TIAGO DE OLIVEIRA CORRÊA

A Domain-Specific Language for Operator Theory

UNIVERSIDADE DO ALGARVE

FACULDADE DE CIÊNCIAS E TECNOLOGIA

2022

TIAGO DE OLIVEIRA CORRÊA

A Domain-Specific Language for Operator Theory

Master Thesis in Informatics Engineering

Work done under the supervision of:

Professor Doctor Paula Cristina Negrão Ventura Martins

Professor Doctor Ana Isabel da Costa Conceição Guerra

UNIVERSIDADE DO ALGARVE

FACULDADE DE CIÊNCIAS E TECNOLOGIA

2022

A Domain-Specific Language for Operator Theory

Declaração de autoria de trabalho/Declaration of authorship

Declaro ser o autor deste trabalho, que é original e inédito. Autores e trabalhos consultados estão devidamente citados no texto e constam da listagem de referências incluída.

I hereby declare to be the author of this work, which is original and unpublished. Authors and works consulted are properly cited in the text and included in the reference list.

_____

(Tiago de Oliveira Corrêa)

Resumo

Recentemente as aplicações conhecidas como sistema de álgebra computacional, CAS, compostas por muitas funções para computação simbólica estão disponíveis ao público em geral. Com esse tipo de aplicação, utilizadores puderam delegar ao computador toda, ou uma parte significativa dos cálculos simbólicos presentes em muitos algoritmos matemáticos. Os modelos matemáticos, que são uma descrição de um sistema usando linguagem e conceitos matemáticos, são muito utilizados nas ciências naturais e engenharia, bem como nas ciências sociais.

O principal objetivo deste trabalho é o de criar uma linguagem textual simples e eficiente para a formalização de modelos matemáticos no domínio das integrais singulares. Por outras palavras, facilitar o trabalho de programação a um não especialista, como é o caso dos matemáticos, quando estão formulando problemas e fazendo uso de linguagens de programação.

A nova linguagem criada, também chamada linguagem de domínio específico (DSL), denominada SIOL, Linguagem para Operadores de Integrais Singulares, foi criada, não apenas para resolução de integrais singulares, mas para oferecer outras informações sobre conceitos da teoria de operadores, não tendo a complexidade que normalmente é encontrada nas linguagens de uso geral. Com recurso ao Xtext e Eclipse, os autores criaram uma linguagem com destaque de sintaxe, verificação de erros e um editor automático para alguma das tarefas da teoria dos operadores, relacionadas com integrais singulares, gerando resultados que usam o *Wolfram Mathematica*.

Palavras-chave: Symbolic Computation, Mathematical Models, Xtext, DSL, Operator Theory

Abstract

Recently, the applications known as computer algebra system, CAS, packed with extensive capabilities of symbolic computation have been available to the general public. With these software applications, users were able to delegated to the computer all, or a significant part, of the symbolic calculations present in many mathematical algorithms. Mathematical models, a description of a system using mathematical concepts and language, are largely used in natural sciences and engineering, as well as in social sciences.

The main goal of this work is to provide a simple and efficient textual language to formalize mathematical models in the domain of singular integrals. In other words, to facilitate the programming task to a non-specialist, like mathematicians, when formulating problems using a computer language.

The new created language, also known as a Domain-Specific Language (DSL), named SIOL, Singular Integral Operator Language, created not only to compute singular integrals but to provide with other information about operator theory concepts, will not have the complexity that is normally found in general-purpose languages. With Xtext and Eclipse, the authors will create a syntax highlighting, error checking and auto-completion editor for some of the operator theory tasks related to the singular integrals, that generates its outputs that uses Wolfram Mathematica.

Keywords: Symbolic Computation; Mathematical Models; Xtext; DSL; Operator Theory.

Acknowledgements

This master thesis work would not have been possible without the constant guidance and support of my two promoters **Professor Doctor Paula Ventura Martins** and **Professor Doctor Ana Conceição**. I have benefited greatly from your wealth of knowledge and meticulous editing. Your invaluable assistance and insights were crucial to the writing of this thesis as well as your friendship during all processes. I am extremely grateful that you took me on as your student and continued to have faith in my work.

I also would like to express my gratitude to the **Faculty of Sciences and Technology of Algarve University** for the given opportunity to perusing this master's degree.

Thanks to my parents, **Marta** and **Pedro**, for all the love and for being there, even far, supporting the idea of living abroad and concluding this project.

To my dear **Tio Toninho** and **Haroldo**, for inspiring me during all my life to become the person I am and to get to where I am today.

My warm and heartfelt thanks to **Natália Lourenço**, in the representation of my Portuguese family, for the countless moments where nothing could have moved ahead without their help.

And to my love, **Tânia Margarida Lourenço**, for being my guardian angel, for never letting me give up the idea of this project. She was there to support me on the complex reviews, understand the stressful days, share my pain of working on a complex project, study on an even more complex project, and live in a totally different culture. For the love, for the caring, for the days where all was dark, YOU were there. This project is ours.

Por tudo isso, Muito Obrigado.

Index

List of Tables

List of Figures

List of Abbreviations

AST – Abstract Syntax Tree

CAS – Computer Algebraic System

DSL – Domain-Specific Language

EMF- Eclipse Model Framework

GAMS – Algebraic Modelling Language

GHS – Glasgow Haskell Computer

GPL – General Processing Language

IDE – Integrated Development Environment

IT – Information Technology

NICTA – National ICT Australia

SIOL – Singular Integral Operator Language

1. Introduction

1.1. Motivation

“Normally when we think of computers, we imagine constructing machines or programs for specific purposes – to perform tasks we want. Certainly, this is what Turing had in mind when he set up Turing machines or discussed how intelligent machines could be built” (Wolfram, 2011). Humans have been trying to develop, for a long time, machines that can assist them with several tasks, including performing calculations and processing data. As populations grew and society became more sophisticated over time, this need to process data increased dramatically. The lack of portability between different old computers, and computational scenarios, led to the need for the development of high-level languages — denominated this way because they permitted the computer programmer to ignore low-level details of a computer's hardware. Further, it was recognized that the closer the syntax, rules, and mnemonics of a programming language could be to the natural language, the less likely it became that the programmer would inadvertently introduce errors into the program (Malik, 1998).

Computer programming languages, in general, are used to create algorithms, which are a precisely defined sequence of rules telling how to produce specific output information. In recent years, computer programming languages called algebraic modelling languages, have been accepted worldwide and have been adopted by all sorts of users as a key feature to develop computer systems, like large-scale optimization problems (Fourer, 1998). In an attempt to formulate mathematical equations, and consequently its algorithms, from given real-world problems, a methodology called mathematical modelling was created to aid mathematicians, physicists, and other scientists (Aris, 1995). Aligned with mathematical modelling concepts and provided with extensive capabilities of symbolic computation, the algebraic modelling languages are available to the general public and are known as computer algebra systems (CAS). They allow delegating to a computer all, or at least a significant part, of the symbolic calculations present in many mathematical algorithms (Heid & Edwards, 2001).

These modelling languages provide the best approach for non-programmers to represent complex problems since no sophisticated programming skills are required. According to a recent survey among operator theory researchers, performed in May 2021, it was verified that they do not include themselves in the computer programming experts’ group. This research also verified that the lack of knowledge on programming can be itself an important barrier (see

chapter 4). So, it is crucial to provide simple and attractive languages to implement their systems without using general and complex programming languages (Martins & Conceição, 2017).

In mathematics, operator theory is concerned with the study of linear operators, usually on vector spaces whose elements are functions and whose vector spaces are usually infinite-dimensional. These calculations normally involve a large number of properties and classification of great complexity operators, which can be a barrier to professionals and researchers using the available general-purpose languages (GPL), when developing new, or even when understanding related algorithms.

1.2. Objective

In software development and domain engineering, a domain-specific language (DSL) is a programming or specification language dedicated to a particular problem domain. In the market exists different kinds of mathematical modelling languages, dedicated to a particular domain, frequently used for describing and solving very complex problems in different mathematical areas of study. As an example, we mention the operational research area, that has used modelling languages to solve combinatorial optimization problems over the years.

Although these languages are very interesting and have great features, they have focused on optimization problems limiting their expressiveness to this purpose of optimization. Considering the focus of our work, the main and more interesting feature of these languages resides in the fact that the model can be written in a style very close to the mathematical one, which is a facilitator to an audience that might not have an experience on common GPL languages (Martins & Conceição, 2017). Despite this, none of the languages could entirely satisfy the needs of this work, when working with problems involving operator theory concepts, in particular with Cauchy type singular integrals such as described in (Conceição et al., 2013), due to the extensive inherent symbolic and numeric calculations on its algorithm. Considering the lack of modelling languages and related tools, that can describe and process the different classes of problems related to singular integrals, we decided to develop a DSL for this particular application domain. Since this is an ongoing project, this work has the potential to be extended to many other problems inside operator theory.

1.3.Contributions

During this work development, a paper was published, and three oral presentations were performed to reinforce the interest of the scientific community:

- Corrêa,T., Conceição, A.C., Martins, P.V. - A domain specific language for operator theory. Proceedings of the 5th International Conference on Numerical and Symbolic Computation. Developments and Applications (ECCOMAS Thematic Conference). A. Loja, M. Bezzeghoud, J.I. Barbosa, J.A. Rodrigues (Eds.), pp. 63-71
- Corrêa,T., Conceição, A.C., Martins, P.V. - A Domain-Specific Language for Operator Theory - International Congress on Interdisciplinarity in Social and Human Sciences, 2016 - Universidade do Algarve, Portugal
- Corrêa,T. A domain-specific language to compute singular integrals - (WOAT 2016) International Workshop on Operator Theory and Operator Algebras, 2016 – Instituto Superior Técnico, Lisbon, Portugal
- Conceição, A.C., Martins, P.V., Tiago Corrêa - The design of operator theory algorithms and the creation of a domain specific language – WOTCA 2021- Workshop on Operator Theory and Complex Analysis – (WOTCA 2021) - Instituto Superior Técnico, Portugal

1.4.Document Structure

The structure of this thesis is presented following a set of steps to create the final artefact. The second chapter is a description of operator theory fundamentals where some existing algorithms are described. Special attention was given to the singular integrals and the [SInt] algorithm since they are the basis of this project. As part of the state of the art, the chapter also includes an explanation of the DSL concept and a comparison of the existing market mathematical languages.

The third chapter describes the investigation methodology used in the project development. The fourth chapter presents the problem concerning the lack of a language dedicated to operator theory.

Following this set of steps, in the fifth chapter, we propose the creation of a DSL as a solution to solve the existing problem. Moreover, the development environment used for its creation is described. In chapter six, a detailed description of the created language is presented.

In the seventh chapter, the newly created algorithms to test the application of the new language are described, and in the eighth chapter, an evaluation of the results obtained during the test of

the new DSL is shown. In the last chapter, conclusions were made and future developments were proposed to continue this work.

This work also contains an extensive and updated bibliography – chapter 9, and two appendixes.

2.State of the Art

This chapter presents the theory involved in the development of the topics of this research, both operator theory and DSL construction. In section 2.1 a brief introduction and explanation of the specific approached operator theory concepts on this research are made, as well as the [SInt] algorithm used as the basis of this project. Section 2.2 introduces the DSL concept and how it was used to reach the solution for the proposed problem in this thesis work and discussed further on. Finally, section 2.3 presents a comparison of the currently existing market solutions for mathematical languages that also serves as a justification for the need of developing a new, more efficient, and operator-oriented language for this purpose.

2.1.Fundamentals of Operator Theory

2.1.1. Introduction

Operator theory is a branch of functional analysis that deals with bounded linear operators and their properties. The operator theory has many applications in several main scientific research areas, such as structural mechanics, aeronautics, quantum mechanics, ecology, probability theory, electrical engineering, among others, and the importance of its study is globally acknowledged (Conceição, 2021).

Some progress has been achieved for some classes of singular integral operators whose properties allow the use of particular strategies. However, the existing algorithms allow, in general, to study the concepts but they are not designed to be implemented on a computer.

In the last years, the computer algebra system *Mathematica* was used to implement on a computer, different analytical algorithms within the operator theory: calculation techniques to compute singular integrals, analytical algorithms for solving integral equations (Conceição, 2007; Conceição et al., 2010), analytical algorithms to factorize scalar and matrix functions (Conceição et al., 2012; Conceição et al., 2010), and more recently analytical algorithms to study the spectrum (Conceição & Pereira, 2016; Conceição & Pereira, 2017) and the kernel (Conceição et al., 2016) of several special classes of singular integral operators.

The design of the algorithms, referenced in this work, is focused on the possibility of implementing on a computer all, or a significant part, of the extensive symbolic and numeric calculations, present in analytical algorithms. Some of the already existing methods rely on

innovative techniques of the operator theory and have the potential for extension to more complex and general problems (Conceição & Pires, 2022).

As the outcome of this work, the computer algebra system *Wolfram Mathematica* was used. This system is a symbolic mathematical computation program, conceived by Stephen Wolfram, used in many scientific, engineering, and computing fields.

2.1.2. Basic Concepts

Considering that $\mathbb{T}$ represents the unit circle in the complex plane, let $\mathbb{T}_+$ and $\mathbb{T}_-$ denote the open unit disk and the exterior region of the unit circle including the ∞, respectively. As usual, $L_\infty(\mathbb{T})$ represents the space of all essentially bounded functions defined on $\mathbb{T}$ and $H_\infty(\mathbb{T})$ the class of all bounded and analytic functions in $\mathbb{T}_+$. Let $\mathcal{R}(\mathbb{T})$ be the algebra of rational functions without poles on $\mathbb{T}$ and $\mathcal{R}_\pm(\mathbb{T})$ the subsets of $\mathcal{R}(\mathbb{T})$ whose elements have no poles in $\mathbb{T}_\pm$, respectively.

It is well known that the singular integral operator with Cauchy kernel, S_Γ, defined almost everywhere, by

$$S_\Gamma\varphi(t) = \frac{1}{\pi i}\int_T \frac{\varphi(\tau)}{\tau - t}\,d\tau,\ t \in T \qquad (1)$$

where the integral is understood in the sense of its principal value, represents a bounded linear operator in the Lebesgue space $L_2(\mathbb{T})$. In addition, S_Γ is a self-adjoint and unitary operator in $L_2(\mathbb{T})$ (Gohberg & Krupnik, 1992). Thus, we can associate with S_Γ two complementary Cauchy projection operators

$$P_\pm = (I \pm S_\Gamma) / 2 \qquad (2)$$

where I represents the identity operator.

The projections (2) allow us to decompose the algebra $\mathcal{R}(\mathbb{T})$ in the topological direct sum

$$\mathcal{R}(\mathbb{T}) = \mathcal{R}_+(\mathbb{T}) \oplus \mathcal{R}^0_-(\mathbb{T}) \qquad (3)$$

Where $\mathcal{R}_+(\mathbb{T}) = P_+\mathcal{R}(\mathbb{T})$ and $\mathcal{R}^0_-(\mathbb{T}) = P_-\,\mathcal{R}(\mathbb{T})$. We also have $\mathcal{R}_-(\mathbb{T}) = \mathcal{R}^0_-(\mathbb{T}) \oplus \mathbb{C}$

2.1.3. [SInt] Algorithm

As the basis for this project, the [SInt] (Conceição et al., 2013) algorithm was used. This algorithm is capable of computing some classes of Cauchy-type singular integrals defined on the unit circle.

There exist several numerical algorithms and approximation methods for evaluating some classes of singular integrals. There are also several analytical techniques that allow the exact computation of singular integrals for particular cases. However, the [SInt] and the [SIntAFact] algorithms (Conceição et al., 2013), up to our knowledge, are the only analytical algorithm designed and implemented for computing singular integrals with essentially bounded functions defined on the unit circle. Both algorithms were implemented using the numeric and symbolic capabilities of the computer algebra system *Mathematica*. In particular, the implementation of the [SInt] algorithm, makes the results of lengthy and complex calculations available in a simple way to researchers of different areas (Conceição & Pires, 2022).

The kind of algorithms like [SInt] is of big importance to the design of factorization and spectral algorithms. The reason why this algorithm was chosen was because of the relevance of its steps to many operator theory-related algorithms. These steps can be organized in a way to compose other interesting algorithms, by reusing the already created procedures.

The [SInt] algorithm computes (1) when we can represent the function φ as

$$\varphi(t) = \mathrm{r}(\mathrm{t})[x_+(t) + y_-(t)] \tag{4}$$

where x_+, $\overline{y_-} \in H_\infty(\mathbb{T})$, where the overline denotes the complex conjugate of y_- in the unit circle, and $r \in \mathcal{R}(\mathbb{T})$.

The algorithm extensively uses the properties of the projection operators (2), that emerge when those operators are applied to function in $H_\infty(\mathbb{T})$, (e.g., x_+), and in $\overline{H_\infty(\mathbb{T})}$, where the overline denotes the complex conjugate of $H_\infty(\mathbb{T})$, (e.g., y_-). The [SInt] explores the rationality of *r(t)* reducing all possible situations to a few basic cases. After the decomposition of the rational function *r(t)* in elementary fractions, the singular integrals are computed using the formulas described in (Conceição et al., 2013).

Figure 2-1 illustrates the design of this algorithm where the auxiliary functions *X(t)* and *Y(t)* are defined as $X(t) = r(t)x_+(t)$ and $Y(t) = r(t)y_-(t)$.

For each input of functions *r(t)*, $x_+(t)$ and $y_-(t)$ the [SInt] algorithm computes the singular integrals $S_r X(t)$ and $S_r Y(t)$, i.e., compute the singular integral $S_r\varphi(t)$.

It must be noted that the user may choose not to assign any particular expression to the input functions $x_+(t)$ and $y_-(t)$. In this case, the obtained singular integrals are general functions of x_+ and/or y_-.

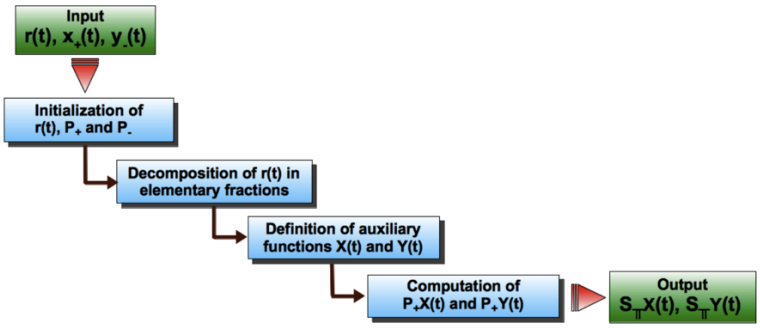


Figure 2-1- *Flowchart of the [SInt] algorithm*

On the output of the algorithm, the singular integrals $S_\tau X(t)$ and $S_\tau Y(t)$ are defined as expressions in closed form and therefore, can be used in further calculations just like any other function in *Mathematica* (Conceição & Pires, 2022).

