 1
  tuformat: 'mattug'
     tumv: 81
    tumnQ: 0
    tuSi: [510 509 507 503 495 479 447 383 255]
    tuvi: [0 0 0 0 0 0 0 0 0]
   tustpt: []
```

Analogously, we create the class object p_TuVal called *p_fclv* by

```
>> p_fclv=p_TuVal(v_p,'cv')

p_fclv =

 p_TuVal with properties:

    tu_sh: []
    tu_ad: []
    tu_ow: []
    tu_my: []
   tu_myus: []
   tu_psus: []
   tu_pshs: []
    tu_sl: []
   tu_csl: []
   tu_slsh: []
   tu_asl: []
    tu_cs: []
   tu_ptn: []
    tu_us: []
    tu_hs: []
  us_valid: 0
  hs_valid: 0
  tuvalues: [1x511 double]
   tusize: 511
  tuplayers: 9
   tutype: 'cv'
   tuessQ: 1
  tuformat: 'mattug'
     tumv: 81
    tumnQ: 0
     tuSi: [510 509 507 503 495 479 447 383 255]
     tuvi: [0 0 0 0 0 0 0 0 0]
   tustpt: []
```

We observe between both class objects no differences in its properties. However, the type of class object is given at the top of the listing. Different class objects invoke different methods even when its properties are identical. Furthermore, at this stage nothing has been computed. Similar to the class object TuSol the properties which are inherited from its super-class object TuGame will be distributed to it. Let us first notice which additional properties can be assigned with some data. The class objects *fclv* and *p_fclv* created by the method *TuVal()* and *p_TuVal()* respectively, store the following fairness and related values:

- *tu_sh* stores the Shapley value;
- *tu_ad* stores the Aumann-Drèze value;
- *tu_ow* stores the Owen value;
- *tu_my* stores the Myerson value w.r.t. a communication sitatuation à la Myerson;
- *tu_myus* stores the Myerson value w.r.t. a union stable system;
- *tu_psus* stores the position value w.r.t. a union stable system;
- *tu_pshs* stores the position value w.r.t. a hypergraph system;
- *tu_sl* stores the solidarity value;
- *tu_csl* stores the coalition solidarity value;
- *tu_slsh* stores the solidarity Shapley value;
- *tu_asl* stores the solidarity value w.r.t. à priori unions;
- *tu_cs* stores a communication structure à la Myerson (mandatory);
- *tu_ptn* stores a partition of the grand coalition;

- *tu_us* stores a union stable system;
- *tu_hs* stores a hypergraph system.;
- *us_valid* stores a true (1), if *tu_us* is a union stable system, otherwise false (0);
- *hs_valid* stores a true (1), if *tu_hs* is a hypergraph communication system, otherwise false (0).

In contrast to the class objects discussed in the previous subsections, the present class objects need the assignment of coalition structures to trigger a computation. In the sequel, we discuss different methods of how to assign the data to the class objects TuVal and p_TuVal. The simplest and quickest method is to use the method *setCoalitionStructures()* for a serial computation under TuVal or to invoke the method *p_setCoalitionStructures()* for a parallel computation under p_TuVal. Both methods require at least two input arguments, the class object that should be updated and a communication situation à la Myerson as its second input argument. From this communication situation all other missing coalition structures can be derived. Further optional input arguments are à priori unions, a union stable, and a hypergraph system. No empty set can be supplied when the coalition structures should be set with one of these methods.

Exemplary, we set the above introduced coalition structures with the method *setCoalitionStructures()* to the class object *fclv* for executing later on a serial computation.

```
>> fclv=setCoalitionStructures(fclv,lm,pm,fm1,hm1)

fclv =

 TuVal with properties:

     tu_sh: []
     tu_ad: []
     tu_ow: []
     tu_my: []
   tu_myus: []
   tu_psus: []
   tu_pshs: []
    tu_sl: []
    tu_csl: []
   tu_slsh: []
    tu_asl: []
     tu_cs: [3 5 6 12 20 24 36 40 48 68 132 192 260 320 384]
    tu_ptn: [51 76 384]
     tu_us: [1 2 4 8 15 16 32 60 63 64 76 79 124 127 128 256 384]
     tu_hs: [15 60 63 76 384]
  us_valid: 1
  hs_valid: 1
  tuvalues: [1x511 double]
    tusize: 511
  tuplayers: 9
    tutype: 'cv'
    tuessQ: 1
  tuformat: 'mattug'
      tumv: 81
     tumnQ: 0
     tuSi: [510 509 507 503 495 479 447 383 255]
     tuvi: [0 0 0 0 0 0 0 0 0]
   tustpt: []
```

As a shortcut, the coalition structures can be set step by step through

```
>> set(p_fclv,'tu_cs',lm)
>> set(p_fclv,'tu_ptn',pm)
>> set(p_fclv,'tu_us',fm1)
>> set(p_fclv,'tu_hs',hm1)
```

or alternatively by

```
>> p_fclv.tu_cs=lm;
>> p_fclv.tu_ptn=pm;
>> p_fclv.tu_us=fm1;
>> p_fclv.tu_hs=hm1;
```

In both cases the coalition structure are set to the class object *p_fclv* to be able to invoke later on a parallel evaluation of the game values.

```
>> p_fclv

p_fclv =

 p_TuVal with properties:

     tu_sh: []
     tu_ad: []
     tu_ow: []
     tu_my: []
    tu_myus: []
    tu_psus: []
    tu_pshs: []
     tu_sl: []
    tu_csl: []
   tu_slsh: []
    tu_asl: []
     tu_cs: [3 5 6 12 20 24 36 40 48 68 132 192 260 320 384]
    tu_ptn: [51 76 384]
     tu_us: [1 2 4 8 15 16 32 60 63 64 76 79 124 127 128 256 384]
     tu_hs: [15 60 76 384]
   us_valid: 1
   hs_valid: 1
   tuvalues: [1x511 double]
    tusize: 511
  tuplayers: 9
    tutype: 'cv'
    tuessQ: 1
   tuformat: 'mattug'
      tumv: 81
     tumnQ: 0
      tuSi: [510 509 507 503 495 479 447 383 255]
      tuvi: [0 0 0 0 0 0 0 0 0]
    tustpt: []
```

After the setting of the coalition structures to the class objects, we are now ready to start some calculations. Different methods are available to activate the computation of the game values. First, we study how to compute the different game values step by step on the class object *fclv*. In the same vein as we set the coalition structures, we can invoke the computation of the game values as a shortcut through

```
>> set(fclv,'tu_sh','')
>> set(fclv,'tu_ad','')
>> set(fclv,'tu_ow','')
>> set(fclv,'tu_my','')
>> set(fclv,'tu_myus','')
>> set(fclv,'tu_psus','')
>> set(fclv,'tu_pshs','')
>> set(fclv,'tu_sl','')
>> set(fclv,'tu_csl','')
>> set(fclv,'tu_slsh','')
>> set(fclv,'tu_asl','')
```

for assigning the results to the class object *fclv*. No method names must be remembered, it is just enough to recall the class object name and its properties in connection with the empty string to start the evaluation of a game value. Or alternatively as

```
>> fclv.tu_sh=[];
>> fclv.tu_ad=[];
>> fclv.tu_ow=[];
>> fclv.tu_my=[];
>> fclv.tu_myus=[];
>> fclv.tu_psus=[];
>> fclv.tu_pshs=[];
>> fclv.tu_sl=[];
>> fclv.tu_csl=[];
>> fclv.tu_slsh=[];
>> fclv.tu_asl=[];
```

In both cases, we obtain

```
>> fclv

fclv =

 TuVal with properties:

     tu_sh: [9 9 9 9 9 9 9 9 9]
     tu_ad: [4 4 3 3 4 4 3 2 2]
     tu_ow: [9 9 9 9 9 9 9 9 9]
     tu_my: [7.8282 7.8282 14.6746 8.4448 8.4448 8.4448 8.4448 8.4448 8.4448]
    tu_myus: [5.1984 5.1984 11.3032 11.3032 5.1984 5.1984 3.4698 1.0651 1.0651]
    tu_psus: [4.9375 4.9375 13.8472 13.8472 4.9375 4.9375 3.9722 -1.2083 -1.2083]
    tu_pshs: [5.5278 5.5278 11.5556 11.5556 5.5278 5.5278 3.7778 2 2]
     tu_sl: [9.0000 9.0000 9.0000 9.0000 9.0000 9.0000 9.0000 9.0000 9.0000]
    tu_csl: [9 9 9 9 9 9 9 9 9]
   tu_slsh: [7.6319 7.6319 8.9815 8.9815 7.6319 7.6319 8.9815 11.7639 11.7639]
    tu_asl: [7.6319 7.6319 8.9815 8.9815 7.6319 7.6319 8.9815 11.7639 11.7639]
     tu_cs: [3 5 6 12 20 24 36 40 48 68 132 192 260 320 384]
    tu_ptn: [51 76 384]
     tu_us: [1 2 4 8 15 16 32 60 63 64 76 79 124 127 128 256 384]
     tu_hs: [15 60 63 76 384]
   us_valid: 1
   hs_valid: 1
   tuvalues: [1x511 double]
    tusize: 511
  tuplayers: 9
    tutype: 'cv'
    tuessQ: 1
   tuformat: 'mattug'
      tumv: 81
     tumnQ: 0
     tuSi: [510 509 507 503 495 479 447 383 255]
     tuvi: [0 0 0 0 0 0 0 0 0]
    tustpt: []
```

Singling out the position value w.r.t. the first hypergraph system to compare it with the result obtained by Algaba et al. (2004, p. 471), we observe that our computation confirms this result. Hence,

```
>> rats(fclv.tu_pshs)

ans =

  199/36     199/36     104/9      104/9      199/36     199/36     34/9       2          2
```

Recall the discussion of Subsection 2.6 where we have mentioned that the position value is a dual approach while focusing on the role of communication links represented by arcs. The communicative strength of a communication link is measured by the outcome of the Shapley value of an arc-game. This game type is defined on the arcs of the communication graph. Each player has a veto power of the use of communication links (arcs) at which he is an endpoint. Therefore, it is reasonable to assume that players on a communication link have equal power and the worth of an arc should be divided equally among them (cf. also with Borm et al. (1992)).

Now, let us consider a third method to evaluate the game values. For this example, we turn our attention back to the class object *p_fclv*. For this purpose, we reset the coalition structures to *p_fclv* with

```
>> p_fclv=p_setCoalitionStructures(p_fclv,lm,pm,fm1,hm1)

p_fclv =

 p_TuVal with properties:

     tu_sh: []
     tu_ad: []
     tu_ow: []
     tu_my: []
    tu_myus: []
    tu_psus: []
    tu_pshs: []
     tu_sl: []
    tu_csl: []
   tu_slsh: []
    tu_asl: []
     tu_cs: [3 5 6 12 20 24 36 40 48 68 132 192 260 320 384]
    tu_ptn: [51 76 384]
     tu_us: [1 2 4 8 15 16 32 60 63 64 76 79 124 127 128 256 384]
     tu_hs: [15 60 63 76 384]
   us_valid: 1
   hs_valid: 1
   tuvalues: [1x511 double]
    tusize: 511
  tuplayers: 9
    tutype: 'cv'
    tuessQ: 1
   tuformat: 'mattug'
      tumv: 81
     tumnQ: 0
      tuSi: [510 509 507 503 495 479 447 383 255]
      tuvi: [0 0 0 0 0 0 0 0 0]
    tustpt: []
```

Again, we compute the game values step by step while presenting an alternative permissible method to compute a particular value of a game with coalition structure. Thus, this permissible evaluation method in parallel is:

```
>> p_fclv=p_setShapley(p_fclv);
>> p_fclv=p_setADvalue(p_fclv);
>> p_fclv=p_setOwen(p_fclv);
>> p_fclv=p_setMyerson(p_fclv);
>> p_fclv=p_setMyersonUS(p_fclv);
>> p_fclv=p_setPosition(p_fclv);
>> p_fclv=p_setPositionHS(p_fclv);
>> p_fclv=p_setSolidarity(p_fclv);
>> p_fclv=p_setCoalitionSolidarity(p_fclv);
>> p_fclv=p_setSolidarityShapley(p_fclv);
>> p_fclv=p_setApuSolidarity(p_fclv);
```

Of course, one can also use alternatively the *set* command as we can observe by the following example:

```
>> set(p_fclv,'tu_sh','')
>> set(p_fclv,'tu_ad','')
>> set(p_fclv,'tu_ow','')
>> set(p_fclv,'tu_my','')
>> set(p_fclv,'tu_myus','')
>> set(p_fclv,'tu_psus','')
>> set(p_fclv,'tu_pshs','')
>> set(p_fclv,'tu_sl','')
>> set(p_fclv,'tu_csl','')
>> set(p_fclv,'tu_slsh','')
>> set(p_fclv,'tu_asl','')
```

Notice, that the serial counterparts have the same name without the prefix p_. All these methods set the result to its corresponding class property as we can assure oneself by inspecting the following data.

```
>> p_fclv

p_fclv =

 p_TuVal with properties:

     tu_sh: [9 9 9 9 9 9 9 9 9]
     tu_ad: [4 4 3 3 4 4 3 2 2]
     tu_ow: [9 9 9 9 9 9 9 9 9]
     tu_my: [7.8282 7.8282 14.6746 8.4448 8.4448 8.4448 8.4448 8.4448 8.4448]
    tu_myus: [5.1984 5.1984 11.3032 11.3032 5.1984 5.1984 3.4698 1.0651 1.0651]
    tu_psus: [4.9375 4.9375 13.8472 13.8472 4.9375 4.9375 3.9722 -1.2083 -1.2083]
    tu_pshs: [5.5278 5.5278 11.5556 11.5556 5.5278 5.5278 3.7778 2 2]
     tu_sl: [9.0000 9.0000 9.0000 9.0000 9.0000 9.0000 9.0000 9.0000 9.0000]
    tu_csl: [9 9 9 9 9 9 9 9 9]
   tu_slsh: [7.6319 7.6319 8.9815 8.9815 7.6319 7.6319 8.9815 11.7639 11.7639]
    tu_asl: [7.6319 7.6319 8.9815 8.9815 7.6319 7.6319 8.9815 11.7639 11.7639]
     tu_cs: [3 5 6 12 20 24 36 40 48 68 132 192 260 320 384]
    tu_ptn: [51 76 384]
     tu_us: [1 2 4 8 15 16 32 60 63 64 76 79 124 127 128 256 384]
     tu_hs: [15 60 63 76 384]
   us_valid: 1
   hs_valid: 1
   tuvalues: [1x511 double]
    tusize: 511
  tuplayers: 9
    tutype: 'cv'
    tuessQ: 1
   tuformat: 'mattug'
      tumv: 81
     tumnQ: 0
     tuSi: [510 509 507 503 495 479 447 383 255]
     tuvi: [0 0 0 0 0 0 0 0 0]
    tustpt: []
```

Singling out again the position value w.r.t. the first hypergraph situation, we have reproduced the result of our serial computation. Notice that the payoffs do not add up to 81, the worth of the grand coalition of game *v_p* rather than 53, the maximal amount that can be distributed under the associated conference game.

```
>> rats(p_fclv.tu_pshs)

ans =

  199/36     199/36     104/9      104/9      199/36     199/36     34/9       2          2
```

It remains to compute the position value for the second specified hypergraph situation. This allows us to demonstrate the last method to determine all game values from scratch. We start by creating two new fresh class objects, called *fclv2* and *p_fclv2* respectively, and we set in the next step all coalition structures to these new class objects. Notice, that we now use the second hypergraph system ceteris paribus instead of the first, this is indicated through the input argument *hm2* instead of *hm1*. Thus,

```
>> fclv2=TuVal(v_p,'cv');
>> fclv2=setCoalitionStructures(fclv2,lm,pm,fm1,hm2);
>> p_fclv2=p_TuVal(v_p,'cv');
>> p_fclv2=p_setCoalitionStructures(p_fclv2,lm,pm,fm1,hm2);
```

Setting all game values from scratch can be accomplished with the method *p_setAllValues()* for the parallel case, the serial counterpart is named *setAllValues()*. Both methods require just one input argument, the class object to which the game values should be assigned. The serial evaluation lasts slightly more than 3.1 seconds, whereas a parallel computation has been completed in about 2.7 seconds. The performance gain is not even a second in comparison to a serial evaluation. This problem is too small in order to observe a real benefit from a parallel computation.

