COMPUTER-ASSISTED INVESTIGATIONS ON FUNCTIONAL EQUATIONS AND CONVEXITY

Thesis for the Degree of Doctor of Philosophy (PhD)

LAN NHI TO
Supervisor: DR. ATTILA GILÁNYI

UNIVERSITY OF DEBRECEN
Doctoral Council for Natural Sciences and Engineering
Doctoral School of Mathematical and Computational Sciences

Debrecen, 2025

Hereby, I declare that I prepared this thesis within the Doctoral Council for Natural Sciences and Engineering, Doctoral School of Mathematical and Computational Sciences, University of Debrecen, in order to obtain a PhD Degree in Natural Sciences at the University of Debrecen.
The results published in the thesis are not reported in any other PhD theses.

Debrecen, 2025.

Lan Nhi To
signature of the candidate

Hereby, I confirm that Lan Nhi To, the candidate, conducted her studies with my supervision within the Mathematical Analysis, Functional Equations and Inequalities Doctoral Program of the Doctoral School of Mathematical and Computational Sciences between 2020 and 2025. The independent studies and research work of the candidate significantly contributed to the results published in the thesis.
I also declare that the results published in the thesis are not reported in any other theses.
I support the acceptance of the thesis.

Debrecen, 2025.

Attila Gilányi
signature of the supervisor

Computer-assisted investigations on functional equations and convexity

Dissertation submitted in partial fulfillment of the requirements for the doctoral (PhD) degree
In Mathematics and Computing

Written by Lan Nhi To, certified Master of Science in Computer Science

Prepared in the framework of the Mathematical and Computational Sciences doctoral school of the University of Debrecen
(Mathematical analysis, functional equations and inequalities programme)

Dissertation supervisor: Dr. Attila Gilányi

The official opponents of the dissertation:

Dr. .....
Dr. .....

The evaluation board:

chairperson:	Dr. .....
members:	Dr. .....
	Dr. .....
	Dr. .....
	Dr. .....

The date and venue of the dissertation defence: …… 2025

Acknowledgement

First of all, I would like to express my heartfelt gratitude to my supervisor, Assoc. Prof. Dr. Attila Gilányi, for his invaluable support, guidance, and inspiration throughout my research. I am deeply thankful for his patience, encouragement, and the time he dedicated to helping me expand my knowledge and achieve the best possible outcome for my dissertation.

I am also sincerely grateful to all the professors, lecturers, and staff of the Doctoral School of Mathematical and Computational Sciences and the Department of Analysis, Institute of Mathematics, University of Debrecen. Their enthusiasm for teaching and their unwavering support greatly enriched my academic journey.

My sincere thanks go to the Stipendium Hungaricum Scholarship Programme, funded by the Tempus Public Foundation and the Ministry of Education and Training of Vietnam, for granting me the opportunity to pursue my studies in Hungary. This experience has been both rewarding and memorable.

Finally, I wish to extend my deepest appreciation to my family and friends for their constant support, encouragement, and belief in me. Their unwavering presence throughout the challenges of this journey has been invaluable. This achievement would not have been possible without them.

Contents

Introduction 1

1 Alienness of linear functional equations 5

1.1 Introduction 5

1.2 Basic properties of linear functional equations . . 7

1.3 Extensions of the concepts of alienness and strong alienness 9

1.4 Description of the program function `isalien` . . . 10

1.5 Explanation of the program function `isalien` . . 23

2 Levi-Civita type functional equations 31

2.1 Introduction 31

2.2 Basic results about solutions of Levi-Civita type functional equations 32

2.3 Examples of solving Levi-Civita type functional equations 34

2.4 Description of the program function `LCfesolve` . 40

2.5 Explanation of the program function `LCfesolve` . 44

3 Types of functional equations 49

3.1 Introduction 49

3.2 Types of functional equations considered 50

3.3 Description of the program 53

3.4 Explanation of the program 57

4 m-convex hulls of sets of points **65**

4.1 Introduction 65

4.2 The concept of m-convexity 66

4.3 The animation of the m-convex hull of sets on the Cartesian two-dimensional plane 67

4.4 The animation of the m-convex hull of sets on the Cartesian three-dimensional space 68

4.5 Description of the program functions `mconvex2d`, `mconvex3d` and the package `MConvexHull` 69

4.6 Barycentric coordinate system 75

4.7 Description of the program functions `PointInMCVHull2d` and `PointInMCVHull3d` 80

4.8 Explanation of the program 84

Summary **97**

References **108**

Appendix A List of author's publications **109**

Appendix B The package FunctionalEquations **113**

Appendix C The package MConvexHull **203**

Introduction

Mathability is a concept within the field of cognitive infocommunications that examines how natural (human) and artificial (machine) cognitive abilities can work together in mathematical thinking. This includes a wide range of activities, from basic arithmetic to advanced symbolic reasoning. The term was first introduced in a paper [10], in 2013. Later, the concept of mathability has been studied by several authors, including [12], [13], [14], [19], [20], and [48]. In recent years, many researchers have focused on using computer-based tools to investigate mathematical problems in connection with mathability.

The rapid development in information technology and computer sciences over the past decades has significantly impacted all majors and disciplines. Natural sciences, especially mathematics, are not an exception.

In the 1970s, when the proof of the Four-Color Theorem with the aid of the computer was announced, many opposing opinions about the reliability and accuracy of the proof were raised. Until today, a huge number of mathematical studies that have the assistance of computers or computer programs have been published.

However, one of the mathematical problems that computers still find challenging is functional equations. In the 1980s, a question was raised by János Aczél, whether it is possible to solve functional equations with the help of computer programs. Since then, many researchers have used computers as assistants for their studies of functional equations and related topics ([7–

9, 15–17, 21, 40, 42–44, 46, 47, 57, 61–64]).

The main goal of this dissertation is to perform some studies on functional equations and their related topic – convexity, with the help of computers. More precisely, we introduce the computer packages developed in the Computer Algebra system Maple that investigate some specific functional equations and convexity problems. The programs are compatible with Maple 2021 and later versions. They enrich the field of computer algebraic investigations of functional equations and inequalities described.

Maple, developed by Maplesoft, is a commercial computer algebra system widely used in education and scientific research, especially in mathematics. It was first developed in late 1980 at the University of Waterloo (Canada). Then, Maplesoft continued developing it and sold it commercially in 1988. The newest version of Maple was released in 2024.

The most important property of Maple is the ability to perform symbolic computations, which means Maple can handle mathematical expressions exactly, without approximations. All of our computer programs use symbolic computations. As a result, they will give the exact solutions to the given problems. By this property, using the terminology of 'mathability', they 'increases the level of mathability' (or 'improves the mathability') of the underlying systems.

This dissertation is divided into four chapters, the first three chapters containing results on the topic of functional equations, and the last chapter relating to the convexity area, which plays a very important role in the theory of functional equations.

In Chapter 1, we present a computer program that investigates the alienness and strong alienness of linear functional equations of the type

$$\sum_{i=0}^{n+1} f_i(p_i x + q_i y) = 0 \qquad (x, y \in X), \tag{1}$$

where n is a positive integer, $p_0, \ldots, p_{n+1}$ and $q_0, \ldots, q_{n+1}$ are

rational numbers, X, Y are linear spaces and $f_0, \ldots, f_{n+1} : X \to Y$ are unknown functions. This chapter includes theoretical results about linear functional equations of type (1), our extension of the concepts of alienness and strong alienness that were introduced by J. Dhombres in 1988 ([24]), and the full description of a computer program **isalien** with some explanations on the program's code.

In Chapter 2, we introduce a computer program that investigates solutions of Levi-Civita type functional equations, i.e., the class

$$f(x+y) = \sum_{i=1}^{n} g_i(x)h_i(y), \tag{2}$$

where n is a positive integer, G is an Abelian group and $f, g_i, h_i : G \to \mathbb{C}$ $(i = 1, 2, \ldots, n)$ are unknown functions. The theoretical results about solutions of Levi-Civita type functional equations (2) determined by L. Székelyhidi ([72], [70]), with examples on solving some well-known Levi-Civita functional equations, are included. The main focus of this chapter is the full description of the computer program **LCfesolve**. At the end of the chapter, the explanation of how the program works is presented.

In Chapter 3, a computer program that determines types of functional equations is mentioned. In this chapter, we briefly review some well-known functional equation types and classes considered in our program. Similarly to the previous chapters, detailed descriptions of computer programs **fetype**, **fewhattype**, **feinfo** are the central part of the chapter. In the final, we show some program codes with detailed explanations.

The last chapter, Chapter 4, relates to the convexity topic. In this chapter, we first present computer programs, **mconvex2d** and **mconvex3d**, that visualize the animation of the m-convex hull of sets of points on the Cartesian coordinate systems for a two-dimensional plane and a three-dimensional space. According to G. Toader [73], let $m \in [0, 1]$ be a fixed real

number, a set $C \subseteq \mathbb{R}^n$ is called m-convex if for each $t \in [0,1]$

$$tx + m(1-t)y \in C, \quad \text{for all } x, y \in C.$$

Moreover, this chapter also includes the description of two other programs, **PointInMCVHull2d** and **PointInMCVHull3d**, which determine whether a given point is an element of the m-convex hull of a set of points or not in both two and three-dimensional cases. Similarly to the previous chapters, we present some technical clarifications of our functions.

Chapter 1

Alienness of linear functional equations

In this chapter, we present a computer program, developed in the computer algebra system Maple, that investigates the alienness and strong alienness of linear functional equations. The results discussed in this chapter are based on the paper [40].

1.1 Introduction

The concepts of alienness and strong alienness of functional equations were introduced by J. Dhombres in his paper [24] in the following form. Let $E_1(f) = 0$ and $E_2(f) = 0$ be two functional equations for a function f defined on a nonempty set X and mapping to a groupoid Y with an identity element 0. The equations E_1 and E_2 are called alien with respect to X and Y, if each solution $f : X \to Y$ of

$$E_1(f) + E_2(f) = 0$$

is a solution of the system

$$\begin{aligned} E_1(f) &= 0 \\ E_2(f) &= 0. \end{aligned}$$

The equations E_1 and E_2 are said to be strongly alien with respect to X and Y, if all solutions $f, g : X \to Y$ of

$$E_1(f) + E_2(g) = 0$$

are solutions of the system

$$\begin{aligned} E_1(f) &= 0 \\ E_2(g) &= 0. \end{aligned}$$

These properties of functional equations have been investigated by several authors during the last decades (cf., e.g., [5, 6, 32, 33, 35, 49, 58, 59, 66]).

We note that in those publications, mainly the term 'alienation' was used for the concept which is referred to here as alienness.

In this chapter, we consider the alienness phenomenon for linear functional equations of the type

$$\sum_{i=0}^{n+1} f_i(p_i x + q_i y) = 0 \qquad (x, y \in X), \tag{1.1}$$

where n is a positive integer, $p_0, \ldots, p_{n+1}$ and $q_0, \ldots, q_{n+1}$ are rational numbers, X, Y are linear spaces over the field of rational numbers and $f_0, \ldots, f_{n+1} : X \to Y$ are unknown functions.

Classical results on the class of functional equations above and on its important sub-classes were published by M. Fréchet ([29], [30]) and W. H. Wilson ([77]). Their solutions, under general conditions, were determined by L. Székelyhidi ([69], [72]; cf., also, [71] and [68]). Recent studies of (1.1) and its generalizations were performed, among others, in [36], [45], [61] and [67]. The alienness property, connected to linear functional equations, was also considered in several papers (e.g. in [4], [27], [34] and [67]).

We aim to present a computer program developed in the computer algebra system Maple which investigates the alienness and strong alienness of linear functional equations belonging to class (1.1).

In the first part of this chapter, we overview some basic properties of linear functional equations, which are important for our investigations. In Section 1.3 we define some extensions of the concept of alienness of functional equations. The primary focus of this chapter will be Section 1.4, where we present our program. Finally, in the last section, Section 1.5, we explain how our program works.

1.2 Basic properties of linear functional equations

In this section, we present some general theorems on solutions of linear functional equations of type (1.1). In order to formulate them, we give some basic definitions.

Let X, Y be linear spaces over the field of rational numbers, $f : X \to Y$ be a function. Let us define the translation function T

$$T_y f(x) = f(x+y), \qquad (x, y \in X) \tag{1.2}$$

and the difference function Δ

$$\begin{aligned} \Delta_y^1 f(x) = \Delta_y f(x) &= T_y f(x) - f(x) \\ &= f(x+y) - f(x), \qquad (x, y \in X) \end{aligned} \tag{1.3}$$

and for a positive integer n,

$$\Delta_y^{n+1} f(x) = \Delta_y^1 \Delta_y^n f(x), \qquad (x, y \in X) \tag{1.4}$$

For an arbitrary positive integer n, it can be proved by induction that

$$\Delta_y^n f(x) = \sum_{k=0}^{n} (-1)^{n-k} \binom{n}{k} f(x+ky), \qquad (x, y \in X) \tag{1.5}$$

If n is a non-negative integer, a function $f : X \to Y$ is called a polynomial function of degree n if it satisfies

$$\Delta_y^{n+1} f(x) = 0 \qquad (x, y \in X),$$

f is said to be a monomial function of degree n if it fulfills

$$\Delta_y^n f(x) - n! f(y) = 0 \qquad (x, y \in X).$$

Now we formulate some fundamental results on solutions of functional equations of type (1.1) determined by L. Székelyhidi ([69], [72]; cf., also, [77]). In these theorems and the following parts of the chapter, we use the convention $0^0 = 1$.

Theorem 1. *Let X and Y be linear spaces over the field of the rationals, let $p_0, p_1, \dots, p_{n+1}$ and $q_0, q_1, \dots, q_{n+1}$ be rational numbers and assume that*

$$M_k^{(i)} : X \to Y$$

are monomial functions of degree k for $i = 0, \dots, n+1$ and $k = 0, \dots, n$. The functions $f_0, \dots, f_{n+1} : X \to Y$,

$$f_i(x) = \sum_{k=0}^{n} M_k^{(i)}(x) \quad (x \in X,\ i = 0, \dots, n+1)$$

solve functional equation (1.1) if and only if the monomial functions $M_k^{(i)} : X \to Y$ given above fulfill

$$\sum_{i=0}^{n+1} p_i^j q_i^{k-j} M_k^{(i)}(x) = 0 \qquad (x \in X,\ k = 0, \dots, n,\ j = 0, \dots, k).$$

Theorem 2. *Let X and Y be linear spaces over the field of the rationals, let $p_0, p_1, \dots, p_{n+1}$ and $q_0, q_1, \dots, q_{n+1}$ be rational numbers that satisfy*

$$p_i q_j \neq p_j q_i \quad (i, j = 0, \dots, n+1,\ i \neq j). \tag{1.6}$$

The functions $f_0, \dots, f_{n+1} : X \to Y$ solve functional equation (1.1) if and only if they are of the form

$$f_i(x) = \sum_{k=0}^{n} M_k^{(i)}(x) \qquad (x \in X,\ i = 0, \dots, n+1),$$

where

$$M_k^{(i)} : X \to Y \quad (i = 0, \dots, n+1,\ k = 0, \dots, n)$$

are monomial functions of degree k satisfying the equations

$$\sum_{i=0}^{n+1} p_i^j q_i^{k-j} M_k^{(i)}(x) = 0 \qquad (x \in X,\ k = 0, \dots, n,\ j = 0, \dots, k).$$

1.3 Extensions of the concepts of alienness and strong alienness

With the help of computers, one can consider alienness and strong alienness properties for more than two functional equations and for more than one (or two) functions as well. In order to perform such investigations, we extend the corresponding definitions of J. Dhombres ([24]) in the following way.

Let m, n be positive integers ($m \geq 2$) and let

$$\begin{aligned} E_1(f_1, \dots, f_n) &= 0 \\ &\vdots \\ E_m(f_1, \dots, f_n) &= 0 \end{aligned}$$

be functional equations for the unknown functions $f_1, \dots, f_n$ defined on a nonempty set X and mapping to a groupoid Y with an identity element 0.

Definition 1. The equations $E_1, \dots, E_m$ are called alien with respect to X and Y, if all solutions $f_i : X \to Y$, $(i = 1, \dots, n)$, of

$$E_1(f_1, \dots, f_n) + \cdots + E_m(f_1, \dots, f_n) = 0 \tag{1.7}$$

are solutions of the system

$$\begin{aligned} E_1(f_1, \dots, f_n) &= 0 \\ &\vdots \\ E_m(f_1, \dots, f_n) &= 0, \end{aligned} \tag{1.8}$$

too.

Definition 2. The equations $E_1, E_2, \dots, E_m$ are said to be strongly alien with respect to X and Y, if all solutions

$f_{ij} : X \to Y$, $(i = 1, \ldots, m,\ j = 1, \ldots, n)$ of

$$E_1(f_{11}, \ldots, f_{1n}) + \cdots + E_m(f_{m1}, \ldots, f_{mn}) = 0 \tag{1.9}$$

are solutions of the system

$$\begin{aligned} E_1(f_{11}, \ldots, f_{1n}) &= 0 \\ &\vdots \\ E_m(f_{m1}, \ldots, f_{mn}) &= 0 \end{aligned} \tag{1.10}$$

as well.

1.4 Description of the program function `isalien`

In this section, we describe the computer program `isalien`, which investigates the alienness and strong alienness of linear functional equations of type (1.1).

The program extensively uses another user-defined Maple program `lfesolve`, which determines solutions of functional equations belonging to class (1.1). (Cf. [17], furthermore, [15], [16], [43] and [44].) The function `lfesolve` is contained in the Maple package `FunctionalEquations`, sub-package `LFESolve`. Like other Maple programs, it can be accessed by the form

```
> with(FunctionalEquations:-LFESolve);
> lfesolve(arguments);
```

To use the program `lfesolve`, users have to give at least two input parameters: the system of linear functional equations to be solved as a Maple list called `e` and the list of the unknown functions included in the given system `f`. In addition, optional parameters can be used. Here, all equations in the given system must be in type (1.1) and the variables of unknown functions of the system should be denoted by `x` and `y`.

User option includes three part:

– OUTPUT = {'`brief`', '`verbose`'}

– METHOD = {'`strict`', '`permissive`'}

– MAXDEG : positive integer number

Default options: ['`brief`', '`strict`', `infinity`].

In case, all the input parameters are given correctly, the program will provide the solution of the system of functional equations `e`. According to Theorem 1 and 2, the solution are presented in the form of linear combination of monomial functions. The monomial function of degree k $M_k^i, (i = 0, \ldots, n+1, k = 0, \ldots, n)$ will be presented in Maple in the form `M(i,k)`. In the following part, some examples will be shown.

```
> fe := f(x + 3y) + f(y) - f(3x + 2y)
        + g(x - y);
> lfesolve(fe, [f, g], 'verbose');
```

yields

```
The form of the solutions is
     f = M(0, 0) + M(0, 1) + M(0, 2)
     g = M(1, 0) + M(1, 1) + M(1, 2)
where M(.,k) are monomial functions of degree
k, for which
     M(0, 0) = –M(1, 0)
     M(0, 2) = 0
     M(1, 1) = 2 M(0, 1)
     M(1, 2) = 0
Thus,
     f = –M(1, 0) + M(0, 1)
     g = M(1, 0) + 2 M(0, 1)
     {f = –M(1, 0) + M(0, 1),
          g = M(1, 0) + 2M(0, 1)}.
```

In case, the condition (1.6) is not valid, with the option OUTPUT 'verbose', the program `lfesolve` will show a warning as well as the detail solution of the given functional equation.

```
> fe := f(2*x + y) + f(x - y) + g(y)
```

```
        - g(-x + y);
> lfesolve(fe, [f, g], 'verbose');
```

yields

```
Warning, for functional equations of this
type the program may not provide all
solutions. The form of the polynomial
solutions of degree at most 2 is
     f = M(0, 0) + M(0, 1) + M(0, 2)
     g = M(1, 0) + M(1, 1) + M(1, 2)
where M(.,k) are monomial functions of degree
k, for which
     M(0, 0) = 0
     M(0, 2) = 0
     M(1, 1) = -3 M(0, 1)
     M(1, 2) = 0
Thus,
     f = M(0, 1)
     g = M(1, 0) - 3M(0, 1)
     {f = M(0, 1), g = M(1, 0) - 3 M(0, 1)}.
```

More information about the function `lfesolve` with examples and detail explanations can be found in paper [17].

Similarly to `lfesolve`, `isalien` is contained in the Maple package `FunctionalEquations`, sub-package `AlienationCheck`. Analogously to other Maple programs, the function `isalien` can be accessed by the forms

```
> with(FunctionalEquations:-AlienationCheck);
> isalien(arguments);
```

To run the program, users have to give at least two input parameters: the list of linear functional equations as a Maple list called `e` and the list of the unknown functions included in the given equations called `f`. Both of the lists `e` and `f` have to be given as a Maple list structure between square brackets, all

elements in the list are separated by a comma. In addition, optional parameters can be used.

Moreover, similarly to `lfesolve` all of the input functional equations have to belong to the class (1.1) and the variables of all unknown functions of equations in `e` should be represented by 'x' and 'y'. An important property of `e` is that all of the functional equations in `e` must have the same number of unknown functions.

Some examples of the correct form of `e` and `f`

$$\begin{aligned} e &= [f_1(x+y) - f_1(x) - f_2(y) = 0, \\ &\qquad g_1(x-2y) + 3g_2(x) = 0], \\ f &= [[f_1, f_2], [g_1, g_2]] \end{aligned} \tag{1.11}$$

or

$$\begin{aligned} &e = [f(x+y) - f(x) - f(y) = 0, g(x-2y) + 3g(x) = 0], \\ &f = [[f], [g]] \end{aligned}$$

or simply

$$\begin{aligned} e &= [f(x+y) - f(x) - f(y), g(x-2y) + 3g(x)], \\ f &= [f, g]. \end{aligned}$$

We note that the order of unknown functions in list `f` is also important, since it will affect the final solution (cf. the definitions in Section 1.3). For example, the solution for the input (1.11) and the input

$$\begin{aligned} e &= [f_1(x+y) - f_1(x) - f_2(y) = 0, \\ &\qquad g_1(x-2y) + 3g_2(x) = 0], \\ f &= [[f_1, f_2], [g_2, g_1]] \end{aligned}$$

are different.

Furthermore, according to the definition of strong alienness of functional equations and their extensions, when users want to check the strong alienness of functional equations given in `e`, the unknown functions appearing in the equations in `e` have to be pairwise different.

The program has three types of options:

– CHECK = {'`alien`', '`strongalien`'}

– OUTPUT = {'`brief`', '`verbose`', '`full`'}

– ErrorWarn_STYLE = {'`ewYes`', '`ewNo`'}

The default options are: ['`alien`', '`brief`', '`ewYes`'].

As it was mentioned above, `isalien` uses the Maple function `lfesolve`, which determines the exact solutions of systems of linear functional equations containing equations of type (1.1). As a result, our approach for the investigations is solving the functional equation (1.7) and the system of functional equation (1.8) and then comparing their solution, in case of strong alien, (1.9) and (1.10), respectively.

It turned out, that warning and error messages of the program `lfesolve` could be misleading for the users in the output of `isalien`. For this reason, we decided to modify `lfesolve` accordingly. With the new version, we can manage the warning and error messages directly in our main program and any other applications in the future.

If the input is given correctly, the program will provide an answer about the alienness or the strong alienness of the functional equations given in `e`. In the following, some examples with different options and their meanings will be shown.

When users give values for the parameters `e` and `f` only, the default values of the options will be used. This means that the program will investigate the alienness of the given equations and return an answer `True` or `False`. For example:

```
> E1 := f(x + y) − f(x) − f(y) = 0:
  E2 := 2g(x − 3y) − g(3x − 8y)
        − g(−x + 2y) = 0:
  isalien([E1, E2], [f, g]);
```

gives

```
True.
```

For more information about the solutions of the functional equations considered, the OUTPUT option has to be modified into '`verbose`' or '`full`'. E.g.,

```
> E1 := f(x + y) − f(x) − f(y) = 0:
  E2 := 2g(x − 3y) − g(3x − 8y)
        − g(−x + 2y) = 0:
  isalien([E1, E2], [f, g], verbose);
```

gives

```
The functional equations
E1:
     f(x + y) − f(x) − f(y) = 0
E2:
     2f(x − 3y) − f(3x − 8y)
          − f(−x + 2y) = 0
are alien.
```

With the option '`full`', the program will present an output which contains the solutions of the functional equation $E_1 + \cdots + E_m = 0$ and the solutions of the system $[E_1 = 0, \ldots, E_m = 0]$.

```
> E1 := f(x + y) − f(x) − f(y) = 0:
  E2 := 2g(x − 3y) − g(3x − 8y)
        − g(−x + 2y) = 0:
  isalien([E1, E2], [f, g], full);
```

yields

```
The functional equations
E1:
     f(x + y) − f(x) − f(y) = 0
E2:
     2f(x − 3y) − f(3x − 8y)
          − f(−x + 2y) = 0
are alien
In details,
```

```
The solution of the functional equation
E1 + E2 = 0 and the solution of the
system [E1 = 0, E2 = 0] is
    {f = M(0, 1)}.
```

For a decision about the strong alienness of functional equations, users can use the CHECK option with value '`strongalien`' combined with the appropriate OUTPUT option as mentioned above. For example,

```
> E1 := f(x + y) − f(x) − f(y) = 0:
  E2 := 2g(x − 3y) − g(3x − 8y)
        − g(−x + 2y) = 0:
  isalien([E1, E2], [f, g], strongalien);
```

gives

```
True.
```

```
> E1 := f(x + y) − f(x) − f(y) = 0:
  E2 := 2g(x − 3y) − g(3x − 8y)
        − g(−x + 2y) = 0:
  isalien([E1, E2], [f, g], strongalien,
        full);
```

returns

```
The functional equations
E1:
    f(x + y) − f(x) − f(y) = 0
E2:
    2g(x − 3y) − g(3x − 8y)
        − g(−x + 2y) = 0
are strongly alien
In details,
The solution of the functional equation
E1 + E2 = 0 and the solution of the
system [E1 = 0, E2 = 0] are
    {f = M(0, 1), g = M(1, 0) + M(1, 1)}.
```

```
> E1 := f(x + y) - f(x) - f(y) = 0:
  E2 := g(2x + 3y) - 2g(x - y) = 0:
  isalien([E1, E2], [f, g], strongalien,
          full);
```

gives

```
      The functional equations
      E1:
           f(x + y) - f(x) - f(y) = 0
      E2:
           g(2x + 3y) - 2g(x - y) = 0
      are not strongly alien
      In details,
      The solution of the functional equation
      E1 + E2 = 0 are
           {f = - M(1, 0) + M(0, 1),
                g = M(1, 0)}
      while the solution of the system [E1 = 0,
      E2 = 0] are
           {f = M(0, 1), g = 0}.
```

As mentioned in Section 1.3, our program is able to handle situations when the input contains more than two functional equations and each equation has more than one unknown function. In the following example, the program works with four functional equations

```
> E1 := f(x + y) - f(x) - f(y) = 0:
  E2 := g(x + 5y) + 2g(2x - y)
        - 2g(x - y) = 0:
  E3 := h(2x - 3y) - h(2x + 7y) = 0:
  E4 := k(x + 3y) + 2k(x - y) = 0:
  isalien([E1, E2, E3, E4], [f, g, h, k],
          verbose);
```

yields

```
The functional equations
E1:
     f(x + y) − f(x) − f(y) = 0
E2:
     f(x + 5y) + 2f(2x − y)
          − 2f(x − y) = 0
E3:
     f(2x − 3y) − f(2x + 7y) = 0
E4:
     f(x + 3y) + 2f(x − y) = 0
are alien.
```

In case the equations contain two unknown functions,

```
> E1 := f_1(x + y) − f_1(x) − f_1(y)
        + f_2(2x + 5y) − f_2(x − 6y) = 0:
  E2 := g_1(2x + y) − g_1(x − 3y)
        − g_1(x − 2y) + g_2(x − y)
        − g_2(x + 2y) = 0:
  isalien([E1, E2], [[f_1, f_2], [g_1, g_2]],
        full);
```

has the output

```
The functional equations
E1:
     f_1(x + y) − f_1(x) − f_1(y)
          + f_2(2x + 5y) − f_2(x − 6y) = 0
E2:
     f_1(2x + y) − f_1(x − 3y)
          − f_1(x − 2y) + f_2(x − y)
          − f_2(x + 2y) = 0
are alien
In details,
The solution of the functional equation
E1 + E2 = 0 and the solution of the
system [E1 = 0, E2 = 0] are
```

```
{f_1 = 0, f_2 = M(1, 0)}.
```

As mentioned before, our program decides about the alienness or strong alienness of the linear functional equations considered by comparing the solutions of equation (1.7) or (1.9) and the solutions of the systems of equation (1.8) or (1.10), respectively. According to Theorem 2, we can find all solutions of functional equations which belong to class (1.1) in case these functional equations fulfill condition (1.6). On the other hand, if the condition (1.6) is not valid, by Theorem 1, we are still able to determine all polynomial solutions of (1.1). This means that, in case condition (1.6) is not valid in (1.7) or (1.9), we can only find its polynomial solutions and cannot decide whether the remaining solutions are also solutions of (1.8) and (1.10), respectively, or not. Therefore, the final conclusion about the alienness and the strong alienness might be affected.

If there exists a polynomial solution of (1.7) or (1.9) which do not fulfill (1.8) or (1.10), respectively, we can return that the given functional equations are not alien or not strongly alien, respectively. E.g.,

```
> E1 := f(x + y) - f(2x + y) + f(x) = 0:
  E2 := g(x + 2y) - g(y) - g(x) = 0:
  E3 := h(2x - 3y) - h(2x + 7y) = 0:
  isalien([E1, E2, E3], [f, g, h],
          strongalien, full);
```

gives

```
The functional equations
E1:
     f(x + y) - f(2x + y) + f(x) = 0
E2:
     g(x + 2y) - g(y) - g(x) = 0
E3:
     h(2x - 3y) - h(2x + 7y) = 0
are not strongly alien
```

```
In details,
The polynomial solution of the functional
equation E1 + E2 + E3 = 0 are
    {f = M(1, 0) + M(0, 1), g = M(1, 0)
    + 10M(2, 1), h = M(2, 0) + M(2, 1)}
while the solution of the system [E1 = 0,
E2 = 0, E3 = 0] are
    {f = M(0, 1), g = 0, h = M(2, 0)}.
```

However, if all polynomial solutions of (1.7) or (1.9) satisfy (1.8) or (1.10), respectively, the program would be unable to decide about the alienness or the strong alienness of the functional equations given. In this case, the program will provide a warning message to announce the problem and will determine a result, if polynomial solutions are considered only.

```
> E1 := f_1(2x - y) - 2f_1(x) + f_1(y)
        + f_2(x + 5y) + 2f_2(2x - 7y)
        - 2f_2(x) - f_2(y) + f_3(2x - 3y)
        - f_3(y) = 0:
  E2 := g_1(x + y) - g_1(x) - g_1(y)
        + g_2(x + y) + 2g_2(x - y) - 3g_2(x)
        - g_2(y) + g_3(2x - y) - 2g_3(y) = 0:
  isalien([E1, E2], [[f_1, f_2, f_3],
        [g_1, g_2, g_3]], full);
```

gives

```
The functional equations
E1:
    f_1(2x - y) - 2f_1(x) + f_1(y)
        + f_2(x + 5y) + 2f_2(2x - 7y)
        - 2f_2(x) - f_2(y)
        + f_3(2x - 3y) - f_3(y) = 0
E2:
    f_1(x + y) - f_1(x) - f_1(y)
        + f_2(x + y) + 2f_2(x - y)
```

```
                - 3f_2(x) - f_2(y)
                + f_3(2x - y) - 2f_3(y) = 0
      are alien if polynomial solutions are
      considered only.
      In details,
      The polynomial solution of the functional
      equation E1 + E2 = 0 and the polynomial
      solution of the system [E1 = 0, E2 = 0]
         are
          {f_1 = - M(1, 0) - M(2, 0) + M(0, 1),
               f_2 = M(1, 0), f_3 = M(2, 0)}.
```

In the following example, the functional equation $E1+E2 = 0$ yields $0 = 0$, which means that an arbitrary function f satisfies this equation. Compared to the results of the system of functional equations $[E1 = 0, E2 = 0]$, the program will give the conclusion that the given functional equations $E1$ and $E2$ are not alien.

```
> E1 := f(x + y) - f(x) - f(y) = 0:
  E2 := - g(x + y) + g(x) + g(y) = 0:
  isalien([E1, E2], [f, g], full);
```

we obtain

```
      The functional equations
      E1:
           f(x + y) - f(x) - f(y) = 0
      E2:
          - f(x + y) + f(x) + f(y) = 0
      are not alien
      In details,
      The solution of the functional equation
      E1 + E2 = 0 are arbitrary || [f]
      while the solution of the system [E1 = 0,
      E2 = 0] is
           {f = M(0, 1)}.
```

Obviously, besides checking the alienness and strong alienness of linear functional equations 'directly', our program can also be used by other applications. In such cases, it might be confusing, if the error or warning messages appear in the form mentioned above (e.g., in the application output using our program). To be able to manage such situations, we added one more parameter in the options part of the program called ErrorWarn STYLE, which accepts two values: '`ewYes`' and '`ewNo`'. This option can be used to switch on and switch off the warning and error messages in the output. In the following part, some examples relating to that option will be given.

```
> E1 := f(x + y) - g(x + 2y) + g(x) = 0:
  E2 := h(2x + 2y) + k(y) - k(2x - 3y) = 0:
  isalien([E1, E2], [[f, g], [h, k]], ewNo);
```

yields

```
    True (If polynomial solutions are
       considered only).
```

In the case when the input is incorrect, the program will show an appropriate error message with a short hint about the problem (following the built-in error message's structure of Maple). For example,

```
> isalien([0 = 0, g(x + y) + g(x) + g(y) = 0],
        [f, g], full);
```

has the output

```
    Error, (in error_management) [07]
        invalid input: 0
```

the input

```
> E1 := f(x + y) - f(x) - f(y) = 0:
  E2 := g(x + y) - g(x) - h(y) = 0:
  isalien([E1, E2], [f, g], full);
```

gives

```
Error, (in error_management) [12] invalid
input in list of unknown functions.
```

and

```
> E1 := f(2x) - f(x) + f(y) = 0:
  E2 := 2g(x) - g(y) - g(x + 5y) = 0:
  isalien([E1, E2], [f,g], full, abc);
```

yields

```
Error, (in error_management) invalid
arguments in the input, [full, abc].
```

moreover

```
> E1 := f(2xy) - 2f(x) + f(y) = 0:
  E2 := 2g(x) - g(y) - g(x + 5y) = 0:
  isalien([E1, E2], [f,g], full);
```

gives

```
Error, (in error_management) [06] invalid
argument: 2xy in f(2xy).
```

The option ErrorWarn_STYLE can be used to switch off error messages and to return error codes (which can be used by other applications):

```
> E1 := f(2xy) - 2f(x) + f(y) = 0:
  E2 := 2g(x) - g(y) - g(x + 5y) = 0:
  isalien([E1, E2], [f,g], full, ewNo);
```

yields

```
["E06", [2xy, f(2xy)]].
```

1.5 Explanation of the program function `isalien`

In this section, the main procedure code will be shown with detailed explanations to illustrate how our program `isalien` works.

```
1  test := proc(equations, functions)
2     local lequations, lfunctions,
         lfunctions_new, new_input, listOfEqs,
         listOfFuncs, numOfEqs, E, i, A, B, b,
         Remark, WarningList;
3     lequations := convert(equations, list);
4     lfunctions := convert(functions, list);
5     lfunctions_new :=
         normalize_listOfFunctions(lequations,
         lfunctions);
6     if CHECK = 'alien' then
7        new_input :=
            normalize_input_alien(lequations,
            lfunctions_new);
8     else
9        if nops(Flatten(lfunctions_new)) <>
            numelems(convert(Flatten(
            lfunctions_new), set)) then
10           return error_management("E17",
                lfunctions_new);
11       else new_input := [lequations,
            lfunctions_new];
12       end if;
13    end if;
14    listOfEqs := convert(new_input[1], list);
15    listOfFuncs :=
         Flatten(convert(new_input[2], list));
16    numOfEqs := nops(listOfEqs);
17    E := 0;
18    for i to numOfEqs do
19       E := E + listOfEqs[i];
20    end do;
21    A := lfesolve(listOfEqs, listOfFuncs,
         'errorNo');
```

```
22    B := lfesolve([E], listOfFuncs,
          'errorNo');
23    if type(A[1], string) then
24       return error_management(A[1], A[2]);
25    end if;
26    if type(B[1], string) then
27       if B = ["E07", 0] then
28          b := cat(arbitrary, listOfFuncs);
29          return [false, listOfEqs, [A[1],
               b]];
30       end if;
31       return error_management(B[1], B[2]);
32    end if;
33    if evalb("W02" in Flatten(B)) then
34       return error_management("E12");
35    end if;
36    Remark := 0;
37    if nops(A[2]) <> 0 then
38       WarningList := [op(WarningList), A[2]];
39    end if;
40    if nops(B[2]) <> 0 then
41       WarningList := [op(WarningList), B[2]];
42    end if;
43    if A[1] = B[1] then
44       if evalb("W01" in Flatten(B)) then
45          Remark := 1;
46       end if;
47       return [true, listOfEqs, [A[1]]];
48    else
49       if evalb("W01" in Flatten(B)) then
50          if evalb("W01" in Flatten(A)) then
51             Remark := -2;
52          else Remark := -1;
53          end if;
```

```
54        end if;
55        return [false, listOfEqs, [A[1],
              B[1]]];
56    end if;
57 end proc;.
```

As mentioned in Section 1.4, our program determines the alienness and strong alienness of linear type functional equations by comparing the solution of the equations (1.7) or (1.9) with the solution of the system of equations (1.8) or (1.10), respectively. Therefore, the main structures of the procedure are as follows:

- Lines 3 – 16: Preprocessing the given inputs to get the proper arguments for the main steps. More precisely, in the case of checking the 'alien', after the first steps, the procedure will make sure that all functional equations will share the same list of unknown functions (lines 6 – 7). And, an error will show up in case the given list of unknown functions are incorrect by both numbers or names of them (lines 9 – 12). Finally, the correct forms of arguments will be made (lines 14 – 15).

