Durham E-Theses

Routines and Applications of Symbolic Algebra Software

PRICE, DOMINIC,THOMAS

How to cite:

PRICE, DOMINIC,THOMAS (2023) *Routines and Applications of Symbolic Algebra Software*, Durham theses, Durham University. Available at Durham E-Theses Online: http://etheses.dur.ac.uk/14811/

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a link is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, Durham University, University Office, Old Elvet, Durham DH1 3HP
e-mail: e-theses.admin@dur.ac.uk Tel: +44 0191 334 6107
http://etheses.dur.ac.uk

Routines and Applications of Symbolic Algebra Software

Dominic Price

A thesis presented for the degree of
Doctor of Philosophy

Department of Mathematical Sciences
University of Durham

August 2022

Abstract

Computing has become an essential resource in modern research and has found application across a wide range of scientific disciplines. Developments in symbolic algebra tools have been particularly valuable in physics where calculations in fields such as general relativity, quantum field theory and physics beyond the standard model are becoming increasing complex and unpractical to work with by hand. The computer algebra system *Cadabra* is a tensor-first approach to symbolic algebra based on the programming language Python which has been used extensively in research in these fields while also having a shallow learning curve making it an excellent way to introduce students to methods in computer algebra.

The work in this thesis has been concentrated on developing Cadabra, which has involved looking at two different elements which make up a computer algebra program. Firstly, the implementation of algebraic routines is discussed. This has primarily been focused on the introduction of an algorithm for detecting the equivalence of tensorial expressions related by index permutation symmetries. The method employed differs considerably from traditional canonicalisation routines which are commonly used for this purpose by using Young projection operators to make such symmetries manifest.

The other element of writing a computer algebra program which is covered is the infrastructure and environment. The importance of this aspect of software design is often overlooked by funding committees and academic software users resulting in an anti-pattern of code not being shared and contributed to in the way in which research itself is published and promulgated. The focus in this area has been on implementing a packaging system for Cadabra which allows the writing of generic libraries which can be shared by the community, and interfacing with other scientific computing packages to increase the capabilities of Cadabra.

Contents

Declaration

The work in this thesis is based on research carried out at the Department of Mathematical Sciences, Durham University, United Kingdom. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Copyright © 2022 by Dominic Price

The copyright of this thesis rests with the author. No quotations from it should be published without the author's prior written consent and information derived from it should be acknowledged

Acknowledgements

I would first of all like to thank my supervisor Kasper Peeters not only for all his endless patience and help during the course of my research but also for all the hard work he has done over the years to forward the cause of open source software in academia, in particular for creating the Cadabra project and maintaining it for this whole time.

I must also show my appreciation to the many friends I have made over the course of my time in Durham and who made it such a memorable part of my life of which I will always hold the fondest of memories.

Finally I would be remiss to not thank the members of my family who have always pushed and encouraged me in my studies, in particular my mother but also my father and grandmother who although are no longer here in person continue by their intercession to guide my life every day.

Conventions

A variety of different programming languages, libraries and software packages are used throughout, and therefore in an attempt to make these as accessible as possible without requiring any deep prerequisite knowledge of these a convention of code highlighting is used throughout to make the analysis of code snippets simpler. These conventions will be demonstrated with Cadabra code, but a consistent convention is used for all languages.

The language used in each snippet should be made clear from context, but the left frame is also coloured according to the language:

```
# Python/Cadabra code
```

```
; Lisp code
```

```
// C/C++ code
```

```
(* Mathematica code *)
```

```
// Groovy/Redberry code
```

```
# Python/Sage code
```

A standard highlighting is also used across the languages to indicate how the syntax of the language works

- The keywords and inbuilt functions of a language are displayed in red:

  ```
  l = [2*i for i in range(3)]
  ```

- Comments are displayed in grey

  ```
  # Commented out
  ```

- Functions which are provided by a pertinent library or package, but are not inbuilt, are displayed in blue.

  ```
  expand_delta(ex)
  ```

- String literals are displayed in green:

  ```
  s = "Hello world"
  ```

- If a language supports a facility for specifying maths literals, these are displayed in yellow:

  ```
  ex = $a + b$
  ```

Definitions

A list of commonly used symbols is provided here for quick reference. Note that some symbols may be used with a different meaning in some contexts, but this will be made clear as and when they are used.

$g_{\mu\nu}$ An arbitrary metric tensor

${R^a}_{bcd}$ The Riemann tensor

R_{ab} The Ricci tensor given by ${R^c}_{bcd}$

R The Ricci scalar given by R_{aa}

W_{abcd} The Weyl tensor given in dimension n by

$$R_{iklm}+\frac{1}{n-2}(R_{im}g_{kl}-R_{il}g_{km}+R_{kl}g_{im}+R_{km}g_{il})+\frac{1}{(n-1)(n-2)}R(g_{il}g_{km}-g_{im}g_{kl})$$

∇_a Covariant derivative operator

∂_a Partial derivative operator

$T_{i_1 i_2 \ldots}$ An arbitrary tensor quantity

$T_{i_1 i_2 \ldots ; j_1 j_2 \ldots}$ Equivalent to $\nabla_{j_1 j_2 \ldots} T_{i_1 i_2 \ldots}$

$T_{[i_1 i_2 \ldots i_n] j_1 j_2 \ldots}$ Antisymmetrisation in the indices in square brackets

M_{ab} An arbitrary matrix quantity

S_n The n element symmetric group

Y An arbitrary Young tableau

$P(Y)$ A Young projection operator acting on the given tableau

$H(Y)$ A Hermitian projection operator acting on the given tableau

$[A,B]$ Commutator of A and B

[1]

— Butler Lampson

[1] *"All problems in computer science can be solved by another level of indirection"*

Chapter 1

Introduction

Ever since the modern era of computing began in the mid 20th century, it has had an enormous impact on the way we conduct scientific research. The advent of the commonplace use of computers in the home and at work is often coined the *digital revolution* and its application in the scientific community has been perhaps the most far-reaching and important revolution to the way in which we conduct research since the invention of the scientific method [1]. Not only has it given us the ability to perform calculations which were once beyond our reach, allowed for the analysis of incredibly complex systems and led to an era where proof by exhaustion has become possible [2], but the invention of the internet and the ability for knowledge to become so accessible has led to a flurry of advances at an unprecedented speed. More recently, the take-off of open-source software thanks to the pushes made by people such as Richard Stallman and Linus Torvalds and the infrastructure which they have developed (at the core of which is the GNU/Linux operating system [3] and the development of the Free Software Foundation [4]) allowed not only the results and knowledge gained from the use of computing to become widely available, but also the underlying software and tooling which has afforded the community the ability to contribute to this wealth of resources without requiring the backing of institutions to provide the expensive facilities which were once essential.

As well as providing the ability to numerically simulate complex physical systems, the advent of computing also allowed for the construction of tools for the algorithmic manipulation of algebraic expressions. This new field of computer algebra systems (CAS) developed as a result of the variety of different tools which were written for specific purposes as people explored the use of computing in applications such as symbolic integration, linear algebra, differential equation solving and the simplification of expressions.

It is taken for granted nowadays that almost all algebraic problems can be solved by the use of computer software, removing the time spent in calculating difficult integrals or expanding out long equations; by looking through the bibliography of papers from the last ten years one does not need to search long to find references to various software tools which have been used, indeed a Google Scholar search for papers mentioning *Mathematica* from the last year enumerates over 3,000 results. Further, in some disciplines of modern physics many results would be completely unfeasible to calculate without the assistance of modern computing, which can be seen from the very first uses of CAS in the early 1960s, for example Martinus J. G. Veltman's calculation of the quadrupole moment of the W-boson using his very early CAS *Schoonschip* to expand three-vertex Feynman diagrams which involved intermediate expressions involving roughly 50,000 terms [5]. Since then, the number of different fields, as well as the numbers involved, have increased enormously. Examples of different areas where computing has become important include solid state physics where total energy calculations of complicated systems such as phonons and those involving many inequivalent atoms have become possible since the 1970s [6], as well as general relativity where some systems can be examined by the use of perturbative techniques where the number of terms increases exponentially with the perturbative order (as demonstrated in the paper *Cadabra and Python algorithms in General Relativity II: Gravitational Waves* [7] which is introduced in Section 6.5) and innovative techniques in *numerical relativity* allow the study of extreme systems such as the coalescence of neutron stars and black holes [8].

Further afield, calculations in physics beyond the standard model also make heavy use of modern computational power in their calculations, most notably string theory where not only are component calculations made more expensive by the larger number of dimensions, but also the vast size of the possible 'string landscapes' has led to the need for the computation of huge sets such as the classification of all 473,800,776 reflexive polyhedra by Kreuzer and Skarke [9] as well as provoking the development of novel techniques in network science [10] and machine learning [11].

Despite the necessity of these algebra systems in research and the speed with which computing power and software has developed, cooperation between researchers in adopting standard tools and platforms on which they are able to develop and work as a community has not had anywhere near as much impetus. In some aspects this falls along the natural boundaries of different disciplines with tools for calculations in QFT clearly having different design objectives to those of algebraic topology, and it is natural that where the design objectives differ to such a great extent there is little motivation for the respective communities to consolidate their approaches. However within disciplines there is a widespread lack of consensus on the natural choice of software: calculations in general relativity can be performed on systems such as Mathematica [12], Sage [13], Maple [14], Maxima [15], Cadabra [16, 17] and GraviPy [18] to name but a few of the options, and the problem is only exacerbated by the multiple different packages available for some of these platforms designed to provide the additional functionality required for tensor calculations.

The wide array of options is in many ways positive as it displays a keen interest in development in this field and allows users to choose between power, flexibility and ease of use when selecting software. However beneath this is the much larger problem that many researchers choose to develop personal projects for dealing with their own research goals and these tools rarely enter into public view, and even in cases where they are made available they are often accompanied by a steep learning curve due to a combination of the home-brew style of their development, niche scope and an absence of documentation. This can lead to a situation where people find it easier to write their own tools in preference to learning how to use these programs.

In this way, a non negligible amount of the time spent developing software either solves a problem for which a tool already exists, or does not contribute to the overall wealth of tools available to the scientific community. There are many underlying causes for this but the separation of the concepts of software development and research output is a predominant factor as this provides an incentive to publish research papers, but there is often no direct incentive from grant bodies and institutions to promote researchers to openly publish software, spend time developing it into a user-friendly form and maintain it.

This tendency to work on personal projects has a knock-on effect on the quality of the software produced in two ways: for one, the task of developing even simple tools is non trivial and requires a range of skills which a background in physics will not necessarily provide, for another what our philosophy may dictate as being the logical way to tackle a problem may well alienate many of the people who would otherwise benefit from our work. As an example of a way in which this might surface, I refer to a section of a talk given in 2015 by Brian Kernighan [1] on *successful language design* in which he presents a simple programming problem and asks which language people might use to solve it. Among some normal suggestions such as awk or the shell, there are also more esoteric proposals such as simula and BASIC. While these may be the environments we personally find most comfortable and productive, insisting on always using our favourite language — a choice which is often motivated more by philosophy than practicality — not only results in a program which the majority of people will find it hard to read and impossible to contribute to, but which leads to the complicated and bug-prone code which using the wrong tool for the job often leads to.

I make these comments on the current position software development has in research in order to provide the backdrop against which this thesis is written, as I have come across these issues on numerous occasions during my time working in computer algebra. I therefore feel it necessary to address these issues alongside the projects I have worked on; partly in order to show the impact this has on research and partly in order to provide a motivation for some of the projects I have decided to work on which certainly do not pretend to be entirely mathematical in nature but which I have nonetheless found to be just as necessary and in some cases just as non trivial.

Now that I have explained the philosophy behind my writing I will comment on how I have decided to structure the various projects which make up this thesis. The content is divided between two main parts: the first *Algorithms in Symbolic Algebra* on the technicalities of some specific algorithms and core functionality I have been involved with implementing and comparing this with the tools available in other CAS packages, and the second *Software in Academia* on collaborative projects I have been involved with using Cadabra, as well as parts of my research which have not been centred on functionality but instead of accessibility and attempting to break down the barrier to entry into the field.

In the first part I will begin by offering a comparison of some of the most widely used computer software and which fields they are of particular use to. In particular I will examine the packages which most closely align with the design goals of Cadabra and the features which we believe to have been lacking in them and which motivate the continuing support and development of Cadabra. I will then present the theoretical background and implementation details of the canonicalisation algorithm `meld` which uses projection operators to generalise the concept of a canonicaliser into a more general tool which detects dependence of terms in an expression on each other rather than attempting to rewrite each term in a standard form. There will also be an opportunity to discuss various other routines which can be used alongside this in order to expand the number of expressions which can be simplified.

Following this I will present the packaging functionality which I have introduced into Cadabra and which contains a wide variety of smaller algorithms which provide convenience tools as well as more powerful routines which are used for the everyday manipulation of expressions. Part of the motivation for the development of this functionality was to make the sharing of libraries between members of the community easier and more transparent, as well as making it simpler for people to contribute algorithms for inclusion in Cadabra. Not all of the routines which I have implemented into the 'standard library' of Cadabra will be looked at, but some illustrative examples will be showcased.

This discussion of packages and the importance of the community to open source projects leads naturally into the second half of this thesis where I will examine more closely the rela-

[1] Available at `https://www.youtube.com/watch?v=Sg4U4r_AgJU`

tionship between software and academia. A lot of the work here is due to my participation with researchers writing papers on calculations in general relativity, in particular higher order perturbative solutions to gravitational wave equations, and therefore there while the main focus of this section is not to concentrate on the results from these papers I will use them to illustrate the points which I make as the discussions which we had during the course of writing these papers drove me towards focussing on particular parts of Cadabra's functionality.

The first point which I will showcase is the potential for computer algebra to make its way into the undergraduate classroom as both an aid in the teaching as well as an opportunity for letting students consider more complicated models than they would be able to look at using hand calculations. I will argue that the LaTeX style of input in Cadabra, as well as the use of Python which is widely taught at an undergraduate level in physics and maths departments, makes it an ideal candidate to be applied in field theory classes.

Secondly, I will look at the ways in which software is currently shared amongst the scientific community and ways in which we might encourage and facilitate contributing to the deposit of software which is generated. There is of course a larger theoretical problem that this entails of how to design an infrastructure whereby it is possible for people to find and obtain software which can solve a particular problem, but this begins to go beyond the purview of my research and I instead will be talking about many of the common problems which someone writing research software will come across repeatedly and present the solutions which may make sense in different scenarios. From this I hope to touch upon many questions which people starting out writing software will face and attempt to independently solve when in reality there are already a large number of suitable implementations and designs from which to build. This includes not only the format in which the user interacts with the program but also cross-platform compatibility, notebook-style interfaces compared to scripts, and the choice of other core features which make software usable. This will mainly focus on the areas in which I have provided additional functionality.

Finally, I bring all these points to bear by discussing how an open-source approach to writing research software provides an excellent environment for research software by allowing a transparent layer between developers and users, facilitating community contributions and importantly ensuring that there is free access to the software for all people. While there are objections from many researchers that putting their work into the public domain does not reward them for the nontrivial amount of work writing software entails, I believe this to be not a failing of the concept of an open-source environment but the current relationship which universities, grant committees and research groups themselves have with software development and that there is a fundamental disparity between the way we view research and the way we view software which needs to be narrowed.

I intend to make clear throughout this thesis the urgency with which I feel it is imperative our attitude to the availability and distribution of academic software must change, an opinion I draw not only from my own personal perspective of having worked in the development of the Cadabra program, but also from the interaction and feedback from its community and that of the mathematical sciences department. This will naturally come across more strongly in the second part where I discuss my work with other researchers in the community, but I hope that there will also be numerous occasions during the more technical discussions where it becomes clear that the different approaches software packages have taken can lead to a 'tunnel vision' where people are not open to using some software packages in the way they were designed to be used, preferring to mould it into the style they are comfortable.

Before ending this introduction and entering into a more technical discussion, I think it worthwhile to illustrate the relationship between researchers and the people who write software for research with a short anecdote from my day as a PhD student. After the end of the introductory lecture we were asked to move to a different room based on our specialisation: pure mathematics, applied mathematics, statistics and particle physics. After enquiring of the coordinator with which group I should associate, having explained my field and supervisor, I was met with a quizzical pause followed by “Ah yes, you are an edge case.”

I intend to demonstrate that the role of computing is integral enough to modern research that we can no longer afford to consider it as just an edge case.

Part I

Algorithms in Symbolic Algebra

Chapter 2

Computer Algebra Systems

In almost all fields of study, an understanding of the history and major advancements which have moulded the subject into the form it takes today is often informative and can help to deepen our knowledge and competency: subjects tend to become more complex and solutions dependent on more assumed knowledge the older they become, so looking back to the subject when it was still in its nascency can help to isolate the core ideas and a study of why certain conventions and ways of thinking were developed can help to clarify why things work the way they do instead of just how they work. The study of symbolic algebra is no different and therefore in order to introduce the subject I will start presenting the topic at the beginning and follow the development of the field to the modern day. After this we will concentrate more specifically on the use of computer algebra tools in the field of theoretical physics and introduce the Cadabra software package which forms the basis of the overall thesis. In order to contextualise better how Cadabra fits into the greater ecosystem of computer algebra tools which exist nowadays we will then follow through a comparison of how a general relativity calculation is performed in both Cadabra and another extremely popular open source program SageMath.

2.1 History of CAS

CAS tools have been an important part of physics and mathematics since their inception in the early 1960s with the introduction of programs such as Martinus J. G. Veltman's aforementioned *Schoonschip* [5], written primarily for applications in particle physics, and Carl Engleman's *MATHLAB*[19] which was a more general-purpose tool for scalar algebra, although it also included facilities for some linear algebra calculations such as solving systems of linear equations. Other early programs include *Macsyma* [20] which would later become *Maxima* and of which Carl Engleman was also a primary developer, *Scratchpad II* which would become *AXIOM*, and *REDUCE* which is still in active development today.

Apart from Schoonschip which was developed using an assembler, all of these programs were developed using various dialects of Lisp which had become the de facto language for writing such tools [21] just as FORTRAN was the primary language for engineering and numerical applications, although some CAS tools were also developed in FORTRAN such as the successful early system FORMAC [22]. The reasons why Lisp was regarded as such a useful tool for developing symbolic algebra tools was a combination of both the design of the language itself which lends itself readily to being extended into dialects which make the translation from mathematical problems to Lisp code simple [23], as well as the lack of other higher-level programming languages to choose from.

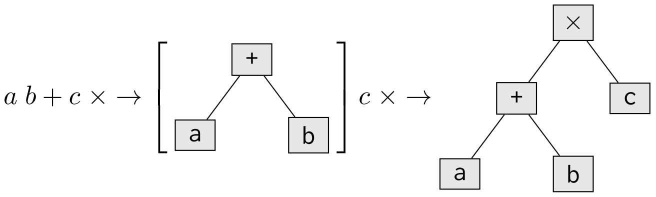
To understand why Lisp is such a natural language for symbolic algebra and the impact this had on future computer algebra systems, we will take a brief look at how mathematical expressions can be visualised as trees. The concept can find its roots in its modern application at least as far back as 1924 with the logician Jan Łukasiewicz. When writing mathematical expressions, we often use an infix notation, where operators are placed *in*between their operands, as in $a + b$. However, Łukasiewicz was wary of the several disadvantages to this notation, predominantly the ambiguity raised by an expression such as $a + b \times c$ where one must rely on a convention of operator precedence in order to distinguish between $(a + b) \times c$ and $a + (b \times c)$. He instead proposed a *prefix* style of notation with operators coming before their operands [24], allowing the above expressions to be represented as $\times + a\ b\ c$ and $+\ a \times b\ c$ respectively, if we assume that all operators are strictly binary. The prefix notation was dubbed *Polish notation*, purportedly as his name was too difficult for the English speaking community to pronounce.

In the 1950s, computer scientists recognised the application of using non-infix notations in computing [25, 26] due to the natural way in which it extended to the concept of a stack. Of particular interest was postfix notation (which, despite the notation being used for a very different purpose to Łukasiewicz's original aim of writing logic, is known as reverse Polish notation in his honour) which yielded a natural correspondence between an expression and the instruction sequence an assembler performs:

a	Push a onto the stack
b	Push b onto the stack
$+$	Pop two values off the stack and push their sum
c	Push c onto the stack
$\times$	Pop two values off the stack and push their product

Had the expression been written using an infix notation the one-to-one correspondence between the expression and stack instructions would be lost, with a more expensive parsing algorithm using a technique like recursive-descent to calculate the order of operations. Such was the convenience of postfix notations that several languages were developed to use it, most notably Adobe's Postscript [27] which would become the precursor to the modern PDF format.

Expressions in this form also make manifest the correspondence to a tree structure, where expressions are viewed as operators with parent nodes and their operands as children nodes. Again using the same expression $a\ b\ +\ c\ \times$ with binary operators, the tree is built by reading three tokens from the expression and forming a subtree out of these with the first two tokens as children of the third. This subtree is then substituted in to replace the three tokens and the process continued until all tokens have been parsed:



Writing expressions in this form allows many of the techniques in graph theory to be applied to the manipulation of expressions, importantly including efficient techniques for traversing the expression in a structured manner. If the operators are not strictly binary, then brackets are required in order to disambiguate which operator a token is a child of, however importantly this does not affect the correspondence to a tree structure.