```
>> tic;fclv2=setAllValues(fclv2);toc
Elapsed time is 3.150991 seconds.
>> tic;p_fclv2=p_setAllValues(p_fclv2);toc
Elapsed time is 2.766988 seconds.

>> p_fclv2

p_fclv2 =

 p_TuVal with properties:

    tu_sh: [9 9 9 9 9 9 9 9 9]
    tu_ad: [4 4 3 3 4 4 3 2 2]
    tu_ow: [9 9 9 9 9 9 9 9 9]
    tu_my: [7.8282 7.8282 14.6746 8.4448 8.4448 8.4448 8.4448 8.4448 8.4448]
   tu_myus: [5.1984 5.1984 11.3032 11.3032 5.1984 5.1984 3.4698 1.0651 1.0651]
   tu_psus: [4.9375 4.9375 13.8472 13.8472 4.9375 4.9375 3.9722 -1.2083 -1.2083]
   tu_pshs: [4.8333 4.8333 13.1111 13.1111 4.8333 4.8333 3.4444 2 2]
    tu_sl: [9.0000 9.0000 9.0000 9.0000 9.0000 9.0000 9.0000 9.0000 9.0000]
    tu_csl: [9 9 9 9 9 9 9 9 9]
   tu_slsh: [7.6319 7.6319 8.9815 8.9815 7.6319 7.6319 8.9815 11.7639 11.7639]
    tu_asl: [7.6319 7.6319 8.9815 8.9815 7.6319 7.6319 8.9815 11.7639 11.7639]
    tu_cs: [3 5 6 12 20 24 36 40 48 68 132 192 260 320 384]
    tu_ptn: [51 76 384]
    tu_us: [1 2 4 8 15 16 32 60 63 64 76 79 124 127 128 256 384]
    tu_hs: [15 60 76 384]
  us_valid: 1
  hs_valid: 1
  tuvalues: [1x511 double]
   tusize: 511
  tuplayers: 9
    tutype: 'cv'
    tuessQ: 1
  tuformat: 'mattug'
     tumv: 81
    tumnQ: 0
     tuSi: [510 509 507 503 495 479 447 383 255]
     tuvi: [0 0 0 0 0 0 0 0 0]
   tustpt: []
```

One should notice, however, that in both cases the computation of the position value w.r.t. a union stable system requires more than 40 percent of the elapsed computation time. As consequence, even under

the parallel mode, the evaluation of the position value of a game with 10-players and a large union stable system can last more than half an hour.

It remains to compare the position value of the second hypergraph situation with the result obtained by Algaba et al. (2004, p. 471). Selecting out the associated position value, we can convince ourselves that the result is identical to that obtained by these authors.

```
>> rats(p_fclv2.tu_pshs)

ans =

   29/6      29/6      118/9      118/9      29/6      29/6      31/9       2        2
```

Thus, the difference between both position values is given by

```
>> rats(get(p_fclv2,'tu_pshs')-get(p_fclv,'tu_pshs'))

ans =

  -25/36    -25/36     14/9       14/9      -25/36    -25/36    -1/3       0        0
```

Again, to summarize our discussion we first enlist all methods provided by the class object TuVal,

- *TuVal* creates the class object TuVal;
- *setAllValues* sets all values listed below to the class object TuVal;
- *setCoalitionStructures* sets the required coalition structures to the class object TuVal;
- *setShapley* sets the Shapley value to the class object TuVal;
- *setADvalue* sets the Aumann-Dreze value to the class object TuVal;
- *setOwen* sets the Owen value to the class object TuVal;
- *setMyerson* sets the Myerson value to the class object TuVal;
- *setMyersonUS* sets the Myerson value w.r.t. a union stable system to the class object TuVal;
- *setPosition* sets the position value w.r.t. a union stable system to the class object TuVal;
- *setPositionHS* sets the position value w.r.t. a hypergraph system to the class object TuVal;
- *setSolidarity* sets the solidarity value to the class object TuVal;
- *setCoalitionSolidarity* sets the coalition solidarity value to the class object TuVal;
- *setSolidarityShapley* sets the solidarity Shapely value to the class object TuVal;
- *setApuSolidarity* sets the solidarity value w.r.t. a priori unions to the class object TuVal;

and finally all methods provided by the class object p_TuVal to start a parallel evaluation.

- *p_TuVal* creates the class object p_TuVal.
- *p_setAllValues* sets all values listed below to the class object p_TuVal.
- *p_setCoalitionStructures* sets the required coalition structures to the class object p_TuVal.
- *p_setShapley* sets the Shapley value to the class object p_TuVal.
- *p_setADvalue* sets the Aumann-Dreze value to the class object p_TuVal.
- *p_setOwen* sets the Owen value to the class object p_TuVal.
- *p_setMyerson* sets the Myerson value to the class object p_TuVal.
- *p_setMyersonUS* sets the Myerson value w.r.t. a union stable system to the class object p_TuVal.

- *p_setPosition* sets the position value w.r.t. a union stable system to the class object p_TuVal.
- *p_setPositionHS* sets the position value w.r.t. a hypergraph system to the class object p_TuVal.
- *p_setSolidarity* sets the solidarity value to the class object p_TuVal.
- *p_setCoalitionSolidarity* sets the coalition solidarity value to the class object p_TuVal.
- *p_setSolidarityShapley* sets the solidarity Shapely value to the class object p_TuVal.
- *p_setApuSolidarity* sets the solidarity value w.r.t. a priori unions to the class object p_TuVal.

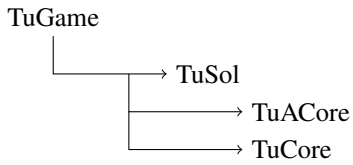
4.4. Class Object TuCore

The class object TuCore is a subclass of TuSol. Since TuSol is a subclass object of TuGame, TuCore inherits besides all properties and methods from TuSol also everything from the super-class TuGame. Apart from setting solution concepts the class object TuCore stores the core and imputation vertices to draw the core for three and four person games whenever it exists.

There also exists a class object TuACore to create and store the relevant data to plot the anti core of a TU game, if it is non-empty. This class object resides on the same hierarchy level as the class object TuCore. Since the use of both methods is identical we only focus on class object TuCore in the sequel.

To recall the hierarchy structure of the class object TuCore and TuACore have a look on the next Figure.

Figure 7: Class Object TuCore Hierarchy



In order discuss the usage of the class object TuCore, we depart for the moment from our main example and resume the bankruptcy game example of Subsection 2.2 to be able to draw a core of a four person game, the largest dimension we can handle to draw a three dimensional core. The coalitional values of the four-person bankrutpcy game are given by

```
>> bv

bv =

   0   0   0   0   0   0  40   0  10  40 100 100 160 190 250
```

Creating a class object TuCore is done with the method *TuCore()*. At the moment a class object TuCore will be created, an event will be triggered to compute all core and imputation vertices.

```
>> clv_cr=TuCore(bv,'cv')
size = 16 x 5
Number Type = real
(Initially added rows ) = 7 11 13 14 15
(Iter, Row, #Total, #Curr, #Feas)=   6  12    7   6   0
(Iter, Row, #Total, #Curr, #Feas)=   7  10    8   6   0
(Iter, Row, #Total, #Curr, #Feas)=   8   9   11   8   0
(Iter, Row, #Total, #Curr, #Feas)=   9   8   20  15   3
(Iter, Row, #Total, #Curr, #Feas)=  10   4   27  17   4
```

```
(Iter, Row, #Total, #Curr, #Feas)=  11   2    34   17   7
(Iter, Row, #Total, #Curr, #Feas)=  12   6    34   17   7
(Iter, Row, #Total, #Curr, #Feas)=  13   1    41   17   12
(Iter, Row, #Total, #Curr, #Feas)=  14   5    41   17   12
(Iter, Row, #Total, #Curr, #Feas)=  15   3    41   17   12
(Iter, Row, #Total, #Curr, #Feas)=  16  16    41   12   12
size = 6 x 5
Number Type = real
(Initially added rows ) = 1 2 3 4 5
(Iter, Row, #Total, #Curr, #Feas)=   6   6    5    4    4

clv_cr =

 TuCore with properties:

    tu_crv: [12x4 double]
   tu_cddv: [12x4 double]
    tu_imp: [4x4 double]
   tu_cddi: [4x4 double]
   tu_vals: [0 0 0 0 0 0 40 0 10 40 100 100 160 190 250]
    tu_prk: [0.1200 0.1800 0.3000 0.4000]
    tu_prn: [0.1200 0.1800 0.3000 0.4000]
   tu_prk2: []
     tu_sh: [0.1167 0.1767 0.2967 0.4100]
   tu_tauv: [0.1176 0.1765 0.2941 0.4118]
    tu_bzf: []
   tu_aprk: [0.1000 0.1600 0.3200 0.4200]
  prk_valid: 1
  prn_valid: 1
 prk2_valid: 0
 aprk_valid: 1
  tuvalues: [0 0 0 0 0 0 0.1600 0 0.0400 0.1600 0.4000 0.4000 0.6400 0.7600 1]
    tusize: 15
 tuplayers: 4
    tutype: 'cv'
    tuessQ: 1
  tuformat: 'mattug'
      tumv: 1
     tumnQ: 0
     tuSi: [14 13 11 7]
     tuvi: [0 0 0 0]
    tustpt: []
```

The class object TuCore provides the following additinonal properties

- *tu_crv* stores the core vertices of the zero-one normalized game (cdd);
- *tu_cddv* stores the core vertices of the zero-one normalized game (cddmex);
- *tu_imp* stores the imputation vertices of the zero-one normalized game (cdd);
- *tu_cddi* stores the imputation vertices of the zero-one normalized game (cddmex);
- *tu_vals* stores the original game information.

Inspecting the data of the class object *clv_cr*, we realize that the game solutions are also presented in terms of the corresponding zero-one normalized game. A zero-one normalization is useful to set a strong ϵ-core and a core in correct relation.

The class object *clv_cr* contains all relevant data to draw the core of the game with respect to the pre-kernel, pre-nucleolus and Shapley value. The core can now be drawn without invoking any new computation whenever the function *CorePlot()* and *CddCorePlot()* will be overloaded. The reader should compare this approach with the approach described in Subsection 2.10.

In order to plot the core we call the overloading function *CorePlot()* with the following arguments

```
>> CorePlot(clv_cr,'all',1)
```

Alternatively, we can also plot the core with the data created by the *Cddmex* library while using the overloading function *CddCorePlot()* instead.

```
>> CddCorePlot(clv_cr,'all',1)
```

In case that more information about the usage and creating of a class object are required, one types help ClassObjectName on the Matlab command line. Referred to the current example, we type

```
>> help TuCore
 TUCORE is a subclass object of TUSOL to perform several computations for retrieving
 and modifying game data. It stores relevant game information needed to draw the
 core/imputation set by overloading functions.

 Usage: clv = TuCore(v,'gtype','gformat')

 Define variables:
 output:
 clv       -- TuCore class object (subclass of TuSol).

 input:
 v        -- A Tu-Game v of length 2^n-1.
 gtype      -- A string to define the game type.
               Permissible types are:
               -- 'cv' (convex/average-convex, semi-convex).
               -- 'cr' game with non-empty core.
               -- 'sv' simple game.
               -- 'acr' game with non-empty anti core.
               -- ''  empty string. (default)
 gformat     -- A string to define the game format.
               Permissible formats are:
               -- 'mattug' i.e., unique integer representation to perform computation
                     under MatTuGames. (default)
               -- 'mama'  i.e. generic power set representation, i.e Mathematica.

 TuCore properties:
  tu_crv     -- stores the core vertices of the zero-one normalized game (cdd).
  tu_cddv    -- stores the core vertices of the zero-one normalized game (cddmex).
  tu_imp     -- stores the imputation vertices of the zero-one normalized game (cdd).
  tu_cddi    -- stores the imputation vertices of the zero-one normalized game (cddmex).
  tu_vals    -- stores the original game information.


 Properties inherihed from the superclass TuSol
  tu_prk     -- stores a pre-kernel element of the zero-one normalized game.
  tu_prk2    -- stores a second pre-kernel element instead of the pre-nucleolus.
  tu_prn     -- stores the pre-nucleolus of the zero-one normalized game.
  tu_sh      -- stores the Shapley value of the zero-one normalized game.
  tu_tauv    -- stores the Tau value of the zero-one normalized game.
  tu_bzf     -- stores the Banzhaf value of the zero-one normalized game.
  tu_aprk    -- stores the anti prek-kernel of the zero-one normalized game.
  prk_valid   -- returns 1 if tu_prk stores a pre-kernel element, otherwise 0.
  prk2_valid  -- returns 1 if tu_prk2 stores a pre-kernel element, otherwise 0.
  prn_valid   -- returns 1 if tu_prn stores the pre-nucleolus, otherwise 0.

  Properties inherited from the superclass TuGame:

  tuvalues    -- stores the characteristic values of a Tu-game.
  tusize     -- stores the length of the game array/vector.
  tuplayers   -- stores the number of players involved.
  tutype     -- stores the game type information ('cv','cr', or 'sv').
```

```
 tuessQ    -- stores if the game is essential.
 tuformat   -- stores the format how the game is represented.
 tumv      -- stores the value of the grand coalition.
 tumnQ     -- stores the information whether a proper coalition has a higher value
          than the grand coalition
 tuSi      -- stores the coalitions having size of n-1.
 tuvi      -- stores the values of singelton coalitions.
 tustpt     -- stores a starting point for doing computation. Has lower priority than
          providing a second input argument as an alternative starting point.
          Thus, if a starting point is provided as second input argument the
          information stored in tustpt will not be used.

TuCore methods:
 TuCore       -- creates the class object TuCore.
 setAllSolutions -- sets all solutions listed below to the class object TuSol.
 setPreKernel   -- sets a pre-kernel element to the class object TuSol.
 setPreNuc     -- sets the pre-nucleolus to the class object TuSol.
 setShapley     -- sets the Shapley value to the class object TuSol.
 setTauValue    -- sets the Tau value to the class object TuSol.
 setBanzhaf     -- sets the Banzhaf value to the class object TuSol.
 setAntiPreKernel -- sets an anti pre-kernel element to the class object TuSol.

 Methods inherited from the superclass TuGame:

 startpt    -- sets a starting point for doing computation.
```

Information of a method can be attained with *help method_name* . For instance, getting some help for the method *setAllSolutions()*, it is enough to type

```
>> help setAllSolutions
--- help for TuSol/setAllSolutions ---

 SETALLSOLUTIONS sets a set of game solutions to the class object TuSol.

 Usage: clv = setAllSolutions(clv)

 output:
  clv    -- TuSol class object.

 input:
   clv    -- TuSol class object.
```

to get some basic information about the usage and its arguments.

4.5. Class Object TuVert

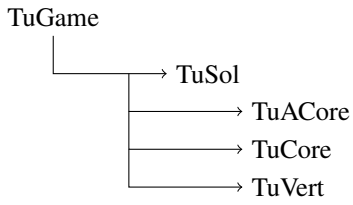
Similar to the class object TuCore the class object TuVert is a subclass of TuSol. But due to the fact that TuSol is a subclass object of TuGame, TuVert inherits besides all properties and methods from TuSol also everything from the super-class TuGame. However, TuVert is on the same hierarchy level as TuCore nothing can be inherited or bequeathed between these two subclasses. Apart from setting solution concepts the class object TuVert sets also the core and imputation vertices triggered by an event relying on two different methods.

To recall the hierarchy structure of the class objects TuCore and TuVert have a look on the next Figure.

Creating a class object TuVert is done by the above described procedure. Here, we changed slightly the procedure to create a class object while calling the method *TuVert()* with three arguments.