- Lines 17 – 22: Solving the equations (1.7) or (1.9) and the system of equations (1.8) or (1.10). Lines 17 – 20, the system of equations (1.8) or (1.10) is generated from the given list. Then we call the program `lfesolve` (which is mentioned in the previous section) to determine the solutions of (1.7) or (1.9) and the system of equations (1.8) or (1.10) (lines 21 – 22). Then we have two sets of solutions, A and B.

- Lines 23 – 50: Comparing the previous solutions A, B and catching errors if appeared (line 23 – 34). Then, the alienness and strong alienness of the given functional equations are determined (lines 35 – 50).

The `isalien` is built up by many more procedures that manage the appeared errors, warnings, or manipulating the raw inputs and final outputs or some necessary technical issues. However, the code presented in this section is the main procedure that can somehow point out how our program works. Generally, the main structure of function isalien is as follow

```
58 isalien := proc()
59     local solution;
60     #processing the input arguments args
61     solution := test(args[1], args[2]);
62     # managing the solution (alien,
           strongalien, warnings, errors, output)
63 end proc;.
```

In the input processing step (line 60), the program will check the validity of the given input arguments. As mentioned in Section 1.4, there are many options for input, and the number of input arguments can vary from two to five. For example, when only a list of functional equations and a list of unknown functions are given (line 64 nargs = number of input = 2) arguments, then all of the rest arguments will be set up as the default option (lines 65 – 67).

```
64 if nargs = 2 then
65     CHECK := 'alien';
66     OUTPUT := 'brief';
67     ErrorWarn_STYLE := 'ewYes';.
```

Similarly, the program will check the condition for `nargs = 3, 4, 5` and manage also the cases when the number of arguments is not valid `nargs < 2` or `nargs > 5`. All other conditions about the correctness of each argument were managed in the previous procedure test.

After processing the input parameters, the function test was called as the main step (line 61). As a result, variable `solution` is a list of three elements

– [true, listOfEqs, [A[1]]] i.e, given functional equations are alien or strongly alien,

– [false, listOfEqs, [A[1], B[1]]] i.e, given functional equations are not alien or not strongly alien.

Based on that `solution` variable, in isalien we manage the output in each case. For example, in case the given functional equations are alien or strongly alien

```
68 printf("The functional equations \n");
69 for i to nops(solution[2]) do
70    printf("E%d:",i);
71    print(solution[2][i]);
72 end do;
73 if solution[1] then
74    if CHECK = 'alien' then
75       printf("\n are alien");
76    else
77       printf("\n are strongly alien");
78    end if;
79    if OUTPUT='full' then
80       if nops(solution[2])=2 then
81          printf("\n In details,\n ");
82          printf("The solution of the
                functional equation E1 + E2 = 0
                and the solution of the system
                [E1 = 0, E2 = 0]");
83          if nops(solution[3][1])=1 then
84             printf(" is");
85          else printf(" are");
86          end if;
87          print(solution[3][1]);
88       end if;
89    end if;
90 end if;.
```

To make it clear, let us bring back one example from Section 1.4

```
> E1 := f(x + y) - f(x) - f(y) = 0:
  E2 := 2g(x - 3y) - g(3x - 8y)
        - g(-x + 2y) = 0:
  isalien([E1, E2], [f, g], strongalien,
          full);
```

returns

```
      The functional equations
      E1:
           f(x + y) - f(x) - f(y) = 0
      E2:
           2g(x - 3y) - g(3x - 8y)
               - g(-x + 2y) = 0
      are strongly alien
      In details,
      The solution of the functional equation
      E1 + E2 = 0 and the solution of the
      system [E1 = 0, E2 = 0] are
           {f = M(0, 1), g = M(1, 0) + M(1, 1)}.
```

- Lines 68 – 72: showing the first part of the output, giving information about the considered functional equations (i.e., E1, E2).
- Lines 73 – 78: showing the result whether they are alien or strongly alien.
- Lines 79 – 90: in case the output option is `full`, the information about the solution of the function equation `E1 + E2 = 0` and the solution of the system `[E1 = 0, E2 = 0]` is presented.

Remark 1. In the whole program, more complex conditions and cases are considered, to cover all possible cases that can happen.

Chapter 2

Levi-Civita type functional equations

In this chapter, a computer program, developed in the computer algebra system Maple is presented, which investigates solutions of Levi-Civita type functional equations. The results outlined in this chapter are based on the paper [41].

2.1 Introduction

We consider Levi-Civita type functional equations, i.e., the class

$$f(x+y) = \sum_{i=1}^{n} g_i(x)h_i(y), \tag{2.1}$$

where n is a positive integer, G is an Abelian group and $f, g_i, h_i : G \to \mathbb{C}$ $(i = 1, 2, \ldots, n)$ are unknown functions.

This chapter aims to introduce a computer program developed in the Maple computer algebra system, which investigates solutions to Levi-Civita-type functional equations (2.1).

In the first part of the chapter, we provide an overview of some results related to the solutions of Levi-Civita-type functional equations. In Section 2.3, we present examples of solving functional equations that belong to the class (2.1). Following that, Section 2.4 introduces our program. Finally, in Section

2.5, we provide a brief overview of the code to illustrate how our function operates.

2.2 Basic results about solutions of Levi-Civita type functional equations

In this section, we formulate some fundamental results on solutions of functional equations of type (2.1) determined by L. Székelyhidi ([72], [70]).

According to L. Székelyhidi, any function $f : G \to \mathbb{C}$, satisfying the Levi-Cevita functional equation (2.1), is a normal exponential polynomial of order at most n. And the converse remains true: any normal exponential polynomial f fulfills the equation (2.1).

If the functions $h_1, h_2, \ldots, h_n$ are linearly independent, then $g_1, g_2, \ldots, g_n$ are linear combinations of translates of f. Hence, they are also normal exponential polynomial of order at most n. Moreover, they are built up from the same additive and exponential functions, as f.

We call the case when the functions $g_1, g_2, \ldots, g_n$ and the functions $h_1, h_2, \ldots, h_n$ are linearly independent as the non-degenerate case.

In the non-degenerate case, the functions f, g_i, h_i, $(i = 1, \ldots, n)$ have the form:

$$\begin{aligned} f(x) &= \sum_{j=1}^{k} P_j(a_{j,1}(x), \ldots, a_{j,n_j-1}(x))m_j(x) \\ g_i(x) &= \sum_{j=1}^{k} Q_{i,j}(a_{j,1}(x), \ldots, a_{j,n_j-1}(x))m_j(x) \\ h_i(x) &= \sum_{j=1}^{k} R_{i,j}(a_{j,1}(x), \ldots, a_{j,n_j-1}(x))m_j(x), \end{aligned} \tag{2.2}$$

where

– $k, n_1, \dots, n_k$ are positive integers such that $n_1 + \cdots + n_k = n$,

– $m_1, \dots, m_k$ are different non-zero complex exponentials,

– $\{a_{j,1}(x), \dots, a_{j,n_j-1}(x)\}$ are sets of linearly independent real additive functions for $j = 1, \dots, k$,

– $P_j, Q_{i,j}, R_{i,j}$ are complex polynomials of degree at most $n_j - 1$ in $n_j - 1$ variables for $i = 1, \dots, n$, $j = 1, \dots, k$

With the same assumptions, for any $j = 1, \dots, k$ and for any multi-indices $A_j = (\alpha_1, \alpha_2, \dots, \alpha_{n_j-1})$ and $B_j = (\beta_1, \beta_2, \dots, \beta_{n_j-1})$ in $\mathbb{N}^{n_j-1}$, we introduce the matrix $M_j(P; A_j, B_j)$ and the matrix $N_j(Q; A_j)$ as follows:

– for any choice of $p, q = 0, \dots, n_j - 1$,

$$M_j(P; A_j, B_j)_{n_j-p, n_j-q} =$$

$$\begin{cases} \frac{1}{p!q!}\partial_1^{\alpha_1} \dots \partial_p^{\alpha_p} \partial_1^{\beta_1} \dots \partial_q^{\beta_q} P_j(0, \dots, 0) & \text{for } p + q < n_j \\ 0, & \text{for } p + q \geq n_j, \end{cases} \tag{2.3}$$

– for any choice of $p = 1, \dots, n_j$, and $q = 1, \dots, n$,

$$N_j(Q; A_j)_{p,q} = \frac{1}{(n_j - p)!}\partial_1^{\alpha_1} \dots \partial_{n_j-p}^{\alpha_{n_j-p}} Q_{q,p}(0, \dots, 0). \tag{2.4}$$

Then, we define two block matrices

$$M(P; A_1, \dots, A_k, B_1, \dots, B_k) =$$

$$\begin{bmatrix} M_1(P; A_1, B_1) & 0 & \dots & 0 \\ 0 & M_2(P; A_2, B_2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & M_k(P; A_k, B_k) \end{bmatrix}; \tag{2.5}$$

and

$$N(Q; A_1, \ldots, A_k) = \begin{bmatrix} N_1(Q; A_1) \\ \vdots \\ N_k(Q; A_k) \end{bmatrix}. \tag{2.6}$$

Then we have a necessary and sufficient condition in terms of the coefficients of the complex polynomials $P_j, Q_{i,j}, R_{i,j}$ such that f, g_i, h_i is a non-degenerate solution of the Levi-Civita functional equation (2.1)

$$M(P; A_1, \ldots, A_k, B_1, \ldots, B_k) =$$

$$N(Q; A_1, \ldots, A_k) N(R; B_1, \ldots, B_k)^T \tag{2.7}$$

holds for any choice of the multi-indices A_j, B_j in $\mathbb{N}^{n_j - 1}$ $(j = 1, \ldots, k)$.

And the equation (2.7) is equivalent to the system of equations

$$N_p(Q; A_p) N_q(R; B_q)^T = \begin{cases} 0 & \text{for } p \neq q, \\ M_p(P; A_p, B_p) & \text{for } p = q, \end{cases} \tag{2.8}$$

for any $p, q = 1, 2, \ldots, k$ and for any choice of multi-indices A_p in $\mathbb{N}^{n_p - 1}$, B_q in $\mathbb{N}^{n_q - 1}$.

2.3 Examples of solving Levi-Civita type functional equations

2.3.1 The case n = 1

In the case of $n = 1$, we have the well-known functional equation - the Pexider equation

$$f(x + y) = g(x)h(y). \tag{2.9}$$

When $n = 1$, there is only one sub-case for considering, followed by the decomposition of number 1, which is $1 = 1$.

We have the general non-degenerate solution

$$\begin{aligned} f(x) &= \alpha m(x) \\ g(x) &= \beta m(x) \\ h(x) &= \gamma m(x), \end{aligned} \tag{2.10}$$

where m is a nonzero complex exponential, and the constants fulfill the condition

$$\beta\gamma = \alpha.$$

Therefore, we have the only non-degenerate solution

$$\begin{aligned} f(x) &= \beta\gamma m(x) \\ g(x) &= \beta m(x) \\ h(x) &= \gamma m(x), \end{aligned}$$

where m is a nonzero complex exponential and β, γ are constants.

2.3.2 The case n = 2

$$f(x+y) = g_1(x)h_1(y) + g_2(x)h_2(y). \tag{2.11}$$

When $n = 2$, there are two possibilities according to the decomposition of the number 2.

(2a) $2 = 2$,

(2b) $2 = 1 + 1$.

In the case (2a) $2 = 2$, we have the general non-degenerate solution

$$\begin{aligned} f(x) &= (\alpha_1 a(x) + \alpha_2)m(x) \\ g_1(x) &= (\beta_{1,1} a(x) + \beta_{1,2})m(x) \\ g_2(x) &= (\beta_{2,1} a(x) + \beta_{2,2})m(x) \\ h_1(x) &= (\gamma_{1,1} a(x) + \gamma_{1,2})m(x) \\ h_2(x) &= (\gamma_{2,1} a(x) + \gamma_{2,2})m(x), \end{aligned} \tag{2.12}$$

where m is a nonzero complex exponential, a is a nonzero complex additive function, and the constants fulfill the condition

$$\begin{bmatrix} \beta_{1,1} & \beta_{2,1} \\ \beta_{1,2} & \beta_{2,2} \end{bmatrix} \begin{bmatrix} \gamma_{1,1} & \gamma_{1,2} \\ \gamma_{2,1} & \gamma_{2,2} \end{bmatrix} = \begin{bmatrix} 0 & \alpha_1 \\ \alpha_1 & \alpha_2 \end{bmatrix}. \tag{2.13}$$

In the case (2b) $2 = 1+1$, we have the general non-degenerate solution

$$\begin{aligned} f(x) &= \alpha_1 m_1(x) + \alpha_2 m_2(x) \\ g_1(x) &= \beta_{1,1} m_1(x) + \beta_{1,2} m_2(x) \\ g_2(x) &= \beta_{2,1} m_1(x) + \beta_{2,2} m_2(x) \\ h_1(x) &= \gamma_{1,1} m_1(x) + \gamma_{1,2} m_2(x) \\ h_2(x) &= \gamma_{2,1} m_1(x) + \gamma_{2,2} m_2(x), \end{aligned} \tag{2.14}$$

where m_1, m_2 are nonzero complex exponentials, and the constants fulfill the condition

$$\begin{bmatrix} \beta_{1,1} & \beta_{2,1} \\ \beta_{1,2} & \beta_{2,2} \end{bmatrix} \begin{bmatrix} \gamma_{1,1} & \gamma_{1,2} \\ \gamma_{2,1} & \gamma_{2,2} \end{bmatrix} = \begin{bmatrix} \alpha_1 & 0 \\ 0 & \alpha_2 \end{bmatrix}. \tag{2.15}$$

Especially, the two well-known functional equations, sine and cosine equations, are special cases of Levi-Civita type with $n = 2$. In detail,

$$f(x+y) = f(x)g(y) + g(x)f(y), \tag{2.16}$$

and

$$f(x+y) = f(x)f(y) - g(x)g(y). \tag{2.17}$$

Considering the sine equation (2.16), we also have two cases based on the decomposition of the number 2. In the first case (2a) $2 = 2$, applying the general non-degenerate solution (2.12) and condition (2.13) of the constants, we have the non-degenerate solution form of the sine equation (2.16)

$$\begin{aligned} f(x) &= (\alpha_1 a(x) + \alpha_2) m(x) \\ g(x) &= (\gamma_1 a(x) + \gamma_2) m(x), \end{aligned} \tag{2.18}$$

where m is a nonzero complex exponential, a is a nonzero complex additive function, and the constants fulfill the condition

$$\begin{bmatrix} \alpha_1 & \gamma_1 \\ \alpha_2 & \gamma_2 \end{bmatrix} \begin{bmatrix} \gamma_1 & \gamma_2 \\ \alpha_1 & \alpha_2 \end{bmatrix} = \begin{bmatrix} 0 & \alpha_1 \\ \alpha_1 & \alpha_2 \end{bmatrix}. \tag{2.19}$$

From the condition (2.19), we can get the value of the constants by solving the system of equations

$$\begin{cases} 2\alpha_1\gamma_1 & = 0 \\ \alpha_1\gamma_2 + \alpha_2\gamma_1 & = \alpha_1 \\ 2\alpha_2\gamma_2 & = \alpha_2. \end{cases} \tag{2.20}$$

By solving the system of equation (2.20), we get the trivial solution with $f(x) = 0$, and arbitrary function g; and two other solutions as following

$$\begin{cases} f(x) = \alpha m(x) \\ g(x) = \frac{1}{2} m(x) \end{cases}, \begin{cases} f(x) = \alpha a(x) m(x) \\ g(x) = m(x) \end{cases},$$

where m is a nonzero complex exponential, a is a nonzero complex additive function and α is a constant.

In the second case (2b) $2 = 1 + 1$, applying the general non-degenerate solution (2.14) and condition (2.15) of the constants, we have the non-degenerate solution form of the sine equation (2.16)

$$\begin{aligned} f(x) &= \alpha_1 m_1(x) + \alpha_2 m_2(x) \\ g(x) &= \gamma_1 m_1(x) + \gamma_2 m_2(x), \end{aligned} \tag{2.21}$$

where m_1, m_2 are nonzero complex exponentials, and the constants fulfill the condition

$$\begin{bmatrix} \alpha_1 & \gamma_1 \\ \alpha_2 & \gamma_2 \end{bmatrix} \begin{bmatrix} \gamma_1 & \gamma_2 \\ \alpha_1 & \alpha_2 \end{bmatrix} = \begin{bmatrix} \alpha_1 & 0 \\ 0 & \alpha_2 \end{bmatrix}. \tag{2.22}$$

From the condition (2.22), we can get the value of the constants

by solving the system of equations

$$\begin{cases} 2\alpha_1\gamma_1 & = \alpha_1 \\ \alpha_1\gamma_2 + \alpha_2\gamma_1 & = 0 \\ 2\alpha_2\gamma_2 & = \alpha_2. \end{cases} \tag{2.23}$$

By solving the system of equation (2.23), we get the trivial solution with $f(x) = 0$, and arbitrary function g; and two other solutions as following

$$\begin{cases} f(x) = \alpha m(x) \\ g(x) = \frac{1}{2} m(x) \end{cases}, \begin{cases} f(x) = \alpha m_1(x) - \alpha m_2(x) \\ g(x) = \frac{1}{2} m_1(x) + \frac{1}{2} m_2(x) \end{cases},$$

where m, m_1, m_2 are nonzero complex exponentials and α is a constant.

Similarly, we consider the cosine equation (2.17). In the first case (2a) $2 = 2$, applying the general non-degenerate solution (2.12) and condition (2.13) of the constants, we have the non-degenerate solution form of the cosine equation (2.17)

$$\begin{aligned} f(x) &= (\alpha_1 a(x) + \alpha_2) m(x) \\ g(x) &= (\beta_1 a(x) + \beta_2) m(x), \end{aligned} \tag{2.24}$$

where m is a nonzero complex exponential, a is a nonzero complex additive function, and the constants fulfill the condition

$$\begin{bmatrix} \alpha_1 & -\beta_1 \\ \alpha_2 & -\beta_2 \end{bmatrix} \begin{bmatrix} \alpha_1 & \alpha_2 \\ \beta_1 & \beta_2 \end{bmatrix} = \begin{bmatrix} 0 & \alpha_1 \\ \alpha_1 & \alpha_2 \end{bmatrix}. \tag{2.25}$$

From the condition (2.25), we can get the value of the constants by solving the system of equations

$$\begin{cases} \alpha_1^2 - \beta_1^2 & = 0 \\ \alpha_1\alpha_2 - \beta_1\beta_2 & = \alpha_1 \\ \alpha_2^2 - \beta_2^2 & = \alpha_2. \end{cases} \tag{2.26}$$

We get solutions as follows

$$\begin{cases} f(x) = \alpha m(x) \\ g(x) = \beta m(x) \\ \texttt{where}\ \ \beta^2 = \alpha^2 - \alpha \end{cases}, \begin{cases} f(x) = (\alpha a(x) + 1)m(x) \\ g(x) = \pm \alpha a(x) m(x) \end{cases},$$

where m is a nonzero complex exponential, a is a nonzero complex additive function and α, β are constants.

In the second case (2b) $2 = 1 + 1$, applying the general non-degenerate solution (2.14) and condition (2.15) of the constants, we have the non-degenerate solution form of the cosine equation (2.17)

$$\begin{aligned} f(x) &= \alpha_1 m_1(x) + \alpha_2 m_2(x) \\ g(x) &= \beta_1 m_1(x) + \beta_2 m_2(x), \end{aligned} \tag{2.27}$$

where m_1, m_2 are nonzero complex exponentials, and the constants fulfill the condition

$$\begin{bmatrix} \alpha_1 & -\beta_1 \\ \alpha_2 & -\beta_2 \end{bmatrix} \begin{bmatrix} \alpha_1 & \alpha_2 \\ \beta_1 & \beta_2 \end{bmatrix} = \begin{bmatrix} \alpha_1 & 0 \\ 0 & \alpha_2 \end{bmatrix}. \tag{2.28}$$

From the condition (2.28), we can get the value of the constants by solving the system of equations

$$\begin{cases} \alpha_1^2 - \beta_1^2 & = \alpha_1 \\ \alpha_1\alpha_2 - \beta_1\beta_2 & = 0 \\ \alpha_2^2 - \beta_2^2 & = \alpha_2. \end{cases} \tag{2.29}$$

We get solutions as follows

$$\begin{cases} f(x) = \alpha m(x) \\ g(x) = \beta m(x) \\ \texttt{where}\ \ \beta^2 = \alpha^2 - \alpha \end{cases}, \begin{cases} f(x) = \alpha m_1(x) + \beta m_2(x) \\ g(x) = (1-\alpha) m_1(x) - \beta m_2(x) \\ \texttt{where}\ \ \beta^2 = \alpha^2 - \alpha \end{cases},$$

where m, m_1, m_2 are nonzero complex exponentials, and α, β are constants.

2.4 Description of the program function `LCfesolve`

In this section, we describe the computer program `LCfesolve`, which investigates solutions of Levi-Civita type functional equations (2.1).

The program `LCfesolve` is included in the Maple package called `FunctionalEquations:-LeviCivitafesolve`. Accessing `LCfesolve` follows the same conventions as other Maple procedures

```
> with(FunctionalEquations:-LeviCivitafesolve);
> LCfesolve(arguments);
```

To execute the program, users must provide at least one input parameter: the functional equations, denoted as `e`. Additionally, an optional parameter related to the output can be specified.

The functional equation `e` must belong to the Levi-Civita type (2.1), and the variables in the function `e` should be denoted by 'x' and 'y'.

The program offers two output options:

$$\text{OUTPUT} = \{\text{`}\texttt{brief}\text{'}, \text{`}\texttt{full}\text{'}\}$$

The default option is: '`brief`'.

If the input is correctly specified, the program will return a result regarding the solutions of the given functional equation `e`. The following section presents examples demonstrating various output options and their clarification.

Assuming the `FunctionalEquations:-LeviCivitafesolve` package has already been loaded in the current Maple session, providing only parameter `e` will activate the default output setting. In that case, the program will attempt to solve the functional equation `e` and display the general form of the solution, along with conditions applied on its coefficients. For instance:

Let us consider the sine equation example, and comparing the solution oh the program with the hand-computing one that mentioned in the previous section, i.e, equation (2.16).

```
> LCfesolve(f(x + y) = f(x)g(y) + g(x)f(y));
```

yields

Case: [1, 1]

$f = \alpha_1 m_1(x) + \alpha_2 m_2(x)$

$g = \gamma_1 m_1(x) + \gamma_2 m_2(x)$

Moreover, the coefficients fulfill the equation Q.R = P, with the matrices Q, R, P as follows, respectively

$$Q = \begin{bmatrix} \alpha_1 & \gamma_1 \\ \alpha_2 & \gamma_2 \end{bmatrix}$$

$$R = \begin{bmatrix} \gamma_1 & \gamma_2 \\ \alpha_1 & \alpha_2 \end{bmatrix}$$

$$P = \begin{bmatrix} \alpha_1 & 0 \\ 0 & \alpha_2 \end{bmatrix}$$

The coefficients fulfill the system of equations:

$$\{2\alpha_1\gamma_1 - \alpha_1, 2\alpha_2\gamma_2 - \alpha_2, \alpha_1\gamma_2 + \alpha_2\gamma_1\}.$$

Case: [2]

$f = (\alpha_1 A_1(x) + \alpha_2) m_1(x)$

$g = (\gamma_1 A_1(x) + \gamma_2) m_1(x)$

Moreover, the coefficients fulfill the equation Q.R = P, with the matrices Q, R, P as follows, respectively

Q =

$$\begin{bmatrix} \alpha_1 & \gamma_1 \\ \alpha_2 & \gamma_2 \end{bmatrix}$$

R =

$$\begin{bmatrix} \gamma_1 & \gamma_2 \\ \alpha_1 & \alpha_2 \end{bmatrix}$$

P =

$$\begin{bmatrix} 0 & \alpha_1 \\ \alpha_1 & \alpha_2 \end{bmatrix}$$

The coefficients fulfill the system of equations:

$$\{2\alpha_1\gamma_1, 2\alpha_2\gamma_2 - \alpha_2, \alpha_1\gamma_2 + \alpha_2\gamma_1 - \alpha_1\}.$$

In the case when the option '`full`' is given by the user, the program will show also the solution of the corresponding system of equations that the coefficients have to fulfill. Let us examine the cosine equation with '`full`' output option and then compare the final solutions with the computations presented in the last section, i.e, equation (2.17).

```
> fe := f(x + y) = f(x)f(y) - g(x)g(y):
> LCfesolve(fe, full);
```

yields

Case: [1, 1]

$$f = \alpha_1 m_1(x) + \alpha_2 m_2(x)$$

$$g = \beta_1 m_1(x) + \beta_2 m_2(x)$$

Moreover, the coefficients fulfill the equation Q.R = P, with the matrices Q, R, P as follows, respectively

Q =

$$\begin{bmatrix} \alpha_1 & -\beta_1 \\ \alpha_2 & -\beta_2 \end{bmatrix}$$

R =

$$\begin{bmatrix} \alpha_1 & \alpha_2 \\ \beta_1 & \beta_2 \end{bmatrix}$$

P =

$$\begin{bmatrix} \alpha_1 & 0 \\ 0 & \alpha_2 \end{bmatrix}$$

The coefficients fulfill the system of equations:

$$\{\alpha_1\alpha_2 - \beta_1\beta_2, \alpha_1^2 - \beta_1^2 - \alpha_1, \alpha_2^2 - \beta_2^2 - \alpha_2\}.$$

The solutions of the above system of equations:

$$\{\alpha_1 = 0, \alpha_2 = \alpha_2, \beta_1 = 0, \beta_2 = RootOf(\_Z^2 - \alpha_2^2 + \alpha_2)\},$$

$$\{\alpha_1 = -\alpha_2 + 1, \alpha_2 = \alpha_2, \beta_1 = -RootOf(\_Z^2 - \alpha_2^2 + \alpha_2), \beta_2 = RootOf(\_Z^2 - \alpha_2^2 + \alpha_2)\},$$

$$\{\alpha_1 = \alpha_1, \alpha_2 = 0, \beta_1 = RootOf(\_Z^2 - \alpha_1^2 + \alpha_1), \beta_2 = 0\}.$$

Case: [2]

$f = (\alpha_1 A_1(x) + \alpha_2)m_1(x)$

$g = (\beta_1 A_1(x) + \beta_2)m_1(x)$

Moreover, the coefficients fulfill the equation Q.R = P, with the matrices Q, R, P as follows, respectively

Q =

$$\begin{bmatrix} \alpha_1 & -\beta_1 \\ \alpha_2 & -\beta_2 \end{bmatrix}$$

R =

$$\mathrm{P} = \begin{bmatrix} \alpha_1 & \alpha_2 \\ \beta_1 & \beta_2 \end{bmatrix}$$

$$\begin{bmatrix} 0 & \alpha_1 \\ \alpha_1 & \alpha_2 \end{bmatrix}$$

The coefficients fulfill the system of equations:

$$\{\alpha_1^2 - \beta_1^2, \alpha_2^2 - \beta_2^2 - \alpha_2, \alpha_1\alpha_2 - \beta_1\beta_2 - \alpha_1\}.$$

The solutions of the above system of equations:

$$\{\alpha_1 = 0, \alpha_2 = \alpha_2, \beta_1 = 0, \beta_2 = RootOf(\_Z^2 - \alpha_2^2 + \alpha_2)\},$$

$$\{\alpha_1 = \alpha_1, \alpha_2 = 1, \beta_1 = -\alpha_1, \beta_2 = 0\},$$

$$\{\alpha_1 = \alpha_1, \alpha_2 = 1, \beta_1 = \alpha_1, \beta_2 = 0\}.$$

2.5 Explanation of the program function `LCfesolve`

This section presents codes of some main functions and their detailed explanations to clarify how our program `LCfesolve` works.

```
1 LCfesolveGeneral := proc(n)
2    local n_decomp, i, i_case, rs;
3    n_decomp := partition(n);
4    rs := [];
5    for i to nops(n_decomp) do
6       i_case := Coefficients_1case(n,
           n_decomp[i]);
7       rs := [op(rs), [n_decomp[i], i_case]];
```

```
8     end do;
9     return rs;
10 end proc;.
```

In the general case, with the given terms `n`, procedure LCfesolveGeneral above will process step by step, as follows:

- Line 3: decomposing `n` into sum of positive integers,
- Lines 5 – 8: we will obtain one solution for each decomposition of `n` by calling the procedure Coefficients_1case,
- Line 9: this function will return all solutions in the form of a Maple list.

Now, we can look at the procedure Coefficients_1case and its steps.

```
11 Coefficients_1case := proc(n, n_list)
12     local N_Q, N_Rt, M;
13     pre_char_n := 0;
14     N_Q := Compute_N(n, n_list, beta);
15     N_Rt :=
           LinearAlgebra[Transpose](Compute_N(n,
           n_list, gamma));
16     M := Compute_M(N_Q, N_Rt, n_list);
17     return [N_Q, N_Rt, M];
18 end proc;.
```

As mentioned in Section 2.2, for each decomposition of `n`, the function will construct three block matrices `N_Q, N_Rt, M`, which satisfy the necessary and sufficient condition (2.7). According to the equations (2.5), (2.6), (2.3), (2.4), the following functions are developed to build up those two block matrices `M, N`.

```
19 Compute_M := proc(N_Q, N_Rt, n_list)
20     local k, M_list, j, n_j, M_j, M, pre_num,
           pre_char;
```

```
21     k := nops(n_list); M_list := [];
22     pre_num := 0;
23     pre_char := 0;
24     for j to k do
25         n_j := n_list[j];
26         M_j := Compute_M_j(N_Q, N_Rt, j, n_j,
               pre_num, pre_char);
27         M_list := [op(M_list), M_j];
28         if 3 <= n_j then
29             pre_char := pre_char + n_j - 1;
30             pre_num := pre_num + 1;
31         else pre_num := pre_num + n_j;
32         end if;
33     end do;
34     M := DiagonalMatrix(M_list);
35     return M;
36 end proc;.
```

Following the above function Compute_M, and the equation (2.5), the block matrix `M` is constructed of `k` matrices on the diagonal (line 34). While each `M_j` for $j = 0, \ldots, k$ are computed in the function Compute_M_j (lines 24 – 32).

Moreover, based on the equation (2.3), the function Compute_M_j below is used for computing each matrix `M_j` (lines 64 – 70).

```
37 Compute_M_j := proc(N_Q, N_Rt, j, n_j,
       pre_num, pre_char)
38     local M, p, q, ind, L, pre_size, index_q,
           index_r;
39     pre_size := pre_num + pre_char;
40     M := Matrix(n_j);
41     if 3 <= n_j then
42         for p from 0 to n_j - 1 do
43             for q from 0 to n_j - 1 do
44                 if p + q < n_j then
```

```
45                  ind := p + q;
46                  if ind <> 0 then
47                     index_q := N_Q[n_j - p +
                          pre_size, 1][3];
48                     index_r := N_Rt[1, n_j - q
                          + pre_size][3];
49                     if type(index_q, list) and
                          type(index_r, list) then
50                        L := Flatten([index_q,
                             index_r]);
51                     elif type(index_q, list)
                          then
52                        L := index_q;
53                     else
54                        L := index_r;
55                     end if;
56                     M[n_j - p, n_j - q] := [(p
                          + q)!/(p!*q!), alpha,
                          L];
57                  else
58                     M[n_j - p, n_j - q] := [(p
                          + q)!/(p!*q!), alpha,
                          pre_num + 1];
59                  end if;
60               end if;
61            end do;
62         end do;
63      else
64         for p from 0 to n_j - 1 do
65            for q from 0 to n_j - 1 do
66               if p + q < n_j then
67                  M[n_j - p, n_j - q] := [(p +
                       q)!/(p!*q!), alpha,
                       pre_num + n_j - p - q];
```

```
68                end if;
69             end do;
70          end do;
71       end if;
72       return M;
73 end proc;.
```

It is easy to see that in both functions Compute_M and Compute_M_j, in case $n_j \geq 3$, more complex codes are needed to handle the situations (lines 28 – 32 and lines 41 – 62). In the cases of small value of n_j, such as $n_j = 1, n_j = 2$, it is easy to assign variable names for the matrices, for example: α_1, α_2. However, when the value of n_j grows, there will be many variable names, that need to be assigned automatically. Those variable names have to be logical, understandable, and does not conflict between matrices. The follow-up example shows how the program reacts to a big value n_j.

In the following, an example of one block matrix `M` in case $n = 4 = 1 + 3$ is presented to emphasize that situation.

Let a Levi-Civita functional equation be:

$$f(x+y) = g_1(x)h_1(y) + g_2(x)h_2(y) + g_3(x)h_3(y) + g_4(x)h_4(y).$$

Here $n = 4$, consider the case when $n = 4 = 1 + 3$. We have the block matrix `M`

$$\begin{bmatrix} \alpha_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \alpha_{[a,b]} \\ 0 & 0 & 2\alpha_{[a,c]} & \alpha_{[a]} \\ 0 & \alpha_{[a,c]} & \alpha_{[a]} & \alpha_2 \end{bmatrix},$$

for $a, b, c, d = 1, 2$.