When making use of the [SInt] algorithm, the user must be aware that, it is up to his responsibility to provide a rational function r belonging to the $\mathcal{R}(\mathbb{T})$, a function x_+ in $H_\infty(\mathbb{T})$, and a function y_- that has its conjugate in $H_\infty(\mathbb{T})$. If a non-valid input is considered, [SInt] outputs an error. Furthermore, since the poles of r are crucial information for this calculation technique, the success of the [SInt] algorithm is dependent on the possibility of finding these poles by solving a polynomial equation. For instance, if a rational function r is provided with a fifth-degree or higher polynomials, the algorithm will most of the time give an incorrect output since it cannot apply the properties of the projectors.

2.1.4. [SInt] Example

This subsection presents some examples of non-trivial singular integrals computed with [SInt](Conceição & Pires, 2022). For each input of functions $r(t)$, $x_+(t)$, and $y_-(t)$, the [SInt] algorithm computes the singular integrals $S_\tau X(t)$ and $S_\tau Y(t)$, for $X(t)=r(t)x_+(t)$ and $Y(t)=r(t)y_-(t)$. It is important to remember that the user has the responsibility to introduce valid r, x_+ and y_-.

The example on **Figure2-2** uses the [SInt] algorithm to input the rational function r and considers x_+ as a general expression and $y_-(t) = t^{-k}$.

```
SetDirectory[NotebookDirectory[]];

Clear[Evaluate[Context[] <> "*"]];

optR = 1;

R[t_] = (1 + 10 t^4) / ((t - 4)^2 t^3)
```

Figure 2-2 *– Part of the [SInt] algorithm indicating the input of the rational function r(t).*

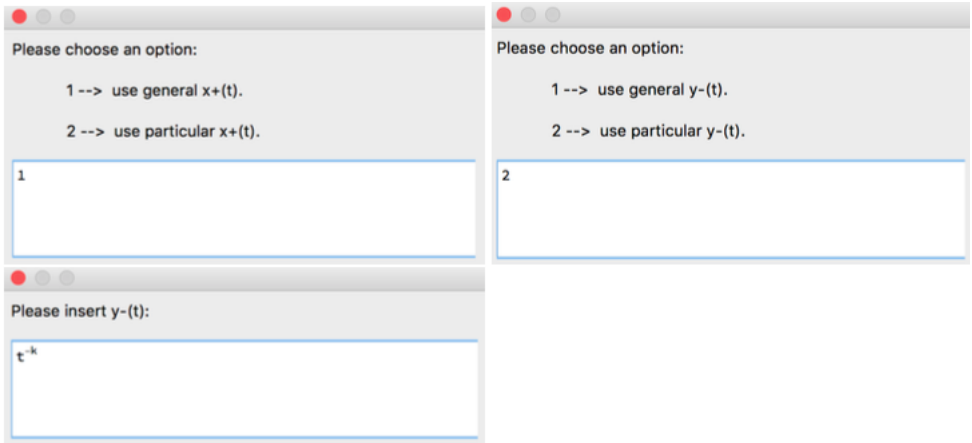


Figure 2-3 *- Part of the structure of the [SInt] algorithm indicating the Input of the auxiliary functions x+ and y_.*

$$S_{\mathbb{T}}X(t) = \frac{1}{128\,(-4+t)^2\,t^3}\left(128\left(1+10\,t^4\right)x_+[t] - (-4+t)^2\left(\left(16+8\,t+3\,t^2\right)x_+[0] + 8\,t\left((2+t)\,x_+'[0] + t\,x_+''[0]\right)\right)\right)$$

$$S_{\mathbb{T}}Y(t) = \frac{2^{-7-2k}\,t^{-3-k}\left(-2^{7+2k} - 5\times 4^{4+k}\,t^4 + 4\,(4+2561\,k)\,t^{3+k} + (2557-2561\,k)\,t^{4+k}\right)}{(-4+t)^2}$$

Figure 2-4 *- Output given by [SInt] algorithm*

After the execution, the [SInt] will give as an output, the singular integrals presented in **Figure 2-4**. In **Figure 2-2**, the [SInt] algorithm considers k ≥ 0 and considers that in the input, $\overline{y_-} \in H_\infty(\mathbb{T})$, that allows the user to include arbitrary constants, and the outputs visible on **Figure 2-4**.

During the development of this work, it was possible to improve the existing [SInt], discovering the position of the poles, avoiding errors on the outputs (when a non-valid function is input) and showing the user the proper error message.

Recently, a new algorithm was developed and named $[SInt]_{2.0}$ (Conceição & Pires, 2022), which was created also to improve the original [SInt]. This improved version was created directly on *Mathematica.* The idea behind this project is different, since it aims to allow users

to directly create improvements using another approach involving a language using new simpler coding syntax to create and improve algorithms related to the operator theory.

2.1.5. Other operator theory algorithms

Besides the [SInt], other algorithms were used as inspiration for this work. The explicit rational function factorization algorithms [ARFact-Scalar] and matrix [ARFact-Matrix] can compute explicit factorizations of given non-singular rational matrix functions defined on the unit circle and can be used as a basis to design new operator theory algorithms.

Both algorithms were also implemented using the CAS *Mathematica*. The design of new analytical methods, even if only for some restricted special classes of functions, is still very significant to the development of such a theory. The created [ARFact-Scalar] algorithm always computes the factorization index of any considered factorable scalar rational function defined on the unit circle (Conceição, 2020). On the other hand, the generalized factorization algorithm [AFact] uses the inner-outer factorization concept (Conceição A.C., 2020)

On the study of the kernel of an operator, the importance of the factorization theory is also well known (Gohberg & Krupnik, 1992; Litvinchuk & Spitkovskii, 1987). It has already been demonstrated that the generalized factorization concept has relations with estimations of the dimension of the kernel of some classes of singular integral operators with the non-Carleman shift. It has also already been described how the use of algorithms like [AFact], [ARFact-Matrix], and [SInt] (Conceição, 2021) can estimate the dimension of the kernel of some classes of singular integral operators.

The analytical algorithms [ADimKerPaired-Scalar], [AKerPaired-Scalar], and [ADimKerPaired-Matrix] described in (Conceição, 2021), were designed reusing features of the already existing algorithm [ARFact-Scalar] and [ARFact-Matrix] (**Figure 2-5**).

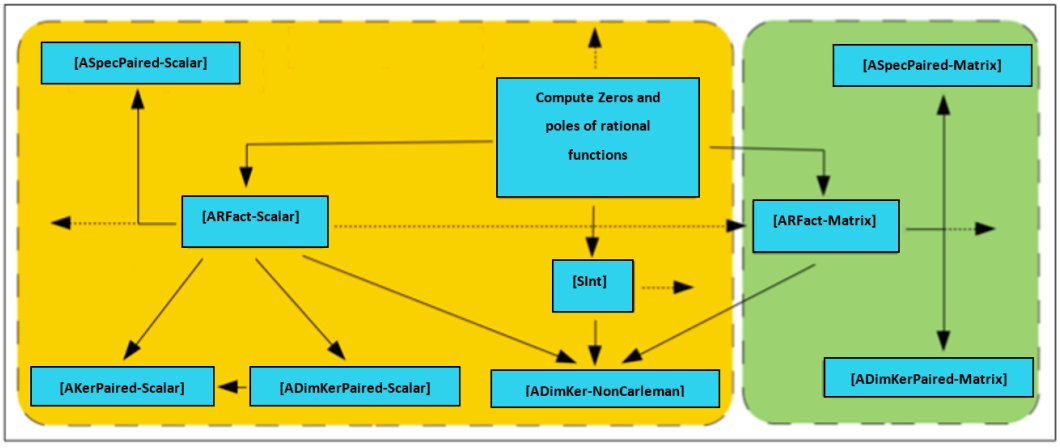


Figure 2-5 *- Relations between the algorithms, adapted from* Conceição (2021).

An interesting line of research is to improve the existing operator theory algorithms, creating new and more efficient versions. Some of the existing operator theory algorithms can be improved by introducing more possibilities for "input" and "output". The possibility of storing information about "root objects", obtained by the resolution of equations, can also improve the effectiveness of some algorithms. During the development of this language, detailed in section 6, created to facilitate the development of algorithms related to the operator theory, the [SInt] algorithm could be improved by introducing the possibility of saving information obtained through the "root objects". The information provided by such objects helps the user controlling whether the roots of the provided function lie over, inside, or outside the unit circle, and by knowing this, wrong outputs of the algorithm could be avoided. This improvement resulted in a new version of the algorithm with greater efficiency, called [SInt_SIOL].

It is obvious, the need for continuous design and development of new analytical algorithms in the area of operator theory, considering new concepts and new classes of operators and functions. In this sense, parts of the [SInt] algorithm code can be used to design new analytical algorithms for solving integral equations and for studying the spectrum and the kernel of special classes of singular integral integrals with essentially bounded coefficients.

The success of the created algorithms depends on the possibility of finding zeros and poles by solving polynomial equations. This can be a serious limitation when working with polynomials of the fifth degree or higher. However, even in this case, thanks to the symbolic and numeric capabilities of *Mathematica*, it is still possible to obtain the zeros and the poles of a rational function. For instance, if a rational function

$$r(t) = r_1(t)/r_2(t) \tag{5}$$

has a numerator with a high degree, we cannot find explicit formulas for its zeros. The root object is not a mere denoting symbol but rather an expression that can be symbolically manipulated and numerically evaluated. In particular, it is still possible to know if any given root lies in $\mathbb{T}$, in the interior, or in the exterior of the unit circle, which is all the information needed by some algorithms to give the desired output.

This limitation could be treated with the creation of the [SInt_SIOL] algorithm, by checking the conditions imposed by the algorithms, informing the user, and preventing the error to happen. The solution created analyses the poles and decides whether it was possible or not to have an accurate solution with the expression used (Conceição, 2021). Recently with the development of the $[\text{SInt}]_{2.0}$ (Conceição & Pires, 2022) this problem could be fixed, but the

changes on the [SInt] that could overcome this limitation were not yet transferred to definitions of SIOL language. This is an improvement for the next version.

Other algorithms, described with more details in chapter 7 were created with the help of the DSL, to verify when a function belongs to $\mathcal{R}_{+}(\mathbb{T})$ or to $\mathcal{R}^{o}{}_{-}(\mathbb{T})$ and to compute singular integrals associated with the Cauchy projection operators $P_{\pm}$. These calculations can be very useful when writing new algorithms.

2.2. Domain-specific language (DSL)

A programming language, as mentioned, is a notation for writing programs that are specifications of a computation or algorithm. It comprises a set of instructions used to produce outputs, that are very often computer software. The description of a programming language can be split into two components, syntax, which corresponds to its form, the way it is written, and semantics corresponds to its meaning.

A domain-specific language is a programming language with a higher level of abstraction optimized for a specific class of problems. A DSL uses the concepts and rules, applied to its syntax and semantics, from the field or domain and can reduce the costs related to software development and maintenance, by facilitating the users' ability to reuse code (Barisic et al., 2011). The main purpose of a DSL is to encapsulate and abstract the generic code into a new interface, offering a restricted suite of notations and abstractions that can be easily understood by a domain expert user. This is contradictory to a GPL (**Table 2-1**). A GPL, as mentioned before, is a type of computer programming language designed to be used for software writing in the widest variety of application domains. By definition, it can run on or compile to any system. Even though some languages might seem fitter to a task than others, due to speed or hardware, programmer-friendly, or even simply because they are easier to learn, does not make them specific to a domain/purpose. As an example, three of the most well-known programming languages existing nowadays, Java, Python, and C++, could technically be called a GPL.

In many knowledge areas, it is common to develop solutions using languages that focus on the problem's specific purposes and algebraic modelling languages are the ones accepted and adopted for mathematical modelling.

Language workbenches make the development of new languages affordable and, therefore, support a new quality of language engineering, where sets of syntactically and semantically integrated languages can be built with comparably little effort. They are tools that support the efficient definition, reuse, and composition of languages and their Integrated Development

Environment (IDE). The usage of the language workbenches can lead to multi-paradigm and language-oriented programming environments that can address important software engineering challenges (Fowler, 2006). Language workbenches are currently enjoying significant growth in number and diversity, driven by both academia and industry. Existing language workbenches are so different in design, supported features, and use terminology that it is hard for users and developers to understand the underlying principles and design alternatives. To this end, a systematic overview would be helpful. Textual workbenches like JastAdd, Rascal, Spoofax, and the one used on this project, Xtext, can be seen as very modern and up-to-date options, leveraging advances in editor technology of mainstream IDEs (Erdweg et al, 2011).

Table 2-1 *– Comparison between GPL and DSL.*

	DSL	**GPL**
Syntax	Custom-made	Generic use
Expressiveness	High, since it is focused on a Domain	Lower, used for all sot of solutions
Executability	Does not need to be executed	Needs to be executed
Reuse	The primary contribution is to enable reusable articles	Can or cannot enable the reuse
Efficiency	High, since focused on what	Lower, a lot of effort is invested on the how
Development	Easier, does not demand many computer programmer skills	Demands sophisticated computer knowledge
Target audience and usability	Focus on a specific audience and easy to use	For a general audience, and demands a complex training

Having the concept of a DSL in mind, a variety of languages for a plethora of different domains was developed. In the next section, there is an analysis of some of the existing ones for the mathematical domain.

2.3. Comparison of existing mathematical languages

To make sure that there are no other DSL that could deal with the calculations required by the operator theory algorithms, other mathematical programming modelling languages GAMS, AMPL, and ZINC were analysed in detail.

These languages were developed to alleviate many of the difficulties associated with the development of large, complex mathematical programming models, and to allow their direct formulation and resolution on a computer (Martins & Conceição, 2017). These languages have a similar purpose but they have differences in the syntax and in the way they handle different presented problems (Bussieck & Meeraus, 2004; Fourer, 1998; Rafeh et al., 2005). The idea behind these languages is to write a solver program like writing math itself.

The modelling language GAMS (General Algebraic Modelling System) impetus for development arouse at the World Bank to facilitate the solution of multi-sectoral economy-wide models, where several manuals, time-consuming, and error-prone solutions using FORTRAN programs (Backus, 1957), had been previously used(Bussieck & Nelißen, 2020). The modelling language AMPL - a modelling language for mathematical programming - is a powerful algebraic modelling language for linear and non-linear optimization developed at AT&T Bell Laboratories, to express mathematical programs intuitively (Paarsch & Golyaev, 2016). Zinc is a modelling language developed as part of the G12 project, a project started by National ICT Australia (NICTA) to develop a platform for solving large-scale combinatorial optimization problems (Rafeh et al., 2005; Stuckey et al., n.d.), and is the newest among these languages.

From a syntax point of view, as expected, these CASs kept the problem definition and the actual solution implementation separated. During this research, it was possible to verify in (Fourer, 1998), (Chen et al., 2009), and (Bussieck & Nelißen, 2020) that the running code used by the solvers is generated in a process apart from problem design.

Although all the presented languages are very interesting and have great features, their focus limits their expressiveness to the purpose of each one. Regarding the focus of this project, the most interesting feature of these languages resides in the fact that the solution can be written in a style very close to the mathematical one, which is a facilitator to an audience that might not have experience with common GPL languages, like Python or Java, among others (Martins & Conceição, 2017). Despite this, none of the presented languages could entirely satisfy this project's needs when working with problems involving the operator theory and Cauchy integrals as the ones described in (Conceição et al., 2013), due to their extensive symbolic and numeric calculations on its algorithm. Also, the lack of a proper parameter structure and the fact that the language reserved words used do not match the context of the desired domain, shows that any attempt to build this type of solution would drastically fail. A brief comparison between the languages is shown in **Table 2-2**.

***Table 2-2** - Comparison between languages.*

Dimension	AMPL	Zinc	GAMS
Mathematical-like notation	√	√	√
Language type	Declarative and imperative	Declarative and functional	Declarative and procedural
Types: Integer, float, array, sets	√	√	√
Types: Boolean, tuples, records		√	
Constraints	√	√	√
User-defined predicates and functions	√	√	√
Type checking	√	√	√
Libraries	√	√	√
Model separation	√	√	√
Solver	Independent	Independent	Independent

One good example of an existing DSL, that gets close to the needs of this project has been published in a paper called "Mathematical analysis using functional programming" (Ionescu & Jansson, 2016). The creators took as motivation a known problem: students of computer languages are comfortable with the "computer science perspective" which does not apply to mathematical expressions. The idea that a mathematical notation is often ambiguous and context-dependent, and that there is no attempt to even make this ambiguity explicit, explains the discomfort. Behind this lays the notion that any proofs in computer science tend to be more formal, and often uses an equational logic format with explicit mention of the rules that justify a given step, whereas mathematical proofs are presented in natural language, with many steps being justified by an appeal to intuition and semantic content, leaving a more precise justification to the reader. The solution proposed by the authors is that the students should approach a mathematical domain the same way they would for any other domain that they are supposed to model a software system, more specifically, the approach that a functional programmer would take. In computer science, functional programming is a programming paradigm where programs are constructed by applying and composing functions. It is called a declarative programming paradigm, the one, in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program. The importance of a well-designed syntax and

the creation of reusable functions that have a clear objective stated in their names are examples of inspirational ideas. Another inspiration that could be used is the creation of types that increase the readability and writability of the new code created using the language.

A second work that has an interesting view on DSLs for mathematics is "A Domain-Specific Language for Discrete Mathematics" (Jha et al., 2013) which has been developed to enable the implementation of discrete mathematics concepts. The DSL consists of a library of functions and data structures for the branches of set theory, graph theory, mathematical logic, number theory, linear algebra, combinatorics, and functions. As well as the "Mathematical analysis using functional programming" mentioned before, this DSL focuses on a functional programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data, because of that, they are an ideal choice for developing mathematical tools. The language is a pre-processed DSL, with the Haskell programming language as the base language and Glasgow Haskell Compiler (GHC) as the compiler. The reason for selecting Haskell is that it is purely functional and hence has no side effects. Among the advantages of using Haskell, the author cites: the lazy evaluation, where the evaluation of an expression is delayed until its value is needed; the expressive system, that creates and uses algebraic data types and performs pattern matching, polymorphic types, and functions; the fact that the language is list comprehensive and can have the logic theory and its operators and quantifiers implemented.

Table 2-3 *- Comparison between languages.*

	Mathematical Analysis Using Functional Programming	**Domain Specific Language for Discrete Mathematics**
Functional Programming	√	√
Well Designed Syntax	√	√
Reusable Function	√	√
Typing	√	√
Pre-processed		√
Advanced Mathematics Functions		√

This language also presents support to mathematics relations and some other features like linear congruence and relations of the form ax ≡ b as well as the evaluation of modular operations

such as addition, multiplication, and exponentiation. The DSL also guarantees support to data structures for vectors. One of the main objectives of this created DSL was to keep its syntax very close to the notation followed for discrete mathematics.

The two mentioned publications were a real contribution to inspiration in the creation of the DSL format proposed in this work, but, because of the extensive symbolic features of the operator theory, none of the proposed languages can be used as a solution to the problem.

Considering the lack of a DSL that can provide a proper approach to the symbols and operators involved in the operator theory problems, section 6 will describe a new language focused on problems related to the computation of singular integrals defined in the unit circle.

