TuGames v1.1

by Holger I. Meinhardt (2005)

Contents

1 Introduction

TuGames is a *MATHEMATICA*® package to determine and to check some game properties of transferable utility games. It provides about 70 different functions and it can calculate, for instance, kernel elements, excess payoffs, marginal values, the tau-value, and the vertices of a core. Moreover, it verifies if the game is convex, average-convex or super-additive just to mention some interesting functions. This package is an extension of the package *CooperativeGames* that has been developed by M. Carter. It must be mentioned in this place that some commands of *TuGames* require routines that have been provided by the package *CooperativeGames*. Therefore, you have also to install the package *CooperativeGames* to use the new functions properly. A description of the package *CooperativeGames* can be found in Economic and Financial Modeling with *MATHEMATICA*®, ed. Hal. R. Varian, Telos Springer Publisher,1993, Chapter 8. Furthermore, if one is interested in computing the vertices of a core the *MATHEMATICA*® package *VertexEnum* written by K. Fukuda and I. Mizukoshi must also be installed on the computer. But note that this function is very slow in computing all vertices of a core on old computers like Pentium II systems. You can overcome these shortcomings of VertexEnum by installing the C-library cddmathlink written by the same authors to perform the same computational task more efficiently. It can be found under http://www.cs.mcgill.ca/ fukuda/ for various UNIX and for Window systems. The library is linked via MathLink with the *MATHEMATICA*® Kernel. For Windows Operating Systems pre-build binaries are available and are included in the subdirectory cddml. Note that the functions that based on this library are not activated. To use these functions on your operating systems you must comment them out in the package TuGames. Moreover, you have to adjust the $Path variable where the library can be found. The default value is set to "/usr/local/bin" for UNIX systems. For changing this value edit the file *TUGames* with an editor of your choice and search for **SetDirectory[]**. The package *TuGames* can be used under Windows XP/2000/NT, MacOS and UNIX platforms running *MATHEMATICA*® Version 3.0 or later. The author has tested the functions extensively under LINUX x86/64, HP-UX and AIX. Furthermore, the package was also be installed and tested successfully under Windows XP. For Windows 2000/NT and Mac OS the author has no experience. But the programming language of *MATHEMATICA*® is system independent, thus, there should no problems occur on these operating systems.

2 Installation

2.1 Unix

1. Create a directory "**TuGames**" in your **$HOME/.Mathematica /3.0/AddOns/Applications** directory. Copy now all files in the directory "**TuGames**", but for the file *CooperativeGames.m* create a subdirectory "**coop**" in *TuGames* and move the file in this subdirectory.
2. Create the variable **$TuGamesPath** in the *init.m* file in your **$HOME/.Mathematica/3.0/Kernel** directory. If no *init.m* file exists, create the file. Set this variable to
$TuGamesPath = $HomeDirectory<>".Mathematica/3.0/AddOns/Applications/TuGames"
3. Add to the global variable **$Path** in the *init.m* file the location of *TuGames* by
AppendTo[$Path, $TuGamesPath]
Now you can start *TuGames.*

Note that for UNIX/LINUX systems is also a graphical extension available. See also the remark at the end of the README file.

2.2 Windows

1. Create the directory "**AddOns\Applications\TuGames**" at the location where your *MATHEMATICA*® files are located. This can be find out by typing
$TopDirectory
in your *MATHEMATICA*® Notebook.
2. Copy now all files in this directory.
2.1 Create for the file *CooperativeGames.m* the subdirectory "**coop**" and move the file in this directory.
2.2 Create for the file VertexEnum.m the subdirectory "vertex" and move the file in this directory.
2.3 Create for the library cddml_w32new the subdirectory "cddml" and move the file in this directory.
3. Create in your *init.m* file the variable **$TuGamesPath** to locate the various packages. Set this variable to
$TuGamesPath = $TopDirectory<>"\\AddOns\\Applications\\TuGames"
4. Add to the variable **$Path** in the *init.m* file the location of *TuGames*
AppendTo[$Path, $TuGamesPath]
Now you can start *TuGames.*

Note that the double backslash (\\) is important. A simple backslash is not sufficient to set the directory path correctly. In cases that you have still problems with the init.m file consult the README file.

2.3 Mac OS

Sorry, but at the moment no installation instruction for Mac OS is available.

3 Getting Started

In the next step we assume that you have installed properly the files mentioned above on your computer. To start with the calculation, we have first to load some packages. This can be done by the following commands.

```
In[1]:= Needs["coop`CooperativeGames`"]

        Needs["TuGames`"]

        Needs["VertexEnum`"]
```

To get an overview of all available functions call just

```
In[2]:= funcname = Names["TuGames`*"]
```

```
Out[2]= {AdjustedEffUpperVectors, AdjustedEffVector, AdjustedWorthVectors,
          AllConstraints, AllMaxSurpluses, Assign, AvConvexQ, AvSumCheck, BelongToCoreQ,
          CddGmpVerticesCore, CddVerticesCore, CddVerticesImputationSet, CddVerticesReasonableSet,
          ChangeInternalEps, Concession, ContributionSum, ConvCheck, ConvexQ, ConvexStrQ,
          ConvexUnanConditionQ, ConvStrQ, CoordS, CoreElementsQ, DefineGame,
          DeltaLP, DisagreeConvex, Disagreement, DualGame, EpsCore, EpsilonValue, EpsValue,
          EvalSumMinCoord, ExcessPayoff, FirstCriticalVal, FourthCriticalVal,
          GameMonotoneQ, Gap, IncreasingMargContributions, kCover, Kernel,
          KernelCalculation, KernelImputationListQ, KernelImputationQ, LargestAmount,
          MargValue, MaxSurplus, MaxExcessBalanced, MinUnanimityCoordinates,
          MonotoneQ, NewShapley, Nuc1convex, OneNormalization, PreKernelQ, Quota,
          RationalApproximate, RationalExact, ReasonableOutcome, ReasonableSet, Scrb, ScrbSolution,
          SecondCriticalVal, SelectSuperSets, SmallestContribution, SmallestContributionVector,
          StarCriticalVal, StrictlyConvexUnanConditionQ, StrIncreasMargContrib,
          SuperAdditiveQ, TauValue, ThirdCriticalVal, Transfer, TransferConstraints,
          UnanimityCoordinates, UtopiaPayoff, UtopiaSum, UtopiaVector, VerticesCore,
          WeaklySuperAdditiveQ, ZeroMonotoneQ, ZeroNormalization, ZeroOneNormalization}
```

For getting the most recent listing of all available functions in the package *TUGames* type

```
In[3]:= ?TuGames`*
```

TuGames`

AdjustedEffUpperVectors
AdjustedEffVector
AdjustedWorthVectors
AllConstraints
AllMaxSurpluses
Assign
AvConvexQ
AvSumCheck
BelongToCoreQ
CddGmpPlotCore
CddGmpPlotImputationSet
CddGmpPlotReasonableSet
CddGmpVerticesCore
CddVerticesCore
CddVerticesImputationSet
CddVerticesReasonableSet
ChangeInternalEps
coal
Concession
ContributionSum
ConvCheck
ConvexQ
ConvexStrQ
ConvexUnanConditionQ
ConvStrQ
CoordS
CoreElementsQ
DefineGame
DeltaLP
DisagreeConvex
Disagreement
DisplayAllResults
DisplayMatrixForm
DualGame
EpsCore
EpsilonValue
EpsValue
EvalSumMinCoord
ExcessPayoff
FilledCore
FirstCriticalVal
FourthCriticalVal
GameMonotoneQ
Gap
ImputationQ
IncreasingMargContributions
kCover
Kernel
KernelCalculation
KernelImputationListQ
KernelImputationQ
KernelPoints
KernelSubCall
KernelSubCallEmpty
KernelSubCallZero
LargestAmount
MargValue
MaxExcessBalanced
MaxSurplus
MinUnanimityCoordinates
MonotoneQ
NewShapley
Nuc1convex
OneNormalization
PreKernelQ
Quota
RationalApproximate
RationalExact
ReasonableOutcome
ReasonableSet
Scrb
ScrbSolution
SecondCriticalVal
SelectSuperSets
SetGameToNonZeroMonotonic
SmallestContribution
SmallestContributionVector
StarCriticalVal
StrictlyConvexUnanConditionQ
StrIncreasMargContrib
SuperAdditiveQ
TauValue
ThirdCriticalVal
TIJsets
Transfer
TransferConstraints
UnanimityCoordinates
UtopiaPayoff
UtopiaSum
UtopiaVector
Vec3DToSimplex
VerticesCore
WeaklySuperAdditiveQ
WithIncidences
ZeroMonotoneQ
ZeroNormalization
ZeroOneNormalization

Just click with your mouse pointer on the functions name to get a short description how to use it.

```
ConvexQ[game] checks if the Tu - game is convex.
 It returns the value 'True' or 'False'.
```

To obtain a brief documentation about individual functions type

```
In[4]:= ?TuGames`ConvexQ
 ConvexQ[game] checks if the Tu - game is
     convex. It returns the value 'True' or 'False'.
```

if you are interested in a specific function name like **ConvexQ[]**.

3.1 How to define Games?

Note first that in the sequel we indicate the whole player set in the commands with the symbol T instead of N. We must deviate from the general game theoretical convention, since under *MATHEMATICA*® the symbol N is already occupied by the built-in function **N[]** to evaluate an expression numerically. Whereas in the definitions and results we use the symbol N in the usual convention.
We can define a game in two different ways. The first one is the way as it was implemented by Carter in his *MATHEMATICA*® package *CooperativeGames*. This procedure becomes very inconvenient for large games. The package *TuGames* provides, in addition, the function **DefineGame[]** to assign the values of the characteristic function for large games. Let us now present the representation of a TU-game used by Carter. The values of the characteristic function are assigned by

```
In[5]:= ExpGame1 := (T = {1, 2, 3, 4}; Clear[v];
            v[{}] = 0;
            v[{1}] = 0;
            v[{2}] = 0;
            v[{3}] = 0;
            v[{4}] = 0;
            v[{1, 2}] = 0;
            v[{1, 3}] = 0;
            v[{1, 4}] = 0;
            v[{2, 3}] = 0;
            v[{2, 4}] = 0;
            v[{3, 4}] = 0;
            v[{1, 2, 3}] = 40;
            v[{1, 2, 4}] = 0;
            v[{1, 3, 4}] = 40;
            v[{2, 3, 4}] = 0; v[T] = 90; )
```

To call the function **DefineGame[]** we define in a first step the player set and create a vector with zeros of length 2^n.

```
In[6]:= T2 = {1, 2, 3, 4};

        vec2 = Table[0, {i, 16}];
```

Then we can assign positive values to some or all coalitions. It is also possible to assign the coalitional values directly to the vector, but in this case you have to know the exact order how the components of the vector are assigned to the coalitions. Note that you should not assign negative or floating point values to the coalitions. The commands in the package *TuGames* require nonnegative or rational numbers for the coalitional values to perform the computational task correctly. For instance, if you use floating point numbers some functions can run into an infinite loop and never terminate. In this case choose from the menu **Kernel -> Abort Evaluation**.

```
In[7]:= ExpGame2 := (DefineGame[T2, vec2]; v[{1, 2, 3}] = 40;
            v[{1, 3, 4}] = 40; v[T] = 90; );
```

Here is an example with five players. The example is borrowed from Maschler, Peleg and Shapley (1979) (cf. p. 317).

```
In[8]:= T3 = {1, 2, 3, 4, 5};

        vec3 = Table[0, {i, 32}];

In[9]:= ExpGame3 := (DefineGame[T3, vec3]; v[{1, 2, 4}] = 4; v[{1, 2, 5}] = 4;
           v[{1, 3, 4}] = 4; v[{1, 3, 5}] = 4; v[{2, 3, 4}] = 4; v[{2, 3, 5}] = 4;
           v[{4, 5}] = 4; v[T] = 7; );
```

Next an average-convex game:

```
In[10]:= ExpGame4 := (DefineGame[T2, vec2]; v[{1, 2}] = 1/4; v[{1, 4}] = 1/4;
            v[{2, 3}] = 1/4; v[{3, 4}] = 3/8; v[{1, 2, 3}] = 1/2; v[{1, 2, 4}] = 1/2;
            v[{1, 3, 4}] = 1/2; v[{2, 3, 4}] = 1/2; v[T] = 1; );
```

Note that we have just assigned the values to the characteristic functions, but these values are at the moment not evaluated. Since under the operator ":=" the right hand side, that means, in our case the game, is maintained in an unevaluated form. The game is at that moment evaluated when a function is invoked that requires as an input the pattern *game_name,* that is, in the case that the left hand expression *game_name* appears, it is replaced by the right hand expression, the game. To see that at the moment no game is really evaluated type

```
In[11]:= ??v
v[S] describes the worth of coalition S
```

Due to the fact that nothing is evaluated at the moment you get a short function description back. Problems encountered with the operator ":=" in the context of our package is briefly discussed in Section 6.

4 Some Functions

In this section we will discuss some basic functions provided by *TuGames* to examine some game properties like convexity, average-convexity, superadditivity, monotonicity and zero-monotonicity. Furthermore, *TuGames* will also enable us to evaluated solutions of a game like the Shapley value, tau-value or the kernel. In the sequel of the section we will discuss as well functions related to the notion of k-convexity and unanimity coordinates. At this place some remarks are required concerning the presentation of the material in this notebook. The purpose of this notebook is to give the potential user a short description at hand how to use the new commands related to transferable utility games correctly. We will not embed the descriptions of the commands in a thorough game theoretical presentation of definitions and results on which the commands are based on. We just introduce definitions and results in cases where it is according to our opinion useful to do so. This is done to mention differences between functions that do ostensible the same or where we have used special results in our functions that are not quite common. In addition, we will also dispense from any code presentation and discussion. Users interested in this material should examine and try out for themselves the code presented in the package *TuGames*. Moreover, we assume that users bring along good knowledge in cooperative game theory to asses the results correctly in a game theoretical context. Nevertheless, we hope that the package might be useful for research as well as for teaching purpose in constructing examples in both of these fields. Certainly, it is assumed that users have some experience with *MATHEMATICA®,* but it is not a prerequisite for the use of the package.