The characters `a, b, c` were auto-generated for visualizing needed variables. As a result, the bigger value of `n` causes more complex processes in our program.

Chapter 3

Types of functional equations

In this chapter, a computer program, developed in the computer algebra system Maple is presented, which determines the type of functional equations. The results shown in this chapter are based on the paper [42].

3.1 Introduction

The continuous development and improvement in computer technology and computational science provided solid support for mathematical models, making them more efficient in terms of computing time and resources. Computers help solve problems quickly and accurately, offering scientists effective ways of treating unsolved problems or topics that are yet to be studied in depth. Due to some basic features of the theory of functional equations, the application of computer-assisted methods to problems related to this field started relatively late, at the end of the 20th century.

In the 1980s, János Aczél posed the question if it was possible to develop a computer program to determine the solutions of two variables linear functional equations using László Székelyhidi's method published in his paper [69]. The first program to addressing such equations was created 30 years ago and

described in the thesis [43] (cf. also [44]; and, for more general versions, [15], [16] and [17]). Over the last few decades, many researchers have used computers to study functional equations (cf., e.g., [7], [8], [9], [21], [40], [41], [47], [57], [61], [62], [63] and [64]).

Today, symbolic calculations combined with general computer programming are strong methods for exploring functional equations. We are currently developing a software package that includes programs related to functional equations. It comes as a natural idea to create a program capable of determining the types or classes of functional equations to which they belong. In this chapter, we present this program and its applications.

In the first part of the chapter, Section 3.2, we list the types and classes of functional equations considered in our program. In the main section, Section 3.3, we present our program with a complete description as well as examples of the running program. The last section, Section 3.4, shows some short codes and detailed explanations to illustrate how our program works.

3.2 Types of functional equations considered

In this section, we shortly describe the functional equations recognized by our computer program.

The *Cauchy functional equation*

$$f(x+y) = f(x) + f(y) \tag{3.1}$$

is fundamental not only to the theory of functional equations, but also in all of mathematics. It is named after Augustin-Louis Cauchy, who was the first to determine its general continuous solutions [18]. It had been studied earlier by several authors, among others, by Adrien-Marie Legendre [55] and Carl Friedrich Gauss [31]. It has been a central topic of several books on functional equations, including [50] and [51] by Marek Kuczma. It is frequently referred to as the equation of

homomorphism or the *Cauchy additive functional equation*, to distinguish it from other 'Cauchy type' functional equations:

– the Cauchy exponential functional equation

$$f(x+y) = f(x)f(y) \tag{3.2}$$

– the Cauchy logarithmic functional equation

$$f(xy) = f(x) + f(y) \tag{3.3}$$

– the Cauchy multiplicative functional equation

$$f(xy) = f(x)f(y) \tag{3.4}$$

(cf., [18], and [2] and [3] among others).

The *Pexider functional equations* are generalizations of the Cauchy equations mentioned above, when different unknown functions are allowed on each side of the equation ([65]):

$$f(x+y) = g(x) + h(y), \tag{3.5}$$
$$f(x+y) = g(x)h(y), \tag{3.6}$$
$$f(xy) = g(x) + h(y), \tag{3.7}$$
$$f(xy) = g(x)h(y). \tag{3.8}$$

The following functional equation

$$f\left(\frac{x+y}{2}\right) = \frac{f(x)+f(y)}{2} \tag{3.9}$$

is well-known as the *Jensen functional equation*. This equation characterizes functions that preserve the midpoint operation, which is strongly related to the concept of convexity.

The functional equation

$$f(x+y-xy) + f(xy) = f(x) + f(y) \tag{3.10}$$

is referred to as the *Hosszú functional equation*. It was introduced by Miklós Hosszú at the International Symposium on Functional Equations held in Zakopane, Poland, in 1967.

At the Symposium on Functional and Differential Equations held in Zawoja, Poland, in 1971, Jan Mikusiński considered the functional equation

$$f(x+y)[f(x+y)-f(y)-f(x)]=0. \tag{3.11}$$

Later, this equation was named after him.

If the function $f:\mathbb{R}\to\mathbb{R}$ is additive, then it also satisfies the following equation

$$[f(x+y)]^2=[f(x)+f(y)]^2. \tag{3.12}$$

This immediately yields either $f(x+y)=f(x)+f(y)$, or $f(x+y)=-[f(x)+f(y)]$. Due to its form, it is referred to as an *alternative functional equation.*

The functional equation

$$f(x+y)+f(x-y)=2f(x)+2f(y) \tag{3.13}$$

is known as the *quadratic functional equation.*

The *d'Alembert functional equation* (cf., e.g., [25], [26], furthermore, [2] and [3]) named after the French mathematician Jean le Rond d'Alembert and inspired by the well-known trigonometric identity $\cos(x+y)+\cos(x-y)=2\cos(x)\cos(y)$ is of the form

$$f(x+y)+f(x-y)=2f(x)f(y). \tag{3.14}$$

The *inhomogeneous linear functional equation of order* 1 has the form

$$f(ax+by+c)=Af(x)+Bf(y)+C, \tag{3.15}$$

where a,b,A,B are real numbers such that $abAB\neq 0$. (It was considered, e.g., in [1], [22], [23], [50] and [56].)

The trigonometric functions $f(x)=\cos x$ and $g(x)=\sin x$ satisfy the following functional equations

$$f(x+y)=f(x)f(y)-g(x)g(y), \tag{3.16}$$
$$f(x-y)=f(x)f(y)+g(x)g(y), \tag{3.17}$$
$$g(x+y)=g(x)f(y)+f(x)g(y), \tag{3.18}$$
$$g(x-y)=g(x)f(y)-g(y)f(x). \tag{3.19}$$

Hence equations (3.16) – (3.19) are called *trigonometric functional equations* (cf., e.g., [78]).

One of the most important special families of functional equations is the class of *(two variable) linear functional equations*

$$\sum_{i=0}^{n+1} f_i(p_i x + q_i y) = 0, \tag{3.20}$$

where n is a positive integer, $p_0, \dots, p_{n+1}$ and $q_0, \dots, q_{n+1}$ are real numbers, and $f_0, \dots, f_{n+1}$ are unknown functions. It has been considered in several papers and books (cf., e.g., [69], [72] and [77]). Several important functional equations belong to this class, among others, the Cauchy equation (3.1), the Pexider equation (3.5), the Jensen equation (3.9) and the quadratic equation (3.13) as a special case.

Another important class of functional equations is that of *Levi-Civita functional equations*

$$f(x+y) = \sum_{i=1}^{n} g_i(x) h_i(y), \tag{3.21}$$

where n is a positive integer, f, g_i, h_i, $(i = 1, \dots, n)$ are unknown functions (cf., [70], [72]). This class also contains several well-known functional equations mentioned before. (For example, the Cauchy functional equations (3.1), (3.2); the Pexider functional equations (3.5), (3.6); trigonometric functional equations (3.16), (3.18).)

3.3 Description of the program

In this section, we describe the computer program, that determines the type of functional equations. More precisely, we will give detailed descriptions of three program functions `fewhattype`, `fetype`, `feinfo`.

Three program functions, `fewhattype`, `fetype`, and `feinfo` are contained in the Maple package

`FunctionalEquations:-FETypes`. Similarly to other Maple programs, `fewhattype`, `fetype`, and `feinfo` can be accessed with the forms

```
> with(FunctionalEquations:-FETypes);
> fewhattype(arguments);
> fetype(arguments);
> feinfo();.
```

A detailed description of each function is given in the following part.

3.3.1 The function `fewhattype`

To run this function, at least one input parameter has to be given: the functional equations called `fe`. In addition, extra parameters `funcs` and `vars` can be used to highlight the names of the unknown functions and the variables of the functions contained. Thus, the input of the program should be of the form

```
> fewhattype(fe, [funcs, vars]);.
```

The list of the unknown functions `funcs` and the list of the functions' variables `vars` have to be given as a Maple list structure. The detailed description of Maple list structure was mentioned in Chapter 1, Section 1.4. By default,

```
funcs := [f,g,h,k,l];
vars := [x,y,z,u,v];.
```

Hence, when users provide only the argument `fe`, the set of the unknown functions of the given functional equation has to be a subset of $\{f, g, h, k, l\}$. Similarly, the functions' variables must be in the set $\{x, y, z, u, v\}$. If not, users need to include additional parameters for the program to run properly.

Some examples of the correct form of input arguments are:

$$f(x+y) - f(x) - f(y) = 0, funcs = [f], vars = [x, y]$$

or simply

$$f(x+y) - f(x) - f(y) = 0$$

or

$$m(a+b) - n(a) - n(b) = 0, funcs = [m, n], vars = [a, b].$$

If the input is given correctly, the program will list all types and classes to which the given functional equation `fe` belongs. In the following, we present some examples with different options and their meanings.

Assume that the package `FunctionalEquations` has already been called in the current Maple session.

```
> fewhattype(f(x + y) - f(x) - f(y));
```

yields

```
Cauchy equation
Inhomogeneous linear functional equation
     of order 1
Linear type equation
Levi-Civita type equation
```

The program is able to simplify the given functional equation before starting the main process to determine its types or classes. For example:

```
> fe:= sqrt(3).f(x + y) + sqrt(3).f(x - y) = sqrt(12).f(x)
        + sqrt(12).f(y);
> fewhattype(fe);
```

gives

```
Linear type equation
Quadratic equation.
```

When the unknown functions and their variables are not in the default case, users can add extra parameters to use the program, as follows:

```
> fe := s(a + b) = s(a).c(b) + c(a).s(b);
> fewhattype(fe, funcs = [s, c], vars = [a, b]);
```

yields

```
Levi-Civita type equation
Functional equation for trigonometric
   functions, addition formula for sine.
```

3.3.2 The function `fetype`

If users input a functional equation `fe` and specify a type or class of functional equations, this function will simply return the value `True` or `False` to indicate whether `fe` is the given type. Similarly to the previous program, extra parameters `funcs` and `vars` can be added as optional arguments to specify the list of unknown functions and their variables. Therefore, the input of the program should be

```
> fetype(fe, type, [funcs, vars]);.
```

Notes on the three parameters `fe`, `funcs`, and `vars` can be found in the previous subsection. The parameter `type` should be a string like those given in the list of considered types and classes of functional equations mentioned in the second section.

For example,

```
> fetype(f(xy) - f(x) - f(y) = 0, Cauchy);
```

shows

```
                        true.
> fetype(f(x + y) - g(x) - h(y) = 0, Cauchy);
```

gives

```
                        false.
```

The program is able to simplify the considered functional equation before evaluating it, as follows:

```
> fetype(2.f((x + y)/2) = f(x) + f(y), Jensen);
```

yields

```
                        true.
```

The extra parameter `funcs` is used in the example below to show how our program work in case the unknown function is not contained in the default set:

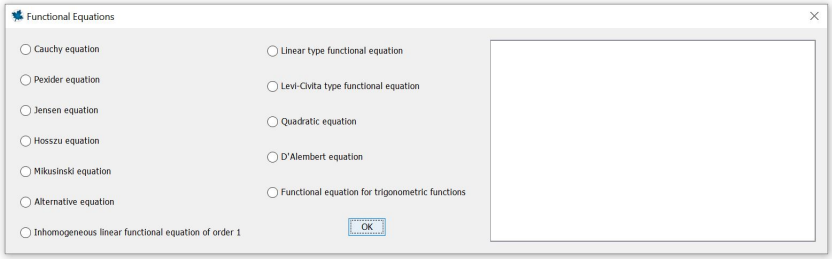
```
> fe := m(x + y − xy) + m(xy) = m(x) + m(y);
> fetype(fe, Hosszu, funcs = [m]);
```

gives

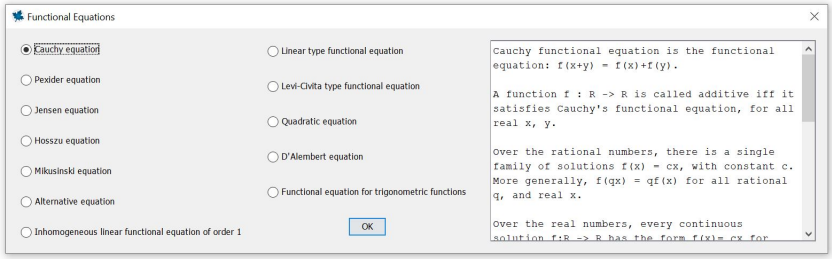
```
true.
```

3.3.3 The function `feinfo`

The function *feinfo()* is used to get information about all the considered types and classes of functional equations. When *feinfo()* is called, the following window will pop up



When one type or class of the functional equation is chosen, the corresponding information will be shown in the right-hand side textbox. For example, when the *Cauchy equation* is chosen, you will see



.

3.4 Explanation of the program

In this section, we show some procedure codes with detailed explanations to illustrate how our program works.

3.4.1 The functions `fewhattype` and `fetype`

First, it can be observed that many types of functional equations share common terms. For example, the term $f(x + y)$

appears in the Cauchy functional equations (3.1), (3.2), the Pexider functional equations (3.5), (3.6), the Jensen functional equation (3.9), the quadratic functional equation (3.13), the class of Levi-Civita functional equations (3.21), etc. Similarly, the term $f(xy)$ is present in the Cauchy functional equations (3.3), (3.4), the Pexider functional equations (3.7), (3.8), and the Hosszú functional equation (3.10).

For this reason, our program checks the presence of common terms in the given functional equation and then determines its types or classes.

The following procedure checks the term $cf(x+y)$ where c is a constant in the given functional equation. Therefore, this procedure will be called later to check types and classes that contain the term $f(x+y)$, as mentioned above.

```
checkcfxplusy := proc(fe, var)
    local Svar, flag, nfe, i, t, Lfe, Rfe,
       ufunc;
    Svar := convert(var, `+`);
    flag := false;
    nfe := fe;
    for i to nops(nfe) do
       t := op(i, nfe);
       if type(t, function) and op(t) = Svar
          then
          ufunc := op(0, t);
          if not ufunc in FUNCS then
             error "Invalid unknown
                functions";
          end if;
          Lfe := t;
          Rfe := Lfe - nfe;
          flag := true;
          break;
       elif type(t, `*`) then
```

```
18            if type(op(-1, t), function) and
                 op(op(-1, t)) = Svar then
19               ufunc := op(0, op(-1, t));
20               if not ufunc in FUNCS then
21                  error "Invalid unknown
                       functions";
22               end if;
23               Lfe := simplify(t/op(1 .. -2,
                    t));
24               Rfe := simplify(Lfe - nfe/op(1
                    .. -2, t));
25               flag := true;
26               break;
27            end if;
28         end if;
29      end do;
30      if flag = true then
31         return 1, Lfe, Rfe, ufunc;
32      else return 0;
33      end if;
34   end proc;
```

In this procedure, line 3 returns the sum expression of the unknown functions' variables, for example, when `var = [x, y]` then `Svar = x + y`. The program will run through all the terms in the given functional equation (line 6), then from line 7 to line 16, checks if the term $f(x+y)$ is there (it can be $g(u+v)$ or $m(x+y)$ depending on the unknown function `ufunc` and the list of variables `var`). Additionally, if the term appears as $cf(x+y)$, where c is a constant, lines 17 – 26 will not only recognize the term $cf(x+y)$, but the program will also cancel the constant c out of the given functional equation (lines 23 – 24). After checking all the terms contained, we have the procedure

– `return 0`: if the term does not appear;

– `return 1, Lfe, Rfe, ufunc`: term appears, where
`Lfe`: the appeared term,
`Rfe`: the rest terms of given functional equation
`ufunc`: unknown function in Lfe.

Similarly, the procedure checkcfxtimesyorxy checks for the term $f(xy)$ and also the term $cf(xy)$, where c is a constant, in general. In mathematics, the expressions `xy` and `x*y` are both understood as the multiplication of two variables x and y. Therefore, two variables, `Mvar1` and `Mvar2`, are used to ensure the program considers all the given cases. For example, if `var = [x, y]` then we have `Mvar1 = x*y` and `Mvar2 = xy`.

With the same structure and technique as in the previous procedure, the program will check each term of the given functional equation (line 41) in both cases, with and without the constant coefficient c (lines 43 – 51 and lines 52 – 63, respectively). The procedure checkcfxtimesyorxy returns the same output as the procedure checkcfxplusy.

```
35 checkcfxtimesyorxy := proc(fe, var)
36    local Mvar1, Mvar2, flag, nfe, i, t, Lfe,
         Rfe, ufunc;
37    Mvar1 := convert(var, `*`);
38    Mvar2 := cat(seq(var[i], i = 1 .. 2));
39    flag := false;
40    nfe := fe;
41    for i to nops(nfe) do
42       t := op(i, nfe);
43       if type(t, function) and (op(t) =
            Mvar1 or op(t) = Mvar2) then
44          ufunc := op(0, t);
45          if not ufunc in FUNCS then
46             error "Invalid unknown
                  functions";
47          end if;
48          Lfe := t;
```

```
49            Rfe := Lfe - nfe;
50            flag := true;
51            break;
52         elif type(t, '*') then
53            if type(op(-1, t), function) and
                 (op(op(-1, t)) = Mvar1 or
                 op(op(-1, t)) = Mvar2) then
54               ufunc := op(0, op(-1, t));
55               if not ufunc in FUNCS then
56                  error "Invalid unknown
                       functions";
57               end if;
58               Lfe := simplify(t/op(1 .. -2,
                    t));
59               Rfe := simplify(Lfe - nfe/op(1
                    .. -2, t));
60               flag := true;
61               break;
62            end if;
63         end if;
64      end do;
65      if flag = true then
66         return 1, Lfe, Rfe, ufunc;
67      else return 0;
68      end if;
69   end proc;.
```

Let us consider all types of Cauchy equations (3.1)–(3.4) and Pexider equations (3.5)–(3.8). Their left-hand sides are considered by the two procedures above, and our program will check the rest of the terms, i.e., the right-hand side of the equation, to give the conclusion. The procedure for this process is as follows

```
70 checkrhsCauchyPexider := proc(rhseq, var)
71    local t1, t2, ufunc;
```

```
72    if nops(rhseq) <> 2 then return 0; end if;
73    if not (type(rhseq, `+`) or type(rhseq,
         `*`)) then
74       return 0;
75    end if;
76    t1 := op(1, rhseq);
77    t2 := op(2, rhseq);
78    if not (type(t1, function) and type(t2,
         function)) then
79       return 0;
80    end if;
81    if evalb([op(t1), op(t2)] = var) or
         evalb([op(t2), op(t1)] = var) then
82       if not (op(0, t1) in FUNCS and op(0,
            t2) in FUNCS) then
83          error "Invalid unknown functions";
84       end if;
85       ufunc := MakeUnique([op(0, t1), op(0,
            t2)]);
86       return 1, whattype(rhseq), ufunc;
87    else return 0;
88    end if;
89 end proc;.
```

Some conditions are checked for the right-hand side of the equation.

- Exactly two terms are contained – line 72 and the operator between them is addition + or multiplication ∗ (lines 73 –74);

- Both terms are in the form of a function (lines 78– 80) and the unknown function's variables are correct (lines 81 – 87).

The procedure:

- `return 0`: when the given functional equation does not belong to any of the Cauchy equations (3.1)–(3.4) or Pexider equations (3.5)–(3.8);
- `return 1, operator, ufunc`: where
  - `operator`: the operator between the terms can be addition $+$ or multiplication $*$,
  - `ufunc`: list of unknown functions, which is important for distinguishing between Cauchy types and Pexider types.

3.4.2 The function `feinfo`

With the procedure feinfo, we want to introduce a very surprising element of the Maple programming language – a graphic user interface (GUI). An interactive window can be created with the function `Maplet` contained in the Maple's package `Maplets:-Elements`.

```
1 feinfo := proc()
2     local maplet;
3     maplet := Maplet(Window('layout' = 'BL1',
          'title' = "Functional Equations"),
4         BoxLayout['BL1'](
5             BoxColumn('halign'='left',
6                 BoxRow(RadioButton['RB1']("Cauchy
                      equation", 'group' = 'FE',
                      Evaluate(#Cauchy equation's
                      information))),
7                 # other RadioButtons for
                      different types and classes
                      of functional equations
8                 Button("OK", Shutdown()))),
9             BoxColumn(BoxRow(TextBox['TB1']("",
                  'editable' = 'false','width'=
                  50)))), ButtonGroup['FE']()));
```

```
10     Maplets[Display](maplet):
11 end proc;.
```

In this procedure, a window simply pops up (line 10) which contains:

- Lines 6 – 7: a list of radio buttons that allows users to choose their interested functional equation type or class,
- Line 9: a text box showing information on the chosen functional equation type or class,
- Line 8: a button `OK` for shutting down the application (i.e., closing the popped-up window).

Chapter 4

m-convex hulls of sets of points

In this chapter, a computer program, developed in the computer algebra system Maple, is presented, which visualizes the animation of the m-convex hulls of sets of points on the Cartesian coordinate systems for a two-dimensional plane and a three-dimensional space. The results discussed in this chapter are based on the paper [39].

4.1 Introduction

The concept of m-convexity in the sense of G. Toader is a mathematical phenomenon that was introduced about thirty-five years ago, and investigated by many authors (cf., eg., [73], [54], [53], [52], [11]). However, the geometrical properties of m-convex sets have not been researched deeply. With the assistance of computers in general or computer programming in particular, we can get a geometric visualization of m-convexity for further investigation on the concept of convexity as well as its applications in functional equations and inequalities problems (cf., eg., [37], [38]).

In this chapter, we present computer programs developed in the computer algebra system Maple, which visualizes the animation of the m-convex hull of sets of points on the Cartesian

coordinate systems , and determines the relative position of a given point and the m-convex hull of a set of points in the two- and the three-dimensional cases.

In Section 4.2, we provide an overview of some basic information about convexity and m-convexity concepts, which are important for our investigations. In Section 4.3 and 4.4 we describe the animation of the m-convex hull of finite sets of points in both two and three dimensions. Later, in Section 4.5, the computer program, which visualizes the animation of the m-convex hull of sets of points, is presented.

Next, in Section 4.6, we summarize some information about the Barycentric coordinate system, which later serves as the main theoretical background of our programs. And in Section 4.7, we show the programs which determine whether a given point belongs to the m-convex hull of the set of points or not, in both two and three-dimensional cases.

In the last section, Section 4.8, we provide short codes along with detailed explanations to demonstrate how our program functions.

4.2 The concept of m-convexity

In this section, some basic definitions and main theorems about the m-convexity concept are presented.

It is well known that the set $C \subseteq \mathbb{R}^n$ is called convex if

$$tx + (1-t)y \in C, \quad \text{for all } x, y \in C \text{ and } t \in [0,1]$$

The convex hull of a non-empty set $A \subseteq \mathbb{R}^n$ is defined as the intersection of all convex subsets of $\mathbb{R}^n$ containing A.

From the geometric point of view, the set C is convex if for any two points $x, y \in C$, the segment xy is also contained in C. It is easy to see that the convex hull of two points $x, y \in \mathbb{R}^2$ ($x \neq y$) and the origin $O(0,0) \in \mathbb{R}^2$ is the triangle $T = \triangle(x, y, O)$ (cf., eg., [51]). Moreover, in the three-dimensional space, the convex hull of three points $x, y, z \in \mathbb{R}^3$ ($x \neq y \neq z$) and the

origin $O(0,0,0) \in \mathbb{R}^3$ in case the four points are non-coplanar is the tetrahedron (so-called the triangular pyramid) with four vertex corners x, y, z, O, respectively.

According to G. Toader [73], let $m \in [0,1]$ be a fixed real number, a set $C \subseteq \mathbb{R}^n$ is called m-convex if for each $t \in [0,1]$

$$tx + m(1-t)y \in C, \quad \text{for all } x, y \in C$$

The m-convex hull of a non-empty set $A \subseteq \mathbb{R}^n$ is defined as the intersection of all m-convex subsets of $\mathbb{R}^n$ containing A.

From a geometric point of view, the set C is m-convex, if for any two points $x, y \in C$, the segment $[x, my]$ connecting x and my is also included in C.

4.3 The animation of the m-convex hull of sets on the Cartesian two-dimensional plane

It has been shown in [52] that the m-convex hull generated by two different points $x, y \in \mathbb{R}^2$ can be obtained by dropping the triangle $T_m = \triangle(x, y, z)$ from triangle $T = \triangle(O, x, y)$, where O is the origin and z is the point defined by

$$z = \frac{m}{m+1}(x+y).$$

Thus,

$$T - T_m = \triangle(O, x, y) \setminus \triangle(x, y, z) = \triangle(O, x, z) \cup \triangle(O, z, y).$$

This is the basic idea for generating an m-convex hull of a finite set of points on the Cartesian plane. In the following part, we will present the algorithm for generating the m-convex hull of a finite set of points P in $\mathbb{R}^2$, which fulfills the following conditions:

– The set contains non-zero points.

– All of the points are different from each other.

– All points belong to the first quadrant on the plane Oxy.

– None of the points is on the vertical axis Oy.

Algorithm–Generating an animation of the m-convex hull of a finite set of points in $\mathbb{R}^2$

– Input: Finite set of points P, $m \in [0, 1)$.

– Output: The m-convex hull of P.

– Method: The m-convex hull C_P of P can be generated by the following, for all A, B in P, $(A \neq B)$,

$$C_P = C_P \cup \triangle(OAz) \cup \triangle(OzB),$$

$$\text{where } z = \frac{m}{m+1}(A + B), \text{ and } O(0, 0).$$

4.4 The animation of the m-convex hull of sets on the Cartesian three-dimensional space

Similarly, the m-convex hull generated by three different points $x, y, z \in \mathbb{R}^3$ can be obtained by dropping the polyhedron, which is composed of six vertex corners: x, y, z, t, u, v from the tetrahedron generated by four points O, x, y, z, where O is the origin and u, v, t are points defined by

$$t = \frac{m}{m+1}(x + y), u = \frac{m}{m+1}(x + z), v = \frac{m}{m+1}(y + z).$$

Accordingly to the two-dimensional plane result, it can be considered as the union of four tetrahedron which are generated by $[O, x, t, u]$, $[O, y, t, v]$, $[O, z, u, v]$, $[O, t, u, v]$, respectively.

The finite set of points P in $\mathbb{R}^3$ also needs to fulfill the following conditions:

– The set contains non-zero points.

– All of the points are pairwise different.

– All points belong to the first octant of the $Oxyz$.

Therefore, the algorithm is as follows.

Algorithm–Generating an animation of the m-convex hull of a finite set of points in $\mathbb{R}^3$

– Input: Finite set of points P, $m \in [0, 1)$.

– Output: The m-convex hull of P.

– Method: The m-convex hull C_P of P can be generated by the following, for all A, B, C in P, $(A \neq B \neq C)$,

$$\begin{aligned} C_P = C_P &\cup tetrahedron[O, A, T, U] \\ &\cup tetrahedron[O, B, T, V] \\ &\cup tetrahedron[O, C, U, V] \\ &\cup tetrahedron[O, T, U, V], \end{aligned}$$

where

$$T = \frac{m}{m+1}(A+B), \quad U = \frac{m}{m+1}(A+C),$$
$$V = \frac{m}{m+1}(B+C), \quad O(0,0,0).$$

4.5 Description of the program functions `mconvex2d`, `mconvex3d` and the package `MConvexHull`

In this section, the computer program package `MConvexHull`, which contains program functions related to the m-convex hull of a set of points, is presented.

Our programs `mcvhull2d` and `mcvhull3d` are contained in the Maple package `MConvexHull`. Similar to other Maple programs, our programs can be accessed through the forms

```
> with(MConvexHull);
> mcvhull2d(arguments);
> mcvhull3d(arguments);
```

To run the programs in both cases `mcvhull2d` and `mcvhull3d`, users have to give at least one input parameter: the list of points `P`.

The list of points `P`, and all of the input points have to be given as a Maple list structure. The detailed description of Maple list structure was mentioned in Chapter 1, Section 1.4. Moreover, the given list of points has to fulfill the following conditions:

- Points in the list `P` are pairwise different.
- The list of points `P` contains at least two different points in the case of `mcvhull2d` and three distinct points in the case of `mcvhull3d`.
- The list of points `P` does not include the origin, i.e, $O(0,0)$ in two-dimensional case, and $O(0,0,0)$ in three-dimensional case.
- All points contain two coordinate values in the case of using `mcvhull2d` and three coordinate values in the case of using `mcvhull3d`.
- All points have non-negative coordinates.

Moreover, optional parameters can be used. Thus, the input of the program should be of the form

```
> mcvhull2d(P);
```

result will be six figures of the m-convex hull of `P` with value $m \in [0,1)$, i.e., $m = 0$, 0.18, 0.36, 0.54, 0.72, 0.9, respectively.

```
> mcvhull2d(P, m);
```

where `m` is the fixed real number, such that $m \in [0,1)$. Then the result will be the figure of the m-convex hull of `P`.

```
> mcvhull2d(P, m1, m2);
```

here `m1` and `m2` are fixed real numbers, such that $0 \leq m1 < m2 < 1$. In this case, the program will show six figures of the m-convex hull of `P`, where $m1 \leq m \leq m2$.

```
> mcvhull2d(P, m1, m2, dfr);
```

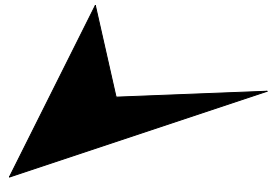
where `m1` and `m2` are fixed real numbers, such that $0 \leq m1 < m2 < 1$ and `dfr` is the number of frames (or figures) the user wants to be shown. Therefore, the result will be `dfr` figures of the m-convex hull of `P`, where $m1 \leq m \leq m2$.

All options of the program `mcvhull3d` are similar to the above description. In the following part, examples of running the programs will be given.

Firstly, we need to execute the package `MConvexHull`, then we can call the program `mcvhull2d` for generating the m-convex hull of sets in $\mathbb{R}^2$.

```
> with(MConvexHull);
> mcvhull2d([[3, 1], [1, 2]], 0.45);
```

Figure 4.1 shows a m-convex hull of set P, which contains two points $(3,1),(1,2) \in \mathbb{R}^2$, in case $m = 0.45$



m = 0.45

Figure 4.1

In case the user only gives the set of points `P`,

```
> P := [[14, 16], [13, 12], [5, 20], [25, 13],
  [7, 10], [10, 25], [1.5, 15], [16, 6],
  [26, 5], [20, 5], [17, 13], [15, 9.5]];
> mcvhull2d(P);
```

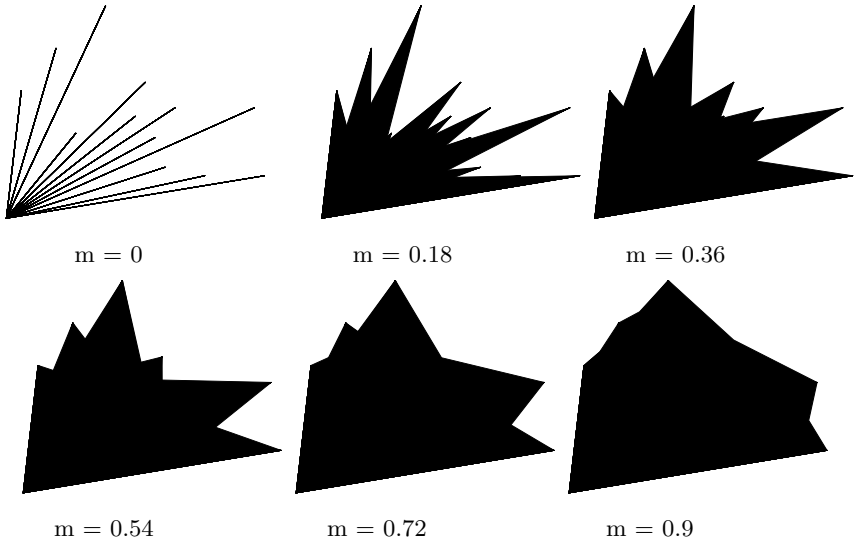
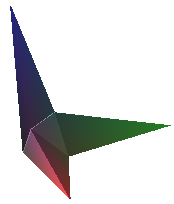


Figure 4.2

The program returns the animation which contains six frames as the m-convex hull of `P` with the value $m = 0, 0.18, 0.36, 0.54, 0.72, 0.9$, respectively (Figure 4.2).

In the three-dimensional space, the following example gives the m-convex hull of the set of three points $(0, 0, 1), (0, 1, 0), (1, 0, 0) \in \mathbb{R}^3$ with $m = 0.3$.

```
> P := [[1,0,0], [0,0,1], [0,1,0]]:
> mcvhull3d(P, 0.3);
```



m = 0.3

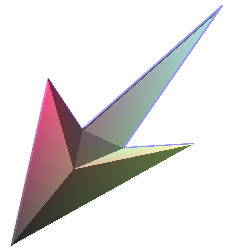
Figure 4.3

When the input set contains duplicate points, the program will show a warning message on the screen and continue to run normally after removing the duplication.

```
> P := [[2, 2, 1], [2, 2, 3], [3, 0, 1],
       [2, 2, 1]]:
> mcvhull3d(P, 0.4);
```

yields

```
Warning, There are duplicate points in
the set of points.
```



m = 0.4

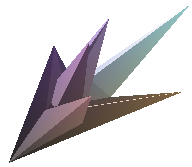
Figure 4.4

User can also give the range of the value `m`, $m1 \leq m \leq m2$ and number of figures `dfr`,

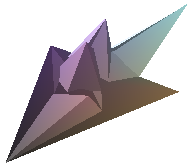
```
> P := [[2, 1, 4], [3, 2, 2], [3, 2, 4],
       [5, 5, 2], [3, 3, 5], [3, 7, 6]];
> mcvhull3d(P, 0.2, 0.8, 4);
```

Similar to the 2-dimensional case, in the 3-dimensional case, the program also returns the animation constructed by four frames. Figure 4.5 shows 4 figures as the m-convex hull of `P` with the value $m = 0.2, 0.4, 0.6, 0.8$, respectively.

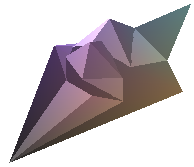
m = 0.2



m = 0.4



m = 0.6



m = 0.8

Figure 4.5

When the input is incorrect, the program will show an appropriate error message with a short hint about the problem. For example:

```
> P := [[0, 0], [2, 2], [3, 0], [3, 2]]:
> mcvhull2d(P, 0.3);
```

yields

```
Error, (in draw2d) The list of points
can not contain the origin.
```

```
> mcvhull2d([[3, 0]], 0.5);
```

shows

```
Error, (in draw2d) Invalid input, the set
of points needs at least 2 elements.
```

```
> P := [[2, 2, 1], [3, 0, 2], [3, 2, 4]]:
> mcvhull3d(P, 2);
```

gives

```
Error, (in draw3d) Invalid m, 0 <= m < 1.
```

4.6 Barycentric coordinate system

4.6.1 Introduction

The concept of barycentric coordinates was introduced by Möbius in 1827 ([60]). Barycentric coordinate systems are often used in geometry and computer graphics. Moreover, it is a very useful tool for investigating the concept of convexity, and has been studied by many authors (cf. [76], [75], [28]).

Cartesian coordinates are a foundational concept in mathematics, representing positions based on distances from a fixed origin point. In contrast, barycentric coordinates define a point's location relative to other specific points, not an origin. This makes them a type of local coordinate system. Moreover, the number of barycentric coordinates of a point does not correspond to the dimension of the space, but to the number of defined reference points.

The barycentric coordinates of a point P can be presented as the masses placed at the reference points, such that the point P is in the center of mass of these masses.

Let us consider the center of mass of two masses. Consider the case when two masses M_A and M_B, which are placed on two sides A and B of a massless stick, respectively. We want to find a pivot point in the stick such that the stick stays balanced.

If $M_A = M_B$, it is well known that the pivot should be placed at the midpoint of the segment AB. Let us consider the case that $M_A \neq M_B$.