Prefix notation was also adopted by some languages, most pertinently to our discussion Lisp is based on list structures (the name itself deriving from the phrase **Lis**t **P**rogramming). In Lisp, all lists are treated as functions where the first element is the operator and the rest of the list its operands. Therefore all Lisp programs can be viewed as just such a tree, and the expression $(\times\ (+\ a\ b)\ c)$ is a valid Lisp program (assuming the variables a, b and c are defined). It is essentially this which makes Lisp such a natural choice of language for developing symbolic algebra algorithms. To illustrate, we will quickly demonstrate how a substitute function might be implemented in GNU Common Lisp[1]:

```
(defun subs
    (ex in out)
    (if (atom ex)
        (if (equal ex in) out ex)
        (cons (subs (car ex) in out) (subs (cdr ex) in out))))
```

This defines a function `subs` which takes three parameters: `ex`, the expression to act on, `in`, the search expression, and `out` which contains the replacement expression. The function recursively travels through the tree trying to match subtrees, so in order to make the substitution $c \to kl$ in our expression from above we can call it as follows

```
>> (subs `(* (+ a b) c) `c `(* k l))
(* (+ a (* k l)) c)
```

Notice in particular how only the core functionality of the language is used, and while more complicated routines will require using less natural constructs the basic data types `list` and `symbol` encapsulate a lot of the low level structure.

Nowadays we find that most systems are developed using object-oriented languages which allow for a very natural connection between mathematical objects and programming objects, with classes representing expression trees and more complicated nodes carrying information in a structured manner using object-oriented techniques such as inheritance and polymorphism, but at this time object-oriented languages were very much in their infancy with Simula, which is often regarded as the first such language, only defining the concepts of class and type in 1967 [29].

While it is possible to produce efficient code in Lisp, one of the major advantages comes from the fact that it is an interpreted language, and for the large sorts of problems which computer algebra systems become most essential for the inefficiencies which result from running in an interpreter, coupled with the limited processing power available in the early days of computing, limited the capabilities of these systems. One of the major developments which spurred the next wave of CAS tools was the development of the C programming language in 1972 by Dennis Ritchie, possibly the most influential programming language ever released and which would find application in all areas of computing from systems programming and the early implementation of Unix for which it was originally designed to gaming, embedded system applications and building interpreters and compilers for other higher level programming languages. Although not as easy or natural to write CAS tools in as Lisp, C was far more powerful and flexible than using an assembler, allowing the creation of cross-platform native binaries and a much friendlier programming environment. This also saw a move away from using Lisp as an interface language and towards providing dedicated scripting languages with which to interact with tools.

One of the first computer algebra systems written in C was *SMP* which was developed in 1981 by Stephen Wolfram who would use the experience gained from developing this, along with the knowledge he gained from using the Macsyma system [30], to develop along with Theodore

[1] Adapted from the `subst` function written by John McCarthy in his original paper on Lisp [28]

Gray the program *Mathematica* in 1988 which has become one of the most popular CAS tools since then.

This of course only covers a small fraction of the systems which were developed during this period, and an emphasis has been placed on mentioning the more general-purpose tools as opposed to more domain-specific tools such as PARI [31], primarily a number theory tool, and CAYLEY [32] built for group theory applications. However, the programs discussed illustrate some of the key features which went on to influence the direction future tools would take. In particular, two aspects surfaced as being of primary value: an efficient back-end capable of handling large expressions and complex mathematical structures, and an interactive interpreted front-end which allows less technical users to interface with the programs. Initially these two goals were difficult to unify, with development in assembler requiring the non trivial task of developing an interactive interface, whilst using an existing interpreted language such as Lisp sacrificed the back-end efficiency.

2.2 Modern CAS Programs

The projects at Wolfram were among the first to combine both goals by developing a purpose built language, the *Wolfram Language*, which was used to interact with Mathematica and was designed specifically for algebraic programming. The Wolfram Language, while being in and of itself a fully-equipped programming language, was mainly intended not to be used to write algorithms but instead to define expressions and apply algorithms written in efficient C code to them. By introducing this language, Mathematica also paved the way for an environment on top of which many other packages and tools can, and have, been built[2].

Many of the CAS tools which are used nowadays had their roots in earlier projects, such as Maxima and AXIOM which were mentioned earlier, with some old systems still in use today such as FORM which was released in 1989 and is still in active development[3] and Maple [14] which is a proprietary tool released in 1982 and is still one of the most popular products. Other common tools which have been released more recently include SageMath [13], a general purpose open-source CAS system, and GiNaC [33] which is a C++ library used by several open-source tools including SageMath although it can also be used as a stand-alone tool for doing calculations directly in C++. A comprehensive review of the range of tools now available has been written by Malcolm MacCallum in his paper *Computer algebra in gravity research* [34]. This not only discusses the suitability of various programs and packages specifically aimed at problems in general relativity but also gives useful overviews of ten CAS tools and their application in computer algebra more generally.

The number of programming languages has grown enormously in the last few decades with the introduction of many general-purpose languages as well as domain-specific languages. Many allow for interfacing with lower- level languages such as C and C++ and therefore it is less common for modern CAS tools to provide their own language for interacting with them, preferring to provide an interface to another language. Much as Lisp was one of the most common languages for the designs of early CAS tools, Python has become a very popular language nowadays in not only symbolic computing but also other for numerical analysis and simulation in scientific disciplines. As well as tools which are specifically written for Python such as SymPy [35] and GraviPy [18], bindings for CAS libraries written in other languages (e.g. PyGiNaC [36] and

[2] Some relevant packages will be mentioned, but a sense of the popularity can be gained by looking at repositories such as `http://packagedata.net` which list over 200 packages at the time of writing.

[3] By looking at the status of the github repository `https://github.com/vermaseren/form` at the time of writing

cypari2 [37]) and Python interfaces of other CAS programs (e.g. Pythonica [38] and Sage [13]) are very common.

There are three main features of Python which drive its wide appeal in the scientific community: the ease and flexibility of the language, the object-oriented approach which is central to the language, and the wide variety of scientific libraries which is in part due to the relatively simple API for exporting C/C++ libraries to Python (especially with use of templated C++ libraries such as Boost.Python [39] and pybind11 [40]).

The focus of my research has been on the CAS program *Cadabra*, originally written by my supervisor Kasper Peeters in 2001 as a tool for doing calculations in tensor field theory. Originally written in C++ with its own interface language, it was rewritten in 2007 as a C++ library with Python bindings as well as an extension to the Python language to make some operations clearer.

2.3 Design of Cadabra

Cadabra is designed as first and foremost a tool for manipulation tensor objects with scalar manipulations and linear algebra algorithms not supported directly but by interfacing with a supported ‘backend’ CAS program, currently Mathematica and SymPy are supported. Importantly, tensors are treated as objects in their own right and are not necessarily tied to any particular representation as components; the dimension of a particular index set, coordinate values, metric connections, and other properties of a tensor which other systems require as prerequisite knowledge for defining a tensor is not required in Cadabra and can be dynamically associated with a tensor.

The main interface to Cadabra is a notebook-based GUI which allows writing scripts as a series of cells which can be edited, run and the output displayed. A single Python instance is maintained between all the cells, and so the cells not only split a script into logical sections but allow rerunning small individual sections of the code without needing to rerun the entire script. An interactive console based on the Python interpreter is also provided as well as the ability to use the Jupyter notebook interface. The input language is Python with a Cadabra preprocessor which extends the language with some extra syntax to make mathematical input clearer. These features will be presented below as their motivation is introduced.

The overall architecture revolves around three central elements: the `Ex` class which represents the expression tree structure, the `Algorithm` class which represents a transformation which can be applied to a `Ex` object and the `Property` class which is used to associate mathematical properties with tensors and other objects which make up the `Ex` tree. The connection between these types is the `Kernel` class, whose purpose is to parse input strings into `Ex` objects, store property information and allow algorithms to query this property information against expressions and sub-expressions.

2.3.1 The `Ex` class

The `Ex` class is based on a generic tree library `tree.hh` [4] with nodes of type `str_node` which is structured as follows

```
struct str_node {
    nset_t name;
    rset_t multiplier;
```

[4] `https://github.com/kpeeters/tree.hh`

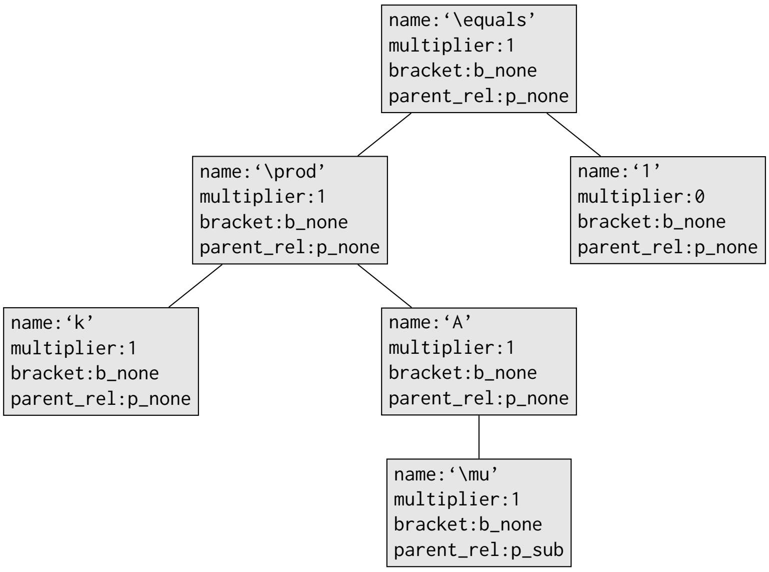


Figure 2.1: Visualisation of Cadabra’s tree representation of the expression $kA_\mu = 0$

```
    struct flag_t {
        bool keep_after_eval;
        bracket_t bracket;
        parent_rel_t parent_rel;
        bool line_per_node;
    } fl;
}
```

Here `nset_t` and `rset_t` are iterators into `std::sets` of strings and rationals respectively, which serve as pools for interning the names and multipliers of the nodes. The `bracket_t` and `parent_rel_t` types are enumerations, `bracket_t` determining the type of bracket the children of the node should be wrapped in and `parent_rel_t` information about how a node relates to its parent, e.g. as a covariant or contravariant index, as a specific component of the parent object or as an operand to an operator. The final two fields, `keep_after_eval` and `line_per_node` are bookkeeping nodes which are not used in the current implementation.

Fig. 2.1 shows the structure of the tree for the expression $kA_\mu = 0$. Here, the index μ on A_μ is a child node of A with the `parent_rel` value `p_sub` representing a covariant index. Also of note is that the value 0 is represented by a node with name `1`, representing any rational number, with the multiplier set to 0.

Traversing the expression tree is performed through the use of the four iterator types which `tree.hh` provides. The order in which these iterators visit the nodes of Fig. 2.1 are shown in table 2.1. Two more iterator types are provided, `fixed_depth_iterator` which iterates over all nodes at a given depth in the tree and `sibling_iterator` which iterates over all the direct children of a node.

`Ex` objects can be created from a LaTeX-style maths string which is parsed into an expression tree, although there are several differences in Cadabra’s ‘input form’ which disambiguate the semantic meaning of some expressions:

- The `**` symbol is used to denote exponentiation in line with the Python syntax, as `^` and `_` are reserved for writing indices.

- Multiplication using `*` is required to disambiguate when an object may be a function, for example `a(b + c)` results in a node `a` with a child node `(b + c)`, but `a * (b + c)` yields

Iterator class	Iteration technique	Example order of traversal
`pre_order_iterator`	Element before child	$=, \times, k, A, {}_{\mu}, 0$
`post_order_iterator`	Child before element	$k, {}_{\mu}, A, \times, 0, =$
`breadth_first_iterator`	Layer by layer	$=, \times, 0, k, A, {}_{\mu}$
`leaf_iterator`	Only lowest level	$k, {}_{\mu}, 0$

Table 2.1: The four main tree-traversal iteration types of the `Ex` class

the product of `a` and `(b + c)`.

- Multiple symbols in a product must be separated by whitespace or a `*` symbol; the input `abc` will produce a single node with name abc.
- Fractions can be entered in 'simple form': so `1/2` is valid as well as `\frac{1}{2}`

Expressions can also contain wildcard nodes, such expressions are called *patterns*. For example the expression `\partial{#}` is a pattern matching a `\partial` node with any number of children and can be used to match against expressions such as $\partial_i x^i$, ∂R and $\partial_{\mu\nu} T_\rho$. Named wildcards can also be used, such as `\exp{Q??}` which allows defining rules such as `$f(Q??, R??) -> c_0 + a_1 Q?? + b_1 R??$`.

Cadabra offers two extensions to Python's syntax in order to make defining expressions easier: the first is dollar-sign quotes as in LaTeX:

```
x = $k A_{\mu} = 1$
```

The second is the `:=` operator which allows for multi-line expressions which are terminated by any of the characters `:`, `.` and `;`

```
ch := \Gamma^{\mu}_{\nu\tau} =
    \frac{1}{2} g^{\mu\sigma} (\partial_{\tau}{g_{\nu\sigma}} +
    \partial_{\nu}{g_{\tau\sigma}} -
    \partial_{\sigma}{g_{\nu\tau}}).
```

The semicolon as a final character carries a special meaning in Cadabra, it takes the result of evaluating the line and then dispatches an appropriate routine to display the result; in the terminal this resolves to a call to `print` but in the notebook interface this will run LaTeX to generate the output.

2.3.2 The `Algorithm` class

Algorithms are implemented as classes both as a means of encapsulation and so that they can retain state if construction is non trivial or there are multiple nodes in the `Ex` tree where the algorithm can be applied.

Defining new algorithms requires inheriting from the base `Algorithm` class, and must minimally override the virtual methods `can_apply` and `apply`. When the class is interfaced to Python, it is exposed as a function which iterates over the expression tree using either a `pre_order_iterator` or `post_order_iterator` calling `can_apply` at each node until reaching the first node which returns true at which `apply` is called. The behaviour can be modified by passing flags to the algorithm when calling it, `deep` specifying whether the algorithm should descend through the tree to find nodes to act at, and `repeat` which causes the algorithm to repeatedly call `apply` at each node until the expression doesn't change.

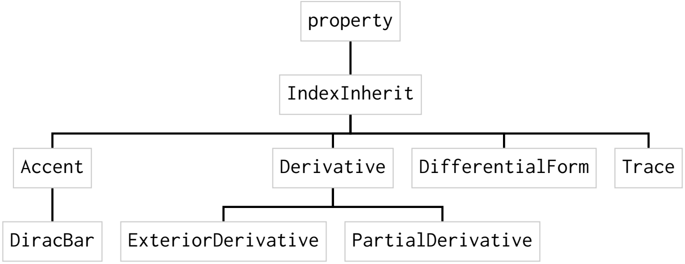


Figure 2.2: Section of the `property` inheritance tree centering around `IndexInherit`. Multiple inheritance has not been shown for clarity, but properties such as `Trace` also inherit from other properties (`Distributable`, `TableauInherit`, `NumericalFlat`).

2.3.3 The `Property` class

The `Property` class defines properties in a hierarchical structure, making use of multiple inheritance build up more complicated properties, for example the `AntiSymmetric` property inherits from both `TableauBase` and `Traceless`; more examples of how some properties relate to each other is given in Fig. 2.2.

Property objects can be stateful, for example the `Indices` property which stores information about the relationship between covariant and contravariant indices (`free` for $A_\mu = A^\mu$, `fixed` for $A_\mu B^\mu = A^\mu B_\mu$ and `independent` for no relation), coordinate values the indices can take and the name of the index set. A property object is then associated with an expression or pattern in the `Kernel` which stores a map between properties and patterns. Properties can be read from the kernel by giving it an expression which it will then attempt to match against the patterns it has stored, returning the property associated with that pattern if it of the same type as property requested.

The Python API exposes properties through a wrapper class which binds a property and pattern to a single object. Properties are inserted into the kernel by constructing a property with an expression

```
Indices($\mu, \nu, \rho$, $vector, position=free, values={t,x,y,z}$)
```

The first expression in the constructor is the pattern to assign the property to, and the optional second parameter is a list of arguments to construct the property with. Cadabra offers an alternative syntax for defining properties

```
{\mu, \nu, \rho}::Indices(vector, position=free, values={t,x,y,z}).
```

2.4 Comparison of Cadabra to other CAS

We must also look at Cadabra in the context of the other CAS programs which provide comparable functionality to see the motivation behind some of the design choices. Cadabra is written with tensors at the fore while many other computer algebra systems focus on scalar manipulations and linear algebra operations providing varying levels of support for tensors, from treating them as multi-dimensional arrays to having dedicated routines for manipulating them. Alongside Cadabra, other notable CAS programs which support tensor manipulations are Maple with the Physics package, Mathematica by making use of packages like MathTensor, Cartan, Ricci and xAct packages, Maxima, and Redberry which like Cadabra is also designed with tensor objects as a primary focus. Comparisons between the functionality of these tools has been examined in several papers; an overview of Maple, Mathematica, Maxima and Cadabra with a detailed comparison of the latter two is given in [41] and a comparison of Maple and Cadabra in [42]. Sections 6 and 7 of the previously referenced paper by MacCallum [34] are also an excellent reference.

We will first have a look at Cadabra compared to some other CAS tools; in order to keep the scope fairly narrow we will focus on three other systems which represent a fairly diverse selection of designs: Mathematica with the xAct package [43], Maple Physics, and RedBerry. In the next section to see how these differences surface when performing a calculation we will look specifically at how to approach some general relativity computations in both Cadabra and SageMath (which will also be referred to as simply Sage).

2.4.1 Accessibility

A primary factor to consider in choosing any program is its accessibility, which not only includes the platform support and requirements but also the price and availability of the software. Cadabra along with RedBerry are open source tools and thus freely available to download and use. Mathematica and Maple on the other hand are commercial tools and therefore a licence for them must be purchased which can be relatively expensive. This is however often offset by the fact that universities, companies and other institutions will often hold licences which afford their employees access the software without having to purchase it themselves. Additionally, Mathematica also offers a cloud-based interface[5] with a free option, although this does not provide as much power as the full program.

Both Mathematica and Maple are available on Windows, Linux and MacOS. RedBerry is implemented in Java and is therefore also supported on all three systems as well as other systems supported by Java. Cadabra was originally only available for Linux and MacOS, but recently support for Windows has been increasing although currently binaries are not available and it must be built from source.

2.4.2 Interface

Both Cadabra and Mathematica support a notebook interface which allows calculations to be split up into logical sections using cells which can be rerun and edited interactively. Maple makes use of a worksheet interface which also allows organisation of more complicated calculations into logical parts as well as the ability to include objects from other parts of Maple. Redberry is a Java library and is run through the Groovy language, and so calculations are written inside a `.groovy` script and then run.

All the systems allow the output to be converted to LaTeX, with RedBerry and Cadabra printing expressions in LaTeX by default, and Maple and Mathematica giving an export option and allowing the output format to be changed by the use of e.g. Maple's `Setup(mathematicalnotation=true)` command. Additionally Mathematica, Maple and Cadabra allow the whole notebook, or worksheet, to be exported to a `.tex` document allowing the easy presentation of a notebook as a stand-alone document.

2.4.3 Defining expressions

At the core of a CAS program is the ability to construct and manipulate expression. As noted earlier, Cadabra parses a LaTeX-style language into expressions and also produces output which is fed straight into a LaTeXcompiler. Of the tools under comparison only RedBerry also provides

[5] `https://www.wolframcloud.com`

a LaTeX-style input language, although the compromises made to avoid ambiguity are different to Cadabra's; compare the example from the RedBerry paper [44]

```
def t = 'F_mn*(A^ab + M_c*N^cab)'.t
```

which in Cadabra would be

```
t := F_{m n}*(A^{a b} + M_{c} * N^{c a b}).
```

One of the advantages of accepting LaTeX as input is that it allows expressions to be copied from other sources without much conversion needing to be done [42]. In both cases, the input is parsed from a string into an expression object. This is different from the other systems where expressions are built up by defining the constituent objects as variables and combining them using the operators of the language. For example in the xAct package the definition would first require defining the tensors on a manifold

```
DefManifold[Man, 3, {a,b,c,m,n}]
DefTensor[F[a,b], Man]
DefTensor[A[-a,-b], Man]
DefTensor[M[a], Man]
DefTensor[N[-a,-b,-c], Man]
```

where the dash is used to convey a contravariant index, and it can then be entered as

```
F[m,n] (A[-a,-b] + M[c] N[-c, -a, -b])
```

2.4.4 Manipulating expressions

Once constructed all systems allow algorithms to be applied to the resulting expression objects, either through function calls or using methods of the object. Cadabra is also unique in having mutable expressions, and thus a single copy of an expression can be used throughout a notebook with algorithms modifying it. The other packages all implement immutable expressions and the result of all algorithms is a new object. There are advantages to both systems; immutability allows some more efficient use of resources through interning expressions and also ensures that in notebooks cells can be run without modifying the expressions in other cells, on the other hand mutable expressions mean that lots of potentially expensive copies of expressions are avoided and the expression can be modified while iterating over it (although of course the power to do so does not always mean it is the right thing to do). Of course, a Cadabra expression can be copied if it needs to be used in several different calculations.