4.1 Basic Functions

In general, functions are invoked by the function name and its parameters, in most cases a function needs as an input parameter the *game_name*, a list of imputations (payoffs), a set of coalitions or a player from the player set. To make the point more precise let us first start with some functions that need just one parameter to examine some well known game properties. The function that examines the convexity condition of the game is called **ConvexQ[]** and it needs just one parameter, namely the *game_name* to perform the necessary calculations to check on this game property. We can verify convexity by invoking

```
In[12]:= ConvexQ[ExpGame1]
Out[12]= True

In[13]:= ConvexQ[ExpGame2]
Out[13]= True

In[14]:= ConvexQ[ExpGame3]
Out[14]= False

In[15]:= ConvexQ[ExpGame4]
Out[15]= False
```

Additionally to the possibility to check whether a game is convex, we are also able to examine with the command **AvConvexQ[]** a generalized convexity property, the so-called average convexity. Since, average-convexity generalize convexity it should be clear that convexity implies average-convexity, but the converse is not true. The property of average-convexity has been introduced by Inarra and Usategui (1993) in the literature. Before we will discuss this command note first that average-convexity is defined as follows

$$\sum_{i \in S} [v(S) - v(S\backslash i)] \leq \sum_{i \in S} [v(R) - v(R\backslash i)] \quad \text{for all } S \subseteq R \subseteq N,\ S \neq \phi.$$

Similar to the function **ConvexQ[]** the function **AvConvexQ[]** that test the average-convexity condition of the game is called with one parameter, the *game_name*. Hence, we need just to type

```
In[16]:= AvConvexQ[ExpGame1]
Out[16]= True

In[17]:= AvConvexQ[ExpGame2]
Out[17]= True

In[18]:= AvConvexQ[ExpGame3]
Out[18]= False

In[19]:= AvConvexQ[ExpGame4]
Out[19]= True
```

Superadditivity can be checked by

```
In[20]:= SuperAdditiveQ[ExpGame1]
Out[20]= True

In[21]:= SuperAdditiveQ[ExpGame2]
Out[21]= True

In[22]:= SuperAdditiveQ[ExpGame3]
Out[22]= False

In[23]:= SuperAdditiveQ[ExpGame4]
Out[23]= True
```

Let the coalitions $S_1, ..., S_k$ partition the grand coalition T. A necessary condition for the core $C(v)$ of a game v to be nonempty is that the condition

$$\sum_{i=1}^{k} v(S_i) \leq v(N)$$

is satisfied. This property is captured by the function **WeaklySuperAdditiveQ[].**

```
In[24]:= WeaklySuperAdditiveQ[ExpGame1]
Out[24]= True

In[25]:= WeaklySuperAdditiveQ[ExpGame2]
Out[25]= True

In[26]:= WeaklySuperAdditiveQ[ExpGame3]
Out[26]= True

In[27]:= WeaklySuperAdditiveQ[ExpGame4]
Out[27]= True
```

Certainly, we can also verify on monotonicity and zero-monotonicity of the game. The commands are **GameMonotoneQ[], MonotoneQ[]** and **ZeroMonotoneQ[]**.

```
In[28]:= GameMonotoneQ[ExpGame1]
Out[28]= True

In[29]:= MonotoneQ[ExpGame1]
Out[29]= True

In[30]:= ZeroMonotoneQ[ExpGame1]
Out[30]= True

In[31]:= GameMonotoneQ[ExpGame2]
Out[31]= True

In[32]:= MonotoneQ[ExpGame2]
Out[32]= True

In[33]:= ZeroMonotoneQ[ExpGame2]
Out[33]= True

In[34]:= GameMonotoneQ[ExpGame3]
Out[34]= False

In[35]:= MonotoneQ[ExpGame3]
Out[35]= False

In[36]:= ZeroMonotoneQ[ExpGame3]
Out[36]= False

In[37]:= GameMonotoneQ[ExpGame4]
Out[37]= True

In[38]:= MonotoneQ[ExpGame4]
Out[38]= True
```

To demonstrate that the package *TuGames* works properly together with *CooperativeGames* let us now call the command **CoreQ[]** written by Carter.

```
In[39]:= CoreQ[ExpGame1]
Out[39]= True

In[40]:= CoreQ[ExpGame2]
Out[40]= True

In[41]:= CoreQ[ExpGame3]
Out[41]= True

In[42]:= CoreQ[ExpGame4]
```

```
Out[42]= True
```

Now determine the quotas of the games

```
In[43]:= Quota[ExpGame1]
Out[43]= {80/3, -40/3, 80/3, -40/3}

In[44]:= Quota[ExpGame2]
Out[44]= {80/3, -40/3, 80/3, -40/3}

In[45]:= Quota[ExpGame3]
Out[45]= {0, 0, 0, 0, 0}

In[46]:= Quota[ExpGame4]
Out[46]= {1/6, 1/6, 1/6, 1/6}
```

4.2 k-convexity

In this section we present some basic functions related to the notion of *k*-convexity that has been introduced by Driessen. The usefulness of this notion stems from the fact that TU-games with this property having similar properties like convex games, although convexity does not imply *k*-convexity, and the converse is also not true in general. There is just a small subclass of games that satisfy both properties, for instance, bankruptcy games are convex as well as *k*-convex. The package *TuGames* provides some useful functions to determine properties of *k*-convex games. To demonstrate some of these functions, let us define to this end four new games. The first three games have been borrowed from Driessen (1988), whereas two of them are 2-convex (Exp. 4.5 on page 198 and Exp 5.3 on page 202) and one of them is 1-convex "*ExpGame7*" (cf. Exp. 5.6 on page 75). The last game presented in the series below is discussed in Meinhardt (2002) and satisfies also the property of 2-convexity.

```
In[47]:= ExpGame5 := (DefineGame[T2, Table[0, {i, 2^Length[T2]}]];
             v[{1, 2}] = v[{1, 3}] = 1; v[{2, 3}] = 4; v[{2, 4}] = 6;
             v[{1, 4}] = 3; v[{3, 4}] = 7; v[{1, 2, 3}] = 9; v[{1, 2, 4}] = 12;
             v[{ 1, 3, 4}] = 13; v[{2, 3, 4}] = 15; v[T] = 20; );

In[48]:= ExpGame6 := (DefineGame[T2, Table[0, {i, 2^Length[T2]}]];
             v[{1, 2, 3}] = v[{1, 2, 4}] = v[{1, 3, 4}] = v[{2, 3, 4}] = 3;
             v[T] = 5; );

In[49]:= T4 = {1, 2, 3};

          vec4 = Table[0, {i, 2^Length[T4]}];

In[50]:= ExpGame7 := (DefineGame[T4, Table[0, {i, 2^Length[T4]}]];
             v[{2}] = v[{3}] = 6; v[{1, 2}] = v[{1, 3}] = 9; v[{2, 3}] = 15;
             v[T] = 18; );
```

As mentioned above you could assign the values of the characteristic function directly by constructing a vector of length 2^nsimply by

```
In[51]:= coalval = {0, 26.7, 26.7, 26.7, 26.7, 800/3, 800/3, 800/3, 800/3,
              800/3, 800/3, 1600/3, 1600/3, 1600/3, 1600/3, 800};
```

The order of the coalitions is obtained by the functions

```
In[52]:= Subsets[T2]
Out[52]= {{}, {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}}
```

To define a game you need no more than the player set *T2* and the vector of the values of the game, that is, *coalval*.

```
In[53]:= Options[DefineGame]
Out[53]= {RationalApproximate → True}

In[54]:= ExpGame8 := (DefineGame[T2, coalval]);
```

Let us first check whether some of them have an empty or non-empty core.

```
In[55]:= CoreQ[ExpGame5]
Out[55]= True

In[56]:= CoreQ[ExpGame6]
Out[56]= True
```

Let $k \in \mathbb{N}$. Note that an n-person game v_k is called the k-cover of the game v if the original game v satisfies the condition

$$(1) \quad g^v(S) \geq g^v(N) \quad \text{for all} \quad S \subset N \text{ with } |S| \geq k, \quad \text{(cf. Driessen 1988, pp. 173)}$$

where $g^v(S) := \sum_{j \in S} b_j^v - v(S)$ and $b_j^v := v(N) - v(N \backslash i)$. A TU-game v_k is defined as follows

$$(2)\ v_k(S) := v(S) \text{ if } |S| < k \text{ and } v_k(S) := \sum_{j \in S} b_j^v - g^v(N) \text{ if } |S| \geq k.$$

A TU-game v is called k-convex if the k-cover of the game is convex, that is, the condition (1) is fulfilled and the TU-game v_k is convex. The k-cover of the game v can be verified by calling the function **Gap[]**. The output of the function **Gap[]** is the value of $g^v(S)$ for each coalition in the order set by the command **Subsets[]**. In the example given below the coalition size two starts at position six whereas coalition size three starts at position twelve.

```
In[57]:= Gap[ExpGame5]
Out[57]= {0, 5, 7, 8, 11, 11, 12, 13, 11, 12, 12, 11, 11, 11, 11, 11}

In[58]:= Gap[ExpGame6]
Out[58]= {0, 2, 2, 2, 2, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3}

In[59]:= Gap[ExpGame7]
Out[59]= {0, 3, 3, 3, 3, 3, 3, 3}

In[60]:= Gap[ExpGame8]
Out[60]= {0, 7199/30, 7199/30, 7199/30, 7199/30, 800/3, 800/3,
           800/3, 800/3, 800/3, 800/3, 800/3, 800/3, 800/3, 800/3, 800/3}

In[61]:= Gap[ExpGame8]//N
Out[61]= {0., 239.967, 239.967, 239.967, 239.967,
           266.667, 266.667, 266.667, 266.667, 266.667,
           266.667, 266.667, 266.667, 266.667, 266.667, 266.667}
```

Game "*ExpGame7*" satisfies the condition of a 1-cover, whereas for the other games the condition of a 2-cover is satisfied. In the next step, we have to verify that the associated v_k game is convex. By doing so, we can derive the associated v_k game by using the function **kCover[]** and by setting *k=2*. First, we have to generate the coalitional values of the v_2 games, this is done by calling

```
In[62]:= kval1 = kCover[ExpGame5, 2]
Out[62]= {0, 0, 0, 0, 0, 1, 2, 5, 4, 7, 8, 9, 12, 13, 15, 20}

In[63]:= kval2 = kCover[ExpGame6, 2]
Out[63]= {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 5}

In[64]:= kval3 = kCover[ExpGame7, 1]
Out[64]= {0, 0, 6, 6, 9, 9, 15, 18}
```

```
In[65]:= kval4 = kCover[ExpGame8, 2]
Out[65]= {0, 267/10, 267/10, 267/10, 267/10, 800/3, 800/3, 800/3,
          800/3, 800/3, 800/3, 1600/3, 1600/3, 1600/3, 1600/3, 800}
```

In the second step we define new TU-games v_2 for "*ExpGame5*, *ExpGame6, ExpGame7* and *ExpGame8*" respectively, by the function **DefineGame[]** and by using the associated vectors of coalitional values which we have determined in the previous step. These games are defined simply by typing

```
In[66]:= Gamevk1 := (DefineGame[T2, kval1]);

In[67]:= Gamevk2 := (DefineGame[T2, kval2]);

In[68]:= Gamevk3 := (DefineGame[T4, kval3]);

In[69]:= Gamevk4 := (DefineGame[T2, kval4]);
```

We have already found out that the original games satisfy condition (1), hence, it remains to check whether the corresponding games v_k are convex to conclude that the games under considerations are 2-convex. To this end, we call the function **ConvexQ[].**

```
In[70]:= ConvexQ[Gamevk1]
Out[70]= True

In[71]:= ConvexQ[ExpGame5]
Out[71]= False

In[72]:= ConvexQ[Gamevk2]
Out[72]= True

In[73]:= ConvexQ[ExpGame6]
Out[73]= False

In[74]:= ConvexQ[Gamevk3]
Out[74]= True

In[75]:= ConvexQ[ExpGame7]
Out[75]= True

In[76]:= ConvexQ[Gamevk4]
Out[76]= True

In[77]:= ConvexQ[ExpGame8]
Out[77]= True
```

Since the condition (1) and the convexity condition are fulfilled, all games under consideration are *k*-convex. Exemplary, we demonstrate that the game "*ExpGame5*" is not 1-convex.

```
In[78]:= kval5 = kCover[ExpGame5, 1]
Out[78]= {0, -6, -4, -3, 0, 1, 2, 5, 4, 7, 8, 9, 12, 13, 15, 20}

In[79]:= Gamevk5 := (DefineGame[T2, kval5]);

In[80]:= ConvexQ[ExpGame5]
Out[80]= False

In[81]:= ConvexQ[Gamevk3]
Out[81]= True
```

The convexity condition holds, but recall that the *k*-cover of the game "*ExpGame5*" is not an 1-cover, it is just a 2-cover.

```
In[82]:= Gap[ExpGame5]
```

```
Out[82]= {0, 5, 7, 8, 11, 11, 12, 13, 11, 12, 12, 11, 11, 11, 11, 11}
```

In this case condition (1) does not hold and therefore the game "*ExpGame5*" is not 1-convex.