It has been studied that (cf. [74]), the pivot position can be determined as

$$P = \frac{M_A}{M_A + M_B} A + \frac{M_B}{M_A + M_B} B. \tag{4.1}$$

In the expression (4.1), $\frac{M_A}{M_A + M_B}$ and $\frac{M_B}{M_A + M_B}$ are called the barycentric coordinates of the point P relative to the points A and B.

If we extend the number of masses to three, i.e, M_A, M_B, and M_C, which are placed at three vertices of a triangle $\triangle ABC$, then we can write

$$\begin{aligned} P = \frac{M_A}{M_A + M_B + M_C} A + \frac{M_B}{M_A + M_B + M_C} B \\ + \frac{M_C}{M_A + M_B + M_C} C. \end{aligned} \tag{4.2}$$

Let us consider the barycentric coordinates from within the linear interpolation.

Given two points A, and B, we would like to find the position of point P, stands at the ratio t between A and B $(0 \leq t \leq 1)$. It is well known that

$$P = (1 - t)A + tB \tag{4.3}$$

The expression (4.3) is the linear interpolation between A and B using the parameter t, also well known as the convex combination of A and B.

Let us consider $p_1 = 1 - t$, and $p_2 = t$, we have

$$P = p_1 A + p_2 B,$$

and

$$p_1 + p_2 = 1.$$

Similar to the barycentric coordinates in the sense of the center of mass, here (p_1, p_2) are the barycentric coordinates of the point P relative to A and B.

Let us assume, $A(x_A, y_A)$ and $B(x_B, y_B)$ in two-dimensional plane. We have

$$x_P = p_1 x_A + p_2 x_B$$
$$y_P = p_1 y_A + p_2 y_B,$$

and $p_1 + p_2 = 1$.

The values of p_1 and p_2 can be used to identify the relative position of the point P to A and B. In detail, when:

- $0 < p_1, p_2 < 1$, then the point P slided along the segment AB connect two points A and B;
- $p_1 = 0$, then the point P is coincident with the points B;
- $p_2 = 0$, then the point P is coincident with the points A.

Let us extend the number of reference points to three, i.e, A, B, C, and consider them in the form of a triangle $\triangle ABC$. We have

$$P = p_1 A + p_2 B + p_3 C,$$

where

$$p_1 + p_2 + p_3 = 1.$$

Giving the scalars for A, B, C as $A(x_A, y_A)$, $B(x_B, y_B)$ and $C(x_C, y_C)$, then we get

$$x_P = p_1 x_A + p_2 x_B + p_3 x_C$$
$$y_P = p_1 y_A + p_2 y_B + p_3 y_C,$$

and $p_1 + p_2 + p_3 = 1$.

Here, p_1, p_2, p_3 are the barycentric coordinates of the point P relative to A, B, and C.

Simmilarly, the position of P corresponds to the triangle $\triangle ABC$ can be determined, as follows

– If $p_i < 0$, for any values $i = 1, 2, 3$, then the point P stands outside the triangle $\triangle ABC$;

– If $p_i \geq 0$, for all values $i = 1, 2, 3$, then the point P is inside the triangle $\triangle ABC$.

More precisely, the case of "inside" the triangle $\triangle ABC$, contains the cases that the point P is lying on edges or coincident with the vertices of the $\triangle ABC$. In depth,

– If $p_1 = 0$, then the point P is laying on the edge BC of the triangle $\triangle ABC$;

– If $p_2 = 0$, then the point P is laying on the edge AC of the triangle $\triangle ABC$;

– If $p_3 = 0$, then the point P is laying on the edge AB of the triangle $\triangle ABC$;

– If $p_1 = 1$, then the point P is coincident with the vertex A of the triangle $\triangle ABC$;

– If $p_2 = 1$, then the point P is coincident with the vertex B of the triangle $\triangle ABC$;

– If $p_3 = 1$, then the point P is coincident with the vertex C of the triangle $\triangle ABC$.

By the linear interpolation form of the barycentric coordinates, in geometry, a barycentric coordinate system is also used to specify the location of a point within a simplex. In which the position of a point is defined with respect to the vertices of the simplex, i.e, the point is represented as a convex combination of all vertices of the simplex. In the following parts, we show the barycentric coordinate systems in $\mathbb{R}^2$ within the triangle and in $\mathbb{R}^3$ within the tetrahedron.

4.6.2 Barycentric coordinates on triangles

Consider a triangle T defined by its three vertices A, B, C. Each point Q located inside this triangle can be written as a unique convex combination of the three vertices.

In other words, for each Q there is a unique sequence of three numbers $\lambda_1, \lambda_2, \lambda_3 \geq 0$ such that

$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$

$$\text{and } Q = \lambda_1 A + \lambda_2 B + \lambda_3 C.$$

The three numbers $\lambda_1, \lambda_2, \lambda_3$ indicate the barycentric coordinates of the point Q with respect to the triangle T.

Given the points specified as $A = (a_x, a_y), B = (b_x, b_y), C = (c_x, c_y)$ and a point $Q = (q_x, q_y)$, the barycentric coordinates of the point Q can be found by solving the following system of linear equations:

$$\begin{aligned} ax \cdot \lambda_1 + b_x \cdot \lambda_2 + c_x \cdot \lambda_3 &= q_x \\ a_y \cdot \lambda_1 + b_y \cdot \lambda_2 + c_y \cdot \lambda_3 &= q_y \\ \lambda_1 + \lambda_2 + \lambda_3 &= 1. \end{aligned} \tag{4.4}$$

If $\lambda_1, \lambda_2, \lambda_3 \geq 0$, then Q is inside the triangle.

4.6.3 Barycentric coordinates on tetrahedrons

Since a tetrahedron is a 3-simplex, we can extend the concept to the three-dimensional space. Given the vertices of the tetrahedron as $A = (a_x, a_y, a_z)$, $B = (b_x, b_y, b_z)$, $C = (c_x, c_y, c_z)$, $D = (d_x, d_y, d_z)$ and a point $Q = (q_x, q_y, q_z)$, the barycentric coordinates of the point Q can be found by solving the following system of linear equations:

$$\begin{aligned} a_x \cdot \lambda_1 + b_x \cdot \lambda_2 + c_x \cdot \lambda_3 + d_x \cdot \lambda_4 &= q_x \\ a_y \cdot \lambda_1 + b_y \cdot \lambda_2 + c_y \cdot \lambda_3 + d_y \cdot \lambda_4 &= q_y \\ a_z \cdot \lambda_1 + b_z \cdot \lambda_2 + c_z \cdot \lambda_3 + d_z \cdot \lambda_4 &= q_z \\ \lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 &= 1. \end{aligned} \tag{4.5}$$

If $\lambda_1, \lambda_2, \lambda_3, \lambda_4 \geq 0$, then Q is inside the tetrahedron.

In the previous section, we presented the m-convex hulls of sets of finitely many points in $\mathbb{R}^2$ and $\mathbb{R}^3$. In detail, in $\mathbb{R}^2$ the m-convex hulls of sets are constructed by the union of finitely many triangles with known vertices. Similarly, in $\mathbb{R}^3$ the m-convex hulls of sets are built up by the union of finitely many tetrahedrons with known vertices. With that structure, by applying the barycentric coordinate system to determine the position of a given point with respect to a simplex (triangle or tetrahedron), we develop a program to decide, whether a given point in $\mathbb{R}^2$ or in $\mathbb{R}^3$ is an element of a m-convex hull of set or not.

In the next section, a description of the program and examples of how it is run will be shown.

4.7 Description of the program functions `PointInMCVHull2d` and `PointInMCVHull3d`

Our programs `PointInMCVHull2d` and `PointInMCVHull3d` are also contained in the Maple package `MConvexHull`. Similar to other Maple programs, these programs can be accessed through the forms

```
> with(MConvexHull);
> PointInMCVHull2d(arguments);
> PointInMCVHull3d(arguments);
```

To run the program, users have to give four input parameters: the given point called `Q`, the list of points `P` in $\mathbb{R}^2$ or in $\mathbb{R}^3$, the fixed real value `m`, and a boolean argument as the last parameter. Thus, the input of the program should be of the form

```
> PointInMCVHull2d(Q, P, m, true/false);
> PointInMCVHull3d(Q, P, m, true/false);.
```

The program will decide whether the point `Q` is an element of the `m`-convex hull of the set of points `P` or not. Moreover,

if the last parameter is `true`, the program will also show the animation for more information.

The given point `Q`, the list of points `P`, and all of the points in `P` have to be given as a Maple list structure. Additional requirements and conditions of the list of points `P` can be found in the description of functions `mcvhull2d` and `mcvhull2d`.

In the following part, examples of running the programs will be shown.

```
> with(MConvexHull);
  P:=[[3, 1], [1, 2]];
  PointInMCVHull2d([1, 1], P, 0.45, false);
```

returns

```
True.
```

In this example, the program briefly answers `True`. It means the given point $(1, 1)$ is inside the m-convex hull of the set P. Moreover, if the last argument has the value `true`, the program will show the figure of the m-convex hull and the point for giving users more information.

```
> P:=[[3, 1], [1, 2]];
  PointInMCVHull2d([1, 1], P, 0.45, true);
```

gives

```
True.
```

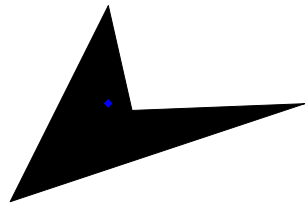


Figure 4.6

```
> P:=[[3, 1], [1, 2]];
  PointInMCVHull2d([3, 1], P, 0.45, true);
```

yields

```
True.
```

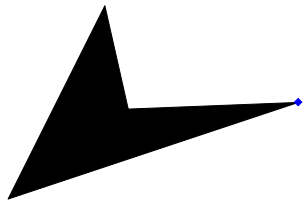


Figure 4.7

With the help of the computer, we can get the decision even in the case when the set P contains many points, for example

```
> P := [[14, 16], [13, 12], [5, 20], [25, 13],
    [7, 10], [10, 25], [1.5, 15], [16, 6],
    [26, 5], [20, 5], [17, 13], [15, 9.5]];
  PointInMCVHull2d([15, 16], P, 0.3, true);
```

shows

```
False.
```

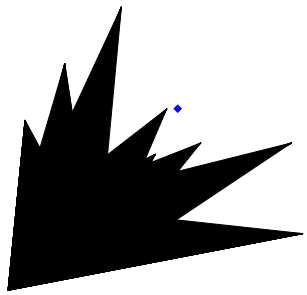


Figure 4.8

Similarly, in the three-dimensional space, the next example gives the answer whether the point $\left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right)$ is inside the m-convex hull of the set of the three points $\left(\frac{1}{2}, \frac{1}{2}, 1\right), (1,0,0), (0,1,0)$ in $\mathbb{R}^3$ with $m = 0.25$.

```
> Q := [1/2, 1/2, 1/2]:
  P := [[1/2, 1/2, 1], [1, 0, 0], [0, 1, 0]]:
  PointInMCVHull3d(Q, P, 0.25, true);
```

shows

```
false.
```

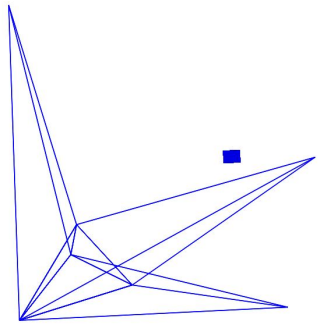


Figure 4.9

Remarks 1. The m-convex hull of a set in $\mathbb{R}^3$ is a solid polyhedron as a union of finitely many tetrahedrons. Therefore, in the above example, the m-convex hull's transparency is set to 0 just for easily getting the position of the point with respect to the polyhedron, especially in case when the point is inside the polyhedron, as we can see in the next example:

```
> Q := [1/4, 1/4, 1/4]:
  P := [[1/2, 1/2, 1], [1, 0, 0], [0, 1, 0]]:
  PointInMCVHull3d(Q, P, 0.5, true);
```

returns

True.

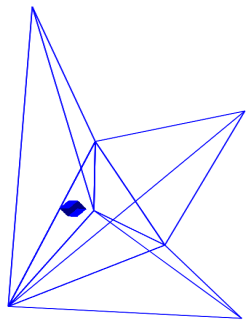


Figure 4.10

4.8 Explanation of the program

In this section, some main procedure codes will be presented with detailed explanations to illustrate how our programs work.

4.8.1 The function `mconvex2d`

```
1 f_plot2d := proc(m)
2     local f, i, j;
3     f := (i, j, m) -> display(polygon([P[i],
              m*(P[i] + P[j])/(m + 1), P[j],
          [0, 0]]));
4     display(seq(seq(f(i, j, m), j = i+1..n),
          i = 1..n-1), scaling=constrained,
          axes=none);
5 end proc;
```

According to the algorithm in Section 4.3, our program visualizes the m-convex hull of the given set of points `P` in the two-dimensional plane by displaying the union of all m-convex hulls of each pair of points contained in `P`. That is the main process in the above procedure f_plot2d. Here we have two important global variables `P` – the given set of points, and `n` – the

number of points of the set `P`.

In detail, for every pair `P[i], P[j]` in `P` (line 4: $j = i + 1..n, i = 1..n-1$), the procedure will display the polygon with four vertices

$$[0,0], P[i], P[j], \frac{m}{m+1}(P[i]+P[j]),$$

(line 3), where `m` is a given parameter. It is equivalent to the union of two triangles

$$\left[[0,0], P[i], \frac{m}{m+1}(P[i]+P[j])\right],$$

and

$$\left[[0,0], P[j], \frac{m}{m+1}(P[i]+P[j])\right],$$

as mentioned in Section 4.3.

In our program, preprocessing and error catching also play key roles. Those steps are included in the following procedure.

```
6  draw2d := proc(L, M1, M2, dfr)
7      local p, k, m, m1, m2;
8      P := L;
9      m1 := convert(M1, rational, exact);
10     m2 := convert(M2, rational, exact);
11     if not type(P, list) then
12         error "Invalid set of points.";
13     end if;
14     n := numelems(P);
15     P := MakeUnique(P);
16     if numelems(P) < n then
17         WARNING("There are duplicate points in
               the set of points.");
18         n:= numelems(P);
19     end if;
20     if n < 2 then
21         error "Invalid input, the set of
               points needs at least 2 elements.";
```

```
22    end if;
23    for p in P do
24       if not type(p, list) then
25          error "Invalid set of points.";
26       end if;
27       if nops(p)<>2 then
28          error "Invalid set of points.";
29       end if;
30       if not type(evalf(p[1]), numeric) then
31          error "Invalid set of points.";
32       end if;
33       if not type(evalf(p[2]), numeric) then
34          error "Invalid set of points.";
35       end if;
36    end do;
37    for k to n do
38       if P[k][1] = 0 and P[k][2] = 0 then
39          error "The list of points can not
               contain the origin.";
40       end if;
41       if P[k][1] < 0 or P[k][2] < 0 then
42          error "All input points have to
               have non-negative coordinates.";
43       end if;
44    end do;
45    if not type(evalf(m1), numeric) then
46       error "Invalid input, m value.";
47    end if;
48    if not type(evalf(m2), numeric) then
49       error "Invalid input, m value.";
50    end if;
51    if not type(dfr, integer) then
52       error "Invalid number of frames.";
53    end if;
```

```
54      if m1 < 0 or 1 <= m1 then
55          if m1 <> m2 then
56              error "Invalid m1, 0 <= m1 < 1.";
57          else
58              error "Invalid m, 0 <= m < 1.";
59          end if;
60      end if;
61      if m2 < 0 or 1 <= m2 then
62          error "Invalid m2, 0 <= m2 < 1.";
63      end if;
64      if m2 < m1 then
65          error "Invalid input, 0 <= m1 <= m2 <
                1.";
66      end if;
67      if m1 = m2 then
68          if dfr <> 1 then
69              error "Invalid number of frames."
70          end if;
71          print(M1);
72          print(f_plot2d(m1));
73      else
74          if dfr < 2 then
75              error "Invalid number of frames."
76          end if;
77          animate(f_plot2d, [m], m = m1..m2,
                frames=dfr);
78      end if;
79  end proc;
```

Following the code of function draw2d by top to bottom:

- Lines 8 – 9: rational values of `m1` and `m2` are computed to make sure our computations are exact and do not contain any approximations.
- Lines 11 – 13: checking the valid form of given set of points `P`. As mentioned in Section 4.5, `P` has to be in the form of

a Maple list.

- Lines 16 – 19: checking whether there are duplicate points in `P` and giving warning message if necessary.
- Lines 20 – 22: checking the condition about minimum number of points in `P` (two points in two-dimensional plane and three points in three-dimensional space).
- Lines 23 – 44: checking the conditions related to coordinates for each points `p` in `P`. In detail, each point has two coordinate values, coordinate values are positive numbers, and lastly, the origin is not contained in the set of points `P`.
- Lines 45 – 66: checking the conditions for given values `m1, m2, dfr`.
  - `m1, m2` are numbers,
  - `dfr` is a positive integer,
  - $0 \leq$ `m1` $\leq$ `m2` < 1.
- Lines 67 – 78: when the given parameters pass all previous error catches, with the correct number of frames `dfr`, the procedure `f_plot2d` will be called as a singular plot in case `m1` $=$ `m2` (lines 67 – 72) and an animation when `m1` $<$ `m2` (lines 73 – 78).

```
80 mcvhull2d:= proc()
81     if nargs = 1 then
82         draw2d(args,0,0.9,6);
83     elif nargs = 2 then
84         draw2d(args, args[2],1);
85     elif nargs = 3 then
86         draw2d(args,6);
87     elif nargs = 4 then
88         draw2d(args);
```

```
89     else
90         error "Invalid input parameters.";
91     end if;
92 end proc;.
```

Lastly, the main function mcvhull2d will accept one to four arguments. All options are mentioned in Section 4.5. Since procedure draw2d needs four arguments `P, m, m2, dfr`, necessary parameters will be added as follows:

- Line 82: when only a list of points `P` is given, the function will return an animation made of six figures with value $0 \leq \texttt{m} \leq 0.9$, respectively.

  ```
  > mcvhull2d(P);
  ```

- Line 84: when a list of points `P` and a value `m` are given, the function will return a single plot (`m1` = `m2` and `dfr` = 1).

  ```
  > mcvhull2d(P, m);
  ```

- Line 86: function will return an animation made of six figures with value $\texttt{m1} \leq \texttt{m} \leq \texttt{m2}$, when three arguments are given.

  ```
  > mcvhull2d(P, m1, m2);
  ```

- Line 88: when full four arguments are given, the function will call the procedure draw2d.

  ```
  > mcvhull2d(P, m1, m2, dfr);
  ```

4.8.2 The function `mconvex3d`

In the case of the three-dimensional space, our programs will work similarly. The preprocessing, error catching will included in `draw3d`, which works correspondingly to draw2d and the function `mcvhull3d` will work similarly to the function

mcvhull2d. Therefore, we will show and explain only the main procedure f_plot3d.

```
1 f_plot3d := proc(m)
2     local f, i, j, k, mF;
3     mF := m/(m+1);
4     f := (i, j, k, mF) ->
          display({tetrahedron([P[i], mF * (P[i]
          + P[j]), mF * (P[i] + P[k]), [0, 0,
          0]]), tetrahedron([P[j], mF * (P[i] +
          P[j]), mF * (P[j] + P[k]), [0, 0,
          0]]), tetrahedron([P[k], mF * (P[i] +
          P[k]), mF * (P[j] + P[k]), [0, 0,
          0]]), tetrahedron([mF * (P[i] + P[j]),
          mF * (P[i] + P[k]), mF * (P[j] +
          P[k]), [0, 0, 0]])});
5     display(seq(seq(seq(f(i, j, k, mF), k =
          j+1..n), j = i+1..n-1), i = 1..n-2),
          scaling=constrained, axes=none);
6 end proc;
```

According to the algorithm in Section 4.4, our program f_plot3d visualizes the m-convex hull of the given set of points `P` in three-dimensional space by displaying the union of m-convex hulls of each group of three points contained in `P`. That is the main process in the above procedure `f_plot3d`.

In detail, for every group of three points `P[i], P[j], P[k]` in `P` (line 5: $k = j + 1..n, j = i + 1..n - 1, i = 1..n - 2$), the procedure will display the union of four tetrahedrons (line 4)

1. tetrahedron[P[i], mF(P[i] + P[j]), mF(P[i] + P[k]), [0, 0, 0]],
2. tetrahedron[P[j], mF(P[i] + P[j]), mF(P[j] + P[k]), [0, 0, 0]],
3. tetrahedron[P[k], mF(P[i] + P[k]), mF(P[j] + P[k]), [0, 0, 0]],

4. tetrahedron[mF(P[i] + P[j]), mF(P[i] + P[k]), mF(P[j] + P[k]), [0, 0, 0]],

where $mF = \dfrac{m}{(m+1)}$ (line 3).

4.8.3 The functions `PointInMCVHull2d` and `PointInMCVHull3d`

According to Section 4.3 and Section 4.4, the m-convex hull of a set of points is the union of triangles (in the two-dimensional case) and tetrahedrons (in the three-dimensional case). To determine whether a given point belongs to the m-convex hull of a set of points or not, our programs will check whether the point belongs to any of those triangles and tetrahedrons. For that, as mentioned in Section 4.6, we will solve the system of linear equations (4.4), (4.5), and then check their results.

```
1  TriangleList := proc(P, m)
2      local mR, mF, T, T1, T2, i, j;
3      T := [];
4      mR := convert(m, rational, exact);
5      mF := mR/(mR + 1);
6      for i to nops(P) - 1 do
7          for j from i + 1 to nops(P) do
8              T1 := [P[i], mF * (P[i] + P[j]),
                   [0, 0]];
9              T2 := [P[j], mF * (P[i] + P[j]),
                   [0, 0]];
10             T := [op(T), T1, T2];
11         end do;
12     end do;
13     return T;
14 end proc;.
```

In this procedure TriangleList, the list of triangles `T` is computed (in the two-dimensional plane), each triangle is presented

by its vertices. Lines 6 – 12, for each pair of points `P[i], P[j]` in `P` two triangles are built

1. T1 = [P[i], mF(P[i] + P[j]), [0, 0]] (line 8),
2. T2 = [P[j], mF(P[i] + P[j]), [0, 0]] (line 9),

where $mF = \frac{m}{(m+1)}$ (line 5).

The next procedure TetrahedronList will construct the list of tetrahedrons `T` (in the three-dimensional plane), similarly, each tetrahedron is presented by its vertices. In lines 20 – 30, for each group of three points `P[i], P[j], P[k]` in `P` four tetrahedrons are built

1. T1 = [P[i], mF(P[i] + P[j]), mF(P[i] + P[k]), [0, 0, 0]] (line 23),
2. T2 = [P[j], mF(P[i] + P[j]), mF(P[j] + P[k]), [0, 0, 0]] (line 24),
3. T3 = [P[k], mF(P[i] + P[k]), mF(P[j] + P[k]), [0, 0, 0]] (line 25),
4. T4 = [mF(P[i] + P[j]), mF(P[i] + P[k]), mF(P[j] + P[k]), [0, 0, 0]] (line 26),

where $mF = \frac{m}{(m+1)}$ (line 19).

```
15 TetrahedronList := proc(P, m)
16     local mR, mF, T, T1, T2, T3, T4, i, j, k;
17     T := [];
18     mR := convert(m, rational, exact);
19     mF := mR/(mR + 1);
20     for i to nops(P) - 2 do
21         for j from i + 1 to nops(P)-1 do
22             for k from j+1 to nops(P) do
23                 T1 := [P[i], mF * (P[i] + P[j]),
                       mF * (P[i] + P[k]), [0, 0,
                       0]];
```

```
24                T2 := [P[j], mF * (P[i] + P[j]),
                     mF * (P[j] + P[k]), [0, 0,
                     0]];
25                T3 := [P[k], mF * (P[i] + P[k]),
                     mF * (P[j] + P[k]), [0, 0,
                     0]];
26                T4 := [mF * (P[i] + P[j]), mF *
                     (P[i] + P[k]), mF * (P[j] +
                     P[k]), [0, 0, 0]];
27                T := [op(T), T1, T2, T3, T4];
28             end do;
29          end do;
30       end do;
31       return T;
32    end proc;.
```

The following procedure PointPosition will check the position between the given point `p` and one triangle or tetrahedron `tr`.

```
33    PointPosition:= proc(p, tr)
34       local i, M, u, A, T;
35       A:=p;
36       T:=tr;
37       for i to nops(T) do
38          T[i] := [op(T[i]), 1];
39       end do;
40       M := Matrix(T);
41       M := LinearAlgebra:-Transpose(M);
42       A:=[op(A),1];
43       A:=convert(A, Vector);
44       u:=LinearSolve(M,A);
45       for i from 1 to Dimension(u) do
46          if u[i] < 0 then return 0; end if;
47       end do;
48       return 1;
```

```
49 end proc;.
```

In both two and three-dimensional cases, for constructing a system of linear equations, we need to extend one more coordinate in both point vector of coordinate (line 42) and matrix of triangle or tetrahedron (lines 37 – 39) (confirmed in Section 4.6, equations (4.4), (4.5)) simply with number 1. Then, the program will solve the system of linear equations (lines 40 – 44) and check the result (lines 45 – 48). If all coordinates of the result are positive, then the program will return `1`, i.e, the given point `p` belongs to the triangle or tetrahedron. Otherwise, it will return `0`.

```
50 PointInMCVHull2d := proc(p, P, m, animate)
51     local TL, i, po, flag;
52     flag:=0;
53     TL:=TriangleList(P,m);
54     for i from 1 to nops(TL) do
55         po:=PointPosition(p, TL[i]);
56         if po = 1 then flag:=1; break; end if;
57     end do;
58     if (flag=1) then
59         print(True) else print(False);
60     end if;
61     if (animate=true) then
62         f_plot2dwithpoint(p, P, nops(P), m);
63     end if;
64 end proc;.
```

Lastly, the main procedure PointInMCVHull2d first builds the list of all triangles (line 53). Therefore, in case any of the triangles of that list contains the given point `p`, the function will return `True`, i.e, the given point `p` belongs to the m-convex hull of a set of points `P`. In other case, it will return `False` (lines 54 – 60). In case the user chooses the option to see the animation to get more visual information about the answer, procedure `f_plot2dwithpoint` will be called for showing

the simple plot (lines 61 – 63). This function is the same as `f_plot2dwithpoint`, which shows the given point `p` and the m-convex hull.

The procedure `PointInMCVHull3d` works similarly, with the list of tetrahedrons.

Summary

In this section, we summarize the main results of this doctoral dissertation. Specifically, we developed new programs using the computer algebra system Maple to investigate certain functional equations and convexity problems. All results have been thoroughly covered in the previous chapters and are comprehensively elaborated in the papers [39], [40], [41], and [42]. All of our computer programs use symbolic computations, which means they can handle mathematical expressions exactly, without approximations. As a result, they will give the exact solutions to the given problems. By this property, using the terminology of 'mathability', they 'increase the level of mathability' (or 'improve the mathability') of the underlying systems.

With the motivation of developing program functions related to functional equations, and inspired by the function `lfesolve` developed by A. Gilányi (in [43], [44], and [17]). In the first part of the dissertation, we introduced the Maple package `FunctionalEquations`, which contains program functions that investigate problems related to functional equations. More precisely, the Maple package FunctionalEquations includes four sub-packages:

1. `LFESolve`, which includes the function `lfesolve`;
2. `AlienationCheck`, which includes the function `isalien`;
3. `LeviCivitafesolve`, which includes the function `LCfesolve`;

4. `FETypes`, which includes three functions `fewhattype`, `fetype`, and `feinfo`.

Three sub-packages `AlienationCheck, LeviCivitafesolve, FETypes` are newly developed as new results of the dissertation, and were presented in Chapter 1, Chapter 2, Chapter 3, respectively.

In detail, in Chapter 1, we presented a computer program `isalien` that investigates the alienness and strong alienness of linear functional equations of the type

$$\sum_{i=0}^{n+1} f_i(p_i x + q_i y) = 0 \qquad (x, y \in X), \tag{1}$$

where n is a positive integer, $p_0, \ldots, p_{n+1}$ and $q_0, \ldots, q_{n+1}$ are rational numbers, X, Y are linear spaces and $f_0, \ldots, f_{n+1} : X \to Y$ are unknown functions.

In Chapter 2, we introduced a computer program `LCfesolve`, that investigates solutions of Levi-Civita type functional equations, i.e., the class

$$f(x + y) = \sum_{i=1}^{n} g_i(x) h_i(y), \tag{2}$$

where n is a positive integer, G is an Abelian group and $f, g_i, h_i : G \to \mathbb{C}$ $(i = 1, 2, \ldots, n)$ are unknown functions.

In Chapter 3, three computer programs related to the determination of the types of functional equations are mentioned. In depth, the program `fewhattype(fe)` determines all types or classes to which the given functional equations `fe` belongs; the program `fetype(fe, type)` indicates whether the given functional equation `fe` belongs to the given `type`, or not; and the program `feinfo()` is used to get information about all considered types of functional equations.

Moreover, related to the concept of m-convexity, a new Maple package `MConvexHull` was developed, which includes four functions `mcvhull2d, mcvhull3d, PointInMCVHull2d,`

`PointInMCVHull3d`. This package is the main result, which was shown in Chapter 4.

In detail, we presented two computer programs, `mconvex2d` and `mconvex3d`, that visualize the animation of the m-convex hull of sets of points on the Cartesian coordinate systems for a two-dimensional plane and a three-dimensional space.

Lastly, this chapter also includes the detailed description of two other programs, `PointInMCVHull2d` and `PointInMCVHull3d`, which determine whether a given point is an element of the m-convex hull of a set of points or not in both two and three-dimensional cases.

Bibliography

[1] J. Aczél. Über eine Klasse von Funktionalgleichungen. *Comment. Math. Helv.*, 21:247–252, 1948.

[2] J. Aczél. *Vorlesungen über Funktionalgleichungen und ihre Anwendungen*. Birkhäuser, Basel, 1961. Lehrbücher und Monographien aus dem Gebiete der exakten Wissenschaften, Mathematische Reihe, Bd. 25.

[3] J. Aczél. *Lectures on Functional Equations and Their Applications*, volume 19 of *Mathematics in Science and Engineering*. Academic Press, New York–London, 1966.

[4] M. Adam. Alienation of the quadratic and additive functional equations. *Anal. Math.*, 45:449–460, 04 2019.

[5] Y. Aissi, D. Zeglami, and B. Fadli. Alienation of drygas' and cauchy's functional equations. *Ann. Math. Sil.*, 35, 04 2021.

[6] A. Bahyrycz and J. Sikorska. On a general bilinear functional equation. *Aequationes Math.*, 95(6):1257–1279, 2021.

[7] S. Baják and Z. Páles. Computer aided solution of the invariance equation for two-variable Gini means. *Comput. Math. Appl.*, 58:334–340, 2009.

[8] S. Baják and Z. Páles. Computer aided solution of the invariance equation for two-variable Stolarsky means. *Appl. Math. Comput.*, 216(11):3219–3227, 2010.

[9] S. Baják and Z. Páles. Solving invariance equations involving homogeneous means with the help of computer. *Appl. Math. Comput.*, 219(11):6297–6315, 2013.

[10] P. Baranyi and A. Gilányi. Mathability: emulating and enhancing human mathematical capabilities. In 4^{th} *IEEE International Conference on Cognitive Infocommunications (CogInfoCom)*, pages 555–558. IEEE, 2013.

[11] M. Bessenyei. Toader convexity revisited. The 61st International Symposium on Functional Equations, 2025.

[12] P. Biró and M. Csernoch. Deep and surface metacognitive processes in non-traditional programming tasks. In 5^{th} *IEEE Conference on Cognitive Infocommunications (CogInfoCom)*, pages 49–54. IEEE, 2014.

[13] P. Biró and M. Csernoch. The mathability of computer problem solving approaches. In 6^{th} *IEEE Conference on Cognitive Infocommunications (CogInfoCom)*. IEEE, 2015.

[14] P. Biró and M. Csernoch. The mathability of spreadsheet tools. In 6^{th} *IEEE Conference on Cognitive Infocommunications (CogInfoCom)*. IEEE, 2015.

[15] G. G. Borus and A. Gilányi. On a computer program for solving systems of functional equations. In 4^{th} *IEEE International Conference on Cognitive Infocommunications (CogInfoCom)*, page 939. IEEE, 2013.

[16] G. G. Borus and A. Gilányi. Solving systems of linear functional equations with computer. In 4^{th} *IEEE International Conference on Cognitive Infocommunications (CogInfoCom)*, pages 559–562. IEEE, 2013.

[17] G. G. Borus and A. Gilányi. Computer assisted solution of systems of two variable linear functional equations. *Aequationes Math.*, 94:723–736, 2020.

[18] A. L. Cauchy. *Cours d'analyse de l'École Royale Polytechnique, Première Partie, Analyse algébrique.* Paris, 1821.

[19] K. Chmielewska and A. Gilányi. Mathability and computer aided mathematical education. In 6^{th} *IEEE Conference on Cognitive Infocommunications (CogInfoCom)*, pages 473–477. IEEE, 2015.

[20] K. Chmielewska and A. Gilányi. Educational context of mathability. *Acta Polytech. Hung.*, 15:223–237, 2018.

[21] S. Czirbusz. Testing regularity of functional equations with computer. *Aequationes Math.*, 84(3):271–283, 2012.

[22] Z. Daróczy. Notwendige und hinreichende Bedingungen für die Existenz von nichtkonstanten Lösungen linearer Funktionalgleichungen. *Acta Sci. Math. (Szeged)*, 22:31–41, 1961.

[23] Z. Daróczy. A bilineáris függvényegyenletek egy osztályáról (On a class of bilinear functional equations), (in Hungarian). *Mat. Lapok*, 15:52–86, 1964.

[24] J. Dhombres. Relations de dépendance entre les équations fonctionnelles de Cauchy. *Aequationes Math.*, 35(2-3):186–212, 1988.

[25] J. d'Alembert. Addition au mémoire sur la courbe que forme une corde tendue mise en vibration. *Hist. Acad. Berlin*, 6:355–360, 1750.

[26] J. d'Alembert. Memoire sur les principes de mecanique. *Hist. Acad. Sci*, pages 278–286, 1769.

[27] B. Fadli. Alienation of Cauchy's and the quadratic functional equations on semigroups. *Aequationes Math.*, 95(3):527–534, 2021.

[28] M. Floater, K. Hormann, and G. Kós. A general construction of barycentric coordinates over convex polygons. *Adv. Comput. Math.*, 24:311–331, 01 2006.

[29] M. Fréchet. Une définition fonctionnelle des polynômes. *Nouv. Ann.*, 49:145–162, 1909.

[30] M. Fréchet. Les polynômes abstraits. *Journal de Math. Pures et Appl. Sér. IX*, 8:71–92, 1929.

[31] C. F. Gauss. *Theoria Motus Corporum Coelestium*. Hamburg, 1809.

[32] R. Ger. Additivity and exponentiality are alien to each other. *Aequationes Math.*, 80(1-2):111–118, 2010.

[33] R. Ger. Alienation of additive and logarithmic equations. *Ann. Univ. Sci. Budap. Rolando Eötvös, Sect. Comput.*, 40:269–274, 01 2013.

[34] R. Ger. On alienation of two functional equations of quadratic type. *Aequationes Math.*, 95:1169–1180, 2021.

[35] R. Ger and M. Sablik. Alien functional equations: a selective survey of results. In *Developments in functional equations and related topics*, volume 124 of *Springer Optim. Appl.*, pages 107–147. Springer, Cham, 2017.

[36] A. Gilányi and A. Lewicka. On linear functional equations modulo $\mathbb{Z}$. *Aequationes Math.*, 95(6):1301–1311, 2021.