Another motivation behind having mutable expressions in Cadabra is to discourage copy-and-pasting parts of expressions in order to work on particular terms. The alternative method that Cadabra provides is to be able to zoom in on a expression by specifying a pattern for the terms that the user wishes to work on; for instance to select all terms in the expression $a + b + \partial_t c + \partial_t d$ which are derivatives the `zoom` algorithm can be used

```
\partial{#}::PartialDerivative.
t::Coordinate.
ex := a + b + \partial_{t}{c} + \partial_{t}{d}.
zoom(ex, $\partial_{t}{Q??}$);
```

which outputs $\cdots + \partial_t c + \partial_t d$. The benefit of this mechanism is that it is more flexible if the original expression is later modified and the notebook rerun.

2.4.5 Treatment of tensors

All the programs have different ways of implementing tensors, but whereas in Cadabra and RedBerry tenso Ricci differentiates between *tensor expressions* where indices are not explicitly defined and *component expressions* where the indices are inserted, although most of the algorithms are designed to work on both classes of expressions. Tensor expressions can be converted into component form by specifying some indices:

```
M [L[i], U[j]]
```

In Cadabra expressions are always treated as being in component form, although in some situations such as spinor mechanics where indices are conventionally suppressed it is possible to use the `ImplicitIndex` property to define indices for an object without having to have to explicitly in expressions.

Both RedBerry and Maple also use component notation, however they permit less freedom with index positions. By default Cadabra assumes that the position of an index does not contain any information, as differentiating between covariant and contravariant indices is a constraint which the user can enforce. RedBerry will however throw an index error if dummy indices are not balanced and Maple will automatically rearrange the indices to balance the dummy indices.

2.4.6 Conventions and properties

Cadabra attempts to enforce as few conventions as possible, allow the user to define notations which are natural for their problem and to tie them to as few conventions as possible, for instance by using symbols of their choice for common tensors and operations such as partial differentiation. Some examples of common property definitions in Cadabra would be

```
\partial{#}::PartialDerivative.
R_{a b c d}::RiemannTensor.
\delta_{a}^{b}::KroneckerDelta.
g_{\mu \nu}::Metric.
```

however in theory any symbols (as long as they have the correct index structure, number of arguments etc...) could be used in place of these. The other tools all contain some degree of inflexibility with regards to conventions; for instance RedBerry uses g_{ij} for the metric tensor and $d^{i}{}_{j}$ for the Kronecker delta. Maple also comes with some predefined tensors such as the gamma matrices γ_{μ}, the derivative operator ∂_{μ} and the metric tensor $g_{\mu\nu}$.

The other tools also generally require some information about the space to be defined before a tensor is used, as we have seen xAct requires the Manifold to be defined. The ability to define things as and when needed makes Cadabra more flexible when working in fields where it is undesirable or even impossible to explicitly define quantities such as the number of dimensions before a calculation.

2.5 Calculations in GR using Cadabra and SageMath

To show how Cadabra differs from other CAS tools we will compare how a calculation in general relativity is performed in both Cadabra and SageMath with the SageManifolds package. SageMath has been chosen as a comparison for a few reasons: it also uses a Python notebook interface which allows us to examine the differences in its approach and functionality rather than differences imposed by the language itself, it is also an open-source tool, and it has functionality whose approach is representative of the style of many of the alternative programs we migh consider. Similar calculations performed in Maple, Mathematica and other packages for comparison can be found in e.g. [45, 46, 47].

The calculation we perform is from the comparison paper [48] which begins with a comparison of the open-source tools SageMath, GraviPy and Maxima and then concentrates on the features of SageMath with a demonstrative calculation of calculating wave equations and geodesic orbits in a Schwarzschild spacetime which allows us to explore the abstract tensor capabilities, handling of scalar expressions as well as ability to perform numeric computations. We will follow through the calculation and comment on notable differences between the approach in Cadabra and SageMath. The output of the two notebooks is taken directly from the output cells. The mathematics and physics in this section can be found in most introductions and reference texts on General Relativity and Steven Weinberg's *Gravitation and Cosmology* [49] is recommended as an accessible book on the topic. A more detailed reference on the Schwarzschild spacetime specifically can be found in chapter 16 of [50].

The Schwarzschild metric describes the spacetime around a spherically symmetric body with no electric charge or angular momentum and where the cosmological constant is set to 0. It was first described independently by Karl Schwarzschild and Johannes Droste within a few months of the publication of Einstein's paper introducing General Relativity. Other than the case of a completely flat spacetime, it was the first analytic solution discovered for the Einstein equation. As many objects in the universe including stars, planets and black holes can be approximated to a high degree of accuracy as spherical objects it continues to prove to be one of the most useful and widely used metrics. Other than its use as an exact solution to the equations it is also frequently used as a starting point for perturbative solutions.

The form of the metric in terms of an invariant spacetime interval ds at a distance r from the center of a body of mass M, in natural units where the speed of light and gravitational constant are set to 1, is given in spherical coordinates (t, r, θ, ϕ) by

$$ds^2 = \left(1 - \frac{2M}{r}\right) dt^2 - \left(\frac{1}{1 - \frac{2M}{r}}\right) dr^2 - r^2 d\theta^2 - (r \sin\theta)^2 d\phi^2 \tag{2.1}$$

where the metric signature is timelike. At $r = 0$ there is a singularity at the time and radial components and so this point must be excluded. There is a second singularity in the radial component when $r = 2M$ which is referred to as the *event horizon*. However, unlike the singularity at $r = 0$, by choosing a suitable coordinate system it is possible to remove this singularity. For example, in the reference frame of a free-falling observer the metric appears like flat space flowing radially inwards [51]:

$$g = dt'^2 - \left(dr - \sqrt{\frac{2M}{r}} dt'\right)^2 - r^2 d\theta^2 - (r \sin\theta)^2 d\phi \tag{2.2}$$

where t' is the proper time of such an observer. Here there is no problem with letting $r = 2M$

showing that this is a consequence of the specific coordinate system which is used. We can however assign some physical meaning to the fact that this singularity occurs in spherical coordinates as this is nonetheless a valid reference frame an observer might view from, and as the radial component diverges at the event horizon we can ascertain that it is impossible for a far off observer to see or gain any information beyond the region where $2M \leq r$.

2.5.1 Defining the spacetime

We begin by setting up basic variables of the spacetime, which in both systems requires defining the set of coordinates and metric tensor. In Sage these stem from a `Manifold` object for which there is no equivalent in Cadabra, with the connection between the coordinates and metric instead being given by the indices we define.

Cadabra

```
coordinates := {t,r,\theta,\phi}.
Coordinate(coordinates)

indices := {\mu,\nu,\rho,\sigma,\lambda
     ,\kappa,\chi,\gamma}.
Indices(indices, $values=@(coordinates),
     position=fixed$)

g_{\mu\nu}::Metric.
g^{\mu\nu}::InverseMetric.
g::Determinant(g_{\mu\nu}).

\partial{#}::PartialDerivative.
i::ImaginaryI.
```

SageMath

```
reset()

Man = Manifold(4, 'Man', r'\mathcal{M}')
BL.<t,r,th,ph> = Man.chart(r't r:(0,+oo)
     th:(0,pi):\theta ph:(0,2*pi):\phi')
g = Man.lorentzian_metric('g')
```

Here we get to see syntaxes specific to both systems: the `:=` and `$...$` tokens in Cadabra for defining expressions and the angular brackets `<...>` in Sage for defining a list of names. One of the most important differences between the two is that defining a symbol in Sage returns an object which can then be used in constructing expressions in an index-free style using the Python operators and functions to combine objects; in Cadabra however as expressions are generated from LaTeXstyle strings with different objects identified by property definitions which are assigned to patterns, all the forms of the metric we will use in the calculations ($g_{\mu\nu}$, $g^{\mu\nu}$, g) must be declared. The next step is to define the components of the Schwarzschild metric

Cadabra

```
metric := {
    g_{t t} = (1-2 M/r),
    g_{r r} = -1/(1-2 M/r),
    g_{\theta\theta} = -r**2,
    g_{\phi\phi}=-r**2 \sin(\theta)**2}.
complete(metric, $g^{\mu\nu}$)
complete(metric, $g$);

sqrtdetg = substitute($sqrt(-g)$, metric
    )
map_sympy(sqrtdetg, "powdenest", "force=
    True")
substitute(sqrtdetg, $Abs(Q??) -> Q??$);
```

$$\left[g_{tt} = 1 - 2Mr^{-1},\; g_{rr} = -\left(1 - 2Mr^{-1}\right)^{-1},\; g_{\theta\theta} = -r^{2},\; g_{\phi\phi} = -r^{2}(\sin\theta)^{2},\; g^{tt} = \left(-2Mr^{-1} + 1\right)^{-1},\; g^{rr} = 2Mr^{-1} - 1,\; g^{\theta\theta} = -r^{-2},\; g^{\phi\phi} = -\left(r^{2}(\sin\theta)^{2}\right)^{-1},\; g = -r^{4}(\sin\theta)^{2}\right]$$

$$r^2 \sin\theta$$

SageMath

```
M = var('M')
g[0,0] = (1-(2*M)/r)
g[1,1] = -1/(1-(2*M)/r)
g[2,2] = -r^2
g[3,3] = -(r*sin(th))^2

ginv = g.inverse()
sqrtabsdetg = g.sqrt_abs_det().expr()

show(g.display())
show(ginv.display())
show(sqrtabsdetg)
```

$$g = \left(-\frac{2\,M}{r} + 1\right) \mathrm{d}t \otimes \mathrm{d}t + \left(\frac{1}{\frac{2\,M}{r} - 1}\right) \mathrm{d}r \otimes \mathrm{d}r - r^{2} \mathrm{d}\theta \otimes \mathrm{d}\theta - r^{2} \sin\left(\theta\right)^{2} \mathrm{d}\phi \otimes \mathrm{d}\phi$$

$$g^{-1} = \left(-\frac{r}{2\,M - r}\right) \frac{\partial}{\partial t} \otimes \frac{\partial}{\partial t} + \left(\frac{2\,M - r}{r}\right) \frac{\partial}{\partial r} \otimes \frac{\partial}{\partial r} - \frac{1}{r^{2}} \frac{\partial}{\partial \theta} \otimes \frac{\partial}{\partial \theta} - \frac{1}{r^{2} \sin(\theta)^{2}} \frac{\partial}{\partial \phi} \otimes \frac{\partial}{\partial \phi}$$

$$r^{2} \sin\left(\theta\right)$$

In Sage the metric tensor is defined and accessed as a matrix with the values inserted using integer indices. In Cadabra's approach components of the metric can be stored in an expression consisting of comma-separated definitions which act as substitution rules. Generating the inverse and determinant is done using `complete` which uses the components already defined to 'complete' the list of substitution rules. This relies on the `InverseMetric` and `Determinant` properties for Cadabra to recognise the relationship between the components which have already been defined and the object it is asked to calculate. Calculating the value of $\sqrt{(-g)}$ in Cadabra requires some more steps than in Sage, as Cadabra does not gain enough information about the coordinates to simplify square roots in general, so SymPy's `powdenest` is required. We strip the absolute functions with a substitution using the named wildcard `Q??` which allows the substitution rule to work with arbitrary arguments.

In order to calculate a covariant derivative on the metric we need to determine the metric connection in our spacetime which allows us to connect different points on the spacetime manifold. This connection is given by the Christoffel symbols

$$\Gamma^{\rho}{}_{\mu\nu} = \frac{1}{2} g^{\rho\lambda} (\partial_{\nu} g_{\lambda\mu} + \partial_{\mu} g_{\lambda\nu} - \partial_{\lambda} g_{\mu\nu}) \tag{2.3}$$

which are calculated in both programs as follows.

Cadabra	SageMath

`ch := \Gamma^{\rho}_{\mu \nu} = 1/2 g^{\`
`    rho \lambda} (`
`    \partial_{\nu}{g_{\lambda \mu}}`
`    + \partial_{\mu}{g_{\lambda \nu}}`
`    - \partial_{\lambda}{g_{\mu \nu}}).`
`evaluate(ch, metric, rhsonly=True);`
 	
`import cdb.core.component as comp`
`comp.get_component(ch, $\phi, \theta, \`
`    phi$);`
	`ch = g.connection()`
	`show(ch.display())`
	`show("Component [3,2,3] = ", ch[3,2,3])`

$\Gamma^{\rho}{}_{\mu\nu} = \Box_{\mu\nu}{}^{\rho} \begin{cases} \Box_{\phi r}{}^{\phi} = r^{-1} \\ \Box_{\phi\theta}{}^{\phi} = (\tan\theta)^{-1} \\ \Box_{\theta r}{}^{\theta} = r^{-1} \\ \Box_{rr}{}^{r} = M(r\,(2M-r))^{-1} \\ \Box_{tr}{}^{t} = M(r\,(-2M+r))^{-1} \\ \Box_{r\phi}{}^{\phi} = r^{-1} \\ \Box_{\theta\phi}{}^{\phi} = (\tan\theta)^{-1} \\ \Box_{r\theta}{}^{\theta} = r^{-1} \\ \Box_{rt}{}^{t} = M(r\,(-2M+r))^{-1} \\ \Box_{\phi\phi}{}^{r} = (2M-r)\,(\sin\theta)^{2} \\ \Box_{\phi\phi}{}^{\theta} = -\frac{1}{2}\sin(2\theta) \\ \Box_{\theta\theta}{}^{r} = 2M-r \\ \Box_{tt}{}^{r} = M\,(-2M+r)\,r^{-3} \end{cases}$ $\Gamma^{\phi}{}_{\theta\phi} = (\tan\theta)^{-1}$	$\begin{aligned} \Gamma^{t}{}_{t\,r} &= -\frac{M}{2\,Mr-r^{2}} \\ \Gamma^{t}{}_{r\,t} &= -\frac{M}{2\,Mr-r^{2}} \\ \Gamma^{r}{}_{t\,t} &= -\frac{2\,M^{2}-Mr}{r^{3}} \\ \Gamma^{r}{}_{r\,r} &= \frac{M}{2\,Mr-r^{2}} \\ \Gamma^{r}{}_{\theta\,\theta} &= 2\,M-r \\ \Gamma^{r}{}_{\phi\,\phi} &= (2\,M-r)\sin(\theta)^{2} \\ \Gamma^{\theta}{}_{r\,\theta} &= \frac{1}{r} \\ \Gamma^{\theta}{}_{\theta\,r} &= \frac{1}{r} \\ \Gamma^{\theta}{}_{\phi\,\phi} &= -\cos(\theta)\sin(\theta) \\ \Gamma^{\phi}{}_{r\,\phi} &= \frac{1}{r} \\ \Gamma^{\phi}{}_{\theta\,\phi} &= \frac{\cos(\theta)}{\sin(\theta)} \\ \Gamma^{\phi}{}_{\phi\,r} &= \frac{1}{r} \\ \Gamma^{\phi}{}_{\phi\,\theta} &= \frac{\cos(\theta)}{\sin(\theta)} \end{aligned}$ `Component [3,2,3] =` $\frac{\cos(\theta)}{\sin(\theta)}$

As the metric is implemented as a Python object in Sage, generating the components is done by calling the `connection` method, which by default uses the symbol Γ. In Cadabra the definition of the connection in terms of the metric tensor must first be entered and then `evaluate` can be called which calculates the explicit components of each free index using the substitution rules calculated for the metric. This also returns an Ex object but with a components node which allows a specific component to be selected using the `get_component` function from the standard library which resolves the components node and all remaining free indices. In Sage, components are accessed like the metric using array indices.

The central equation in General Relativity is the Einstein equation which determines the relationship between the curvature of a spacetime with metric tensor $g_{\mu\nu}$ and the density of energy it contains in terms of the stress-energy tensor $T_{\mu\nu}$:

$$G_{\mu\nu} + \Lambda g_{\mu\nu} = \kappa T_{\mu\nu} \tag{2.4}$$

where is $G_{\mu\nu}$ the Einstein tensor, Λ is the cosmological constant and κ is the Einstein gravitational constant with a value of approximately $2.08 \times 10^{-43}\mathrm{N}^{-1}$. The Einstein tensor can be expressed in terms of the Ricci tensor $R_{\mu\nu}$ and Ricci scalar R

$$G_{\mu\nu} = R_{\mu\nu} - \frac{1}{2} R g_{\mu\nu} \tag{2.5}$$

and so if the $R_{\mu\nu} = 0$ and $\Lambda = 0$ then it can be seen that the stress-energy tensor is also zero. Such a solution is known as a *vacuum solution*. To show that the Schwarzschild spacetime is a vacuum solution we calculate the Ricci tensor here:

Cadabra

```
riemann := R^{\rho}_{\sigma \mu \nu} =
    \partial_{\mu}(\Gamma^{\rho}_{\nu \
        sigma}) -
    \partial_{\nu}(\Gamma^{\rho}_{\mu \
        sigma}) +
    \Gamma^{\rho}_{\mu \lambda} \Gamma
        ^{\lambda}_{\nu \sigma} -
    \Gamma^{\rho}_{\nu \lambda} \Gamma
        ^{\lambda}_{\mu \sigma}.
ricci := R_{\mu \nu} = R^{\rho}_{\mu \
    rho \nu}.
substitute(ricci, riemann)
substitute(ricci, ch)
evaluate(ricci, metric, rhsonly=True);
comp.get_component(ricci, $r, r$);
```

$$R_{\mu\nu} = 0$$

$$R_{rr} = 0$$

SageMath

```
ricci = g.ricci()
show(ricci.display())
show("Component [1,1] = ", ricci[1 ,1])
```

$\mathrm{Ric}\left(g\right) = 0$

```
Component [1,1] =0
```

Calculating the Ricci tensor in Cadabra is performed in two steps, first constructing the definition of the Ricci tensor from the definition of the Riemann tensor and then substituting in the computed values of the connection and metric tensors. Both systems show the zero without having to expand the object out into components, Cadabra by leaving the free indices on the tensor and Sage by showing the object in an index-free notation.

2.5.2 Computing the Klein-Gordon equation

In order to study the dynamics of a particle bound in a Schwarzschild spacetime we need to use a relativistic wave equation, i.e. a covariant analogue of the Schrödinger equation. For the analysis here, we use the Klein-Gordon equation [52]

$$\frac{1}{\sqrt{-g}}\partial_\mu(\sqrt{-g}g^{\mu\nu}\partial_\nu\Phi) = \frac{\partial V(\Phi)}{\partial\Phi} \tag{2.6}$$

To study the spherically-symmetric Schwarzschild spacetime we assume the Ansatz solution given in [53]

$$\Phi = e^{-i\omega t}e^{ik\phi}R(r)S(\theta) \tag{2.7}$$

We start by computing the expression $\partial_\mu(\sqrt{-g}g^{\mu\nu}\partial_\nu\Phi)$

Cadabra

```
R::Depends(r).
\Theta::Depends(\theta).

ansatz := \Phi = \exp(-i \omega t) * \
    exp(i k \phi) * R * \Theta.
kg := \partial_{\mu}{g^{\mu \nu} @(
    sqrtdetg) \partial_{\nu}{\Phi}}.
substitute(kg, ansatz)
evaluate(kg, metric)
substitute(kg, $i**2 -> -1$);
```

$$\Big(k^{2} \left(2M-r\right) R\Theta+\omega^{2}r^{3}R\Theta(\sin\theta)^{2}+(2M -r) \left(r \left(2M-r\right) \Theta \sin\theta\partial_{r r}R-r\Theta \sin\theta\partial_{r}R + \left(2M - r\right) \Theta \sin\theta\partial_{r}R - R\sin\theta\partial_{\theta\theta}\Theta - R\cos\theta\partial_{\theta}\Theta\right) \sin\theta\Big) \exp\left(i \left(k\phi - \omega t\right)\right) \left(\left(2M - r\right) \sin\theta\right)^{-1}$$

SageMath

```
var('omega, k, KG')
Phi = function('Phi')(*BL)
R = function('R')(r)
Theta = function('Theta')(th)
Phi = exp(-I*omega*t) * exp(I*k*ph) * R
    * Theta

kg = 0
for mu in range(len(BL[:])):
    for nu in range(len(BL[:])):
        kg += diff(
            (ginv[mu,nu].expr() *
                sqrtabsdetg * diff(Phi,
                BL[nu])),
            BL[mu])

show(kg)
```

$$\frac{\omega^{2} r^{3} R(r) \Theta(\theta) e^{\left(i \, k \phi - i \, \omega t\right)} \sin(\theta)}{2 \, M - r} + (2 \, M - r) r \Theta\left(\theta\right) e^{\left(i \, k \phi - i \, \omega t\right)} \sin\left(\theta\right) \frac{\partial^{2}}{(\partial r)^{2}} R\left(r\right) + (2 \, M - r) \Theta\left(\theta\right) e^{\left(i \, k \phi - i \, \omega t\right)} \sin\left(\theta\right) \frac{\partial}{\partial r} R\left(r\right) - r \Theta\left(\theta\right) e^{\left(i \, k \phi - i \, \omega t\right)} \sin\left(\theta\right) \frac{\partial}{\partial r} R\left(r\right) + \frac{k^{2} R(r) \Theta(\theta) e^{\left(i \, k \phi - i \, \omega t\right)}}{\sin(\theta)} - R\left(r\right) \cos\left(\theta\right) e^{\left(i \, k \phi - i \, \omega t\right)} \frac{\partial}{\partial \theta} \Theta\left(\theta\right) - R\left(r\right) e^{\left(i \, k \phi - i \, \omega t\right)} \sin\left(\theta\right) \frac{\partial^{2}}{(\partial \theta)^{2}} \Theta\left(\theta\right)$$

In Sage we loop through the coordinates calculating each term and adding it to the resulting expression. In Cadabra we can write the equation using free indices, and then after substituting in the Ansatz solution we use `evaluate` which automatically expands the dummy indices and substitutes in the previously calculated metric components. The value of $\sqrt{-g}$ is 'pulled in' to the expression using the `@(...)` syntax. It is of course also possible in Cadabra to build up the expression by looping through the `coordinates` expression defined earlier and pulling in each:

```
for mu in coordinates.top().children():
    for nu in coordinates.top().children():
        term := \partial_{@(mu)}{
            g^{@(mu) @(nu)} @(sqrtdetg) \partial_{@(nu)}{\Phi}.
        kg += term
```

This also shows that Cadabra does support the Python operators `+` and `*` between `Ex` objects

We now want to split the equation into terms relying on the radial coordinate r and the angular coordinate θ. First we divide through by the common factor $\Phi\sin\theta$, and then loop over the terms and calculate their derivative with respect to r

Cadabra

```
kg *= $1/(\Phi sin(\theta))$
substitute(kg, ansatz)
substitute(kg, $2M - r -> Mp$)
distribute(kg)
substitute(kg, $Mp -> 2M - r$)

laux::LaTeXForm("\lambda_{\mbox{\
    scriptsize{aux}}}").
kg_rad := laux.
kg_ang := -laux.
for term in kg.top().children():
    der = simplify($\partial_{r}{@(term)
        }$)
    if der == 0:
        kg_ang += term
    else:
        kg_rad += term
```

SageMath

```
kg = expand(kg / (Phi * sin(th)))

var('lambda_aux')
kg_rad = lambda_aux
kg_ang = -lambda_aux
for term in kg.operands():
    if diff(term, r) == 0:
        kg_ang += term
    else:
        kg_rad += term
```

In order for the `kg.top().children()` iterator to loop over the terms in a sum, we first `distribute` the Cadabra expression, sandwiching the call between `substitutes` to ensure that the bracket $(2M - r)$ isn't expanded as a side effect. As we built the equation in Sage up term by term we do not need to distribute it first.