```
>> clv_cvr = TuVert(v,'cr','mattug')
```

Figure 8: Class Object TuVert Hierarchy




```
clv_cvr =

 TuVert with properties:

    tu_crv: []
   tu_cddv: []
    tu_imp: []
   tu_cddi: []
    tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
   tu_prk2: []
     tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
   tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
    tu_bzf: []
   tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
  prk_valid: 1
  prn_valid: 1
 prk2_valid: 0
 aprk_valid: 1
   tuvalues: [1x63 double]
    tusize: 63
  tuplayers: 6
    tutype: 'cr'
    tuessQ: 1
  tuformat: 'mattug'
      tumv: 24
     tumnQ: 0
     tuSi: [62 61 59 55 47 31]
     tuvi: [0 0 0 0 0 0]
    tustpt: []
```

In contrast to TuSol where one has to execute a computation for setting the solutions, TuVert sets all the solutions from scratch. However, the computation of the vertices of the core can be quite intense, since at most $n!$ vertices must be computed, so this is left open to the user. The user should be aware about the game type and size to estimate the expected time for evaluating all vertices.

The following new properties were provided by the class object TuVert

- *tu_crv* stores the core vertices (cdd);
- *tu_cddv* stores the core vertices (cddmex);
- *tu_imp* stores the imputation vertices (cdd);
- *tu_cddi* stores the imputation vertices (cddmex).

The two different methods for evaluating the vertices of the core or imputation set are already described in the Subsection 2.10.

Now, let us evaluate all vertices of the core and imputation set. Since, we study a game with 6-persons, this can be done relatively quickly, since at most 720 vertices must be computed under the worst case.

However, this game is an assignment game much less vertices need to be computed. Thus, it is a relatively safe affair to compute all vertices. In this example only 16 core vertices and 6 vertices of the imputation set need to be computed while calling the method *setAllVertices()* as we can see from below.

```
>> clv_crv=setAllVertices(clv_cvr)
size = 64 x 7
Number Type = real
(Initially added rows ) = 29 31 54 59 61 62 63
(Iter, Row, #Total, #Curr, #Feas)=   8   55     16   13    1
(Iter, Row, #Total, #Curr, #Feas)=   9   53     24   17    2
(Iter, Row, #Total, #Curr, #Feas)=  10   43     33   21    2


clv_crv =

 TuVert with properties:

    tu_crv: [16x6 double]
   tu_cddv: [16x6 double]
    tu_imp: [6x6 double]
   tu_cddi: [6x6 double]
    tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
   tu_prk2: []
     tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
   tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
    tu_bzf: []
   tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
  prk_valid: 1
  prn_valid: 1
 prk2_valid: 0
 aprk_valid: 1
   tuvalues: [1x63 double]
     tusize: 63
  tuplayers: 6
    tutype: 'cr'
    tuessQ: 1
  tuformat: 'mattug'
      tumv: 24
     tumnQ: 0
     tuSi: [62 61 59 55 47 31]
     tuvi: [0 0 0 0 0 0]
    tustpt: []
```

To retrieve the data stored on the property *clv_crv.tu_crv* in order to get the whole list of vertices computed, one simply types

```
>> clv_crv.tu_crv

ans =

   0   0   0   8   9   7
   0   1   0   8   9   6
   4   0   0   4   9   7
   4   5   0   4   9   2
   8   5   9   0   0   2
   8   3   7   0   2   4
   8   3   4   0   5   4
   4   5   9   4   0   2
   6   7   2   2   7   0
   6   7   9   2   0   0
   8   7   4   0   5   0
   8   7   9   0   0   0
```

```
   5   0   4   3   5   7
   5   0   1   3   8   7
   0   1   5   8   4   6
   0   0   4   8   5   7
```

Similar for retrieving the contents on the property *clv_crv.tu_cddv*, hence

```
>> clv_crv.tu_cddv

ans =

  5.0000     0  4.0000  3.0000  5.0000  7.0000
  8.0000  3.0000  7.0000     0  2.0000  4.0000
     0  1.0000 -0.0000  8.0000  9.0000  6.0000
     0     0 -0.0000  8.0000  9.0000  7.0000
  8.0000  7.0000  9.0000     0 -0.0000     0
  6.0000  7.0000  9.0000  2.0000 -0.0000     0
  8.0000  5.0000  9.0000     0 -0.0000  2.0000
  4.0000  5.0000  9.0000  4.0000 -0.0000  2.0000
     0  1.0000  5.0000  8.0000  4.0000  6.0000
     0     0  4.0000  8.0000  5.0000  7.0000
  6.0000  7.0000  2.0000  2.0000  7.0000     0
  8.0000  7.0000  4.0000     0  5.0000     0
  4.0000  5.0000 -0.0000  4.0000  9.0000  2.0000
  4.0000     0 -0.0000  4.0000  9.0000  7.0000
  8.0000  3.0000  4.0000     0  5.0000  4.0000
  5.0000     0  1.0000  3.0000  8.0000  7.0000
```

Instead of setting immediately all vertices, it is also possible to set the vertices step by step with the methods enlisted below. This might be useful for situations where it is not so clear in advance how many vertices must be computed and what might be the elapsed computation time.

We summarize our discussion while enlisting all methods related to the class object TuVert.

- *TuVert* creates the class object TuVert.
- *setAllVertices* sets all vertices listed below.
- *setVertices* sets all core vertices (cdd).
- *setCddVertices* sets all core vertices (cddmex).
- *setImpVertices* sets the imputation vertices (cdd).
- *setCddImpVertices* sets the imputation vertices (cddmex).

4.6. Class Object TuRep

The class object TuRep is a subclass of the class object TuSol. This class object resides on the same hierarchy level as the class objects TuCore and TuVert. Resume this from the hierarchy structure Figure presented below. The class object TuRep is designed to perform the relatively complex task of replicating a pre-kernel element appropriately, and to assist in that use.

There exists also a class object p_TuRep with the same attributes as the class object TuRep but with one crucial difference, the former is attributed for doing computation in parallel whereas the latter for performing the same operations in serial. Moreover, the class objects p_TuRep and TuRep are subclass objects of the class TuGame and inherits therefore all properties from it. However, the former is a subclass of the class object p_TuSol whereas the latter is a subclass of the class object TuSol. Hence, both class objects resides on the same hierarchy level (see the Figure below). Since the usage of both methods is identical – only the names of the methods differ – we discard the discussion of the latter.

Figure 9: Class Object TuRep Hierarchy

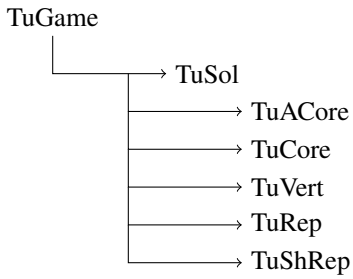
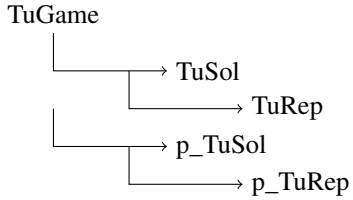


Figure 10: Class Object p_TuRep Hierarchy



Creating a class object *clv_rep* is done while calling the method *TuRep()* in an appropriate manner.

```
>> clv_rep = TuRep(v,'cr')

clv_rep =

 TuRep with properties:

     RepSol: []
      tu_x: []
       scl: 1
       smc: 1
  x_prk_valid: 0
     tu_prk: []
     tu_prn: []
    tu_prk2: []
     tu_sh: []
    tu_tauv: []
     tu_bzf: []
    tu_aprk: []
   prk_valid: 0
   prn_valid: 0
  prk2_valid: 0
  aprk_valid: 0
    tuvalues: [1x63 double]
     tusize: 63
   tuplayers: 6
     tutype: 'cr'
     tuessQ: 1
    tuformat: 'mattug'
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

The class object TuRep provides the following additional properties

- *RepSol* stores the result of a pre-kernel element replication (cf. Subsection 2.7);
- *tu_x* stores the pre-kernel vector that has been replicated;
- *x_prk_valid* returns 1 if *tu_x* is a pre-kernel element;
- *scl* stores the scaling factor μ, default value is one;
- *smc* stores the cardinality of most effective coalitions, that is, smallest/largest (1/0), default value is one.

Our purpose is to calculate the spanning system within the game space that replicates a pre-kernel element of the default game *v* as a pre-kernel element of all games which belongs to this subspace of linear independent games. For replicating a pre-kernel element remember first its components.

```
>> prk_v

prk_v =

  5.0000   3.0000   4.0000   3.0000   5.0000   4.0000
```

The set of linear independent games that replicates the above pre-kernel element is obtained with the method *setReplicate_Prk()*. The class object *clv_rep* contains just the default values. Calling the method *setReplicate_Prk()* triggers an event to compute all missing relevant data. As input arguments one has to supply the class object which should be updated, hence *clv_rep*, a pre-kernel element of the original game *clv_sol.tu_prk* and a scaling factor μ. In this case, we set this value to -0.4, whereas its default value is set to 1. The first two input arguments are obligatory.

```
>> clv_rep=setReplicate_Prk(clv_rep,clv_sol.tu_prk,-0.4)

>> clv_rep

clv_rep =

 TuRep with properties:

     RepSol: [1x1 struct]
       tu_x: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
        scl: -0.4000
        smc: 1
  x_prk_valid: 1
     tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
     tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_prk2: []
     tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
    tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
     tu_bzf: []
    tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
   prk_valid: 1
   prn_valid: 1
  prk2_valid: 0
  aprk_valid: 1
    tuvalues: [1x63 double]
     tusize: 63
   tuplayers: 6
     tutype: 'cr'
     tuessQ: 1
    tuformat: 'mattug'
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

Retrieving the contents of the property *clv_rep.RepSol* to see whether the pre-kernel element is replicable can be accomplished with

```
>> clv_rep.RepSol

ans =

     V_PrkQ: [0 0 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]
      V_SPC: [57x63 double]
       SBCQ: [0 0 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]
        SBC: [1x1 struct]
      Mat_W: [16x63 double]
    P_Basis: [57x63 double]
 VarP_Basis: [57x63 double]
```

Here, we remember the meaning of the output arguments that have been already discussed in Subsection 2.7. The first field variable *clv_rep.RepSol.V_PrkQ* yields a sequence of games that possess *clv_sol.tu_prk* as pre-kernel element indicated by a one (true) or not indicated by a zero (false). The second field variable *clv_rep.RepSol.V_SPC* yields the list of all linear independent games which were derived from the assignment game v. The third field variable *clv_rep.RepSol.SBCQ* indicates whether the equivalence class that contains the element *clv_sol.tu_prk* has changed or has remained invariant against a change in the basis of the game parameter. The fourth output variable *clv_rep.RepSol.SBC* contains the information about the equivalence classes for all linear independent games. It returns a field that contains the information about the set of most effective coalitions that constitute the equivalence classes containing *clv_sol.tu_prk*. The fifth field variable *clv_rep.RepSol.Mat_W* returns the matrix W of Equation 7.5.1 in Meinhardt (2013). Furthermore, the field variable *clv_rep.RepSol.P_Basis* contains the information about the scaled up parameter basis by the scaling factor μ. Finally, the field variable *clv_rep.RepSol.VarP_Basis* returns the amplitude of the change in the game values with respect to the default game v.

Here, we observe that just one game allows the replication of the above pre-kernel element. In addition, to notice that the game that replicates the pre-kernel element is really different from the initial game, we verify the differences of both game with

```
>> tol=10^6*eps;
>> abs(clv_rep.RepSol.V_SPC(4,:)-v)<tol

ans =

  Columns 1 through 23

    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0

  Columns 24 through 46

    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0

  Columns 47 through 63

    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    0    1
```

The unique coalition value that coincides in both games is the value of the grand coalition. In the next step, we check that the pre-kernel element is also a pre-kernel point of the selected game. And indeed, we obtain an affirmation by

```
>> PrekernelQ(clv_rep.RepSol.V_SPC(4,:),prk_v)

ans =

   1
```

or we can again calculate the pre-kernel point of the selected game directly while overloading the function *PreKernel()*.

```
>> PreKernel(clv_rep.RepSol.V_SPC(4,:))

ans =

  5.0000   3.0000   4.0000   3.0000   5.0000   4.0000
```

Hence, this solution point is also a pre-kernel element of the selected as well as of our initial game.

The intented reader will ask whether is not possible to exchange data between class objects of different hierarchy levels. Of course, this is possible through a copy command. To demonstrate how one can copy data between class objects of different hierarchy levels, we create a fresh class object TuRep called *clv_rep2*.

```
>> clv_rep2=TuRep(v,'cr')

clv_rep2 =

 TuRep with properties:

     RepSol: []
       tu_x: []
        scl: 1
        smc: 1
  x_prk_valid: 0
     tu_prk: []
     tu_prn: []
    tu_prk2: []
     tu_sh: []
    tu_tauv: []
     tu_bzf: []
    tu_aprk: []
   prk_valid: 0
   prn_valid: 0
  prk2_valid: 0
  aprk_valid: 0
    tuvalues: [1x63 double]
     tusize: 63
   tuplayers: 6
     tutype: 'cr'
     tuessQ: 1
    tuformat: 'mattug'
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

We know that we have alreay assigned the solutions to the class object *clv_sol*, that is, we have

```
>> clv_sol
```

```
clv_sol =

 TuSol with properties:

    tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
   tu_prk2: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
   tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
    tu_bzf: []
   tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
  prk_valid: 1
  prn_valid: 1
 prk2_valid: 1
 aprk_valid: 1
   tuvalues: [1x63 double]
    tusize: 63
  tuplayers: 6
    tutype: 'cr'
    tuessQ: 1
  tuformat: 'mattug'
     tumv: 24
     tumnQ: 0
     tuSi: [62 61 59 55 47 31]
     tuvi: [0 0 0 0 0 0]
   tustpt: []
```

Instead of executing a complete new computation to assign the solution data to the class object *clv_rep2* we simply copy the data from *clv_sol* to the class object *clv_rep2* with the method *copyTuSol()*. This method requires two input argument and one output argument. The input arguments are the class objects *clv_rep2* and *clv_sol*. That is, the class object that should be modified and the class object from which the data should be copied. One has to keep exactly this order. The output argument is the class object that should receive the data, this means, the class object which should be updated.

```
>> clv_rep2=copyTuSol(clv_rep2,clv_sol)

clv_rep2 =

 TuRep with properties:

    RepSol: []
     tu_x: []
      scl: 1
      smc: 1
 x_prk_valid: 0
    tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
   tu_prk2: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
   tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
    tu_bzf: []
   tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
  prk_valid: 1
  prn_valid: 1
 prk2_valid: 1
 aprk_valid: 1
   tuvalues: [1x63 double]
    tusize: 63
  tuplayers: 6
    tutype: 'cr'
    tuessQ: 1
  tuformat: 'mattug'
```

```
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

With this copy command we have avoided the repetition of a costly computation. Now, we can proceed as above to find out the game space that allows a replication of a pre-kernel element. Thus, we do

```
>> clv_rep2=setReplicate_Prk(clv_rep2,clv_rep2.tu_prk,-0.4)

clv_rep2 =

  TuRep with properties:

      RepSol: [1x1 struct]
        tu_x: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
         scl: -0.4000
         smc: 1
 x_prk_valid: 1
      tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
      tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
     tu_prk2: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
       tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
     tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
      tu_bzf: []
     tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
   prk_valid: 1
   prn_valid: 1
  prk2_valid: 1
  aprk_valid: 1
    tuvalues: [1x63 double]
      tusize: 63
   tuplayers: 6
      tutype: 'cr'
      tuessQ: 1
    tuformat: 'mattug'
        tumv: 24
       tumnQ: 0
        tuSi: [62 61 59 55 47 31]
        tuvi: [0 0 0 0 0 0]
      tustpt: []
```

This is exactly the result we obtained by our first computation. To observe this, retrieve the data contained in the property *clv_rep2.RepSol*

```
>> clv_rep2.RepSol

ans =

      V_PrkQ: [0 0 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]
       V_SPC: [57x63 double]
        SBCQ: [0 0 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]
         SBC: [1x1 struct]
       Mat_W: [16x63 double]
     P_Basis: [57x63 double]
  VarP_Basis: [57x63 double]
```

and compare now the games that allow a pre-kernel replication by

```
>> abs(clv_rep.RepSol.V_SPC(4,:)-clv_rep2.RepSol.V_SPC(4,:))<tol
```

```
ans =

 Columns 1 through 23

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1

 Columns 24 through 46

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1

 Columns 47 through 63

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

Both games coincides within a numerical tolerance level, hence, they are identical. What we want to establish.