[37] A. Gilányi, N. Merentes, and R. Quintero. Mathability and an animation related to a convex-like property. In 7^{th} *IEEE Conference on Cognitive Infocommunications (CogInfoCom)*, pages 227–232. IEEE, 2016.

[38] A. Gilányi, N. Merentes, and R. Quintero. Presentation of an animation of the m-convex hull of sets. In 7^{th} *IEEE Conference on Cognitive Infocommunications (CogInfoCom)*, pages 307–308. IEEE, 2016.

[39] A. Gilányi, R. Quintero, and L. N. To. Computer-assisted investigations of m-convex hulls of sets. *Manuscript*, 2025.

[40] A. Gilányi and L. N. To. Computer assisted investigation of alienness of linear functional equations. *Aequationes Math.*, 97:1185–1199, 2023.

[41] A. Gilányi and L. N. To. Computer-assisted investigation of Levi-Civita type functional equations. *Manuscript*, 2025.

[42] A. Gilányi and L. N. To. On a computer program determining types of functional equations. *Aequationes Math. (Published online first)*, 2025.

[43] A. Gilányi. *Charakterisierung von monomialen Funktionen und Lösung von Funktionalgleichungen mit Computern*. Diss., Universität Karlsruhe, Karlsruhe, Germany, 1995.

[44] A. Gilányi. Solving linear functional equations with computer. *Math. Pannon.*, 9(1):57–70, 1998.

[45] E. Gselmann, G. Kiss, and C. Vincze. On a class of linear functional equations without range condition. *Aequationes Math.*, 94(3):473–509, 2020.

[46] A. Házy. Solving functional equations with computer. In *MicroCAD 2004 International Scientific Conference, vol. Section E*, page 35–39. University of Miskolc, Miskolc, 2004.

[47] A. Házy. Solving linear two variable functional equations with computer. *Aequationes Math.*, 67(1-2):47–62, 2004.

[48] A. Joós and A. Kovari. Convexity and mathability. *Acta Polytech. Hung.*, 19(1), 2022.

[49] Z. Kominek and J. Sikorska. Alienation of the logarithmic and exponential functional equations. *Aequationes Math.*, 90, 2016.

[50] M. Kuczma. *An Introduction to the Theory of Functional Equations and Inequalities*, volume 489 of *Prace Naukowe Uniwersytetu Śląskiego w Katowicach*. Państwowe Wydawnictwo Naukowe — Uniwersytet Śląsk, Warszawa–Kraków–Katowice, 1985.

[51] M. Kuczma. *An introduction to the theory of functional equations and inequalities. Cauchy's equation and Jensen's inequality*. Birkhäuser Verlag, 2nd edition, 2009.

[52] T. Lara, N. Merentes, Z. Páles, R. Quintero, and E. Rosales. On m-convexity on real linear spaces. *UPI-JMB*, 1(2), 2018.

[53] T. Lara, R. Quintero, E. Rosales, and J. L. Sánchez. On a generalization of the class of Jensen convex functions. *Aequationes Math.*, 90(3):569–580, 2016.

[54] T. Lara, E. Rosales, and J. L. Sánchez. New properties of m-convex functions. *Int. J. Math. Anal.*, 9(15):735–742, 2015.

[55] A. M. Legendre. *Eléments de Géometrie*. Paris, 1791.

[56] L. Losonczi. Bestimmung aller nichtkonstanten Lösungen von linearen Funktionalgleichungen. *Acta Sci. Math. (Szeged)*, 25:250–254, 1964.

[57] A. Lundberg. Some methods for a Sutô–Aczél project. *Aequationes Math.*, 89:957–980, 2015.

[58] G. Maksa. On the alienation of the logarithmic and exponential cauchy equations. *Aequationes Math.*, 92(3):543–547, 2018.

[59] G. Maksa and M. Sablik. On the alienation of the exponential Cauchy equation and the Hosszú equation. *Aequationes Math.*, 90(1):57–66, 2016.

[60] A. F. Möbius. *Der barycentrische Calcul.* J.A. Barth., Leipzig, 1827.

[61] T. Nadhomi, C. P. Okeke, M. Sablik, and T. Szostok. On a new class of functional equations satisfied by polynomial functions. *Aequationes Math.*, 95(6):1095–1117, 2021.

[62] T. Nadhomi, C. P. Okeke, M. Sablik, and T. Szostok. On a class of functional inequalities, a computer approach. *Period. Math. Hung.*, 2025.

[63] C. P. Okeke, W. I. Ogala, and T. Nadhomi. On symbolic computation of c.p.okeke functional equations using python programming language. *Aequationes Math.*, 98:483–502, 2024.

[64] C. P. Okeke and M. Sablik. Functional equation characterizing polynomial functions and an algorithm. *Results Math.*, 77(3), 2022.

[65] J. V. Pexider. Notiz über funktionaltheoreme. *Monatsh. Math. Phys.*, 14:293–301, 1903.

[66] B. Sobek. Alienation of the jensen, cauchy and d'alembert equations. *Ann. Math. Sil.*, 30, 01 2016.

[67] T. Szostok. Alienation of two general linear functional equations. *Aequationes Math.*, 94(2):287–301, 2020.

[68] L. Székelyhidi. The stability of linear functional equations. *C. R. Math. Rep. Acad. Sci. Canada*, 3(2):63–67, 1981.

[69] L. Székelyhidi. On a class of linear functional equations. *Publ. Math. Debrecen*, 29(1-2):19–28, 1982.

[70] L. Székelyhidi. *On the Levi-Civita Functional Equation.* Berichte der mathematisch-statistischen Sektion in der Forschungsgesellschaft Joanneum Graz, Graz, 1988.

[71] L. Székelyhidi. On a linear functional equation. *Aequationes Math.*, 38(2-3):113–122, 1989.

[72] L. Székelyhidi. *Convolution Type Functional Equations on Topological Abelian Groups*. World Scientific Publishing Co. Inc., Teaneck, NJ, 1991.

[73] G. Toader. Some generalizations of the convexity. In *Proc. Coll. Approx. and Opt. (Cluj-Napoca)*, pages 329–338. Univ. Cluj-Napoca, Cluj-Napoca, 1984.

[74] J. Vince. *Barycentric Coordinates*, chapter 16, pages 371–399. Springer, sixth edition, 2022.

[75] J. Warren. Barycentric coordinates for convex polytopes. *Adv. Comput. Math.*, 6:97–108, 1996.

[76] J. Warren, S. Schaefer, A. Hirani, and M. Desbrun. Barycentric coordinates for convex sets. *Adv. Comput. Math.*, 27:319–338, 10 2007.

[77] W. H. Wilson. On a certain general class of functional equations. *Amer. J. Math.*, 40(3):263–282, 1918.

[78] W. H. Wilson. On certain related functional equations. *Bull. Amer. Math. Soc.*, 26:300–312, 1920.

Appendix A

List of author's publications

List of publications related to the dissertation

[**P1**] A. Gilányi and L. N. To. On a computer program determining types of functional equations.

Aequationes. Math. (Published online first), 2025.

ISSN: 0001-9054.

DOI: http://dx.doi.org/10.1007/s00010-025-01184-3

SJR: Q3 (2024)

IF: 0.7 (2024)

[**P2**] A. Gilányi and L. N. To. Computer assisted investigation of alienness of linear functional equations.

Aequationes. Math. 97, 1185-1199, 2023.

ISSN: 0001-9054.

DOI: http://dx.doi.org/10.1007/s00010-023-00978-7

SJR: Q2 (2023)

IF: 0.9 (2023)

[**P3**] A. Gilányi and L. N. To. Computer-assisted investigation of Levi-Civita type functional equations.

Manuscript, 2025.

[**P4**] A. Gilányi, R. Quintero, and L. N. To. Computer-assisted investigations of m-convex hulls of sets.

Manuscript, 2025.

List of other abstracts

[**A1**] L. N. To and A. Gilányi. Computer assisted investigation of Levi-Civita type functional equations. In report of Meeting: The Twenty-third Katowice-Debrecen Winter Seminar on Functional Equations and Inequalities Brenna (Poland), January 31 - February 3, 2024.

Ann. Math. Silesianae. 38 (2), 394, 2024.

ISSN: 0860-2107.

DOI: http://dx.doi.org/10.2478/amsil-2024-0010

[**A2**] L. N. To and A. Gilányi. On computer assisted investigations of m-convex hulls of sets. In: Report of Meeting: The 60th International Symposium on Functional Equations, Hotel Rewita, Kościelisko (Poland), June 9-15, 2024.

Aequationes. Math. 98 (6), 1703, 2024.

ISSN: 0001-9054.

DOI: http://dx.doi.org/10.1007/s00010-024-01126-5

[**A3**] L. N. To and A. Gilányi. On computer assisted visualization of m-convex hulls of sets. In: Report of meeting: The 59th International Symposium on Functional Equations Hotel Aurum, Hajdúszoboszló (Hungary), June 18-25, 2023.

Aequationes. Math. 97, 1274, 2023.

ISSN: 0001-9054.

DOI: http://dx.doi.org/10.1007/s00010-023-01007-3

[**A4**] L. N. To and A. Gilányi. Computer assisted investigation of the alienness of linear functional equations. In: Report of Meeting: 19th International Conference on Functional Equations and Inequalities, Bedlewo, Poland, September 11-18, 2021.

Ann. Univ. Paedag. Cracoviensis. Studia Math. 20 (1), 166-167, 2021.

ISSN: 2081-545X.

DOI: http://dx.doi.org/10.2478/aupcsm-2021-0009

List of talks related to the topic of the dissertation

[**T1**] *PhD Qualification at the End of the First Year*, Institute of Mathematics, University of Debrecen, June 18, 2021.

[**T2**] *Computer assisted investigation of the alienness of linear functional equations*, The 19th International Conference on Functional Equations and Inequalities, Bedlewo, Poland, September 11–18, 2021.

[**T3**] *Complex Exam*, Institute of Mathematics, University of Debrecen, June 29, 2022.

[**T4**] *PhD Qualification at the End of the Third Year*, Institute of Mathematics, University of Debrecen, June 6, 2023.

[**T5**] *On computer assisted visualization of m-convex hulls of sets*, The 59th International Symposium on Functional Equations, Hajdúszoboszló, Hungary, June 18–25, 2023.

[**T6**] *Computer assisted investigation of Levi-Civita type functional equations*, The 23rd Katowice–Debrecen Winter Seminar on Functional Equations and Inequalities, Brenna, Poland, January 31–February 3, 2024.

[**T7**] *On computer assisted investigations of m-convex hulls of sets*, The 60th International Symposium on Functional Equations, Kościelisko, Poland, June 9–15, 2024.

[**T8**] *Computer-assisted investigation of alienness of linear functional equations*, Dr. Varecza Árpád emlékkonferencia, University of Nyíregyháza, November 14–15, 2024.

[**T9**] *Computer-assisted investigation of types of functional equations*, The 61st International Symposium on Functional Equations, Cluj-Napoca, Romania, June, 15–21, 2025.