We have also introduced the auxiliary variable λ_{aux}. Sage automatically creates a LaTeX name for the variable although a different name can be assigned using the `latex_name` option to `var`. In Cadabra the name is used directly in a LaTeX expression and by default all subscripts of an object are treated as covariant indices; in order to avoid this we have chosen to create a proxy name `laux` and assigned it a `LaTeXForm` property so that it displays properly, but it would also be possible to use the `Symbol` property

```
aux::Symbol.
```

to override the default behaviour of treating this name as an index. Before printing the two expressions we apply some of the tidying routines to isolate the functions $R(R)$, $S(\theta)$ and their derivatives

Cadabra

```
display("Radial component:")
kg_rad *= $R$
simplify(kg_rad)
substitute(kg_rad, $2M - r -> Mp$)
distribute(kg_rad)
factor_out(kg_rad, $\partial_{r}{R}$)
factor_out(kg_rad, $\partial_{r r}{R}$)
factor_out(kg_rad, $R$)
sort_product(kg_rad)
collect_factors(kg_rad)
substitute(kg_rad, $Mp -> 2M - r$);

display("Angular component:")
kg_ang *= $\Theta$
simplify(kg_ang)
factor_out(kg_ang, $\partial_{\theta}{\
    Theta}$)
factor_out(kg_ang, $\partial_{\theta \
    theta}{\Theta}$)
factor_out(kg_ang, $\Theta$);
```

SageMath

```
kg_rad = expand(kg_rad * R)  \
    .simplify_full()         \
    .collect(R)              \
    .collect(diff (R,r))     \
    .collect(diff(R,r,r))

kg_ang = expand(kg_ang * Theta)  \
    .simplify_full()             \
    .collect(Theta)              \
    .collect(diff(Theta, th))    \
    .collect(diff(Theta,th,th))

show("Radial component")
show(kg_rad)
show("Angular component")
show(kg_ang)
```

```
Radial component:
```

$$\partial_{r}R\left(-2r+2M\right)+\left(2M-r\right)\partial_{rr}Rr + R\left(\lambda_{\text{aux}}+\left(2M-r\right)^{-1}\omega^{2}r^{3}\right)$$

```
Angular component:
```

$$-\partial_{\theta}\Theta(\tan\theta)^{-1}-\partial_{\theta\theta}\Theta+\Theta\left(k^{2}(\sin\theta)^{-2}-\lambda_{\text{aux}}\right)$$

```
Radial component
```

$$\left(\frac{\omega^{2}r^{3}}{2\,M-r}+\frac{2\,M\lambda_{aux}}{2\,M-r}-\frac{\lambda_{aux}r}{2\,M-r}\right)R\left(r\right) + 2\left(\frac{2\,M^{2}}{2\,M-r}-\frac{3\,Mr}{2\,M-r}+\frac{r^{2}}{2\,M-r}\right)\frac{\partial}{\partial r}R\left(r\right) + \left(\frac{4\,M^{2}r}{2\,M-r}-\frac{4\,Mr^{2}}{2\,M-r}+\frac{r^{3}}{2\,M-r}\right)\frac{\partial^{2}}{(\partial r)^{2}}R\left(r\right)$$

```
Angular component
```

$$-\left(\lambda_{aux}-\frac{k^{2}}{\sin(\theta)^{2}}\right)\Theta\left(\theta\right) - \frac{\cos(\theta)\frac{\partial}{\partial\theta}\Theta(\theta)}{\sin(\theta)} - \frac{\partial^{2}}{(\partial\theta)^{2}}\Theta\left(\theta\right)$$

2.5.3 Visualising the radial component

We now concentrate on the radial component. The solution to this equation can be given in terms of the confluent Heun functions which are not implemented by either program, and so we use the numeric tools available to examine the solution. As it is second order in derivatives of r, we introduce the auxiliary function $R_{\text{aux}}(r) = \partial_r(R(r))$ to create a system of two first order equations and then use a linear solver to isolate the derivatives on one side.

Cadabra

```
from cdb.sympy.solvers import linsolve
Raux::Depends(r).
Raux::LaTeXForm("R_{\mbox{\scriptsize{
    aux}}}").

rad1 := \partial_{r}{R} - Raux.
rad2 = substitute(kg_rad.top().ex(), $\
    partial_{r}{R} -> Raux, \partial_{r
    r}{R} -> \partial_{r}{Raux}$)

dR = linsolve(rad1, $\partial_{r}{R}$)
    [0][0];
dRaux = linsolve(rad2, $\partial_{r}{
    Raux}$)[0][0];
```

$$\partial_r R \to R_{\text{aux}}$$

$$\partial_r R_{\text{aux}} \to \left(-4M^2 R_{\text{aux}} - 2M\lambda_{\text{aux}}R + 6MrR_{\text{aux}} + \lambda_{\text{aux}}rR - \omega^2 r^3 R - 2r^2 R_{\text{aux}}\right)\left(r\left(4M^2 - 4Mr + r^2\right)\right)^{-1}$$

SageMath

```
R_aux = function('R_aux')(r)

rad1 = diff(R, r) - R_aux
rad2 = kg_rad                    \
    .subs(diff(R, r) == R_aux) \
    .subs(diff(R, r, r) == diff(R_aux, r
        ))

dR = (solve(rad1 == 0, diff(R, r))[0].
    right()
dRaux = (solve(rad2 == 0, diff(R_aux, r)
    ))[0].right()

show("dR/dr = ", dR)
show("dRaux/dr = ", dRaux)
```

dR/dr =$R_{\text{aux}}\left(r\right)$

dRaux/dr = $-\frac{\left(\omega^2 r^3+2\,M\lambda_{aux}-\lambda_{aux}r\right)R(r)+2\left(2\,M^2-3\,Mr+r^2\right)R_{\text{aux}}(r)}{4\,M^2r-4\,Mr^2+r^3}$

Sage can make use of Maxima's rk differential equation solver which uses a fourth-order Runge-Kutta method; in Cadabra we make use of the integrate_ode function which is a wrapper for the SciPy function scipy.integrate.odeint.

In both cases we must pass both R and R_{aux} as variables instead of functions of r which we do by making the substitution R -> Rc and R_aux -> Rc_aux.

Cadabra

```
from matplotlib import pyplot as plt
from cdb.numeric.integrate import
    integrate_ode

ex1 = substitute(dR[1], $R -> Rc, Raux
    -> Rcaux$)
ex2 = substitute(dRaux[1], $R -> Rc,
    Raux -> Rcaux$)

rs, Rs, Rauxs = integrate_ode(
    (ex1, ex2),
    $r, Rc, Rcaux$,
    (0.3, 1, 0.5),
    100, 0.01,
    $M->0.1, \omega->0.2, k->2, laux->2$
        )
```

SageMath

```
from sage.calculus.desolvers import
    desolve_system_rk4

Rc, Rc_aux, r = var('Rc, Rc_aux, r')
ex1 = dR.subs(R == Rc, R_aux == Rc_aux)
ex2 = dRaux.subs(R == Rc, R_aux ==
    Rc_aux)

ex2 = ex2.subs(M=0.1, omega=0.2, k=2.0,
    lambda_aux=2.0)
radsol = desolve_system_rk4(
    [ex1, ex2],
    [Rc, Rc_aux],
    ics=[0.3,1,0.5],
    ivar=r, end_points=100, step=0.01)
```

To check the solution we also implement an asymptotic approximation of the solution which is provided in [54]:

$$R_l = \frac{C_l}{r}\sin\left[\omega r + 2M\omega\ln(r) - \frac{l\pi}{2} + \arg(\Gamma(l+1-2iM\omega))\right] \tag{2.8}$$

with the parameter C_l set to match the amplitude of the numeric solution

Cadabra

```
from scipy.special import gamma
import numpy as np

def Rasym(r, Cl, M, omega, L):
    return (Cl/r) * np.sin(omega * r + 2
         * M * omega * np.log(r) - \
            0.5 * L * np.pi + np.angle(
                gamma(L+1-2j*M*omega)))

rs_asymp = np.linspace(20, 100, 200)
Rs_asymp = [Rasym(r, 250, 0.1, 0.2, 1.0)
     for r in rs_asymp]
```

SageMath

```
var('Cl ,L')
Rasym = Cl*(1/r)*sin(
    omega*r + 2*M*omega*log(r) - (L*pi
        /2)
    + arg(gamma(L + 1 - 2*I*M*omega)))
Rasymnum = Rasym.subs(Cl=250, M=0.1,
    omega=0.2, L=1.0)
```

In Cadabra we have made use of NumPy to implement this as a pure Python function, although we could also have written it as a Ex object and used the `cdb.numeric.evaluate` library to evaluate it numerically. We now plot the solution to the differential equations against this asymptotic formula. Both systems use `matplotlib` for plotting, but while SageMath provides some convenience functions allowing us to write `Rasym` as a Sage expression and pass it directly to the `plot` function when plotting in Cadabra we use the `pyplot` interface. Notice that the Cadabra semicolon works analogously to the Sage `show` command, displaying not only expressions but also the plot inside the notebook.

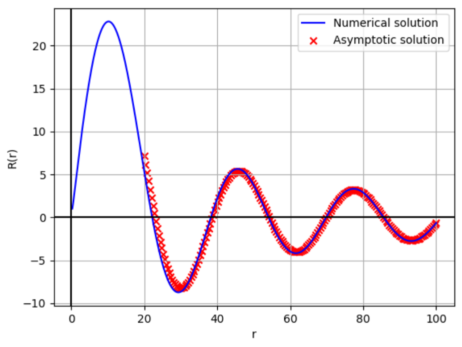
Cadabra

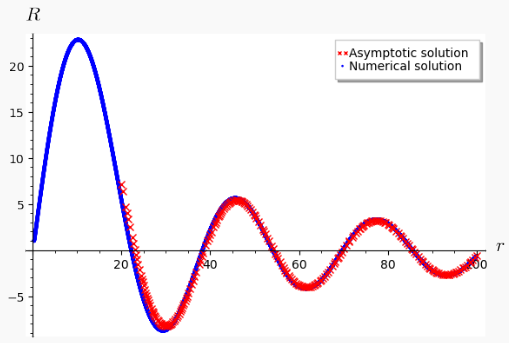
```
fig, ax = plt.subplots()
ax.plot(rs, Rs, color='b', label='
    Numerical solution')
ax.scatter(rs_asymp, Rs_asymp, color='r'
    , marker='x', label='Asymptotic
    solution')

ax.set_xlabel("r")
ax.set_ylabel("R(r)")
ax.legend()
ax.grid(True, which='both')
ax.axhline(y=0, color='black')
ax.axvline(x=0, color='black')
fig;
```

SageMath

```
points = [[i,j] for i,j,k in radsol]
radial_solution = list_plot(
    points, axes_labels=['$r$', '$R$'],
    legend_label='Numerical solution')
asympplot = plot(
    Rasymnum, (r, 20, 100), linestyle=''
        ,
    marker='x', color='red',
    legend_label='Asymptotic solution ')
show(asympplot + radial_solution)
```





2.5.4 Visualising geodesics

A geodesic is a generalisation of the concept of a straight line to a curved surface by describing the path of the shortest line connecting two points. Free particles follow geodesics, and so to examine their dynamics we need to calculate geodesics in the Schwarzschild spacetime. One way to do this is by solving the Hamilton-Jacobi equation

$$\frac{\partial S}{\partial \eta} - \frac{1}{2} g^{\mu\nu} \frac{\partial S}{\partial x^\mu} \frac{\partial S}{\partial x^\nu} = 0 \tag{2.9}$$

for some affine parameter η. In order to solve this we will use the decomposition [50]

$$S = \frac{m^2}{2}\eta - Et + L\phi + F(r) + G(\theta) \tag{2.10}$$

In Cadabra we proceed as for the Klein-Gordon equation, entering the equation with dummy indices after declaring the coordinate η and then evaluating it over the metric. Similarly in Sage the expression is built up term by term by looping over the coordinates:

Cadabra

```
{\eta}::Coordinate.
F::Depends(r).
G::Depends(\theta).

ansatz := S = (\eta m**2) / 2 - w * t +
    L \phi + F + G.
hj := \partial_{\eta}{S} - 1/2g^{\mu \nu
    } \partial_{\mu}{S} \partial_{\nu}{S
    };
substitute(hj, ansatz)
evaluate(hj, metric);
```

$$\partial_\eta S - \frac{1}{2} g^{\mu\nu} \partial_\mu S \partial_\nu S$$

$$\begin{aligned}&\frac{1}{2}L^2\left(r^2(\sin\theta)^2\right)^{-1} + \frac{1}{2}m^2\\ &\quad + w^2\left(4Mr^{-1} - 2\right)^{-1}\\ &\quad - \left(Mr^{-1} - \frac{1}{2}\right)(\partial_r F)^2 + \frac{1}{2}(\partial_\theta G)^2 r^{-2}\end{aligned}$$

SageMath

```
var('eta, m, w, L, S, HJfull')
F = function('F')(r)
G = function('G')(th)

# Define the principal function Ansatz
S = ((eta * m ^2) / 2) -w*t + L*ph + F +
     G

# Calculate the Hamilton - Jacobi
    equation
hj = 0
for i in range(len(BL[:])):
    for j in range(len(BL[:])):
         hj += ginv[i, j].expr() * diff(
              S, BL[i]) * diff (S, BL[j])
hj = diff(S, eta) - (1/2)*hj
show (hj)
```

$$\frac{1}{2}\,m^2 + \frac{rw^2}{2\,(2\,M-r)} - \frac{(2\,M-r)\frac{\partial}{\partial r}F(r)^2}{2\,r} + \frac{\frac{\partial}{\partial \theta}G(\theta)^2}{2\,r^2} + \frac{L^2}{2\,r^2\sin(\theta)^2}$$

We also calculate the right hand side of the orbital equations

$$\frac{dx^\mu}{d\eta} = -g^{\mu\nu}\frac{\partial S}{\partial x^\nu} \tag{2.11}$$

in a similar fashion. In Sage the components are inserted as elements of a vector, while in Cadabra a set of component rules for the free index μ is generated.

Cadabra

```
orbital := g^{\mu \nu} \partial_{\nu}{S}
.
substitute(orbital, ansatz)
evaluate(orbital, metric);
```

$$\square^{\mu}\begin{cases}\square^{t} = rw(2M-r)^{-1}\\ \square^{r} = (2M-r)\,\partial_{r}Fr^{-1}\\ \square^{\theta} = -\partial_{\theta}Gr^{-2}\\ \square^{\phi} = -L\Big(r^{2}(\sin\theta)^{2}\Big)^{-1}\end{cases}$$

SageMath

```
orbital = zero_vector(SR, len(BL[:]))

for mu in range(len(BL[:])):
    for nu in range(len(BL[:])):
        orbital[mu] = orbital[mu] - (
            ginv[mu, nu].expr()) * diff(
            S, BL[nu])

show(orbital)
```

$$\left(-\frac{rw}{2\,M-r},\ -\frac{(2\,M-r)\frac{\partial}{\partial r}F(r)}{r},\ \frac{\frac{\partial}{\partial \theta}G(\theta)}{r^{2}},\ \frac{L}{r^{2}\sin(\theta)^{2}}\right)$$

To look at solutions on the equator we make the substitution $\theta \to \frac{\pi}{2}$, and thus also $\partial_{\theta}G \to 0$, to both the Hamilton-Jacobi expression as well as the orbital equations which is done similarly in both programs

Cadabra

```
equatorial := \partial_{\theta}{G} -> 0,
      \theta -> \pi/2.
substitute(hj, equatorial)
simplify(hj);

substitute(orbital, equatorial);
```

$$\frac{1}{2}\Big(L^{2}\,(2M-r) + m^{2}r^{2}\,(2M-r) + r^{3}w^{2} - r(2M-r)^{2}(\partial_{r}F)^{2}\Big)\,\big(r^{2}\,(2M-r)\big)^{-1}$$

$$\square^{\mu}\begin{cases}\square^{t} = rw(2M-r)^{-1}\\ \square^{r} = (2M-r)\,\partial_{r}Fr^{-1}\\ \square^{\frac{1}{2}\pi} = 0\\ \square^{\phi} = -L\left(r^{2}\sin\left(\frac{1}{2}\pi\right)^{2}\right)^{-1}\end{cases}$$

SageMath

```
hj = hj.subs(diff(G)==0).subs(th=pi/2)
show(hj)

orbital = orbital.subs(diff(G)==0).subs(
    th=pi/2)
show(orbital)
```

$$\frac{1}{2}\,m^{2} + \frac{rw^{2}}{2\,(2\,M-r)} - \frac{(2\,M-r)\frac{\partial}{\partial r}F(r)^{2}}{2\,r} + \frac{L^{2}}{2\,r^{2}}$$

$$\left(-\frac{rw}{2\,M-r},\ -\frac{(2\,M-r)\frac{\partial}{\partial r}F(r)}{r},\ 0,\ \frac{L}{r^{2}}\right)$$

We wish to visualise $\partial_{t}r$ and $\partial_{t}\phi$ which we calculate from $\partial_{\eta}x^{\mu}$ using the chain rule and by solving the Hamilton-Jacobi for $\partial_{r}F$ which is then also substituted in.

Cadabra

```
dFsquared = linsolve(hj, $\partial_{r}{F
    }**2$)[0][0][1]
dF = $\partial_{r}{F} = \sqrt(@(
    dFsquared))$
gt = comp.get_component(orbital, $t$)

# dr/dt = dr/deta / dt/deta
gr = comp.get_component(orbital, $r$)
dr = substitute($@(gr) / @(gt)$, dF)
simplify(dr)

# dphi/dt = dphi/deta / dt/deta
gphi = comp.get_component(orbital, $\phi
    $)
dphi = $@(gphi) / @(gt)$
simplify(dphi)
```

SageMath

```
dF = solve(hj, diff(F, r))[1]

dr = (orbital[1] / orbital[0])     \
    .subs(diff(F, r) == dF.rhs())
dphi = (orbital[3] / orbital[0])
```

We define a set of fairly arbitrary parameters in both systems and call the numeric integration routine. In Cadabra we again use `integrate_ode`, and in Sage we import and use `desolve_odeint` which is also implemented using the SciPy `odeint` function. We set the parameters to fairly arbitrary values to give an aesthetically pleasing illustration of a possible result, in particular by setting m_{aux} and M_{aux} to distinct values greater than 0 we generate the non-stable orbit of a timelike particle.