The following chapters will present the investigation methodology, the problem analysis, and the proposed solution.

3.Investigation Methodology

3.1.Design Science Research

Since the goal of this work was to build a new DSL, the methodology chosen to support this development was Design Science Research (DSR). Among all existent research methodologies, DSR presents itself as the ideal choice, once it is defined as a strict process to work with artefacts focusing on problem-solving, project analysis, or, in a case where it is already working, study its behaviour and after that, communicate the results obtained (Çağdaş & Stubkjær, 2011). It aims to produce actionable knowledge for professionals as well as contribute to the body of knowledge. DSR is motivated by the desire to improve the environment, and the human condition by the introduction of new and innovative artefacts and the processes for building these artefacts (Simon, 1996).

DSR can be understood as an embodiment of three closely related cycles of activities, which are: the relevance cycle, the rigour cycle, and the central design cycle. Drechsler and Hevner (Drechsler & Hevner, 2016) also propose a fourth cycle for the DSR, change, and impact, to capture the impact in the time of the artefacts in the wider socio-technical system context where it is utilized. The recognition of these three cycles in a research project position differentiates design science from other research paradigms (Hevner, 2007).

This methodology is driven by the desire to solve a field problem and not fill a knowledge gap, taking the perspective of the professional who has the problem, not from a neutral one, and aiming to develop generic solutions to field problems, not just describe or explain them. "*Knowledge claims*" and other research artefacts from DSR are based on pragmatic validity, whether or not they are true. It is used in transdisciplinary research, communicating among them through analogous thinking. In DSR, system thinking is used for the analysis and understanding of wholes and the relationships between the parts used to make them coherent (Lutkevich, 2021). DSR adds a pragmatic and normative orientation on how this understanding should be used to design actions or objects to achieve desired outcomes (Ropes, 2018).

The **relevance cycle** (**Figure 3-1**) provides the research problem, the requirements, and the acceptance criteria for the artefact's utility in the field (Drechsler & Hevner, 2016). This cycle links the environment to the artefact and constitutes an interface between its inner workings and the elements of its environment (Simon, 1996). From the starting point on the identification of the initial problem, co-creation and co-design play a prominent role in leading the design process. During this project's problem explanation, a mathematics specialist and a computer

sciences researcher through exploratory group discussions elucidates the challenges related to the proposed problem. The identified challenges were then assessed through brainstorming technological solutions to create a prototype from the perspective of the end-users. Following this, through the process of co-creation, user requirements, proposed by the mathematics part, of the practical implementation of the technological solutions are brought into the light. More specifically, the problem to be addressed is the need of having a new DSL for mathematical purposes, with different features from the ones already in the market and evaluated by the final user. Finally, to ensure the relevance of the proposal, a questionnaire was presented to a group of relevant end-users proving the need for the project.

The **rigor cycle** provides past knowledge to the research project to ensure its innovation. The aim is to identify what solution would provide a meaningfully contextualized artefact to tackle the problem under analysis. The end-user requirements are the cornerstone for the design and development of the artefact. For this work, extensive research and analysis on the market and research options were done, and three of the most used options for mathematical languages have been selected to demonstrate the inexistence of a solution that can suffice the purposes of this project. The knowledge base is referenced in this project to guarantee that the methodology produced is research contribution and not routine designs based upon the application of well-known processes (Drechsler & Hevner, 2016). "It is the rigour of constructing IT artefacts that distinguish Information Systems as design science from the practice of building IT artefacts." (Hevner, 2007). So, the additions to the knowledge base as a result of this research will include extensions to the original theories and methods made during the research, the new meta-stages of the software development cycle artefacts (the language produced and the editor), and all experiences gained from performing the research and field tests of the artefact in the application environment.

The **design cycle** is the most important part of any DSR project. This cycle of research activities iterates more rapidly between the construction of an artefact, its evaluation, and subsequent feedback to refine the design further. During the design and development of the solution artefact, there was a continuous iterative contact with end-users, represented in this project by the mathematics specialist, through codesign presentations to pool together the collective creativity and to adjust the original problem and user requirements to fit the needs to a satisfactory level. During the demonstration and evaluation, end-users used and gave their feedback on the implemented artefact. This feedback was used for improvements or changes in the solution to meet the users' needs and to realize its full potential of intended usage as suggested in (Simon, 1996), where the author describes the nature of this cycle as generating

design alternatives and evaluating the alternatives against requirements until a satisfactory design is achieved. As discussed above, the requirements are input from the relevance cycle and the design and evaluation theories and methods. The deliverables of this work will be a DSL definition, a methodology of use, available in chapter 6, and an editor to be used with it. During the construction, Eclipse and Xtext (Efftinge, 2021), will be used as the development tools. As part of the development phase, tests were performed to identify possible defects and, when problem-free, the artefact was presented to the final users. At first, a small, selected group will be part of the quality assurance team and give feedback on the usage of the language. Progressively, the language can be improved with the use of those comments.

The output from the DSR must be returned to the environment for study and evaluation in the application domain and analysed to examine how the environment has been improved and if this improvement can be measured (Cole et al., 2005; Jarvinen, 2007). The impact of the DSL will be visible by the way mathematical expressions, initially, the ones related to the operator theory can be represented and demonstrated, and this impact can be measured by the final user's feedback, which will also determine whether additional iterations of the relevance cycle are needed.

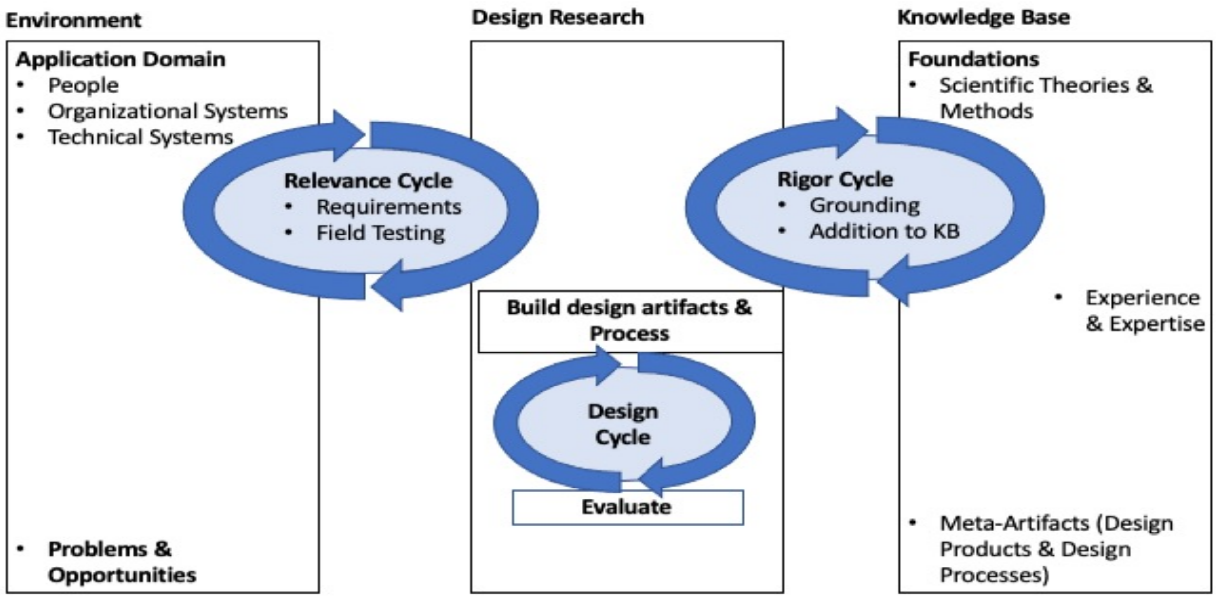


Figure 3-1 *- DSR scheme.*

4.Problem Analysis

4.1.3 Introduction

To better understand the research scenario, the impact of a DSL on the everyday life of the users, and the possible improvements in the results obtained by them, a questionnaire was applied to a discrete group of mathematicians. The questionnaire was performed in May 2021 and the group involved 50 researchers whereas only 25 answered the questions. The questions were distributed using a Google forms survey template (Appendix A) and the complete set of results can be seen in Appendix B. Even though there is no direct mention of the participants' age, this survey, answered by a diversified group of ages, was part of this experiment. All the group members had a mathematical background. The main goal, in the context of this group of researchers, was to understand how many users were able to effectively perform their programming tasks, and how much the specificities of the proposed language could affect their research production.

The following charts were created from the survey responses. The final results allowed moving forward with the proposed project.

4.2.Survey Results

This section will display figures with graphs, created as a result of the previously explained questionnaire. Here the goal is to justify the creation of a DSL for operator theory by pointing out its relevancy and need throughout the scientific community, represented by the participants in the survey. The biggest part of the questioned group works at a research centre for mathematics. All of them are, somehow, connected to research in the area of operator theory (**Figure 4-1**).

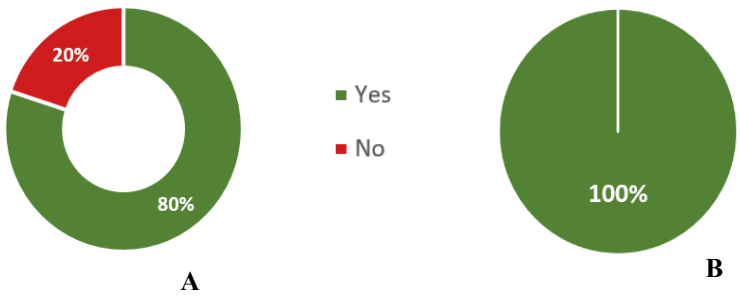


Figure 4-1 *- Results obtained from the response to questions* ***A*** *"I am a member of a mathematic scientific research center" and* ***B*** *"My research field includes topics related to the operator theory".*

None of them confirmed having any advanced knowledge in computer programming and a relevant group of researchers are beginners or have a basic knowledge of computer programming (**Figure 4-2**). Also, the *Wolfram Mathematica* has proven to be very popular among the participants (**Figure 4-3**).

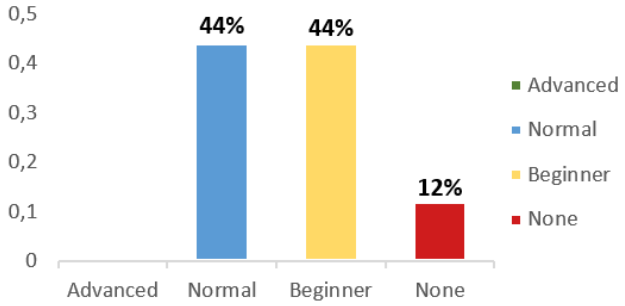


***Figure 4-2** – Results obtained from the response to the question to measure the "Level of computer programming language knowledge of the participants".*

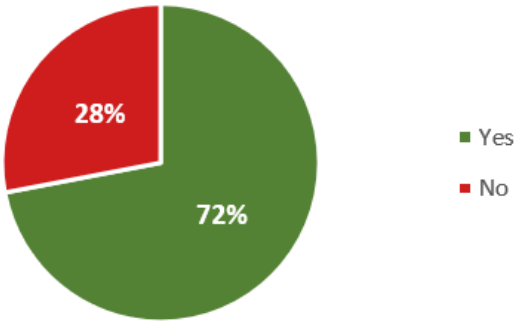


***Figure 4-3** – Results obtained from the response to the question related to: "Popularity of the Wolfram Mathematica among mathematics researchers".*

From the evaluation of the results, a major part of the participants has declared a real interest in improving their skills in computer programming (**Figure 4-4**). Also, the smallest part of the questioned group affirmed that using a CAS is a significant part of their everyday tasks (***Figure 4-5***) but would like to take advantage of them more often (**Figure 4-6**).

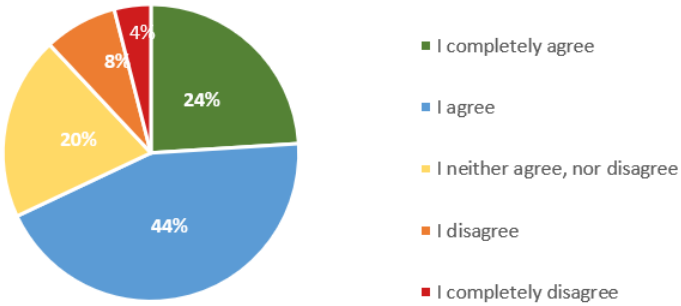


***Figure 4-4 -** Results obtained from the the participants' response "Desire of programming skills improvement".*

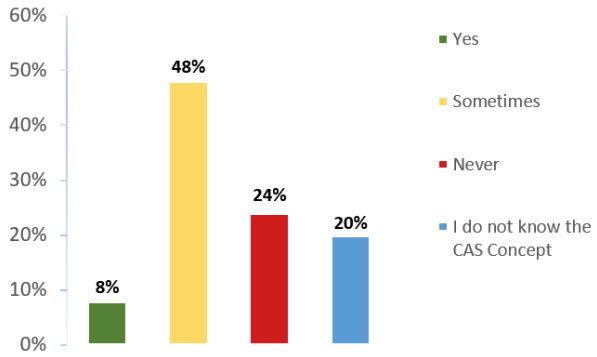


Figure 4-5 *– Results obtained from the participant's response to "CAS is a significant part of their everyday tasks".*

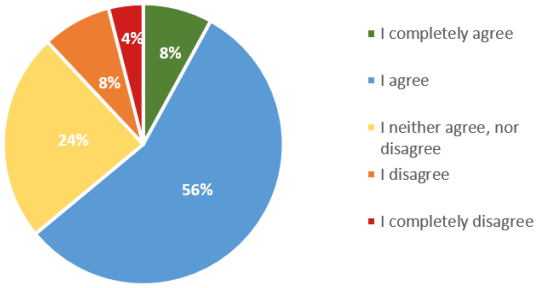


Figure 4-6 – *Results obtained from the response of "interest of users in CAS's".*

In the end, most all users declared being interested in the existence of a high-level programming language to solve specific problems in operator theory, that can be used without the support of a programming expert (**Figure 4-7**).

Concerning the results above, it is obvious the intention of using a language like SIOL, a language focused on the operator theory area, making use of a set of features that can diminish the existing barrier between the researcher and the programming language used for their everyday tasks. It is visible that the lack of experience with programming languages is real and that making this part of mathematician research easier is necessary and relevant.

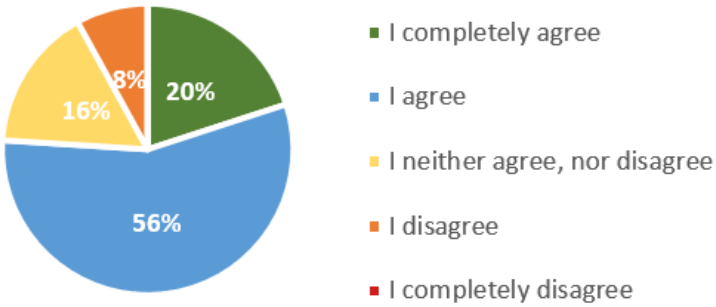


Figure 4-7 – *Results obtained from the response of "interest of users on a programming language focused on the operator theory".*

When analysing these results, it is also possible to see a high participant percentage that would like to invest more in using any sort of CAS (**Figure 4-8**), showing interest in this type of solution. The answers from researchers using *Wolfram Mathematica* (**Figure 4-9**) also show that the results provided by SIOL are very useful for this type of user. Among those who participated, a considerable part of the scientists claimed to know the already existing *Mathematica* algorithms, focused by the DSL, for singular integrals and matrix factorization (**Figure 4-10 A** and **B**).

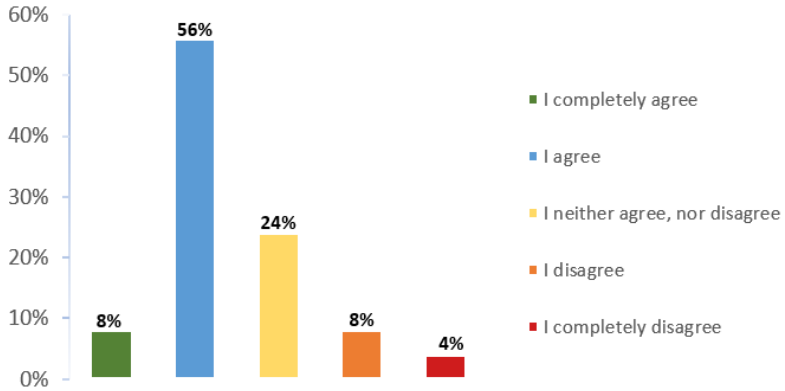


Figure 4-8 *– Results obtained from the response of "interest of the researchers on CAS".*

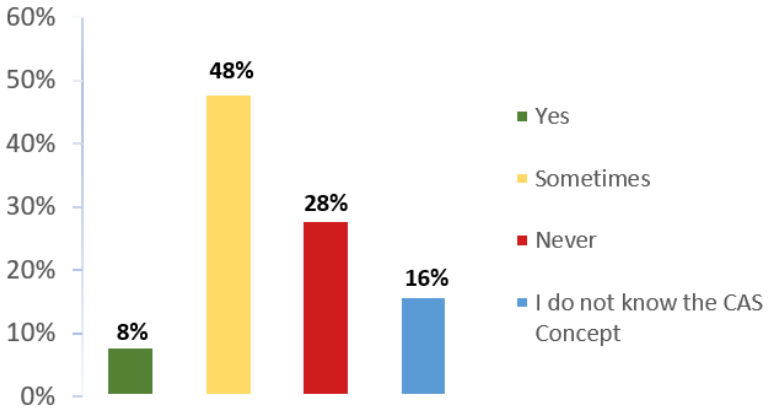


Figure 4-9 *– Results obtained from the response to the question related to: "researchers usage Wolfram Mathematica".*

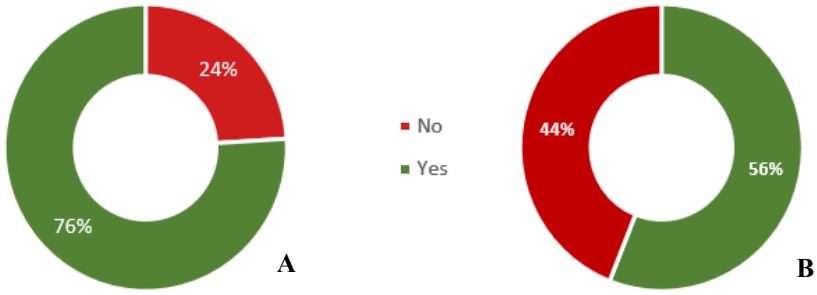


***Figure 4-10** – Results obtained from the response to the questions related to A "Knowledge of algorithms, implemented with the Wolfram Mathematica Language, for matrix functions factorizations and B "Knowledge of techniques for calculating singular integrals implemented with the Wolfram Mathematica language".*

Many of the participants know or at least would like to know more about the concept of a DSL (**Figure 4-11**) and the biggest part showed interest and affirmed that they would be a user of a new language created in the area of operator theory (**Figure 4-12**). With all this information, it is easy to see how the proposed language may help to improve the solution in operator theory tasks.