Appendix B

The package FunctionalEquations

```
1  FunctionalEquations := module()
2    option package;
3    export LFESolve, AlienationCheck,
        LeviCivitafesolve, FETypes;
4
5    LFESolve := module()
6      option package;
7      local error_management, conv, col, solver,
          pqRegular, checker, olddegs,

          selectFunctionsInEquation,
          getFunctionDegrees, METHOD, MAXDEG,
          OUTPUT, ERROR_STYLE, WarningList;
8      export lfesolve;
9      METHOD := 'strict';
10     error_management:=proc()
11       local case, arg;
12       if ERROR_STYLE = 'errorNo' then
13         return [args];
14       else
15         case:=args[1];
16         if nargs >1 then
17           arg := args[2];
18         end if;
19         if case = "E01" then
20           error "Invalid column number";
```

```
21          end if;
22          if case = "E02" then
23            error "[02] invalid input: %1", arg;
24          end if;
25          if case = "E03" then
26            error "[03] invalid input: %1", arg;
27          end if;
28          if case = "E05" then
29            error "[05] invalid input: %1", arg;
30          end if;
31          if case = "E06" then
32            error "[06] invalid argument: %1 in
                %2", arg[1], arg[2];
33          end if;
34          if case = "E07" then
35            error "[07] invalid input: %1", arg;
36          end if;
37          if case = "E08" then
38            error "[08] invalid input: %1", arg;
39          end if;
40          if case = "E09" then
41            ERROR("invalid argument: %1", arg);
42          end if;
43          if case = "E10" then
44            ERROR("invalid input", arg);
45          end if;
46          if case = "E11" then
47            ERROR("invalid input", arg);
48          end if;
49          if case = "W01" then
50            WARNING("for functional equations of
                this type the program may not
                provide all solutions.");
51            return [args];
52          end if;
53          if case = "W02" then
54            WARNING("missing function: %1", arg);
55            return [args];
56          end if;
57          if case = "W03" then
58            WARNING("lfesolve couldn't solve the
                given equations. You should try
```

```
                again using the 'permissive'
                option");
59            return [args];
60          end if;
61          if case = "W04" then
62            WARNING("lfesolve couldn't solve the
                given equations.");
63            return [args];
64          end if;
65        end if;
66      end proc;
67
68      col := proc(Mat::Matrix, c::posint)::set;
69        local i;
70        if op(1, Mat)[2] < c then
71          return error_management("E01");
72        end if;
73        return {seq(Mat[i, c], i = 1 .. op(1,
            Mat)[1])};
74      end proc;
75
76      conv := proc(parameqs, paramfcts)
77        local eqs, fction, i, t;
78        eqs, fction := parameqs, paramfcts;
79        if not type(eqs, list) then
80          eqs := [eqs];
81        end if;
82        if not type(fction, list) then
83          fction := [fction];
84        end if;
85        for i to nops(eqs) do
86          if type(eqs[i], `=`) then
87            eqs := subsop(i = lhs(eqs[i]) -
                  rhs(eqs[i]), eqs);
88          end if;
89        end do;
90        for i in fction do
91          if not has(eqs, i) then
92            if member(i, fction, 't') then
93              fction := subsop(t = NULL, fction);
94            end if;
95          end if;
```

```
96        end do;
97        return eqs, fction;
98      end proc;
99
100     pqRegular := proc(eqs, P, Q)::boolean;
101       local i, j, k, reg, irregeqs;
102       reg := true;
103       irregeqs := [];
104       for i to nops(eqs) do
105         for j to nops(eqs[i]) do
106           for k to j do
107             if not (j = k) and P[i, j]*Q[i, k]
                    = P[i, k]*Q[i, j] then
108               irregeqs := [eqs[i],
                      op(irregeqs)];
109               reg := false;
110             end if;
111           end do;
112         end do;
113       end do;
114       if not reg then
115         WarningList := [op(WarningList),
              error_management("W01")];
116       end if;
117       return reg, irregeqs;
118     end proc;
119
120     olddegs := proc(eqs, fncs, reg)
121       local i, j, k, degs;
122       degs := [];
123       for i in fncs do
124         k := [];
125         for j in eqs do
126           if has(j, i) then
127             k := [op(k), nops(j) - 2];
128           end if;
129         end do;
130         degs := [op(degs), min(op(k))];
131       end do;
132       return degs;
133     end proc;
134
```

```
135     selectFunctionsInEquation := proc(eq, funs)
136       local i, slok; slok := {};
137       for i in funs do
138         if has(eq, i) then
139           slok := slok union {i};
140         end if;
141       end do;
142       return slok;
143     end proc;
144
145     getFunctionDegrees := proc(eqs::list,
            irregeqs::list, funs::list)
146       local i, e, f, regeqs, degs,
             funsInRegEqs, funsInIrregEqs,
             funsWithDeg, funsWithoutDeg, kal, torg;
147       regeqs := convert(convert(eqs, set) minus
             convert(irregeqs, set), list);
148       funsInRegEqs := {};
149       funsInIrregEqs := {};
150       funsWithDeg := {};
151       for f in funs do
152         degs[f] := infinity;
153         for e in regeqs do
154           if has(e, f) then
155             funsInRegEqs := funsInRegEqs union
                  {f};
156             degs[f] := min(degs[f], nops(e) -
                  2);
157             funsWithDeg := funsWithDeg union
                  {f};
158           end if;
159         end do;
160         for e in irregeqs do
161           if has(e, f) then
162             funsInIrregEqs := funsInIrregEqs
                  union {f};
163           end if;
164         end do;
165       end do;
166       while nops(funsWithDeg) < nops(funs) do
167         funsWithoutDeg := convert(funs, set)
              minus funsWithDeg;
```

```
168         for e in irregeqs do
169           kal := selectFunctionsInEquation(e,
                 funsWithoutDeg);
170           if nops(kal) = 1 then
171             torg :=
                   selectFunctionsInEquation(e,
                   funsWithDeg);
172             if 0 < nops(torg) then
173               torg := convert(torg, list);
174               for f in kal do
175                 degs[f] := min(degs[f],
                     max(seq(degs[torg[i]], i = 1
                     .. nops(torg))));
176                 funsWithDeg := funsWithDeg
                     union {f};
177               end do;
178             end if;
179           end if;
180         end do;
181         if nops(funs) = nops(funsWithDeg) +
             nops(funsWithoutDeg) then
182           for e in irregeqs do
183             kal := selectFunctionsInEquation(e,
                   funsWithoutDeg);
184             if 0 < nops(kal) then
185               for f in kal do
186                 degs[f] := max(nops(e) - 2,
                     op(map2(op, 2, select(x ->
                     op(1, x) in
                     selectFunctionsInEquation(e,
                     funsWithDeg) and op(2, x) <>
                     infinity, op(op(1,
                     degs))))));
187                 funsWithDeg := funsWithDeg
                     union {f};
188               end do;
189             end if;
190           end do;
191         end if;
192       end do;
193       return [seq(degs[funs[f]], f = 1 ..
            nops(funs))];
```

```
194     end proc;
195
196     solver := proc (eqs, fction)
197       local P, Q, numOfEqs, maxNumOfTerms,
             numOfTerms, numOfFuncs, i, j, k, t, z,
             arg, listOfEqs, listOfFuncs,
             listMatFv, listMatMon, listOfirregEqs,
              MM, MF, kif, sgn, fvnev, fvk, MatMeg,
             listMatMeg, result, setMeg, reszmeg,
             vegsomegoldas, pqReg, listOfDeg;
198       listOfEqs, listOfFuncs := conv(eqs,
             fction);
199       numOfEqs := nops(listOfEqs);
200       maxNumOfTerms := max(op(map(nops,
             listOfEqs)));
201       numOfFuncs := nops(listOfFuncs);
202       P := Matrix(numOfEqs, maxNumOfTerms);
203       Q := Matrix(numOfEqs, maxNumOfTerms);
204       listMatMon := [];
205       listMatFv := [];
206       for i to numOfEqs do
207         numOfTerms := nops(listOfEqs[i]);
208         MM := Matrix(maxNumOfTerms,
                maxNumOfTerms-1);
209         MF := Matrix(maxNumOfTerms,
                maxNumOfTerms-1);
210          for j to numOfTerms do
211            kif := listOfEqs[i];
212            sgn := 1;
213            if type(kif, `+`) then
214              kif := op(j, kif);
215            elif not type(kif, `*`) and not
                  type(kif, function) then
216              #`error "[07] invalid input: %1",
                    kif;`
217              return error_management("E07", kif);
218            end if;
219            if type(kif, numeric) or type(kif,
                  symbol) then
220              #`error "[08] invalid input: %1",
                    kif;`
221              return error_management("E08", kif);
```

```
222             end if;
223             if type(kif, '*') then
224               if type(op(1, kif), numeric) then
225                 sgn := op(1, kif);
226                 kif := subsop(1=1, kif);
227               end if;
228               if not type(kif, function) then
229                 if type(op(1, kif), function) and
                       type(op(2, kif), function) then
230                   #'error "[02] invalid input:
                         %1", kif; '
231                   return error_management("E02",
                         kif);
232                 else
233                   #'error "[03] invalid input:
                         %1", kif; '
234                   return error_management("E03",
                         kif);
235                 end if;
236               end if;
237             end if;
238             fvnev := op(0, kif);
239             arg := op(kif);
240             do
241               if member(fvnev, listOfFuncs, 't')
                    then
242                 for k to maxNumOfTerms-1 do
243                   MM[j, k] := sgn*M(t-1, k-1);
244                   MF[j, k] := M(t-1, k-1);
245                 end do;
246               break;
247               else
248                 #WARNING("missing function: %1",
                       fvnev);
249                 WarningList:=[op(WarningList),
                       error_management("W02",
                       fvnev)];
250                 listOfFuncs := [op(listOfFuncs),
                       fvnev];
251                 numOfFuncs := nops(listOfFuncs);
252               end if;
253             end do;
```

```
254           if not type(arg, polynom(rational,
                  [x, y])) then
255             #'error "[05] invalid input: %1",
                    arg;'
256             return error_management("E05", arg);
257           end if;
258           if degree(arg, [x, y]) <> 1  then
259             #'error "[06] invalid argument: %1
                    in %2", arg, fvnev'(arg);
260             return error_management("E06",
                    [arg, fvnev(arg)]);
261           end if;
262           P[i, j] := coeff(arg, x);
263           Q[i, j] := coeff(arg, y);
264         end do;
265         listMatMon := [op(listMatMon), MM];
266         listMatFv := [op(listMatFv), MF];
267       end do;
268       pqReg, listOfirregEqs := pqRegular(
            listOfEqs, P, Q);
269       if METHOD='strict' then
270         listOfDeg := olddegs(listOfEqs,
              listOfFuncs, pqReg);
271       else
272         listOfDeg :=
              getFunctionDegrees(listOfEqs,
              listOfirregEqs, listOfFuncs);
273       end if;
274       listMatMeg := [];
275       for i to numOfEqs do
276         MatMeg := Matrix(maxNumOfTerms-1,
              maxNumOfTerms-1);
277         for k from 0 to maxNumOfTerms-2 do
278           for j from 0 to k do
279             MatMeg[j+1, k+1] := add(P[i,
                    z]^j*Q[i,
                    z]^(k-j)*listMatMon[i][z, k+1],
                    z = 1 .. maxNumOfTerms) = 0;
280           end do;
281         end do;
282         listMatMeg := [op(listMatMeg), MatMeg];
283       end do;
```

```
284       setMeg := {};
285       for j to maxNumOfTerms-1 do
286         reszmeg :=
              solve(`union`(`$`('col(listMatMeg[i],
              j)', 'i' = 1 .. numOfEqs)),
              `union`(`$`('col(listMatFv[i], j)',
              'i' = 1 .. numOfEqs)) minus {0} );
287         for k in reszmeg do
288           if not op(1, k) = op(2, k) then
289             setMeg := `union`(setMeg, {k});
290           end if;
291         end do;
292       end do;
293       fvk := [];
294       for i to numOfFuncs do
295         fvk := [op(fvk), listOfFuncs[i] =
              sum(M(i-1, z), z = 0 ..
              listOfDeg[i])];
296       end do;
297       kif := `$`('op(1, op(z, setMeg))', 'z' =
            1 .. nops(setMeg));
298       vegsomegoldas :=
            solve(`union`(convert(fvk, set),
            setMeg), `union`(convert(listOfFuncs,
            set), {kif}));
299       result := `minus`(vegsomegoldas, setMeg);
300       return [listOfFuncs, listOfDeg, setMeg,
            result, pqReg, checker(listOfEqs,
            result) ];
301     end proc;
302
303     checker := proc(eq, sol)
304       local MM, neweq, res, i;
305       neweq := subs(sol, eq);
306       MM := (j, k) -> P -> add(binomial(k,
            i)*coeff(P, x)^i*coeff(P, y)^(k -
            i)*A[j](x $ i, y $ (k - i)), i = 0 ..
            k);
307       neweq := eval(subs(M = MM, neweq));
308       res := true;
309       if not type(neweq, list) then
310         neweq := [neweq];
```

```
311       end if;
312       for i in neweq do
313         if i <> 0 then
314           res := false;
315           break;
316         end if;
317       end do;
318       return res;
319     end proc;
320
321     lfesolve := proc()
322       local solution, i;
323       ERROR_STYLE:='errorYes';
324       if nargs=2 then
325         OUTPUT:='brief';
326         METHOD:='strict';
327         MAXDEG:=infinity;
328         ERROR_STYLE:='errorYes';
329       elif nargs = 3 then
330         if args[3] in {'brief', 'verbose'} then
331           OUTPUT:=args[3];
332           METHOD:='strict';
333           MAXDEG:=infinity;
334           ERROR_STYLE:='errorYes';
335         elif args[3] in {'strict',
              'permissive'} then
336           OUTPUT:='brief';
337           METHOD:=args[3];
338           MAXDEG:=infinity;
339           ERROR_STYLE:='errorYes';
340         elif type(args[3], posint) then
341           OUTPUT:='brief';
342           METHOD:='strict';
343           MAXDEG:=args[3];
344           ERROR_STYLE:='errorYes';
345         elif args[3] in {'errorYes', 'errorNo'}
              then
346           OUTPUT:='brief';
347           METHOD:='strict';
348           MAXDEG:=infinity;
349           ERROR_STYLE:=args[3];
350         else
```

```
351           #ERROR("invalid argument: %1",
                  args[3]);
352           return error_management("E09",
                  args[3]);
353         end if;
354       elif nargs = 4 then
355         if (args[3] in {'brief', 'verbose'})
              and (args[4] in {'strict',
              'permissive'}) then
356           OUTPUT:=args[3];
357           METHOD:=args[4];
358           MAXDEG:=infinity;
359           ERROR_STYLE:='errorYes';
360         elif (args[3] in {'strict',
              'permissive'}) and type(args[4],
              posint) then
361           OUTPUT:='brief';
362           METHOD:=args[3];
363           MAXDEG:=args[4];
364           ERROR_STYLE:='errorYes';
365         elif (args[3] in {'brief', 'verbose'})
              and (args[4] in {'errorYes',
              'errorNo'}) then
366           OUTPUT:=args[3];
367           METHOD:='strict';
368           MAXDEG:=infinity;
369           ERROR_STYLE:=args[4];
370         elif (args[3] in {'strict',
              'permissive'}) and (args[4] in
              {'errorYes', 'errorNo'}) then
371           OUTPUT:='brief';
372           METHOD:=args[3];
373           MAXDEG:=infinity;
374           ERROR_STYLE:=args[4];
375         else
376           #ERROR("invalid input", args[3..-1]);
377           return error_management("E10",
                  [args[3..-1]]);
378         end if;
379       elif nargs = 5 then
380         if (args[3] in {'brief', 'verbose'})
              and (args[4] in {'strict',
```

```
            'permissive'}) and type(args[5],
            posint) then
381           OUTPUT:=args[3];
382           METHOD:=args[4];
383           MAXDEG:=args[5];
384           ERROR_STYLE:='errorYes';
385         elif (args[3] in {'brief', 'verbose'})
              and (args[4] in {'strict',
              'permissive'}) and (args[5] in
              {'errorYes', 'errorNo'}) then
386           OUTPUT:=args[3];
387           METHOD:=args[4];
388           MAXDEG:=infinity;
389           ERROR_STYLE:=args[5];
390         elif (args[3] in {'brief', 'verbose'})
              and type(args[4], posint) and
              (args[5] in {'errorYes', 'errorNo'})
              then
391           OUTPUT:=args[3];
392           METHOD:='strict';
393           MAXDEG:=args[4];
394           ERROR_STYLE:=args[5];
395         elif (args[3] in {'strict',
              'permissive'}) and type(args[4],
              posint) and (args[5] in {'errorYes',
              'errorNo'}) then
396           OUTPUT:='brief';
397           METHOD:=args[3];
398           MAXDEG:=args[4];
399           ERROR_STYLE:=args[5];
400         else
401           #ERROR("invalid input", args[3..-1]);
402           return error_management("E10",
                [args[3..-1]]);
403         end if;
404       elif nargs = 6 then
405         if (args[3] in {'brief', 'verbose'})
              and (args[4] in {'strict',
              'permissive'}) and type(args[5],
              posint) and (args[6] in {'errorYes',
              'errorNo'}) then
406           OUTPUT:=args[3];
```

```
407           METHOD:=args[4];
408           MAXDEG:=args[5];
409           ERROR_STYLE:=args[6];
410         else
411           #ERROR("invalid input", args);
412           return error_management("E11", args);
413         end if;
414       else
415         #ERROR("invalid input", args);
416         return error_management("E11", args);
417       end if;
418       WarningList:=[];
419       solution := solver( args[1], args[2]);
420       #`error_and_errorNo `
421       if nops(solution)=2 then
422         return solution;
423       end if;
424       if not solution[6] and METHOD='strict'
            then
425         #WARNING("lfesolve couldn't solve the
              given equations. You should try
              again using the 'permissive'
              option");
426         WarningList:=[op(WarningList),
              error_management("W03")];
427         if ERROR_STYLE= 'errorNo' then
428           return WarningList;
429         end if;
430         return {};
431       elif not solution[6] and
            METHOD='permissive' then
432         #WARNING("lfesolve couldn't solve the
              given equations.");
433         WarningList:=[op(WarningList),
              error_management("W04")];
434         if ERROR_STYLE= 'errorNo' then
435           return WarningList;
436         end if;
437         return {};
438       end if;
439       if OUTPUT='verbose' then
440         if solution[5] then
```

```
441           printf("The form of the solutions
                 is\n");
442         else
443           printf("The form of the polynomial
                 solutions of degree at most %d
                 is\n", max( op( solution[2] )));
444         end if;
445         for i to nops( solution[1] ) do
446           print(solution[1][i] = sum(M(i-1, z),
                 z = 0 .. solution[2][i] ) );
447         end do;
448         printf("where M(., k) are monomial
                 functions of degree k");
449         if nops(solution[3])>0 then
450           printf(", for which\n");
451           for i in solution[3] do
452             print(i);
453           end do;
454         else
455           printf(".\n");
456         end if;
457         printf("Thus, \n");
458         for i in solution[4] do
459           print(i);
460         end do;
461       end if;
462       if (OUTPUT='brief') and (ERROR_STYLE=
             'errorNo') then
463         return [solution[4], WarningList];
464       end if;
465       return solution[4];
466     end proc;
467   end module;
468
469   AlienationCheck := module()
470     option package;
471     local CHECK, OUTPUT, ErrorWarn_STYLE,
           WarningList, Remark, error_management,
           normalize_listOfFunctions,
           normalize_input_alien, test;
472     export isalien;
473
```

```
474     with(ListTools);
475     Remark := 0;
476
477     error_management := proc()
478       local case, arg;
479       if ErrorWarn_STYLE = 'ewNo' then
480         return [args];
481       else case := args[1];
482         if 1 < nargs then
483           arg := args[2];
484         end if;
485         if case = "E01" then
486           error "Invalid column number";
487         end if;
488         if case = "E02" then
489           error "[02] invalid input: %1", arg;
490         end if;
491         if case = "E03" then
492           error "[03] invalid input: %1", arg;
493         end if;
494         if case = "E05" then
495           error "[05] invalid input: %1", arg;
496         end if;
497         if case = "E06" then
498           error "[06] invalid argument: %1 in
                %2", arg[1], arg[2];
499         end if;
500         if case = "E07" then
501           error "[07] invalid input: %1", arg;
502         end if;
503         if case = "E08" then
504           error "[08] invalid input: %1", arg;
505         end if;
506         if case = "E09" then
507           ERROR("invalid argument: %1", arg);
508         end if;
509         if case = "E10" then
510           ERROR("invalid input", arg);
511         end if;
512         if case = "E11" then
513           ERROR("invalid input", arg);
514         end if;
```

```
515         if case = "E12" then
516           error "[12] invalid input in list of
                unknown functions";
517         end if;
518         if case = "E13" then
519           error "[13] invalid input: %1", arg;
520         end if;
521         if case = "E14" then
522           error "[14] invalid input: %1", arg;
523         end if;
524         if case = "E15" then
525           error "[15] invalid input: %1", arg;
526         end if;
527         if case = "E16" then
528           error "[16] invalid input: %1", arg;
529         end if;
530         if case = "E17" then
531           error "[17] invalid input in case
                checking strongly alien: %1", arg;
532         end if;
533         if case = "E18" then
534           ERROR("invalid argument in the input:
                %1", arg);
535         end if;
536         if case = "E19" then
537           ERROR("invalid arguments in the
                input", arg);
538         end if;
539         if case = "E20" then
540           ERROR("invalid arguments in the
                input", arg);
541         end if;
542         if case = "E21" then
543           ERROR("invalid arguments in the
                input", arg);
544         end if;
545       end if;
546     end proc:
547
548     normalize_listOfFunctions :=
          proc(lequations, lfunctions)
549       local i, n, lfunctions_new;
```

```
550       if nops(lequations) <> nops(lfunctions)
             then
551         return error_management("E13", args[2]);
552       end if;
553       if type(lfunctions[1], symbol) then
554         for i to nops(lfunctions) do
555           if not(type(lfunctions[i], symbol))
                 then
556             return error_management("E14",
                   args[2]);
557           end if;
558         end do;
559         lfunctions_new :=
               [LengthSplit(lfunctions, 1)];
560       elif type(lfunctions[1], list) then
561         for i to nops(lfunctions) do
562           if not(type(lfunctions[i], list)) or
                 nops(lfunctions[i]) <>
                 nops(lfunctions[1]) then
563             return error_management("E15",
                   args[2]);
564           end if;
565         end do;
566         lfunctions_new := lfunctions;
567       else
568         return error_management("E16", args[2]);
569       end if;
570       return lfunctions_new;
571     end proc:
572
573     normalize_input_alien := proc(lequations,
          lfunctions)
574       local numEqs, lequa_new, i, x,
            lfuncs_new, numvar, j;
575       numEqs := nops(lequations);
576       numvar := nops(lfunctions[1]);
577       lequa_new := [lequations[1]];
578       for i from 2 to numEqs do
579         x := subs({seq(lfunctions[i][j] =
               lfunctions[1][j], j = 1 .. numvar)},
               lequations[i]);
580         lequa_new := [op(lequa_new), x];
```

```
581       end do;
582       lfuncs_new := lfunctions[1];
583       return [lequa_new, lfuncs_new];
584     end proc:
585
586     test := proc(equations, functions)
587       local lequations, lfunctions,
             lfunctions_new, new_input, listOfEqs,
             listOfFuncs, numOfEqs, E, i, A, B, b;
588       #Remark := 0;
589       lequations := convert(equations, list);
590       lfunctions := convert(functions, list);
591       lfunctions_new :=
             normalize_listOfFunctions(lequations,
             lfunctions);
592       if CHECK = 'alien' then
593         new_input :=
               normalize_input_alien(lequations,
               lfunctions_new);
594       else
595         if nops(Flatten(lfunctions_new)) <>
               numelems(convert(Flatten(lfunctions_new),
               set)) then
596           return error_management("E17",
                 lfunctions_new);
597         else
598           new_input := [lequations,
                 lfunctions_new];
599         end if;
600       end if;
601       listOfEqs := convert(new_input[1], list);
602       listOfFuncs :=
             Flatten(convert(new_input[2], list));
603       numOfEqs := nops(listOfEqs);
604       E := 0;
605       for i to numOfEqs do
606         E := E + listOfEqs[i];
607       end do;
608       A := LFESolve:-lfesolve(listOfEqs,
             listOfFuncs, 'errorNo');
609       B := LFESolve:-lfesolve([E], listOfFuncs,
             'errorNo');
```

```
610       if type(A[1], string) then
611         return error_management(A[1], A[2]);
612       end if;
613       if type(B[1], string) then
614         if B = ["E07", 0] then
615           b := cat(arbitrary, listOfFuncs);
616           return [false, listOfEqs, [A[1], b]];
617         end if;
618         return error_management(B[1], B[2]);
619       end if;
620       if evalb("W02" in Flatten(B)) then
621         return error_management("E12");
622       end if;
623       Remark := 0;
624       if nops(A[2]) <> 0 then
625         WarningList := [op(WarningList), A[2]];
626       end if;
627       if nops(B[2]) <> 0 then
628         WarningList := [op(WarningList), B[2]];
629       end if;
630       if A[1] = B[1] then
631         if evalb("W01" in Flatten(B)) then
632           Remark := 1;
633         end if;
634         return [true, listOfEqs, [A[1]]];
635       else
636         if evalb("W01" in Flatten(B)) then
637           if evalb("W01" in Flatten(A)) then
638             Remark := -2;
639           else
640             Remark := -1;
641           end if;
642         end if;
643         return [false, listOfEqs, [A[1], B[1]]];
644       end if;
645       end proc:
646
647      isalien := proc()
648       local solution, i;
649       ErrorWarn_STYLE :='ewYes';
650       if nargs=2 then
651         CHECK:='alien';
```

```
652           OUTPUT:='brief';
653           ErrorWarn_STYLE :='ewYes';
654         elif nargs = 3 then
655           if args[3] in {'alien', 'strongalien'}
                then
656             CHECK:=args[3];
657             OUTPUT:='brief';
658             ErrorWarn_STYLE :='ewYes';
659           elif args[3] in {'full', 'verbose',
                'brief'} then
660             CHECK:='alien';
661             OUTPUT:=args[3];
662             ErrorWarn_STYLE :='ewYes';
663           elif args[3] in {'ewYes', 'ewNo'} then
664             CHECK:='alien';
665             OUTPUT:='brief';
666             ErrorWarn_STYLE :=args[3];
667           else
668             return error_management("E18",
                    args[3]);
669           end if;
670         elif nargs = 4 then
671           if (args[3] in {'alien',
                'strongalien'}) and (args[4] in
                {'full', 'verbose', 'brief'}) then
672             CHECK:=args[3];
673             OUTPUT:=args[4];
674             ErrorWarn_STYLE :='ewYes';
675           elif (args[3] in {'alien',
                'strongalien'}) and (args[4] in
                {'ewYes', 'ewNo'}) then
676             CHECK:=args[3];
677             OUTPUT:='brief';
678             ErrorWarn_STYLE :=args[4];
679           elif (args[3] in {'full', 'verbose',
                'brief'}) and (args[4] in {'ewYes',
                'ewNo'}) then
680             CHECK:='alien';
681             OUTPUT:=args[3];
682             ErrorWarn_STYLE :=args[4];
683           else
684             return error_management("E19",
```

```
            [args[3..-1]]);
685       end if;
686     elif nargs = 5 then
687       if (args[3] in {'alien',
            'strongalien'}) and (args[4] in
            {'full', 'verbose', 'brief'}) and
            (args[5] in  {'ewYes', 'ewNo'})then
688         CHECK:=args[3];
689         OUTPUT:=args[4];
690         ErrorWarn_STYLE :=args[5];
691       else
692         return error_management("E20",
              args[[3..-1]]);
693       end if;
694     else
695       return error_management("E21", args);
696     end if;
697
698     WarningList:=[];
699     solution := test( args[1], args[2]);
700
701     if type(solution[1], string)  then
702       return solution;
703     end if;
704     if Remark =1 then
705       if  ErrorWarn_STYLE = 'ewYes' then
706         WARNING("for functional equations of
              this type, isalien may not be able
              to decide about their alienness or
              strong alienness.");
707       end if;
708       if OUTPUT = 'brief'then
709         printf("True (If polynomial solutions
              are considered only)");
710       else #'output = verbose or full'
711         printf("The functional equations\n");
712         for i to nops(solution[2]) do
713           printf("E%d:",
                i);print(solution[2][i]);
714         end do;
715         if CHECK = 'alien' then
716           printf("\nare alien if polynomial
```

```
                solutions are considered only.");
717           else
718             printf("\nare strongly alien if
                   polynomial solutions are
                   considered only.");
719           end if;
720           if OUTPUT='full' then
721             if nops(solution[2])=2 then
722               printf("\nIn details,\n");
723               printf("The polynomial solution
                     of the functional equation E1
                     + E2 = 0 and the polynomial
                     solution of the system [E1 =
                     0, E2 = 0]");
724               if nops(solution[3][1])=1 then
                     printf(" is");   else
                     printf(" are"); end if;
725               print(solution[3][1]);
726             elif nops(solution[2])=3 then
727               printf("\nIn details,\n");
728               printf("The polynomial solution
                     of the functional equation E1
                     + E2 + E3= 0 and the
                     polynomial solution of the
                     system [E1 = 0, E2 = 0, E3 =
                     0]");
729               if nops(solution[3][1])=1 then
730                 printf(" is");
731               else
732                 printf(" are");
733               end if;
734               print(solution[3][1]);
735             else
736               printf("\nIn details,\n");
737               if nops(solution[3][1])=1 then
738                 printf("The polynomial solution
                       of the functional equation
                       E1 + ... + E%d = 0 and the
                       polynomial solution of the
                       system [E1 = 0, ..., E%d =
                       0] is ", nops(solution[2]),
                       nops(solution[2]));
```

```
739               else
740                 printf("The polynomial solution
                        of the functional equation
                        E1 + ... + E%d = 0 and the
                        polynomial solution of the
                        system [E1 = 0, ..., E%d =
                        0] are", nops(solution[2]),
                        nops(solution[2])) ;
741               end if;
742               print(solution[3][1]);
743             end if;
744           end if;
745         end if;
746       elif Remark =-1 then
747         if OUTPUT = 'brief' then
              printf("False");
748         else #'output = verbose or full'
749           printf("The functional equations\n");
750           for i to nops(solution[2]) do
751             printf("E%d:", i);
752             print(solution[2][i]);
753           end do;
754           if CHECK = 'alien' then
755             printf("\nare not alien.");
756           else
757             printf("\nare not strongly alien.");
758           end if;
759           if OUTPUT='full' then
760             if nops(solution[2])=2 then
761               printf("\nIn details,\n");
762               printf("The polynomial solution
                      of the functional equation E1
                      + E2 = 0");
763               if nops(solution[3][2])=1 then
764                 printf(" is");
765               else
766                 printf(" are");
767               end if;
768               print(solution[3][2]);
769               printf("while the solution of the
                      system [E1 = 0, E2 = 0]");
770               if nops(solution[3][1])=1 then
```

```
771                 printf(" is");
772               else
773                 printf(" are");
774               end if;
775               print(solution[3][1]);
776             elif nops(solution[2])=3 then
777               printf("\nIn details,\n");
778               printf("The polynomial solution
                     of the functional equation E1
                     + E2 + E3= 0");
779               if nops(solution[3][2])=1 then
780                 printf(" is");
781               else
782                 printf(" are");
783               end if;
784               print(solution[3][2]);
785               printf("while the solution of the
                     system [E1 = 0, E2 = 0, E3 =
                     0]");
786               if nops(solution[3][1])=1 then
787                 printf(" is");
788               else
789                 printf(" are");
790                end if;
791               print(solution[3][1]);
792             else
793               printf("\nIn details,\n");
794               if nops(solution[3][2])=1 then
795                 printf("The polynomial solution
                       of the functional equation
                       E1+ ... + E%d = 0 is",
                       nops(solution[2]));
796               else
797                 printf("The polynomial solution
                       of the functional equation
                       E1+ ... + E%d = 0 are",
                       nops(solution[2]));
798               end if;
799               print(solution[3][2]);
800               if nops(solution[3][1])=1 then
801                 printf("while the solution of
                       the system [E1 = 0, ..., E%d
```

```
                    = 0] is ",
                    nops(solution[2]));
802               else
803                 printf("while the solution of
                    the system [E1 = 0, ..., E%d
                    = 0] are ",
                    nops(solution[2]));
804               end if;
805               print(solution[3][1]);
806             end if;
807           end if;
808         end if;
809       elif Remark =-2 then
810         if OUTPUT = 'brief' then
              printf("False");
811         else #'output = verbose or full'
812           printf("The functional equations\n");
813           for i to nops(solution[2]) do
814             printf("E%d:", i);
815             print(solution[2][i]);
816           end do;
817           if CHECK = 'alien' then
818             printf("\nare not alien.");
819           else
820             printf("\nare not strongly alien.");
821           end if;
822           if OUTPUT='full' then
823             if nops(solution[2])=2 then
824               printf("\nIn details,\n");
825               printf("The polynomial solution
                    of the functional equation E1
                    + E2 = 0");
826               if nops(solution[3][2])=1 then
827                 printf(" is");
828               else
829                 printf(" are");
830               end if;
831               print(solution[3][2]);
832               printf("while the polynomial
                    solution of the system [E1 =
                    0, E2 = 0]");
833               if nops(solution[3][1])=1 then
```

```
834                 printf(" is");
835               else
836                 printf(" are");
837               end if;
838               print(solution[3][1]);
839             elif nops(solution[2])=3 then
840               printf("\nIn details,\n");
841               printf("The polynomial solution
                     of the functional equation E1
                     + E2 + E3= 0");
842               if nops(solution[3][2])=1 then
843                 printf(" is");
844               else
845                 printf(" are");
846               end if;
847               print(solution[3][2]);
848               printf("while the polynomial
                     solution of the system [E1 =
                     0, E2 = 0, E3 = 0]");
849               if nops(solution[3][1])=1 then
850                 printf(" is");
851               else
852                 printf(" are");
853               end if;
854               print(solution[3][1]);
855             else
856               printf("\nIn details,\n");
857               if nops(solution[3][2])=1 then
858                 printf("The polynomial solution
                       of the functional equation
                       E1+ ... + E%d = 0 is ",
                       nops(solution[2]));
859               else
860                 printf("The polynomial solution
                       of the functional equation
                       E1+ ... + E%d = 0 are",
                       nops(solution[2]));
861               end if;
862               print(solution[3][2]);
863               if nops(solution[3][1])=1 then
864                 printf("while the polynomial
                       solution of the system [E1 =
```

```
                    0, ..., E%d = 0] is ",
                    nops(solution[2]));
865               else
866                 printf("while the polynomial
                    solution of the system [E1 =
                    0, ..., E%d = 0] are",
                    nops(solution[2]));
867               end if;
868               print(solution[3][1]);
869             end if;
870           end if;
871         end if;
872       else #remark=0                    `
873         if OUTPUT = 'brief' then
874           if solution[1] then
875             printf("True");
876           else
877             printf("False");
878           end if;
879         else #`output = verbose or full`
880           printf("The functional equations\n");
881           for i to nops(solution[2]) do
882             printf("E%d:", i);
883             print(solution[2][i]);
884           end do;
885           if solution[1] then
886             if CHECK = 'alien' then
887               printf("\nare alien");
888             else
889               printf("\nare strongly alien");
890             end if;
891             if OUTPUT='full' then
892               if nops(solution[2])=2 then
893                 printf("\nIn details,\n");
894                 printf("The solution of the
                    functional equation E1 + E2
                    = 0 and the solution of the
                    system [E1 = 0, E2 = 0]");
895                 if nops(solution[3][1])=1 then
                    printf(" is"); else printf("
                    are"); end if;
896                 print(solution[3][1]);
```

```
897             elif nops(solution[2])=3 then
898               printf("\nIn details,\n");
899               printf("The solution of the
                     functional equation E1 + E2
                     + E3 = 0 and the solution of
                     the system [E1 = 0, E2 = 0,
                     E3 = 0]");
900               if nops(solution[3][1])=1 then
                     printf(" is"); else printf("
                     are"); end if;
901               print(solution[3][1]);
902             else
903               printf("\nIn details,\n");
904               if nops(solution[3][1])=1 then
905                 printf("The solution of the
                       functional equation E1+
                       ... + E%d = 0 and the
                       solution of the system [E1
                       = 0, ..., E%d = 0] is ",
                       nops(solution[2]),
                       nops(solution[2]));
906               else
907                 printf("The solution of the
                       functional equation E1+
                       ... + E%d = 0 and the
                       solution of the system [E1
                       = 0, ..., E%d = 0] are",
                       nops(solution[2]),
                       nops(solution[2]));
908               end if;
909               print(solution[3][1]);
910             end if;
911           end if;
912         else #`solution[1] false`
913           if CHECK = 'alien' then
914             printf("\nare not alien");
915           else
916             printf("\nare not strongly
                    alien");
917           end if;
918           if OUTPUT='full' then
919             if nops(solution[2])=2 then
```

```
920                 printf("\nIn details,\n");
921                 printf("The solution of the
                        functional equation E1 + E2
                        = 0");
922                 if nops(solution[3][2])=1 then
923                   printf(" is");
924                 else
925                   printf(" are");
926                 end if;
927                 print(solution[3][2]);
928                 printf("while the solution of
                        the system [E1 = 0, E2 =
                        0]");
929                 if nops(solution[3][1])=1 then
930                   printf(" is");
931                 else
932                   printf(" are");
933                 end if;
934                 print(solution[3][1]);
935               elif nops(solution[2])=3 then
936                 printf("\nIn details,\n");
937                 printf("The solution of the
                        functional equation E1 + E2
                        + E3= 0");
938                 if nops(solution[3][2])=1 then
939                   printf(" is");
940                 else
941                   printf(" are");
942                 end if;
943                 print(solution[3][2]);
944                 printf("while the solution of
                        the system [E1 = 0, E2 = 0,
                        E3 = 0]");
945                 if nops(solution[3][1])=1 then
946                   printf(" is");
947                 else
948                   printf(" are");
949                 end if;
950                 print(solution[3][1]);
951               else
952                 printf("\nIn details,\n");
953                 if nops(solution[3][2])=1 then
```

```
954                     printf("The solution of the
                           functional equation E1+
                           ... + E%d = 0 is ",
                           nops(solution[2]));
955                   else
956                     printf("The solution of the
                           functional equation E1+
                           ... + E%d = 0 are",
                           nops(solution[2]));
957                   end if;
958                   print(solution[3][2]);
959                   if nops(solution[3][1])=1 then
960                     printf("while the solution of
                           the system [E1 = 0, ...,
                           E%d = 0] is ",
                           nops(solution[2]));
961                   else
962                     printf("while the solution of
                           the system [E1 = 0, ...,
                           E%d = 0] are",
                           nops(solution[2]));
963                   end if;
964                   print(solution[3][1]);
965                 end if;
966               end if;
967             end if;
968           end if;
969         end if;
970       end proc;
971     end module;
972
973     LeviCivitafesolve := module()
974       option package;
975       local Compute_M_j, Compute_M, Compute_N_j,
             Compute_N, pre_char_n,
976       list_N_Q, list_N_Rt, list_M,
             Coefficients_1case, LCfesolveGeneral,
977       Normalize_CoefficientMatrix,
             ListUnknownFunction, UnknownFunction_var,
978       Modified_CoefficientMatrix,
             GeneralSolutionForm, SimpleSolve;
979       export LCfesolve;
```

```
980
981     with(LinearAlgebra);
982     with(StringTools);
983     with(ListTools);
984     with(combinat,partition);
985
986     Compute_M_j := proc(N_Q, N_Rt, j, n_j,
            pre_num, pre_char)
987       local M, p, q, ind, L, pre_size, index_q,
              index_r;
988       pre_size := pre_num + pre_char;
989       M := Matrix(n_j);
990       if 3 <= n_j then
991         for p from 0 to n_j - 1 do
992           for q from 0 to n_j - 1 do
993             if p + q < n_j then
994               ind := p + q;
995               if ind <> 0 then
996                 index_q := N_Q[n_j - p +
                      pre_size, 1][3];
997                 index_r := N_Rt[1, n_j - q +
                      pre_size][3];
998                 if type(index_q, list) and
                      type(index_r, list) then
999                   L := Flatten([index_q,
                        index_r]);
1000                elif type(index_q, list) then
1001                  L := index_q;
1002                else
1003                  L := index_r;
1004                end if;
1005                M[n_j - p, n_j - q] := [(p +
                      q)!/(p!*q!), alpha, L];
1006              else
1007                M[n_j - p, n_j - q] := [(p +
                      q)!/(p!*q!), alpha, pre_num
                      + 1];
1008              end if;
1009            end if;
1010          end do;
1011        end do;
1012      else
```

```
1013        for p from 0 to n_j - 1 do
1014          for q from 0 to n_j - 1 do
1015            if p + q < n_j then
1016              M[n_j - p, n_j - q] := [(p +
                    q)!/(p!*q!), alpha, pre_num +
                    n_j - p - q];
1017            end if;
1018          end do;
1019        end do;
1020      end if;
1021      return M;
1022    end proc;
1023
1024    Compute_M := proc(N_Q, N_Rt, n_list)
1025      local k, M_list, j, n_j, M_j, M, pre_num,
            pre_char;
1026      k := nops(n_list); M_list := [];
1027      pre_num := 0;
1028      pre_char := 0;
1029      for j to k do
1030        n_j := n_list[j];
1031        M_j := Compute_M_j(N_Q, N_Rt, j, n_j,
              pre_num, pre_char);
1032        M_list := [op(M_list), M_j];
1033        if 3 <= n_j then
1034          pre_char := pre_char + n_j - 1;
1035          pre_num := pre_num + 1;
1036        else pre_num := pre_num + n_j;
1037        end if;
1038      end do;
1039      M := DiagonalMatrix(M_list);
1040      return M;
1041    end proc;
1042
1043    Compute_N_j := proc(n, j, n_j, var, pre_num)
1044      local N, p, q, ind, L, i;
1045      N := Matrix(n_j, n);
1046      if 3 <= n_j then
1047        for p to n_j do
1048          for q to n do
1049            ind := n_j - p;
1050            if ind <> 0 then
```

```
1051              L := [seq(convert(Char(i), name),
                      i = 97 + pre_char_n .. 96 +
                      pre_char_n + ind)];
1052              N[p, q] := [1, var[q], L];
1053            else N[p, q] := [1, var[q], pre_num
                   + 1];
1054            end if;
1055          end do;
1056        end do;
1057      else
1058        for p to n_j do
1059          for q to n do
1060            N[p, q] := [1, var[q], pre_num + p];
1061          end do;
1062        end do;
1063      end if;
1064      return N;
1065    end proc;
1066
1067    Compute_N := proc(n, n_list, var)
1068      local k, N_list, j, n_j, N_j, N, pre_num;
1069      k := nops(n_list);
1070      N_list := [];
1071      pre_num := 0;
1072      for j to k do
1073        n_j := n_list[j];
1074        N_j := Compute_N_j(n, j, n_j, var,
                 pre_num);
1075        N_list := [op(N_list), [N_j]];
1076        if 3 <= n_j then
1077          pre_char_n := pre_char_n + n_j - 1;
1078          pre_num := pre_num + 1;
1079        else
1080          pre_num := pre_num + n_j;
1081        end if;
1082      end do;
1083      N := Matrix(N_list);
1084      return N;
1085    end proc;
1086
1087    Coefficients_1case := proc(n, n_list)
1088      local N_Q, N_Rt, M;
```

```
1089        pre_char_n := 0;
1090        N_Q := Compute_N(n, n_list, beta);
1091        N_Rt :=
              LinearAlgebra[Transpose](Compute_N(n,
              n_list, gamma));
1092        M := Compute_M(N_Q, N_Rt, n_list);
1093        return [N_Q, N_Rt, M];
1094      end proc;
1095
1096      LCfesolveGeneral := proc(n)
1097        local n_decomp, i, i_case, rs;
1098        n_decomp := partition(n);
1099        rs := [];
1100        for i to nops(n_decomp) do
1101          i_case := Coefficients_1case(n,
                n_decomp[i]);
1102          rs := [op(rs), [n_decomp[i], i_case]];
1103        end do;
1104        return rs;
1105      end proc;
1106
1107      Normalize_CoefficientMatrix := proc(M, n)
1108        local i, j, coeff, var, L;
1109        for i to n do
1110          for j to n do
1111            if type(M[i, j], list) then
1112              coeff := M[i, j][1];
1113              var := M[i, j][2];
1114              L := M[i, j][3];
1115              M[i, j] := coeff*var[L];
1116            end if;
1117          end do;
1118        end do;
1119        return M;
1120      end proc;
1121
1122      ListUnknownFunction := proc(fe)
1123        local Rfe, Lfe, n, cX, X, Y, i, b, c, d,
              e, flag, t, lfv, nfe;
1124        if type(fe,'=') then
1125          nfe := lhs(fe) - rhs(fe);
1126        else nfe := fe;
```

```
1127        end if;
1128        flag := false;
1129        for i to nops(nfe) do
1130          t := op(i, nfe);
1131          if type(t, function) and op(t) = 'x +
                y' then
1132            lfv := op(0, t);
1133            Lfe := t;
1134            Rfe := Lfe - nfe;
1135            flag := true;
1136            break;
1137          elif type(-t, function) and op(-t) = 'x
                + y' then
1138            lfv := op(0, -t);
1139            Lfe := -t;
1140            Rfe := nfe + Lfe;
1141            flag := true;
1142            break;
1143          end if;
1144        end do;
1145        if flag = false then
1146          ERROR("Input missing f(x+y)");
1147        else
1148          nfe := Lfe = Rfe;
1149        end if;
1150        if type(Rfe, `+`) then
1151          n := nops(Rfe);
1152        elif type(Rfe, `*`) and nops(Rfe) = 2 then
1153          n := 1;
1154        end if;
1155        cX := [];
1156        X := [];
1157        Y := [];
1158        for i to n do
1159          b := Rfe;
1160          if type(b, `+`) then
1161            b := op(i, b);
1162          elif not (type(b, `*`) or type(b,
                function)) then
1163            error "invalid input: %1", b;
1164          end if;
1165          if type(b, `*`) then
```

```
1166            if nops(b) = 3 then
1167              c := op(1, b);
1168              d := op(2, b);
1169              e := op(3, b);
1170              if type(c, integer) and type(d,
                     function) and type(e, function)
                     then
1171                cX := [op(cX), c];
1172                if op(d) = x and op(e) = y then
1173                  X := [op(X), op(0, d)];
1174                  Y := [op(Y), op(0, e)];
1175                elif op(d) = y and op(e) = x then
1176                  X := [op(X), op(0, e)];
1177                  Y := [op(Y), op(0, d)];
1178                else
1179                  error "invalid input: %1", b;
1180                end if;
1181              else
1182                error "invalid input: %1", b;
1183              end if;
1184            elif nops(b) = 2 then
1185              c := op(1, b);
1186              d := op(2, b);
1187              if type(c, function) and type(d,
                     function) then
1188                cX := [op(cX), 1];
1189                if op(c) = x and op(d) = y then
1190                  X := [op(X), op(0, c)];
1191                  Y := [op(Y), op(0, d)];
1192                elif op(c) = y and op(d) = x then
1193                  X := [op(X), op(0, d)];
1194                  Y := [op(Y), op(0, c)];
1195                else error "invalid input: %1", b;
1196                end if;
1197              elif type(c, integer) and type(d,
                     function) then
1198                cX := [op(cX), c];
1199                if op(d) = y then X := [op(X), 1];
1200                  Y := [op(Y), op(0, d)];
1201                elif op(d) = x then
1202                  X := [op(X), op(0, d)];
1203                  Y := [op(Y), 1];
```

```
1204              else
1205                error "invalid input: %1", b;
1206              end if;
1207            else
1208              error "invalid input: %1", b;
1209            end if;
1210          end if;
1211        else
1212          cX := [op(cX), 1];
1213          if op(b) = x then
1214            X := [op(X), op(0, b)];
1215            Y := [op(Y), 1];
1216          elif op(b) = y then
1217            X := [op(X), 1];
1218            Y := [op(Y), op(0, b)];
1219          else
1220            error "invalid input: %1", b;
1221          end if;
1222        end if;
1223      end do;
1224      return [cX, X, Y];
1225    end proc;
1226
1227    UnknownFunction_var := proc(X, Y)
1228      local A, var_beta, var_gamma, i, u, v,
           full_var, var;
1229      A := [op(X), op(Y)];
1230      A := MakeUnique(A);
1231      A := subs(f = NULL, A);
1232      A := subs(1 = NULL, A);
1233      var_beta := [];
1234      var_gamma := [];
1235      for i to nops(A) do
1236        u := Search(A[i], X);
1237        v := Search(A[i], Y);
1238        if v = 0 or u <> 0 and v <> 0 and u <=
             v then
1239          var_beta := [op(var_beta), A[i]];
1240        else var_gamma := [op(var_gamma), A[i]];
1241        end if;
1242      end do;
1243      full_var := [[f, alpha]];
```

```
1244        if nops(var_beta) = 1 then
1245          full_var := [op(full_var),
                [var_beta[1], beta]];
1246        else
1247          for i to nops(var_beta) do
1248            var := beta[i];
1249            full_var := [op(full_var),
                  [var_beta[i], var]];
1250          end do;
1251        end if;
1252        if nops(var_gamma) = 1 then
1253          full_var := [op(full_var),
                [var_gamma[1], gamma]];
1254        else
1255          for i to nops(var_gamma) do
1256            var := gamma[i];
1257            full_var := [op(full_var),
                  [var_gamma[i], var]];
1258          end do;
1259        end if;
1260        return full_var;
1261      end proc;
1262
1263      Modified_CoefficientMatrix := proc(cX, X,
           Y, full_var, Q, R)
1264        local n, i, var, position, new_var, j,
             new_full_var;
1265        n := nops(X);
1266        new_full_var := Flatten(full_var);
1267        for i to n do var := Q[1, i][2];
1268          if type(X[i], symbol) then
1269            position := Search(X[i],
                  new_full_var);
1270            new_var := new_full_var[position + 1];
1271            for j to n do
1272              Q[j, i] := subs(var = new_var, Q[j,
                    i]);
1273              Q[j, i][1] := cX[i];
1274            end do;
1275          elif X[i] = 1 then
1276            for j to n - 1 do
1277              Q[j, i] := 0;
```

```
1278            end do;
1279            Q[n, i] := cX[i];
1280          end if;
1281        end do;
1282        for i to n do
1283          var := R[i, 1][2];
1284          if type(Y[i], symbol) then
1285            position := Search(Y[i],
                  new_full_var);
1286            new_var := new_full_var[position + 1];
1287            for j to n do
1288              R[i, j] := subs(var = new_var, R[i,
                    j]);
1289            end do;
1290          elif Y[i] = 1 then
1291            for j to n - 1 do
1292              R[i, j] := 0;
1293            end do;
1294            R[i, n] := 1;
1295          end if;
1296        end do;
1297        return [Q, R];
1298      end proc;
1299
1300      GeneralSolutionForm := proc(n, n_list, var)
1301        local Q, k, pre, final, j, n_j, rs, u, L,
             S, addfunc, w, L1, L2;
1302        Q := Coefficients_1case(n, n_list)[1];
1303        k := nops(n_list); pre := 0;
1304        final := 0;
1305        for j to k do n_j := n_list[j];
1306          if 3 <= n_j then rs := 0;
1307            for u from pre + 1 to pre + n_j do
1308              L := Q[u, 1][3];
1309              if type(L, list) then
1310                S := seq(A[L[j]](x), j = 1 ..
                      nops(L));
1311                addfunc := 1;
1312                if 1 < nops(L) then
1313                  for w to nops(L) do
1314                    addfunc := addfunc*S[w];
1315                  end do;
```

```
1316                else
1317                  addfunc := addfunc*S;
1318                end if;
1319                rs := rs + Sigma(var[L]*addfunc);
1320              else
1321                rs := rs + var[L];
1322              end if;
1323            end do;
1324            final := final + [rs]*m[j](x);
1325          elif n_j = 2 then
1326            rs := 0;
1327            u := pre + 1;
1328            L1 := Q[u, 1][3];
1329            L2 := Q[u + 1, 1][3];
1330            addfunc := A[L1](x);
1331            rs := var[L1]*addfunc + var[L2];
1332            final := final + rs*m[j](x);
1333          else
1334            u := pre + 1;
1335            L := Q[u, 1][3];
1336            rs := var[L];
1337            final := final + rs*m[j](x);
1338          end if;
1339          pre := pre + n_j;
1340        end do;
1341        return final;
1342      end proc;
1343
1344      SimpleSolve := proc(fe)
1345        local n, cX, X, Y, full_var,
              general_form, CoM, Q, R, P, S, Sys,
              rs, i, j;
1346        cX := ListUnknownFunction(fe)[1];
1347        X := ListUnknownFunction(fe)[2];
1348        Y := ListUnknownFunction(fe)[3];
1349        n := nops(X);
1350        full_var := UnknownFunction_var(X, Y);
1351        print(full_var);
1352        general_form := GeneralSolutionForm(n,
              [n], alpha);
1353        print(general_form);
1354        CoM := Coefficients_1case(n, [n]); Q :=
```

```
        CoM[1];
1355    R := CoM[2];
1356    P := CoM[3];
1357    Q := Modified_CoefficientMatrix(cX, X, Y,
        full_var, Q, R)[1];
1358    R := Modified_CoefficientMatrix(cX, X, Y,
        full_var, Q, R)[2];
1359    Q := Normalize_CoefficientMatrix(Q, n);
1360    R := Normalize_CoefficientMatrix(R, n);
1361    P := Normalize_CoefficientMatrix(P, n); S
        := (Q . R) - P;
1362    Sys := {seq(seq(S[i, j], j = 1 .. n), i =
        1 .. n)};
1363    rs := solve(Sys);
1364    return [Q, R, P, [rs]];
1365   end proc;
1366
1367   LCfesolve := proc()
1368    local OUTPUT, fe, n, cX, X, Y, full_var,
        GeneralCoM, Q, R, P, S, Sys, rs, i,
        case, list_n, general_form, j,
        list_var, p, list_var_full, q;
1369    if nargs=1 then
1370     OUTPUT:='brief';
1371    elif nargs = 2 then
1372     if args[2] in {'brief', 'full'} then
1373      OUTPUT:=args[2];
1374     else
1375      ERROR("invalid argument: %1",
          args[2]);
1376     end if;
1377    else
1378     ERROR("invalid input", args);
1379    end if;
1380    fe:=args[1];
1381    cX:=ListUnknownFunction(fe)[1];
1382    X:=ListUnknownFunction(fe)[2];
1383    Y:=ListUnknownFunction(fe)[3];
1384    n:=nops(X);
1385    full_var:=UnknownFunction_var(X, Y);
1386    GeneralCoM:=LCfesolveGeneral(n);
1387    for i from 1 to nops(GeneralCoM) do
```

```
1388        case:=GeneralCoM[i];
1389        list_n:=case[1];
1390        printf("Case: %a",list_n);
1391        Q:=case[2][1];
1392        R:=case[2][2];
1393        P:=case[2][3];
1394        for j from 1 to nops(full_var) do
1395          general_form:=GeneralSolutionForm(n,
                list_n, full_var[j][2]);
1396          print(full_var[j][1]=general_form);
1397        end do;
1398        list_var_full:=[seq(convert(Char(i),
              name), i = 97..97 + pre_char_n - 1)];
1399        if nops(list_var_full)<>0 then
1400          printf("for %a",list_var_full[1]);
1401          for p from 2 to
                (nops(list_var_full))/(2) do
1402            printf(", %a",list_var_full[p]);
1403          end do;
1404          if nops(list_var_full)=4 then
1405            printf(" = 1, 2.");
1406          else printf(" = 1,...,%d.",
                (nops(list_var_full))/(2));
1407          end if;
1408        end if;
1409        Q := Modified_CoefficientMatrix(cX,X,
              Y, full_var, Q, R)[1];
1410        R := Modified_CoefficientMatrix(cX,X,
              Y, full_var, Q, R)[2];
1411        Q := Normalize_CoefficientMatrix(Q,n);
1412        R := Normalize_CoefficientMatrix(R,n);
1413        P := Normalize_CoefficientMatrix(P,n);
1414        printf("\nMoreover, the coefficients
              fulfill the equation Q.R=P, with the
              matrices Q,R,P as follows,
              respectively\n");
1415        printf("\t\t\t\t\tQ = ");
1416        print(Q);
1417        printf("\t\t\t\t\tR = ");
1418        print(R);
1419        printf("\t\t\t\t\tP = ");
1420        print(P);
```

```
1421          if nops(list_var_full)<>0 then
1422            printf("for %a",list_var_full[1]);
1423            for q from 2 to nops(list_var_full) do
1424              printf(", %a",list_var_full[q]);
1425            end do;
1426            if nops(list_var_full)=4 then
1427              printf(" = 1, 2.");
1428            else printf(" = 1,...,%d.",
                   (nops(list_var_full))/(2));
1429            end if;
1430          end if;
1431          S:=Q.R-P;
1432          Sys:={seq(seq((S)[i,j], j=1..n),
                 i=1..n)};
1433          printf("The coefficients fulfill the
                 system of equations:");
1434          print(Sys);
1435          if OUTPUT='full' then