Cadabra

```
m_aux = 1
M_aux = 1.0005
L_aux = 4.0
w_aux = 1.0
step_size = 0.01
eta_end = 500
r_initial = 2.1*M_aux
phi_initial = 0.3
params = {"w": w_aux, "L": L_aux, "m":
    m_aux, "M": M_aux}

ts, rs, phis = integrate_ode(
    (dr, dphi),
    $t, r, \phi$,
    [0, r_initial, phi_initial],
    eta_end, step_size, params)
```

SageMath

```
from sage.calculus.desolvers import
    desolve_odeint

var('m_aux, L_aux, w_aux, M_aux,
    r_initial, \
    ph_initial, eta_end, step_size')
m_aux = 1
M_aux = 1.0005
L_aux = 4
w_aux = 1.0
step_size = 0.1
eta_end = 500
r_initial = 2.1*M_aux
phi_initial = 0.3

params = {"w": w_aux, "L": L_aux, "m":
    m_aux, "M": M_aux}

sol = desolve_odeint(
    [dr.subs(**params), dphi.subs(**
        params)], [r_initial,
        phi_initial],
    srange(0, eta_end, step_size), [r,
        ph])
```

Finally, we plot $r\cos(\phi)$ against $r\sin(\phi)$, making use of NumPy's array operators in both systems and also taking advantage of some of the convenience functions `line` and `circle` in Sage.

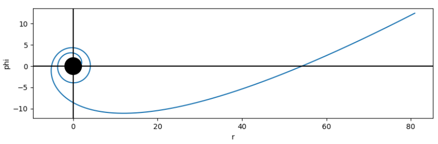
Cadabra

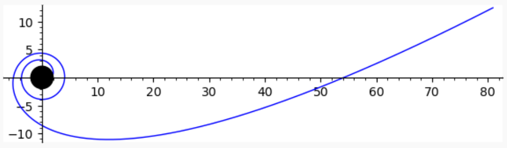
```
fig, ax = plt.subplots()
plt.plot(rs * np.cos(phis), rs * np.sin(
    phis))
ax.set_xlabel("r")
ax.set_ylabel("phi")
ax.axhline(y=0, color='k')
ax.axvline(x=0, color='k')
ax.add_patch(plt.Circle((0,0), 2*M_aux,
    color='black'))
ax.set_aspect('equal')
fig.set_size_inches(10,3)
fig;
```

SageMath

```
from sage.plot.circle import Circle

p = line(zip(sol[:,0] * cos(sol[:,1]),
    sol[:,0] * sin(sol[:,1])))
C = circle((0,0), 2*M_aux, fill=True,
    rgbcolor='black')
show(C + p)
```





2.5.5 Conclusion

Here we have shown how SageMath and Cadabra can both be used in General Relativity problems. Both systems provide ample inbuilt functionality for performing the calculation, but the approach which they take is markedly different. In particular, SageMath uses an object oriented approach where manipulating objects is performed primarily by calling methods on the objects it uses to represent mathematical quantities, whereas in Cadabra expressions are treated in a similar fashion to pen and paper calculations where algorithms and substitutions must be applied to them in order to perform the stages of a calculation.

This may require more work in order to manipulate an expression in Cadabra, and there is less “magic” provided in the software in the sense that the user is still required to possess the knowledge of how to manipulate the expressions by hand and is simply letting the software take care of calculating the intermediate steps. Much of this is hidden in SageMath, for example when calculating the Ricci tensor no prior calculation of the Riemann tensor is required. Of course, by creating a custom algorithm in Cadabra this process can be automated and even shipped as a third-party library. This is also true of auxiliary tools such as numerical solving and plotting tools which in SageMath are provided to work directly with its objects whereas Cadabra relies heavily on external libraries which it interfaces with. This can make it easier to get something up and running in SageMath, but allows Cadabra by design to be more customizable by allowing the user to plug the resulting expressions into any tool they want more easily.

As with almost all tools, there is no one solution which is appropriate in all cases, and whilst in this example both tools are suitable for performing the calculation if the requirements become more specialised it will become clear that one tool or the other provides more advantages. It is therefore important to be aware of the overall ecosystem of CAS tools available and their respective strengths and weaknesses.

Chapter 3

Canonicalisation

In the previous chapter we looked at the various capabilities of computer algebra systems and the types of problems which they can be used to solve. In this thesis we will focus on a specific function called *canonicalisation* which will culminate in a description of a novel approach to implementing this functionality. The concept of canonicalisation will be presented in this chapter with motivation for why it is an important and difficult topic in the field in order to prepare the way for the technical discussion on the mathematics and algorithmic implementation which will follow.

During the course of a calculation, it is often necessary, or at least desirable, to perform rearrangements of the expression to recast it into a *canonical* form. The exact meaning of canonical is imprecise and can vary depending on the exact field and context in which it is used[1] but can generally be described as a unique representation of an object which satisfies some (usually arbitrary) criteria such as being the simplest representation, the least representation if an ordering on the object can be defined, or a form which is generally accepted as being standard either by convention or to prepare the object for future calculations.

There are many different why a canonicalisation step may be needed in a calculation. Some common motives for transforming expressions into particular canonical forms include:

1. **Revealing recognisable forms for which solutions are known**

   Sometimes by collecting terms in a particular way solutions a solution can be made manifest. A good example of this is an expression which can be written as a total derivative:

   $$2xe^y\frac{dx}{dt} + x^2e^y\frac{dy}{dt} \equiv \frac{d}{dt}(x^2e^y) \tag{3.1}$$

   By collecting the terms like this the solution to the differential equations becomes obvious.

2. **Simplifying the calculation by removing redundant terms**

   Although this can be viewed an obvious or trivial step to apply, one of the most basic simplifications which can be made is to collect equivalent terms in a expression:

   $$x + x \equiv 2x \tag{3.2}$$

   In this example the simplification is very trivial, but in very large calculations (for instance perturbative expansions) where terms might be separated by thousands of other terms

[1] c.f. definitions given in `https://en.wiktionary.org/wiki/canonical_form` and `https://mathvault.ca/math-glossary/#canonical`

or factorized inside other expressions. By regularly checking an expression for equivalent terms and collecting these the overall complexity of a calculation is reduced.

3. **Finding expressions which are identical, or identically zero**

   There are numerous ways which expressions can be shown to equal zero through simplification: either through a special case of the removal of redundant terms which leaves no remaining terms in the expression

$$(x + y - 2z) + (y + z - 2x) + (z + x - 2y) \equiv 0 \tag{3.3}$$

   or by using some property of an object in the expression, like in this example where A is a antisymmetric matrix and Tr is the trace operator

$$Tr(A) \equiv 0 \tag{3.4}$$

4. **Highlighting particular parts or dependencies in the expression**

   Sometimes we would like a particular aspect of an expression to stand out. This is commonly achieved by factorising it out of the expression: here if we want to show that the expression has a linear dependence on w we can use the identity

$$wl + 2bw + m \equiv w(l + 2b) + m \tag{3.5}$$

5. **Creating a more aesthetically pleasing final form of an expression**

   A common final step in a calculation is to perform some simplification to make the result as intuitive to read as possible, a process which is very often subject to opinion or convention. One common example of this is the rationalisation of fractions:

$$\frac{\sqrt{x}+1}{\sqrt{x}-1} \equiv \frac{(\sqrt{x}+1)^2}{x-1} \tag{3.6}$$

This is by no means a comprehensive list of all the possible motives for wanting to find a canonical form or all the possible techniques which can be employed for doing so. In the study of tensors, canonicalisation most commonly refers to the rearrangement of the object's indices to bring them in to a standard form; the simplest example of this would be a symmetric rank 2 tensor $S_{ab} \equiv S_{ba}$. The discussion of the symmetries of these objects and how these can be used to produce canonical forms of tensorial expressions is the subject of this chapter.

3.1 Tensor symmetries

The symmetry of a tensor refers to the permutations of its indices which leave the object unchanged. Examples of ways in which these symmetries can manifest themselves in mathematical expressions include

1. A redundancy in the construction of the tensor, as is the case for the metric tensor g_{ij} whose entries are formed by the coefficients in the form of an infinitesimal line element

$$ds^2 = g_{00}dx^0dx^0 + g_{01}dx^0dx^1 + \cdots + g_{mm}dx^mdx^m \tag{3.7}$$

   where commutativity of the elements dx_i guarantees $g_{ij} = g_{ji}$

2. A consequence of the definition, as is the case of the electromagnetic tensor

$$F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu \tag{3.8}$$

where clearly by letting $\mu \leftrightarrow \nu$ the definition requires $F_{\mu\nu} = F_{\nu\mu}$

3. In physical contexts, these symmetries can derive from the nature of the object or the space in which it lives such as the stress tensor $\sigma_{ij} = \sigma_{ji}$ where the entries correspond to the stress of an object in each plane; if the off diagonal elements were not balanced then the object would have an innate torque which is not physically realised.

In all these cases the symmetries are all fundamental properties of the objects, but the purpose is to illustrate how one can observe these symmetries as being mathematically expressed and how they are applied to a particular tensor.

The symmetries described thus far have all involved an object being symmetric or antisymmetric under the exchange of some indices and can all be described by the general equation

$$T_{i_1 i_2 \cdots i_n} \equiv \epsilon_P T_{P(i_1 i_2 \ldots i_n)} \tag{3.9}$$

for some permutation of the indices P. This type of symmetry is referred to as a mono-term symmetry as it only relates single forms of the object, and these symmetries lead to a fairly natural definition of a canonical form for a tensor, as we can assign a lexicographic order to the indices and formulate the problem as finding the lexicographically least order of these indices which is equivalent to the object.

There are also a more general form of symmetries known as multi-term symmetries described by the general equation

$$\sum_j (\epsilon_{P_j} T_{P_j(i_1 i_2 \ldots i_n)}) \equiv 0 \tag{3.10}$$

By far the most famous class of these symmetries are the Bianchi identities, with the classic example being the first Bianchi identity of the Riemann tensor

$$R_{ijkl} + R_{iljk} + R_{iklj} \equiv 0 \tag{3.11}$$

These symmetries make the definition of a canonical form more problematic, as these identities mean that forms which have a least ordering of the indices may contain a far greater number of terms than other forms with fewer terms but less well ordered indices. Definitions of a canonical form for these have been proposed but the utility of them in practical applications is not always clear; they will still satisfy the requirement of distinct equivalent forms always mapping onto the same canonical form, but as will be discussed this may not be an optimal solution if this is the ultimate motive, and the form produced may not be desirable if it is a intermediate step in a calculation or an aesthetically pleasing final form is sought for.

3.2 Existing algorithms for tensor canonicalisation

Before moving on to describing the canonicalisation algorithm which we have developed, a look at existing attempts at building such an algorithm will be discussed. This will not only introduce some common ideas which will be useful throughout chapter 4 where we propose our own algorithm, but also give us an opportunity to look more specifically at the benefits and drawbacks of various approaches.

3.2.1 Canonicalisation in Early CAS

The introduction of indexed tensors into computer algebra systems came in the 1970s with systems such as CLAM [55] which emphasised its utility in the field of general relativity. However, general algorithms for canonicalisation of tensor expressions had not yet been developed and canonicalisation routines remained rooted in the other simplifications outlined above. Methods for calculating the equivalence of two expressions often required users to manually spot identities, or to expand out expressions into basic elements and use the (hopefully) simple symmetries of these objects, for instance by expanding out the Riemann tensor in terms of the metric tensor and its derivatives:

$$\begin{aligned}
R_{\rho\sigma\mu\nu} =& \frac{1}{4}\Big(2\partial_{\mu\sigma}(g_{\rho\nu}) + 2\partial_{\mu\nu}(g_{\rho\sigma}) - 2\partial_{\mu\rho}(g_{\nu\sigma}) - 2\partial_{\nu\sigma}(g_{\rho\mu}) - 2\partial_{\nu\mu}(g_{\rho\sigma}) \\
&+ 2\partial_{\nu\rho}(g_{\mu\sigma}) + \partial_{\lambda}(g_{\rho\mu})\partial_{\sigma}(g_{\lambda\nu}) + \partial_{\lambda}(g_{\rho\mu})\partial_{\nu}(g_{\lambda\sigma}) - \partial_{\lambda}(g_{\rho\mu})\partial_{\lambda}(g_{\nu\sigma}) \\
&+ \partial_{\mu}(g_{\rho\lambda})\partial_{\sigma}(g_{\lambda\nu}) + \partial_{\mu}(g_{\rho\lambda})\partial_{\nu}(g_{\lambda\sigma}) - \partial_{\mu}(g_{\rho\lambda})\partial_{\lambda}(g_{\nu\sigma}) - \partial_{\rho}(g_{\mu\lambda})\partial_{\sigma}(g_{\lambda\nu}) \\
&- \partial_{\rho}(g_{\mu\lambda})\partial_{\nu}(g_{\lambda\sigma}) + \partial_{\rho}(g_{\mu\lambda})\partial_{\lambda}(g_{\nu\sigma}) - \partial_{\lambda}(g_{\rho\nu})\partial_{\sigma}(g_{\lambda\mu}) - \partial_{\lambda}(g_{\rho\nu})\partial_{\mu}(g_{\lambda\sigma}) \\
&+ \partial_{\lambda}(g_{\rho\nu})\partial_{\lambda}(g_{\mu\sigma}) - \partial_{\nu}(g_{\rho\lambda})\partial_{\sigma}(g_{\lambda\mu}) - \partial_{\nu}(g_{\rho\lambda})\partial_{\mu}(g_{\lambda\sigma}) + \partial_{\nu}(g_{\rho\lambda})\partial_{\lambda}(g_{\mu\sigma}) \\
&+ \partial_{\rho}(g_{\nu\lambda})\partial_{\sigma}(g_{\lambda\mu}) + \partial_{\rho}(g_{\nu\lambda})\partial_{\mu}(g_{\lambda\sigma}) - \partial_{\rho}(g_{\nu\lambda})\partial_{\lambda}(g_{\mu\sigma})\Big)
\end{aligned} \tag{3.12}$$

where, as both partial derivatives and metric tensor are simple-symmetric in both their indices, the transformation into a canonical form is trivial and can be compared term-wise against any other index permutation of $R_{\rho\sigma\mu\nu}$ to check for equality.

Intelligent canonicalisation routines started to appear in the 1990s with the introduction of packages which integrated into existing CAS, such as the *ATENSOR* package for the REDUCE program in 1996 [56] and the `Arrange` and `SymmApply` functions from the 1998 *Tools of Tensor Calculus* package for Mathematica [57] and the improved version of their algorithm `SimplifyAllIndex` described in the follow-up paper [58]. Both these tools rely on directly applying side relations in order to find a canonical form, which for objects with many side-relations, such as the Riemann tensor, becomes very cumbersome.

It is also at around this time that the literature begins to separate the problem of canonicalisation into the two separate problems of dummy symmetries and slot symmetries, although the use of both types was of course already well understood.

Slot symmetries arise as a result of the reasons outlined above and directly relate one form of a tensor to another by permuting two or more of its index slots, for example $T_{abcd} \equiv T_{adcb}$. Dummy symmetries however arise from the extra degrees of freedom in labelling indices introduced by contracted indices, e.g. $T^{a}{}_{a}{}^{b}{}_{b} \equiv T^{b}{}_{b}{}^{a}{}_{b}$. Furthermore, depending on the symmetry of the metric associated with these indices we could generate further identities by raising and lowering the indices. Two general approaches for handling dummy indices were in circulation at this time; the first, used by the ATENSOR package, was to treat these as another set of side relations e.g.

$$S_{jk}T^{kji} - S_{kj}T^{jki} = 0 \tag{3.13}$$

however the number of extra relations this produces grows with the number of permutations of dummy index names resulting in an $O(n!)$ algorithm. Mathematica packages could make use of the included pattern-matching functions to find expressions which are equivalent up to renamings, much as how nowadays we might use the regular expression `/T\^(\w)_\1\^(\w)_\2/` to match both $T^{a}{}_{a}{}^{b}{}_{b}$ and $T^{b}{}_{b}{}^{a}{}_{a}$. This does not of course find a canonical form of an expression and does not integrate well with the canonicalisation of slot symmetries.

At around the same time the representation of tensors using group theory, which had been discussed at least as early as 1946 in Weyl's *The Classical Groups* [59] which examined the relationship between the symmetric group and tensor symmetries, started to garner interest in the computing community with the publishing of papers such as [60] where a more advanced technique for dealing with multi-term side relations using the construction of a Gröbner basis and a way of dealing with dummy indices generally was introduced.

3.2.2 Permutations: A Brief Digression

As the language of permutations will be central not only to the following discussion but throughout the rest of the chapter, before continuing some of the concepts and conventions will be covered here.

Permutations are described by elements of the symmetric group S_n whose elements σ_i consist of all the $n!$ possible ways of arranging the elements $1, 2, \ldots, n$. These elements are commonly written in a cyclic notation, so that the element $(1, 4, 5)$, or if unambiguous (145), indicates the rearrangement $1 \to 4$, $4 \to 5$ and $5 \to 1$. Elements can be combined by following where each cycle goes to, for example $(1, 2, 3) \star (2, 4) \equiv (2, 4, 1, 3)$. Combinations which do not all contain the same set of elements may be disjoint and cannot be combined into a single cycle, such as $(1, 2)(3, 4)$. Each element of the group can be classified as either even or odd depending on the number of swaps it requires to construct, for instance $(2, 3)$ which only requires one swap $2 \leftrightarrow 3$ is an odd permutation whilst $(1, 2, 3, 4, 5)$ which uses 4 swaps is even.

Another way of describing a permutation is as a *permutation list*, where the ith item of the list describes where the element i maps to, for instance the element $(1, 4, 5)$ as a permutation list could be described as $[4, 2, 3, 5, 1, 6, 7, \ldots]$ where the length of the list is given by the size of the group S_n. It is often convenient to describe the objects we wish to calculate in terms of permutation lists and the elements of the group in cyclic notation as this makes calculations more intuitive: for instance starting from a permutation object $[1, 3, 5, 4, 2]$, applying the group element $(2, 3)$ simply involves permuting the elements at these positions: $(2, 3) \star [1, 3, 5, 4, 2] = [1, 5, 3, 4, 2]$ which is equivalent to $(2, 3) \star (2, 3, 5) = (2, 5)$

3.2.3 The Butler-Portugal Algorithm

From here on the application of group theory to the problem stimulated great advances, particularly into the area of mono-term canonicalisation with the publishing of the paper [61] which presented a new algorithm, now known as the *Butler-Portugal* algorithm, for efficiently dealing with expressions containing arbitrary slot-symmetries and in the second half of the paper extending this to also handle dummy indices. It is worth going through the idea behind the algorithm to show why it is difficult to extend this approach for handling multi-term symmetries.

The algorithm can quickly become tedious to calculate by hand, however to elucidate its behaviour we will follow through a simple example of finding the canonical form of the Riemann monomial

$$-R^{dcab} \tag{3.14}$$

where we define the expected lexicographic ordering

$$L = [a, b, c, d] \tag{3.15}$$

This particular choice is made in order to illustrate as many different aspects of the algorithm as possible while still only requiring a small amount of manual calculation.

The mono-term symmetries of the Riemann tensor can be described in numerous different ways; commonly we use a set of identities

$$\begin{aligned} R_{abcd} &\equiv R_{cdab} \\ R_{abcd} &\equiv -R_{bacd} \end{aligned} \tag{3.16}$$

From these two identities, a total of 7 representations of the Riemann tensor can be reached by suitable combinations, for example $-R_{abdc} \equiv -R_{dcab} \equiv R_{cdab} \equiv R_{abcd}$. A complete list of the the permutations required to reach all 7 identical representations is another way of representing the group:

$$G = \{I, -(0,1), -(2,3), (0,2)(1,3), -(0,3,1,2), -(0,2,1,3), (0,3)(1,2)\} \tag{3.17}$$

A representation such as this is far more explicit, however for larger groups with more symmetry it can become unwieldy to hold the complete set of allowable permutations, especially as the full symmetric group S_n has $n!$ elements. In order to mitigate this a third representation consisting of two sets, a *base generating set* and a *strong generating set* which are often collectively referred to as a *BSGS* (base and strong generating set), is often preferred for actual computations.

The base generating set B is an ordered set of elements Q such that the pointwise stabiliser of B in G is trivial. The stabiliser subgroup of a point b_i is a set of elements of G which leave b_i fixed: $\text{Stab}_G(b_i) = \{g \in G \mid b_i \star g = b_i\}$. For example, in the group of Riemann symmetries

$$\text{Stab}_G(0) = \{I, -(2,3)\} \tag{3.18}$$

as all other elements permute 0. The pointwise stabiliser generalises this to a set of points:

$$\text{PointwiseStab}_G(B) = \{g \in G \mid \forall b_i \in B, b_i \star g = b_i\} \tag{3.19}$$

For a given group G there are multiple different base generating sets we could choose, to make computations as efficient as possible a *reduced* base set, i.e. the smallest set we can form, is preferred. For the Riemann tensor symmetries, there are a variety of reduced base generating sets including $[0,2]$, $[2,0]$ and $[1,3]$.