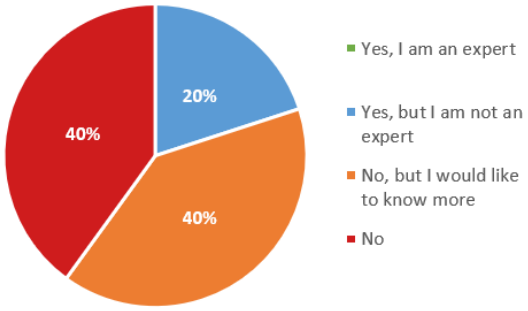


***Figure 4-11** – Results obtained from the response to the questions related to the knowledge of "the concept of Domain Specific Language (DSL)".*

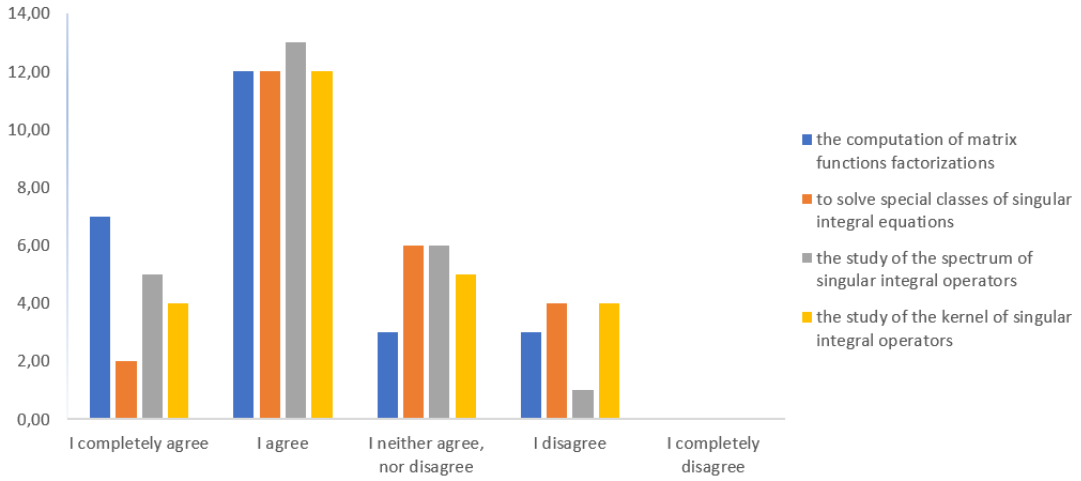


***Figure 4-12** – Results obtained from the Acceptance of a created DSL for the operator theory evaluation.*

5.The Proposed Solution

5.1.Introduction

The DSL proposed in this work aims to eliminate the need for an expert by a non-programmer specialist, the main goal of this project is to create a powerful extensible textual language that will formalize the mathematical models in the domain of the singular integrals focused on the "what" instead of the "how" and making the language as declarative as possible. In the end, instead of having to formulate complex algorithms in a generic environment, this DSL will allow the user to have models defined in a specialized language with reusable functions and parameters, focused on solving different classes of problems related to the computation of singular integrals. All the functions created for this new language were based on existing algorithms for operator theory, which could be combined to create other new algorithms. These methods were created with identifications (names) that are meaningful to this mathematica's area of knowledge, thus making the usage simple and trivial for any user, but especially for the mathematicians. The created language is divided into three parts, the *inputs*, the *block* of functions, and the *output*. In the section called *inputs***,** the necessary inputs to the algorithms are created, on the next section, the *block* is where the actual computation of the functions happens, and the methods can be used, for example, the [SInt] (Conceição et al., 2012)**,** that calculates the Singular Integrals based on an algorithm of the same name**,** or the one responsible for calculating the **singular integrals associated with** the projection operator P_+, designed by [PplusIntRationalSIOL]. The last section called *output* is where some types of results, like an algorithm result expression or the time taken to execute the created procedure, can be invoked. The advantage of using a DSL is that its syntax and semantics try to resemble the domain audience expertise, in this case, mathematical expression and the operator theory. Thus, to use the DSL, the users do not need to be familiar with the programming language. The syntax of the created DSL is kept close to the notation followed for the operator theory type of problems. The language grammar is an important characteristic of the project because it defines its syntax lexical rules.

Another important part and very helpful for the end-user is the language validation. The syntactical correctness of any textual input is validated automatically by the DSL. After all the verifications are complete, and the code has no errors, the final part is converting the algorithm written in the new language to a final GPL language.

This DSL translations mechanism performs a conversion of programs written in SIOL into the equivalent *Wolfram Mathematica* representations (**Figure 5-1)**. It is advantageous to use this approach for development as it allows the new language to have a converted language commonly used by computer programmers of this area of knowledge and takes advantage of the *Mathematica* computing power. An example of the produced code is included in Appendix C.

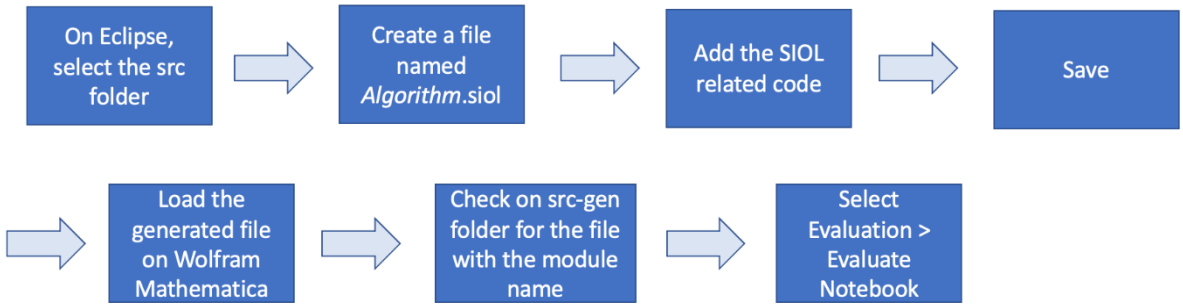


***Figure 5-1**- Step by step on the output file generation.*

5.2. The Development Environment

The main tool employed in creating this DSL was Eclipse. The reason to choose it lies, especially, in the Eclipse Modelling Framework (EMF) which is a modelling framework and code generation facility to build tools and other applications based on a structured data model (EMF, 2021), allowing Xtext to be used smoothly. EMF provides interesting features like code completion, syntax highlighting, outline automated parsing, support for quick fixes and warnings, and advanced bracket handling, which are very important for the end user's everyday usage.

Xtext is an open-source EMF-based tool used for developing programming languages and domain-specific languages (Guntli, 2010), and for this project was extensively used in the grammar design. Xtext (Efftinge, 2021) is a mature open-source framework for the development of programming languages and DSLs. It is designed based on proven compiler construction patterns and ships with many commonly used language features, such as a workspace indexer and a reusable expression language. Its flexible architecture allows developers to start by reusing well-established and commonly understood default semantics for many language aspects, but Xtext scales up to full programming language implementations, where every single aspect can be customized in straightforward ways through dependency injection. Companies like Google, IBM, BMW, and many others have built external and internal products based on Xtext (Erdweg et al., 2011)

Unlike standard parser generators, Xtext generates not only a parser, which converts text to an EMF model but also a simple and easy-to-read diagram for the abstract syntax tree. Another big advantage when using Xtext is that it is possible to make use of the fully-featured, customizable Eclipse-based IDE, Integrated Development Environment, very popular among developers, specially Java ones(Efftinge, 2006).

Xtext uses concrete syntax. The concrete syntax focuses on how the structure of the data and how it is going to be represented; for example, it includes features like parentheses or commas, and how they are positioned on the code. It contrasts with the abstract syntax, which only includes information about the data (Howe, 2022).

Concerning the validations, the error messages are generated by the underlying parser technology. Any syntax errors are shown in real-time and marked underlying the wrong declaration on the algorithm written in the new language. These rules are declared on an Xtend file part of the DSL (Efftinge, 2006).

Xtend is a statically typed programming language that translates to comprehensible Java source code. Syntactically and semantically Xtend has its roots in the Java programming language. As soon as the Xtext artefacts are generated from the grammar, a code generator stub is put into the runtime project newly created language. Xtend is responsible to integrate its code generator with EMF (Efftinge, 2021)

The EMF uses another Xtend file with a set of rules that will determine how every piece of code written in the new language will be translated to the solver language. The type and name of the final generated file are also defined in this step.

To be able to use the SIOL, and have its parts built, a user's system should be able to compile and run Xtext codes on an Eclipse IDE. The minimum configuration used during testing is in **Table 5-1**.

Table 5-1 *– Minimum configuration.*

Java	**JustJ OpenJDK Hotspot JRE Complete 16.0.1.v20210528-1205**
Xtext	2.25.0.v20210301-1429
Operational System	MacOs Big Sur version 11.6
Wolfram Mathematica	12.3.1.0 for Mac OS X x86 (64-bit)

6.The SIOL Language

6.1.Introduction

Considering that there is no proper modelling approach to represent operator theory problems, in this section we intend to describe a new language related to the problem of singular integrals computation defined in the unit circle. The Singular Integral Operator Language (SIOL) was created to facilitate the development of operator theory algorithms, in most cases by people with little or no knowledge of any other programming language.

The main goal of this work is to develop a Domain Specific Language (DSL) that has been idealized to enable the implementation of operator theory concepts, namely a set of reusable functions that provides the functionalities frequently required by users. The proposed programming language can be defined as a language that is used to execute instructions and algorithms on the user's machine. The generated instructions or algorithms are *Wolfram's Mathematica* files, more specifically, the type of file called Notebook with an extension *nb*. When developing the generated code, there was a particular concern that the program had the qualities of reliability, robustness, usability and efficiency.

The use of SIOL enables the creation of new algorithms only by reusing existing ones. Relevant parts of the known algorithms have been divided into smaller reusable parts and encapsulated into easily recognizable functions, which, if necessary, can be used to shape new programs. Since the SIOL code is smaller, it has become easier to write and create new algorithms. Another important factor to which the foundation of this new DSL contributed is reliability for the final user, since in SIOL there is a large set of validations focused on operator theory, avoiding errors related to this specific domain. This means, in the end, that fewer things can go wrong, and with this, time and effort can be saved. This is very important, especially if the user is dealing with critical subjects. Since most mathematicians, as explained before in chapter 4, are not programming experts, the errors, whenever they happen, will be errors specific to the domain, thus, easier for them to understand. The key audience for this new language is academics and researchers.

6.2.Concepts

The language SIOL has rules on its grammar to check and validate the list of inputs used in the created algorithms. The rules involve the definition of the rational function *r(t)*, the declaration of the poles and zeros, all the simple algebraic expressions used to compose any of the algorithm components, among other validations that concern deeper concepts of operator

theory. The language presented is an ongoing process that may have added to its features other modules associated with the operator theory.

6.2.1. Module Structure

The presented items represent the concepts that create the structure of the module. "*Module",* is the name used to refer to the SIOL code itself.

Module: The *Module* expression is used to name the algorithm. This string added also names the *Mathematica* file. For example, **Module** SInt, creates the file *SInt.nb.*

inputs: This section will include a set of expressions that will represent the inputted data, by the user, to the algorithm. These are mathematical expressions, that must follow the known rules of mathematical writing. The section is delimited by the word *end.* The operations accepted so far are basic arithmetic.

block: This section will contain a set of expressions representing the process of the algorithm. This is the section, where existing algorithms can be reused to perform tasks. This area is delimited by the word *end.*

output: This section will include a set of expressions that will display the results from the processing performed in the *block* section. Here the user can select what kind of results will be printed. These outputs can be the direct expected results or supplementary information like the time took to operate. This section is delimited by the word *end.*

6.2.2. *Block* functions

The presented items represent the concepts that compose the structure of the *Block*.

computeResultMinusX: Calculates the singular integral $P_{-}X(t)$.

computeResultPlusX: Calculates the singular integral $P_{+}X(t)$.

computeResultMinusY: Calculates the singular integral $P_{-}Y(t)$.

computeResultPlusY: Calculates the singular integral $P_{+}Y(t)$.

computeResultMinus: This function is used to verify if the rational function, *r(t)*, provided as an input belongs or not to $R^{o}_{-}(\mathbb{T})$. The result is binary, meaning, *1* belongs, *0* does not belong.

computeResultPlus: This function is used to verify if the rational function, *r(t)*, provided as an input belongs or not to $R_{+}(\mathbb{T})$. The result is binary, meaning, *1* belongs, *0* does not belong.

SInt: Executes the algorithm [SIntSIOL].

PMinusIntRational: Executes the algorithm [PMinusIntRationalSIOL]. This algorithm can be also written as a composition of other functions using *SInt*, *computeResultMinusX.*

PPlusIntRational: Executes the algorithm [PPlusIntRationalSIOL]. This algorithm can be also written as a composition of other functions using *SInt*, *computeResultPlusX.*

AMinusRational: Executes the algorithm [AMinusRationalSIOL]. This algorithm can be also written as a composition of other functions using *PMinusIntRational*, *computeResultMinusX,* and *computeResultMinus.*

APlusRational: Executes the algorithm [APlusRationalSIOL]. This algorithm can be also written as a composition of other functions using *PPlusIntRational*, *computeResultPlusX,* and *computeResultPlus.*

PPlusMinusInt: Executes the algorithm [PPlusIntRationalSIOL]. This algorithm can be also written as a composition of other functions using *computeResultPlusX, computeResultMinusX, computeResultPlusY, computeResultMinusY, SInt.*

6.2.3. *Output* Functions

The presented items represent the concepts that create the structure of the *Output*.

printInputOptions: Prints on the screen all the input options used by the user on the input area.

printResultMinus: Prints on the screen, whether *r(t)* belongs or not to $R^{o}_{-}(\mathbf{T})$.

printResultPlus: Prints on the screen, whether *r(t)* belongs or not to $R_{+}(\mathbf{T})$.

printResultXMinus: Prints on the screen the obtained $P_{-}X(t)$.

printResultXPlus: Prints on the screen the singular integral $P_{+}X(t)$.

printResultYMinus: Prints on the screen the singular integral $P_{-}Y(t)$.

printResultYPlus: Prints on the screen the singular integrals $P_{+}Y(t)$.

printPMinusRt: Prints on the screen the obtained $P_{-}r(t)$.

printPPlusRt: Prints on the screen the obtained $P_{+}r(t)$.

printOutputAPlusRational: Prints on the screen the results of the [APlusRationalSIOL] algorithm, meaning whether it belongs or not to the $R_{+}(\mathbf{T})$.

printOutputAMinusRational: Prints on the screen the results of the [AMinusRationalSIOL] algorithm, meaning whether it belongs or not to the $R^{o}_{-}(\mathbf{T})$.

printOutputPPlusMinusIntRational: Prints on the screen the results of the PPlusMinusIntRational algorithm, meaning whether it belongs or not to the $R^{o}_{-}(\mathbf{T})$ or $R_{+}(\mathbf{T})$.

printSingularIntegrals: Prints on the screen the singular integrals, $S_{T}X$, and $S_{T}Y$ calculated by the [SIntSIOL] algorithm.

printRBiggestExponent Responsible for printing on the screen the biggest exponent of the rational function

printXBiggestExponent Responsible for printing on the screen the biggest exponent of the auxiliary function x_{+}.

printYBiggestExponent Responsible for printing on the screen the biggest exponent of the auxiliary function y_{-}.

printExecutionTime: Prints on the screen how long in seconds the procedure took.

6.3.Grammar

The kernel of a language and what makes it a language rather than an arbitrary sequence of symbols is its grammar. A grammar specifies the order in which the symbols of a language may be combined to make up legitimate statements in the language. Human languages have rather relaxed informal grammars that we pick up as children. Computer languages are sometimes called formal languages because they obey an explicitly specified grammar (Mikhail Barash, 2018).

The EMF provides a simple mechanism to edit and maintain an Xtext created grammar. Xtext uses EMF models as the in-memory representation of any parsed text files. This in-memory object graph is called the abstract syntax tree (AST). The AST should contain the essence of the textual models. A model is every component of the grammar as seen in **Figure 6-1**, it abstracts over syntactical information. It is used in later processing steps, such as validation and compilation.

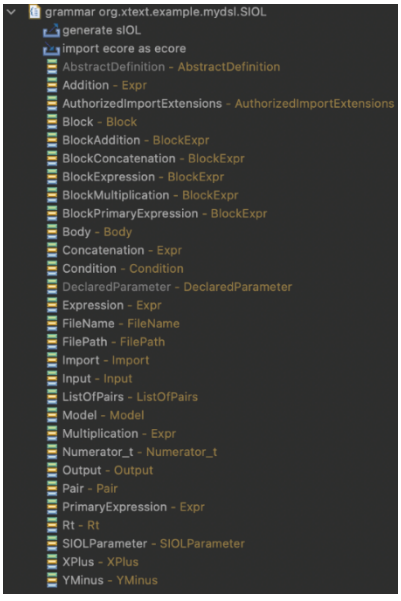


Figure 6-1 – Xtext AST.

In the EMF a model is made up of instances of *EObjects* that are connected. An *EObject* is an instance of an *EClass*. A set of *EClasses* can be contained in a so-called *EPackage*, which are both concepts of the *Ecore* language.