Furthermore, we can calculate the upper or utopia payoff that represents an upper bound of the core by

```
In[83]:= UtopiaVector[ExpGame5]
Out[83]= {5, 7, 8, 11}

In[84]:= UtopiaVector[ExpGame6]
Out[84]= {2, 2, 2, 2}
```

A lower bound of the core is the so-called disagreement vector and it can be computed by calling

```
In[85]:= Disagreement[ExpGame5]
Out[85]= {0, 0, 0, 0}

In[86]:= Disagreement[ExpGame6]
Out[86]= {0, 0, 0, 0}
```

Since the tau-value can be regarded as some fair compromise between the utopia and the disagreement vector, we are now in the position to compute the tau-value of the previous games.

```
In[87]:= tau1 = TauValue[ExpGame5]
Out[87]= {100/31, 140/31, 160/31, 220/31}

In[88]:= tau2 = TauValue[ExpGame6]
Out[88]= {5/4, 5/4, 5/4, 5/4}
```

In the next step, we want to verify if the computed tau-values are contained in the corresponding cores of the game "*ExpGame5* and *ExpGame6*", respectively. For this purpose let us call the command **InCoreQ[]** provided by the package *CooperativeGames.* The command requires as an input one imputation and the *game_name.*

```
In[89]:= InCoreQ[tau1, ExpGame5]
Out[89]= True

In[90]:= InCoreQ[tau2, ExpGame6]
Out[90]= True
```

It is well known that for convexity of an n-person Tu-game it is necessary and sufficient that the core of the game contains all $n!$ marginal worth vectors. A similar result holds for k-convex games. For k-convexity it is necessary and sufficient that all $((n-k)!)^{-1}\, n!$ adjusted marginal worth vectors are contained in the core of the game (cf. Driessen 1988). The adjusted marginal worth vectors of k-convex games can be evaluated with the function **AdjustedWorthVectors[]** . There is also a special function available to compute the adjusted worth vectors of 1-convex games. The function name in this case is **AdjustedEffUpperVectors[]**. Let us now calculate for the above games the adjusted marginal worth vectors.

```
In[91]:= v5 = AdjustedWorthVectors[ExpGame5, 2]
Out[91]= {{0, 1, 8, 11}, {0, 1, 8, 11}, {0, 7, 2, 11}, {0, 7, 8, 5}, {0, 7, 2, 11},
           {0, 7, 8, 5}, {1, 0, 8, 11}, {1, 0, 8, 11}, {2, 7, 0, 11},
           {5, 7, 8, 0}, {2, 7, 0, 11}, {5, 7, 8, 0}, {5, 0, 4, 11}, {5, 0, 8, 7},
           {5, 4, 0, 11}, {5, 7, 8, 0}, {5, 7, 0, 8}, {5, 7, 8, 0}, {5, 0, 4, 11},
           {5, 0, 8, 7}, {5, 4, 0, 11}, {5, 7, 8, 0}, {5, 7, 0, 8}, {5, 7, 8, 0}}

In[92]:= u5 = v5//Union
Out[92]= {{0, 1, 8, 11}, {0, 7, 2, 11}, {0, 7, 8, 5}, {1, 0, 8, 11}, {2, 7, 0, 11},
           {5, 0, 4, 11}, {5, 0, 8, 7}, {5, 4, 0, 11}, {5, 7, 0, 8}, {5, 7, 8, 0}}

In[93]:= Length[u5]
Out[93]= 10
```

As expected the number of marginal worth vectors does not exceed the threshold value of $((n-k)!)^{-1}\, n!$ vertices. For 2-convex games the maximal number of marginal worth vectors is 12. All adjusted marginal worth vectors must be contained in the core. We check this by

```
In[94]:= CoreElementsQ[ExpGame5, u5]
Out[94]= {True, True, True, True, True, True, True, True, True, True}
```

As well as for the class of convex games the class of k-convex games can be characterized by the vertices of the core, it is necessary and sufficient for the k-convexity property that the vertices of the core coincide exactly with the adjusted marginal worth vectors. We are able to check on this property by using the following steps: first, we compute the vertices of the core. In a second step we find out the imputations that are contained in the core and examine if the imputations we have selected are really core elements. Then we compute the adjusted marginal worth vectors and compare the result with our vertex result. To compute core vertices, we can perform this task with the command **VerticesCore[]**. The command returns two output values, the vertices and a list of "active-variable" sets, indicating which inequalities are satisfied with equalities at each vertex (cf. *VertexEnum*). In our example below, the symbol *cvert5* captures the informations associated with the vertices and the symbol *nonvert5* grabs all informations of the "active -variable" set.

```
In[95]:= {cvert5, nonvert5} = VerticesCore[ExpGame5]
Out[95]= {{{5, 7, 8, 0}, {3, 9, 10, 0}, {6, 6, 9, 0}, {6, 8, 7, 0}, {0, 9, 10, 3},
            {6, 0, 9, 6}, {6, 8, 0, 7}, {5, 7, 0, 8}, {2, 7, 0, 11}, {1, 8, 0, 12},
            {5, 4, 0, 11}, {0, 7, 8, 5}, {0, 7, 2, 11}, {0, 8, 1, 12},
            {0, 1, 8, 11}, {5, 0, 8, 7}, {5, 0, 4, 11}, {1, 0, 8, 11}},
          {{16, 18, 19, 17, 8, 4}, {16, 11, 8, 17, 4}, {16, 18, 8, 13, 4},
            {14, 18, 8, 17, 4}, {16, 11, 5, 17, 1}, {16, 18, 6, 13, 2},
            {14, 18, 7, 17, 3}, {7, 18, 19, 17, 3}, {7, 15, 19, 17, 3},
            {7, 15, 10, 17, 3}, {7, 18, 19, 15, 3, 12}, {16, 5, 19, 17, 1},
            {15, 5, 19, 17, 1}, {15, 5, 10, 17, 1}, {16, 5, 19, 15, 9, 1},
            {16, 18, 19, 6, 2}, {15, 18, 19, 6, 2, 12}, {16, 15, 19, 6, 2, 9}}}
```

The package provides two different functions to check whether a list of imputations is contained in the core. In addition to the function **CoreElementsQ[]** which we have already introduced above and that is based on the function **InCoreQ[]** by Carter, we provide a second function that is called **BelongToCoreQ[]** which is based on the property that an imputation x belongs to the core $C(v)$ iff

$$x(S) = \sum_{i \in S} x_i \;\; \leq \;\; v(N) \; - \; v(S) \quad \text{for all } S \subset N.$$

We demonstrate how the function **BelongToCoreQ[]** works in comparison to the function **CoreElementsQ[]** before we proceed on our problem. The difference between the function is discussed in more detail in Section 5.

```
In[96]:= cor5 = CoreElementsQ[ExpGame5, cvert5]
Out[96]= {True, False, False, False, False, False, False, True,
           True, False, True, True, True, False, True, True, True, True}

In[97]:= blcor5 = BelongToCoreQ[ExpGame5, cvert5]
Out[97]= {True, False, False, False, False, False, False, True,
           True, False, True, True, True, False, True, True, True, True}
```

Since, some of imputations aren't contained in the core, we select the core imputations by using the command

```
In[98]:= pos5 = Position[cor5, True]
Out[98]= {{1}, {8}, {9}, {11}, {12}, {13}, {15}, {16}, {17}, {18}}

In[99]:= bcs5 = Position[blcor5, True]
Out[99]= {{1}, {8}, {9}, {11}, {12}, {13}, {15}, {16}, {17}, {18}}
```

We got with the command **Position[]** the exact location of the core elements, we can use this information by extracting all core imputations typing

```
In[100]:= ext5 = Extract[cvert5, pos5]
Out[100]= {{5, 7, 8, 0}, {5, 7, 0, 8}, {2, 7, 0, 11}, {5, 4, 0, 11}, {0, 7, 8, 5},
            {0, 7, 2, 11}, {0, 1, 8, 11}, {5, 0, 8, 7}, {5, 0, 4, 11}, {1, 0, 8, 11}}
```

```
In[101]:= bcext5 = Extract[cvert5, bcs5]
Out[101]= {{5, 7, 8, 0}, {5, 7, 0, 8}, {2, 7, 0, 11}, {5, 4, 0, 11}, {0, 7, 8, 5},
            {0, 7, 2, 11}, {0, 1, 8, 11}, {5, 0, 8, 7}, {5, 0, 4, 11}, {1, 0, 8, 11}}
```

Let us verify that all elements we have selected from our initial list are contained in the core.

```
In[102]:= CoreElementsQ[ExpGame5, ext5]
Out[102]= {True, True, True, True, True, True, True, True, True, True}

In[103]:= BelongToCoreQ[ExpGame5, bcext5]
Out[103]= {True, True, True, True, True, True, True, True, True, True}

In[104]:= Length[ext5]
Out[104]= 10

In[105]:= to5 = Union[ext5, u5]
Out[105]= {{0, 1, 8, 11}, {0, 7, 2, 11}, {0, 7, 8, 5}, {1, 0, 8, 11}, {2, 7, 0, 11},
            {5, 0, 4, 11}, {5, 0, 8, 7}, {5, 4, 0, 11}, {5, 7, 0, 8}, {5, 7, 8, 0}}

In[106]:= Length[to5]
Out[106]= 10
```

To get more evidence that the function **AdjustedWorthVectors[]** computes the exact values. Let us consider the game "*ExpGame8*".

```
In[107]:= v8 = AdjustedWorthVectors[ExpGame8, 2]
Out[107]= {{267/10, 7199/30, 800/3, 800/3}, {267/10, 7199/30, 800/3, 800/3},
            {267/10, 800/3, 7199/30, 800/3}, {267/10, 800/3, 800/3, 7199/30},
            {267/10, 800/3, 7199/30, 800/3}, {267/10, 800/3, 800/3, 7199/30},
            {7199/30, 267/10, 800/3, 800/3}, {7199/30, 267/10, 800/3, 800/3},
            {7199/30, 800/3, 267/10, 800/3}, {7199/30, 800/3, 800/3, 267/10},
            {7199/30, 800/3, 267/10, 800/3}, {7199/30, 800/3, 800/3, 267/10},
            {800/3, 267/10, 7199/30, 800/3}, {800/3, 267/10, 800/3, 7199/30},
            {800/3, 7199/30, 267/10, 800/3}, {800/3, 7199/30, 800/3, 267/10},
            {800/3, 800/3, 267/10, 7199/30}, {800/3, 800/3, 7199/30, 267/10},
            {800/3, 267/10, 7199/30, 800/3}, {800/3, 267/10, 800/3, 7199/30},
            {800/3, 7199/30, 267/10, 800/3}, {800/3, 7199/30, 800/3, 267/10},
            {800/3, 800/3, 267/10, 7199/30}, {800/3, 800/3, 7199/30, 267/10}}
```

Removing all duplicated results we obtain

```
In[108]:= u8 = v8//Union
Out[108]= {{267/10, 7199/30, 800/3, 800/3}, {267/10, 800/3, 7199/30, 800/3},
            {267/10, 800/3, 800/3, 7199/30}, {7199/30, 267/10, 800/3, 800/3},
            {7199/30, 800/3, 267/10, 800/3}, {7199/30, 800/3, 800/3, 267/10},
            {800/3, 267/10, 7199/30, 800/3}, {800/3, 267/10, 800/3, 7199/30},
            {800/3, 7199/30, 267/10, 800/3}, {800/3, 7199/30, 800/3, 267/10},
            {800/3, 800/3, 267/10, 7199/30}, {800/3, 800/3, 7199/30, 267/10}}
```

We obtain the expected number of adjusted marginal worth vectors, since

```
In[109]:= Length[u8]
Out[109]= 12
```

For an intermediate step, let us verify that the computed vectors are all efficient. This could be done by

```
In[110]:= Apply[Plus, #] & /@ u8
Out[110]= {800, 800, 800, 800, 800, 800, 800, 800, 800, 800, 800, 800}
```

```
In[111]:= {cvert8, nonvert8} = VerticesCore[ExpGame8]
Out[111]= {{{800/3, 800/3, 267/10, 7199/30}, {800/3, 800/3, 7199/30, 267/10},
             {7199/30, 800/3, 800/3, 267/10}, {267/10, 800/3, 800/3, 7199/30},
             {267/10, 7199/30, 800/3, 800/3}, {7199/30, 800/3, 267/10, 800/3},
             {267/10, 800/3, 7199/30, 800/3}, {7199/30, 267/10, 800/3, 800/3},
             {800/3, 7199/30, 800/3, 267/10}, {800/3, 267/10, 800/3, 7199/30},
             {800/3, 7199/30, 267/10, 800/3}, {800/3, 267/10, 7199/30, 800/3}},
           {{19, 18, 7, 17, 14}, {19, 18, 8, 17, 14}, {19, 16, 8, 17, 11},
            {19, 16, 5, 17, 11}, {19, 15, 5, 16, 9}, {19, 15, 7, 17, 10},
            {19, 15, 5, 17, 10}, {19, 16, 6, 15, 9}, {19, 18, 8, 16, 13},
            {19, 18, 6, 16, 13}, {19, 18, 7, 15, 12}, {19, 18, 6, 15, 12}}}
```

```
In[112]:= Length[cvert8]
Out[112]= 12
```

```
In[113]:= CoreElementsQ[ExpGame8, cvert8]
Out[113]= {True, True, True, True, True,
           True, True, True, True, True, True, True}
```

```
In[114]:= BelongToCoreQ[ExpGame8, cvert8]
Out[114]= {True, True, True, True, True,
           True, True, True, True, True, True, True}
```

```
In[115]:= to8 = Union[cvert8, v8]
Out[115]= {{267/10, 7199/30, 800/3, 800/3}, {267/10, 800/3, 7199/30, 800/3},
           {267/10, 800/3, 800/3, 7199/30}, {7199/30, 267/10, 800/3, 800/3},
           {7199/30, 800/3, 267/10, 800/3}, {7199/30, 800/3, 800/3, 267/10},
           {800/3, 267/10, 7199/30, 800/3}, {800/3, 267/10, 800/3, 7199/30},
           {800/3, 7199/30, 267/10, 800/3}, {800/3, 7199/30, 800/3, 267/10},
           {800/3, 800/3, 267/10, 7199/30}, {800/3, 800/3, 7199/30, 267/10}}
```

```
In[116]:= Length[to8]
Out[116]= 12
```

Do not try to use the function **AdjustedEffUpperVectors[]** for k-convex games, with $k \geq 2$. In this case you will get wrong results.

```
In[117]:= AdjustedEffUpperVectors[ExpGame8]
Out[117]= {{0, 800/3, 800/3, 800/3}, {800/3, 0, 800/3, 800/3},
           {800/3, 800/3, 0, 800/3}, {800/3, 800/3, 800/3, 0}}
```

You can just use this function simply to compute the adjusted marginal worth vectors for 1-convex games as the next example will demonstrate it.

```
In[118]:= AdjustedEffUpperVectors[ExpGame7]
Out[118]= {{0, 9, 9}, {3, 6, 9}, {3, 9, 6}}

In[119]:= AdjustedWorthVectors[ExpGame7, 1]//Union
Out[119]= {{0, 9, 9}, {3, 6, 9}, {3, 9, 6}}

In[120]:= VerticesCore[ExpGame7][[1]]
Out[120]= {{3, 9, 6}, {0, 9, 9}, {3, 6, 9}}
```