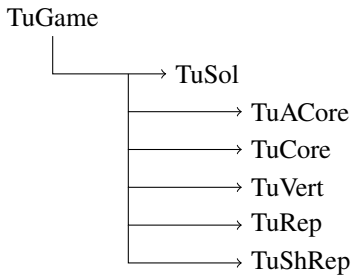
To summarize our discussion we enlist the new set of methods provided by the class object TuRep below.

- *TuRep* creates the class object TuRep.
- *setReplicate_Prk* replicates a pre-kernel solution $\vec{x}$ as a pre-kernel of the game space *v_sp*.
- *copyTuSol* copies all solutions from TuSol/p_TuSol to TuRep.
- *copy_p_TuRep* copies replication results from p_TuRep to TuRep.

4.7. Class Object TuShRep

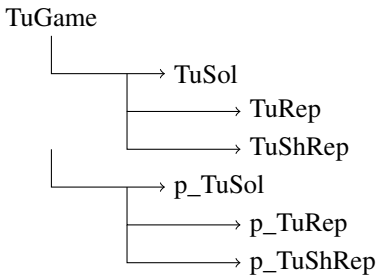
The class object TuShRep is a subclass of the class object TuSol. This class object resides on the same hierarchy level as the class objects TuACore, TuCore, TuVert, and TuRep. Resume this from the hierarchy structure Figure presented below. The class object TuShRep is designed to perform the relatively complex task of replicating the Shapley value of a default game appropriately.

Figure 11: Class Object TuShRep Hierarchy



Similar to the class object TuRep it also exists for the class object TuShRep a counterpart for doing all computation in parallel rather than in serial. This class object is referred to p_TuShRep which have the same attributes as the class object TuShRep. Moreover, the class objects p_TuShRep and TuShRep are subclass objects of the class TuGame and inherits therefore all properties from it. However, the former is a subclass of the class object TuSol whereas the latter is a subclass of the class object p_TuSol. Hence, both class objects resides on the same hierarchy level. See the Figure below to retrieve its hierarchy level. Again, the usage of both methods is identical – only the names of the methods differ – we discard the discussion of the latter.

Figure 12: Class Object p_TuShRep Hierarchy



In a first step we have to create a class object, referred to this time *clv_shrep*, while supplying the TU game *v* and the game type *cr*, thus

```
>> clv_shrep=TuShRep(v,'cr')

clv_shrep =

 TuShRep with properties:

    RepShap: []
      tu_x: []
       scl: 1
       tol: 2.2204e-10
  x_sh_valid: 0
     tu_prk: []
     tu_prn: []
    tu_prk2: []
     tu_sh: []
    tu_tauv: []
     tu_bzf: []
    tu_aprk: []
   prk_valid: 0
   prn_valid: 0
  prk2_valid: 0
  aprk_valid: 0
    tuvalues: [1x63 double]
     tusize: 63
   tuplayers: 6
     tutype: 'cr'
     tuessQ: 1
    tuformat: 'mattug'
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

The class object TuShRep provides the following additional properties

- *RepShap* stores the result of a Shapley value replication (cf. Subsection 2.8);
- *tu_x* stores the Shapley value that has been replicated;
- *x_sh_valid* returns 1 if *tu_x* is the Shapley value;
- *scl* stores the scaling factor μ, default value is one;.
- *tol* stores the tolerance value.

In the course of our discussion, we want to perform the task to replicate the Shapley value for a set

of linear independent game derived from the above assignment game v. For this purpose, we recall the Shapley value of the assignment game, which is given by.

```
>> clv_sol.tu_sh

ans =

  4.1167   3.5167   4.3667   3.8333   4.5833   3.5833
```

The set of linear independent games that replicates the Shapley value of the original game is obtained with the method *setReplicate_Shapley()*. The class object *clv_shrep* contains just the default values. The missing values must be computed while invoking the method *setReplicate_Shapley()* which triggers an event to compute all missing relevant data. As an input argument it is obligatory to issue the class object which should be updated, this is in this case the object *clv_shrep*. All other arguments are optional. Optional arguments are the Shapley value of the default game, a scaling factor μ, and a tolerance value *tol*. Instead of supplying the Shapley value it is also allowed to supply the empty set. In this concrete example, we use the Shapley value stored in the class object *clv_sol* and setting the scaling factor μ to 10, default value is 1.

```
>> clv_shrep=setReplicate_Shapley(clv_shrep,clv_sol.tu_sh,10)

clv_shrep =

 TuShRep with properties:

    RepShap: [1x1 struct]
      tu_x: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
       scl: 10
       tol: 2.2204e-10
  x_sh_valid: 1
     tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
     tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_prk2: []
     tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
    tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
     tu_bzf: []
    tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
   prk_valid: 1
   prn_valid: 1
  prk2_valid: 0
  aprk_valid: 1
    tuvalues: [1x63 double]
     tusize: 63
   tuplayers: 6
     tutype: 'cr'
     tuessQ: 1
    tuformat: 'mattug'
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

The property *clv_rep.RepShap* contains all information related to the process of replicating the Shapley value of the original game. We retrieve the contents of this element by

```
>> clv_shrep.RepShap
```

```
ans =

    shQ: 1
   shQv: [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]
    shm: [58x6 double]
     vm: [58x63 double]
```

We observe that all derived games replicates the Shapley value, indicated by *clv_shrep.RepShap.shQ* the one (true) and the sequence of ones (true) given by *clv_shrep.RepShap.shQv*. Additional output arguments are the matrix of computed Shapley values *clv_shrep.RepShap.shm* and the set of linear independent games that replicates the Shapley value of the game under consideration, that is *clv_shrep.RepShap.vm*.

According to the fact that the derived system of related games which replicates the Shapley value of the original game is too large to be displayed here, we only verify that all these games are indeed linear independent.

```
>> rank(clv_shrep.RepShap.vm)

ans =

   58
```

The result confirms that all 58 are linear independent. But the dimension of the spanning system of games is larger than $2^n - 1 - n = 2^6 - 1 - 6 = 57$ indicating again that the linear manifold of games induced by the linear basis of a TU game is a subspace of the null space of the Shapley value.

Next we present only a fraction of the matrix of computed Shapley values *clv_shrep.RepShap.shm* to observe whether the Shapley values are identical. Here are the first ten rows of the matrix

```
>> clv_shrep.RepShap.shm(1:10,:)

ans =

    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
```

and here are the last ten rows of the Shapley value matrix

```
>> clv_shrep.RepShap.shm(48:58,:)

ans =

    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
    4.1167    3.5167    4.3667    3.8333    4.5833    3.5833
```

As we have mentioned above, it is also allowed to supply as a second input argument instead of the Shapley value the empty set. This deserve some further discussion. For this purpose, we create a new class object *clv_shrep2* to make this point more explicit.

```
>> clv_shrep2=TuShRep(v,'cr')

clv_shrep2 =

 TuShRep with properties:

    RepShap: []
      tu_x: []
       scl: 1
       tol: 2.2204e-10
  x_sh_valid: 0
     tu_prk: []
     tu_prn: []
    tu_prk2: []
     tu_sh: []
    tu_tauv: []
     tu_bzf: []
    tu_aprk: []
   prk_valid: 0
   prn_valid: 0
  prk2_valid: 0
  aprk_valid: 0
    tuvalues: [1x63 double]
     tusize: 63
   tuplayers: 6
     tutype: 'cr'
     tuessQ: 1
    tuformat: 'mattug'
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

In principle every solution or payoff vector can be supplied to the method *setReplicate_Shapley()*. In a first step the supplied non-empty payoff vector will be verified if it is a Shapley value of the default game. If not the vector will be written on the property *tu_x* but indicated as an invalid Shapley value, that is *x_sh_valid: 0*. However, in case that an empty argument is supplied, this empty vector will be substituted by the correct Shapley value. In the next step, it will then be written on *tu_x* and, of course, be indicated as a valid Shapley value, that is *x_sh_valid: 1*.

This can be seen in the next example, where we supply the method *setReplicate_Shapley()* with an empty value, setting the scaling factor of the spanning system μ to 10, and decreasing slightly the numerical tolerance from $10^6 * eps$ to $10^5 * eps$ for which a computed Shapley value from the subspace of linear independent games is considered as valid.

```
>> tol=10^5*eps

tol =

  2.2204e-11

>> clv_shrep2=setReplicate_Shapley(clv_shrep2,[],10,tol)

clv_shrep2 =
```

```
TuShRep with properties:

    RepShap: [1x1 struct]
       tu_x: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
        scl: 10
        tol: 2.2204e-11
  x_sh_valid: 1
     tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
     tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_prk2: []
      tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
    tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
     tu_bzf: []
    tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
   prk_valid: 1
   prn_valid: 1
  prk2_valid: 0
  aprk_valid: 1
    tuvalues: [1x63 double]
     tusize: 63
   tuplayers: 6
     tutype: 'cr'
     tuessQ: 1
    tuformat: 'mattug'
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

Retrieving the return argument *clv_shrep2.RepShap* of the computation, we realize that all derived game have the same Shapley value.

```
>> clv_shrep2.RepShap

ans =

   shQ: 1
  shQv: [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]
   shm: [58x6 double]
    vm: [58x63 double]
```

For completeness, we enlist the new set of methods provided by the class object TuShRep below.

- *TuShRep* creates the class object TuShRep.
- *setReplicate_Shapley* replicates the Shapley value of game *v* as the Shapley vlaue of the game space *vm*.
- *copyTuSol* copies all solutions from TuSol/p_TuSol to TuShRep.
- *copy_p_TuShRep* copies replication results from p_TuShRep to TuShRep.

We close this subsection while enlisting the complete list of variables created by its corresponding class identifier.

```
>> whos
 Name       Size        Bytes  Class    Attributes

 ans        1x63          63  logical
 clv        1x1          112  TuGame
 clv_crv     1x1          112  TuVert
 clv_cvr     1x1          112  TuVert
```

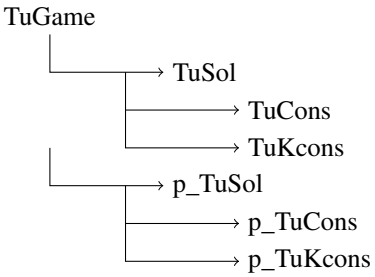
```
clv_pr     1x1         112 TuProp
clv_rep    1x1         112 TuRep
clv_rep2   1x1         112 TuRep
clv_shrep  1x1         112 TuShRep
clv_shrep2 1x1         112 TuShRep
clv_sol    1x1         112 TuSol
clv_sol1   1x1         112 TuSol
n          1x1         8 double
name       1x7         14 char
old_dir    1x28        56 char
pfm        3x3         72 double
prk_v      1x6         48 double
pwd_path   1x29        58 char
sl         1x1         8 double
slv        1x3         24 double
slv_sol    1x1         112 TuSol
tol        1x1         8 double
v          1x63        504 double
v1         1x63        504 double
```

4.8. Class Objects p_TuCons and p_TuKcons

The class objects p_TuCons and p_TuKcons are designed to assist the user in performing the relative complex task of verifying various consistency properties on a proposed game solution in parallel.[11] Its serial counterparts are called TuCons and TuKcons instead. Both class objects are subclass objects of the class object p_TuSol from which they inherit all properties and methods besides its own. However, the class objects TuCons and TuKcons are subclass objects of the class object TuSol.

From the Figure that represents the class object hierarchy structure of these objects, we can immediately recognize that the class objects p_TuCons and p_TuKcons resides on the same hierarchy level. No data can be exchanged between these objects. The same argument holds true for the class objects TuCons and TuKcons.

Figure 13: Class Object p_TuCons and p_TuKcons Hierarchy



Verifying consistency properties by a procedural approach requires to set several parameter in the correct order. Moreover, many output arguments will be returned that overflow the Matlab workspace with so many variable names that the overview is quickly lost. After performing some operations it might not be anymore clear which variable belongs to which parameter constellation. Especially, deriving consistency properties on proposed solutions for games of size 10 and larger produce very quickly several dozens variable names. In order to assign a consistency check on a proposed solution to the correct parameter constellation and game, we designed these class objects which should help the user to concentrate on the

[11] For more details see Subsections 2.12 and 3.2.

more game theoretical aspects of an analysis instead of losing his time with an unproductive sequencing of results.

Consistency checks for games with 10-persons or more are very resource demanding and can last over several hours or even days. To speed up the computational process we strongly recommend to perform such an analysis in parallel. In our example, which is relatively small, we launch four Matlab workers.

```
>> matlabpool open MyProfile1
Starting matlabpool using the 'MyProfile1' profile ... connected to 4 workers.
```

Class Object p_TuCons

In the next step, we create a class object *pclv_con* with the the method *p_TuCons()* to be able in the sequel to verify the reduced game, reconfirmation, and converse reduced game property of a solution. In general, all operations which can be invoked are very intense, so all default values are empty except those which indicate the game.

```
>> pclv_con=p_TuCons(v,'cr')

pclv_con =

 p_TuCons with properties:

       tu_x: []
    tu_CRGP_PRN: []
   tu_CRGPC_PRN: []
    tu_CRGP_PRK: []
   tu_CRGPC_PRK: []
   tu_CRGP_SHAP: []
  tu_CRGPC_SHAP: []
    tu_RCP_PRN: []
   tu_RCPC_PRN: []
    tu_RCP_PRK: []
   tu_RCPC_PRK: []
   tu_RCP_SHAP: []
  tu_RCPC_SHAP: []
  tu_RCP_HMS_PN: []
 tu_RCPC_HMS_PN: []
  tu_RCP_HMS_PK: []
 tu_RCPC_HMS_PK: []
    tu_RGP_PRN: []
   tu_RGPC_PRN: []
    tu_RGP_PRK: []
   tu_RGPC_PRK: []
   tu_RGP_SHAP: []
  tu_RGPC_SHAP: []
  tu_RGP_HMS_PN: []
 tu_RGPC_HMS_PN: []
  tu_RGP_HMS_PK: []
 tu_RGPC_HMS_PK: []
     tu_prk: []
    tu_prk2: []
     tu_prn: []
      tu_sh: []
    tu_tauv: []
     tu_bzf: []
    tu_aprk: []
   prk_valid: 0
   prn_valid: 0
  prk2_valid: 0
  aprk_valid: 0
```

```
 tuvalues: [1x63 double]
  tusize: 63
tuplayers: 6
  tutype: 'cr'
  tuessQ: 1
 tuformat: 'mattug'
    tumv: 24
   tumnQ: 0
    tuSi: [62 61 59 55 47 31]
    tuvi: [0 0 0 0 0 0]
  tustpt: []
```

We present a short description of the properties used in the class object p_TuCons. We assume that the user is familiar with the definitions of the various consistency properties mentioned here.