The set of rules that defines SIOL includes the formal definition of the structure of the code, including the name of the algorithm created, and the divisions in input, block, and output (all visible in **Figure 6-1**). The need to create the main inputs of the base algorithms is due to the private words of the language and also to highlight their importance to the created solutions. Some of the rules go as follows:

- Rule to represent the different main components of the language: **Figure 6-2** concerns the main structure containing the three main components: *Inputs*, *Block* (of Statements), and *Output*. We use the *Module* to name the algorithm as well as the generated .nb file.

```
Model:
    'Module' name=NAMES ';'
    (body=Body);

Body:
    {Body}
    ('inputs'
        (inputs+=Input)+
    'end')?
    ('block'
        (block+=Block)+
    'end')?
    ('output'
        (output+=Output)+
    'end')?;
```

Figure 6-2 *– Main Structure of the Operator Model.*

- Rule to describe input data elements used in the mathematical model: In **Figure 6-3** each *Input* element has an identifier that is initialised with a value or an expression. This component also includes the request of user inputs. The inputs used in this section are mathematical expressions, in **Figure 6-4,** we have the example of an expression rule.

```
Input:
    ('r(t)' eq='=' (rt=Rt) |
    'numerator(t)'  eq='=' (num=Numerator_t) |
    xPlus=XPlus |
    yMinus=YMinus |
    name=LITERAL eq='=' ass=(Expression)) ';';
```

Figure 6-3 - *Partial code of the input rule.*

```
Addition returns Expr:
    Multiplication
    (( {Plus.left=current} '+' | {Minus.left=current} '-') right=Multiplication)*;

Multiplication returns Expr:
    PrimaryExpression
    (({Multi.left=current} '*' | {Div.left=current} '/' | {Pow.left=current} '^') right=PrimaryExpression)*;

PrimaryExpression returns Expr:
    {BracketExpression} openBrackets='(' func=Expression closeBrackets=')' |
    {SimpleNumber} (signal='-')? value=NUMBER |
    {StringLiteral} (signal='-')? value=LITERAL |
    {NumberLiteral} (signal='-')? value=NUMLITERAL |
    {FunctionCall} func=[AbstractDefinition]
        ('(' args+=Expression (',' args+=Expression)* ')')?;
```

Figure 6-4 *– Example of an expression rule.*

- Rule to describe how the block section is created: Shown in **Figure 6-6**, are some of the possible functions to be used on the block section from **Figure 6-5**.

```
Block:
    (variable=LITERAL eq='=' SIOLExpression=BlockExpression in=';');
```

Figure 6-5 *– Rule of the composition of the "Block".*

```
SIOLParameter:
    SInt = 'SInt'|
    PPlus = 'computeResultPlusX'|
    PPlusY = 'computeResultPlusY'|
    PMinus = 'computeResultMinusX'|
    PMinusY = 'computeResultMinusY'|
    checkResultPlus = 'computeResultPlus'|
    checkResultMinus = 'computeResultMinus'|
    PPlusMinusInt = 'PPlusMinusInt'|
    PPlusIntRational = 'PPlusIntRational'|
    PMinusIntRational = 'PMinusIntRational'|
    PPlusMinusIntRational = 'PPlusMinusIntRational'|
    AMinusRational = 'AMinusRational'|
    APlusRational = 'APlusRational'|
    (calculateResultXMinus='resultXMinus' in='();')|
    (calculateResultXPlus='resultXPlus' in='();')|
    (calculateResultYMinus='resultYMinus' in='();')|
```

Figure 6-6 *– Some of the possible functions to be called on the block section.*

- Rule to describe the result of the processing.

The *Output* section includes a plethora of functions representing the information expected from the execution of the algorithms created in the block section. **Figure 6-7** shows some of the possible functions that can be used as outputs.

***Figure 6-7** – Partial code of the output rule.*

6.4. Validations

When developing a DSL, the created model can be checked for errors according to certain validation rules. To make it custom, these rules will have to be defined by the modeller in the meta-model to fit the needs of the created language. A defined rule can generate errors, warnings, or info. Errors are marked with a red underline on the code itself, warnings, and info messages can be reported and attached to model elements or text locations. The syntactical correctness of any textual input is validated automatically by the parser. These messages are displayed in the Eclipse model text editor(Mooij & Hooman, 2017).

Some kinds of validation are done automatically, but some of them need to be specified as additional constraints specific to the DSL Ecore model. Describing the code itself, a validation rule is marked as such with the annotation "*@Check*" in the xtend/java code. The argument is a model element of the type to be checked. After having it defined, the rule is checked by typing, in the new editor instance created by Eclipse during execution (Efftinge, 2021).

In the case of this project DSL, a set of validations were created to check some rules concerning the basic mathematics, but more importantly the ones dedicated to operator theory. Part of the validators was created concerning the flow of the algorithm, to prevent the generator from creating faulty *Wolfram Mathematica* code when the user finishes the SIOL code implementation.

In **Figure 6-9**, we present a set of rules that were created to guarantee the correct code flow generation. When analysing the first rule, it is possible to see the DSL controlling the need for an input when a block is declared. This means that the user will not be able to compile the code without the declaration of an input. The same applies to the outputs. No output can be defined without an input. Following the conditional expression (if constructs), it is possible to see some kind of rules to ensure that the generated code does not fail and also that the expected result is

obtained. These kinds of validations were designed simply to help SIOL users to avoid simple errors.

```
@Check
void checkInput(Body b, Input i) {
    var input = b.getInputs();
    var block = b.getBlock();
    var output = b.getOutput();


    if (input == null &&  block != null){
        error("No input defined. Please define an input to be able to process the block", SIOLPackage.Literals.BODY__INPUTS);
    }

    if (input == null &&  output != null){
        error("No input defined. Please define an input to have an output", SIOLPackage.Literals.BODY__INPUTS);
    }

    if (output == null){
        warning("No output defined. No results will be printed", SIOLPackage.Literals.INPUT__LZ);
    }

    if (block == null){
        warning("No block defined. No processings will be performed", SIOLPackage.Literals.INPUT__LZ);
    }
    if (input != null){
        if((i.getRt() != null)&&(i.getLz() != null)&&(i.getLp() != null)&&(i.getNum() != null)){
            warning("Please do not define other inputs when defining r(t)", SIOLPackage.Literals.INPUT__LZ);
        }
    }

}
```

Figure 6-8 *– Xtext SIOL validations example.*

6.5. Conversion

Xtend, part of the EMF, is a highly optimized tool to be used in the code generation step. The goal here is to transform the code created in SIOL in a *Wolfram Mathematica* code, compliable and executable, that has the same behaviour and result as if it was created directly with *Mathematica*.

To accomplish this task, Xtend has an optimized template engine fully integrated with the Eclipse IDE. Among the characteristics that call the attention, we mention static typing, just like writing code for Java, seamless integration to any existing project since it compiles to java. Another important part regards the features focused on DSL developers. The IDE is prepared to make the code as readable as possible, marking the output with a grey zone and keeping the local indentation on the destination code and other interesting features like rename refactoring, content assist, formatting, among others. It has also a very easy-to-use debugger and since this is an ongoing project it allows easy increments to be created, due to its organization in classes. The generation process, as seen in **Figure 6-9** uses the abstract syntax tree, AST, as a basis for code generation.

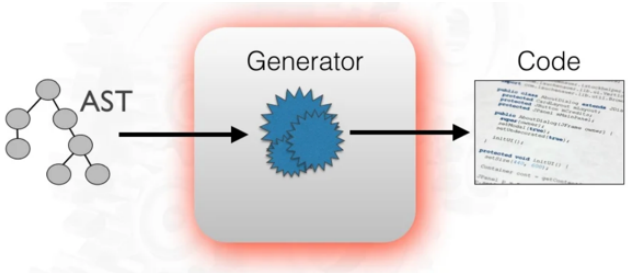


***Figure 6-9** – Code Generation, adapted from (Mooij A. & Hooman J., 2017)*

6.6.Conversion in Practice

When using the DSL, Eclipse creates a new instance of the IDE, with a prepared environment to be used with the newly created language. In this new instance, two important folders are created, one called *src*, where the files with the extension *siol* are created, and another one, *src-gen*, where the *Wolfram Mathematica* files are automatically generated after the compilation of the correspondent *siol* file. In **Figure 6-10** it is possible to see the file structure created by EMF.

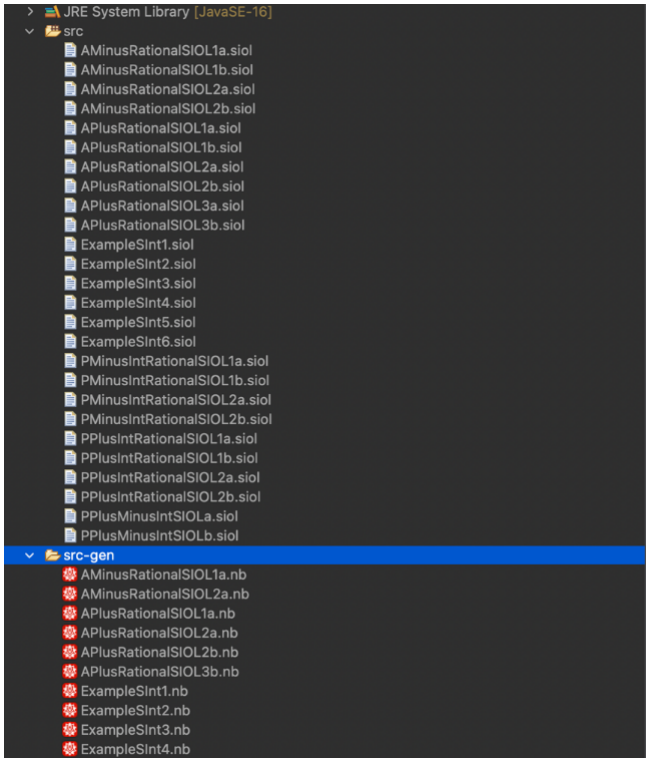


***Figure 6-10** – SIOL File Structure.*

7.SIOL Algorithms (Test Cases)

In sections 7.1 (general functions) and 7.2 (rational case) the tests cases for the proposed DSL – SIOL are described. The algorithms created try to cover as many possibilities as necessary to prove the relevance, demonstrate the operational way of dealing with the language, and also to exemplify DSL usage. Finally, section 7.3 presents the results concerning the expected and obtained data.

7.1.General Functions

During the development of this project, as a good opportunity to use the power of SIOL, some algorithms and changes to an existing one were developed. All these algorithms could be generated using the SIOL language, sometimes with a direct call to an exact function, otherwise using parts that together compose the algorithm. This is how SIOL can be used to implement not only known but also new algorithms.

7.1.1. [SInt SIOL] algorithm

Figure 7-1 shows the flowchart of the new version for the algorithm [SInt] (Conceição et al., 2013) named [SInt_SIOL]. In this version, new verifications were added, symbolized by 3 yellow diamonds. The first check is if none of the poles of the rational input, *r(t),* lies in T, in the case it happens, the output displays a message informing $r \notin R(\mathbb{T})$. After that, if the auxiliary functions were declared, and any of these functions have any pole lying in $T \cup T_+$, the algorithm will prevent failure of the execution or a wrong output, by showing a message informing that the auxiliary function, $x_+(t)$ and $y_-(t)$ does not belong to $R_+(\mathbb{T})$. These new verifications were also created as functions of the SIOL.

SInt_SIOL

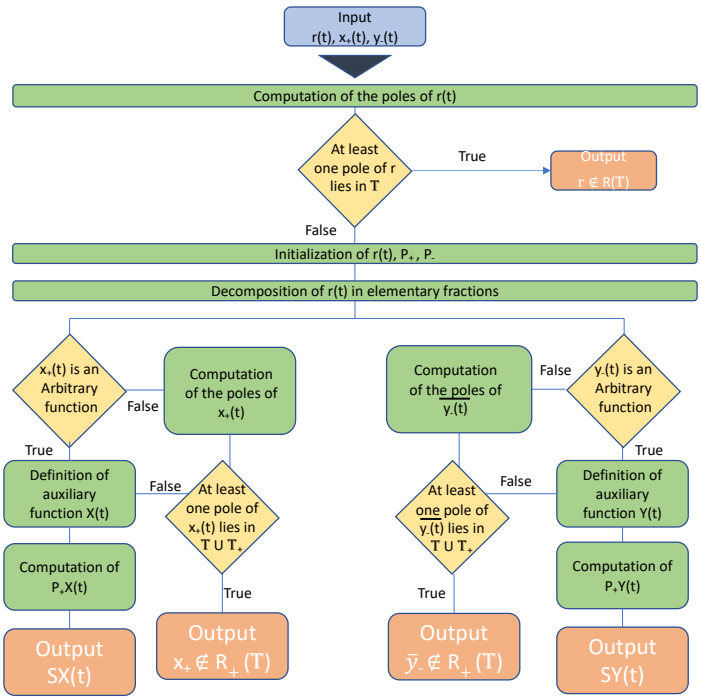


Figure 7-1 *- SInt_SIOL algorithm.*

In **Figure 7-2**, it is possible to see the implementation of the algorithm [SInt_SIOL] *algorithm* as described in **Figure 7-1**. In the first line, the *Module* statement is mandatory as well as the following sections, *inputs*, *block*, and *output*. In this example, the rational function *r(t)* was declared. In the cases where no $y_-(t)$ *or* $x_+(t)$ were declared, such as **Figure 7-2**, the algorithm assumes that it is a general expression, meaning any function such that $y_+(t)= \overline{y_-(t)} \in H_\infty(T)$ and $x_+(t) \in H_\infty(T)$. In the *block* section, the [SIntSIOL] algorithm is called directly, simply by using the function SInt. In the *output* section, the interest was in displaying the singular integrals, S_TX and S_TY, done by simply calling the *printSingularIntegrals* function.

```
Module ExampleSInt1;
inputs
    r(t)=t+(1/t);
end
block
    SInt(r);
end
output
    printSingularIntegrals();
end
```

Figure 7-2 *– [SInt_SIOL] example using SIOL.*

In the first area, the selected inputs are displayed, and in the second one are the two expected singular integrals representing the output of the example from **Figure 7-2**. Besides this

example, others were created as described in **Table 7-1**. **Figure 7-3**, shows the result of an execution of [SInt SIOL] with $r(t) = t + \frac{1}{t}$.

Input Options

r(t) = t+(1/t)

Output

Singular Integrals:

$$S_T X(t) = \frac{-2\,x_+[0] + \left(1+t^2\right) x_+[t]}{t}$$

$$S_T Y(t) = 2\,t\,\overline{y_+[0]} + 2\,\overline{y_+'[0]} - \frac{\left(1+t^2\right) y_-[t]}{t}$$

Figure 7-3 *– [SInt SIOL] output.*

Table 7-1 *– [SInt] Examples.*

Inputs		**Outputs**
Rational Function	Auxiliary Function	
$\boldsymbol{r(t) = t + \frac{1}{t}}$	$x_+(t) = \frac{1}{t}$	$x_+(t) \notin R_+(T)$
$\boldsymbol{r(t) = \frac{1}{(t-1)(t^2+4)}}$	-	$r(t) \notin R(T)$
$\boldsymbol{r(t) = 1}$	$x_+(t) = \frac{1}{t-i}$	$x_+(t) \notin R_+(T)$
$\boldsymbol{r(t) = \frac{1}{5t^4 + t^6 - 6t + 1}}$	-	It is not possible to solve the desired problem due to the polynomial degree of the expression
$\boldsymbol{r(t) = t}$	$y_-(t) = t$	$\overline{y_-(t)} \notin R_+(T)$

7.1.2. [PPlusMinusIntSIOL] algorithm

In **Figure 7-4**, the [PPlusMinusIntSIOL] algorithm is described. The algorithm is composed of some steps and rewriting it directly on *Mathematica* could become quite complex, but with SIOL, the work is simplified by using the functions that compose the algorithm, or even by simply calling the specific created function directly. The goal of the [PPlusMinusSIOL] is to calculate the Cauchy singular integrals associated with the projection operators (2).

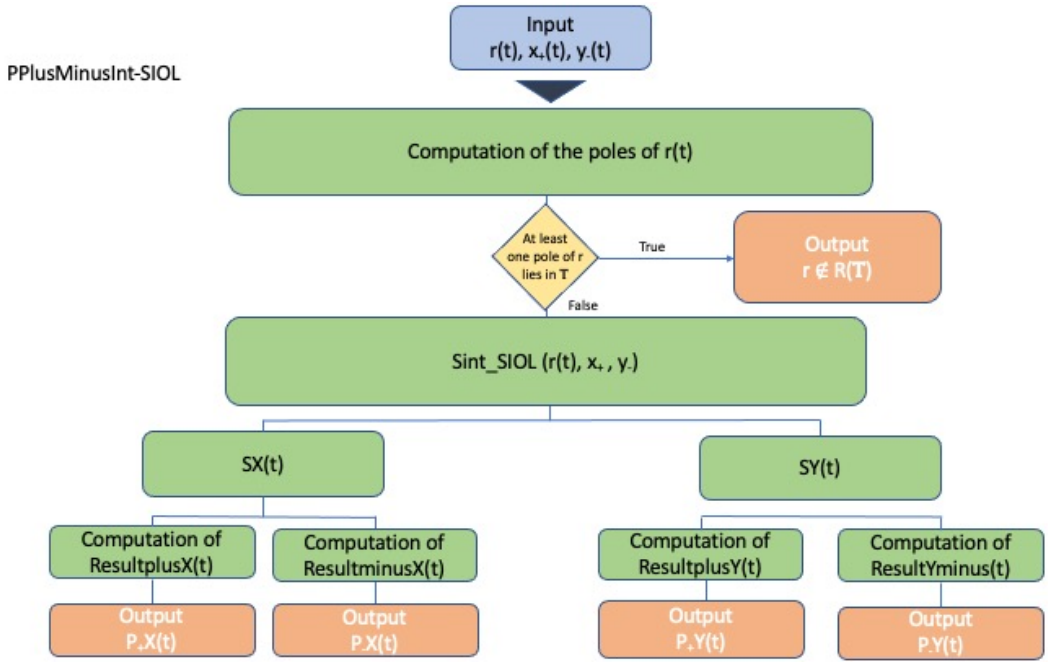


Figure 7-4 *– [PPlusMinusIntSIOL]algorithm*

In **Figure 7-5**, it is possible to see the SIOL implementation of the algorithm [PPlusMinusIntSIOL] as described in Figure 7-4. In this example, it is possible to see the rational function *r(t)* declared in the inputs section. In the *block* section, the algorithm is decomposed into the steps that create it. The output requests the *ResultPlusX(t), ResultMinusX(t), ResultPlusY(t), ResultMinusY(t).*

```
Module PPlusMinusIntSIOL1a;
inputs
    r(t)=t+(1/t);
end
block
    s=computeResultPlusX(computeResultMinusX(computeResultPlusY(computeResultMinusY(SInt(r)))));

end
output
    printResultXMinus();
    printResultXPlus();
    printResultYMinus();
    printResultYPlus();
end
```

Figure 7-5 *– [PPlusMinusIntSIOL] algorithm.*

The Output section is represented by the results of all the functions called from the *output* section on the example from **Figure 7-5**. **Figure 7-6** shows the result of an execution of [PPlusMinusIntSIOL] algortihm with input $r(t) = t + \frac{1}{t}$. Also in this case, the functions x+(t) and y-(t) were not declared, so the algorithm assumes that are general valid functions.

Output

$$\text{ResultMinusX(t)}=\frac{x_+[0]}{t}$$

$$\text{ResultPlusX(t)}=\frac{-x_+[0]+\left(1+t^2\right)x_+[t]}{t}$$

$$\text{ResultMinusY(t)}=-t\,\overline{y_+[0]}-\overline{y_+'[0]}+\frac{y_-[t]}{t}+t\,y_-[t]$$

$$\text{ResultPlusY(t)}=t\,\overline{y_+[0]}+\overline{y_+'[0]}$$

***Figure 7-6** – [PPlusMinusIntSIOL] algorithm Output.*

7.2. Rational Case

This section is dedicated to the rational case. The rational case normally is easier to manipulate given the properties of this kind of function.

7.2.1. [PPlusIntRationalSIOL] and [PMinusIntRationalSIOL] algorithms

Figure 7-7 shows us the rational algorithms [PPlusIntRationalSIOL] and the [PMinusIntRationalSIOL]. The goal of this algorithm is to compute the singular integral $P_+r(t)$ and $P_-r(t)$ respectively. Using SIOL, this algorithm can be written in two different ways:

- one calling directly a function;
- or by the concatenation of other SIOL other functions that will result in the steps described.

It is interesting to notice that the colour scheme refers directly to the sections of the SIntSIOL algorithm, where the blue boxes represent the inputs, the green represents the Block section, and the orange one is the outputs.