1436            printf("The solutions of the above
                   system of equations:");
1437            rs:=solve(Sys);
1438            print(rs);
1439          end if;
1440        end do;
1441      end proc;
1442    end module;
1443
1444    FETypes := module()
1445      option package;
1446      local EQ, FUNCS, VARS, INDEX,
           checkconstant, presim, conv, checkcoeff,
           checkfxplusy, checkfxtimesyorxy,
           checkcfxplusy, checkcfxtimesyorxy,
           checkrhsCauchyPexider, CauchyPexiderfe,
           Jensenfe, checkfxplusyminusxy, Hosszufe,
           Mikusinskife, Alternativefe, Linearfe,
           LeviCivitafe, checkfaxplusbyplusc,
           checkcfaxplusbyplusc, checkAfxorBfy,
           GeneralLinearfe, checkfxminusy,
           checkcfxminusy, Quadraticfe,
           DAlembertfe, GeneralDAlembertfe,
           Trigonometric1fe, Trigonometric2fe,
```

```
      Trigonometric3fe, Trigonometric4fe,
      solver, mapindex, checktype, G, testf;
1447  export fewhattype, fetype, feinfo;
1448
1449  with(ListTools);
1450  with(Maplets[Elements]);
1451
1452  testf := Sum(f_i((p_i . x) + (q_i . y)), i
        = 0 .. n + 1);
1453  checkconstant := proc(c)
1454    local i;
1455    if type(c, rational) then
1456      return 1;
1457    elif type(c, radical) and type(op(1, c),
         integer) then
1458      return 1;
1459    elif type(c, `*`) then
1460      for i in op(c) do
1461        if checkconstant(i) = 0 then
1462          return 0;
1463        end if;
1464      end do;
1465      return 1;
1466    else return 0;
1467    end if;
1468  end proc;
1469
1470  presim := proc(fe)
1471    local nfe, n, i;
1472    nfe := factor(fe);
1473    i := 1;
1474    if type(nfe, `*`) then
1475      n := nops(nfe);
1476      while i<=n do
1477        if checkconstant(op(i, nfe)) = 1 then
1478          nfe := subsop(i = 1, nfe);
1479          i := i - 1;
1480          n := n - 1;
1481        end if;
1482        i := i+1;
1483      end do;
1484    end if;
```

```
1485        return nfe;
1486      end proc;
1487
1488      conv := proc(parameq, paramfcts, paramvars)
1489        local eq, fction, var, i, t;
1490        eq, fction, var := parameq, paramfcts,
              paramvars;
1491        if not type(fction, list) then
1492          fction := [fction];
1493        end if;
1494        if not type(var, list) then
1495          var := [var];
1496        end if;
1497        if type(eq, `=`) then
1498          eq := lhs(eq) - rhs(eq);
1499        end if;
1500        for i in fction do
1501          if not has(eq, i) then
1502            if member(i, fction, 't') then
1503              fction := subsop(t = NULL, fction);
1504            end if;
1505          end if;
1506        end do;
1507        for i in var do
1508          if not has(eq, i) then
1509            if member(i, var, 't') then
1510              var := subsop(t = NULL, var);
1511            end if;
1512          end if;
1513        end do;
1514        if nops(fction) = 0 then
1515          error "Invalid unknown functions";
1516        end if;
1517        if nops(var) = 0 then
1518          error "Invalid variables";
1519        end if;
1520        return eq, fction, var;
1521      end proc;
1522
1523      checkcoeff := proc(fe)
1524        local n, i, t, j, pos, c;
1525        n := nops(fe);
```

```
1526        if n = 1 and not type(fe, function) then
1527          if checkconstant(fe) = 0 then
1528            return 0;
1529          end if;
1530        elif type(fe, `+`) then
1531          for i to n do
1532            t := op(i, fe);
1533            if nops(t) = 1 and not type(t,
                  function) then
1534              if checkconstant(t) = 0 then
1535                return 0;
1536              end if;
1537            elif 1 < nops(t) and type(t, `*`) then
1538              for j to nops(t) do
1539                if type(op(j, t), function) then
1540                  pos := j;
1541                  break;
1542                end if;
1543              end do;
1544              c := `*`(op(1 .. pos - 1, t));
1545              if checkconstant(c) = 0 then
1546                return 0;
1547              end if;
1548            end if;
1549          end do;
1550          return 1;
1551        elif type(fe, `*`) then
1552          for i to n do
1553            t := op(i, fe);
1554            checkcoeff(t);
1555          end do;
1556        else return 0;
1557        end if;
1558      end proc;
1559
1560      checkfxplusy := proc(fe, var)
1561        local Svar, flag, nfe, i, t, lfv, Lfe,
              Rfe, ufunc;
1562        Svar := convert(var, `+`);
1563        flag := false;
1564        nfe := fe;
1565        for i to nops(nfe) do
```

```
1566          t := op(i, nfe);
1567          if type(t, function) and op(t) = Svar
                 then
1568            lfv := op(0, t);
1569            Lfe := t;
1570            Rfe := Lfe - nfe;
1571            flag := true;
1572            ufunc := op(0, t);
1573            if not ufunc in FUNCS then
1574              error "Invalid unknown functions";
1575            end if;
1576            break;
1577          end if;
1578        end do;
1579        if flag = true then
1580          return 1, Lfe, Rfe, ufunc;
1581        else return 0;
1582        end if;
1583      end proc;
1584
1585      checkfxtimesyorxy := proc(fe, var)
1586        local Mvar1, Mvar2, flag, nfe, i, t, lfv,
              Lfe, Rfe, ufunc;
1587        Mvar1 := convert(var, `*`);
1588        Mvar2 := cat(seq(var[i], i = 1 .. 2));
1589        flag := false;
1590        nfe := fe;
1591        for i to nops(nfe) do
1592          t := op(i, nfe);
1593          if type(t, function) and (op(t) = Mvar1
                 or op(t) = Mvar2) then
1594            lfv := op(0, t);
1595            Lfe := t;
1596            Rfe := Lfe - nfe;
1597            flag := true;
1598            ufunc := op(0, t);
1599            if not ufunc in FUNCS then
1600              error "Invalid unknown functions";
1601            end if;
1602            break;
1603          end if;
1604        end do;
```

```
1605        if flag = true then
1606          return 1, Lfe, Rfe, ufunc;
1607        else return 0;
1608        end if;
1609      end proc;
1610
1611      checkcfxplusy := proc(fe, var)
1612        local Svar, flag, nfe, i, t, Lfe, Rfe,
              ufunc;
1613        Svar := convert(var, `+`);
1614        flag := false;
1615        nfe := fe;
1616        for i to nops(nfe) do
1617          t := op(i, nfe);
1618          if type(t, function) and op(t) = Svar
                then
1619            Lfe := t;
1620            Rfe := Lfe - nfe;
1621            flag := true;
1622            ufunc := op(0, t);
1623            if not ufunc in FUNCS then
1624              error "Invalid unknown functions";
1625            end if;
1626            break;
1627          elif type(t, `*`) then
1628            if type(op(-1, t), function) and
                  op(op(-1, t)) = Svar then
1629              Lfe := simplify(t/op(1 .. -2, t));
1630              Rfe := simplify(Lfe - nfe/op(1 ..
                     -2, t));
1631              flag := true; ufunc := op(0, op(-1,
                     t));
1632              if not ufunc in FUNCS then
1633                error "Invalid unknown functions";
1634              end if;
1635              break;
1636            end if;
1637          end if;
1638        end do;
1639        if flag = true then
1640          return 1, Lfe, Rfe, ufunc;
1641        else return 0;
```

```
1642        end if;
1643      end proc;
1644
1645      checkcfxtimesyorxy := proc(fe, var)
1646        local Mvar1, Mvar2, flag, nfe, i, t, Lfe,
              Rfe, ufunc;
1647        Mvar1 := convert(var, `*`);
1648        Mvar2 := cat(seq(var[i], i = 1 .. 2));
1649        flag := false;
1650        nfe := fe;
1651        for i to nops(nfe) do
1652          t := op(i, nfe);
1653          if type(t, function) and (op(t) = Mvar1
                or op(t) = Mvar2) then
1654            Lfe := t;
1655            Rfe := Lfe - nfe;
1656            flag := true;
1657            ufunc := op(0, t);
1658            if not ufunc in FUNCS then
1659              error "Invalid unknown functions";
1660            end if;
1661            break;
1662          elif type(t, `*`) then
1663            if type(op(-1, t), function) and
                  (op(op(-1, t)) = Mvar1 or
                  op(op(-1, t)) = Mvar2) then
1664              Lfe := simplify(t/op(1 .. -2, t));
1665              Rfe := simplify(Lfe - nfe/op(1 ..
                     -2, t));
1666              flag := true; ufunc := op(0, op(-1,
                     t));
1667              if not ufunc in FUNCS then
1668                error "Invalid unknown functions";
1669              end if;
1670              break;
1671            end if;
1672          end if;
1673        end do;
1674        if flag = true then
1675          return 1, Lfe, Rfe, ufunc;
1676        else return 0;
1677        end if;
```

```
1678    end proc;
1679
1680    checkrhsCauchyPexider := proc(rhseq, var)
1681      local t1, t2, ufunc;
1682      if nops(rhseq) <> 2 then
1683        return 0;
1684      end if;
1685      if not (type(rhseq, `+`) or type(rhseq,
            `*`)) then
1686        return 0;
1687      end if;
1688      t1 := op(1, rhseq);
1689      t2 := op(2, rhseq);
1690      if not (type(t1, function) and type(t2,
            function)) then
1691        return 0;
1692      end if;
1693      if evalb([op(t1), op(t2)] = var) or
            evalb([op(t2), op(t1)] = var) then
1694        if not (op(0, t1) in FUNCS and op(0,
              t2) in FUNCS) then
1695          error "Invalid unknown functions";
1696        end if;
1697        ufunc := MakeUnique([op(0, t1), op(0,
              t2)]);
1698        return 1, whattype(rhseq), ufunc;
1699      else return 0;
1700      end if;
1701    end proc;
1702
1703    CauchyPexiderfe:=proc(fe, func, var)
1704      local nfe, nfunc, nvar, flaglplus,
            flaglmul, flagrplus, flagrmul,
            checklplus, checklmul, rhsfe, ufl,
            checkrhs, oper, ufr, flagCau, flagPex;
1705      nfe, nfunc, nvar := fe, func, var;
1706      flaglplus := false;
1707      flaglmul := false;
1708      flagrplus := false;
1709      flagrmul := false;
1710      checklplus := checkcfxplusy(nfe,nvar);
1711      checklmul := checkcfxtimesyorxy(nfe,
```

```
          nvar);
1712      if checklplus <> 0 then
1713        flaglplus := true;
1714        rhsfe := checklplus[3];
1715        ufl := checklplus[4];
1716      elif checklmul <> 0 then
1717        flaglmul := true;
1718        rhsfe := checklmul[3];
1719        ufl := checklmul[4];
1720      else
1721        return 0;
1722      end if;
1723      checkrhs:=checkrhsCauchyPexider(rhsfe,
          nvar);
1724      if checkrhs<>0 then
1725        oper:=checkrhs[2];
1726        ufr:=checkrhs[3];
1727        if oper='+' then
1728          flagrplus:=true;
1729        elif  oper='*' then
1730          flagrmul:=true;
1731        else
1732          return 0;
1733        end if;
1734      end if;
1735      if nops(ufr)=1 and ufl=ufr[1] then
1736        flagCau := true;
1737      elif nops(ufr)=2 and (not
          member(ufl,ufr)) then
1738        flagPex:=true;
1739      end if;
1740      if flagCau=true then
1741        if flaglplus = true and flagrplus= true
            then
1742          return 1.1,ufl;
1743        elif flaglplus = true and flagrmul=
            true then
1744          return 1.2;
1745        elif flaglmul = true and flagrplus=
            true then
1746          return 1.3;
1747        elif flaglmul = true and flagrmul= true
```

```
            then
1748          return 1.4;
1749        else
1750          return 0;
1751        end if;
1752      elif flagPex = true then
1753        if flaglplus = true and flagrplus =
              true then
1754          return 3.1;
1755        elif flaglplus = true and flagrmul =
              true then
1756          return 3.2;
1757        elif flaglmul = true and flagrplus =
              true then
1758          return 3.3;
1759        elif flaglmul = true and flagrmul =
              true then
1760          return 3.4;
1761        else
1762          return 0;
1763        end if;
1764      else
1765        return 0;
1766      end if;
1767    end proc;
1768
1769    Jensenfe := proc(fe, func, var)
1770      local nfe, nfunc, nvar, checklplus,
            rhsfe, ufl, checkrhs, oper, ufr;
1771      nfe, nfunc, nvar := fe, func, var;
1772      checklplus := checkcfxplusy(nfe,
            1/2*nvar);
1773      if checklplus <> 0 then
1774        rhsfe := checklplus[3];
1775        ufl := checklplus[4];
1776      else
1777        return 0;
1778      end if;
1779      checkrhs :=
            checkrhsCauchyPexider(2*rhsfe, nvar);
1780      if checkrhs <> 0 then
1781        oper := checkrhs[2];
```

```
1782          ufr := checkrhs[3];
1783          if oper = `+` and nops(ufr) = 1 and ufl
                 = ufr[1] then
1784            return 2;
1785          else
1786            return 0;
1787          end if;
1788        else
1789          return 0;
1790        end if;
1791      end proc;
1792
1793      checkfxplusyminusxy := proc(fe, var)
1794        local Svar, Mvar1, Mvar2, check1, check2,
                 flag, nfe, i, t, lfv, Lfe, Rfe, ufunc;
1795        Svar := convert(var, `+`);
1796        Mvar1 := convert(var, `*`);
1797        Mvar2 := cat(seq(var[i], i = 1 .. 2));
1798        check1 := Svar - Mvar1;
1799        check2 := Svar - Mvar2;
1800        flag := false; nfe := fe;
1801        for i to nops(nfe) do
1802          t := op(i, nfe);
1803          if type(t, function) then
1804            if evalb(op(t) = check1) or
                   evalb(op(t) = check2) then
1805              lfv := op(0, t);
1806              Lfe := t;
1807              Rfe := Lfe - nfe;
1808              flag := true;
1809              ufunc := op(0, t);
1810              if not ufunc in FUNCS then
1811                error "Invalid unknown functions";
1812              end if;
1813              break;
1814            end if;
1815          end if;
1816        end do;
1817        if flag = true then
1818          return 1, Lfe, Rfe, ufunc;
1819        else
1820          return 0;
```

```
1821        end if;
1822      end proc;
1823
1824      Hosszufe := proc(fe, func, var)
1825        local nfe, nfunc, nvar, checklmul,
              lhsfe1, rhsfe1, ufl1, ufl2, checkrhs1,
              lhsfe2, rhsfe2, checkrhs2, oper, ufr;
1826        nfe, nfunc, nvar := fe, func, var;
1827        checklmul := checkcfxtimesyorxy(nfe,
              nvar);
1828        if checklmul <> 0 then
1829          lhsfe1 := checklmul[2];
1830          rhsfe1 := checklmul[3];
1831          ufl1 := checklmul[4];
1832        else
1833          return 0;
1834        end if;
1835        checkrhs1 := checkfxplusyminusxy(-rhsfe1,
              nvar);
1836        if checkrhs1 <> 0 then
1837          lhsfe2 := checkrhs1[2];
1838          rhsfe2 := checkrhs1[3];
1839          ufl2 := checkrhs1[4];
1840        else
1841          return 0;
1842        end if;
1843        checkrhs2 :=
              checkrhsCauchyPexider(rhsfe2, nvar);
1844        if checkrhs2 <> 0 then
1845          oper := checkrhs2[2];
1846          ufr := checkrhs2[3];
1847          if not (oper = `+`) then
1848            return 0;
1849          end if;
1850        end if;
1851        if nops(ufr) = 1 and ufl1 = ufr[1] and
              ufl2 = ufr[1] then
1852          return 4;
1853        else
1854          return 0;
1855        end if;
1856      end proc;
```

```
1857
1858    Mikusinskife := proc(fe, func, var)
1859      local nfe, nfunc, nvar, t1, t2,
             checklplus, uf1, checkCau, uf2;
1860      nfe, nfunc, nvar := fe, func, var;
1861      nfe := factor(nfe);
1862      if type(nfe, `*`) and nops(nfe) = 2 then
1863        t1 := op(1, nfe);
1864        t2 := op(2, nfe);
1865        checklplus := checkcfxplusy(nfe, nvar);
1866        if checklplus <> 0 then
1867          uf1 := checklplus[4];
1868        else
1869          return 0;
1870        end if;
1871        checkCau := CauchyPexiderfe(t2, nfunc,
              nvar);
1872        if checkCau[1] = 1.1 then
1873          uf2 := checkCau[2];
1874        else
1875          return 0;
1876        end if;
1877        if uf1 = uf2 then
1878          return 5;
1879        else
1880          return 0;
1881        end if;
1882      else
1883        return 0;
1884      end if;
1885    end proc;
1886
1887    Alternativefe := proc(fe, func, var)
1888      local nfe, nfunc, nvar, t1, t2, check1,
             uf1, check2, t3, uf2, check3, oper,
             uf3;
1889      nfe, nfunc, nvar := fe, func, var;
1890      if type(nfe,`*`) and nops(nfe) = 2 then
1891        t1 := op(1, nfe); t2 := op(2, nfe);
1892        check1 := CauchyPexiderfe(t2, nfunc,
             nvar);
1893        if check1[1] = 1.1 then
```

```
1894            uf1 := check1[2];
1895          else
1896            return 0;
1897          end if;
1898          check2 := checkcfxplusy(t1, nvar);
1899          if check2 <> 0 then
1900            t3 := check2[3];
1901            uf2 := check2[4];
1902          else
1903            return 0;
1904          end if;
1905          check3 := checkrhsCauchyPexider(-t3,
                nvar);
1906          if check3 <> 0 then
1907            oper := check3[2];
1908            uf3 := check3[3];
1909            if not (oper = `+`) then
1910              return 0;
1911            end if;
1912          end if;
1913          if nops(uf3) = 1 and uf1 = uf3[1] and
                uf2 = uf3[1] then
1914            return 6;
1915          else
1916            return 0;
1917          end if;
1918        else
1919          return 0;
1920        end if;
1921      end proc;
1922
1923      Linearfe:=proc(fe, func,var)
1924        local nfe, nfunc, nvar, nterms, j,
              eqtemp, arg;
1925        nfe, nfunc, nvar:=fe, func,var;
1926        nterms := nops(nfe);
1927        for j to nterms do
1928          eqtemp := nfe;
1929          if type(eqtemp, `+`) then
1930            eqtemp := op(j, eqtemp);
1931          elif not type(eqtemp, `*`) and not
                type(eqtemp, function) then
```

```
1932          return 0;
1933        end if;
1934        if type(eqtemp, numeric) or
               type(eqtemp, symbol) then
1935          return 0;
1936        end if;
1937        if type(eqtemp, `*`) then
1938          if type(op(1,eqtemp), numeric) then
1939            eqtemp := subsop(1=1,eqtemp);
1940          end if;
1941          if not type(eqtemp, function) then
1942            return 0;
1943          end if;
1944        end if;
1945        if type(eqtemp, function) then
1946          if not (op(0,eqtemp) in FUNCS) then
1947            error "Invalid unknown functions";
1948          end if;
1949          arg := op(eqtemp);
1950          if not type(arg, polynom(rational,
                 nvar)) then
1951            return 0;
1952          end if;
1953          if degree(arg,var) <> 1  then
1954            return 0;
1955          end if;
1956          if ldegree(arg,var) <> 1  then
1957            return 0;
1958          end if;
1959        else return 0;
1960        end if;
1961      end do;
1962      return 8;
1963    end proc;
1964
1965    LeviCivitafe := proc(fe, func, var)
1966      local nfe, nfunc, nvar, checklhs, Rfe, n,
             i, b, c, checkrhs, oper, d;
1967      nfe, nfunc, nvar := fe, func, var;
1968      checklhs := checkcfxplusy(nfe, nvar);
1969      if checklhs <> 0 then
1970        Rfe := checklhs[3];
```

```
1971        else
1972          return 0;
1973        end if;
1974        if type(Rfe, `+`) then
1975          n := nops(Rfe);
1976        elif type(Rfe, `*`) and nops(Rfe) = 2 or
              type(Rfe, function) then
1977          n := 1;
1978        end if;
1979        for i to n do
1980          b := Rfe;
1981          if type(b, `+`) then
1982            b := op(i, b);
1983          elif not (type(b, `*`) or type(b,
                 function)) then
1984            return 0;
1985          end if;
1986          if type(b, `*`) then
1987            if nops(b) = 3 then
1988              c := op(1, b);
1989              if type(c, integer) then
1990                checkrhs :=
                       checkrhsCauchyPexider(b/c,
                       nvar);
1991                if checkrhs <> 0 then
1992                  oper := checkrhs[2];
1993                  if not (oper = `*`) then
1994                    return 0;
1995                  end if;
1996                else
1997                  return 0;
1998                end if;
1999              else
2000                return 0;
2001              end if;
2002            elif nops(b) = 2 then
2003              c := op(1, b);
2004              if type(c, integer) then
2005                d := op(2, b);
2006                if type(d, function) then
2007                  if not op(0, d) in FUNCS then
2008                    error "Invalid unknown
```

```
                    functions";
2009                end if;
2010                if `not`(op(d) = nvar[1] or
                       op(d) = nvar[2]) then
2011                  return 0;
2012                end if;
2013              else
2014                return 0;
2015              end if;
2016            else
2017              checkrhs :=
                     checkrhsCauchyPexider(b, nvar);
2018              if checkrhs <> 0 then
2019                oper := checkrhs[2];
2020                if not (oper = `*`) then
2021                  return 0;
2022                end if;
2023              else
2024                return 0;
2025              end if;
2026            end if;
2027          end if;
2028        else
2029          if not op(0, b) in FUNCS then
2030            error "Invalid unknown functions";
2031          end if;
2032          if `not`(op(b) = nvar[1] or op(b) =
                 nvar[2]) then
2033            return 0;
2034          end if;
2035        end if;
2036      end do;
2037      return 9;
2038    end proc;
2039
2040    checkfaxplusbyplusc := proc(fe, var)
2041      local flag, nfe, i, t, in_term, a, b,
           Lfe, Rfe, ufunc;
2042      flag := false;
2043      nfe := fe;
2044      for i to nops(nfe) do
2045        t := op(i, nfe);
```

```
2046          if type(t, function) then
2047            in_term := op(t);
2048            if type(in_term, polynom(numeric,
                   var)) and degree(in_term, var) = 1
                   then
2049              a := coeff(in_term, var[1], 1);
2050              b := coeff(in_term, var[2], 1);
2051              if a <> 0 and b <> 0 then
2052                flag := true;
2053                Lfe := t;
2054                Rfe := Lfe - nfe;
2055                ufunc := op(0, t);
2056                if not ufunc in FUNCS then
2057                  error "Invalid unknown
                         functions";
2058                end if;
2059                break;
2060              end if;
2061            end if;
2062          end if;
2063        end do;
2064        if flag = false then
2065          return 0;
2066        else
2067          return 1, Lfe, Rfe, ufunc;
2068        end if;
2069      end proc;
2070
2071      checkcfaxplusbyplusc := proc(fe, var)
2072        local flag, nfe, i, t, in_term, a, b,
             Lfe, Rfe, ufunc, coeff, func_term;
2073        flag := false;
2074        nfe := fe;
2075        for i to nops(nfe) do
2076          t := op(i, nfe);
2077          if type(t, function) then
2078            in_term := op(t);
2079            if type(in_term, polynom(numeric,
                   var)) and degree(in_term, var) = 1
                   then
2080              a := coeff(in_term, var[1], 1);
2081              b := coeff(in_term, var[2], 1);
```

```
2082              if a <> 0 and b <> 0 then
2083                flag := true;
2084                Lfe := t;
2085                Rfe := Lfe - nfe;
2086                ufunc := op(0, t);
2087                if not ufunc in FUNCS then
2088                  error "Invalid unknown
                         functions";
2089                end if;
2090                break;
2091              end if;
2092            end if;
2093          elif type(t, `*`) then
2094            coeff := op(1 .. -2, t);
2095            func_term := op(-1, t);
2096            if type(func_term, function) then
2097              in_term := op(func_term);
2098              if type(in_term, polynom(numeric,
                    var)) and degree(in_term, var) =
                    1 then
2099                a := coeff(in_term, var[1], 1);
2100                b := coeff(in_term, var[2], 1);
2101                if a <> 0 and b <> 0 then
2102                  Lfe := t/coeff;
2103                  Rfe := Lfe - nfe/coeff;
2104                  flag := true;
2105                  ufunc := op(0, func_term);
2106                  if not ufunc in FUNCS then
2107                    error "Invalid unknown
                           functions";
2108                  end if;
2109                  break;
2110                end if;
2111              end if;
2112            end if;
2113          end if;
2114        end do;
2115        if flag = false then
2116          return 0;
2117        else
2118          return 1, Lfe, Rfe, ufunc;
2119        end if;
```

```
2120      end proc;
2121
2122      checkAfxorBfy := proc(fe, var)
2123        local flag, nfe, i, t, nrhs, ufunc, func;
2124        flag := false;
2125        nfe := fe;
2126        for i to nops(nfe) do
2127          t := op(i, nfe);
2128          if type(t, function) then
2129            if op(t) = var then
2130              flag := true;
2131              ufunc := op(0, t);
2132              if not ufunc in FUNCS then
2133                error "Invalid unknown functions";
2134              end if;
2135              break;
2136            end if;
2137          elif type(t, `*`) then
2138            func := op(-1, t);
2139            if type(func, function) then
2140              if op(func) = var then
2141                flag := true;
2142                ufunc := op(0, func);
2143                if not ufunc in FUNCS then
2144                  error "Invalid unknown
                             functions";
2145                end if;
2146                break;
2147              end if;
2148            end if;
2149          end if;
2150        end do;
2151        if flag = false then
2152          return 0;
2153        else
2154          return 1, t, ufunc;
2155        end if;
2156      end proc;
2157
2158      GeneralLinearfe := proc(fe, func, var)
2159        local nfe, nfunc, nvar, check1, rhs, lf,
             check2, nrhs1, rf1, check3, nrhs2, rf2;
```

```
2160        nfe, nfunc, nvar := fe, func, var;
2161        check1 := checkcfaxplusbyplusc(nfe, nvar);
2162        if check1 <> 0 then
2163          rhs := check1[3];
2164          lf := check1[4];
2165        else
2166          return 0;
2167        end if;
2168        check2 := checkAfxorBfy(rhs, nvar[1]);
2169        if check2 <> 0 then
2170          nrhs1 := check2[2];
2171          rf1 := check2[3];
2172        else
2173          return 0;
2174        end if;
2175        check3 := checkAfxorBfy(rhs, nvar[2]);
2176        if check3 <> 0 then
2177          nrhs2 := check3[2];
2178          rf2 := check3[3];
2179        else
2180          return 0;
2181        end if;
2182        if not type(rhs - nrhs1 - nrhs2, numeric)
               then
2183          return 0;
2184        end if;
2185        if lf = rf1 and lf = rf2 then
2186          return 7;
2187        else
2188          return 0;
2189        end if;
2190      end proc;
2191
2192      checkfxminusy := proc(fe, var)
2193        local Mvar1, Mvar2, flag, nfe, i, t, Lfe,
               Rfe, ufunc, po;
2194        if nops(var) <> 2 then
2195          return 0;
2196        end if;
2197        Mvar1 := var[1] - var[2];
2198        Mvar2 := -Mvar1;
2199        flag := false;
```

```
2200        nfe := fe;
2201        for i to nops(nfe) do
2202          t := op(i, nfe);
2203          if not type(t, function) then
2204            break;
2205          end if;
2206          if op(t) = Mvar1 or op(t) = Mvar2 then
2207            Lfe := t;
2208            Rfe := Lfe - nfe;
2209            flag := true;
2210            ufunc := op(0, t);
2211            if not ufunc in FUNCS then
2212              error "Invalid unknown functions";
2213            end if;
2214            if op(t) = Mvar1 then
2215              po := 2;
2216            else
2217              po := 1;
2218            end if;
2219            break;
2220          end if;
2221        end do;
2222        if flag = true then
2223          return 1, Lfe, Rfe, ufunc, po;
2224        else
2225          return 0;
2226        end if;
2227      end proc;
2228
2229      checkcfxminusy:=proc(fe, var)
2230        local Mvar1, Mvar2, flag, nfe, i, t, Lfe,
             Rfe, ufunc, po;
2231        if nops(var)<>2 then
2232          return 0;
2233        end if;
2234        Mvar1:=var[1]-var[2];
2235        Mvar2:=-Mvar1;
2236        flag:=false;
2237        nfe:=fe;
2238        for i from 1 to nops(nfe) do
2239          t:=op(i, nfe);
2240          if type(t, function) and op(t)=Mvar1 or
```

```
          op(t)=Mvar2 then
2241        Lfe:=t;
2242        Rfe:= Lfe-nfe;
2243        flag:=true;
2244        ufunc:=op(0, t);
2245        if not (ufunc in FUNCS) then
2246          error "Invalid unknown functions";
2247        end if;
2248        if op(t)= Mvar1 then
2249          po:=2;
2250        else
2251          po:=1;
2252        end if;
2253        break;
2254      elif type(t, `*`) then
2255        if type(op(-1, t), function) and
              (op(op(-1, t))=Mvar1 or op(op(-1,
              t))=Mvar2) then
2256          Lfe:=simplify(t/(op(1..-2, t)));
2257          Rfe:=simplify(Lfe-nfe/(op(1..-2,
                t)));
2258          flag:=true;
2259          ufunc:=op(0, op(2, t));
2260          if not (ufunc in FUNCS) then
2261            error "Invalid unknown functions";
2262          end if;
2263          if op(op(-1, t))=Mvar1 then
2264            po:=2;
2265          else
2266            po:=1;
2267          end if;
2268          break;
2269        end if;
2270      end if;
2271    end do;
2272    if flag = true then
2273      return 1, Lfe,  Rfe, ufunc, po;
2274    else
2275      return 0;
2276    end if;
2277  end proc;
2278
```

```
2279    Quadraticfe := proc(fe, func, var)
2280      local nfe, nfunc, nvar, checklplus,
            rhsfe1, ufl1, checklminus, rhsfe2,
            ufl2, checkrhs, oper, ufr;
2281      nfe, nfunc, nvar := fe, func, var;
2282      checklplus := checkcfxplusy(nfe, nvar);
2283      if checklplus <> 0 then
2284        rhsfe1 := checklplus[3];
2285        ufl1 := checklplus[4];
2286      else return 0;
2287      end if;
2288      checklminus := checkfxminusy(-rhsfe1,
            nvar);
2289      if checklminus <> 0 then
2290        rhsfe2 := checklminus[3];
2291        ufl2 := checklminus[4];
2292      else return 0;
2293      end if;
2294      checkrhs :=
            checkrhsCauchyPexider(1/2*rhsfe2,
            nvar);
2295      if checkrhs <> 0 then
2296        oper := checkrhs[2];
2297        ufr := checkrhs[3];
2298        if not (oper = `+`) then
2299          return 0;
2300        end if;
2301      end if;
2302      if nops(ufr) = 1 and ufl1 = ufr[1] and
            ufl2 = ufr[1] then
2303        return 10;
2304      else
2305        return 0;
2306      end if;
2307    end proc;
2308
2309    DAlembertfe := proc(fe, func, var)
2310      local nfe, nfunc, nvar, checklplus,
            rhsfe1, ufl1, checklminus, rhsfe2,
            ufl2, checkrhs, oper, ufr;
2311      nfe, nfunc, nvar := fe, func, var;
2312      checklplus := checkcfxplusy(nfe, nvar);
```

```
2313        if checklplus <> 0 then
2314          rhsfe1 := checklplus[3];
2315          ufl1 := checklplus[4];
2316        else
2317          return 0;
2318        end if;
2319        checklminus := checkfxminusy(-rhsfe1,
              nvar);
2320        if checklminus <> 0 then
2321          rhsfe2 := checklminus[3];
2322          ufl2 := checklminus[4];
2323        else
2324          return 0;
2325        end if;
2326        checkrhs :=
              checkrhsCauchyPexider(1/2*rhsfe2,
              nvar);
2327        if checkrhs <> 0 then
2328          oper := checkrhs[2];
2329          ufr := checkrhs[3];
2330          if not (oper = `*`) then return 0; end
                if;
2331        else
2332          return 0;
2333        end if;
2334        if nops(ufr) = 1 and ufl1 = ufr[1] and
              ufl2 = ufr[1] then
2335          return 11;
2336        else
2337          return 0;
2338        end if;
2339      end proc;
2340
2341      GeneralDAlembertfe := proc(fe, func, var)
2342        local nfe, nfunc, nvar, checklplus,
              rhsfe1, checklminus, rhsfe2, checkrhs,
              oper;
2343        nfe, nfunc, nvar := fe, func, var;
2344        checklplus := checkcfxplusy(nfe, nvar);
2345        if checklplus <> 0 then
2346          rhsfe1 := checklplus[3];
2347        else
```

```
2348          return 0;
2349        end if;
2350        checklminus := checkfxminusy(-rhsfe1,
              nvar);
2351        if checklminus <> 0 then
2352          rhsfe2 := checklminus[3];
2353        else
2354          return 0;
2355        end if;
2356        checkrhs := checkrhsCauchyPexider(rhsfe2,
              nvar);
2357        if checkrhs <> 0 then
2358          oper := checkrhs[2];
2359          if not (oper = `*`) then
2360            return 0;
2361          end if;
2362        else
2363          return 0;
2364        end if;
2365        return 12;
2366      end proc;
2367
2368      Trigonometric1fe := proc(fe, func, var)
2369        local nfe, nfunc, nvar, checklplus,
              rhsfe, ufl, flag, term1, term2, temp,
              check1, oper, ufr1, check2, ufr2;
2370        nfe, nfunc, nvar := fe, func, var;
2371        checklplus := checkcfxplusy(nfe, nvar);
2372        if checklplus <> 0 then
2373          rhsfe := checklplus[3];
2374          ufl := checklplus[4];
2375        else
2376          return 0;
2377        end if;
2378        flag := 0;
2379        if type(rhsfe, `+`) and nops(rhsfe) = 2
              then
2380          term1 := op(1, rhsfe);
2381          term2 := op(2, rhsfe);
2382          if nops(term1) = 2 and nops(term2) = 3
                then
2383            flag := 1;
```

```
2384          elif nops(term1) = 3 and nops(term2) =
                 2 then
2385            temp := term1;
2386            term1 := term2;
2387            term2 := temp;
2388            flag := 1;
2389          end if;
2390          if flag <> 1 then
2391            return 0;
2392          end if;
2393          check1 := checkrhsCauchyPexider(term1,
                 nvar);
2394          if check1 <> 0 then
2395            oper := check1[2];
2396            ufr1 := check1[3];
2397            if not (oper = `*`) then return 0;
                   end if;
2398          else
2399            return 0;
2400          end if;
2401          if op(1, term2) = -1 then
2402            term2 := -term2;
2403          else
2404            return 0;
2405          end if;
2406          check2 := checkrhsCauchyPexider(term2,
                 nvar);
2407          if check2 <> 0 then
2408            oper := check2[2];
2409            ufr2 := check2[3];
2410            if not (oper = `*`) then
2411              return 0;
2412            end if;
2413          else
2414            return 0;
2415          end if;
2416        else
2417          return 0;
2418        end if;
2419        if nops(ufr1) = 1 and nops(ufr2) = 1 and
              ufl = ufr1[1] then
2420          return 13.1;
```

```
2421        else
2422          return 0;
2423        end if;
2424      end proc;
2425
2426      Trigonometric2fe := proc(fe, func, var)
2427        local nfe, nfunc, nvar, checklminus,
              rhsfe, ufl, term1, term2, temp,
              check1, oper, ufr1, check2, ufr2;
2428        nfe, nfunc, nvar := fe, func, var;
2429        checklminus := checkcfxminusy(nfe, nvar);
2430        if checklminus <> 0 then
2431          rhsfe := checklminus[3];
2432          ufl := checklminus[4];
2433        else
2434          return 0;
2435        end if;
2436        if type(rhsfe, '+') and nops(rhsfe) = 2
             then
2437          term1 := op(1, rhsfe);
2438          term2 := op(2, rhsfe);
2439          if nops(term1) = 2 and nops(term2) = 2
               then
2440            check1 :=
                  checkrhsCauchyPexider(term1, nvar);
2441            if check1 <> 0 then
2442              oper := check1[2];
2443              ufr1 := check1[3];
2444              if not (oper = '*') then return 0;
                    end if;
2445            else
2446              return 0;
2447            end if;
2448            check2 :=
                  checkrhsCauchyPexider(term2, nvar);
2449            if check2 <> 0 then
2450              oper := check2[2];
2451              ufr2 := check2[3];
2452              if not (oper = '*') then return 0;
                    end if;
2453            else
2454              return 0;
```

```
2455          end if;
2456        else
2457          return 0;
2458        end if;
2459      else
2460        return 0;
2461      end if;
2462      if nops(ufr1) = 1 and nops(ufr2) = 1 and
             (ufl = ufr1[1] or ufl = ufr2[1]) then
2463        return 13.2;
2464      else
2465        return 0;
2466      end if;
2467    end proc;
2468
2469    Trigonometric3fe := proc(fe, func, var)
2470      local nfe, nfunc, nvar, checklplus,
             rhsfe, ufl, term1, term2, check1,
             oper, ufr1, check2, ufr2;
2471      nfe, nfunc, nvar := fe, func, var;
2472      checklplus := checkcfxplusy(nfe, nvar);
2473      if checklplus <> 0 then
2474        rhsfe := checklplus[3];
2475        ufl := checklplus[4];
2476      else
2477        return 0;
2478      end if;
2479      if type(rhsfe, `+`) and nops(rhsfe) = 2
           then
2480        term1 := op(1, rhsfe);
2481        term2 := op(2, rhsfe);
2482        if nops(term1) = 2 and nops(term2) = 2
             then
2483          check1 :=
                checkrhsCauchyPexider(term1, nvar);
2484          if check1 <> 0 then
2485            oper := check1[2];
2486            ufr1 := check1[3];
2487            if not (oper = `*`) then return 0;
                 end if;
2488          else return 0;
2489          end if;
```

```
2490            check2 :=
                   checkrhsCauchyPexider(term2, nvar);
2491            if check2 <> 0 then
2492              oper := check2[2];
2493              ufr2 := check2[3];
2494              if not (oper = `*`) then return 0;
                     end if;
2495            else
2496              return 0;
2497            end if;
2498          else
2499            return 0;
2500          end if;
2501        else
2502          return 0;
2503        end if;
2504        if nops(ufr1) = 2 and nops(ufr2) = 2 and
             evalb(sort(ufr1, lexorder) =
             sort(ufr2, lexorder)) and (evalb(ufl
             in ufr1) or evalb(ufl in ufr2)) then
2505          return 13.3;
2506        else
2507          return 0;
2508        end if;
2509      end proc;
2510
2511      Trigonometric4fe := proc(fe, func, var)
2512        local nfe, nfunc, nvar, checklminus,
             rhsfe, ufl, sign, flag, term1, term2,
             temp, check1, oper, ufr1, check2, ufr2;
2513        nfe, nfunc, nvar := fe, func, var;
2514        checklminus := checkcfxminusy(nfe, nvar);
2515        if checklminus <> 0 then
2516          rhsfe := checklminus[3];
2517          ufl := checklminus[4];
2518          sign := checklminus[5];
2519        else
2520          return 0;
2521        end if;
2522        flag := 0;
2523        if type(rhsfe, `+`) and nops(rhsfe) = 2
             then
```

```
2524          term1 := op(1, rhsfe);
2525          term2 := op(2, rhsfe);
2526          if nops(term1) = 2 and nops(term2) = 3
                 then
2527            flag := 1;
2528          elif nops(term1) = 3 and nops(term2) =
                 2 then
2529            temp := term1;
2530            term1 := term2;
2531            term2 := temp;
2532            flag := 1;
2533          end if;
2534          if flag <> 1 then return 0; end if;
2535          check1 := checkrhsCauchyPexider(term1,
                 nvar);
2536          if check1 <> 0 then
2537            oper := check1[2];
2538            ufr1 := check1[3];
2539            if not (oper = `*`) then return 0;
                   end if;
2540          else
2541            return 0;
2542          end if;
2543          if op(1, term2) = -1 then
2544            term2 := -term2;
2545          else
2546            return 0;
2547          end if;
2548          check2 := checkrhsCauchyPexider(term2,
                 nvar);
2549          if check2 <> 0 then
2550            oper := check2[2];
2551            ufr2 := check2[3];
2552            if not (oper = `*`) then return 0;
                   end if;
2553          else
2554            return 0;
2555          end if;
2556        else
2557          return 0;
2558        end if;
2559        if nops(ufr1) = 2 and nops(ufr2) = 2 and
```

```
          evalb(sort(ufr1, lexorder) =
          sort(ufr2, lexorder)) and (sign = 1
          and ufl = ufr2[1] or sign = 2 and ufl
          = ufr2[2]) then
2560        return 13.4;
2561      else
2562        return 0;
2563      end if;
2564    end proc;
2565
2566    solver := proc(fe, func, var)
2567      local flag; flag := 0;
2568      if CauchyPexiderfe(fe, func, var) <> 0
           then
2569        flag := 1;
2570        if CauchyPexiderfe(fe, func, var)[1] =
             1.1 then
2571          printf("Cauchy equation\n");
2572        elif CauchyPexiderfe(fe, func, var) =
             1.2 then
2573          printf("Cauchy exponential
             equation\n");
2574        elif CauchyPexiderfe(fe, func, var) =
             1.3 then
2575          printf("Cauchy logarithmic
             equation\n");
2576        elif CauchyPexiderfe(fe, func, var) =
             1.4 then
2577          printf("Cauchy multiplicative
             equation\n");
2578        elif CauchyPexiderfe(fe, func, var) in
             [3.1, 3.2, 3.3, 3.4] then
2579          printf("Pexider equation\n");
2580        end if;
2581      end if;
2582      if Jensenfe(fe, func, var) <> 0 then
2583        flag := 1;
2584        printf("Jensen equation\n");
2585      end if;
2586      if Hosszufe(fe, func, var) <> 0 then
2587        flag := 1;
2588        printf("Hosszu equation\n");
```

```
2589        end if;
2590        if Mikusinskife(fe, func, var) <> 0 then
2591          flag := 1;
2592          printf("Mikusinski equation\n");
2593        end if;
2594        if Alternativefe(fe, func, var) <> 0 then
2595          flag := 1;
2596          printf("Alternative equation\n");
2597        end if;
2598        if GeneralLinearfe(fe, func, var) <> 0
              then
2599          flag := 1;
2600          printf("Inhomogeneous linear functional
                 equation of order 1\n");
2601        end if;
2602        if Linearfe(fe, func, var) <> 0 then
2603          flag := 1;
2604          printf("Linear type functional
                 equation\n");
2605        end if;
2606        if LeviCivitafe(fe, func, var) <> 0 then
2607          flag := 1;
2608          printf("Levi-Civita type functional
                 equation\n");
2609        end if;
2610        if Quadraticfe(fe, func, var) <> 0 then
2611          flag := 1;
2612          printf("Quadratic equation\n");
2613        end if;
2614        if DAlembertfe(fe, func, var) <> 0 then
2615          flag := 1;
2616          printf("D'Alembert equation\n");
2617        end if;
2618        if GeneralDAlembertfe(fe, func, var) <> 0
              then
2619          flag := 1;
2620          printf("General D'Alembert equation\n");
2621        end if;
2622        if Trigonometric1fe(fe, func, var) <> 0
              then
2623          flag := 1;
2624          printf("Functional equation for
```

```
            trigonometric functions, addition
            formula for cosine\n");
2625      end if;
2626      if Trigonometric2fe(fe, func, var) <> 0
            then
2627        flag := 1;
2628        printf("Functional equation for
              trigonometric functions, subtraction
              formula for cosine\n");
2629      end if;
2630      if Trigonometric3fe(fe, func, var) <> 0
            then
2631        flag := 1;
2632        printf("Functional equation for
              trigonometric functions, addition
              formula for sine\n");
2633      end if;
2634      if Trigonometric4fe(fe, func, var) <> 0
            then
2635        flag := 1;
2636        printf("Functional equation for
              trigonometric functions, subtraction
              formula for cosine\n");
2637      end if;
2638      if flag = 0 then
2639        printf("The feCategory function can not
              determine the type of input
              functional equation");
2640      end if;
2641    end proc;
2642
2643    fewhattype := proc( eq,{ `funcs` :=
            [f,g,h,k,l],`vars` := [x,y,z,u,v] } )
2644      EQ, FUNCS, VARS := conv(eq, funcs, vars);
2645      EQ := presim(EQ);
2646      if checkcoeff(EQ) = 0 then
2647        error "The feCategory function can not
              determine the type of input
              functional equation with this type
              of coefficients";
2648      end if;
2649      solver(EQ, FUNCS, VARS);
```

```
2650      end proc;
2651
2652      mapindex := proc(typename)
2653        if evalb(typename = "Cauchy") then
2654          return 1.1;
2655        end if;
2656        if evalb(typename = "Cauchy exponential")
             then
2657          return 1.2;
2658        end if;
2659        if evalb(typename = "Cauchy logarithmic")
             then
2660          return 1.3;
2661        end if;
2662        if evalb(typename = "Cauchy
             multiplicative") then
2663          return 1.4;
2664        end if;
2665        if evalb(typename = "Pexider") then
2666          return 2;
2667        end if;
2668        if evalb(typename = "Jensen") then
2669          return 3;
2670        end if;
2671        if evalb(typename = "Hosszu") then
2672          return 4;
2673        end if;
2674        if evalb(typename = "Mikusinski") then
2675          return 5;
2676        end if;
2677        if evalb(typename = "Alternative") then
2678          return 6;
2679        end if;
2680        if evalb(typename ="General linear") then
2681          return 7;
2682        end if;
2683        if evalb(typename = "Linear") then
2684          return 8;
2685        end if;
2686        if evalb(typename = "Levi-Civita") then
2687          return 9;
2688        end if;
```

```
2689        if evalb(typename ="Quadratic") then
2690          return 10;
2691        end if;
2692        if evalb(typename = "D'Alembert") then
2693          return 11;
2694        end if;
2695        if evalb(typename = "General D'Alembert")
               then
2696          return 12;
2697        end if;
2698        if evalb(typename = "Trigonometri") then
2699          return 13;
2700        end if;
2701        return 0;
2702      end proc;
2703
2704      checktype:=proc(fe, func, var, id)
2705        if id = 1.1 and CauchyPexiderfe(fe, func,
               var) <> 0 then
2706          if CauchyPexiderfe(fe, func, var)[1] =
                 1.1 then
2707            return true;
2708          end if;
2709        end if;
2710        if id = 1.2 and CauchyPexiderfe(fe, func,
               var) = 1.2 then
2711          return true;
2712        end if;
2713        if id = 1.3 and CauchyPexiderfe(fe, func,
               var) = 1.3 then
2714          return true;
2715        end if;
2716        if id = 1.4 and CauchyPexiderfe(fe, func,
               var) = 1.4 then
2717          return true;
2718        end if;
2719        if id = 2 and CauchyPexiderfe(fe, func,
               var) <> 0 then
2720          if CauchyPexiderfe(fe, func, var)[1] =
                 1.1 then
2721            return false;
2722          elif CauchyPexiderfe(fe, func, var) in
```

```
            [3.1, 3.2, 3.3, 3.4] then
2723          return true;
2724        end if;
2725      end if;
2726      if id = 3 and Jensenfe(fe, func, var) <>
            0 then
2727        return true;
2728      end if;
2729      if id = 4 and Hosszufe(fe, func, var) <>
            0 then
2730        return true;
2731      end if;
2732      if id = 5 and Mikusinskife(fe, func, var)
            <> 0 then
2733        return true;
2734      end if;
2735      if id = 6 and Alternativefe(fe, func,
            var) <> 0 then
2736        return true;
2737      end if;
2738      if id = 7 and GeneralLinearfe(fe, func,
            var) <> 0 then
2739        return true;
2740      end if;
2741      if id = 8 and Linearfe(fe, func, var) <>
            0 then
2742        return true;
2743      end if;
2744      if id = 9 and LeviCivitafe(fe, func, var)
            <> 0 then
2745        return true;
2746      end if;
2747      if id = 10 and Quadraticfe(fe, func, var)
            <> 0 then
2748        return true;
2749      end if;
2750      if id = 11 and DAlembertfe(fe, func, var)
            <> 0 then
2751        return true;
2752      end if;
2753      if id = 12 and GeneralDAlembertfe(fe,
            func, var) <> 0 then
```

```
2754          return true;
2755        end if;
2756        if id = 13 and (Trigonometric1fe(fe,
               func, var) <> 0 or
               Trigonometric2fe(fe, func, var) <> 0
               or Trigonometric3fe(fe, func, var) <>
               0 or Trigonometric4fe(fe, func, var)
              <> 0) then
2757          return true;
2758        end if;
2759        return false;
2760      end proc;
2761
2762      fetype:= proc(eq, typename, { `funcs` :=
            [f, g, h, k, l],`vars` := [x, y, z, u,
            v] } )
2763        local tname;
2764        tname := convert(typename,string);
2765        INDEX := mapindex(tname);
2766        if(INDEX = 0) then
2767          return "The fetype function can not
                 decide about this given type of
                 functional equation.";
2768        end if;
2769        EQ, FUNCS, VARS := conv(eq, funcs, vars);
2770        EQ := presim(EQ);
2771        if checkcoeff(EQ) = 0 then
2772          error "The fetype function can not
                 decide about type of input
                 functional equation with this type
                 of coefficients";
2773        end if;
2774        checktype(EQ, FUNCS, VARS, INDEX);
2775      end proc;
2776
2777      G := proc(str) Maplets:-Tools:-Set('TB1' =
            str); end proc;
2778
2779      feinfo := proc()
2780        local maplet;
2781        maplet := Maplet(Window('layout' = 'BL1',
             'title' = "Functional Equations"),
```

```
2782          BoxLayout['BL1'](
2783            BoxColumn('halign' = 'left',
2784              BoxRow(RadioButton['RB1']( "Cauchy
                      equation", 'group' = 'FE',
                      Evaluate('function' = 'G("Cauchy
                      functional equation is the
                      functional equation: f(x+y) =
                      f(x)+f(y).
2785
2786    A function f : R -> R is called additive iff
          it satisfies Cauchy's functional equation,
          for all real x, y.
2787
2788    Over the rational numbers, there is a single
          family of solutions f(x) = cx, with
          constant c.
2789    More generally, f(qx) = qf(x) for all
          rational q, and real x.
2790
2791    Over the real numbers, every continuous
          solution f : R -> R has the form f(x)= cx
          for some constant.
2792    Moreover, there are assumptions that, if
          assumed, force f to be f(x) = cx such as:
2793        . f is continuous at a point,
2794        . f is bounded on some interval,
2795        . f is monotonic on some interval.
2796
2797    There exist non-continuous solutions of
          Cauchy functional equation. And the graph
          of discontinuous additive function is
          dense in R^2.
2798
2799    This equation is sometimes referred to as
          Cauchy additive functional equation to
          distinguish it from:
2800        . Cauchy exponential functional equation
              f(x+y)=f(x)f(y),
2801        . Cauchy logarithmic functional equation
              f(xy)=f(x)+f(y),
2802        . and Cauchy multiplicative functional
              equation f(xy)=f(x)f(y).")')))),
```

```
2803                BoxRow(RadioButton['RB2']( "Pexider
                        equation", 'group' = 'FE',
                        Evaluate('function' = 'G("In
                        1903, J.V. Pexider considered
                        the following functional
                        equations:
2804    f(x + y) = g(x) + h(y);
2805    f(x + y) = g(x)h(y);
2806    f(xy) = g(x) + h(y);
2807    f(xy) = g(x)h(y)
2808    for all real x, y, with f, g, h are unknown
            functions.
2809    These functional equations are
            generalizations of Cauchy functional
            equations. He has solved each of these
            functional equations for continuous real
            functions.
2810
2811    The general solution f, g, h of the
            functional equation f(x + y) = g(x) + h(y)
            is given by
2812    f = A + a + b
2813    g = A + a
2814    h = A + b
2815    where A is an additive function and a, b are
            arbitrary real constants.
2816
2817    The general solution f, g, h of the
            functional equation f(x + y) = g(x)h(y) is
            given by
2818    f = a.b.E
2819    g = a.E
2820    h = b.E
2821    where E is exponential and a, b are nonzero
            constants; together with the trivial
            solutions
2822    f = 0
2823    g = 0
2824    h = arbitrary
2825    and
2826    f = 0
2827    g = arbitrary
```

```
2828    h = 0.") ')))),
2829                BoxRow(RadioButton['RB3'](  "Jensen
                        equation", 'group' = 'FE',
                        Evaluate( 'function' = 'G("The
                        following functional equation
2830    f((x+y)/2) = (f(x) + f(y))/2
2831    is called the Jensen functional equation.
2832
2833    The function f : R -> R satisfies the Jensen
            functional equation, if and only if
2834    f(x) = A(x) + a
2835    where A : R -> R is an additive function and
            a is an arbitrary constant.
2836
2837    Let D be a subset of R^N and D be a convex
            set such that int D is not empty. A
            function f : D -> R is a continuous
            solution of Jensen functional equation if
            and only if
2838    f(x) = cx + a
2839    with certain constants c in R^N and a in R.
2840
2841    The general solution of Jensen Functional
            Equation for all x, y in [a, b] is given by
2842    f(x) = A ((x-a)/(b-a)) + c
2843    where c is an arbitrary constant and A : R ->
            R is an additive function.
2844
2845    A function f : R -> R is said to be convex if
            and only if it satisfies the inequality
2846    f((x+y)/2) <= (f(x) + f(y))/2, for all real
            x,y.
2847
2848    Convex functions were first introduced by
            J.L.W.V. Jensen in 1905, although
            functions satisfying the Jensen inequality
            had been treated by Hadamard (1893) and
            Holder (1889)
2849    ") ')))),
2850                BoxRow(RadioButton['RB4'](  "Hosszu
                        equation", 'group' = 'FE',
                        Evaluate( 'function' = 'G("The
```

```
                functional equation
2851    f(x + y - xy) + f(xy) = f(x) + f(y)
2852    is referred to as the Hosszu equation.
2853
2854    Let a function f : R -> R satisfy Hosszu
            equation.
2855    Then there exist an additive function g : R
            -> R and a real constant a such that
2856    f(x) = g(x) + a
2857
2858    For functions f : R -> R the Hosszu equation
            and the Jensen equation are equivalent.
2859
2860    The Hosszu functional equation was mentioned
            for the first time by M. Hosszu at the
            International Symposium on Functional
            Equations (ISFE) held in Zakopane (Poland)
            in October 1967.
2861
2862    ")'))),
2863                BoxRow(RadioButton['RB5'](
                        "Mikusinski equation", 'group' =
                        'FE', Evaluate('function' =
                        'G("At the Symposium on
                        Functional and Differential
                        Equations held in Zawoja
                        (Poland) in October 1971, J.
                        Mikusinski mentioned the equation
2864    f(x + y)[f(x + y) - f(y) - f(x)] = 0
2865    which since has been named after him.
2866
2867    If a function f : R^N -> R satisfies
            Mikusinski equation, then it is
            additive.")'))),
2868                BoxRow(RadioButton['RB6'](
                        "Alternative equation", 'group'
                        = 'FE', Evaluate('function' =
                        'G("If the function f : R^N -> R
                        is additive, then it satisfies
                        also
2869    [f(x + y)]^2 = [f(x) + f(y)]^2.
2870    The above equation yields immediately only
```

```
        either f(x + y) = f(x) + f(y), or f(x + y)
        = -[f(x) + f(y)].
2871    Since mentioned equation has the form of an
        alternative, it is often referred to as an
        alternative functional equation.
2872
2873    If a function f : R^N -> R satisfies
        alternative equation , then it is additive.
2874    ")')))),
2875                BoxRow(RadioButton['RB7'](
                        "Inhomogeneous linear functional
                        equation of order 1", 'group' =
                        'FE', Evaluate('function' =
                        'G("The inhomogeneous linear
                        functional equation of order 1
                        has the following form
2876    f(ax + by + c) = Af(x) + Bf(y) + C, abAB <> 0.
2877
2878    The Cauchy equation is the particular case a
        = b = A = B = 1, c = 0, C = 0. And the
        Jensen equation corresponds to a = b = A =
        B = 1/2, c = 0, C = 0.
2879
2880    If A + B <> 1, then function f(x) = - C/(A +
        B - 1) is the only constant solution of
        the general linear equation.
2881
2882    If A + B = 1 and C = 0, then every constant
        function f : R^N -> R satisfies the
        general linear equation.
2883
2884    If A + B = 1 and C <> 0, then the general
        linear equation has no constant solution.
2885
2886    If the nonconstant function f satisfies the
        general linear equation, then the function
        g(x) = f(x) - f(0) satisfies the Cauchy's
        functional equation g(x+y) = g(x) + g(y).
2887    ")')))),
2888            BoxColumn('halign'='left',
2889              BoxRow(RadioButton['RB8']( "Linear
                        type functional equation",
```

```
                    'group' = 'FE',
                    Evaluate('function' = 'G("Linear
                    functional equation is of the
                    type
2890    sum(f_i(p_i.x + q_i.y)) = 0 (i = 0..n+1 and
           x, y in X)
2891    where n is a positive integer, p_0, . . . ,
           p_n+1 and q_0, . . . , q_n+1 are rational
           numbers, X, Y are linear spaces and f_0, .
           . . , f_n+1 : X -> Y are unknown functions.
2892    ") ')))),
2893                BoxRow(RadioButton['RB9'](
                    "Levi-Civita type functional
                    equation", 'group' = 'FE',
                    Evaluate('function' =
                    'G("Levi-Civita type functional
                    equations, i.e. the class
2894    f(x + y) = sum(g_i(x).h_i(y)) (i=1..n)
2895    where n is a positive integer, G is an
           Abelian group and f, g_i, h_i : G -> C (i
           = 1, 2, . . . , n) are unknown functions.
2896    ") ')))),
2897                BoxRow(RadioButton['RB10'](
                    "Quadratic equation", 'group' =
                    'FE', Evaluate('function' =
                    'G("The following functional
                    equation is known as the
                    quadratic functional equation
2898    f(x + y) + f(x - y) = 2f(x) + 2f(y).
2899
2900    Let f : R -> R be a function that satisfies
           the quadratic functional equation. Then f
           is rationally homogeneous of degree 2.
           Moreover on the set of rational numbers Q,
           f has the form
2901    f(r) = c.r^2
2902    for rational number r , where c is an
           arbitrary constant.
2903
2904    The general continuous solution of the
           quadratic functional equation is given by
2905    f(x) = c.x^2, where c is an arbitrary
```

```
        constant.") ')))),
2906            BoxRow(RadioButton['RB11'](
                    "D'Alembert equation", 'group' =
                    'FE', Evaluate('function' =
                    'G("The well-known trigonometric
                    identity
2907    cos(x + y) + cos(x - y) = 2cos(x)cos(y)
2908    implies the functional equation
2909    f(x + y) + f(x - y) = 2f(x)f(y)
2910    which is known as the D'Alembert functional
        equation.
2911
2912    Let f : R -> R be continuous and satisfy
        D'Alembert equation. Then f is of the form
2913    f(x) = 0;
2914    f(x) = 1;
2915    f(x) = cosh(ax);
2916    f(x) = cos(bx);
2917    where a,b are arbitrary real constants.
2918    ") ')))),
2919            #BoxRow(RadioButton['RB12'](
                    "General D'Alembert equation",
                    'group' = 'FE',
                    Evaluate('function' = 'G("info
                    of General D\'Alembert
                    equation") ')))),
2920            BoxRow(RadioButton['RB13'](
                    "Functional equation for
                    trigonometric functions",
                    'group' = 'FE',
                    Evaluate('function' = 'G("The
                    trigonometric functions f(x) =
                    cos x and g(x) = sin x satisfy
                    the following functional
                    equations
2921    f(x + y) = f(x)f(y) - g(x)g(y)
2922    f(x - y) = f(x)f(y) + g(x)g(y)
2923    g(x + y) = g(x)f(y) + f(x)g(y)
2924    g(x - y) = g(x)f(y) - g(y)f(x)
2925    Hence these functional equations are called
        trigonometric functional equations.
2926
```

```
2927    If a function f : R -> C satisfies with some
           g : R -> C the functional equation
2928    f(x - y) = f(x)f(y) + g(x)g(y)
2929    and is not constant, then it satisfies also
           the d'Alembert equation
2930    f(x + y) + f(x - y) = 2f(x)f(y)
2931
2932    The continuous solution of the functional
           equation
2933    f(x - y) = f(x)f(y) + g(x)g(y)
2934    is given by
2935    f(x) = c, g(x) = sqrt(c(1 - c))
2936    f(x) = c, g(x) = -sqrt(c(1 - c))
2937    f(x) = cos(ax), g(x) =  sin(ax)
2938    f(x) = cos(ax), g(x) =  -sin(ax)
2939    where a and c are arbitrary constants.
2940
2941    Let f, g : R -> C satisfy the functional
           equation
2942    f(x + y) = f(x) g(y) + f(y) g(x).
2943    Then f and g are of the form
2944    f(x) = 0, g arbitrary;
2945    f(x) = A(x).E(x), g(x) = E(x);
2946    f(x) = (E1(x) - E2(x))/(2a), g(x) =
           (E1(x)+E2(x))/2;
2947    where E1, E2 : R -> C* are exponential
           functions, A : R -> C is an additive
           function, and a is a nonzero complex
           constant.")')),
2948                  Button("OK", Shutdown())),
2949               BoxColumn(BoxRow(TextBox['TB1']("",
                     'editable' = 'false', 'width'= 50

2950        ))))), ButtonGroup['FE']());
2951         Maplets[Display](maplet):
2952       end proc;
2953    end module;
2954 end module;
```