- *tu_x* stores the payoff vector to be checked on consistency.
- *tu_CRGP_PRN* stores the information whether *tu_x* satisfies CRGP w.r.t. DM-reduced game in accordance with the pre-nucleolus.
- *tu_CRGPC_PRN* stores the information w.r.t. the DM-reduced games restricted to *tu_x*.
- *tu_CRGP_PRK* stores the information whether *tu_x* satisfies CRGP w.r.t. DM-reduced game in accordance with the pre-kernel.
- *tu_CRGPC_PRK* stores the information w.r.t. the DM-reduced games restricted to *tu_x*.
- *tu_CRGP_SHAP* stores the information whether *tu_x* satisfies CRGP w.r.t. HMS-reduced game in accordance with the Shapley value.
- *tu_CRGPC_SHAP* stores the information w.r.t. the HMS-reduced games restricted to *tu_x*.
- *tu_RCP_PRN* stores the information whether *tu_x* satisfies RCP w.r.t. DM-reduced game in accordance with the pre-nucleolus.
- *tu_RCPC_PRN* stores the information w.r.t. the DM-reduced games restricted to *tu_x*.
- *tu_RCP_PRK* stores the information whether *tu_x* satisfies RCP w.r.t. DM-reduced game in accordance with the pre-kernel.
- *tu_RCPC_PRK* stores the information w.r.t. the DM-reduced games restricted to *tu_x*.
- *tu_RCP_SHAP* stores the information whether *tu_x* satisfies RCP w.r.t. HMS-reduced game in accordance with the Shapley value.
- *tu_RCPC_SHAP* stores the information w.r.t. the HMS-reduced games restricted to *tu_x*.
- *tu_RCP_HMS_PN* stores the information whether *tu_x* satisfies RCP w.r.t. HMS-reduced game in accordance with the pre-nucleolus.
- *tu_RCPC_HMS_PN* stores the information w.r.t. the HMS-reduced games restricted to *tu_x*.
- *tu_RCP_HMS_PK* stores the information whether*tu_x* satisfies RCP w.r.t. HMS-reduced game in accordance with the pre-kernel.
- *tu_RCPC_HMS_PK* stores the information w.r.t. the HMS-reduced games restricted to *tu_x*.
- *tu_RGP_PRN* stores the information whether *tu_x* satisfies RGP w.r.t. DM-reduced game in accordance with the pre-nucleolus.
- *tu_RGPC_PRN* stores the information w.r.t. the DM-reduced games restricted to *tu_x*.
- *tu_RGP_PRK* stores the information whether *tu_x* satisfies RGP w.r.t. DM-reduced game in accordance with the pre-kernel.
- *tu_RGPC_PRK* stores the information w.r.t. the DM-reduced games restricted to *tu_x*.

- *tu_RGP_SHAP* stores the information whether *tu_x* satisfies RGP w.r.t. HMS-reduced game in accordance with the Shapley value
- *tu_RGPC_SHAP* stores the information w.r.t. the HMS-reduced games restricted to *tu_x*.
- *tu_RGP_HMS_PN* stores the information whether *tu_x* satisfies RGP w.r.t. HMS-reduced game in accordance with the pre-nucleolus.
- *tu_RGPC_HMS_PN* stores the information w.r.t. the HMS-reduced games restricted to *tu_x*.
- *tu_RGP_HMS_PK* stores the information whether *tu_x* satisfies RGP w.r.t. HMS-reduced game in accordance with the pre-kernel.
- *tu_RGPC_HMS_PK* stores the information w.r.t. the HMS-reduced games restricted to *tu_x*.

We provide here also a first description of the additional methods supplied by the class object p_TuCons. These are the following:

- *p_TuCons* creates the class object p_TuCons;
- *p_setConverse_RGP* sets results of CRGP to the class object p_TuCons;
- *p_setReconfirmation_property* sets results of RCP to the class object p_TuCons;
- *p_setReduced_game_property* sets results of RGP to the class object p_TuCons;
- *p_copyTuSol* copies all solutions from TuSol/p_TuSol to p_TuCons;
- *p_copyTuCons* copies consistency properties from p_TuCons to TuCons.

By the above methods list, we observe that it is allowed to copy data from a class object TuSol/p_TuSol to a subclass object p_TuCons. Since, the solutions are stored in the class object *clv_sol* we use these data to copy it on the almost empty class object *pclv_con*. This is done through

```
>> pclv_con = p_copyTuSol(pclv_con,clv_sol);

pclv_con =

 p_TuCons with properties:

       tu_x: []
   tu_CRGP_PRN: []
  tu_CRGPC_PRN: []
   tu_CRGP_PRK: []
  tu_CRGPC_PRK: []
  tu_CRGP_SHAP: []
 tu_CRGPC_SHAP: []
   tu_RCP_PRN: []
  tu_RCPC_PRN: []
   tu_RCP_PRK: []
  tu_RCPC_PRK: []
  tu_RCP_SHAP: []
 tu_RCPC_SHAP: []
 tu_RCP_HMS_PN: []
tu_RCPC_HMS_PN: []
 tu_RCP_HMS_PK: []
tu_RCPC_HMS_PK: []
   tu_RGP_PRN: []
  tu_RGPC_PRN: []
   tu_RGP_PRK: []
  tu_RGPC_PRK: []
  tu_RGP_SHAP: []
 tu_RGPC_SHAP: []
 tu_RGP_HMS_PN: []
tu_RGPC_HMS_PN: []
 tu_RGP_HMS_PK: []
```

```
  tu_RGPC_HMS_PK: []
       tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
      tu_prk2: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
       tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
        tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
      tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
       tu_bzf: []
      tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
     prk_valid: 1
     prn_valid: 1
    prk2_valid: 1
    aprk_valid: 1
     tuvalues: [1x63 double]
       tusize: 63
     tuplayers: 6
       tutype: 'cr'
       tuessQ: 1
     tuformat: 'mattug'
         tumv: 24
        tumnQ: 0
        tuSi: [62 61 59 55 47 31]
        tuvi: [0 0 0 0 0 0]
      tustpt: []
```

Now, we can apply the above methods to study whether the computed pre-kernel element satisfies some well established consistency properties. For doing so, it is just enough to supply the methods with two input arguments, the class object *pclv_con* and the solution that should be investigate, that is in this case, the pre-kernel element. This solution is stored in the property *pclv_con.tu_prk*. After the operations are finished they will be written to the class object *pclv_con*. Hence, this operation creates encapsulated data through an instance to itself.

```
>> tic; pclv_con = p_setConverse_RGP(pclv_con,pclv_con.tu_prk);pcrgp_pkt=toc;
>> tic; pclv_con = p_setReconfirmation_property(pclv_con,pclv_con.tu_prk); prcp_pkt=toc;
>> tic; pclv_con = p_setReduced_game_property(pclv_con,pclv_con.tu_prk); prgpp_pkt=toc;
```

When inspecting the elapsed computation time, we realize that these operations are accomplished very quickly. Only a few seconds are needed to complete the whole consistency check on the proposed solution.

```
>> pcrgp_pkt

pcrgp_pkt =

  0.4936

>> prcp_pkt

prcp_pkt =

  6.9465

>> prgpp_pkt

prgpp_pkt =

  4.4026
```

And here the data which are written to the class object *pclv_con* and which should be checked with care while retrieving the results.

```
pclv_con =

 p_TuCons with properties:

       tu_x: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
    tu_CRGP_PRN: [1x1 struct]
   tu_CRGPC_PRN: {'vS' {1x15 cell} 'sV_x' {1x15 cell}}
    tu_CRGP_PRK: [1x1 struct]
   tu_CRGPC_PRK: {'vS' {1x15 cell} 'sV_x' {1x15 cell}}
   tu_CRGP_SHAP: [1x1 struct]
  tu_CRGPC_SHAP: {'vS' {1x15 cell} 'sV_x' {1x15 cell}}
    tu_RCP_PRN: [1x1 struct]
   tu_RCPC_PRN: {'vS' {1x62 cell} 'vS_sol' {1x63 cell} 'vS_y' {1x63 cell}}
    tu_RCP_PRK: [1x1 struct]
   tu_RCPC_PRK: {'vS' {1x62 cell} 'vS_sol' {1x63 cell} 'vS_y' {1x63 cell}}
   tu_RCP_SHAP: [1x1 struct]
  tu_RCPC_SHAP: {'vS' {1x62 cell} 'vS_sol' {1x63 cell} 'vS_y' {1x63 cell}}
  tu_RCP_HMS_PN: [1x1 struct]
 tu_RCPC_HMS_PN: {'vS' {1x62 cell} 'vS_sol' {1x63 cell} 'vS_y' {1x63 cell}}
  tu_RCP_HMS_PK: [1x1 struct]
 tu_RCPC_HMS_PK: {'vS' {1x62 cell} 'vS_sol' {1x63 cell} 'vS_y' {1x63 cell}}
    tu_RGP_PRN: [1x1 struct]
   tu_RGPC_PRN: {'vS' {1x62 cell} 'impVec' {1x63 cell}}
    tu_RGP_PRK: [1x1 struct]
   tu_RGPC_PRK: {'vS' {1x62 cell} 'impVec' {1x63 cell}}
   tu_RGP_SHAP: [1x1 struct]
  tu_RGPC_SHAP: {'vS' {1x62 cell} 'impVec' {1x63 cell}}
  tu_RGP_HMS_PN: [1x1 struct]
 tu_RGPC_HMS_PN: {'vS' {1x62 cell} 'impVec' {1x63 cell}}
  tu_RGP_HMS_PK: [1x1 struct]
 tu_RGPC_HMS_PK: {'vS' {1x62 cell} 'impVec' {1x63 cell}}
      tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
     tu_prk2: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
      tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
       tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
     tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
      tu_bzf: []
     tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
    prk_valid: 1
    prn_valid: 1
   prk2_valid: 1
   aprk_valid: 1
     tuvalues: [1x63 double]
      tusize: 63
    tuplayers: 6
      tutype: 'cr'
      tuessQ: 1
    tuformat: 'mattug'
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

A note on warning must be expressed at this stage. Notice, that the operations performed relied on floating point arithmetic, a wrong result which does not coincide with the theoretical expectations might be originated through numerical errors. Such judgment can only be made by an experienced user. Therefore, be careful with the interpretation of the result. Each result must be investigated on its soundness and

correctness.

Class Object p_TuKcons

After studying various consistency properties of a game, we are also able to study some generalization of the aforementioned consistency properties, which are subsumed under the term the k-analogon consistency properties. Here k is an integer between 2 and n. This means, that the consistency properties are parametrized by a parameter k. In this paragraph, we establish how one can apply a parametrized consistency investigation on a proposed solution while relying on the class object p_TuKcons.

Again, we have to create a class object. In this case a class object p_TuKcons, which we call *pclv_kc*.

```
>> pclv_kc = p_TuKcons(v,'cr');
```

In a further step, we copy the solutions stored in the class object *clv_sol* to *pclv_kc*. This done, with the method *p_copyTuSol()*.

```
>> pclv_kc = p_copyTuSol(pclv_kc,clv_sol)

pclv_kc =

 p_TuKcons with properties:

        tu_x: []
        tu_K: 2
   tu_kCRGP_PRN: []
   tu_kCRGP_PRK: []
  tu_kCRGP_SHAP: []
    tu_kRCP_PRN: []
    tu_kRCP_PRK: []
   tu_kRCP_SHAP: []
  tu_kRCP_HMS_PN: []
  tu_kRCP_HMS_PK: []
   tu_kSCRGP_PRN: []
   tu_kSCRGP_PRK: []
  tu_kSCRGP_SHAP: []
    tu_kRGP_PRN: []
    tu_kRGP_PRK: []
   tu_kRGP_SHAP: []
  tu_kRGP_HMS_PN: []
  tu_kRGP_HMS_PK: []
      tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
     tu_prk2: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
      tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
       tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
     tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
      tu_bzf: []
     tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
    prk_valid: 1
    prn_valid: 1
   prk2_valid: 1
   aprk_valid: 1
    tuvalues: [1x63 double]
     tusize: 63
    tuplayers: 6
      tutype: 'cr'
      tuessQ: 1
    tuformat: 'mattug'
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

Here we provide a short description of the new properties supplied by p_TuKcons. Again, we assume that the user is familiar with the definitions of the various consistency properties mentioned here.

- *tu_x* stores the payoff vector to be checked on consistency.
- *tu_K* a positive integer between 2 and n.
- *tu_kCRGP_PRN* stores the information whether *tu_x* satisfies K-CRGP w.r.t. DM-reduced game in accordance with the pre-nucleolus.
- *tu_kCRGP_PRK* stores the information whether *tu_x* satisfies K-CRGP w.r.t. DM-reduced game in accordance with the pre-kernel.
- *tu_kCRGP_SHAP* stores the information whether *tu_x* satisfies K-CRGP w.r.t. HMS-reduced game in accordance with the Shapley value.
- *tu_kRCP_PRN* stores the information whether *tu_x* satisfies K-RCP w.r.t. DM-reduced game in accordance with the pre-nucleolus.
- *tu_kRCP_PRK* stores the information whether *tu_x* satisfies K-RCP w.r.t. DM-reduced game in accordance with the pre-kernel.
- *tu_kRCP_SHAP* stores the information whether *tu_x* satisfies K-RCP w.r.t. HMS-reduced game in accordance with the Shapley value.
- *tu_kRCP_HMS_PN* stores the information whether *tu_x* satisfies K-RCP w.r.t. HMS-reduced game in accordance with the pre-nucleolus.
- *tu_kRCP_HMS_PK* stores the information whether *tu_x* satisfies K-RCP w.r.t. HMS-reduced game in accordance with the pre-kernel.
- *tu_kSCRGP_PRN* stores the information whether *tu_x* satisfies K-SCRGP w.r.t. DM-reduced game in accordance with the pre-nucleolus.
- *tu_kSCRGP_PRK* stores the information whether *tu_x* satisfies K-SCRGP w.r.t. DM-reduced game in accordance with the pre-kernel.
- *tu_kSCRGP_SHAP* stores the information whether *tu_x* satisfies K-SCRGP w.r.t. HMS-reduced game in accordance with the Shapley value
- *tu_kRGP_PRN* stores the information whether *tu_x* satisfies K-RGP w.r.t. DM-reduced game in accordance with the pre-nucleolus.
- *tu_kRGP_PRK* stores the information whether *tu_x* satisfies K-RGP w.r.t. DM-reduced game in accordance with the pre-kernel.
- *tu_kRGP_SHAP* stores the information whether *tu_x* satisfies K-RGP w.r.t. HMS-reduced game in accordance with the Shapley value.
- *tu_kRGP_HMS_PN* stores the information whether *tu_x* satisfies K-RGP w.r.t. HMS-reduced game in accordance with the pre-nucleolus.
- *tu_kRGP_HMS_PN* stores the information whether *tu_x* satisfies K-RGP w.r.t. HMS-reduced game in accordance with the pre-kernel.

In order to conduct a k-parametrized consistency analysis the class object p_TuKcons provides the following additional methods:

- *p_TuKcons* creates the class object p_TuKcons.
- *p_setKConverse_RGP* sets results of k-CRGP to the class object p_TuKcons.
- *p_setKReconfirmation_property* sets results of k-RCP to the class object p_TukCons.

- *p_setKStrConverse_RGP* sets results of k-SCRGP to the class object p_TuKcons.
- *p_setKReducedGameProperty* sets results of k-RGP to the class object p_TuKcons.
- *p_copyTuSol* copies all solutions from TuSol/p_TuSol to p_TuKcons.
- *p_copyTuKcons* copies k-consistency properties from p_TuKcons to TuKcons.