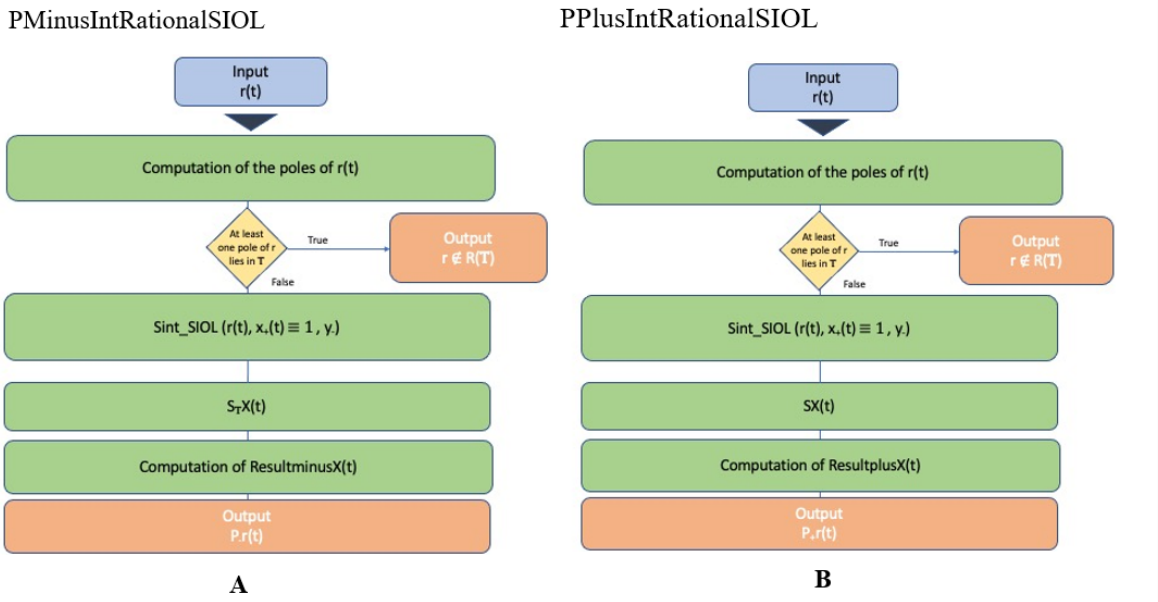


Figure 7-7 – [PMinusIntRationalSIOL] – A and [PPlusIntRationalSIOL]- B algorithm.

In **Figure 7-8**, the example shows the [PPlusIntRationalSIOL], having the rational function and the auxiliary function $x_+(t)$ declared. Since one of the building blocks of this algorithm is the SInt_SIOL, we consider $x_+(t) \equiv 1$, to obtain the correct output to the rational function $r(t)$. In the *block* section, this time, it was used the decomposition of the [PPlusIntRationalSIOL], as explained in Figure 7-7. In this case, the *Computation of the Poles*, mentioned in the flowchart, is already part of the *SInt()* call in the SIOL code. The function *printPPlusR(t),* in the output area, is responsible for printing $P_+r(t)$, the output of the algorithm.

Figure 7-9, has the same input as **Figure 7-8**, that executes the [PPlusIntRationalSIOL] algorithm, but this time the idea was to minimize the code and use the created function to call the [PMinusIntRationmalSIOL] algorithm, on the *block* section, directly, without having to explicitly indicate that the [SInt_SIOL] is being used. The output will follow the flowchart in **Figure 7-7** and print $P_-r(t)$.

```
Module PPlusIntRationalSIOL1a;
inputs
    r(t)=t+(1/t);
    x{+}(t)=1;
end
block
    s=computeResultPlusX(SInt(r));
end
output
    printPPlusRt();
end
```

Figure 7-8 – [PPlusIntRationalSIOL] algorithm using SIOL.

```
Module PMinusIntRationalSIOL1b;
inputs
    r(t)=t+(1/t);
end
block
    s=PMinusIntRational(r);
end
output
    printPMinusRt();
end
```

Figure 7-9 *– [PMinusIntRationalSIOL] algorithm using SIOL.*

Figure 7-10 shows the result of an execution of [PMinusIntRationalSIOL] algorithm. The Output section is represented by the selected inputs, and the output, as expected, is $P_{-}r(t) = 1/t$, and in **Figure 7-11** the output provided by the [PPlusIntRationalSIOL] example, the output as shown is $P_{+}r(t) = t$. Many other examples were created and ***Table 7-2*** shows some of them.

Input Options

r(t) = t+(1/t)

$x_{+}(t)=1$

$X(t) = r(t)x_{+}(t) = \frac{1}{t} + t$

Output

$P_{-}r(t) = \frac{1}{t}$

Figure 7-10 – [*PMinusIntrationalSIOL] algorithm Output.*

Input Options

r(t) = t+(1/t)

$x_{+}(t)=1$

$X(t) = r(t)x_{+}(t) = \frac{1}{t} + t$

Output

$P_{+}r(t)=t$

Figure 7-11 – [*PPlusIntRationalSIOL] algorithm Output.*

***Table 7-2** – [PPlusIntRationalSIOL] and [PMinusIntRationalSIOL] algorithms Examples*

	Rational Function	Output
PPlusIntRationalSIOL	$r(t) = \frac{1}{1+t^6}$	It is not possible to solve the desired problem due to the polynomial degree of the expression
PMinusIntRationalSIOL	$r(t) = t + \frac{1}{t}$	$P_{+}r(t)=t$

7.2.2. [AMinusRationalSIOL] algorithm

In **Figure 7-12,** the steps for [AMinusRationalSIOL] are described. The goal of the created algorithm was to verify if the rational function $r(t) \in R^{o}{}_{-}(T)$, as it is visible on the orange output boxes of the diagram. As well as the [PPlusIntRationalSIOL] mentioned before, this algorithm can call a direct function or use the specific SIOL functions *PminusIntRational_SIOL(r), ResultminusX(t),* and *Resultminus(t)* to obtain these results.

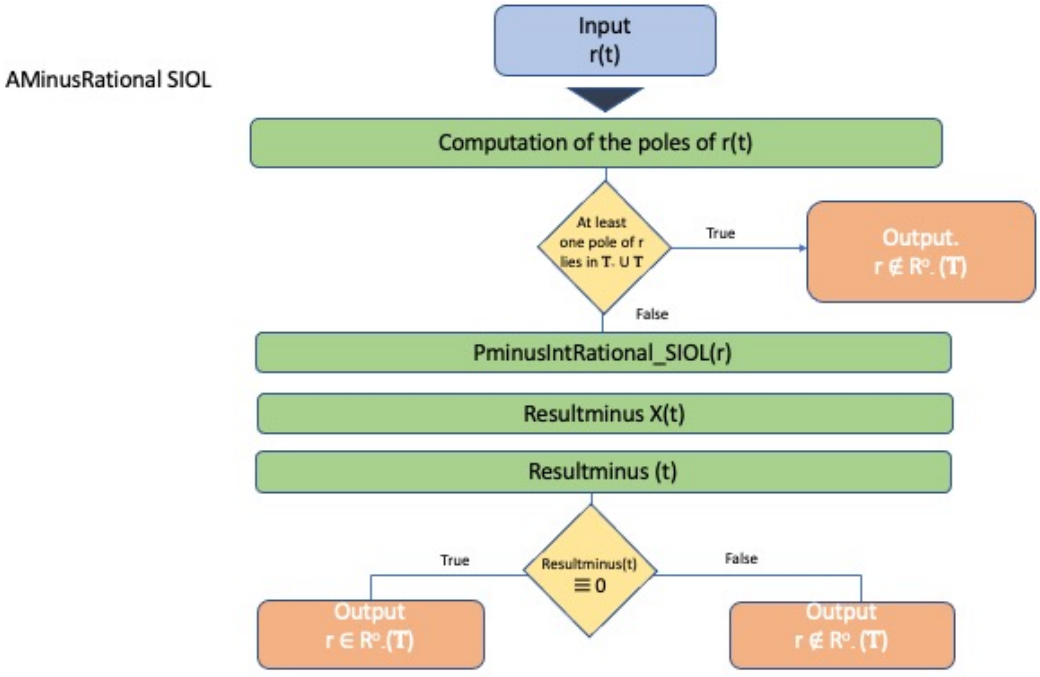


***Figure 7-12** – [AMinusRationalSIOL] algorithm.*

Figure 7-13 shows the [AMinusRationalSIOL] algorithm, in this example, it can be noted the imaginary number, *i*, here used as *I,* can also be used. Inside the *output* block, it is possible to have several function calls, representing as many outputs as desired. In this case, the use of the

functions *printOutputAPlusRational(), ResultXMinus()* and *ResultMinus()* compose the expected output.

```
Module AMinusRationalSIOL1b;
inputs
    r(t)=t/(t-(I/2));
    x{+}(t)=1;
end
block
    s=computeResultMinus(computeResultMinusX(PMinusIntRational(r)));
end
output
    printResultXMinus();
    printResultMinus();
    printOutputAMinusRational();
end
```

Figure 7-13 *– AMinusRationalSIOL using SIOL.*

Figure 7-14 shows the result of an execution of an [AMinusRationalSIOL] algorithm example, the selected input is not displayed this time, and the outputs, as expected, are *ResultMinus(t) = -1, ResultMinusX(t) = i/(-i+2t)* and also that $r(t) \notin R^{o}_{-}(\mathbb{T})$.

Besides this example, another one was created having input $r(t) = \frac{1}{5t^4+t^6-6t+1}$ and output informing that the polynomial has a exponent greater than 5.

Output

ResultMinus(t)= − 1

ResultMinusX(t)= $\frac{\mathbb{i}}{-\mathbb{i} + 2\,t}$

$r(t) \notin R^{o}_{-}(\mathbb{T})$

Figure 7-14 *– [AMinusRationalSIOL] algorithm.*

7.2.3. [APlusRationalSIOL] algorithm

In **Figure 7-15**, the algorithm follows the idea of the [AMinusRationalSIOL] algorithm but having as a final result an output showing if the rational function belongs to $R_{+}(\mathbb{T})$.

Figure 7-16, shows the implementation of [APlusRationalSIOL], where the result for the execution is called, using the SIOL function *printOutputAPlusRational()*.

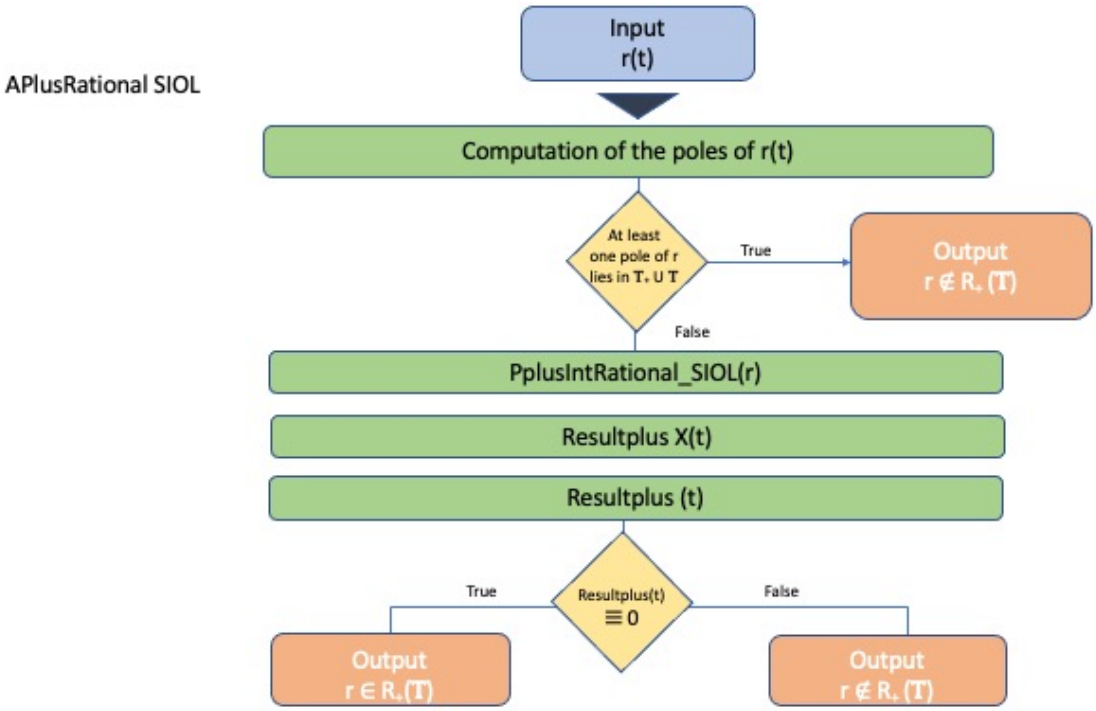


***Figure 7-15** – [APlusRationalSIOL] algorithm.*

```
Module APlusRationalSIOL1a;
inputs
    r(t)=t+(1/t);
end
block
    s=APlusRational(r);
end
output
    printOutputAPlusRational();
end
```

***Figure 7-16 – [**APlusRationalSIOL] algorithm.*

Figure 7-17 shows the result of an execution of a [APlusRationalSIOL] example, the selected input is displayed this time, and the output, as expected, is $r(t) \in R_+(T)$.

Input Options

r(t) = t/(t−2)

$x_+(t)=1$

$X(t) = r(t)x_+(t) = 1 + \frac{2}{-2+t}$

Output

$r \in R_+(T)$

***Figure 7-17** – [APlusRationalSIOL] Output.*

Other examples were created using the data displayed in ***Table 7-3***.

***Table 7-3** – [AMinusRationalSIOL] algorithm examples.*

Input	Output
$\boldsymbol{r(t) = \frac{t}{t-2}}$	$r(t) \in R_{+}(T)$
$\boldsymbol{r(t) = \frac{t}{t^6+1}}$	It is not possible to solve the desired problem due to the polynomial degree of the expression

7.3.Results

In this section, the results of the algorithms built with SIOL and their correctness are analysed. To test the accuracy and proficiency of the generated codes by SIOL, the results of the tests are displayed in **Table 7-5**. Each line of the table is the execution of one of the created algorithms, and the columns are the results obtained. As mentioned before, it is not always possible to obtain the result due to the degree of the involving polynomials of the inputs. The first column indicates the name of the example. The second informs if the algorithm was able to display an expression as result, because sometimes it cannot due to the position of the poles of the inputs. The third indicates if one of the inputs was not valid functions, and consequently the output will not be an expression but a message pointing out which of the inputs was wrong and why. The fourth column indicates if it is an input expression with an exponent higher than 5, in which case the result will display a message explaining this impossibility from an explicit output.

***Table 7-4** – Results.*

	Output is an expression	**$r(t) \notin R(T)$ $x_{+}(t) \notin R(T)$ $y_{-}(t) \notin R(T)$**	**Polynomial with degree higher than 5**
SInt1	√		
SInt2		√	
SInt3		√	
SInt4		√	
SInt5			√
SInt6		√	
PPlusMinusIntSIOL		√	
AMinusRationalSIOL1	√		

AMinusRationalSIOL2			√
APlusRationalSIOL1	√		
APlusRationalSIOL2	√		
APlusRationalSIOL3			√
PMinusIntRationalSIOL1	√		
PMinusIntRationalSIOL2			√
PPlusIntRationalSIOL1	√		
PPlusIntRationalSIOL2			√

From the results displayed in **Table 7-5**, it is possible to affirm that the developed language performs as expected. It displays the correct results, even when they are warnings or information to the user regarding the validity of their inputs. Furthermore, to corroborate these results the same tests were performed on a solution generated directly in Wolfram Mathematica, being the outputs coincident for the cases with explicit output achieved.

8.Conclusions and Future Work

The development of operator theory is stimulated by the need to solve problems emerging from several fields in mathematics and physics, as mentioned before.

The language developed in this work, SIOL, includes a syntax accessible to mathematicians, making it easier and more efficient compared to the use of a general-purpose language (GPL). It shows itself as innovative with a big potential to facilitate the design of a new set of operator theory algorithms, since it is the first textual language tailored to this specific area. It allows the end-user to abstract the models and algorithms to a higher-level language and validates the expressions according to operator theory rules, creating a more user-friendly environment to work in, which we consider the most important feature of this language.

The design of the created DSL and its analytical algorithms are focused on the possibility to implement on a computer all, or a significant part, of the extensive symbolic and numeric calculations, present in the algorithms. The developed methods rely on innovative techniques of operator theory and have the potential to be extended to more complex and general problems. This was the main reason to select them as a basis for the first version of this DSL.

In the future, upon creating this language, we hope that this work within operator theory, and *Mathematica*, will help in the design and implementation of several other analytical algorithms, with numerous applications in many areas of research and technology.

The reason to propose this new DSL, SIOL, is due to our common opinion that the design and implementation of analytical algorithms working with singular integral operators defined on the unit circle can constitute a very interesting research work. In section 7.3, there is a mention about the impossibility of using algorithms with an exponent bigger than 5, due to a limitation on the initial version of the algorithm [SInt] (Conceição et al., 2013). This problem was solved on the newly created version of the algorithm $[\text{SInt}]_{2.0}$ for *Mathematica* (Conceição & Pires, 2021). However, it was not implemented in this work due to time constraints between the conclusion of the other parallel project.

As future work, we are considering the design and implementation of other factorization, spectral, and kernel algorithms, as well as associated with other concepts of operator theory. There are other features than the ones already started to implemented, related to other analytical algorithms within the field of operator theory that are very relevant and should be considered in the an improved version of the SIOL.

9.References

Aris, R. (1995). *Mathematical Modelling Techniques*. Dover Books on Computer Science.

Barisic, A., Amaral, V., Goulão, M., & Barroca, B. (2011). *Quality in Use of Domain Specific Languages: a Case Study*. SPLASH '11: Conference on Systems, Programming, and Applications: Software for Humanity. https://doi.org/10.1145/2089155.2089170

Bussieck, M., & Nelißen, F. (2020). *Quick Start Tutorial*. https://www.gams.com/latest/docs/UG_TutorialQuickstart.html

Çağdaş, V., & Stubkjær, E. (2011). *Design research for cadastral systems. Computers, Environment and Urban Systems*. Computers, Environment and Urban Systems, *35*, 77–87. https://doi.org/10.1016/j.compenvurbsys.2010.07.003

Chen, D., Batson, R. G., & Dang, Y. (2009). *Applied Integer Programming*. Dover Publications, INC.

Cole, R., Purao, S., Rossi, M., & Sein, M. K. (2005). *Being Proactive: Where a Action Research Meets Design Research.* Proceedings of the Twenty Sixth International Conference on Information Systems, 325–336. https://doi.org/10.1016/j.compenvurbsys.2010.07.003

Conceição, A. C. (2007). *Factorization of Some Classes of Matrix Functions and its Applications* (in portuguese). *University of Algarve* .