4.3 Kernel

In this subsection we want to present some functions which will enable us to compute at least a kernel solution or in good cases the whole kernel of a game. The functions we will present in this subsection relying on the linear programming approach to determine outcomes that lie in the core (or strong epsilon core) as well as in the kernel of the game, that is, outcomes that satisfy the bisection property. Note, that for zero-monotonic games the bisection property is necessary and sufficient that an imputation that belongs to the core (strong-epsilon core) is also contained in the kernel of the game (cf. Th 3.7 Maschler, Peleg and Shapley (1979)). More informations about the algorithm can be found in Meinhardt (2004). To start off, let us first consider the command **Kernel[game,opts]** without an option to compute a kernel point of the game. In the second case we make use of an optional parameter *EpsilonValue* which is a critical value to generalize the bisection property to the strong epsilon-cores. You can omit the critical value *EpsilonValue* in the command without harm, since the default value is set to zero. There are different functions available which determine the different types of critical values. One critical value can be calculated, for instance, by the function **FirstCriticalVal[game]**. This function and all the rest of them will be discussed in the sequel of this subsection.

```
In[121]:= ker1 = Kernel[ExpGame1]
Game has nonempty core
Out[121]= {45/2, 45/2, 45/2, 45/2}

In[122]:= ker2 = Kernel[ExpGame2]
Game has nonempty core
Out[122]= {45/2, 45/2, 45/2, 45/2}

In[123]:= ModifiedKernel[ExpGame2]
Out[123]= {45/2, 45/2, 45/2, 45/2}

In[124]:= ker3 = Kernel[ExpGame3]
Game has nonempty core
Out[124]= {1, 1, 1, 2, 2}

In[125]:= ModifiedKernel[ExpGame3]
Out[125]= {1, 1, 1, 2, 2}

In[126]:= ker4 = Kernel[ExpGame4]
Game has nonempty core
Out[126]= {7/32, 7/32, 9/32, 9/32}

In[127]:= ModifiedKernel[ExpGame4]
Out[127]= {7/32, 7/32, 9/32, 9/32}
```

The next function verifies whether the payoff belongs to the kernel of the game.

```
In[128]:= KernelImputationQ[ExpGame1, ker1]
Out[128]= True
```

```
In[129]:= KernelImputationQ[ExpGame2, ker2]
Out[129]= True

In[130]:= KernelImputationQ[ExpGame3, ker3]
Out[130]= True

In[131]:= KernelImputationQ[ExpGame4, ker4]
Out[131]= True
```

Define some payoffs

```
In[132]:= pay = {{45, 0, 45, 0}, {10, 20, 30, 30}, ker1, ker2};
```

If you want to test, if a list of payoffs belong to the kernel then use

```
In[133]:= KernelImputationListQ[ExpGame1, pay]
Out[133]= {False, False, True, True}
```

Specify now the arrangement of all subsets of the grand coalition N.

```
In[134]:= Subsets[{1, 2, 3, 4}]
Out[134]= {{}, {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}}
```

The surpluses of an excess vector are arranged exactly in accordance with the above order of subsets of the grand coalition N. Here, we present the shortened corresponding excess vectors related to the symbol "*pay*".

```
In[135]:= ExcessPayoff[ExpGame1, pay, DisplayMatrixForm → True]
```

Co[{}]	Co[{1}]	Co[{2}]	Co[{3}]	Co[{4}]	Co[{1,2}]	Co[{1,3}]	Co[{1,4}]	…	Co[{1,2,3,4}]
0	-45	0	-45	0	-45	-90	-45	…	0
0	-10	-20	-30	-30	-30	-40	-40	…	0
0	$-\frac{45}{2}$	$-\frac{45}{2}$	$-\frac{45}{2}$	$-\frac{45}{2}$	-45	-45	-45	…	0
0	$-\frac{45}{2}$	$-\frac{45}{2}$	$-\frac{45}{2}$	$-\frac{45}{2}$	-45	-45	-45	…	0

To check if the kernel element coincides with the nucleolus, we use the function **Nucleolus[]** provided by *CooperativeGames.*

```
In[136]:= Nucleolus[ExpGame1]
Out[136]= {45/2, 45/2, 45/2, 45/2}
```

In the package *TuGames* are also some functions available to compute the nucleolus or pre-nucleolus of the game. These are the following functions:

```
In[137]:= ModifiedNucleolus[ExpGame1]
Out[137]= {45/2, 45/2, 45/2, 45/2}

In[138]:= LexiCenter[ExpGame1]
Out[138]= {45/2, 45/2, 45/2, 45/2}

In[139]:= PreNucleolus[ExpGame1]
Out[139]= {45/2, 45/2, 45/2, 45/2}
```

The Options can be called with the following command.

```
In[140]:= Options[ModifiedNucleolus]
Out[140]= {CallMinimize → True, SetRecursionLimit → 128, Silent → True}
```

Due to our experience these functions are more robust than the function **Nucleolus[]** for essential games. If the game is inessential then the result might not be correct and you have to check the result by **KernelImputationListQ[]** or **KernelImputationQ[]**. Since the result is a remapping of a zero-one normalized game back to original game, one has simply to look on this result to find an appropriate remapping to original game. The game "*ExpGame1*" is convex, in this case the kernel is an unique element that coincides with the Nucleolus.

```
In[141]:= ConvexQ[ExpGame1]
```

```
Out[141]= True
```

Recall the kernel element from "*ExpGame3*"

```
In[142]:= ker3
Out[142]= {1, 1, 1, 2, 2}

In[143]:= nuc3 = Nucleolus[ExpGame3]
Out[143]= {1, 1, 1, 2, 2}

In[144]:= ModifiedNucleolus[ExpGame3]
Out[144]= {1, 1, 1, 2, 2}

In[145]:= ModifiedNucleolus[ExpGame3, CallMinimize → False]
Out[145]= {1, 1, 1, 2, 2}
```

If you are interested to find out more kernel elements the package provides the command **KernelCalculation[]**. The function **KernelCalculation[]** computes a kernel element and candidates from $(n(n-1)/2+1)$ LPs, whereas **Kernel[]** computes a kernel element from just one linear maximization problem. But due to the restriction that the built-in function **ConstrainedMax[]** find just one solution and not all solutions of a LP we are sometimes not able to specify the whole kernel with *MATHEMATICA*®. That's why we follow the strategy that the above commands return also the final LP for further investigation outside of *MATHEMATICA*®.

```
In[146]:= {sol3, pay3} = KernelCalculation[ExpGame3, EpsilonValue → 2]
Game is zero - monotone? False
Core is nonempty? True
Game is either zero - monotonic or has nonempty core
A Kernel solution is : {x[1] → 1, x[2] → 1, x[3] → 1, x[4] → 2, x[5] → 2}
Out[146]= {{1, 1, 1, 2, 2}, {{1/6, 4/3, 4/3, 3/2, 8/3}, {1/6, 4/3, 4/3, 8/3, 3/2},
             {1, 1, 1, 2, 2}, {1, 1, 3, 1, 1}, {1, 3, 1, 1, 1},
             {4/3, 1/6, 4/3, 3/2, 8/3}, {4/3, 1/6, 4/3, 8/3, 3/2}, {4/3, 4/3, 1/6, 3/2, 8/3},
             {4/3, 4/3, 1/6, 8/3, 3/2}, {5/3, 5/3, 5/3, 1, 1}, {3, 1, 1, 1, 1}}}

In[147]:= KernelImputationListQ[ExpGame3, pay3]
Out[147]= {False, False, True, False, False,
            False, False, False, False, False, False}
```

All internal results can be obtained by invoking the option *DisplayAllResults.*

```
In[148]:= {sol3, obj3, con3, tra3, pay3} =
            KernelCalculation[ExpGame3, EpsilonValue → 2,
              DisplayAllResults → True]
Game is zero - monotone? False
Core is nonempty? True
Game is either zero - monotonic or has nonempty core
A Kernel solution is : {x[1] → 1, x[2] → 1, x[3] → 1, x[4] → 2, x[5] → 2}
Out[148]= {{1, 1, 1, 2, 2}, x[1] + x[2] + x[3] + x[4] + x[5],
            {-2 - x[1] ≤ -2, -2 - x[2] ≤ -2, -2 - x[3] ≤ -2,
              2 - x[1] - x[2] - x[4] ≤ -2, 2 - x[1] - x[3] - x[4] ≤ -2,
              2 - x[2] - x[3] - x[4] ≤ -2, 2 - x[1] - x[2] - x[5] ≤ -2,
              2 - x[1] - x[3] - x[5] ≤ -2, 2 - x[2] - x[3] - x[5] ≤ -2,
              2 - x[4] - x[5] ≤ -2, x[1] + x[2] + x[3] + x[4] + x[5] ≤ 7},
            {1, 1, 1/6, 1/6, 1, 1/6, 1/6, 1/6, 1/6, 1/3}, {{1/6, 4/3, 4/3, 3/2, 8/3},
              {1/6, 4/3, 4/3, 8/3, 3/2}, {1, 1, 1, 2, 2}, {1, 1, 3, 1, 1}, {1, 3, 1, 1, 1},
              {4/3, 1/6, 4/3, 3/2, 8/3}, {4/3, 1/6, 4/3, 8/3, 3/2}, {4/3, 4/3, 1/6, 3/2, 8/3},
              {4/3, 4/3, 1/6, 8/3, 3/2}, {5/3, 5/3, 5/3, 1, 1}, {3, 1, 1, 1, 1}}}
```

The first return value is the kernel solution, the second the objective function, the third the constraint set, the fourth contains values of the largest bi-symmetrical transfer and the last value returns in general a list of possible kernel candidates that have been computed from the $n(n-1)/2$ initial LPs. Note, that according to this procedure the function becomes very slow for huge games. A good strategy is to find out a first kernel element by the function **Kernel[]**. This function is in average 10 times faster than the function **KernelCalculation[]**. Moreover, we have found out that you can also speed up the calculation with the above commands by providing one of the critical value that have been introduced by Maschler, Peleg and Shapley (1979) to vary the strong epsilon core instead of using the default value zero.

The complete set of options can be seen by

```
In[149]:= Options[KernelCalculation]
Out[149]= {ChangeInternalEps → False, DisplayAllResults → False,
            EpsilonValue → 0, SetGameToNonZeroMonotonic → False}
```

and for the function **Kernel[]** by calling

```
In[150]:= Options[Kernel]
Out[150]= {DisplayAllResults → False, EpsilonValue → 0}
```

Now have a (shortened) look what returns the function **Kernel[]** by invoking the option *DisplayAllResults*

```
In[151]:= {sol3, object3, con3, var3, trans3} =
            Kernel[ExpGame3, DisplayAllResults → True]
Game has nonempty core
Out[151]= {{1, 1, 1, 2, 2}, δ[1, 2] + δ[1, 3] + δ[1, 4] + δ[1, 5]+
             δ[2, 3] + δ[2, 4] + δ[2, 5] + δ[3, 4] + δ[3, 5] + δ[4, 5],
           {x[1] + x[2] + x[3] + x[4] + x[5] == 7, x[1] ≥ 0, x[2] ≥ 0, x[1] + x[2] ≥ 0,
             x[1] + x[2] + x[5] - δ[4, 5] ≥ 4, x[3] + x[5] - δ[4, 5] ≥ 0,
             x[1] + x[3] + x[5] - δ[4, 5] ≥ 4, x[2] + x[3] + x[5] - δ[4, 5] ≥ 4,
             x[1] + x[2] + x[3] + x[5] - δ[4, 5] ≥ 0, δ[4, 5] ≥ 0},
             …
             …
           {x[1], x[2], x[3], x[4], x[5], δ[1, 2], δ[1, 3], δ[1, 4],
             δ[1, 5], δ[2, 3], δ[2, 4], δ[2, 5], δ[3, 4], δ[3, 5], δ[4, 5]},
           {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}}
```

The first three return values are the same as above. The last but one return value stores all variables of the LP and the last provides information about the transfers in each direction such that the endpoints of the line segment remain in the strong epsilon-core.

```
In[152]:= trans3
Out[152]= {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

The result states that it is not possible to transfer a positive amount in each direction without leaving the core. To check this, let us determine, if an endpoint of the line segment 1-2 belongs to the core if we transfer a half unit from player 1 to player 2 from the derived kernel solution. The function **Transfer[]** needs five input parameters, the initial imputation, the whole player set, the transfer amount, the player from whom we extract the transfer amount and the player who receives it.

```
In[153]:= tr1 = Transfer[sol3, T3, 1/2, 1, 2]
Out[153]= {1/2, 3/2, 1, 2, 2}
In[154]:= InCoreQ[tr1, ExpGame3]
Out[154]= False
```

Next let us remain in the initial payoff distribution.

```
In[155]:= tr2 = Transfer[sol3, T3, 0, 1, 2]
Out[155]= {1, 1, 1, 2, 2}
```

```
In[156]:= InCoreQ[tr2, ExpGame3]
Out[156]= True
```

In the last example we take from the initial distribution a half unit away from player 2 to give it player 1.

```
In[157]:= tr3 = Transfer[sol3, T3, -1/2, 1, 2]
Out[157]= {3/2, 1/2, 1, 2, 2}

In[158]:= InCoreQ[tr3, ExpGame3]
Out[158]= False
```

Recall that all kernel commands based on the built-in *MATHEMATICA*® command **ConstrainedMax[]**. Unfortunately, this command offers no control on the solver. That's why, we let us also return the whole final LP that computes us a kernel element with the intention to export this final LP to a mathematical software that offers more control on the solver. In some respect one can overcome this shortcoming by computing kernel elements by varying the strong epsilon core. This can be done by calculating in a first step some critical values. For instance the command **FirstCriticalVal[]** calculates the smallest epsilon value such that the epsilon-core is non-empty (cf. Maschler, Peleg and Shapley (1979)).