To parameterize the consistency analysis on the proposed game solution, the variable k has to run from 2 up to n, that is, for a 6-person game up to 6. But this implies that we need for each parameter k a class object p_TuKcons. However, this is done quite easily by copying the class object *pclv_kc* five times.

```
>> for k=1:n-1,
    pclv_kc_pk(k) = copy(pclv_kc);
  end;
```

Now, we are in the position to conduct a generalized consistency analysis. This task is accomplished by calling the above methods and run an index k from 1 to 5. This is shown through the following sequence of methods.

```
>> tic; for k=1:n-1,
  pclv_kc_pk(k) = p_setKConverse_RGP(pclv_kc_pk(k),pclv_kc_pk(k).tu_prk,k+1);
  end;kcrgp_pkt=toc;
>> tic; for k=1:n-1,
  pclv_kc_pk(k) = p_setKReconfirmation_property(pclv_kc_pk(k),pclv_kc_pk(k).tu_prk,k+1);
  end;krcp_pkt=toc;
>> tic; for k=1:n-1,
  pclv_kc_pk(k) = p_setKStrConverse_RGP(pclv_kc_pk(k),pclv_kc_pk(k).tu_prk,k+1);
  end;kscrgp_pkt=toc;
  tic; for k=1:n-1,
  pclv_kc_pk(k) = p_setKReducedGameProperty(pclv_kc_pk(k),pclv_kc_pk(k).tu_prk,k+1);
  end;ksrgp_pkt=toc;
```

Exemplarily, we enlist the results of the generalized consistency check w.r.t. the proposed pre-kernel solution where we have set the parameter value of k to 4, but this means that the third class object *pclv_kc_pk* must be retrieved from the workspace.

```
>> pclv_kc_pk(3)

ans =

 p_TuKcons with properties:

        tu_x: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
        tu_K: 4
   tu_kCRGP_PRN: [1x1 struct]
   tu_kCRGP_PRK: [1x1 struct]
  tu_kCRGP_SHAP: [1x1 struct]
    tu_kRCP_PRN: [1x1 struct]
    tu_kRCP_PRK: [1x1 struct]
   tu_kRCP_SHAP: [1x1 struct]
  tu_kRCP_HMS_PN: [1x1 struct]
  tu_kRCP_HMS_PK: [1x1 struct]
   tu_kSCRGP_PRN: [3x1 struct]
   tu_kSCRGP_PRK: [3x1 struct]
  tu_kSCRGP_SHAP: [3x1 struct]
    tu_kRGP_PRN: [1x1 struct]
    tu_kRGP_PRK: [1x1 struct]
   tu_kRGP_SHAP: [1x1 struct]
  tu_kRGP_HMS_PN: [1x1 struct]
  tu_kRGP_HMS_PK: [1x1 struct]
```

```
      tu_prk: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
     tu_prk2: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
      tu_prn: [5.0000 3.0000 4.0000 3.0000 5.0000 4.0000]
       tu_sh: [4.1167 3.5167 4.3667 3.8333 4.5833 3.5833]
     tu_tauv: [4 3.5000 4.5000 4 4.5000 3.5000]
      tu_bzf: []
     tu_aprk: [4 3.5000 4.5000 4 4.5000 3.5000]
   prk_valid: 1
   prn_valid: 1
  prk2_valid: 1
  aprk_valid: 1
    tuvalues: [1x63 double]
      tusize: 63
   tuplayers: 6
      tutype: 'cr'
      tuessQ: 1
    tuformat: 'mattug'
       tumv: 24
      tumnQ: 0
       tuSi: [62 61 59 55 47 31]
       tuvi: [0 0 0 0 0 0]
     tustpt: []
```

Now, the results obtained from the generalized consistency investigation must be checked out. For doing this, consult the definitions of the k-analogon consistency properties from the literature, if you are not familiar with these consistency concepts.

Even though for each generalized consistency property 5 calls must be invoked while launching four Matlab workers, the generalized consistency analysis requires only a few seconds more than a standard consistency check on a proposed solution, as we can observe from the enlisted timings.

```
>> kcrgp_pkt

kcrgp_pkt =

   2.5923

>> krcp_pkt

krcp_pkt =

   9.9946

>> kscrgp_pkt

kscrgp_pkt =

   4.2603

>> ksrgp_pkt

ksrgp_pkt =

   4.9675
```

To finish our analysis, we shutdown the parallel computation with

```
>> matlabpool close
Sending a stop signal to all the workers ... stopped.
```

To summarize the whole discussion of this section, we provide the user with a template file to automate the whole process of a consistency analysis on various proposed solutions. Such a batch file can be submitted to a compute node, which should be the preferred procedure whenever the game is large, that is, for games of size 10 and larger.

```
matlabpool open local
n=6;
sl=3;
rand('state',135);
pfm = magic(sl);
slv = 1:sl;
v=assignment_game(slv,pfm);
clv = TuSol(v,'cr');
clv = setAllSolutions(clv);
tic;for k=1:150, pyrd(k,:) = randi([0,v(end)],1,n);prkm(k,:)=PreKernel(clv,pyrd(k,:));end;prkmt=toc;
prkM = unique(prkm,'rows');
pclv = p_TuCons(v,'cr');
pclv = p_copyTuSol(pclv,clv);
pclv_pk = copy(pclv);
pclv_pn = copy(pclv);
pclv_pk2 = copy(pclv);
pclv_sh = copy(pclv);
save('opp_assignment_cons.mat','-v7.3');
tic; pclv_pk = p_setConverse_RGP(pclv_pk,pclv_pk.tu_prk);pcrgp_pkt=toc;
tic; pclv_pk = p_setReconfirmation_property(pclv_pk,pclv_pk.tu_prk); prcp_pkt=toc;
tic; pclv_pk = p_setReduced_game_property(pclv_pk,pclv_pk.tu_prk); prgpp_pkt=toc;
save('opp_assignment_cons.mat','-append','-v7.3');
tic; pclv_pn = p_setConverse_RGP(pclv_pn,pclv_pn.tu_prn);pcrgp_pnt=toc;
tic; pclv_pn = p_setReconfirmation_property(pclv_pn,pclv_pn.tu_prn); prcp_pnt=toc;
tic; pclv_pn = p_setReduced_game_property(pclv_pn,pclv_pn.tu_prn); prgpp_pnt=toc;
save('opp_assignment_cons.mat','-append','-v7.3');
tic; pclv_pk2 = p_setConverse_RGP(pclv_pk2,pclv_pk2.tu_prk2);pcrgp_pk2t=toc;
tic; pclv_pk2 = p_setReconfirmation_property(pclv_pk2,pclv_pk2.tu_prk2); prcp_pk2t=toc;
tic; pclv_pk2 = p_setReduced_game_property(pclv_pk2,pclv_pk2.tu_prk2);prgpp_pk2t=toc;
save('opp_assignment_cons.mat','-append','-v7.3');
tic; pclv_sh = p_setConverse_RGP(pclv_sh,pclv_sh.tu_sh);pcrgp_sht=toc;
tic; pclv_sh = p_setReconfirmation_property(pclv_sh,pclv_sh.tu_sh); prcp_sht=toc;
tic; pclv_sh = p_setReduced_game_property(pclv_sh,pclv_sh.tu_sh);prgpp_sht=toc;
save('opp_assignment_cons.mat','-append','-v7.3');
pclv_kc = p_TuKcons(v,'cr');
pclv_kc = p_copyTuSol(pclv_kc,clv);
for k=1:n-1, pclv_kc_pk(k) = copy(pclv_kc);end;;
for k=1:n-1, pclv_kc_pn(k) = copy(pclv_kc);end;
for k=1:n-1, pclv_kc_pk2(k) = copy(pclv_kc);end;
for k=1:n-1, pclv_kc_sh(k) = copy(pclv_kc);end;
save('opp_assignment_cons.mat','-append','-v7.3');
tic; for k=1:n-1,
  pclv_kc_pk(k) = p_setKConverse_RGP(pclv_kc_pk(k),pclv_kc_pk(k).tu_prk,k+1);
end;kcrgp_pkt=toc;
tic; for k=1:n-1,
  pclv_kc_pk(k) = p_setKReconfirmation_property(pclv_kc_pk(k),pclv_kc_pk(k).tu_prk,k+1);
end;krcp_pkt=toc;
tic; for k=1:n-1,
  pclv_kc_pk(k) = p_setKStrConverse_RGP(pclv_kc_pk(k),pclv_kc_pk(k).tu_prk,k+1);
end;kscrgp_pkt=toc;
tic; for k=1:n-1,
  pclv_kc_pk(k) = p_setKReducedGameProperty(pclv_kc_pk(k),pclv_kc_pk(k).tu_prk,k+1);
end;ksrgp_pkt=toc
save('opp_assignment_cons.mat','-append','-v7.3');
tic; for k=1:n-1,
  pclv_kc_pn(k) = p_setKConverse_RGP(pclv_kc_pn(k),pclv_kc_pn(k).tu_prn,k+1);
end;kcrgp_pnt=toc;
tic; for k=1:n-1,
  pclv_kc_pn(k) = p_setKReconfirmation_property(pclv_kc_pn(k),pclv_kc_pn(k).tu_prn,k+1);
```

```
end;krcp_pnt=toc;
tic; for k=1:n-1,
  pclv_kc_pn(k) = p_setKStrConverse_RGP(pclv_kc_pn(k),pclv_kc_pn(k).tu_prn,k+1);
end;kscrgp_pnt=toc;
tic; for k=1:n-1,
  pclv_kc_pn(k) = p_setKReducedGameProperty(pclv_kc_pn(k),pclv_kc_pn(k).tu_prn,k+1);
end;ksrgp_pnt=toc
save('opp_assignment_cons.mat','-append','-v7.3');
tic; for k=1:n-1,
  pclv_kc_pk2(k) = p_setKConverse_RGP(pclv_kc_pk2(k),pclv_kc_pk2(k).tu_prk2,k+1);
end;kcrgp_pk2t=toc;
tic; for k=1:n-1,
  pclv_kc_pk2(k) = p_setKReconfirmation_property(pclv_kc_pk2(k),pclv_kc_pk2(k).tu_prk2,k+1);
end;krcp_pk2t=toc;
tic; for k=1:n-1,
  pclv_kc_pk2(k) = p_setKStrConverse_RGP(pclv_kc_pk2(k),pclv_kc_pk2(k).tu_prk2,k+1);
end;kscrgp_pk2t=toc;
tic; for k=1:n-1,
  pclv_kc_pk2(k) = p_setKReducedGameProperty(pclv_kc_pk2(k),pclv_kc_pk2(k).tu_prk2,k+1);
end;ksrgp_pk2t=toc
save('opp_assignment_cons.mat','-append','-v7.3');
tic; for k=1:n-1,
  pclv_kc_sh(k) = p_setKConverse_RGP(pclv_kc_sh(k),pclv_kc_sh(k).tu_sh,k+1);
end;kcrgp_sht=toc;
tic; for k=1:n-1,
  pclv_kc_sh(k) = p_setKReconfirmation_property(pclv_kc_sh(k),pclv_kc_sh(k).tu_sh,k+1);
end;krcp_sht=toc;
tic; for k=1:n-1,
  pclv_kc_sh(k) = p_setKStrConverse_RGP(pclv_kc_sh(k),pclv_kc_sh(k).tu_sh,k+1);
end;kscrgp_sht=toc;
tic; for k=1:n-1,
  pclv_kc_sh(k) = p_setKReducedGameProperty(pclv_kc_sh(k),pclv_kc_sh(k).tu_sh,k+1);
end;ksrgp_sht=toc
save('opp_assignment_cons.mat','-append','-v7.3');

clear;
matlabpool close;
exit;
```

After the computation has been completed one will find a mat-file of about 112 MB on the hard disk, that must be loaded in the Matlab workspace to check out the computed results.

5. Calling the Mathematica Package TuGames

The toolbox *MatTuGames* provides also an interface to our *MATHEMATICA*® package *TuGames* when the *MATHEMATICA*® Symbolic Toolbox is properly installed on your system[12]. Then it is possible to call about 90 functions of the Mathematica package *TuGames* within a running Matlab session. Of course, the functionality of these functions is restricted, because most of the options as well as the parallel mode are not available from a running Matlab session, nevertheless, it allows some fundamental access to the *TuGames* package.

The reason for calling the *TuGames* package could be versatile. In the first place is certainly to mention the opportunity to get some additional evidence about the correctness of a result. All basic functions under Matlab have their *MATHEMATICA*® counterparts. Moreover, their code basis differ from *MATHEMATICA*®, hence the functions are differently implemented in Matlab and *MATHEMATICA*®. Apart from any differences between Matlab and *MATHEMATICA*®, their computational process will differ. This imposes a high reliability

[12]See http://www.mathworks.com/matlabcentral/fileexchange/6044-mathematica-symbolic-toolbox-for-matlab-version-2-0.

of the performed computation when both results are equal. Second in place is to mention the possibility to analyze a game with some additional options and intermediate results, which are not provided by the toolbox. Tertiary is to mention the access to some additional functions, which extends the functionality of our Matlab toolbox. A disadvantage might be that these functions are executed very slowly. For instance, to calculate a pre-kernel element of a 12-person game can last at least 25 minutes in a MATHEMATICA® session, in contrast to a computation time of approximately tenth part of a second in Matlab.

In this section, we want to discuss how to call functions of our *TuGames* package within a Matlab session. For this purpose, we discuss some basic functions whose functionality we already acquaint with Matlab, but where the handling of the results is different due to some MATHEMATICA® specifics.

5.1. STARTING THE MATHKERNEL FROM MATLAB

Now, let us treat a first example. In this example, we want check again the convexity property of the game v. But before we can do that, some words about the operating mode are inevitable. In order to call the MATHEMATICA® function *ConvexQ[]*, the MATHEMATICA® Kernel must be loaded. By doing so, a running MATHEMATICA® session will be left in order to avoid any conflicts from an already initialized MATHEMATICA® Kernel. In the next step the MATHEMATICA® Kernel is restarted. When the MATHEMATICA® Kernel is successfully initialized, the package *TuGames* must be loaded before the game can be passed to the MATHEMATICA® Kernel, and the function *ConvexQ[]* can be executed. Finally, the result computed by MATHEMATICA® will be passed back to Matlab. In the example below, the value *True* instead of 1 will be returned as a string value to Matlab, and the MATHEMATICA® Kernel will be quitted again.

```
>> CVQ_v=tug_ConvexQ(v)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)

=====================================================

Loading Package 'TuGames' for Unix

=====================================================

TuGames V2.2 by Holger I. Meinhardt

Release Date: 08.06.2013

Program runs under Mathematica Version 8.0 or later

Version 8.x or higher is recommended

=====================================================

=====================================================

Package 'TuGames' loaded

=====================================================

Passing Game to Mathematica ...
Mathematica Kernel quitting per your request...

CVQ_v =
```

```
  ConvexQ: 'True'
```

In this case the *MATHEMATICA*® specific result will be returned. The result indicates that the game is convex, thus, our previous result was confirmed. Notice that this result cannot be used as an intermediate result under Matlab. For this purpose, this result must be transcribed into a logical value. In general, the results will be returned in Matlab as well as in *MATHEMATICA*® format, whereas a result in *MATHEMATICA*® format starts with the capital letter M. Finally, note that all functions that call a function or some functions from *MATHEMATICA*® starts with the prefix *tug_*.

Since, we confirmed the convexity property with *MATHEMATICA*®, we expect that also the average-convexity property should also be reproducible under *MATHEMATICA*® while issuing the function *AvConvexQ[]*. And indeed, issuing the function *tug_AvConvexQ()* within our Matlab session, produces:

```
>> AvCQ_v=tug_AvConvexQ(v)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)


...
...

Passing Game to Mathematica ...
Mathematica Kernel quitting per your request...

AvCQ_v =

  AverageConvexQ: 'True'
```

5.2. GENERATING TU GAMES WITH MATHEMATICA

Moreover, we can derive some additional game types, which are not available under the toolbox *MatTuGames*. For instance, we can derive several types of symmetric games that induces a kind of balanced collection. Exemplary, we take a coalition of size not less than 2, that is,

```
>> dec2bin(22)

ans =

10110
```

Now, it is possible to derive at least a five-person game by supplying the number of players involved $n = 5$, the coalition identifier $S = 22$ which has a size not less than 2, and finally, the worth $val = 4$ that should be assigned to a set of coalitions that forms a 3-balanced collection. The corresponding symmetric game can be derived with the function *tug_SymGameType2()*, more precisely,

```
>> SYMG=tug_SymGameType2(5,22,4)

Mathematica Kernel loading...
```

```
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)


Passing Data to Mathematica ...
Determing Symmetric Game Type 2 ...
Mathematica Kernel quitting per your request...

SYMG =

   SymGame: [0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 4 0 4 0 0 0 0 0 0 0 0 0 8]
  MSymGame: '{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 4, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

>  0, 0, 0, 0, 0, 8}'
    MGame: [0 0 0 0 0 0 0 0 0 4 0 4 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8]
```

The results are structure elements. In order to retrieve the structure element that is usable under Matlab, one can execute

```
>> v1=SYMG.SymGame

v1 =

 Columns 1 through 23

   0   0   0   0   0   4   0   0   0   0   0   0   0   0   0   0   0   4   0   4   0   0   0

 Columns 24 through 31

   0   0   0   0   0   0   0   8
```

to assign the game to the variable *v1* for instance. To get an idea, which collection of coalitions of size 2 is getting a positive worth, one can issue the following command:

```
>> c2=find(v1)

c2 =

   6   18   20   31
```

For a better understanding which coalitions are singled out, we convert the result into a binary format. This will be done with the Matlab function *dec2bin()*. Recall that one has to read from right to left.

```
>> dec2bin(c2)

ans =

00110
10010
10100
11111
```

Obviously, the set of proper coalitions is 3-balanced.

In addition, one can derive different symmetric game types with the functions *tug_SymGameType3()* or *tug_SymGameType4()*. The former requires as an input argument a coalition of size not less than 3, whereas the latter requires as an argument a coalition of size not less than 4.