Conceição, A. C. (2021). *The Design of New Operator Theory Algorithms*. SymComp2021

Conceição, A. C., Kravchenko, V. G., & Pereira, J. C. (2013). *Computing some classes of Cauchy type singular integrals with Mathematica software*. Advances in Computational Mathematics, *39*(2). https://doi.org/10.1007/s10444-012-9279-7

Conceição, A. C., Kravchenko, V. G., & Pereira J.C. (2012). *Rational functions factorization algorithm: a symbolic computation for the scalar and matrix cases*. Proceedings of the 1st National Conference on Symbolic Computation in Education and Research, 13.

Conceição, A. C., Marreiros, R. C., & Pereira, J. C. (2016). *Symbolic computation applied to the study of the kernel of a singular integral operator with non-Carleman shift and conjugation* . Math. Comput. Sci., 10, 365–386. https://doi.org/10.1007/s11786-016-0271-3

Conceição, A. C., & Pires, J. C. (2022). *Symbolic Computation Applied to Cauchy Type Singular Integrals*. Mathematical and Computational Applications*, 27(1), 3*. https://doi.org/10.3390/mca27010003

Conceição A.C. (2020). *Symbolic Computation Applied to the Study of the Kernel of Special Classes of Paired Singular Integral Operators*. Mathematics in Computer Science. https://doi.org/10.1007/s11786-020-00463-3

Conceição A.C., Kravchenko V.G., & Pereira J.C. (2010). *Factorization Algorithm for Some Special Non-rational Matrix Functions*. Topics in Operator Theory, Operator Theory: Advance and Applications, *202*, 87–109. http://dx.doi.org/10.1007/978-3-0346-0158-0_6

Conceição A.C., & Pereira J.C. (2016). *Exploring the Spectra of Some Classes of Singular Integral Operators with Symbolic Computation*. Mathematics in Computer Science, 10(2). https://doi.org/10.1007/s11786-016-0264-2

Conceição A.C., & Pereira J.C. (2017). *Using Wolfram Mathematica in spectral theory* . Proceedings Fof the Third International Conference on Numerical and Symbolic Computation: Developments and Applications, 295–304.

Drechsler, A., & Hevner, A. (2016). *A four-cycle model of IS design science research: capturing the dynamic nature of IS artifact design*. 11th International Conference on Design Science Research in Information Systems and Technology (DESRIST), 1–8.

Efftinge, S. (2006). *Language Engineering Made Easy!* . http://www.eclipse.org/xtext/.

Efftinge, S. (2021). *Xtend - Modernized Java*. https://www.eclipse.org/xtend/.

Erdweg, S., Storm, T.V., Völter, M., Boersma, M., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G.D., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V.A., Visser, E., Vlist, K.V., Wachsmuth, G., & Woning, J.V. (2013). *The State of the Art in Language Workbenches* - Conclusions from the Language Workbench Challenge. SLE..

Howe, D. *(2022) FOLDOC - Free-Online Dictionary of Computing*. (2022). http://foldoc.org/abstract+syntax

Fourer, R. (1998). Extending a General-Purpose Algebraic Modeling Language to Combinatorial Optimization: A Logic Programming Approach. *Operations Research/Computer Science Interfaces Series Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search*, 31–74. http://dx.doi.org/10.1287/ijoc.14.4.322.2825

Fowler, M. (2006). *Language Workbenches: The Killer-App for Domain Specific Languages*. http://www.martinfowler.com/articles/languageWorkbench.html

Gohberg, I., & Krupnik, N. (1992). *One-Dimensional Linear Singular Integral Equations*. Operator Theory: Advances and Applications, *54*. Springer Basel AG

Guntli, C. (2010). Create a DSL in Eclipse. *University of Applied Science in Rapperswil*.

Heid, M. K., & Edwards, M. T. (2001). *Computer Algebra Systems: Revolution or Retrofit for Today's Mathematics Classrooms*. Theory Into Practice, 40, 128–136. https://doi.org/https://doi.org/10.1207/s15430421tip4002_7

Hevner, A. (2007). A Three Cycle View of Design Science Research. *Scandinavian Journal of Information Systems*, *19*(2).

Ionescu, C., & Jansson, P. (2016). *Domain-Specific Languages of Mathematics: Presenting Mathematical Analysis Using Functional Programming*. TFPIE.

Jarvinen, P. (2007). *Action Research is Similar to Design Science*. Quality & Quantity, 37–54. https://doi.org/https://doi.org/10.1007/s11135-005-5427-1

Jha, R., Samuel, A., Pawar, A., & Kiruthika, M. (2013). *A Domain-Specific Language for Discrete Mathematics*. International Journal of Computer Applications, 6–19. https://doi.org/https://doi.org/10.5120/12036-7257

Litvinchuk, G. S., & Spitkovskii, I. M. (1987). *Factorization of Measurable Matrix Functions*. Operator Theory: Advances and Applications, 25. Springer Basel AG

Lutkevich, B. (2021). *Systems Thinking*. https://www.techtarget.com/searchcio/definition/systems-thinking *Https://Searchcio.Techtarget.Com/Definition/Systems-Thinking*.

Malik, M. A. (1998). *Evolution of the high level programming languages*. ACM SIGPLAN Notices, 33, 72–80. https://doi.org/10.1145/307824.307882

Martins, P. V., & Conceição, A. C. (2017). *Implementing mathematical models for singular integrals*. *SYMCOMP 2017*.

Mikhail Barash. (2018). *Grammars for programming languages*. https://medium.com/@mikhail.barash.mikbar/grammars-for-programming-languages-fae3a72a22c6

Mooij A., & Hooman J. (2017). *Creating a Domain Specific Language (DSL) with Xtext*. Radboud University, 1–37. http://www.cs.kun.nl/J.Hooman/DSL

Paarsch, H. J., & Golyaev, K. (2016). *A Gentle Introduction to Effective Computing in Quantitative Research*. MIT Press

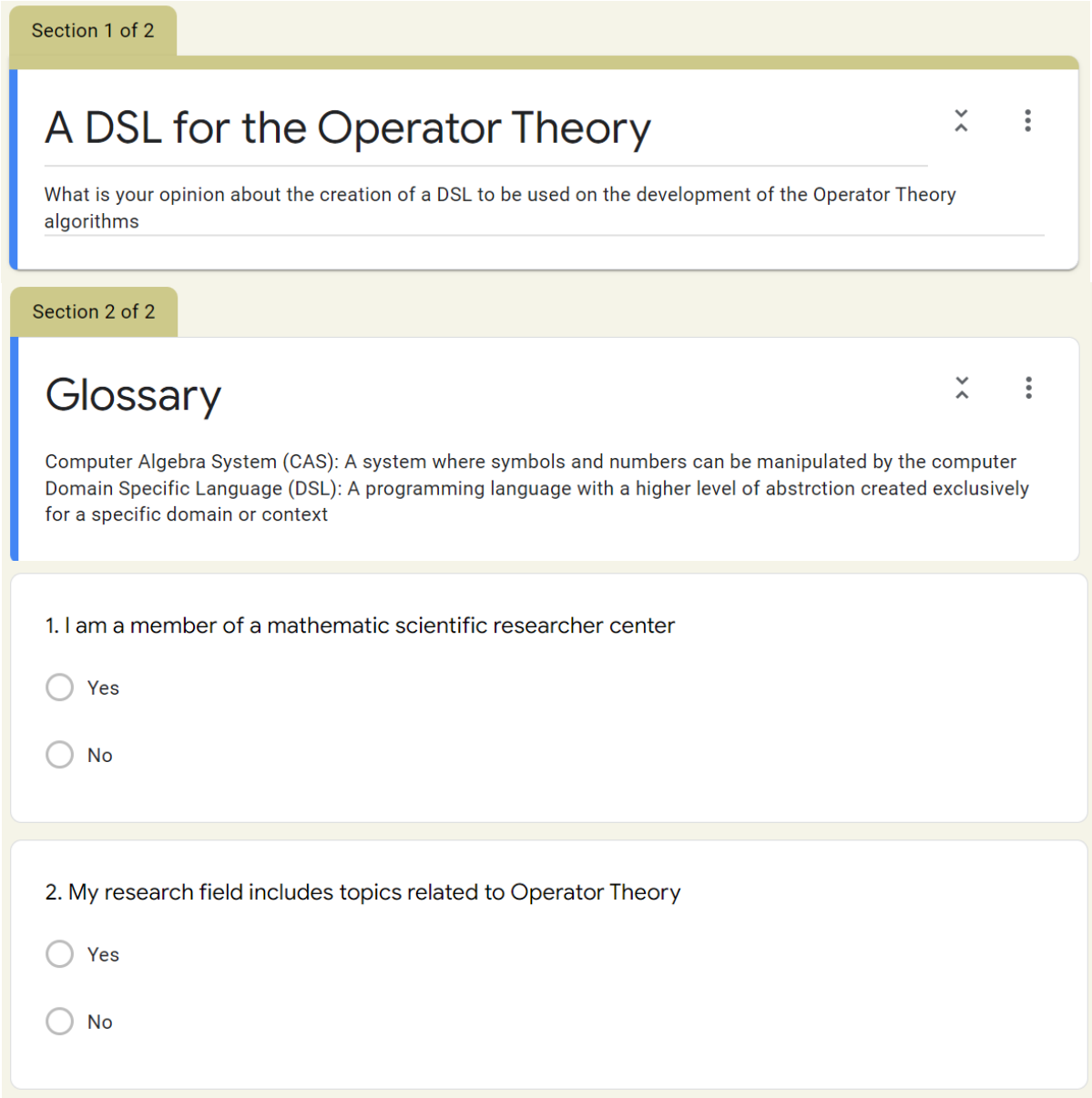
Ropes, D. (2018). *Design Science Research:Bridging the rigor - relevance gap*. https://www.iiis.org/iiis/2018Videos/Spring/TSnrdUrsPLk.pdf.

Simon, H. (1996). *The Sciences of Artificial, 3rd Edition, MIT Press*.

Wolfram, S. (2011, March 6). *Intelligence and the Computational Universe [Online Presentation]*. https://www.youtube.com/watch?v=DJ0WG3D3m1U

Appendixes

Appendix A – Survey questionnaire



3. I have computer programming knowledge

- ○ Advanced
- ○ Normal
- ○ Beginner
- ○ None

4. I know about the programming language Wolfram Mathematica Language

- ○ Yes
- ○ No

5. I would like to learn more about computing programming

- ○ I neither agree nor disagree
- ○ I completely agree
- ○ I agree
- ○ I disagree
- ○ I completely disagree

6. It would be interesting (in terms of investigation) to have better knowledge of computer programming

- ○ I neither agree nor disagree
- ○ I completely agree
- ○ I agree
- ○ I disagree
- ○ I completely disagree

7. I use Computer Algebra Systems (CAS) on my research tasks

- ○ Yes
- ○ Sometime
- ○ Never
- ○ I do not know the Computer Algebra Systems concept

8. I would like to use more often Computer Algebra Systems (CAS) on my research tasks

- ○ I neither agree nor disagree
- ○ I completely agree
- ○ I agree
- ○ I disagree
- ○ I completely disagree

9. I use the Computer Algebra System Wolfram Mathematica on my research tasks

- ○ Yes
- ○ Sometimes
- ○ Never
- ○ I do not know the Computer Algebra Systems concept

10. I believe that additional computer programming knowledge would allow me to improve the results of my research tasks

- ○ I neither agree nor disagree
- ○ I completely agree
- ○ I agree
- ○ I disagree
- ○ I completely disagree

11. I know that there are techniques for calculating singular integrals implemented with the Wolfram Mathematica language

- ○ Yes
- ○ No

12. I know that there are algorithms, implemented with the Wolfram Mathematica Language, for matrix functions factorizations

- ○ Yes
- ○ No

13. I know the concept of Domain Specific Language (DSL)

- ○ Yes, I am an expert
- ○ Yes, but I am not an expert
- ○ No, but I would like to know more
- ○ No

14. I would use a high-level programming language to solve problems in Operator Theory, that can be used without the support of a programming expert

- ○ I neither agree nor disagree
- ○ I completely agree
- ○ I agree
- ○ I disagree
- ○ I completely disagree

15. I consider that the existence of a high-level programming language to solve problems in Operator Theory, that can be used without the support of a programming expert,

	I neither agree ...	I completely ag...	I agree	I disagree	I completely di...
15.1. would be ...	○	○	○	○	○
15.2. would be ...	○	○	○	○	○
15.3 would pro...	○	○	○	○	○
15.4 will allowe...	○	○	○	○	○

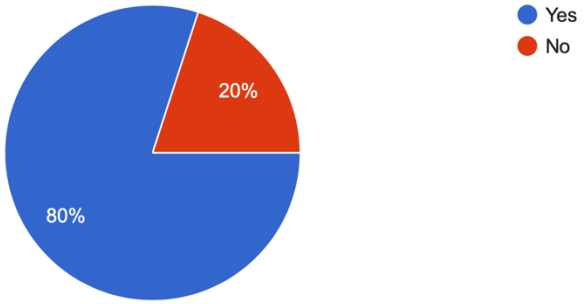
16. I would use a language that allows, without the support of a programming expert,

	I neither agree ...	I completely ag...	I agree	I disagree	I completely di...
16.1 the compu...	○	○	○	○	○
16.2 to solve sp...	○	○	○	○	○
16.3 the study ...	○	○	○	○	○
16.4 the study ...	○	○	○	○	○

Appendix B– Survey participants answers

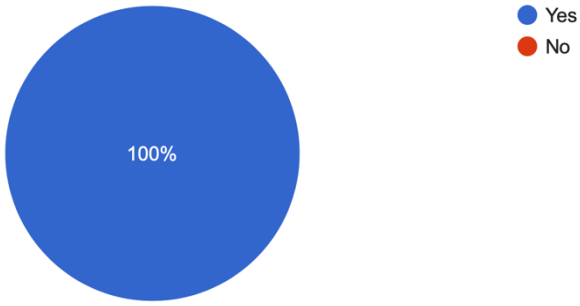
1. I am a member of a mathematic scientific researcher center

25 respostas



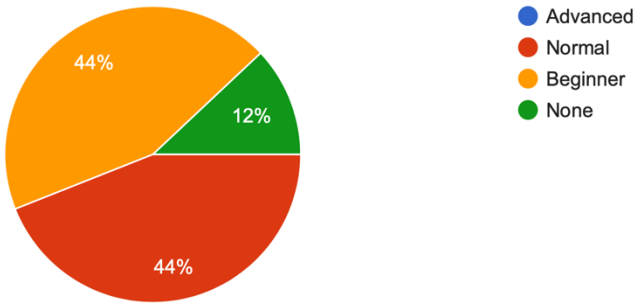
2. My research field includes topics related to Operator Theory

25 respostas



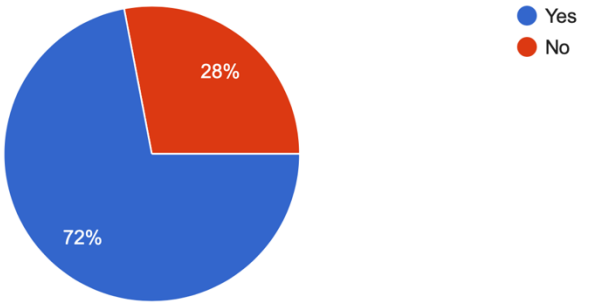
3. I have computer programming knowledge

25 respostas



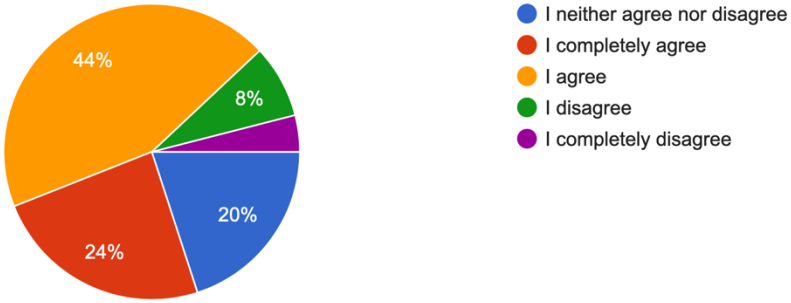
4. I know about the programming language Wolfram Mathematica Language

25 respostas



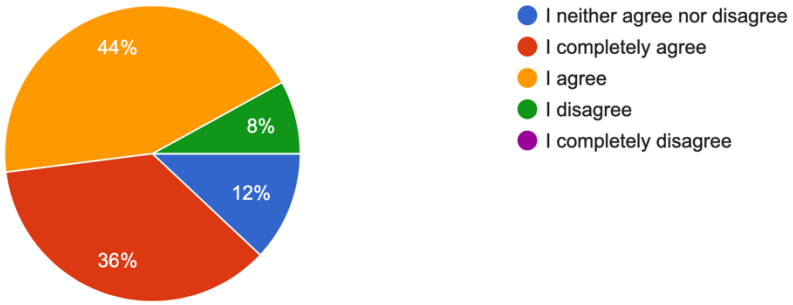
5. I would like to learn more about computing programming

25 respostas



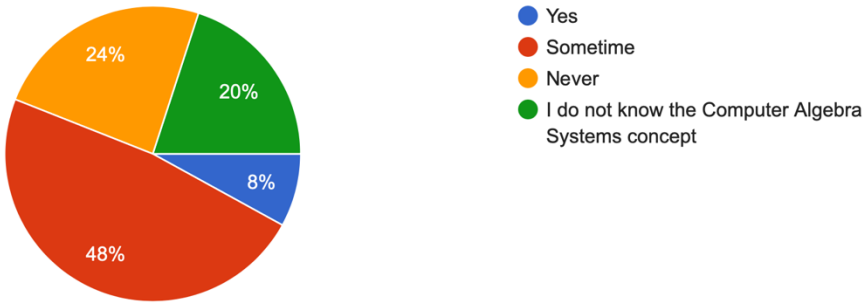
6. It would be interesting (in terms of investigation) to have better knowledge of computer programming

25 respostas



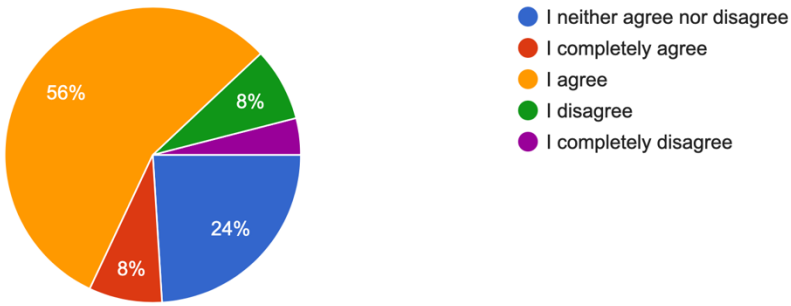
7. I use Computer Algebra Systems (CAS) on my research tasks