For each base generating set B we can define a *stabilizing chain* which is a set of subgroups $G^{(i)}$ of G each of which stabilises another point in B. The subgroups are recursively defined as

$$G^{(i)} = \begin{cases} G & \text{if } i = 1 \\ \text{Stab}_{G^{(i)}}(b_i) & \text{if } i > 1 \end{cases} \tag{3.20}$$

such that for a base generating set of n elements, $G^{(n+1)} = \{I\}$. Given the base generating set $[0,2]$, the stabilising chain for the symmetries (3.17) is

$$\begin{aligned} &G^{(1)} = \{I, -(0,1), -(2,3), (0,2)(1,3), -(0,3,1,2), -(0,2,1,3), (0,3)(1,2)\} \\ &G^{(2)} = \text{Stab}_{G^{(1)}}(0) = \{I, -(2,3)\} \\ &G^{(3)} = \text{Stab}_{G^{(2)}}(2) = \{I\} \end{aligned} \tag{3.21}$$

For a given base generating set, the corresponding strong generating set is a set of generators S for G such that $\langle S \cap G^{(i)} \rangle = G^{(i)}$ for all i in the stabilising chain, where $\langle K \rangle$ represents the group generated by the elements of K. The calculation of the strong generating set is non trivial to do by hand and is often performed alongside the calculation of a base generating set using the

Schreier-Sims algorithm [62], of which there are several variations but which in essence starts from a partial BSGS and extends it by considering new elements to find if they are redundant or not until the full BSGS is developed. A strong generating set for (3.17) relative to the base $[0, 2]$ is given by

$$S = \{-(0,1), -(2,3), (0,2)(1,3)\} \tag{3.22}$$

We can show that this follows the definition above:

$$\begin{aligned} \langle S \cap G^{(0)} \rangle &= \langle I, -(0,1), -(2,3), (0,2)(1,3) \rangle = G^{(0)} \\ \langle S \cap G^{(1)} \rangle &= \langle I, -(2,3) \rangle = G^{(1)} \\ \langle S \cap G^{(2)} \rangle &= \langle I \rangle = G^{(2)} \end{aligned} \tag{3.23}$$

The first line is true because S can generate (3.17), and the second line as

$$-(2,3) \star -(2,3) = I \tag{3.24}$$

so $\langle G^{(1)} \rangle = G^{(1)}$.

The Butler-Portugal algorithm takes as input the BSGS, in order to encode the symmetries it must use to canonicalise an expression, as well as a representation of the tensor as a permutation from the *base canonical form* which is the form the tensor would take it all its indices were perfectly ordered. In our example tensor given in (3.14), this would be $[3, 2, 0, 1]$ where the ith entry is the position of the index at slot i of the tensor in L (which for consistency with most programming languages we zero-index contrary to the discussion above, so slot 0 contains $L[3] = d$, slot 1 contains $L[2] = c$ etc.).

In order to encode the sign of the tensor, two extra elements are appended to the representation which encode the ordering of the indices of a tensor $\epsilon_{ij} = -\epsilon_{ji} = 1$ with the two extra elements i and j appended to L. Thus, the permutation which needs to be supplied as input to the Butler-Portugal algorithm for (3.14) is $[3, 2, 0, 1, 5, 4]$.

The Butler-Portugal algorithm is able to efficiently canonicalise an expression by using the properties of the stabiliser chain to progressively canonicalise the expression. Intuitively, it reduces the problem of finding the canonical form for the entire set of indices into smaller problems based around fixing each element in the base generating set. At each iteration, the slots which are connected by some symmetry with the ith element of the base set are considered and a partial canonicalisation in these slots with respect to the element is performed. Any symmetry element which doesn't fix this point can now be removed from consideration, and the process is repeated for the remaining elements of the base generating set.

Explicitly, the steps performed at each iteration are

1. Calculate Δ^{b_i}, the orbit of b_i (i.e. all slots which can be reached from b_i by applications of elements of K).

2. Calculate k, the position of the minimum element of λ in Δ^{b_i}

3. Calculate p, the kth point of Δ^{b_i}

4. Find the symmetry element ω such that $b_i \star \omega = p$

5. Set $\lambda \to \lambda \star \omega$

6. Remove elements of K which do not fix b_i

In our example we need to perform two iterations:
Iteration 1 ($b_0 = 0$):

1. $\Delta^0 = \{0, 1, 2, 3\}$, as
$$\begin{aligned} &0 \star I = 0 \\ &0 \star (0,1)(4,5) = 1 \\ &0 \star (0,2)(1,3) = 2 \\ &0 \star (0,2)(1,3) \star (2,3)(4,5) = 3 \end{aligned}$$
2. $k = 2$, as $\min(\Delta^0) = 0$ and $\lambda[2] = 0$
3. $p = 2$, as $\Delta^0[2] = 2$
4. $\omega = (0,2)(1,3)$, as $0 \star (0,2)(1,3) = 2$
5. $\lambda = [3,2,0,1,4,5] \star (0,2)(1,3) = [0,1,3,2,4,5]$
6. $K \to \{I, (2,3)(4,5)\}$, as the elements $(0,1)(4,5)$ and $(0,2)(1,3)$ do not fix 0

Iteration 2 ($b_1 = 2$):

1. $\Delta^2 = \{2, 3\}$
2. $k = 3$
3. $p = 0$
4. $\omega = (2,3)(4,5)$
5. $\lambda = [0,1,3,2,4,5] \star (2,3)(4,5) = [0,1,2,3,5,4]$
6. $K \to \{I\}$

We now have a canonical representation of the tensor in permutation form. The last two elements $[5, 4]$ tell us that we have a factor $\epsilon_{ji} = -1$, and the ith index of the tensor is given by the contents of $L[i]$, so the final result is $-R^{abcd}$; as this is the base canonical form it is trivial to confirm that this is the canonical representation of the input.

Note that the method presented here differs slightly from the example in the first paper of [61], as we have defined a different lexical ordering, and have adopted the trick of multiplying through by ϵ_{ij} demonstrated in the follow-up paper in order to more clearly present how the algorithm might be implemented. Some of the steps are also presented differently, in particular (4) which in the paper is described as finding the trace of a Schreier vector which is an important implementation detail but is a less intuitive way of thinking about the logic of the algorithm at this stage.

Of course, this algorithm can only handle the slot symmetries of a tensor. The follow-up presents an algorithm which can also handle the dummy symmetries in a similar manner by constructing the group D of all dummy symmetries, which act on the tensor with $d \star \lambda$ (as opposed to S which acts from the right), and so the complete set of representations is formed by the double coset $D \star \lambda \star S$. There are multiple extra steps which must be taken in order to find the minimum element of this set which are not interesting to the current discussion and so will not be discussed here, however for anyone interested in the mechanics of this I recommend studying the implementation of the function `double_coset_can_rep` in SymPy[2] which follows the algorithm described in the paper very closely, is very readable and contains a long and detailed explanatory docstring.

The main point which I aim to highlight in this discussion is that the use of a generating set in order to detect equivalence works well for mono-term symmetries, but is unsuited to multi-term symmetries. It is possible to imagine an extension where elements of the generating set can map onto multiple terms, for instance in order to include the cyclic symmetry we might construct the generating set

$$K = \{I, (0,1), (2,3), (0,2)(1,3), (1,2,3) + (3,1,2), \text{other combinations...}\} \tag{3.25}$$

however the structure of this is not compatible with many of the techniques used in the algorithm and the result is that we are once again reduced to treating the generating set as a way of producing side relations which can be applied to the expression.

3.2.4 Time Complexity of Canonicalisation

Before moving on to proposing a different general solution to this problem, it is worth quickly commenting on the time complexity of the various approaches described here as the efficiency of an algorithm is one of the main driving factors. It has already been discussed how the ATENSOR implementation gives rise to $O(n!)$ complexity due to the total size of the group of all dummy renamings. Other methods which use side relations are dependent on the number of relations required to fully describe the symmetry, which can grow very large if polynomial expressions are considered.

The Butler-Portugal algorithm as described above runs in an estimated $O(p^5)$ time for Riemann polynomials of degree p, and although for many classes of symmetry runs in polynomial time although the algorithm can run in exponential time for many highly symmetric cases. This was the point raised in the paper [63] whose proposed improvement mitigates runtime in the catastrophic cases of fully symmetric sets of indices such as $T_{abcdef}S^{ebfdac}$ which runs in factorial time. However, the solution is essentially a special-case detection algorithm when fully symmetric subsets of indices occur, and objects described by many pairwise symmetries (e.g. $(i, i+2)(i+1, i+3)$) still suffer the same bottleneck.

The main point to be drawn from this is that while many algorithms may run quickly for most common cases, and the performance overheads of modern implementations solve all but the most pernicious of inputs in a reasonable time, in the general case canonicalisation is still in the worst case a $O(n!)$ problem.

3.3 Tensor symmetries as Young diagrams

The algorithm which will be presented in the next chapter is also based on the idea of representing tensors and their symmetries by permutations, but a specific representation of these symmetries known as *Young symmetrisers* become the natural language for describing this algorithm.

[2] Which can be viewed on github at
`https://github.com/sympy/sympy/blob/00d6469eafdd4aac346a0b598184c15f2560dbe5/sympy/combinatorics/tensor_can.py#L164-L535`

3.3.1 Representations of S_n

A representation of the symmetric group S_n is an operator $\psi : S_n \to V$ for some vector space V. By choosing V to be the space spanned by the index slots of a tensor, the symmetry of a tensor can be described by choosing a suitable representation of the group. For example, the simplest representation we can construct, which is known as the *trivial representation*, maps every element of S_n to the identity element:

$$\psi_{\text{triv}}(\sigma_i) = I \tag{3.26}$$

where ψ_{triv} is the representation and σ_i is an element of S_n. A tensor can be represented by the element of S_n corresponding to the permutation of its indices from base canonical form, and so the symmetry encoded by this representation is that any permutation of indices is equivalent to the base canonical form, i.e. the tensor is completely symmetric. To illustrate with an example, consider a standard metric tensor $g_{\mu\nu}$ which is symmetric in its two indices. We can represent the possible forms of this tensor as elements of the group S_2 which contains the two elements $\{I, (1,2)\}$ corresponding to $\{g_{\mu\nu}, g_{\nu\mu}\}$ respectively. We can find the action of a symmetry operation by seeing where the representation maps each element to. As the metric tensor is symmetric, it is described by the trivial representation

$$\begin{aligned}\psi_{\text{triv}}(g_{\mu\nu}) &= g_{\mu\nu}\\ \psi_{\text{triv}}(g_{\nu\mu}) &= g_{\mu\nu}\end{aligned} \tag{3.27}$$

Another common representation of S_n is given by

$$\psi_{\text{alt}}(\sigma_i) = \begin{cases} I & \text{if } \sigma_i \text{ is an even permutation} \\ -I & \text{if } \sigma_i \text{ is an odd permutation}\end{cases} \tag{3.28}$$

This is known as the *alternating representation* as the sign alternates between positive and negative depending on the parity of the permutation. Objects whose symmetry is described by the alternating representation are totally antisymmetric, for instance $\epsilon_{[ijk]}$[3] is an element of S_3 which obeys

$$\begin{aligned}\psi_{\text{alt}}(\epsilon_{ijk}) = \psi_{\text{alt}}(\epsilon_{jki}) = \psi_{\text{alt}}(\epsilon_{kij}) &= \epsilon_{ijk}\\ \psi_{\text{alt}}(\epsilon_{ikj}) = \psi_{\text{alt}}(\epsilon_{kij}) = \psi_{\text{alt}}(\epsilon_{jik}) &= -\epsilon_{ijk}\end{aligned} \tag{3.29}$$

The natural question to ask is how many representations are needed in order to fully describe a rank-n tensor, or in other words to cover S_n? Clearly for $n = 1$ any representation is equivalent to the trivial representation up to an overall constant and this suffices to cover S_1. For $n = 2$, we can show that $\{\psi_{\text{triv}}, \psi_{\text{alt}}\}$ is able to cover the whole group. The most intuitive way to see this is by decomposing a a general second rank object using the well known formula:

$$M_{ab} \equiv \frac{1}{2}(M_{ab} + M_{ba}) + \frac{1}{2}(M_{ab} - M_{ba}) \tag{3.30}$$

Here we have added and subtracted M_{ba} so this always holds, but importantly the first bracket transforms under ψ_{triv} and the second under ψ_{alt}. As we have imposed no other symmetry on M_{ab} this shows that these two representations are sufficient to cover S_2. Furthermore, as they are both one-dimensional representations they are also irreducible.

[3] Where $[\dots]$ as usual denotes antisymmetry in these indices

3.3.2 Irreps of S_n as Young Tableaux

Finding decompositions of rank-n tensors can therefore be used to find the set of representations which cover S_n, but this is a laborious process in higher dimensions which is easy to get wrong. Instead, a far simpler approach based on constructions of diagrammatic objects introduced by Alfred Young in 1900 [64], and now known as *Young tableaux* is commonly used. By using this method, all irreps of S_n can be constructed by following a set of rules:

1. Calculate all unique partitions of n. A partition, commonly denoted $\lambda \vdash n$, is a set of numbers $\lambda = \{l_1, l_2, \ldots, l_k\}$ such that $\sum_{i=1}^{k}(l_i) = n$ and $l_i \neq 0$.
2. For each partition, construct the corresponding *Young diagram*, which consists of k rows of cells, each row with l_i cells, and ordered so that each row is no longer than the previous row.
3. For each diagram, fill the cells with the labels $1, 2, \ldots, n$ so that in each row and column the numbers are strictly increasing 'south' and 'east'. The filled diagrams are the tableaux.
4. The corresponding irrep is found by symmetrising the elements of S_n in each row and antisymmetrising the elements in each column.

To illustrate let us explicitly calculate the irreps of S_3:

1. The three partitions of 3 are $\lambda_1 = \{3\}$, $\lambda_2 = \{2, 1\}$ and $\lambda_3 = \{1, 1, 1\}$
2. The corresponding diagrams are

$$\lambda_1 = \begin{array}{|c|c|c|}\hline & & \\ \hline \end{array} \quad \lambda_2 = \begin{array}{|c|c|}\hline & \\ \hline \multicolumn{1}{|c|}{} \\ \cline{1-1} \end{array} \quad \lambda_3 = \begin{array}{|c|}\hline \\ \hline \\ \hline \\ \hline \end{array} \tag{3.31}$$

3. There is only one way of filling λ_1 and λ_3:

$$\begin{aligned} \psi_{\text{triv}} &= \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline \end{array} \\ \psi_{\text{alt}} &= \begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array} \end{aligned} \tag{3.32}$$

These diagrams have been suggestively equated with the trivial and alternating representation: in general any tableau consisting of a single row is a trivial representation and a single column is an alternating representation.
There are two ways of filling λ_2:

$$\psi_{\text{std}_1} = \begin{array}{|c|c|}\hline 1 & 2 \\ \hline \multicolumn{1}{|c|}{3} \\ \cline{1-1} \end{array} \tag{3.33}$$

$$\psi_{\text{std}_2} = \begin{array}{|c|c|}\hline 1 & 3 \\ \hline \multicolumn{1}{|c|}{2} \\ \cline{1-1} \end{array} \tag{3.34}$$

These are known as *standard* representations, with various uses described in e.g. [65] and the symmetry they encode is neither fully symmetric nor anti-symmetric.

Although we will be using these tableaux to study the symmetry of tensors, for completeness it should be mentioned that any object or system displaying a similar symmetry under permutation of its elements can be described using representations of the symmetric group. For instance, the wavefunction describing two fermions $\Psi(x_1, x_2)$ is antisymmetric under particle interchange, which is equivalent to a permutation of its two arguments and is therefore described by the irrep of S_2

$$\begin{array}{|c|}\hline x_1 \\ \hline x_2 \\ \hline \end{array} \tag{3.35}$$

In Cadabra, a representation can be associated to a tensor by the use of the `TableauSymmetry` property. For example, associating the tableau (3.34) to T_{abc} can be done as follows:

```
T_{a b c}::TableauSymmetry(shape={2,1}, indices={0,2,1}).
```

where the `shape` parameter is an ordered list of the rows of the tableau, and `indices` are the zero-indexed slots of the tensor which are associated with the cells of the tableau left-to-right and top-to-bottom.

The relation between these diagrams and the symmetry of a tensor is fully encoded in an operation known as a *Young projection* which gives us a way to construct a generic decomposition such as in (3.30). This begins by associating each label in the diagram to an index slot in the tensor. For $\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline \end{array}$ therefore, where we symmetrise across the labels 1, 2 and 3 the associated tensor is $T_{(abc)}$ (symmetric in all three indices) and similarly, the tableau[4] $\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline \end{array}^T$ the tensor $T_{[abc]}$. Note however, that the standard representations describe truly mixed symmetries, not two independent symmetries, which we might notate as something along the lines of $T_{(a[b)c]}$ which is symmetric in slots $1, 2$ and antisymmetric in slots $2, 3$. Indeed, any such object which has an index slot which belongs simultaneously to a set of symmetrised indices and a set of antisymmetrised indices must be identically zero as we can perform an odd number of antisymmetrisations to return to the original permutation of indices:

$$T_{abc} \equiv -T_{acb} \equiv -T_{cab} \equiv T_{cba} \equiv T_{bca} \equiv -T_{bac} \equiv -T_{abc} \tag{3.36}$$

The only way that this can be true is for $T_{abc} \equiv 0$. With more than three indices, we can perform the exact same sequence of permutations selecting a from a set of symmetrised indices and c from a set of antisymmetrised indices of which b belongs to both sets.

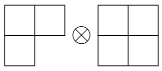
3.3.3 Combinations of Young Diagrams and Tableaux

A useful property of Young diagrams is that it is possible to rewrite any product of two tableaux taken from S_n and S_m as a direct sum of tableaux in S_{n+m}, by a specific application of a process known as the Littlewood-Richardson rule which describes how to decompose a product of two Schur functions into a sum of other Schur functions. The process can be recast in terms of a simple algorithm [66] taking as inputs two diagrams λ_1 and λ_2. We start by labelling each cell in row r_i of λ_2 with i. Then, for each row r_i in λ_2, place each cell in the row to the end of a row in λ_1 subject to the constraints:

1. λ_1 is a valid Young diagram at each stage.
2. There is not more than one cell with the label i in each column of λ_1.
3. Reading right-to-left and top-to-bottom, the number of cells encountered with label i must be no greater than the number of cells with label $i - 1$.

[4] Where the transposition operator ^{T} is occasionally used for convenience and describes $\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline \end{array}^T \equiv \begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array}$

After erasing the labels attached to λ_1, each possible combination is a term in the decomposition. For example, to calculate

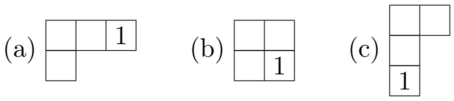


(3.37)

we first label the cells in as follows:

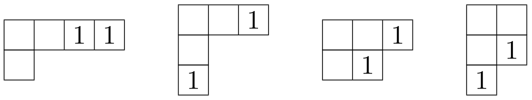
$$\begin{array}{|c|c|}\hline 1 & 1 \\ \hline 2 & 2 \\ \hline \end{array} \qquad (3.38)$$

We attach the two cells labelled 1 in the first row. The first $\boxed{1}$ can be appended to any row:



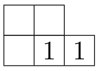
(3.39)

The second $\boxed{1}$ can be added to rows 1 and 2 of (a), 1 and 3 of (b), and only row 1 of (c) which yields an identical diagram, so this generates a total of four diagrams:



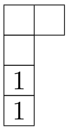
(3.40)

Note that e.g.



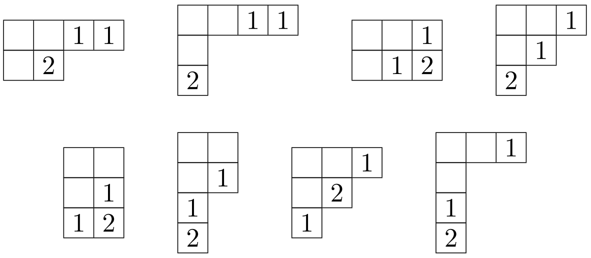
(3.41)

is invalid by rule 1 and



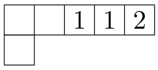
(3.42)

is invalid by rule 2 as $\boxed{1}$ appears in the same column twice. We now move on to the second row consisting of the cells $\boxed{2}$. There are 8 legal positions for the first cell:



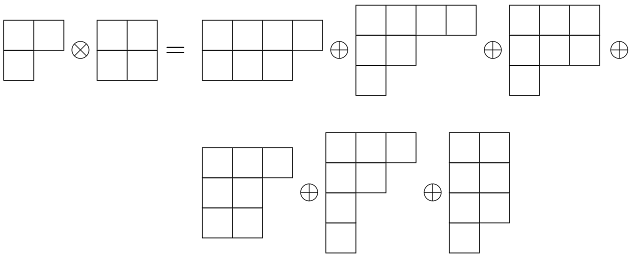
(3.43)

where a diagram such as



(3.44)

is invalid by rule 3. Finally, attaching the last $\boxed{2}$ and removing the labels we get the six unique terms which make up the final sum:



(3.45)

Note that the product commutes:

$$\lambda_1 \otimes \lambda_2 \equiv \lambda_2 \otimes \lambda_1 \tag{3.46}$$

and that it is always more efficient to put the simpler tableau on the right hand side. Many CAS packages include a routine for calculating this decomposition, including the `lr_tensor` function in Cadabra as well as `sage.libs.lrcalc.lrcalc` from Sage which is a binding of the `lrcalc` C program by Anders Buch[5].