With the function *tug_SymGameSizeK()* a different type of a symmetric game can be derived. In this case, for a cycle of coalitions having size k, the same worth val will be attributed, whereas the grand coalition gets $n/2 \cdot val$. Not surprising, the set of proper coalitions attributed with the same worth forms a balanced collection. For the example from below, we have chosen a six-person game selecting a cycle of coalitions of size 3 getting a worth of 2.

```
>> SYMG2=tug_SymGameSizeK(6,3,2)
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)

Passing Data to Mathematica ...
Determing Symmetric Game with size k ...
Mathematica Kernel quitting per your request...

SYMG2 =

   SymGame: [1x63 double]
  MSymGame: [1x206 char]
    MGame: [1x63 double]
```

Similar to the example from above, the structure element that is usable under Matlab is assigned to the variable $v1$.

```
>> v1=SYMG2.SymGame

v1 =

 Columns 1 through 23

   0   0   0   0   0   0   2   0   0   0   0   0   0   2   0   0   0   0   0   0   0   0   0

 Columns 24 through 46

   0   0   0   0   2   0   0   0   0   0   0   2   0   0   0   0   0   0   0   0   0   0   0

 Columns 47 through 63

   0   0   2   0   0   0   0   0   0   2   0   0   0   0   0   0   6
```

Now let us single out all coalitions which get a positive value. This reveals also the cycle of coalitions involved in this game. First, we execute the Matlab built-in-function *find()*.

```
>> c3=find(v1)

ans =

   7  14  28  35  49  56  63
```

Again, a list of coalition identifiers is returned, giving us no real clue which coalitions are selected. Thus, we have to convert the result into a binary format. Executing the Matlab function *dec2bin()*, getting:

```
>> dec2bin(c3)
```

```
ans =

000111
001110
011100
100011
110001
111000
111111
```

To observe that the command *sortsets()* can also sort subsets of a power set, let us sort now the above collection of coalitions. By doing so, getting:

```
>> [co3,bc3]=sortsets(c3,6)

co3 =

   7   35   49   14   28   56   63


bc3 =

000111
100011
110001
001110
011100
111000
111111
```

or to put it differently in an unsorted order {{5, 6, 1}, {6, 1, 2}, {1, 2, 3}, {2, 3, 4}, {3, 4, 5}, {4, 5, 6}}, which is a balanced collection. We can verify this with the function *tug_CollectionBalancedQ()* by eliminating the grand coalition, thus, we have to issue first:

```
>> cS3=co3

cS3 =

   7   35   49   14   28   56   63

>> cS3(end)=[]

cS3 =

   7   35   49   14   28   56
```

to get rid from the grand coalition. Having the grand coalition eliminated from the list, we can now call the function *tug_CollectionBalancedQ()* with two arguments. The first argument is the collection of coalitions *cS3*, and the second argument is related to the number of person involved in the grand coalition; in this example, this value must be set to 6. And indeed, the balancedness of the collection *cS3* is confirmed with this function when calling:

```
>> CS=tug_CollectionBalancedQ(cS3,6)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =
```

```
9.0 for Linux x86 (64-bit) (November 20, 2012)


Is the collection balanced?...
Mathematica Kernel quitting per your request...

CS =

  CollectionBalancedQ: 'True'
```

5.3. STUDYING GAME PROPERTIES

For small games, that is, for games up to 6-person, it is relatively easy to construct an average-convex game. For doing so, one needs just to assign positive values to all unanimity coordinates of coalition size two. In the case of a four-person game, we have 6 such coalitions. Assigning positive values, issuing for instance:

```
>> ucd=[4 9 8 2 4 10];
```

Of course, this procedure can also be used to construct larger games, but then one cannot expect anymore that the derived game still satisfies the average-convexity property.

From these values, we can derive the remaining unanimity coordinates while calling the function *DetUCoord()* under *MATHEMATICA*®. Thus, we execute the function *tug_DetUCoord()* with the arguments *ucd*, and 4 to indicate that a 4-person game should be derived.

```
>> uc=tug_DetUCoord(ucd,4)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)


Passing unanimity coordninates to Mathematica ...
Mathematica Kernel quitting per your request...

uc =

   UnanimityCoordninates: [0 0 4 0 9 2 -2 0 8 4 -3 10 -7.5000 -2 2.3333]
  MUnanimityCoordninates: '{0, 0, 0, 0, 0, 4., 9., 8., 2., 4., 10., -2., -3., -7.5, -2., 2.33333}'
```

Again, the results are structure elements. Similar, to the symmetric game from above, to retrieve the structure element that is usable under Matlab, one has to execute:

```
>> uc.UnanimityCoordninates

ans =

 Columns 1 through 14

     0     0   4.0000     0   9.0000   2.0000  -2.0000     0   8.0000   4.0000  -3.0000  10.0000  -7.5000  -2.0000
```

```
 Column 15

  2.3333
```

and for the *MATHEMATICA*® output that is not usable under Matlab, issuing:

```
uc.MUnanimityCoordninates

ans =

{0, 0, 0, 0, 0, 4., 9., 8., 2., 4., 10., -2., -3., -7.5, -2., 2.33333}
```

From these unanimity coordinates, we are able to derive a four-person game while invoking the *getgame()* function under Matlab. Hence we need just to type:

```
>> v2=getgame(uc.UnanimityCoordninates)

v2 =

 Columns 1 through 14

     0     0   4.0000     0   9.0000   2.0000  13.0000     0   8.0000   4.0000  13.0000  10.0000  19.5000  14.0000

 Column 15

  24.8333
```

Now, we can perform some operations under Matlab to analyze the game. Again let us check the average-convexity. Indeed, this game is average-convex as can be seen next.

```
>> average_convexQ(v2)

ans =

   1
```

Verifying this result under *MATHEMATICA*®, calling:

```
>> AvCQ_v2=tug_AvConvexQ(v2)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)


Passing Game to Mathematica ...
Mathematica Kernel quitting per your request...

AvCQ_v2 =

  AverageConvexQ: 'True'
```

5.4. Computing Game Solutions and Analyzing its Properties

Apart from computing several game properties like convexity, average-convexity, superadditivity or monotonicity to mention only a few of them to verify a property of a game, the *MATHEMATICA*® package *TuGames* offers also a set of functions to compute the (pre-)kernel, (pre-)nucleolus, anti (pre-)kernel and Shapley value. Fairness and related values w.r.t. a coalition structure can not be computed with this package. To see how one can execute some of these functions within a running Matlab session, we start with the computation of the Shapley value. Similar to the above procedure, we execute the command *tug_ShapleyValue()* by

```
>> SOLsh=tug_ShapleyValue(v2)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)

Passing Game to Mathematica ...
Mathematica Kernel quitting per your request...

SOLsh =

   ShapleyValue: [6.9167 3.2500 7.2500 7.4167]
  MShapleyValue: ' 83  13  29  89
                 {--, --, --, --}
                  12  4   4   12'
```

This function requires just one input argument, that is, the game array v_2. We retrieve the structure element that stores the information of the Shapley value result in *MATHEMATICA*® format by

```
>> SOLsh.MShapleyValue

ans =

 83  13  29  89
{--, --, --, --}
 12  4   4   12
```

and to access the structure element that contains the data of the Shapley value in Matlab format we invoke

```
>> SOLsh.ShapleyValue

ans =

  6.9167   3.2500   7.2500   7.4167
```

Comparing this evaluation with the Shapley value computation under Matlab, we observe that indeed both results coincide

```
>> sh_v2=ShapleyValue(v2)

sh_v2 =

  6.9167   3.2500   7.2500   7.4167
```

```
>> sh_v2-SOLsh.ShapleyValue

ans =

   0   0   0   0
```

For the computation of a pre-kernel element under *MATHEMATICA*®, it is sufficient to execute the function *tug_PreKernel* with the game parameter v_2, that is, invoking

```
>> PRK_v2=tug_PreKernel(v2)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)


Passing Game to Mathematica ...
Computing the Pre-Kernel ...
Mathematica Kernel quitting per your request...

PRK_v2 =

    PreKernel: [6.7222 2.6667 7.7222 7.7222]
   MPreKernel: ' 121  8  139  139
               {---, -, ---, ---}
                18  3  18   18'
  MPreKernel_Q: '{True}'
```

It is also possible to call this function with a different starting point rather than its default vector. Even a payoff matrix can be assigned to the function *tug_PreKernel* to use a grid of distinct staring points.

The field variable *PRK_v2* contains the results of the computation in Matlab as well as in *MATHEMATICA*® format. The first structure element contains the computed pre-kernel element, transcribed in Matlab output. The second structure element contains the computed pre-kernel element in *MATHEMATICA*® format, which is indicated by the curly braces and the rational numbers. The last structure element contains the confirmation that we have really found a pre-kernel element. To access the contents of the first and second element, one has to invoke:

```
>> PRK_v2.PreKernel

ans =

  6.7222   2.6667   7.7222   7.7222


>> PRK_v2.MPreKernel

ans =

 121  8  139  139
{---, -, ---, ---}
 18  3  18  18
```

This result is confirmed with our Matlab function *PreKernel()* as can been verified next.

```
>> prk_v2=PreKernel(v2)
```

```
prk_v2 =

  6.7222   2.6667   7.7222   7.7222
```

That this result is also the pre-nucleolus of the game can be verified with the function *tug_BalancedSelectionQ()*. This function is an implementation of Kolberg's Theorem indicating an imputation as the pre-nucleolus of the game iff the induced collections are balanced.

```
>> BSQ=tug_BalancedSelectionQ(v2,prk_v2)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)


Passing Game to Mathematica ...
Is the selection balanced?...
Mathematica Kernel quitting per your request...

BSQ =

  BalancedSelectionQ: 'True'
```

Similar, applying this function on the pre-nucleolus of game v_2, we get some additional evidence that the result must be correct.

```
>> BSQ2=tug_BalancedSelectionQ(v2,prk_v2)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)


Passing Game to Mathematica ...
Is the selection balanced?...
Mathematica Kernel quitting per your request...

BSQ2 =

  BalancedSelectionQ: 'True'
```

Let us now reproduce the above result while calling the function that computes the pre-nucleolus under *MATHEMATICA*®. This function is called *tug_PreNuc()* and is applied with one input argument, namely the TU game under consideration. Thus, we execute this function through

```
>> SOLpn=tug_PreNuc(v2)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =
```

```
9.0 for Linux x86 (64-bit) (November 20, 2012)

Passing Game to Mathematica ...
Mathematica Kernel quitting per your request...

SOLpn =

   PreNucleolus: [6.7222 2.6667 7.7222 7.7222]
  MPreNucleolus: ' 121  8  139  139
                  {---, -, ---, ---}
                   18   3  18   18'
```

This result is confirmed with the Matlab function *PreNucl()* as can be seen next.

```
>> pn_v2=PreNucl(v2)
Optimization terminated.
Optimization terminated.
Optimization terminated.

pn_v2 =

  6.7222   2.6667   7.7222   7.7222
```

This result is identical with the above computed pre-kernel element, this is verified by

```
>> rats(pn_v2-prk_v2)

ans =

     *        *        *        *
```

Of course, we can also evaluate the nucleolus of a TU game. In this case, the package provides two functions. The first one is called *tug_Nuc()* and can be called while executing

```
>> SOLnc=tug_Nuc(v2)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)

Passing Game to Mathematica ...
Mathematica Kernel quitting per your request...

SOLnc =

   Nucleolus: [6.7222 2.6667 7.7222 7.7222]
  MNucleolus: '121  8  139  139
               {---, -, ---, ---}
                18   3  18   18'
```

The second function is called *tug_Mnuc()*, which computes the nucleolus while iteratively solving a system of linear programming problems.

```
>> SOLnc2=tug_Mnuc(v2)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
```

```
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)

Passing Game to Mathematica ...
Mathematica Kernel quitting per your request...

SOLnc2 =

   Nucleolus: [6.7222 2.6667 7.7222 7.7222]
  MNucleolus: '121  8  139  139
         {---, -, ---, ---}
          18  3  18   18'
```

In both cases we receive the correct result. However, the second function has a higher reliability than the first and should be the preferred method to compute the nucleolus under *MATHEMATICA*®. Moreover, by this evaluation, we observe that the pre-nucleolus coincides with the nucleolus of the game, we suppose that the game v_2 should be zero-monotonic. And indeed, while calling the command *tug_ZeroMonotoneQ* our conjecture is confirmed.

```
>> SOLzmQ=tug_ZeroMonotoneQ(v2)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)

Passing Game to Mathematica ...
Mathematica Kernel quitting per your request...

SOLzmQ =

  ZeroMonotoneQ: 'True'
```

It remains to discuss the *MATHEMATICA*® command that computes the anti pre-kernel of a TU game. This solution concept can be evaluated with the function *tug_AntiPreKernel()*

```
>> SOLapk=tug_AntiPreKernel(v2)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)

Passing Game to Mathematica ...
Computing the Anti Pre-Kernel ...
Mathematica Kernel quitting per your request...

SOLapk =

   AntiPreKernel: [7.6111 3 6.6111 7.6111]
  MAntiPreKernel: ' 137   119  137
            {---, 3, ---, ---}
             18    18   18'
```

Again, this result can be affirmed while invoking the toolbox command *Anti_PreKernel()*

```
>> apk_v2=Anti_PreKernel(v2)

apk_v2 =

  7.6111   3.0000   6.6111   7.6111
```

Finally, let us see how we can assign a complete payoff matrix to the function *tug_PreKernel()*. The simplest way to construct a payoff matrix, is to compute all core vertices whenever it exists and the game is not too large. Thus, we invoke

```
>> crv_v2=CddCoreVertices(v2)
Optimization terminated successfully.

crv_v2 =

  10.8333      0   2.1667  11.8333
   9.0000      0  11.8333   4.0000
   4.0000      0  11.8333   9.0000
   4.0000      0   9.0000  11.8333
  10.8333      0  10.0000   4.0000
  10.8333   4.0000  10.0000  -0.0000
   8.0000   5.3333  11.5000      0
   8.0000   5.0000  11.8333      0
   9.0000   4.0000  11.8333      0
   9.5000   5.3333  10.0000      0
  10.8333   4.0000      0  10.0000
  10.8333   2.1667      0  11.8333
   9.0000   5.3333      0  10.5000
   9.0000   4.0000      0  11.8333
   9.5000   5.3333      0  10.0000
      0   5.3333  11.5000   8.0000
      0   5.0000  11.8333   8.0000
      0   5.3333   9.0000  10.5000
      0   4.0000  11.8333   9.0000
  -0.0000   4.0000   9.0000  11.8333
```

Before we answer the question, let us check whether the computed vertices belong to the core of the game v_2. And indeed, all of them are core elements.

```
>> CRQ_v=tug_BelongToCoreQ(v2,crv_v2)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)


Passing Game to Mathematica ...
Mathematica Kernel quitting per your request...

CRQ_v =

  CoreElementQ: [1x127 char]


>> CRQ_v.CoreElementQ
```

```
ans =

{True, True, True, True, True, True, True, True, True, True, True, True, True,

>  True, True, True, True, True, True, True}
```

This result is confirmed by:

```
>> for k=1:20 crq(k)=belongToCoreQ(v2,crv_v2(k,:),'real');end
>> crq

crq =

   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
```

None of these vertices imputations should be the pre-nucleolus, hence none of the induced collections should be balanced, this is verified next by issuing:

```
>> BCRVQ=tug_BalancedSelectionQ(v2,crv)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)


Passing Game to Mathematica ...
Is the selection balanced?...
Mathematica Kernel quitting per your request...

BCRVQ =

  BalancedSelectionQ: [1x147 char]

>> BCRVQ.BalancedSelectionQ

ans =

{False, False, False, False, False, False, False, False, False, False, False,

>  False, False, False, False, False, False, False, False}
```

A note on caution is required here. It is not allowed to supply every *tug*-function with a payoff matrix. Check first by *help function_name* if the required function allows such an input format.