25 respostas



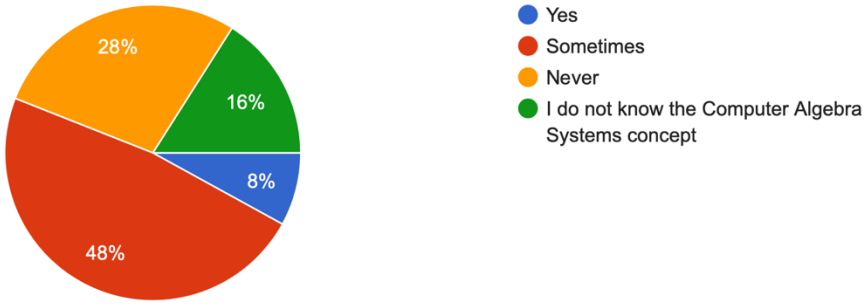
8. I would like to use more often Computer Algebra Systems (CAS) on my research tasks

25 respostas



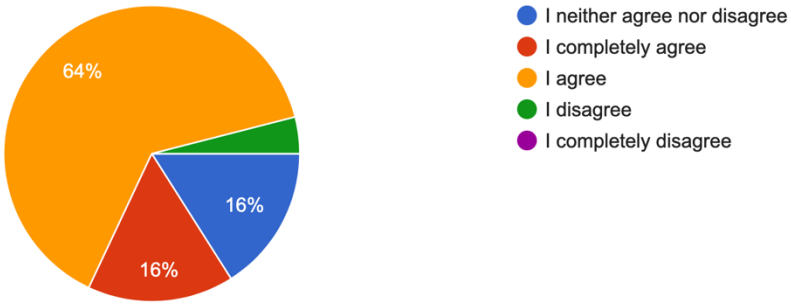
9. I use the Computer Algebra System Wolfram Mathematica on my research tasks

25 respostas



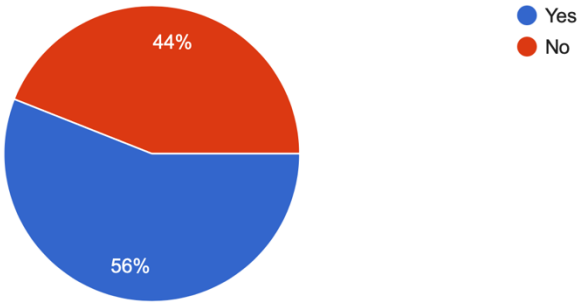
10. I believe that additional computer programming knowledge would allow me to improve the results of my research tasks

25 respostas



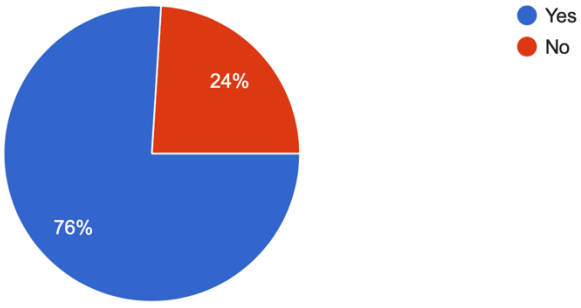
11. I know that there are techniques for calculating singular integrals implemented with the Wolfram Mathematica language

25 respostas



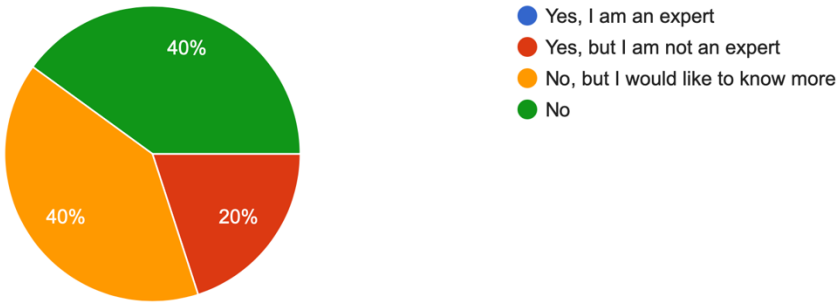
12. I know that there are algorithms, implemented with the Wolfram Mathematica Language, for matrix functions factorizations

25 respostas



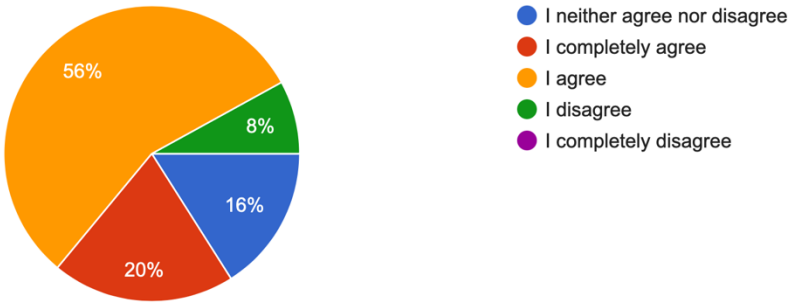
13. I know the concept of Domain Specific Language (DSL)

25 respostas

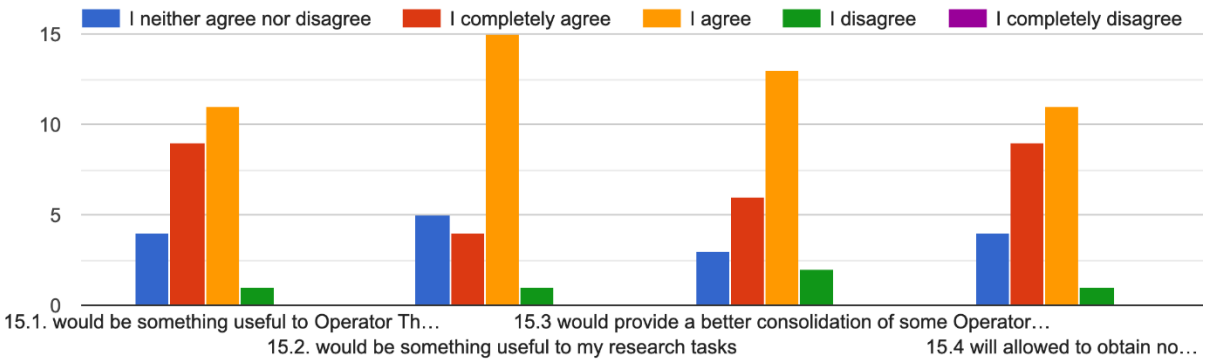


14. I would use a high-level programming language to solve problems in Operator Theory, that can be used without the support of a programming expert

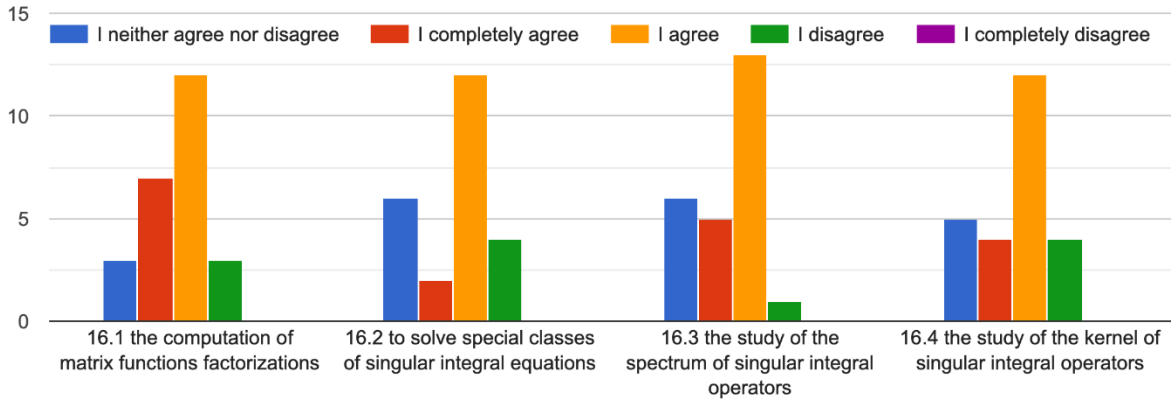
25 respostas



15. I consider that the existence of a high-level programming language to solve problems in Operator Theory, that can be used without the support of a programming expert,



16. I would use a language that allows, without the support of a programming expert,



Appendix C – Wolfram Mathematica Code generated by SIOL

```
SetDirectory[NotebookDirectory[]];
Clear[Evaluate[Context[] <> "*"]];
ruleF = {FractionBox[x_, y_] -> x/y};
inicio0=AbsoluteTime[];
hasError = False;
start0=AbsoluteTime[];
biggestRExp=0;
biggestXExp=0;
biggestYExp=0;
possibleExponents:=True;
R[t_] = t+(1/t);
Xplus[t_] = 1/t;

(* ++++++++++ Computation of the Poles of r(t) +++++++++++++ *)
biggestRExp = Exponent[Denominator[R[t]], t];
biggestXExp = Exponent[Denominator[Xplus[t]], t];
If[((biggestRExp > 5) || (biggestXExp > 5) || (biggestYExp > 5)),
   possibleExponents=False;
   hasErrors=True;
];
rOver = Table[Abs[Root[Denominator[( #1+(1/#1))] &, i]] == 1, {i, biggestRExp}];
rHasPolesOver = AnyTrue[rOver, TrueQ];
If[(rHasPolesOver == False),
(* +++++++ SINT_SIOL +++++++++ *)
(* Initialization of r(t), P+ e P- *)
Clear[Pplus];
Pplus[x_ + y_] := Pplus[x] + Pplus[y];
Pplus[a_ x_] := a Pplus[x] /; FreeQ[a, t];
Pplus[t^n_] := 0 /; n < 0;
Pplus[(t + a_)^n_] := 0 /; Abs[a] < 1 && n < 0;
Pplus[(a_ + b_ t)^n_] := 0 /; Abs[a/b] < 1 && n < 0;
Pplus[Xplus[t]*t^n_] := Xplus[t]*t^n - \!\(
\*UnderoverscriptBox[\(\[Sum]\), \(i = 1\), \(-n\)]\((
\*FractionBox[\((D[Xplus[t], {t, i - 1}] /.
      t -> 0)\), \(\((i - 1)\)!\)]
\*SuperscriptBox[\(t\), \(i + n - 1\)]\)\) /; n < 0;
Pplus[Xplus[t]*(t + a_)^n_] := Xplus[t]*(t + a)^n - \!\(
\*UnderoverscriptBox[\(\[Sum]\), \(i = 1\), \(-n\)]\((
\*FractionBox[\((D[Xplus[t], {t, i - 1}] /.
```

```
        t -> \(-a\)\), \(\((i - 1)\)!\)]
\*SuperscriptBox[\((t + a)\), \(i + n - 1\)]\)\) /;
  Abs[a] < 1 && n < 0;
Pplus[Xplus[t]*(a_ + b_ t)^n_] := b^n*Pplus[Xplus[t]*(t + a/b)^n];
Pplus[Yminus[t]] := Conjugate[Yplus[0]];
Pplus[Yminus[t]*t^n_] := 0 /; n < 0;
Pplus[Yminus[t]*t^n_.] := \!\(
\*UnderoverscriptBox[\(\[Sum]\), \(i = 0\), \(n\)]\(
\*FractionBox[\(Conjugate[D[Yplus[t], {t, i}] /. t -> 0]\), \(i!\)]
\*SuperscriptBox[\(t\), \(n - i\)]\)\) /; n > 0;
Pplus[Yminus[t]*(t + a_)^n_] := 0 /; Abs[a] < 1 && n < 0;
Pplus[Yminus[t]*(t + a_)^n_] := \!\(
\*UnderoverscriptBox[\(\[Sum]\), \(i = 1\), \(-n\)]\((
\*FractionBox[\((D[Yminus[t], {t, i - 1}] /.
        t -> \(-a\)\), \(\((i - 1)\)!\)]
\*SuperscriptBox[\((t + a)\), \(i + n - 1\)]\)\) /;
  Abs[a] > 1 && n < 0;
Pplus[Yminus[t]*(a_ + b_ t)^n_] := b^n*Pplus[Yminus[t]*(t + a/b)^n];
(*  *)
Pminus[t_]=t-Pplus[t];
Pminus[R[t]]=R[t]-Pplus[R[t]];
If[((biggestRExp < 5) && (biggestXExp < 5) && (biggestYExp < 5)),
    Pplus[x_]:=x;
,
    Pplus[t] := t;
    Pplus[t^n_] := t^n /; n > 0;
    Pplus[a_] := a /; FreeQ[a, t];
    possibleExponents:=False;
];

(* Compute de Decomposition of r(t) *)
LeadingCoeff[poly_, t_] := Coefficient[poly, t, Exponent[poly, t]];
decompR = Function[{}, den[t_] = Denominator[R[t]];
                                    a = LeadingCoeff[den[t], t];
                                    solP = Solve[den[t] == 0, t];(* List of rules: t ->
  pole *)
                                    p = Tally[solP[[All, 1, 2]]]; (*
  List of poles and multiplicities *)
                                    n = Length@p;
                                    denominator[t_] = a*\!\(
\*UnderoverscriptBox[\(\[Product]\), \(j = 1\), \(n\)]
\*SuperscriptBox[\((t - p[\([\)\(j, 1\)\(]\)]\), \(p[\([\)\(j,
```

```
    2\)\(]\)]\)]\);
                                RFact[t_] = Numerator[R[t]]/denominator[t];
                                Apart[RFact[t]]
  ];
decR[t_] = decompR[];
(* Computation of the poles of x+ *)
biggestXExp = Exponent[Denominator[Xplus[t]], t];
xIn = Table[Abs[Root[Denominator[(1/#1)] &, i]] < 1, {i, biggestXExp}];
xHasPolesInside = AnyTrue[xIn, TrueQ];
xOver = Table[Abs[Root[Denominator[(1/#1)] &, i]] == 1, {i, biggestXExp}];
xHasPolesOver = AnyTrue[xOver, TrueQ];
xOut = Table[Abs[Root[Denominator[(1/#1)] &, i]] > 1, {i, biggestXExp}];
xHasPolesOutside = AnyTrue[xOut, TrueQ];

(* Define X(t) *)
X[t_] = Expand[decR[t]*Xplus[t]];
If[((xHasPolesInside === False) && (xHasPolesOver === False)),
   (* Computation of PplusX(t) *)
   PplusX[t_] = Simplify[Pplus[X[t]]];
   SX[t_] = Simplify[2*PplusX[t] - X[t]];
,
   hasError = True;
];
(* Define Y(t) *)
Y[t_] = Expand[decR[t]*Yminus[t]];
(* Computation of PplusY(t) *)
PplusY[t_] = Simplify[Pplus[Y[t]]];
SY[t_] = Simplify[2*PplusY[t] - Y[t]];
(* ++++++++++++++ End of SInt-SIOL++++++++++++++++++*)
,
   hasError = True;
];

(* +++++++ PMinusIntRational_SIOL +++++++++ *)
ResultminusY[t_] = Simplify[1/2 (Y[t] - SY[t] /. ruleF)];

(* +++++++ PPlusIntRational_SIOL +++++++++ *)
ResultplusY[t_] = Simplify[1/2 (SY[t] + Y[t] /. ruleF)];

(* +++++++ PMinusIntRational_SIOL +++++++++ *)
ResultminusX[t_] = Simplify[1/2 (X[t] - SX[t] /. ruleF)];
```

```
(* +++++++ PPlusIntRational_SIOL +++++++++ *)
ResultplusX[t_] = Simplify[1/2 (SX[t] + X[t] /. ruleF)];

fim0=AbsoluteTime[];
printFormat = {Xplus -> SubPlus[x], Yplus -> SubPlus[y], Yminus -> SubMinus[y], Conjugate[aa_] ->
\!\(\*OverscriptBox[\(aa\), \(_\)]\)};
Print[Text[Style["Input Options", Black, Bold, 36]]];
Print[Text[Style["r(t) = t+(1/t)", Black, 20]]];
Print[Text[
  Style["\!\(\*SubscriptBox[\(x\), \(+\)]\)(t)=1/t", Black, 26]]];
If[biggestRExp < 5,
    Print[Text[
      Style["X(t) = r(t)\!\(\*SubscriptBox[\(x\), \(+\)]\)(t) =", Black,
       26]], Style[X[t] //.printFormat , 26]];
];
Print[""];
Print[Text[Style["Output", Black, Bold, 36]]];
If[(possibleExponents == False),
    hasError = True;
    Print[Text[
      Style[
       "It is not possible to solve the desired problem due to the \
polynomial degree of the expression", Black, 26]]];
];
If[(hasError == False),
    Print[Text[Style["ResultMinusX(t)=", Black, 26]],
     Style[ResultminusX[t] //.printFormat , 26]];
];


If[(hasError == False),
    Print[Text[Style["ResultPlusX(t)=", Black, 26]],
     Style[ResultplusX[t] //.printFormat , 26]];
];

If[(hasError == False),
    Print[Text[Style["ResultMinusY(t)=", Black, 26]],
     Style[ResultminusY[t] //.printFormat , 26]];
];

If[(hasError == False),
    Print[Text[Style["ResultPlusY(t)=", Black, 26]],
```

```
     Style[ResultplusY[t] //.printFormat , 26]]];
If[((rHasPolesOver == True) && (hasError == True) && (possibleExponents == True)),
   Print[Text[
     Style["r(t) has at least one of its poles over the unit circle",
      Black, 26]]];
   Print[Text[
     Style[
      "r(t)\[NotElement]\!\(\*SubscriptBox[\(R\), \
\(\[DoubleStruckCapitalT]\)]\)", Black, 26]]];
   Print[""];
];
If[(((xHasPolesInside == True)||(xHasPolesOver == True)) && (hasError == True) && (possibleExponents ==
True)),
   Print[Text[
     Style[
      "\!\(\*SubscriptBox[\(x\), \
\(+\)]\)(t)\[NotElement]\!\(\*SubscriptBox[\(R\), \(+\)]\)(\
\[DoubleStruckCapitalT])", Black, 26]]];
   Print[Text[
     Style[
      "At least one pole of \!\(\*SubscriptBox[\(x\), \(+\)]\)(t) lies \
in \[DoubleStruckCapitalT] \[Union] \!\(\*SubscriptBox[\(\
\[DoubleStruckCapitalT]\), \(+\)]\)", Black, 26]]];
   Print[""];
];
```

Appendix D – User's Manual

Step 0: All the versions used in this project are in **Table 5-1**.

Step 1: Eclipse Xtext is implemented in Java, so you must have a Java Runtime Environment installed in order to proceed.

Step 2: Install Eclipse (https://eclipse.org/downloads/). A pre-configured Eclipse distribution is available which has already all the necessary plug-ins installed.

Step 3: Check if the correct version of Xtext is installed.

Step 3.1: On Eclipse menu select Help > About Eclipse IDE > Verify on the available icons if Xtext exists.

Alternative Step 3: If the Xtext is not installed, it can be installed from Eclipse Marketplace. On Eclipse Menu select Help > Eclipse Market Place > Select tools on the dropdown list > On find search for "*xtext*"> On the result list find *Eclipse Xtext* and *Eclipse Xtend* > Select install

Step 4: Add the DSL Files: SIOL.xtext, SIOLGenerator.xtend and SIOLValidator.java.

Step 5: On SIOL.xtext, select "Run As…" > "Generate Xtext Artifacts".

Step 6: Launch Runtime Eclipse.