```
In[159]:= FirstCriticalVal[ExpGame1]
Out[159]= {eps1 → -45/2}

In[160]:= FirstCriticalVal[ExpGame2]
Out[160]= {eps1 → -45/2}

In[161]:= FirstCriticalVal[ExpGame3]
Out[161]= {eps1 → 0}

In[162]:= FirstCriticalVal[ExpGame4]
Out[162]= {eps1 → -3/16}

In[163]:= SecondCriticalVal[ExpGame3]
Out[163]= {eps2 → 4}

In[164]:= ThirdCriticalVal[ExpGame3]
Out[164]= {eps3 → 21}

In[165]:= FourthCriticalVal[ExpGame3]
Out[165]= {eps4 → 4}

In[166]:= FifthCriticalVal[ExpGame3]
Out[166]= {eps5 → 16}

In[167]:= StarCriticalVal[ExpGame3]
Out[167]= {eps5 → 4}

In[168]:= SecondStarCriticalVal[ExpGame3]
Out[168]= {dstareps → 4}

In[169]:= ThirdStarCriticalVal[ExpGame3]
Out[169]= {thstareps → 0}
```

Now let us demonstrate that we are able to compute the complete kernel of a game.

```
In[170]:= ExpGame9 := (DefineGame[T3, vec3]; v[{2, 5}] = 1; v[{3, 5}] = 1;
            v[{4, 5}] = 1; v[{1, 2, 3}] = 11/12; v[{1, 2, 4}] = 1; v[{1, 2, 5}] = 1;
            v[{1, 3, 4}] = 1; v[{1, 3, 5}] = 1; v[{1, 4, 5}] = 1; v[{2, 3, 4}] = 1;
            v[{2, 3, 5}] = 1; v[{2, 4, 5}] = 1; v[{3, 4, 5}] = 1;
            v[{1, 2, 3, 4}] = 1; v[{1, 2, 3, 5}] = 1; v[{1, 2, 4, 5}] = 1;
            v[{2, 3, 4, 5}] = 1; v[T3] = 2; );
```

The kernel of this game is given by the union of the following three line segments (cf. Maschler and Peleg (1966,p.322)):

$J_1 = co\{\{0, 1, 1, 1, 1\}/2, \{1, 11, 11, 12, 13\}/24\}$

$J_2 = co\{\{7, 8, 8, 9, 16\}/24, \{1, 11, 11, 12, 13\}/24\}$

$J_3 = co\{\{7, 8, 8, 9, 16\}/24, \{11, 11, 11, 14, 25\}/36\}$

Note first that specifying an epsilon value can speed up the calculation time by using the *MATHEMATICA*® function **ConstrainedMax[].**

```
In[171]:= {time9, ker9} = AbsoluteTiming[Kernel[ExpGame9]]
Game has nonempty core
Out[171]= {2.14333 Second, {7/24, 1/3, 1/3, 3/8, 2/3}}

In[172]:= {time091, ker9} = AbsoluteTiming[Kernel[ExpGame9, EpsilonValue → 3]]
Out[172]= {0.568597 Second, {7/24, 1/3, 1/3, 3/8, 2/3}}
```

We can also to try to find out some kernel segment by using the command:

```
In[173]:= KernelVertices[ExpGame9]
Game has nonempty core
Out[173]= {{1/24, 11/24, 11/24, 1/2, 13/24}, {7/24, 1/3, 1/3, 3/8, 2/3}}

In[174]:= {time92, {sol9, pay9}} = AbsoluteTiming[KernelCalculation[ExpGame9]]
Game is average - convex? False
 A Kernel solution is :
   {x[1] → 7/24, x[2] → 1/3, x[3] → 1/3, x[4] → 3/8, x[5] → 2/3}
Out[174]= {12.8581 Second, {{7/24, 1/3, 1/3, 3/8, 2/3},
              {{0, 11/24, 11/24, 13/24, 13/24}, {0, 1/2, 1/2, 1/2, 1/2}, {7/24, 1/3, 1/3, 3/8, 2/3},
                {11/36, 11/36, 11/36, 7/18, 25/36}, {1/3, 1/3, 1/3, 1/3, 2/3}}}}

In[175]:= kerli9 = KernelImputationListQ[ExpGame9, pay9]
Out[175]= {False, True, True, True, False}

In[176]:= ps = Position[kerli9, True]
Out[176]= {{2}, {3}, {4}}

In[177]:= kerel = Extract[pay9, ps]
Out[177]= {{0, 1/2, 1/2, 1/2, 1/2}, {7/24, 1/3, 1/3, 3/8, 2/3}, {11/36, 11/36, 11/36, 7/18, 25/36}}

In[178]:= {time9, {sol91, pay91}} =
            AbsoluteTiming[KernelCalculation[ExpGame9, EpsilonValue → 3]]
Game is zero - monotone? False
Core is nonempty? True
Game is either zero - monotonic or has nonempty core
 A Kernel solution is :
   {x[1] → 1/24, x[2] → 11/24, x[3] → 11/24, x[4] → 1/2, x[5] → 13/24}
Out[178]= {6.25416 Second,
            {{1/24, 11/24, 11/24, 1/2, 13/24}, {{0, 0, 1, 1, 0}, {0, 1/2, 1/2, 1/2, 1/2},
              {0, 1, 0, 1, 0}, {0, 1, 1, 0, 0}, {7/24, 1/3, 1/3, 3/8, 2/3},
              {19/24, 1/12, 1/12, 7/8, 1/6}, {1, 0, 1, 0, 0}, {1, 1, 0, 0, 0}}}}
```

```
In[179]:= AppendTo[kerel, sol91]
Out[179]= {{0, 1/2, 1/2, 1/2, 1/2}, {7/24, 1/3, 1/3, 3/8, 2/3},
           {11/36, 11/36, 11/36, 7/18, 25/36}, {1/24, 11/24, 11/24, 1/2, 13/24}}
```

This is exactly the kernel solution.

Instead of computing kernel solutions with an LP approach, we can also compute a kernel element by relying on a convergence algorithm. We implemented an algorithm that was due to Maschler by iteratively carrying out transfers between pairs of players. Details of the algorithm can be found in Faigle, Kern and Kuipers (1998). Note that the algorithm is implemented by using recursion. The default value is set to 512 and can be changed with the option **SetRecursionLimit**. We invoke the algorithm by calling the function **FindKernelSolution[game,payoff,options]** and the options can be find out by calling

```
In[180]:= Options[FindKernelSolution]
Out[180]= {DigitPrecision → 6, RationalTol → 1/10000000, SetRecursionLimit → 512}
```

To check out how the function works, let us first construct some payoff vectors.

```
In[181]:= c1 = {2, 0, 0, 0, 0};

          c2 = Permutations[c1];
```

Computing a pre-kernel element can be done by

```
In[182]:= {tc1, c1kern1} = AbsoluteTiming[FindKernelSolution[ExpGame9, c1]]
Out[182]= {2.83453 Second, {1/4, 17/48, 17/48, 19/48, 31/48}}
```

or

```
In[183]:= {tc2, c1kern2} = AbsoluteTiming[FindKernelSolution[ExpGame9, c2]]
Out[183]= {12.0065 Second, {{1/4, 17/48, 17/48, 19/48, 31/48},
             {1/4, 17/48, 17/48, 19/48, 31/48}, {1/4, 17/48, 17/48, 19/48, 31/48},
             {1/24, 11/24, 11/24, 1/2, 13/24}, {13/48, 11/32, 11/32, 37/96, 21/32}}}
```

By using the kernel candidates of *pay91* we compute

```
In[184]:= {tc3, c1kern3} = AbsoluteTiming[FindKernelSolution[ExpGame9, pay91]]
Out[184]= {13.1125 Second, {{1/24, 11/24, 11/24, 1/2, 13/24}, {0, 1/2, 1/2, 1/2, 1/2},
             {1/24, 11/24, 11/24, 1/2, 13/24}, {1/4, 17/48, 17/48, 19/48, 31/48},
             {7/24, 1/3, 1/3, 3/8, 2/3}, {9/32, 65/192, 65/192, 73/192, 127/192},
             {1/4, 17/48, 17/48, 19/48, 31/48}, {1/4, 17/48, 17/48, 19/48, 31/48}}}
In[185]:= c2k2 = Union[c1kern2]
Out[185]= {{1/24, 11/24, 11/24, 1/2, 13/24},
           {1/4, 17/48, 17/48, 19/48, 31/48}, {13/48, 11/32, 11/32, 37/96, 21/32}}
In[186]:= c2k3 = Union[c1kern3]
Out[186]= {{0, 1/2, 1/2, 1/2, 1/2},
           {1/24, 11/24, 11/24, 1/2, 13/24}, {1/4, 17/48, 17/48, 19/48, 31/48},
           {9/32, 65/192, 65/192, 73/192, 127/192}, {7/24, 1/3, 1/3, 3/8, 2/3}}
```

Verifying that the computed solutions are really kernel elements can be done by

```
In[187]:= KernelImputationListQ[ExpGame9, #]&/@ {c2k2, c2k3}
Out[187]= {{True, True, True}, {True, True, True, True, True}}
```

For non-zero-monotonic games we are able to compute pre-kernel elements by invoking the function **FindPreKernelSolution[]**. The function works in the same vein as **FindKernelSolution[]**, since they are based on the same algorithm. Therefore, for zero-monotonic games they will compute the same results.

```
In[188]:= {tc4, c1kern4} =
            AbsoluteTiming[FindPreKernelSolution[ExpGame9, pay91]]
Out[188]= {14.0878 Second, {{1/24, 11/24, 11/24, 1/2, 13/24}, {0, 1/2, 1/2, 1/2, 1/2},
             {1/24, 11/24, 11/24, 1/2, 13/24}, {1/4, 17/48, 17/48, 19/48, 31/48},
             {7/24, 1/3, 1/3, 3/8, 2/3}, {9/32, 65/192, 65/192, 73/192, 127/192},
             {1/4, 17/48, 17/48, 19/48, 31/48}, {1/4, 17/48, 17/48, 19/48, 31/48}}}

In[189]:= c1kern3 == c1kern4
Out[189]= True
```

For non-zero-monotonic games the pre-kernel and the kernel solution are different. Recall the following three person example given by Maschler, Peleg and Shapley (1979), p 316. The kernel solution is given by {20,0,15} satisfying individual rationality and the pre-kernel is found at {27.5,-5,17.5} where individual rationality is violated. Define the three person game by

```
In[190]:= ExpGame91 := (T = {1, 2, 3};
            Clear[v];
            v[{}] = 0; v[{1}] = 0; v[{2}] = 0; v[{3}] = 0;
            v[{1, 2}] = 20; v[{1, 3}] = 50; v[{2, 3}] = 10;
            v[T] = 40; )
```

and indeed we compute the kernel by relying on the LP approach

```
In[191]:= Kernel[ExpGame91]
Game has empty core
Out[191]= {25, 0, 15}
```

as well as by relying on the convergence algorithm by calling

```
In[192]:= ker91 = FindKernelSolution[ExpGame91, {0, 0, 40}]
Out[192]= {25, 0, 15}
```

Invoking the function to compute the pre-kernel, we get exactly the required result.

```
In[193]:= prk91 = FindPreKernelSolution[ExpGame91, {40, 0, 0}]
Out[193]= {55/2, -5, 35/2}

In[194]:= N[prk91]
Out[194]= {27.5, -5., 17.5}
```

Now, we want to demonstrate some limitation of the kernel computation associated with our approach by relying on the largest bi-symmetrical transfers. The core consists of a unique point {1, 1, 1, 2, 2}. For this purpose, resume the game "*ExpGame3*". Maschler, Peleg and Shapley (1979) found out that the kernel is a line segment extending from the core point {1, 1, 1, 2, 2} to a boundary point of the imputation set. The kernel of this game is

$$\mathrm{K}(\Gamma) := \left\{ \left(t, t, t, \frac{(7-3\,t)}{2}, \frac{(7-3\,t)}{2} \right) \; : \; 0 \le t \le 1 \right\} \quad \text{and the pre-kernel}$$

$$\mathrm{K}^*(\Gamma) := \left\{ \left(t, t, t, \frac{(7-3\,t)}{2}, \frac{(7-3\,t)}{2} \right) \; : \; -0.2 \le t \le 1 \right\} .$$

According to the calculation of the first critical value, we see that the core of the game coincides with the least core of the game. Let us construct some pre-kernel elements from the line segment, this can be done by

```
In[195]:= kerline = Table[{t, t, t, (7 - 3 t)/2, (7 - 3 t)/2}, {t, -3/10, 11/10, 1/10}]
```

```
Out[195]= {{-3/10, -3/10, -3/10, 79/20, 79/20}, {-1/5, -1/5, -1/5, 19/5, 19/5},
           {-1/10, -1/10, -1/10, 73/20, 73/20}, {0, 0, 0, 7/2, 7/2},
           {1/10, 1/10, 1/10, 67/20, 67/20}, {1/5, 1/5, 1/5, 16/5, 16/5},
           {3/10, 3/10, 3/10, 61/20, 61/20}, {2/5, 2/5, 2/5, 29/10, 29/10},
           {1/2, 1/2, 1/2, 11/4, 11/4}, {3/5, 3/5, 3/5, 13/5, 13/5},
           {7/10, 7/10, 7/10, 49/20, 49/20}, {4/5, 4/5, 4/5, 23/10, 23/10},
           {9/10, 9/10, 9/10, 43/20, 43/20}, {1, 1, 1, 2, 2}, {11/10, 11/10, 11/10, 37/20, 37/20}}
```

Verifying that the constructed elements are really contained in the line segment that constitutes the kernel solution we use the function

```
In[196]:= PreKernelQ[ExpGame3, kerline]
Out[196]= {False, True, True, True, True, True, True,
            True, True, True, True, True, True, True, False}

In[197]:= KernelImputationListQ[ExpGame3, kerline]
Out[197]= {False, False, False, True, True, True, True,
            True, True, True, True, True, True, True, False}

In[198]:= Kernel[ExpGame3]
Game has nonempty core
Out[198]= {1, 1, 1, 2, 2}
```

This kernel solution is contained in the core and coincides with the Nucleolus of the game.