This procedure can also be carried out with tableaux where the cells are already labelled. This allows the symmetry of an expression given by the tensor product of two objects to be calculated, for instance if we have a wavefunction of two particles $\Phi(x_1, x_2)$ which is symmetric under interchange, then by adding a third particle x_3 to the system the total symmetry of the system is described by

$$\begin{array}{|c|}\hline x_1 \\ \hline x_2 \\ \hline \end{array} \otimes \begin{array}{|c|}\hline x_3 \\ \hline \end{array} = \begin{array}{|c|c|}\hline x_1 & x_3 \\ \hline x_2 \\ \cline{1-1} \end{array} \oplus \begin{array}{|c|}\hline x_1 \\ \hline x_2 \\ \hline x_3 \\ \hline \end{array} \tag{3.47}$$

If it was then required that the system *not* be antisymmetric under exchange with x_3, the second term would be dropped leaving only the mixed symmetry term. Similarly, the symmetry of the product of tensor expressions, e.g. $A_{ab}C_{cd}$ can be calculated.

3.3.4 The Young Projection Operator

As alluded to above, the use of these tableaux allow us to fully encode the symmetry of a tensor by performing a projection operation, which creates an decomposition of an object which makes the symmetry manifest; that is, any two tensors which are identical under some symmetry transformation will yield identical projections. A simple and intuitive example of a projection operation is given by the two brackets in (3.30): the first is the result of the projection

$$P\left(\begin{array}{|c|c|}\hline 1 & 2 \\ \hline \end{array}\right) M_{ab} = \frac{1}{2}(M_{ab} + M_{ba}) \tag{3.48}$$

and the second

$$P\left(\begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline \end{array}\right) M_{ab} = \frac{1}{2}(M_{ab} - M_{ba}) \tag{3.49}$$

It should be easily verifiable by eye that the symmetry $M_{(ab)}$ is made manifest by this projection, but to explicitly calculate this is similarly simple:

$$P\left(\begin{array}{|c|c|}\hline 1 & 2 \\ \hline \end{array}\right) M_{ba} = \frac{1}{2}(M_{ba} + M_{ab}) = P\left(\begin{array}{|c|c|}\hline 1 & 2 \\ \hline \end{array}\right) M_{ab} \tag{3.50}$$

where no recalculation of the projector is necessary, as it can be deduced from (3.48) by rewriting the indices $a \leftrightarrow b$.

The general formula for calculating a projection operator is given by

$$P^+(Y) = \frac{1}{N} \prod_{r \in Y} \mathcal{S}_r \prod_{c \in Y} \mathcal{A}_c \tag{3.51}$$

Here r and c are the rows and columns of the tableau, $\mathcal{S}$ and $\mathcal{A}$ the symmetrisation and antisymmetrisation operators and N is a normalisation constant given by the product of the hook-lengths of each cell, the hook-length of a cell being the number of cells to the right and

[5] `https://sites.math.rutgers.edu/~asbuch/lrcalc/`

below a cell including the cell itself: in the diagram below each cell contains its hook length

$$\begin{array}{|c|c|c|c|c|}\hline 7 & 5 & 4 & 3 & 1 \\ \hline 5 & 3 & 2 & 1 \\ \cline{1-4} 1 \\ \cline{1-1}\end{array} \tag{3.52}$$

and the normalisation would be $N = 7 \times 5^2 \times 4 \times 3^2 \times 2 \times 1^3 = 12600$.

There are actually two distinct ways of constructing the projection operator depending on the order in which the symmetrisation and antisymmetrisation operators are applied. The operator P^+ defined above applies antisymmetrisations first, the other operator being

$$P^-(Y) = \frac{1}{N} \prod_{c \in Y} \mathcal{A}_c \prod_{r \in Y} \mathcal{S}_r \tag{3.53}$$

Which operator is used is a matter of taste and convention; as long as the two are not mixed the mathematics is the same and symmetries become manifest under both. We will use the form of the projector given in (3.51) throughout and will drop the + label.

The simplest projection operator which mixes a symmetriser with an antisymmetriser is from one of the standard representations given in (3.33) and (3.34). Here we will explicitly calculate the projection for T_{abc} with the tableau

$$Y = \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 \\ \cline{1-1}\end{array} \tag{3.54}$$

First we calculate the projector $P(Y)$. As there is only one column and one row, the formula reduces to

$$P(Y) = \frac{1}{N} \, \mathcal{S}_{12} \mathcal{A}_{13} \tag{3.55}$$

and the hook length is given by $N = 3 \times 1^2$. We now apply this to T_{abc}, first applying the antisymmetriser:

$$P(Y) T_{abc} = \frac{1}{3} \, \mathcal{S}_{12} (T_{abc} - T_{cba}) \tag{3.56}$$

and then the symmetriser to each generated term

$$P(Y) T_{abc} = \frac{1}{3} (T_{abc} + T_{bac} - T_{cba} - T_{bca}) \tag{3.57}$$

There are two identities associated with the symmetry represented by (3.54): one mono-term symmetry

$$T_{abc} \equiv -T_{cba} \tag{3.58}$$

and one cyclic identity

$$T_{abc} + T_{bca} + T_{cab} \equiv 0 \tag{3.59}$$

We can now see the power of the projection by verifying that both these identities are made manifest by the projection. For (3.58) we project both sides of the identity and match the terms on each side:

$$\begin{tikzcd}[column sep=small] P(Y)T_{abc} = \frac{1}{3}(& T_{abc} + \arrow[drr] & T_{bac} - \arrow[drr] & T_{cba} - \arrow[dll] & T_{bca}) \arrow[dll] \\ P(Y)(-T_{cba}) = \frac{1}{3}(& -T_{cba} - & T_{bca} + & T_{abc} + & T_{bac}) \end{tikzcd} \tag{3.60}$$

and for (3.59) project each term on the left hand side to show that each term in the expansion is balanced by one with an opposite sign:

$$\begin{aligned}
P(Y)T_{abc} &= +\,T_{abc} \qquad\qquad + T_{bac} - T_{bca} \qquad\qquad - T_{cba} \\
P(Y)T_{bca} &= \qquad\qquad - T_{acb} \qquad\qquad + T_{bca} - T_{cab} + T_{cba} \\
P(Y)T_{cab} &= -\,T_{abc} + T_{acb} - T_{bac} \qquad\qquad + T_{cab}
\end{aligned} \tag{3.61}$$

In Cadabra, the projection of a tensor can be calculated using the `young_project_tensor` algorithm on an expression where at least one term carries the `TableauSymmetry` property. For example, the identity above can be proved as follows:

```
T_{a b c}::TableauSymmetry(shape={2,1}, indices={0,1,2}).
ex := T_{a b c} + T_{b c a} + T_{c a b};
young_project_tensor(_);
```

where the last line prints `0` as expected.

3.3.5 Hermitian Projection Operators

The projection operator obeys some interesting and slightly counter-intuitive properties:

1. **Completeness**

   As expected, the complete set of Young projectors sum up to the identity:

$$\sum_{Y \in S_n} P(Y) = I \tag{3.62}$$

2. **Linearity**

   The property of completeness does not however imply the linearity of the projection operator:

$$\sum_i P(Y_i) \neq P(\bigoplus_i Y_i) \tag{3.63}$$

   which does not in general hold [67]; in particular

$$P(Y_1 \otimes Y_2) \neq \sum_i P(Y_i') \tag{3.64}$$

   where Y_i' are the terms in the Littlewood-Richardson expansion of $Y_1 \otimes Y_2$. For example the tensor product with a single-cell tableau does not change the projection as the new cell does not contribute any symmetry

$$P(\begin{array}{|c|c|}\hline 1 & 2 \\ \hline\end{array} \otimes \begin{array}{|c|}\hline 3 \\ \hline\end{array})T_{abc} = T_{abc} + T_{bac} \tag{3.65}$$

   but summing the decompositions would lead us to

$$\begin{aligned}
\left(P\left(\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline\end{array}\right) + P\left(\begin{array}{|c|c}\hline 1 & \multicolumn{1}{c|}{2} \\ \hline 3 & \\ \cline{1-1}\end{array}\right)\right) T_{abc} = \frac{1}{6}(3T_{abc} + T_{acb} + 3T_{bac} \\
- T_{bca} + T_{cab} - T_{cba})
\end{aligned} \tag{3.66}$$

   where it is clear the terms introduced by the symmetry and antisymmetry in slot 3 have not cancelled out. A special case is

$$P(\begin{array}{|c|}\hline 1 \\ \hline\end{array} \otimes \begin{array}{|c|}\hline 2 \\ \hline\end{array})T_{ab} = P\left(\begin{array}{|c|c|}\hline 1 & 2 \\ \hline\end{array}\right) + P\left(\begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline\end{array}\right) = T_{ab} \tag{3.67}$$

which is true because of (3.62).

3. **Idempotency**
   Young operators are idempotent, that is

$$P(Y_i)^n = P(Y_i) \tag{3.68}$$

e.g.

$$\begin{aligned}
P\left(\begin{array}{|c|}\hline 1\\\hline 2\\\hline\end{array}\right)^3 T_{ab} &= \frac{1}{2^3}\mathcal{A}_{12}^3 T_{ab}\\
&= \frac{1}{8}\mathcal{A}_{12}^2(T_{ab} - T_{ba})\\
&= \frac{1}{8}\mathcal{A}_{12}(T_{ab} - T_{ba} - T_{ba} + T_{ab})\\
&= \frac{1}{8}(T_{ab} - T_{ba} - T_{ba} + T_{ab} - T_{ba} + T_{ab} + T_{ab} - T_{ba})\\
&= \frac{4}{8}(T_{ab} - T_{ba})\\
&= \frac{1}{2}\mathcal{A}_{12}T_{ab}\\
&= P\left(\begin{array}{|c|}\hline 1\\\hline 2\\\hline\end{array}\right)T_{ab}
\end{aligned} \tag{3.69}$$

4. **Orthogonality**
   Young projectors are not in general orthogonal, that is

$$P(Y_i)P(Y_j) \neq \delta_{ij} \tag{3.70}$$

although this is true for $i = j$ by idempotency, and is also generally true if Y is an irrep of S_n for $n \leq 4$. To illustrate what this means, consider the product $P\left(\begin{array}{|c|c|}\hline 1 & 2\\\hline\end{array}\right)P\left(\begin{array}{|c|}\hline 1\\\hline 2\\\hline\end{array}\right)$ which belong to S_2. The explicit form of this is $\frac{1}{4}\mathcal{S}_{12}\mathcal{A}_{12}$ which symmetrises and anti-symmetrises in the same two indices, so the projection applied to any tensor is identically zero by similar reasoning to (3.36):

$$T_{ab} = T_{ba} = -T_{ab} \tag{3.71}$$

However, in the case of

$$P\left(\begin{array}{|c|c|c|}\hline 1 & 3 & 5\\\hline 2 & 4 \\\cline{1-2}\end{array}\right)P\left(\begin{array}{|c|c|c|}\hline 1 & 2 & 3\\\hline 4 & 5 \\\cline{1-2}\end{array}\right) \tag{3.72}$$

the label $\boxed{5}$ belongs to different symmetrisation and antisymmetrisation groups in each tableaux, and so we cannot use the same reasoning to prove that this combination is zero; and indeed explicit calculation will show that there are some terms left over after applying both these projections. We can show this using the young_project_tensor function of Cadabra:

```
ex := T_{a b c d e}.
T_{a b c d e}::TableauSymmetry(shape={2,3}, indices={0,1,2,3,4}).
young_project_tensor(ex)
T_{a b c d e}::TableauSymmetry(shape={2,3}, indices={0,2,4,1,3}).
young_project_tensor(ex);
```

which produces the output

$$
\begin{aligned}
&-1/54T_{acbde}-1/54T_{bcade}-1/54T_{adbce}-1/54T_{aebdc}-1/54T_{adbec}-1/54T_{aebcd}\\
&-1/54T_{acbed}-1/54T_{bdace}-1/54T_{beadc}-1/54T_{bdaec}-1/54T_{beacd}-1/54T_{bcaed}\\
&+1/54T_{bacde}+1/54T_{cabde}+1/54T_{bdcae}+1/54T_{becda}+1/54T_{bdcea}+1/54T_{becad}\\
&+1/54T_{baced}+1/54T_{cdbae}+1/54T_{cebda}+1/54T_{cdbea}+1/54T_{cebad}+1/54T_{cabed}\\
&-1/54T_{dbcea}-1/54T_{cbdea}-1/54T_{decba}-1/54T_{daceb}-1/54T_{decab}-1/54T_{dacbe}\\
&-1/54T_{dbcae}-1/54T_{cedba}-1/54T_{cadeb}-1/54T_{cedab}-1/54T_{cadbe}-1/54T_{cbdae}\\
&+1/54T_{abdce}+1/54T_{dbace}+1/54T_{acdbe}+1/54T_{aedcb}+1/54T_{acdeb}+1/54T_{aedbc}\\
&+1/54T_{abdec}+1/54T_{dcabe}+1/54T_{deacb}+1/54T_{dcaeb}+1/54T_{deabc}+1/54T_{dbaec}
\end{aligned}
$$

5. **Commutivity**

   Tableaux which share common labels do not in general commute, however tableaux which are independent in this sense produce commuting projectors. This is a result of the constituent symmetrisation operators which the projector is made of; as tableaux which do not share any labels permute independent groups of indices and so do not affect each other. This is not true of symmetrisation operators which share labels and which rarely commute:

$$
\begin{aligned}
(\mathcal{S}_{12}\mathcal{A}_{23})T_{abc} &= -T_{cab}\\
(\mathcal{A}_{23}\mathcal{S}_{12})T_{abc} &= -T_{bca}
\end{aligned}
\tag{3.73}
$$

   This can have a huge impact on the result of the calculation, for example by swapping the order of the two non-orthogonal projectors they become orthogonal to each other:

$$
P\left(\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5\\\cline{1-2}\end{array}\right)P\left(\begin{array}{|c|c|c|}\hline 1&3&5\\\hline 2&4\\\cline{1-2}\end{array}\right)T_{abcde}=0 \tag{3.74}
$$

The fact that linearity and orthogonality do not hold in general can be, as we will observe later on when considering sums of tableaux in section 4.5.2, a hindrance to calculations and ideally we would be able to find a different way of constructing the projection operator which preserves these properties whilst also making all symmetries manifest. It turns out that it is possible to find such a construction, and it is not unique. One solution is to construct a *Hermitian projection operator* which, obeying the properties associated with Hermiticity, automatically satisfies both these properties.

An example of how to construct a Hermitian projection operator is given in the series of three papers [68, 67, 69]. The process takes a Hermitian projection operator $H(Y)$, whose form is unknown, and recursively converts it into a product of Young projection operators:

1. If the projection operator acts on an element of S_1 or S_2, rewrite it as a regular projection operator

2. Otherwise, replace it with the product $H(Y')P(Y)H(Y')$, where Y' is the *ancestor tableau* which is a sub-representation of $Y \in S_n$ which is created by removing the cell $\boxed{n}$ from Y and is a representation of S_{n-1}.

The process is repeated until only Young projection operators remain. This construction is known as the *KS construction* after its authors Keppeler and Sjödahl [70]. Let us calculate the full Hermitian operator for $Y = \begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5\\\cline{1-2}\end{array}$. We begin by writing out the unknown form of the

Hermitian operator:

$$H\left(\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5\\\cline{1-2}\end{array}\right)=H(Y) \tag{3.75}$$

As Y is a representation of S_5, we need the ancestor $Y'=\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4\\\cline{1-1}\end{array}$:

$$H\left(\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5\\\cline{1-2}\end{array}\right)=H(Y')P(Y)H(Y') \tag{3.76}$$

Again, as Y' is a representation is a representation of S_4 we need to continue to process with $Y''=\begin{array}{|c|c|c|}\hline 1&2&3\\\hline\end{array}$:

$$H\left(\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5\\\cline{1-2}\end{array}\right)=H(Y'')P(Y')H(Y'')P(Y)H(Y'')P(Y')H(Y'') \tag{3.77}$$

Y'' is a representation of S_3 so we now continue with $Y'''=\begin{array}{|c|c|}\hline 1&2\\\hline\end{array}$:

$$\begin{aligned}H\left(\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5\\\cline{1-2}\end{array}\right) = \;& H(Y''')P(Y'')H(Y''')P(Y')H(Y''')P(Y'')H(Y''')P(Y)\\ & H(Y''')P(Y'')H(Y''')P(Y')H(Y''')P(Y'')H(Y''')\end{aligned} \tag{3.78}$$

The only Hermitian projection operators we have left on the right hand side are projecting elements of S_2, so the recursion is complete by equating these with normal projection operators:

$$H\left(\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5\\\cline{1-2}\end{array}\right)=\frac{1}{509,607,936}\left(\begin{array}{c}\mathcal{S}_{12}\mathcal{S}_{123}\mathcal{S}_{12}\mathcal{S}_{123}\mathcal{A}_{14}\mathcal{S}_{12}\mathcal{S}_{123}\mathcal{S}_{12}\mathcal{S}_{123}\mathcal{S}_{45}\\ \mathcal{A}_{14}\mathcal{A}_{25}\mathcal{S}_{12}\mathcal{S}_{123}\mathcal{S}_{12}\mathcal{S}_{123}\mathcal{A}_{14}\mathcal{S}_{12}\mathcal{S}_{123}\mathcal{S}_{12}\end{array}\right) \tag{3.79}$$

Hopefully one of the major drawbacks to such a construction is clear: the resulting expression is far lengthier and more complicated than the simple Young projector construction

$$P\left(\begin{array}{|c|c|c|}\hline 1&2&3\\\hline 4&5\\\cline{1-2}\end{array}\right)=\frac{1}{24}\left(\mathcal{S}_{123}\mathcal{S}_{45}\mathcal{A}_{14}\mathcal{A}_{25}\right) \tag{3.80}$$

For this reason, unless the properties are necessary, it is preferable to not use an alternate construction. There are many simplifications which are possible, for example terms such as $\mathcal{S}_{123}\mathcal{S}_{12}$ which symmetrise in the same indices multiple times can be reduced to e.g. $\mathcal{S}_{123}$, however even after simplification the operators are not generally as simple.

As an extension to the current suite of tools in Cadabra for handling tableaux, we have create a new package `cdb.utils.tableau` which contains among other tools the algorithm `create_hermitian_tableau` which can calculate the decomposition of a Hermitian projector into Young projectors. This makes it easy to calculate the Hermitian projection operator with a function such as the following:

```
from cdb.utils.tableau import create_hermitian_tableau, tableau_to_ex
def young_project_hermitian(ex, tab):
    herm = create_hermitian_tableau(tab)
    ident := @(ex).
    for t in reversed(herm):
        TableauSymmetry(ident, tableaux_to_ex(t))
        young_project_tensor(ex)
    return ex
```

Chapter 4

The `meld` Algorithm

The implementation of a canonicalisation routine using the Young projection technique was a primary focus of extending the capabilities of Cadabra and we have developed a concrete implementation of this routine under the name `meld`, which is intended to describe in this context the action of finding combinations of objects in an expression which minimise the overall number of terms, in analogy to how in card games such as rummy the objective is to collect cards into sequences known as melds with the objective of minimising the number of cards not inside a meld.

This is in contrast to conventional canonicalisation algorithms which seek to rewrite each object with indices in a maximally sorted order and then allow combining of terms by collecting identical objects. In this sense, calling `meld` a canonicalisation routine is misleading, the distinction being that the e.g. Butler-Portugal algorithm attempts to canonicalise with respect to an index ordering, whilst `meld` attempts to canonicalise with respect to a set of independent representations of an object. At various points during the discussion where comparison between the two techniques is required, an implementation of the Butler-Portugal algorithm which was already available in Cadabra using the xPerm [71] library under the name `canonicalise` will be used.

The algorithm is also described in our paper *Hiding canonicalisation in tensor computer algebra* [72] which was written with the dual aims of proposing the advantages of this alternative approach to canonicalisation as well as describing the concrete implementation of the algorithm in `meld`. The following discussion will provide a much more detailed examination of the internals of the algorithm and justify many of the design choices which were taken when considering data structures and specific sub-algorithms.

4.1 Description of the algorithm

At a high level view, the algorithm loops through each term in a sum and for each term calculates the Young projection, then attempts to write this projection as a linear combination of the terms which it has previously seen and projected; if such a combination is found then the term is eliminated from the expression and the coefficients of the other terms altered by the weight each contributed to the projection of the eliminated term. For instance, consider the equation

$$T_{abc} + T_{acb} + T_{cab} \tag{4.1}$$

where the symmetry of T_{abc} is described by $\begin{array}{|c|c|}\hline 0 & 1 \\ \hline 2 \\ \cline{1-1}\end{array}$.