Hence, we constructed a grid of 20 different starting points. The result next indicates that all starting points converge to the same pre-kernel element.

```
>> PRK_v2a=tug_PreKernel(v2,crv_v2)
Mathematica Kernel loading...
Mathematica Kernel quitting per your request...
Mathematica Kernel loading...

ans =

9.0 for Linux x86 (64-bit) (November 20, 2012)


PRK_v2a =
```

```
    PreKernel: [6.7222 2.6667 7.7222 7.7222]
   MPreKernel: ' 121 8 139 139
          {{--- , - , --- , ---}}
            18  3 18  18'
  MPreKernel_Q: '{True}'
```

This result is again confirmed with Matlab.

```
>> tic;for k=1:20 prkmat(k,:)=PreKernel(v2,crv_v2(k,:)); end;toc
Elapsed time is 0.065066 seconds.
>> prkmat

prkmat =

   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
   6.7222   2.6667   7.7222   7.7222
```

Giving some evidence that the pre-kernel is single-valued.

6. Limitations

Please, notice that the Matlab toolbox *MatTuGames* is still on an early stage of development (Version 0.4). Some functions are still experimental and should, therefore, be used with care. Especially, all functions that check some consistency properties should be used with extreme care. For the function *assignment_game()* we have to fix a memory problem. For instance, for problems having 9 sellers onward the computation needs at least 90.72 GB, however, for 10 sellers more than 3628.8 GB are needed to perform successfully a computation. This is due to the fact that each asymmetric assignment problem will be transformed into a symmetric one. Thus, in the former case 18-person will be treated in the latter 20-person. Finally, the function *PreKernel()* must be made more robust against precision problems for games having more than 15-players.

In general, all functions should work up to 52 players. Except the functions *gameToMatlab(), gameToMama()*, and *sortsets()*, these functions are limited up to 35 players. Moreover, the function *sortsets()* only coincides with the *MATHEMATICA®* order of a power set up to 16 players, which is due to a Matlab internal representation of numbers with respect to a certain base. Namely, the representation of a particular number with respect to base 17 is different from that of base 18 or 36. This causes a change in the ordering for some coalitions which have been already sorted correctly when using the base 17 instead of

18 or 36. In contrast to our expectation, the sorting process is not invariant against a change in its base number. Finally, due to a code revision of the *p_PreKernel()* function, we could overcome the problem of a transmission limit of the *parfor loop* of 2 GB in Matlab's PCT that triggers a Java Exception failure or a serialization error for 26 players onward. Now, we are able to compute successfully pre-kernel elements up to 32 person TU games in about 44 minutes. However, such an operation requires at least 128 GB memory.

REFERENCES

E. Algaba, J. M. Bilbao, P. Borm, and J. J. López. The Position Value for Union Stable Systems. *Mathematical Methods of Operations Research*, 52:221–236, 2000.

E. Algaba, J. M. Bilbao, P. Borm, and J. J. López. The Myerson Value for Union Stable Structures. *Mathematical Methods of Operations Research*, 54:359–371, 2001.

E. Algaba, J. M. Bilbao, and J. J. López. The Position Value in Communication Structures. *Mathematical Methods of Operations Research*, 59:465–477, 2004.

R. J. Aumann and M. Maschler. Game Theoretic Analysis of a Bankruptcy Problem form the Talmud. *Journal of Economic Theory*, 36:195–213, 1985.

R. J. Aumman and J. H. Drèze. Cooperative Games with Coalition Structures. *International Journal of Game Theory*, 3:217–237, 1974.

P. Borm, G. Owen, and SH. Tijs. On the Position Value for Communication Situations. *SIAM Journal of Discrete Mathematics*, 5:305–320, 1992.

E. Calvo and E. Gutiérrez. A Value for Cooperative Games with a Coalition Structure. Technical report, University of Valencia, Spain, 2012.

E. Calvo and E. Gutiérrez. The Shapley-Solidarity Value for Games with a Coalition Structure. *International Game Theory Review*, 15(1):1350002–1–1350002–24, 2013.

I. Curiel. *Cooperative game theory and applications*, volume 16 of *Theory and Decision Library: Series C*. Kluwer Acad. Publ., Boston, 1997.

J. Derks and J. Kuipers. *Implementing the Simplex Method for Computing the Prenucleolus of Transferable Utility Games*. Department of Mathematics, University of Maastricht, the Netherlands, 1997.

T. Driessen. *Cooperative Games, Solutions and Applications*. Kluwer Academic Publishers, 1988.

U. Faigle, W. Kern, and J. Kuipers. An Efficient Algorithm for Nucleolus and Prekernel Computation in some Classes of TU-Games. *International Journal of Game Theory*, 30(1):79–98, Oct. 2001.

Y. Funaki and H. I. Meinhardt. A Note on the Pre-Kernel and Pre-Nucleolus for Bankruptcy Games. *The Waseda Journal of Political Science and Economics*, 363:126–136, 2006.

S. Hart and A. Mas-Colell. Potential, Value and Consistency. *Econometrica*, 57:589–614, 1989.

E. Iñarra and J.M. Usategui. The Shapley value and average convex games. *International Journal of Game Theory*, 22:13–29, 1993.

I. Katsev and E. Yanovskaya. Between the Prekernel and the Prenucleolus. Technical report, St. Petersburg Institute for Economics and Mathematics, St. Petersburg, Russia, 2010. mimeo.

J-E. Martinez-Legaz. Dual Representation of Cooperative Games based on Fenchel-Moreau Conjugation. *Optimization*, 36: 291–319, 1996.

M. Maschler. The Bargaining Set, Kernel and Nucleolus. In R. J. Aumann and S. Hart, editors, *Handbook of Game Theory*, volume 1, chapter 18, pages 591–668. Elsevier Science Publishers, Amsterdam, 1992.

M. Maschler, B. Peleg, and L. S. Shapley. Geometric Properties of the Kernel, Nucleolus, and Related Solution Concepts. *Mathematics of Operations Research*, 4:303–338, 1979.

R. Meessen. Communication Games. Master's thesis, University of Nijmegen, the Netherlands, 1988. in Dutch.

H. Meinhardt. An LP Approach to Compute the Pre-Kernel for Cooperative Games. *Computers and Operations Research*, 33/2: pp 535–557, 2006.

H. I. Meinhardt. *The Pre-Kernel as a Tractable Solution for Cooperative Games: An Exercise in Algorithmic Game Theory*, volume 45 of *Theory and Decision Library: Series C*. Springer Publisher, Heidelberg/Berlin, 2013. Forthcoming.

A. Meseguer-Artola. Using the Indirect Function to characterize the Kernel of a TU-Game. Technical report, Departament d'Economia i d'Història Econòmica, Universitat Autònoma de Barcelona, Nov. 1997. mimeo.

R. B. Myerson. Graphs and Cooperation in Games. *Mathematics of Operations Research*, 2(3):225–229, August 1977.

R. B. Myerson. Conference Structures and Fair Allocation Rules. *International Journal of Game Theory*, 9(3):169–182, 1980.

R. B. Myerson. *Game Theory: Analysis of Conflict*. Harvard University Press, Cambridge, Massachusetts, London, England, 1991.

A. S. Nowak and T. Radzik. A Solidarity Value for n-Person Transferable Utility Games. *International Journal of Game Theory*, 23:43–48, 1994.

A. Ostmann. Classifying Three-Person Games. Technical Report 140, University of Bielefeld, 1984.

G. Owen. Values of Games with a priori Unions. In R. Henn and O. Moeschlin, editors, *Essays in Mathematical Economics and Game Theory*, pages 76–88. Springer-Verlag, New York, 1977.

G. Owen. *Game Theory*. Academic Press, San Diego, 3 edition, 1995.

B. Peleg and P. Sudhölter. *Introduction to the Theory of Cooperative Games*, volume 34 of *Theory and Decision Library: Series C*. Springer-Verlag, 2 edition, 2007.

R. Rockafellar. *Convex Analysis*. Princeton University Press, Princeton, 1970.

J. Rosenmüller. *The Theory of Games and Markets*. North-Holland, Amsterdam, 1981.

L. M. Ruiz, F. Valenciano, and J. M. Zarzuelo. The Least Square Pre-Nucleolus and the Least Square Nucleolus. Two Values for TU Games based on the Excess Vector. *International Journal of Game Theory*, 25:113–134, 1996.

R.E. Stearns. Convergent Transfer Schemes for N-Person Games. *Transaction American Mathematical Society*, 134:449–459, 1968.

P. Sudhölter. Star-Shapedness of the Kernel of Homogeneous Games. *Mathematical Social Sciences*, (32):179–214, 1996.

A. van den Nouveland, P. Borm, and SH. Tijs. Allocation Rules for Hypergraph Communication Situations. *International Journal of Game Theory*, 20:255–268, 1992.

H. Wolsey. The Nucleolus and Kernel for Simple Games or Special Valid Inequalities for 0-1 Linear Integer Programs. *International Journal of Game Theory*, 5(4):227–238, 1976.

K. Yokote, Y. Funaki, and Y. Kamijo. Linear Basis Approach to the Shapley Value. Working Paper 1303, Waseda University, 2013.

A. PreKernel Computation Timing Tables

Table A.1: Tests conducted on HP XC3000 with *PreKernel()* Version 0.3[a]

	Serial Computation[b]					
# of Players	CONVEX[c]	WEIGHTED[d]	DUAL[e]	errs CONVEX[f]	errs WEIGHTED[f]	errs DUAL[f]
3	0.0017	0.0012	0.0018	0	0	0
4	0.0024	0.0019	0.0030	0	0	0
5	0.0033	0.0032	0.0069	0	0	6
6	0.0045	0.0062	0.0118	0	0	10
7	0.0059	0.0107	0.0183	0	0	12
8	0.0075	0.0173	0.0278	0	0	10
9	0.0096	0.0256	0.0488	0	1	23
10	0.0120	0.0388	0.0763	0	0	22
11	0.0152	0.0518	0.0950	0	0	26
12	0.0189	0.0749	0.0961	0	2	7
13	0.0246	0.0985	0.0826	0	0	7
14	0.0298	0.1129	0.0560	0	1	2
15	0.0402	0.1226	0.0572	0	0	1
16	0.0586	0.1551	0.0887	0	0	0
17	0.0944	0.2015	0.1520	0	0	0
18	0.1617	0.2856	0.2838	0	0	0
19	0.2994	0.4353	0.5449	0	0	0
20	0.6067	0.7392	1.1539	0	0	0
21	1.2133	1.2491	2.5024	0	0	0
22	2.6055	2.3979	4.9713	0	0	0
23	5.3015	5.2917	10.6131	0	0	0
24	9.8454	8.6540	21.8746	0	0	0

[a] For each number of players 1000 randomly generated games have been considered.
[b] Av. comp. time in sec/per game; seed value=135; timing includes games generating; code optimized.
[c] Modest Bankruptcy: $B_{es} = 2/3 \cdot \sum_{k \in N} d_k$. Random claims vector $\vec{d}$.
[d] Weighted Majority: $th = 2/3 \cdot \sum_{k \in N} w_k$. Random weights vector $\vec{w}$.
[e] Greedy Bankruptcy: $B_{es} = 2/3 \cdot \sum_{k \in N} d_k$. Random claims vector $\vec{d}$.
[f] Errors per 1000 games.

Table A.2: Tests conducted on HP XC3000 with *p_PreKernel()* Version 0.3[a]

	Parallel Computation[b,c]					
# of Players	CONVEX[d]	WEIGHTED[e]	DUAL[f]	errs CONVEX[g]	errs WEIGHTED[g]	errs DUAL[g]
20	1.0442	0.9254	0.7180	0	0	0
21	1.6722	1.4013	1.3293	0	0	0
22	3.0278	2.2800	2.5830	0	0	0
23	5.7555	4.0650	5.1338	0	0	0
24	12.0373	8.5176	10.0088	0	0	0
25	24.9151	15.1958	22.4148	0	0	3
26	48.7479	30.6607	42.5573	0	0	0
27	102.0533	71.8561	87.2631	0	0	0
28	212.2603	143.0936	185.4692	0	0	1

[a] For each number of players 1000 randomly generated games have been considered.
[b] 8 workers launched; code optimized.
[c] Av. comp. time in sec/per game; seed value=135; timing includes games generating.
[d] Modest Bankruptcy: $B_{es} = 2/3 \cdot \sum_{k \in N} d_k$. Random claims vector $\vec{d}$.
[e] Weighted Majority: $th = 2/3 \cdot \sum_{k \in N} w_k$. Random weights vector $\vec{w}$.
[f] Greedy Bankruptcy: $B_{es} = 2/3 \cdot \sum_{k \in N} d_k$. Random claims vector $\vec{d}$.
[g] Errors per 1000 games.

Table A.3: Tests conducted on HP XC3000 with *PreKernel()* Version 0.2[a]

# of Players	Serial Timing[b] convex[c]	weighted[d]	dual[e]	errs convex[f]	errs weighted[f]	errs dual[f]
3	0.0018	0.0014	0.0025	0	0	0
4	0.0026	0.0024	0.0044	0	0	0
5	0.0036	0.0043	0.0112	0	0	19
6	0.0050	0.0085	0.0178	0	0	12
7	0.0068	0.0153	0.0281	0	0	14
8	0.0095	0.0266	0.0445	0	0	12
9	0.0138	0.0454	0.0866	0	3	32
10	0.0210	0.0817	0.1505	0	2	40
11	0.0353	0.1274	0.2249	0	3	38
12	0.0636	0.2215	0.3201	0	4	27
13	0.1358	0.4064	0.3599	0	7	15
14	0.2811	0.7345	0.3167	0	1	4
15	0.6206	1.0599	0.3249	0	0	0
16	1.4570	2.0261	0.6362	0	0	0
17	3.3072	4.0059	1.3885	0	0	0
18	7.3814	8.0608	2.7642	0	0	0
19	17.5067	16.7180	5.9576	0	0	0
20	34.6234	34.8509	14.0235	0	0	0
21	78.7356	72.7717	27.7113	0	0	0
22	178.1647	144.7894	58.8437	0	0	0
23		320.9033			0	

[a] For each number of players 1000 randomly generated games have been considered.
[b] Av. sec/per game; seed value=135; timing includes games generating, code non optimized.
[c] Modest Bankruptcy: $E = 2/3 \cdot \sum_{k \in N} d_k$. Random claims vector $\vec{d}$.
[d] Weighted Majority: $th = 2/3 \cdot \sum_{k \in N} w_k$. Random weights vector $\vec{w}$.
[e] Greedy Bankruptcy: $E = 2/3 \cdot \sum_{k \in N} d_k$. Random claims vector $\vec{d}$.
[f] Errors per 1000 games.

Table A.4: Tests conducted on HP XC3000 with *p_PreKernel()* Version 0.2[a]

# of Players	Parallel Timing[b,c] convex[d]	weighted[e]	dual[f]	errs convex[g]	errs weighted[g]	errs dual[g]
20	12.9722	11.8045	5.1242	0	0	0
21	24.8659	21.9850	10.3084	0	0	0
22	39.7223	35.1208	18.3739	0	0	0
23	91.3738	79.0473	42.2971	0	0	0
24	193.3406[h]	162.0199	88.7832[h]	0[h]	0	0[h]
25	467.5221[j]	388.2963[i]	170.5757[j]	0[j]	0[i]	0[j]
26[k]	836.0022[j]	666.1579[j]	328.5669[j]	0[j]	0[j]	0[j]

[a] For each number of players 1000 randomly generated games have been considered.
[b] 8 workers launched, code optimized.
[c] Av. sec/per game; seed value=135; timing includes games generating.
[d] Modest Bankruptcy: $E = 2/3 \cdot \sum_{k \in N} d_k$. Random claims vector $\vec{d}$.
[e] Weighted Majority: $th = 2/3 \cdot \sum_{k \in N} w_k$. Random weights vector $\vec{w}$.
[f] Greedy Bankruptcy: $E = 2/3 \cdot \sum_{k \in N} d_k$. Random claims vector $\vec{d}$.
[g] Errors per 1000 games.
[h] Only 800 randomly generated games have been considered.
[i] Only 700 randomly generated games have been considered.
[j] Only 160 randomly generated games have been considered.
[k] Using WorkerObjectWrapper by Edric Ellis to handle parfor transmission limit of 2 GB.

SUBJECT INDEX