Appendix C

The package MConvexHull

```
1  MConvexHull := module()
2    option package;
3    local P, n, f_plot2d, draw2d, f_plot3d,
        draw3d, TrangleList, TetrahedronList,
        PointPosition, f_plot2dwithpoint,
        f_plot3dwithpoint;
4    export mcvhull2d, mcvhull3d,
        PointInMCVHull2d, PointInMCVHull3d;
5    with(plots);
6    with(plottools);
7    with(ListTools);
8    with(LinearAlgebra);
9
10   f_plot2d := proc(m)
11     local f, i, j;
12     f := (i, j, m) -> display(polygon([P[i],
          m*(P[i] + P[j])/(m + 1), P[j], [0, 0]]));
13     display(seq(seq(f(i, j, m), j = i + 1 ..
          n), i = 1 .. n - 1),scaling =
          constrained, axes = none);
14   end proc;
15
16   draw2d := proc(L, M1, M2, dfr)
17     local p, k, m, m1, m2;
18     P := L;
19     m1 := convert(M1, rational, exact);
```

```
20      m2 := convert(M2, rational, exact);
21      if not type(P, list) then error "Invalid
           set of points."; end if;
22      n := numelems(P);
23      P := MakeUnique(P);
24      if numelems(P)<n then
25        WARNING("There are duplicate points in
             the set of points.");
26        n:= numelems(P);
27      end if;
28      if n < 2 then
29        error "Invalid input, the set of points
             needs at least 2 elements.";
30      end if;
31      for p in P do
32        if not type(p, list) then
33          error "Invalid set of points.";
34        end if;
35        if nops(p)<>2 then
36          error "Invalid set of points.";
37        end if;
38        if not type(evalf(p[1]), numeric) then
39          error "Invalid set of points.";
40        end if;
41        if not type(evalf(p[2]), numeric) then
42          error "Invalid set of points.";
43        end if;
44      end do;
45      for k to n do
46        if P[k][1] = 0 and P[k][2] = 0 then
47          error "The list of points can not
               contain the origin.";
48        end if;
49        if P[k][1] < 0 or P[k][2] < 0 then
50          error "All input points have to have
               non-negative coordinates.";
51        end if;
52      end do;
53      if not type(evalf(m1), numeric) then
54        error "Invalid input, m value.";
55      end if;
56      if not type(evalf(m2), numeric) then
```

```
57        error "Invalid input, m value.";
58      end if;
59      if not type(dfr, integer) then
60        error "Invalid number of frames.";
61      end if;
62      if m1 < 0 or 1 <= m1 then
63        if m1 <> m2 then
64          error "Invalid m1, 0<= m1 <1.";
65        else
66          error "Invalid m, 0<= m <1.";
67        end if;
68      end if;
69      if m2 < 0 or 1 <= m2 then
70        error "Invalid m2, 0<= m2 <1.";
71      end if;
72      if m2 < m1 then
73        error "Invalid input, 0<= m1<= m2 <1.";
74      end if;
75      if m1 = m2 then
76        if dfr <> 1 then
77          error "Invalid number of frames."
78        end if;
79        print(M1);
80        print(f_plot2d(m1));
81      else
82        if dfr < 2 then
83          error "Invalid number of frames."
84        end if;
85        animate(f_plot2d, [m], m = m1 .. m2,
              frames = dfr);
86      end if;
87    end proc;
88
89    mcvhull2d:= proc()
90      if nargs = 1 then
91        draw2d(args,0,0.9,6);
92      elif nargs = 2 then
93        draw2d(args, args[2],1);
94      elif nargs = 3 then
95        draw2d(args,6);
96      elif nargs = 4 then
97        draw2d(args);
```

```
98      else
99        error "Invalid input parameters.";
100     end if;
101   end proc;
102
103   f_plot3d := proc(m)
104     local f, i, j, k, mF;
105     mF := m/(m+1);
106     f := (i, j, k, mF) ->
          display({tetrahedron([P[i], mF*(P[i] +
          P[j]), mF*(P[i] + P[k]), [0, 0, 0]]),
          tetrahedron([P[j], mF*(P[i] + P[j]),
          mF*(P[j] + P[k]), [0, 0, 0]]),
          tetrahedron([P[k], mF*(P[i] + P[k]),
          mF*(P[j] + P[k]), [0, 0, 0]]),
          tetrahedron([mF*(P[i] + P[j]), mF*(P[i]
          + P[k]), mF*(P[j] + P[k]), [0, 0, 0]])});
107     display(seq(seq(seq(f(i, j, k, mF), k = j +
          1 .. n), j = i + 1 .. n - 1), i = 1 .. n
          - 2),scaling=constrained, axes=none);
108   end proc;
109
110   draw3d := proc(L, m1, m2, dfr)
111     local p, k, m;
112     P := L;
113     if not type(P, list) then
114       error "Invalid set of points.";
115     end if;
116     n := numelems(P);
117     P := MakeUnique(P);
118     if numelems(P)<n then
119       WARNING("There are duplicate points in
            the set of points.");
120       n:= numelems(P);
121     end if;
122     if n<3 then
123       error "Invalid input, the set of points
            needs at least 3 elements.";
124     end if;
125     for p in P do
126       if not type(p, list) then
127         error "Invalid set of points.";
```

```
128       end if;
129       if nops(p)<>3 then
130         error "Invalid set of points.";
131       end if;
132       if not type(evalf(p[1]), numeric) then
133         error "Invalid set of points.";
134       end if;
135       if not type(evalf(p[2]), numeric) then
136         error "Invalid set of points.";
137       end if;
138       if not type(evalf(p[3]), numeric) then
139         error "Invalid set of points.";
140       end if;
141     end do;
142     for k to n do
143       if P[k][1] = 0 and P[k][2] = 0 and
             P[k][3] = 0 then
144         error "The list of points can not
               contain the origin.";
145       end if;
146       if P[k][1] < 0 or P[k][2] < 0 or P[k][3]
            < 0 then
147         error "All input points have to have
               non-negative coordinates";
148       end if;
149     end do;
150     if not type(evalf(m1), numeric) then
151       error "Invalid input, m value.";
152     end if;
153     if not type(evalf(m2), numeric) then
154       error "Invalid input, m value.";
155     end if;
156     if not type(dfr, integer) then
157       error "Invalid number of frames.";
158     end if;
159     if m1 < 0 or 1 <= m1 then
160       if m1 <> m2 then
161         error "Invalid m1, 0<= m1 <1.";
162       else
163         error "Invalid m, 0<= m <1.";
164       end if;
165     end if;
```

```
166     if m2 < 0 or 1 <= m2 then
167       error "Invalid m2, 0<= m2 <1.";
168     end if;
169     if m2 < m1 then
170       error "Invalid input, 0<= m1<= m2 <1.";
171     end if;
172     if m1 = m2 then
173       if dfr <> 1 then
174         error "Invalid number of frames."
175       end if;
176       print(m1);
177       print(f_plot3d(m1));
178     else
179       if dfr < 2 then
180         error "Invalid number of frames."
181       end if;
182       animate(f_plot3d, [m], m = m1 .. m2,
            frames = dfr);
183     end if;
184   end proc;
185
186   mcvhull3d:= proc()
187     if nargs = 1 then
188       draw3d(args,0,0.9,6);
189     elif nargs = 2 then
190       draw3d(args,args[2],1);
191     elif nargs = 3 then
192       draw3d(args,5);
193     elif nargs = 4 then
194       draw3d(args);
195     else
196       error "Invalid input parameters.";
197     end if;
198   end proc;
199
200   TrangleList := proc(P,m)
201     local mR, mF, T, T1, T2, i, j;
202     T := [];
203     mR := convert(m, rational, exact);
204     mF := mR/(mR+1);
205     for i to nops(P) - 1 do
206         for j from i + 1 to nops(P) do
```

```
207             T1 := [P[i], mF*(P[i] + P[j]), [0,
                    0]];
208             T2 := [P[j], mF*(P[i] + P[j]), [0,
                    0]];
209             T := [op(T), T1, T2];
210           end do;
211     end do;
212     return T;
213   end proc;
214
215   TetrahedronList := proc(P,m)
216     local mR, mF, T, T1, T2, T3, T4, i, j, k;
217     T := [];
218     mR := convert(m, rational, exact);
219     mF := mR/(mR+1);
220     for i to nops(P) - 2 do
221         for j from i + 1 to nops(P)-1 do
222           for k from j+1 to nops(P) do
223             T1 := [P[i], mF*(P[i] + P[j]),
                    mF*(P[i] + P[k]), [0, 0, 0]];
224             T2 := [P[j], mF*(P[i] + P[j]),
                    mF*(P[j] + P[k]), [0, 0, 0]];
225             T3 := [P[k], mF*(P[i] + P[k]),
                    mF*(P[j] + P[k]), [0, 0, 0]];
226             T4 := [mF*(P[i] + P[j]), mF*(P[i] +
                    P[k]), mF*(P[j] + P[k]), [0, 0,
                    0]];
227             T := [op(T), T1, T2, T3, T4];
228         end do;
229         end do;
230     end do;
231     return T;
232   end proc;
233
234   PointPosition:= proc(p, tr)
235     local i, M, u, A, T;
236     A:=p;
237     T:=tr;
238     for i to nops(T) do
239       T[i] := [op(T[i]), 1];
240     end do;
241     M := Matrix(T);
```

```
242     M := LinearAlgebra:-Transpose(M);
243     A:=[op(A),1];
244     A:=convert(A, Vector);
245     u:=LinearSolve(M,A);
246     for i from 1 to Dimension(u) do
247       if u[i] < 0 then
248         return 0;
249       end if;
250     end do;
251     return 1;
252   end proc;
253
254   f_plot2dwithpoint := proc(a, P, n, m)
255     local f, i, j;
256     f := (i, j, m) -> display(polygon([P[i],
          m*(P[i] + P[j])/(m + 1), P[j], [0, 0]]));
257     display(pointplot(a, color = blue, symbol =
          soliddiamond, symbolsize = 20),
          seq(seq(f(i, j, m), j = i + 1 .. n), i =
          1 .. n - 1), scaling = constrained, axes
          = none);
258   end proc;
259
260   PointInMCVHull2d := proc(a, P, m, animate)
261     local TL, i, po,flag;
262     flag:=0;
263     TL:=TrangleList(P,m);
264     TL:=TrangleList(P,m);
265     for i from 1 to nops(TL) do
266       po:=PointPosition(a, TL[i]);
267       if po = 1 then
268         flag:=1;
269         break;
270       end if;
271     end do;
272     if (flag=1) then
273       print(True)
274     else
275       print(False);
276     end if;
277     if (animate=true) then
278       f_plot2dwithpoint(a, P, nops(P), m);
```

```
279     end if;
280   end proc;
281
282   f_plot3dwithpoint := proc(a, P, n, m)
283     local f, i, j, k;
284     f := (i, j, k, m) ->
          display({tetrahedron([P[i], m*(P[i] +
          P[j])/(m + 1), m*(P[i] + P[k])/(m + 1),
          [0, 0, 0]]), tetrahedron([P[j], m*(P[i]
          + P[j])/(m + 1), m*(P[j] + P[k])/(m +
          1), [0, 0, 0]]), tetrahedron([P[k],
          m*(P[i] + P[k])/(m + 1), m*(P[j] +
          P[k])/(m + 1), [0, 0, 0]]),
          tetrahedron([m*(P[i] + P[j])/(m + 1),
          m*(P[i] + P[k])/(m + 1), m*(P[j] +
          P[k])/(m + 1), [0, 0, 0]])});
285     display(pointplot3d(a, color = blue, symbol
          = solidbox, symbolsize = 20),
          seq(seq(seq(f(i, j, k, m), k = j + 1 ..
          n), j = i + 1 .. n - 1), i = 1 .. n -
          2), scaling = constrained, axes = none,
          transparency = 1.0);
286   end proc;
287
288   PointInMCVHull3d := proc(a, P, m, animate)
289     local TL, i, po, flag;
290     flag:=0;
291     TL:=TetrahedronList(P,m);
292     for i from 1 to nops(TL) do
293       po:=PointPosition(a, TL[i]);
294       if po = 1 then
295         flag:=1;
296         break;
297       end if;
298     end do;
299     if (flag=1) then
300       print(True)
301     else
302       print(False);
303     end if;
304     if (animate=true) then
305       f_plot3dwithpoint(a, P, nops(P), m);
```

```
306       end if;
307     end proc;
308 end module;
```