```
In[199]:= Nucleolus[ExpGame3]
Out[199]= {1, 1, 1, 2, 2}

In[200]:= LexiCenter[ExpGame3]
Out[200]= {1, 1, 1, 2, 2}

In[201]:= kersol = Table[Kernel[ExpGame3, EpsilonValue → j], {j, 10}]
Out[201]= {{1, 1, 1, 2, 2}, {1, 1, 1, 2, 2}, {1, 1, 1, 2, 2},
           {1, 1, 1, 2, 2}, {1, 1, 1, 2, 2}, {1, 1, 1, 2, 2},
           {1, 1, 1, 2, 2}, {1, 1, 1, 2, 2}, {1, 1, 1, 2, 2}, {1, 1, 1, 2, 2}}
```

By varying the critical value we see that we get in any case the same kernel result. This is caused due to the fact that the core consists of just a singleton. Enlarging the core by some epsilon values has no effect on the largest bi-symmetrical transfer and its associated midpoint. Clearly, the largest bi-symmetrical transfer varies if we enlarge the core by some epsilon values, but it still remains the largest transfer at the imputation computed. At the other kernel imputations the bi-symmetrical transfer is smaller than the value we will obtain at {1,1,1,2,2} (cf. Meinhardt (2004)). The same happens if we rely on the function **KernelCalculation[]** that use a different procedure to compute kernel elements. We obtain

```
In[202]:= {sol31, obj31, const31, tra31, pay31} =
            KernelCalculation[ExpGame3, EpsilonValue → 4,
              DisplayAllResults → True]
Game is zero - monotone? False
Core is nonempty? True
Game is either zero - monotonic or has nonempty core
A Kernel solution is : {x[1] → 1, x[2] → 1, x[3] → 1, x[4] → 2, x[5] → 2}
```

```
Out[202]= {{1, 1, 1, 2, 2}, x[1] + x[2] + x[3] + x[4] + x[5],
            {-4 - x[1] ≤ -4, -4 - x[2] ≤ -4, -4 - x[3] ≤ -4, -4 - x[4] ≤ -4,
              -4 - x[1] - x[4] ≤ -4, -4 - x[2] - x[4] ≤ -4, -x[1] - x[2] - x[4] ≤ -4,
              -4 - x[3] - x[4] ≤ -4, -x[1] - x[3] - x[4] ≤ -4,
              -x[2] - x[3] - x[4] ≤ -4, -4 - x[5] ≤ -4, -4 - x[1] - x[5] ≤ -4,
              -4 - x[2] - x[5] ≤ -4, -x[1] - x[2] - x[5] ≤ -4, -4 - x[3] - x[5] ≤ -4,
              -x[1] - x[3] - x[5] ≤ -4, -x[2] - x[3] - x[5] ≤ -4, -x[4] - x[5] ≤ -4,
              -4 - x[1] - x[4] - x[5] ≤ -4, -4 - x[2] - x[4] - x[5] ≤ -4,
              -4 - x[3] - x[4] - x[5] ≤ -4, x[1] + x[2] + x[3] + x[4] + x[5] ≤ 7},
            {3/2, 3/2, 1/6, 1/6, 3/2, 1/6, 1/6, 1/6, 1/6, 1/2},
            {{1/6, 4/3, 4/3, 3/2, 8/3}, {1/6, 4/3, 4/3, 8/3, 3/2},
              {1, 1, 1, 2, 2}, {4/3, 1/6, 4/3, 3/2, 8/3}, {4/3, 1/6, 4/3, 8/3, 3/2},
              {4/3, 4/3, 1/6, 3/2, 8/3}, {4/3, 4/3, 1/6, 8/3, 3/2}, {3/2, 3/2, 4, 0, 0},
              {3/2, 4, 3/2, 0, 0}, {2, 2, 2, 1/2, 1/2}, {4, 3/2, 3/2, 0, 0}}}
```

In some case the $n(n-1)/2$ initial LPs provide some additional kernel elements. These kernel candidates have been stored for the above example in the variable *pay31* and we can test now with the command **KernelImputationListQ[]** if one or some of these candidates is an or are additional kernel element(s).

```
In[203]:= KernelImputationListQ[ExpGame3, pay31]
Out[203]= {False, False, True, False, False,
            False, False, False, False, False, False}

In[204]:= {sol32, pay32} = KernelCalculation[ExpGame3, EpsilonValue → 21]
Game is zero - monotone? False
Core is nonempty? True
Game is either zero - monotonic or has nonempty core
A Kernel solution is : {x[1] → 1, x[2] → 1, x[3] → 1, x[4] → 2, x[5] → 2}
Out[204]= {{1, 1, 1, 2, 2}, {{1/6, 4/3, 4/3, 3/2, 8/3}, {1/6, 4/3, 4/3, 8/3, 3/2},
              {1, 1, 1, 2, 2}, {4/3, 1/6, 4/3, 3/2, 8/3}, {4/3, 1/6, 4/3, 8/3, 3/2},
              {4/3, 4/3, 1/6, 3/2, 8/3}, {4/3, 4/3, 1/6, 8/3, 3/2}, {3/2, 3/2, 4, 0, 0},
              {3/2, 4, 3/2, 0, 0}, {2, 2, 2, 1/2, 1/2}, {4, 3/2, 3/2, 0, 0}}}
```

Let us look again on additional kernel elements.

```
In[205]:= KernelImputationListQ[ExpGame3, pay32]
Out[205]= {False, False, True, False, False,
            False, False, False, False, False, False}
```

In both cases we get the same kernel candidates by relying on the function **KernelCalculation[]** which are not contained in the solution set. Hence, we did not find any additional kernel solution, although the kernel is not a singleton. But now we want provide a strategy to compute the complete kernel solution by using the dual game. This strategy could be successful in cases that the least core of the dual and the primal game are different. For a quick test we are looking on the nucleolus of the dual and the primal game. Let us define the dual game of *ExpGame3*.

```
In[206]:= duvec = DualGame[ExpGame3]
Out[206]= {0, 7, 7, 7, 7, 7, 7, 7, 3, 3, 7, 3, 3, 3,
            3, 7, 3, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7}

In[207]:= DualExpGame3 := (DefineGame[T3, duvec]; );
```

The nucleolus of the dual is

```
In[208]:= Nucleolus[DualExpGame3]
Out[208]= {7/5, 7/5, 7/5, 7/5, 7/5}

In[209]:= nuc3
Out[209]= {1, 1, 1, 2, 2}
```

Note first that the dual is not a zero-monotonic game.

```
In[210]:= ZeroMonotoneQ[DualExpGame3]
Out[210]= False

In[211]:= {duker3, dupay3} = KernelCalculation[DualExpGame3,
            EpsilonValue → 7, SetGameToNonZeroMonotonic → True]
Game is zero - monotone? False
Core is nonempty? False
Game is not zero - monotonic and has empty core
 A Kernel solution is :
  {x[1] → 7/5, x[2] → 7/5, x[3] → 7/5, x[4] → 7/5, x[5] → 7/5}
Out[211]= {{7/5, 7/5, 7/5, 7/5, 7/5}, {{0, 0, 0, 7/2, 7/2}, {0, 0, 7/2, 0, 7/2},
            {0, 0, 7/2, 7/2, 0}, {0, 7/2, 0, 0, 7/2}, {0, 7/2, 0, 7/2, 0},
            {0, 7/2, 7/2, 0, 0}, {7/5, 7/5, 7/5, 7/5, 7/5}, {7/2, 0, 0, 0, 7/2},
            {7/2, 0, 0, 7/2, 0}, {7/2, 0, 7/2, 0, 0}, {7/2, 7/2, 0, 0, 0}}}
```

But note that you should use the option for non zero-monotonic games with care. The algorithm works quite well for zero-monotonic games, using the above option can cause an infinite loop for an non zero-monotonic game.

```
In[212]:= KernelImputationListQ[ExpGame3, dupay3]
Out[212]= {True, False, False, False, False,
            False, False, False, False, False, False}

In[213]:= First[dupay3]
Out[213]= {0, 0, 0, 7/2, 7/2}
```

This is the second extreme point of the kernel segment given by

```
In[214]:= kerline
Out[214]= {{-3/10, -3/10, -3/10, 79/20, 79/20}, {-1/5, -1/5, -1/5, 19/5, 19/5},
            {-1/10, -1/10, -1/10, 73/20, 73/20}, {0, 0, 0, 7/2, 7/2},
            {1/10, 1/10, 1/10, 67/20, 67/20}, {1/5, 1/5, 1/5, 16/5, 16/5},
            {3/10, 3/10, 3/10, 61/20, 61/20}, {2/5, 2/5, 2/5, 29/10, 29/10},
            {1/2, 1/2, 1/2, 11/4, 11/4}, {3/5, 3/5, 3/5, 13/5, 13/5},
            {7/10, 7/10, 7/10, 49/20, 49/20}, {4/5, 4/5, 4/5, 23/10, 23/10},
            {9/10, 9/10, 9/10, 43/20, 43/20}, {1, 1, 1, 2, 2}, {11/10, 11/10, 11/10, 37/20, 37/20}}
```

By invoking the function **FindKernelSolution[]** we find

```
In[215]:= kersol30 = FindKernelSolution[ExpGame3, {0, 0, 0, 0, 7}]
Out[215]= {0, 0, 0, 7/2, 7/2}
```

and by computing kernel elements by the list of permutations from the vector {7,0,0,0,0} we get

```
In[216]:= permut = Permutations[{7, 0, 0, 0, 0}];
```

```
In[217]:= kerper = FindKernelSolution[ExpGame3, permut]//Union
Out[217]= {{0, 0, 0, 7/2, 7/2}, {7/8, 7/8, 7/8, 35/16, 35/16}}

In[218]:= KernelImputationListQ[ExpGame3, kerper]
Out[218]= {True, True}

In[219]:= kersol31 = FindKernelSolution[ExpGame3, pay31]//Union
Out[219]= {{3/4, 3/4, 3/4, 19/8, 19/8}, {29/32, 29/32, 29/32, 137/64, 137/64},
           {31/32, 31/32, 31/32, 131/64, 131/64}, {1, 1, 1, 2, 2}}

In[220]:= kersol32 = FindKernelSolution[ExpGame3, pay32]//Union
Out[220]= {{3/4, 3/4, 3/4, 19/8, 19/8}, {29/32, 29/32, 29/32, 137/64, 137/64},
           {31/32, 31/32, 31/32, 131/64, 131/64}, {1, 1, 1, 2, 2}}
```

Finally, we verify that the computed solutions are kernel elements by calling again

```
In[221]:= KernelImputationListQ[ExpGame3, kersol32]
Out[221]= {True, True, True, True}
```

Now let us check whether the payoffs are balancing the excesses. We can check this by invoking the function

```
In[222]:= AllMaxSurpluses[ExpGame3, kerline, DisplayMatrixForm → True]
```

$$\text{Out[222]=}\begin{pmatrix} \begin{pmatrix} \text{sij[\{1,2\}]} \\ \text{sij[\{2,1\}]} \end{pmatrix} & \begin{pmatrix} \text{sij[\{1,3\}]} \\ \text{sij[\{3,1\}]} \end{pmatrix} & \begin{pmatrix} \text{sij[\{1,4\}]} \\ \text{sij[\{4,1\}]} \end{pmatrix} & \begin{pmatrix} \text{sij[\{1,5\}]} \\ \text{sij[\{5,1\}]} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \text{sij[\{4,5\}]} \\ \text{sij[\{5,4\}]} \end{pmatrix} \\ \begin{pmatrix} \frac{13}{20} \\ \frac{13}{20} \end{pmatrix} & \begin{pmatrix} \frac{13}{20} \\ \frac{13}{20} \end{pmatrix} & \begin{pmatrix} \frac{9}{10} \\ \frac{13}{20} \end{pmatrix} & \begin{pmatrix} \frac{9}{10} \\ \frac{13}{20} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{13}{20} \\ \frac{13}{20} \end{pmatrix} \\ \begin{pmatrix} \frac{3}{5} \\ \frac{3}{5} \end{pmatrix} & \begin{pmatrix} \frac{3}{5} \\ \frac{3}{5} \end{pmatrix} & \begin{pmatrix} \frac{3}{5} \\ \frac{3}{5} \end{pmatrix} & \begin{pmatrix} \frac{3}{5} \\ \frac{3}{5} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{3}{5} \\ \frac{3}{5} \end{pmatrix} \\ \begin{pmatrix} \frac{11}{20} \\ \frac{11}{20} \end{pmatrix} & \begin{pmatrix} \frac{11}{20} \\ \frac{11}{20} \end{pmatrix} & \begin{pmatrix} \frac{11}{20} \\ \frac{11}{20} \end{pmatrix} & \begin{pmatrix} \frac{11}{20} \\ \frac{11}{20} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{11}{20} \\ \frac{11}{20} \end{pmatrix} \\ \begin{pmatrix} \frac{1}{2} \\ 1 \end{pmatrix} & \begin{pmatrix} \frac{1}{2} \\ 1 \end{pmatrix} & \begin{pmatrix} \frac{1}{2} \\ 1 \end{pmatrix} & \begin{pmatrix} \frac{1}{2} \\ 1 \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{1}{2} \\ 1 \end{pmatrix} \\ \begin{pmatrix} \frac{9}{20} \\ \frac{9}{20} \end{pmatrix} & \begin{pmatrix} \frac{9}{20} \\ \frac{9}{20} \end{pmatrix} & \begin{pmatrix} \frac{9}{20} \\ \frac{9}{20} \end{pmatrix} & \begin{pmatrix} \frac{9}{20} \\ \frac{9}{20} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{9}{20} \\ \frac{9}{20} \end{pmatrix} \\ \begin{pmatrix} \frac{2}{5} \\ \frac{2}{5} \end{pmatrix} & \begin{pmatrix} \frac{2}{5} \\ \frac{2}{5} \end{pmatrix} & \begin{pmatrix} \frac{2}{5} \\ \frac{2}{5} \end{pmatrix} & \begin{pmatrix} \frac{2}{5} \\ \frac{2}{5} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{2}{5} \\ \frac{2}{5} \end{pmatrix} \\ \begin{pmatrix} \frac{7}{20} \\ \frac{7}{20} \end{pmatrix} & \begin{pmatrix} \frac{7}{20} \\ \frac{7}{20} \end{pmatrix} & \begin{pmatrix} \frac{7}{20} \\ \frac{7}{20} \end{pmatrix} & \begin{pmatrix} \frac{7}{20} \\ \frac{7}{20} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{7}{20} \\ \frac{7}{20} \end{pmatrix} \\ \begin{pmatrix} \frac{3}{10} \\ \frac{3}{10} \end{pmatrix} & \begin{pmatrix} \frac{3}{10} \\ \frac{3}{10} \end{pmatrix} & \begin{pmatrix} \frac{3}{10} \\ \frac{3}{10} \end{pmatrix} & \begin{pmatrix} \frac{3}{10} \\ \frac{3}{10} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{3}{10} \\ \frac{3}{10} \end{pmatrix} \\ \begin{pmatrix} \frac{1}{4} \\ \frac{1}{4} \end{pmatrix} & \begin{pmatrix} \frac{1}{4} \\ \frac{1}{4} \end{pmatrix} & \begin{pmatrix} \frac{1}{4} \\ \frac{1}{4} \end{pmatrix} & \begin{pmatrix} \frac{1}{4} \\ \frac{1}{4} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{1}{4} \\ \frac{1}{4} \end{pmatrix} \\ \begin{pmatrix} \frac{1}{5} \\ \frac{1}{5} \end{pmatrix} & \begin{pmatrix} \frac{1}{5} \\ \frac{1}{5} \end{pmatrix} & \begin{pmatrix} \frac{1}{5} \\ \frac{1}{5} \end{pmatrix} & \begin{pmatrix} \frac{1}{5} \\ \frac{1}{5} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{1}{5} \\ \frac{1}{5} \end{pmatrix} \\ \begin{pmatrix} \frac{3}{20} \\ \frac{3}{20} \end{pmatrix} & \begin{pmatrix} \frac{3}{20} \\ \frac{3}{20} \end{pmatrix} & \begin{pmatrix} \frac{3}{20} \\ \frac{3}{20} \end{pmatrix} & \begin{pmatrix} \frac{3}{20} \\ \frac{3}{20} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{3}{20} \\ \frac{3}{20} \end{pmatrix} \\ \begin{pmatrix} \frac{1}{10} \\ \frac{1}{10} \end{pmatrix} & \begin{pmatrix} \frac{1}{10} \\ \frac{1}{10} \end{pmatrix} & \begin{pmatrix} \frac{1}{10} \\ \frac{1}{10} \end{pmatrix} & \begin{pmatrix} \frac{1}{10} \\ \frac{1}{10} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{1}{10} \\ \frac{1}{10} \end{pmatrix} \\ \begin{pmatrix} \frac{1}{20} \\ \frac{1}{20} \end{pmatrix} & \begin{pmatrix} \frac{1}{20} \\ \frac{1}{20} \end{pmatrix} & \begin{pmatrix} \frac{1}{20} \\ \frac{1}{20} \end{pmatrix} & \begin{pmatrix} \frac{1}{20} \\ \frac{1}{20} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} \frac{1}{20} \\ \frac{1}{20} \end{pmatrix} \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\ \begin{pmatrix} -\frac{1}{20} \\ -\frac{1}{20} \end{pmatrix} & \begin{pmatrix} -\frac{1}{20} \\ -\frac{1}{20} \end{pmatrix} & \begin{pmatrix} -\frac{1}{20} \\ \frac{3}{10} \end{pmatrix} & \begin{pmatrix} -\frac{1}{20} \\ \frac{3}{10} \end{pmatrix} & \begin{pmatrix} \cdots \\ \cdots \end{pmatrix} & \begin{pmatrix} -\frac{1}{20} \\ -\frac{1}{20} \end{pmatrix} \end{pmatrix}$$

If one is just interested in a special pair of players, in our example for the pair {3,4}, then one should call

```
In[223]:= MaxSurplus[ExpGame3, 3, 4, kerline]
Out[223]= {9/10, 3/5, 11/20, 1/2, 9/20, 2/5, 7/20, 3/10, 1/4, 1/5, 3/20, 1/10, 1/20, 0, -1/20}

In[224]:= MaxSurplus[ExpGame3, 4, 3, kerline]
Out[224]= {13/20, 3/5, 11/20, 1/2, 9/20, 2/5, 7/20, 3/10, 1/4, 1/5, 3/20, 1/10, 1/20, 0, 3/10}
```

4.4 Unanimity Coordinates

The package *TuGames* provides also a small set of functions related to unanimity coordinates. We want to demonstrate the use of these functions for an example given by Rafels and Ybern (1995) on page 120. The example is a 5-player game with the following values:

```
In[225]:= worth = {0, 0, 0, 0, 0, 0, 50.2, 49.7, 47.1, 55.8, 53.3, 101.3,
              137.6, 130.8, 131.2, 188.4, 149.7, 229.5, 259.6, 239.3, 278,
              428.2, 289.1, 308.8, 428.2, 432.8, 471.9, 555.7, 756.5,
              772.5, 712.8, 1209.2};
```

The game is defined by

```
In[226]:= ExpGame9 := (DefineGame[T3, worth, RationalApproximate → False]);
```

Among other things, we can evaluate all unanimity coordinates of the game "*ExpGame9*" with the command

```
In[227]:= UnanimityCoordinates[ExpGame9]
Out[227]= {0, 0, 0, 0, 0, 0, 50.2, 49.7, 47.1, 55.8, 53.3, 101.3, 137.6,
            130.8, 131.2, 188.4, -3.5, 30.9, 16., 11.7, 41.3, 136.9,
            3.7, -13.3, 0.9, -17.6, -3.3, 37.4, -8.6, -2.8, -3.5, 37.6}
```

Furthermore, we can extract the minimum unanimity coordinates of each coalition size by

```
In[228]:= MinUnanimityCoordinates[ExpGame9]
Out[228]= {0, 47.1, -17.6, -8.6, 37.6}
```

In addition to these functions we can also verify the convexity property by relying on unanimity coordinates. The following sufficient condition of convexity worked out in Rafels and Ybern (1995) in Corollary 2.1 is implemented in the function **EvalSumMinCoord[].** First recall that the unanimity coordinates are defined by

$$\lambda_R := \sum_{S \subset R} (-1)^{r-s} v(S), \text{ for all } R \subseteq N, S, R \neq 0.$$

A sufficient condition of convexity of a game v in terms of unanimity coordinates is given by

$$\text{if for all } k=0,\ldots,n-2 \sum_{j=0}^{k} \frac{k!}{j!\,(k-j)!} \alpha_j \geq 0, \text{ then the game is convex, where } \alpha_k := \min_{|S| = k+2} \lambda_S.$$

```
In[229]:= EvalSumMinCoord[ExpGame9]
Out[229]= {47.1, 29.5, 3.3, 6.1}
```

All evaluated sums are strictly positive, hence we conclude that the game is convex. A fortiori, due its strictly positive values the game is even strictly convex. The functions presented next verify just whether the sufficient condition of convexity in terms of unanimity coordinates is satisfied.

```
In[230]:= ConvexUnanConditionQ[ExpGame9]
Out[230]= True

In[231]:= StrictlyConvexUnanConditionQ[ExpGame9]
Out[231]= True

In[232]:= ConvexQ[ExpGame9]
Out[232]= True
```

In cases that all unanimity coordinates are nonnegative (i.e. the game is convex) or that the minimum size of coalition with negative unanimity coordinates is greater than *N-2* the return value of the function **EvalSumMinCoord[]** is the vector of minimum values of unanimity coordinates of each coalition size. In the former case we obtain, for instance,

```
In[233]:= MinUnanimityCoordinates[ExpGame1]
Out[233]= {0, 0, 0, 10}

In[234]:= EvalSumMinCoord[ExpGame1]
```

```
No negative unanimity coordinates found.
Game is convex.
The minimum unanimity coordinates are :
Out[234]= {0, 0, 0, 10}
```

and for the latter

```
In[235]:= MinUnanimityCoordinates[ExpGame8]
Out[235]= {267/10, 3199/15, -5597/30, 2398/15}
In[236]:= EvalSumMinCoord[ExpGame8]
 Minimum coalition size with
      negative unanimity coordinate is equal to 3
Coalition size of T - 2 is 2
3 greater than 2
No sum of minimum unanimity coordinates can be evaluated.
The minimum unanimity coordinates are :
Out[236]= {267/10, 3199/15, -5597/30, 2398/15}
```

5 Computing the Vertices of the Core

The next functions describe some interfaces that call the *cddmathlink* library via *MathLink* to speed up calculation. One can also use the command **CoreVertices[]** to obtain the same result as with the command **CddVerticesCore[]**. But this command is based on *MATHEMATICA*®-Code and it is very slow. That means that on old computers like a Pentium II system your game should not be greater than 6 player if you are interest to get the result in finite time. The examples are performed on a Pentium II with 266 MHz under LINUX x86 and *MATHEMATICA*® Version 4.2/5.0. On a Xeon Processor with 2.4 GHZ the same example was computed in 5 seconds.

If the package VertexEnum is properly installed on your OS, you can calculate the vertices of the core by using the command

```
In[237]:= {time, {vert1, nonvert1}} = Timing[VerticesCore[ExpGame1]]
Out[237]= {106.21 Second,
            {{{0, 50, 40, 0}, {40, 50, 0, 0}, {0, 0, 90, 0}, {90, 0, 0, 0},
                {0, 0, 40, 50}, {40, 0, 0, 50}, {0, 40, 0, 50}, {0, 50, 0, 40}},
              {{17, 8, 11, 19, 5, 1, 4}, {17, 14, 8, 19, 7, 3, 4},
                {16, 11, 13, 19, 5, 6, 8, 9, 2, 1, 4}, {18, 14, 13, 19, 6, 7, 8,
                  12, 3, 2, 4}, {15, 6, 9, 19, 5, 2, 1}, {15, 12, 7, 19, 6, 3, 2},
                {15, 10, 7, 19, 5, 3, 1}, {17, 10, 7, 19, 5, 3, 1}}}}
```

Note, that the *MATHEMATICA*® Version 3.0 is even faster under the command **CoreVertices[]**, you get all vertices on the same platform in 52 seconds instead of 107 seconds.

By calling the *cddmathlink* library, we obtain the same result in 0.06 seconds.

```
In[238]:= {time, vert2} = AbsoluteTiming[CddVerticesCore[ExpGame1]]
Out[238]= {0.059817 Second,
            {{0, 0, 40., 50.}, {0, 0, 90., 0}, {0, 40., 0, 50.}, {0, 50., 0, 40.},
              {0, 50., 40., 0}, {40., 0, 0, 50.}, {40., 50., 0, 0}, {90., 0, 0, 0}}}
```

If you are interested in rational results, then call

```
In[239]:= {time, vert3} = AbsoluteTiming[CddGmpVerticesCore[ExpGame1]]
Out[239]= {0.172962 Second,
            {{0, 0, 40, 50}, {0, 0, 90, 0}, {0, 40, 0, 50}, {0, 50, 0, 40},
              {0, 50, 40, 0}, {40, 0, 0, 50}, {40, 50, 0, 0}, {90, 0, 0, 0}}}
```

Now let us check the result on core elements. This can be done by calling

```
In[240]:= CoreElementsQ[ExpGame1, vert1]
Out[240]= {True, True, True, True, True, True, True, True}

In[241]:= BelongToCoreQ[ExpGame1, vert1]
Out[241]= {True, True, True, True, True, True, True, True}

In[242]:= CoreElementsQ[ExpGame1, vert2]
Out[242]= {True, True, True, True, True, True, True, True}
```

This is the expected result since the game "*ExpGame1"* is convex.

Since we have less than 24 vertices let us verify whether the game is strictly convex.

```
In[243]:= ConvexStrQ[ExpGame1]
Out[243]= False
```

The game "*ExpGame1"* is convex let us also compute the marginal values.

```
In[244]:= marv = MargValue[ExpGame1]
Out[244]= {{0, 0, 40, 50}, {0, 0, 90, 0}, {0, 40, 0, 50}, {0, 50, 0, 40},
            {0, 0, 90, 0}, {0, 50, 40, 0}, {0, 0, 40, 50}, {0, 0, 90, 0},
            {40, 0, 0, 50}, {90, 0, 0, 0}, {0, 0, 90, 0}, {90, 0, 0, 0},
            {0, 40, 0, 50}, {0, 50, 0, 40}, {40, 0, 0, 50}, {90, 0, 0, 0},
            {40, 50, 0, 0}, {90, 0, 0, 0}, {0, 0, 90, 0}, {0, 50, 40, 0},
            {0, 0, 90, 0}, {90, 0, 0, 0}, {40, 50, 0, 0}, {90, 0, 0, 0}}

In[245]:= Length[marv]
Out[245]= 24
```

This is the expected result for convex games that the number of marginal worth vectors is equal to $n!$. To demonstrate that the marginal worth vectors coincide with the vertices of the core, we remove all duplicated vectors by

```
In[246]:= unmarv = Union[marv]
Out[246]= {{0, 0, 40, 50}, {0, 0, 90, 0}, {0, 40, 0, 50}, {0, 50, 0, 40},
            {0, 50, 40, 0}, {40, 0, 0, 50}, {40, 50, 0, 0}, {90, 0, 0, 0}}

In[247]:= Length[Union[marv]]
Out[247]= 8

In[248]:= CoreElementsQ[ExpGame1, unmarv]
Out[248]= {True, True, True, True, True, True, True, True}

In[249]:= BelongToCoreQ[ExpGame1, unmarv]
Out[249]= {True, True, True, True, True, True, True, True}

In[250]:= to1 = Union[marv, vert1]
Out[250]= {{0, 0, 40, 50}, {0, 0, 90, 0}, {0, 40, 0, 50}, {0, 50, 0, 40},
            {0, 50, 40, 0}, {40, 0, 0, 50}, {40, 50, 0, 0}, {90, 0, 0, 0}}

In[251]:= Length[to1]
Out[251]= 8
```

Hence, we get the expected result, that the marginal worth vectors coincide with core vertices.

In particular, you should handle very carefully floating point results by checking on core elements. We make this point more clear by the following example

```
In[252]:= ecken8 = CddVerticesCore[ExpGame8]
Out[252]= {{26.7, 239.967, 266.667, 266.667}, {26.7, 266.667, 239.967, 266.667},
            {26.7, 266.667, 266.667, 239.967}, {239.967, 266.667, 266.667, 26.7},
            {239.967, 266.667, 26.7, 266.667}, {239.967, 26.7, 266.667, 266.667},
            {266.667, 239.967, 266.667, 26.7}, {266.667, 266.667, 26.7, 239.967},
            {266.667, 266.667, 239.967, 26.7}, {266.667, 26.7, 239.967, 266.667},
            {266.667, 26.7, 266.667, 239.967}, {266.667, 239.967, 26.7, 266.667}}
```

These are exactly the core vertices we have already computed in the Section 4.2 . But if we check this result with the usual function **CoreElementsQ[]** we obtain a wrong result by using floating point numbers.

```
In[253]:= CoreElementsQ[ExpGame8, ecken8]
Out[253]= {True, True, True, True, True,
            True, True, True, True, True, True, True}
```

The correct result is delivered by the function **BelongToCoreQ[]**

```
In[254]:= BelongToCoreQ[ExpGame8, ecken8]
Out[254]= {True, True, True, True, True,
            True, True, True, True, True, True, True}
```

To overcome problems associated with floating points result you should convert these results to rational numbers by using the *MATHEMATICA*® built-in function **Rationalize[]**.

```
In[255]:= ravert8 = Rationalize[ecken8]
Out[255]= {{267/10, 7199/30, 800/3, 800/3}, {267/10, 800/3, 7199/30, 800/3},
            {267/10, 800/3, 800/3, 7199/30}, {7199/30, 800/3, 800/3, 267/10},
            {7199/30, 800/3, 267/10, 800/3}, {7199/30, 267/10, 800/3, 800/3},
            {800/3, 7199/30, 800/3, 267/10}, {800/3, 800/3, 267/10, 7199/30},
            {800/3, 800/3, 7199/30, 267/10}, {800/3, 267/10, 7199/30, 800/3},
            {800/3, 267/10, 800/3, 7199/30}, {800/3, 7199/30, 267/10, 800/3}}

In[256]:= Length[ravert8]
Out[256]= 12

In[257]:= CoreElementsQ[ExpGame8, ravert8]
Out[257]= {True, True, True, True, True,
            True, True, True, True, True, True, True}

In[258]:= BelongToCoreQ[ExpGame8, ravert8]
Out[258]= {True, True, True, True, True,
            True, True, True, True, True, True, True}
```

This is exactly the result we have already obtained in the Section 4.2.

You can also calculate the vertices of the imputation set and of the reasonable set by using

```
In[259]:= Rationalize[CddVerticesImputationSet[ExpGame1]]
Out[259]= {{0, 0, 0, 90}, {0, 0, 90, 0}, {0, 90, 0, 0}, {90, 0, 0, 0}}
```

and

```
In[260]:= Rationalize[CddVerticesReasonableSet[ExpGame1]]
Out[260]= {{-100, 50, 90, 50}, {90, -140, 90, 50},
            {90, 50, -100, 50}, {90, 50, 90, -140}}
```

6 Concluding Remarks and Limitations

In this section we want to discuss some limitations that appear in the context of the package *TuGames*. Note first that it is not possible to assign arbitrary names to the players. The players must be named with natural numbers. Naming players in the fashion as we did it in the example below does not work or produces wrong results in bad cases.

```
In[261]:= T10 = {p1, p2, p3}
Out[261]= {p1, p2, p3}
```

```
In[262]:= T11 = {1, 2, 3}
Out[262]= {1, 2, 3}

In[263]:= vec10 = Table[0, {k, 2^Length[T10]}];

In[264]:= ExpGame10 := (DefineGame[T10, vec10]; v[{p2}] = v[{p3}] = 45;
              v[{p1, p2}] = 40; v[{p1, p3}] = 40; v[T] = 90; );
```

If we want to evaluate the Shapley value of the game with the chosen representation of the player set this will produce an error.

```
In[265]:= NewShapley[ExpGame10]
Part :: partw : Part 1 of {} does not exist.
 Take :: seqs : Sequence specification (+n, -n, {+n}, {-n}, {m, n}, or {m, n, s})
      expected at position 2 in Take[{p1, p2, p3}, -1 + {}1].
Part :: partw : Part 2 of {} does not exist.
 Take :: seqs : Sequence specification (+n, -n, {+n}, {-n}, {m, n}, or {m, n, s})
      expected at position 2 in Take[{p1, p2, p3}, -1 + {}2].
Part :: partw : Part 3 of {} does not exist.
 General :: stop :
   Further output of Part :: partw will be suppressed during this calculation.
 Take :: seqs : Sequence specification (+n, -n, {+n}, {-n}, {m, n}, or {m, n, s})
      expected at position 2 in Take[{p1, p2, p3}, -1 + {}3].
 General :: stop :
   Further output of Take :: seqs will be suppressed during this calculation.
Out[265]= {1/6 (v[{p1, {p1, p2, p3}, -1 + {}1}] + v[{p1, {p1, p3, p2}, -1 + {}1}] +
                v[{p1, {p2, p1, p3}, -1 + {}1}] + v[{p1, {p2, p3, p1}, -1 + {}1}] +
                v[{p1, {p3, p1, p2}, -1 + {}1}] + v[{p1, {p3, p2, p1}, -1 + {}1}] -
                v[Take[{p1, p2, p3}, -1 + {}1]] - v[Take[{p1, p3, p2}, -1 + {}1]] -
                v[Take[{p2, p1, p3}, -1 + {}1]] - v[Take[{p2, p3, p1}, -1 + {}1]] -
                v[Take[{p3, p1, p2}, -1 + {}1]] - v[Take[{p3, p2, p1}, -1 + {}1]]),
           1/6 (v[{p2, {p1, p2, p3}, -1 + {}2}] + v[{p2, {p1, p3, p2}, -1 + {}2}] +
                v[{p2, {p2, p1, p3}, -1 + {}2}] + v[{p2, {p2, p3, p1}, -1 + {}2}] +
                v[{p2, {p3, p1, p2}, -1 + {}2}] + v[{p2, {p3, p2, p1}, -1 + {}2}] -
                v[Take[{p1, p2, p3}, -1 + {}2]] - v[Take[{p1, p3, p2}, -1 + {}2]] -
                v[Take[{p2, p1, p3}, -1 + {}2]] - v[Take[{p2, p3, p1}, -1 + {}2]] -
                v[Take[{p3, p1, p2}, -1 + {}2]] - v[Take[{p3, p2, p1}, -1 + {}2]]),
Out[265]=  1/6 (v[{p3, {p1, p2, p3}, -1 + {}3}] + v[{p3, {p1, p3, p2}, -1 + {}3}] +
                v[{p3, {p2, p1, p3}, -1 + {}3}] + v[{p3, {p2, p3, p1}, -1 + {}3}] +
                v[{p3, {p3, p1, p2}, -1 + {}3}] + v[{p3, {p3, p2, p1}, -1 + {}3}] -
                v[Take[{p1, p2, p3}, -1 + {}3]] - v[Take[{p1, p3, p2}, -1 + {}3]] -
                v[Take[{p2, p1, p3}, -1 + {}3]] - v[Take[{p2, p3, p1}, -1 + {}3]] -
                v[Take[{p3, p1, p2}, -1 + {}3]] - v[Take[{p3, p2, p1}, -1 + {}3]])}
```

Renaming the players with natural numbers helps to produce the correct results.

```
In[266]:= T11 = MapThread[Set, {T10, {1, 2, 3}}]
Out[266]= {1, 2, 3}

In[267]:= sh1 = NewShapley[ExpGame10]
Out[267]= {85/3, 185/6, 185/6}

In[268]:= sh2 = ShapleyValue[ExpGame10]
Out[268]= {85/3, 185/6, 185/6}
```

Now, let us briefly discuss limitations that appear in relation to the operator ":=" or **SetDelayed[]** command. To find out what will happen if the game in which we are interested in is not evaluated, we define for this purpose a new game with similar coalitional values like the game "*ExpGame10*". Recall from the discussion of Section 3.1 that the values of the characteristic function are evaluated at the moment when the *game_name* appears in the command. Just defining the game and invoking the return key is not enough to change the values of the characteristic function.

```
In[269]:= ExpGame11 := (DefineGame[{1, 2, 3}, vec10]; v[{2}] = v[{3}] = 30;
            v[{1, 2}] = 40; v[{1, 3}] = 40; v[T] = 90; );
```

To check out which coalitional values are currently assigned can be done by

```
In[270]:= ??v
v[S] describes the worth of coalition S
  v[{}] = 0

  v[{1}] = 0

  v[{2}] = 45

  v[{3}] = 45

  v[{1, 2}] = 40

  v[{1, 3}] = 40

  v[{2, 3}] = 0

  v[{1, 2, 3}] = 90
```

We notice that still the coalitional values for the game "*ExpGame10* are present. If you want to evaluate, for example, the Shapley value for the game *ExpGame11* but you make a typesetting error by forgetting the number 11 in our example, you will obtain the result of the Shapley value for the game *ExpGame10*", since there is no change in the characteristic function.

```
In[271]:= NewShapley[ExpGame]
Out[271]= {85/3, 185/6, 185/6}
```

There is at that moment a change in the values of the characteristic function when a new *game_name* appears in a command that requires as an input parameter a *game_name*. For our example, we get new values for the characteristic function when we call the command **NewShapley[]** with a *game_name* that has been previously defined elsewhere in the notebook.

```
In[272]:= sh3 = NewShapley[ExpGame11]
Out[272]= {100/3, 85/3, 85/3}
In[273]:= sh4 = ShapleyValue[ExpGame11]
Out[273]= {100/3, 85/3, 85/3}
```

Now let us again check out the values of the characteristic function by

```
In[274]:= ??v
v[S] describes the worth of coalition S
```

```
v[{}] = 0

v[{1}] = 0

v[{2}] = 30

v[{3}] = 30

v[{1, 2}] = 40

v[{1, 3}] = 40

v[{2, 3}] = 0

v[{1, 2, 3}] = 90
```

In cases that some unexpected or strange results have occurred after the calculation task has finished, you should check out by the above command which values are currently assigned to the characteristic function.

We did also some effort to examine the data format of some functions to prevent that *MATHEMATICA*® is performing lengthy unnecessary computations. By avoiding such computations you can immediately correct the wrong input parameter instead of waiting some time till *MATHEMATICA*® has finished its evaluation. Here, we present a small and simple example. First let us construct some payoffs.

```
In[275]:= sol11 = KernelCalculation[ExpGame11]
Game is average - convex? False
Out[275]= {{10, 40, 40}, x[1] + x[2] + x[3],
            {-10 - x[1] ≤ -20, 20 - x[2] ≤ -20, 30 - x[1] - x[2] ≤ -20, 20 - x[3] ≤ -20,
              30 - x[1] - x[3] ≤ -20, x[1] + x[2] + x[3] ≤ 90}, {{10, 40, 40}, {30, 40, 20}}}
```

The list *sol11* captures all output informations. If we do not care about the data format of *sol11* and try to hand over the payoff informations contained in it to the function **BelongToCoreQ[]**, this produces

```
In[276]:= BelongToCoreQ[ExpGame11, sol11]
Depth is equal to 7
Usage : BelongToCoreQ[game, payoffs] and CoreElementsQ[game, payoffs]
Input format of the variable 'payoffs' is not correct.
 The variable 'payoffs' must be a
     list of payoff vectors or a single payoff vector.
```

The payoff informations in the list *sol11* are contained at the first and fourth position. To have access on these information one must extract these informations from the list. To extract informations located at the first or fourth position can be done by

```
In[277]:= sol11[[1]]
Out[277]= {10, 40, 40}

In[278]:= sol11[[4]]
Out[278]= {{10, 40, 40}, {30, 40, 20}}
```

These are the correct input formats to hand over to the function **BelongToCoreQ[]**.

```
In[279]:= BelongToCoreQ[ExpGame11, sol11[[4]]]
Out[279]= {True, False}
```

At the end let us mention that *MATHEMATICA*® offers you the possibility to enhance your typesetting productivity. As you have already realized some commands have rather lengthy names that makes it very inconvenient to type in every letter in the notebook. To overcome these inconveniences *MATHEMATICA*® offers you a command and symbol-name completion. This can be achieved by invoking together the control key (abbreviated by **C^**) and the key of the letter **k**. For instance to complete the command **MinUnanimityCoordinates[]** type the first four letters of the command and use the key combination **C^k** to complete the command. Hence

In[280]:= **MinU**

Invoking **C^k** completes the command.

In[281]:= **MinUnanimityCoordinates**

Now introducing the *game_name* in the command can also be performed by the **C^k** key combination. In this case you must again type in the first four letters of the *game_name*

In[282]:= **MinUnanimityCoordinates[ExpG**

then invoke **C^k** and you obtain

In[283]:= **MinUnanimityCoordinates[ExpGame1**

Finally, type the last number and the missing bracket to finish the typesetting.

In[284]:= **MinUnanimityCoordinates[ExpGame11]**

7 References

M. Carter, *Cooperative Games* , in Economic and Financial Modeling with *MATHEMATICA*®, editor Hal R. Varian, Springer Publisher, 167-191, 1993.

Theo Driessen, *Cooperative Games,Solutions and Applications*, Kluwer Academic Publishers, Dordrecht, 1988.

U. Faigle, W. Kern and J. Kuipers, *An efficient algorithm for nucleolus and prekernel computation in some classes of Tu Games*, Memorandum No. 1464, Faculty of Mathematical Sciences, University of Twente, 1998.

E. Inarra and J. Usategui, *The Shapley value and average convex games*, IJGT, 22, 13-29, 1993.

Maschler, M. and Peleg, B.,*A Characterization, Existence Proof and Dimension Bounds for the Kernel of a Game* , *Pacific Journal of Mathematics* 18(2), 289–328, 1966.

M. Maschler,B. Peleg and L.S. Shapley, *Geometric Properties of Kernel, Nucleolus and related Concepts*, in Mathematics of Operations Research, Vol.4, Nov. 1979, pp.303-338.

H. Meinhardt, *Decision Making in Cooperative Common Pool Situations*, Lecture Notes in Economics and Mathematical Systems, Vol. 517, 2002, Springer, Heidelberg.

H. Meinhardt, *An LP approach to compute the pre-kernel for cooperative games, Computers and Operation Research*, Vol 33/2 pp. 535-557, 2005.

C. Rafels and N. Ybern, *Even and Odd Marginal Worth Vectors, Owen's Multilinear Extension and Convex Games*, IJGT, 113-126, 1995.

Hal R. Varian (Ed.), *Economics and Financial Modeling with MATHEMATICA*®, Springer,1993