The `meld` algorithm begins by considering the first term T_{abc} and calculating the Young projection, which it stores in a table with the weight of each term:

Term Component	T_{abc}
T_{abc}	$\frac{1}{3}$
T_{acb}	0
T_{bac}	$\frac{1}{3}$
T_{bca}	-$\frac{1}{3}$
T_{cab}	0
T_{cba}	-$\frac{1}{3}$

The projection is non-zero, and there are no previous terms, so for this term alone there is no possible simplification and it considers the second term:

Term Component	T_{abc}	T_{acb}
T_{abc}	$\frac{1}{3}$	0
T_{acb}	0	$\frac{1}{3}$
T_{bac}	$\frac{1}{3}$	0
T_{bca}	$-\frac{1}{3}$	$-\frac{1}{3}$
T_{cab}	0	$\frac{1}{3}$
T_{cba}	$-\frac{1}{3}$	$-\frac{1}{3}$

The projection is non-zero, and there is one previous term so it attempts to solve the set of linear equations given by matching the coefficients of each term in $c_0 P(T_{abc}) = P(T_{acb})$:

$$\begin{aligned} c_0(\frac{1}{3}) &= 0 \\ c_0(0) &= \frac{1}{3} \\ c_0(\frac{1}{3}) &= 0 \\ c_0(-\frac{1}{3}) &= -\frac{1}{3} \\ c_0(0) &= \frac{1}{3} \\ c_0(-\frac{1}{3}) &= -\frac{1}{3} \end{aligned} \tag{4.2}$$

The system is very over-constrained and there is no solution for c_0 so the algorithm continues on to the next term:

Term Component	T_{abc}	T_{acb}	T_{cab}
T_{abc}	$\frac{1}{3}$	0	$-\frac{1}{3}$
T_{acb}	0	$\frac{1}{3}$	$\frac{1}{3}$
T_{bac}	$\frac{1}{3}$	0	$-\frac{1}{3}$
T_{bca}	$-\frac{1}{3}$	$-\frac{1}{3}$	0
T_{cab}	0	$\frac{1}{3}$	$\frac{1}{3}$
T_{cba}	$-\frac{1}{3}$	$-\frac{1}{3}$	0

As this projection is also non-zero, it attempts to find the coefficients in the equation $c_0P(T_{abc}) + c_1P(T_{acb}) = P(T_{cab})$:

$$\begin{aligned} c_0(\frac{1}{3}) + c_1(0) &= -\frac{1}{3} \\ c_0(0) + c_1(\frac{1}{3}) &= \frac{1}{3} \\ c_0(\frac{1}{3}) + c_1(0) &= -\frac{1}{3} \\ c_0(-\frac{1}{3}) + c_1(-\frac{1}{3}) &= 0 \\ c_0(0) + c_1(\frac{1}{3}) &= \frac{1}{3} \\ c_0(-\frac{1}{3}) + c_1(-\frac{1}{3}) &= 0 \end{aligned} \tag{4.3}$$

This has the unique solution set $(c_0 = -1, c_1 = 1)$ which can be determined from the first two equations and then verified by substitution in the remaining four, and so the contributions from T_{cab} can be subsumed into the terms T_{abc} and T_{acb} by changing their coefficients:

$$\begin{aligned} &T_{abc} \to T_{abc}(1 + c_0) = 0 \\ &T_{acb} \to T_{acb}(1 + c_1) = 2T_{acb} \\ &T_{cab} \to 0 \end{aligned} \tag{4.4}$$

There are no more terms in the expression and so the algorithm terminates returning $2T_{acb}$, the only remaining non-zero term, as its output.

4.2 Architecture

meld is implemented as a C++ algorithm inside Cadabra. Algorithms in Cadabra are implemented as virtual classes which derive from the base Algorithm class and provide two methods: can_apply which examines a tree node and returns whether the algorithm can be applied at this position in the tree, and apply which applies the algorithm at this point. The base Algorithm class defines a apply_pre_order and a apply_generic method which iterate through an entire expression tree, in pre- and post- order respectively, calling can_apply and conditionally apply at each node in the tree. In the Python bindings, the class structure is abstracted away so that the algorithm can be called as a function. Knowing this structure is sufficient for the following discussion, however a more rigorous introduction to the design of algorithms in Cadabra is available in a previous paper [73].

The implementation of meld is split across two files, the header core/algorithms/meld.hh[1] and the implementation core/algorithms/meld.cc[2]. These expose the meld class (which translates into the meld function in Python) which implements the algorithm described in this chapter. Two other implementation files, core/Adjform.hh[3] and core/Adjform.cc[4] provide data structures used by meld and will be referred to. All references to code made in this chapter will use the versions of these files linked to here, as future versions of Cadabra may modify or move these files.

On top of the main canonicalisation routine which is discussed here, there are other related

[1] https://github.com/kpeeters/cadabra2/blob/b03150540db47ccb7536a12022c4a13c775525ac/core/algorithms/meld.hh

[2] https://github.com/kpeeters/cadabra2/blob/b03150540db47ccb7536a12022c4a13c775525ac/core/algorithms/meld.cc

[3] https://github.com/kpeeters/cadabra2/blob/b03150540db47ccb7536a12022c4a13c775525ac/core/Adjform.hh

[4] https://github.com/kpeeters/cadabra2/blob/b03150540db47ccb7536a12022c4a13c775525ac/core/Adjform.cc

simplification routines implemented in `meld` which will be discussed at the end of the chapter. The implementation of all these routines is contained within the `meld` class and divided into separate `can_apply_*` and `apply_*` routines, with the canonicalisation through Young projection marked by the `tableaux` suffix. There is additionally an auxiliary class `ProjectedTerm` defined within `meld`'s class namespace which is specific to the `tableaux` routines.

4.3 Data structures

4.3.1 Dummy independent representations of tensors

One important consideration which was absent in the above example of `meld` is the handling of dummy indices. Unlike implementations in mono-term canonicalisers which treat dummy index symmetries as a separate group of symmetries to be applied as minimising the lexical order is important, in the Young projection algorithm it is enough to remove any possible breaking of the manifest symmetry in the projected expressions due to the ambiguity in the naming of the dummy indices, otherwise even simple expressions such as

$$S_{aa} + S_{bb} \tag{4.5}$$

would never be treated as equal. A naïve approach would be to rename the dummy indices before the projection using an algorithm such as Cadabra's `rename_dummies`. This works for the expression above, but more complicated expressions where the dummy indices are crossed between factors in a product might still result in identical projections up to dummy index namings, for instance the identity

$$R_{abcd}R_{abcd} = 2R_{abcd}R_{acbd} \tag{4.6}$$

which is a result of a Bianchi identity is displayed here with all dummy indices canonicalised. However, the two projections contain no identical terms, with the symmetry only becoming manifest after all terms in the projection are rearranged so that their first factor is R_{abcd}. Indeed, the symmetry can always be restored if the renaming is performed after the projection but in large projections which can contain millions of terms this is an expensive solution.

In order to avoid the problem altogether, the `meld` algorithm instead stores expressions in a way which is independent of dummy index names. This representation is implemented by the `Adjform` class which, as the name suggests, stores connections between index slots in the tensor rather than identifying that indices are contracted by giving them identical labels, in the same way that an adjacency list represents a graph by storing the connections between its nodes. Consider the expression (4.5) above. In the first term S_{aa} the contraction implies that the values in slots 0 and 1 of the tensor are identical. As the exact name used is unimportant, the adjacency representation simply records that the index in slot 0 'points' to the index in slot 1 and vice versa. Representing these by integer labels, we might rewrite the term in adjacency notation as S_{10}. As free indices do carry significant information they would be left unchanged. As the second term has an identical index structure, differing only in the names of the dummy indices, it is identical to the first term. The power of this representation is that we can therefore automatically detect the equivalence between these terms from the notation, without having to apply any further routines.

In the implementation, the `Adjform` class only encodes the index structure of an object, and so for the expression above internally only holds an array of the two integers, $[1, 0]$. In order to increase efficiency, index names for free indices are mapped to negative integers (with the mapping kept track of by the `IndexMap` class) so that the class only needs to hold an array of integers.

One cost of storing the indices in this manner is that algorithms on the adjacency representation are more complicated than on regular index notation, for instance swapping the positions of dummy indices in two different sets requires updating the values of the elements they point to:

$$\begin{aligned} T_{abab} &\to T_{abba} \\ T_{2301} &\to T_{3210} \end{aligned} \tag{4.7}$$

This requires a conditional move for each index in the swap. On modern x86 architectures this is optimised into two independent `cmov` instructions so in reality the extra overhead of this complication is generally minimal. Of course, swapping a dummy index with its partner results in the same adjform, just as an object written in dummy notation does, which can be optimised as an early check.

4.3.2 Representation of a projected monomial

The data structure used to represent terms in the projection is based on the `Adjform` class rather than the `Ex` class which Cadabra uses to store generic expressions. In order for this to be possible, the algorithm must first split an expression into different sets of terms each of which has the same tensorial structure so that each set can be melded independently as objects are generally not distinguishable from their index structure alone (e.g. $A_{ij} \neq B_{ij}$). As the symmetries which `meld` can canonicalise only relate objects to themselves this can be done with no loss of generality. Not only does this reduce the complexity of the equations to be solved when calculating linear dependence between the terms, but it also allows the algorithm to make strong assumptions about the structure of the indices in each term which reduces the overhead associated with storing each tensor. In order to create the sets of objects, each term must be transformed by commuting terms and stripping scalar factors until the minimal amount required to fully represent the index structure of the object remains which can be compared against other terms. For instance, consider the following collection of expressions:

$$\begin{aligned} &\text{(a) } 2A_{\mu\nu}S_{\rho\lambda} \\ &\text{(b) } A_{\mu\nu}kS_{\rho\lambda} \\ &\text{(c) } R_{\mu\nu\rho\lambda} \\ &\text{(d) } A_{\mu\gamma}B_{\gamma\nu}S_{\rho\lambda} \end{aligned} \tag{4.8}$$

Clearly (a) and (b) should be able to be combined, assuming k is an object which can be commuted through $A_{\mu\nu}$. However, simple techniques such as index counting would also match against (c) which consists of a different tensor and is therefore incompatible, or checking whether expressions share a common tensor would match against (d) whose index structure is incompatible. Sorting an expression is also not a general solution, even in this instance lexical ordering might place k between $A_{\mu\nu}$ and $S_{\rho\lambda}$.

`meld` solves this problem by separating the factors making up each term into two separate expressions consisting of the scalar parts of the expression and the indexed parts, which is performed on the initial pass through the expression. The logic is contained inside the constructor of the `ProjectedTerm` class, which is an auxiliary data structure for holding information about each term:

```
struct ProjectedTerm {
    Ex scalar, tensor; // Scalar and tensor parts of the term
    ProjectedAdjform projection; // Container as described above
    Adjform ident; // Representation of the term as an adjform
    Ex::iterator it; // Pointer to the term in the original expression
    bool changed; // Flag to set when this terms contains contributions from
        another term
};
```

The class is a convenient way of encapsulating all the information required about a particular term to avoid having to cross-reference between different data structures. The algorithm begins by creating two new expression trees with empty `\prod` nodes at the top which are assigned to the `scalar` and `tensor` objects in the `ProjectedTerm` object. Then, the individual factors making up the term are iterated over. If the term has indices then it is attached as a new child of the `tensor` expressions. If it has no indices, then before attaching it to the `scalar` expression it is first checked to make sure that it is able to commute through any tensors to its left. This ensures that the structure of the object isn't altered in a way disallowed by the commutation relations of the scalar factors. If the commutation is not possible, then the factor is considered part of the structure of the tensor and appended to the `tensor` node.

Comparing two terms in an expression to check if they have matching tensorial structures is implemented in the `compare` method of the `ProjectedTerm` class. Two objects can only be melded if the `tensor` expression trees are identical up to some allowed differences in indices:

- The indices are in the same index set and are both covariant/contravariant
- The indices are in the same index set and are both 'free' (i.e. covariance/contravariance has no meaning for the index)
- They are dummy indices in the same index set, and they are either
    1. Both covariant/contravariant
    2. Not separated by a derivative

These criteria may be over restrictive, preventing some objects with compatible structures from being compared, however it is felt better to be overly strict and avoid 'false positives'. The structure is checked by simultaneously iterating over both expression trees comparing each node in iteration order. Any mismatching nodes, or one iterator expiring before the other, results in the two trees being considered not equal.

A second consideration is how to store the `Adjform` objects which make up the full projection, which `meld` encapsulates in the `ProjectedAdjform` class. The data structure needs to allow for the following requirements:

- Each `Adjform` object must be associated with a numeric factor corresponding to its contribution to the overall projection
- The objects should be stored in a sorted order so that comparing the coefficients of multiple projections can be done in linear time
- Accessing and inserting elements should be made as efficient as possible, in particular avoiding structures which require linear time to find elements and involve shifting elements during insertions

Two natural choices of container emerge from these criteria: a preallocated array or a binary search tree. A linked list is unsuitable due to linear element access, and a hash table does not guarantee that the order of elements will be the same between containers.

For the purposes of this algorithm a binary search tree has been used to provide the best performance, however it is not immediately clear why this is optimal and in order to form a conclusion a discussion of the internals of both need to be explored which forms the content of the rest of this section.

4.3.3 Contiguous storage solution

In theory, the most efficient storage mechanism is an array preallocated to be able to store all elements in the projection. The preallocation is necessary for two reasons: not only to avoid memory reallocations as the projection is calculated, but also as access to elements near the end of the storage representing later permutations may be required early on in the projection necessitating the other values be present so that the permutation can be identified by its index in the array. This is therefore a dense representation where permutations which do not exist in the projection must be explicitly given the value 0.

The number of terms in the projection of a particular monomial is dependent on the structure of the symmetry as well as the structure of the dummy indices inside the expression. For a given tableau with row lengths given by $Y = \{r_1, r_2, \dots, r_n\}$, the total number of terms in the expression generated by the projection operator acting on a tensor containing only free indices, $N(Y)$, is given by the product of the number of permutations generated by each row and column:

$$N(Y) = \prod_{j=1}^{n}(r_j!\,)\prod_{j=1}^{m}(c_j!\,) \tag{4.9}$$

where c_j are the lengths of the columns of Y, and the value of c_j can be calculated as the number of rows of Y with at least j cells:

$$c_j = \sum_{i=1}^{n}\Theta(r_i - j) \qquad \Theta(x) = \left\{\begin{array}{ll} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{array}\right. \tag{4.10}$$

However, although this will calculate the number of non-zero permutations in the resulting projection, merely allocating space for this number of terms is insufficient as tensors with different starting permutations of indices will have different non-zero permutations in the output and thus space for all $n!$ possible permutations must be allocated.

For a monomial containing dummy indices, the number of terms in the projection is reduced due to some permutations being identical. The way in which this happens is non trivial, and the number of terms can even vary depending on whether the projection is calculated using the symmetrisers or antisymmetrisers first, as is shown here where the projection (up to an overall factor) of the Riemann monomial R_{abad} is explicitly calculated using `symmetrise_slots` which performs a symmetrisation of all the terms in an expression over the given index slots:

```
from cdb.utils.indices import symmetrise_slots
ex := R_{a b a d}:
symmetrise_slots(ex, [0,1], antisym=True)
symmetrise_slots(ex, [2,3], antisym=True)
symmetrise_slots(ex, [0,2])
symmetrise_slots(ex, [1,3]);
```

$$2R_{abad} + 2R_{adab} - R_{abda} - R_{aadb} - R_{dbaa} - R_{daab} - R_{baad} - R_{bdaa} - R_{aabd} - R_{adba}$$
$$+ 2R_{bada} + 2R_{daba}$$

```
ex := R_{a b a d};
symmetrise_slots(ex, [0,2])
symmetrise_slots(ex, [1,3])
symmetrise_slots(ex, [0,1], antisym=True)
symmetrise_slots(ex, [2,3], antisym=True);
```

$$2R_{abad} - 2R_{abda} - 2R_{baad} + 2R_{bada} + 2R_{adab} - 2R_{adba} - 2R_{daab} + 2R_{daba}$$

Two expressions which contain the same number of dummy indices but in different locations will also lead to different sets of non-zero terms. The maximum number of terms in a projection involving n indices with d dummy pairs is given by all possible permutation of n distinguishable objects divided by both the symmetry of interchanging dummy names, i.e. all permutations of dummy labels $d!$, and the further redundancy of swapping the two indices in the dummy pair (2^d). This means that in general space for

$$\frac{n!}{d!\,2^d} \tag{4.11}$$

terms must be allocated.

Another problem with a contiguous storage system is finding the element corresponding to a particular permutation by its index in the array. One possible solution is to store the `Adjform` representation along with each element and perform a binary search of the array to find the correct element, however the amount of extra storage required is undesirable, and the solution is $O(\log(N))$ complexity, where N is the total number of permutations, whereas a contiguous storage system allows for $O(1)$ access. Another solution is to construct an algorithm for mapping each possible permutation to an integer i in the range $0 \leq i < N$. In the case of unique permutations where $N = n!$, this problem is well known and has many solutions starting with Lehmer in 1960 [74] and including recent additions to the literature such as the variable length encoding published in 2021 [75].

The basic idea of the Lehmer code is to represent a sequence as a number written in a factorial base, that is the digit d in position i represents $d \times i!$, in the same way that in base-10 it would represent $d \times 10^i$. The digit in the ith position is the number of objects which compare less than it in the tail of the sequence. For example, to construct the Lehmer code of the sequence $[0, 2, 3, 1]$ the following steps would be taken:

Sequence				Lehmer code				Explanation
4	3	2	1	3!	2!	1!	0!	
0	2	3	1	**0**				0 numbers to the right of 0 less than it
0	**2**	3	1	0	**1**			1 number to the right of 2 less than it
0	2	**3**	1	0	1	**1**		1 number to the right of 3 less than it
0	2	3	**1**	0	1	1	**0**	0 numbers to the right of 1 less than it

Thus as an integer the Lehmer code is $0 \times 3! + 1 \times 2! + 1 \times 1! + 0 \times 0! = 3$. A construction in this manner also happens to give the position of the permutations in the total set of lexicographically ordered permutations:

Lehmer Index	Sequence	Lehmer Index	Sequence	Lehmer Index	Sequence
0	$[0,1,2,3]$	8	$[1,2,0,3]$	16	$[2,3,0,1]$
1	$[0,1,3,2]$	9	$[1,2,3,0]$	17	$[2,3,1,0]$
2	$[0,2,1,3]$	10	$[1,3,0,2]$	18	$[3,0,1,2]$
3	$[\mathbf{0},\mathbf{2},\mathbf{3},\mathbf{1}]$	11	$[1,3,2,0]$	19	$[3,0,2,1]$
4	$[0,3,1,2]$	12	$[2,0,1,3]$	20	$[3,1,0,2]$
5	$[0,3,2,1]$	13	$[2,0,3,1]$	21	$[3,1,2,0]$
6	$[1,0,2,3]$	14	$[2,1,0,3]$	22	$[3,2,0,1]$
7	$[1,0,3,2]$	15	$[2,1,3,0]$	23	$[3,2,1,0]$

When a permutation can include duplicate elements specific mentions in scientific literature are sparse, in particular for the our case where the naming of duplicate elements is irrelevant, however the topic is occasionally mentioned on mathematics forums[5][6]. It is of course still possible to apply the classic Lehmer algorithm, however permutations will still be mapped onto the range $0 \leq i < n!$ with some gaps which do not correspond to any sequence

Lehmer Index	Sequence	Lehmer Index	Sequence
0	$[0,1,1,3]$	9	$[1,1,3,0]$
1	$[0,1,3,1]$	10	$[1,3,0,1]$
4	$[0,3,1,1]$	11	$[1,3,1,0]$
6	$[1,0,1,3]$	18	$[3,0,1,1]$
7	$[1,0,3,1]$	20	$[3,1,0,1]$
8	$[1,1,0,3]$	21	$[3,1,1,0]$

The aim of course is to minimise the amount of storage space required by the algorithm, and so allocating space for $n!$ elements when far fewer are actually required is a suboptimal solution. We will now show that an algorithm to map elements onto the exact range required is achievable; first we will concentrate on a variation of the Lehmer code algorithm adapted for the case when there may be repeated elements and then adapt this so that the naming of the repeated elements is made irrelevant. The number of permutations of n elements where there are d pairs of identical elements is given by

$$P_{\text{dis}}(n,d) = \frac{n!}{2^d} \tag{4.12}$$

where the label "dis" indicates that the pairs are distinguishable from each other. This value is just the total number of permutations of n objects divided through by the symmetry under exchange of the slots corresponding to dummy indices which halves the number of unique permutations for each dummy pair. The concept of the algorithm is to iterate through the permutation and for each index i to calculate w_i, the total number of permutations which can occur before it. The Lehmer code for the sequence is then given by $\sum_i w_i$. For example given the permutation $[2,3,1,2]$:

1. For the first index to be 2, it must come after all permutations with first index 1. If the first index is 1, then the remaining elements are $[2,2,3]$ of which there are $P_{\text{dis}}(3,1) = 3$ permutations, so $w_0 = 3$.

[5] https://math.stackexchange.com/questions/3802978/permutation-with-repetition-index-conversion

[6] https://math.stackexchange.com/questions/879429/lexicographical-rank-of-a-string-with-duplicate-characters

2. Remove the first index and consider the tail $[3, 1, 2]$. For the index 3 to come first, it must come after all permutations starting with 1 or 2 so $w_1 = P(2, 0) + P(2, 0) = 4$.

3. Again look at the tail $[1, 2]$, but as this is already sorted no permutations come before it so $w_2 = 0$.

The final index is thus given by $3 + 4 + 0 = 7$, which again is the position of the permutation in the lexicographic ordering of permutations of this sequence:

Index	Sequence	Index	Sequence
0	$1, 2, 2, 3$	6	$2, 2, 3, 1$
1	$1, 2, 3, 2$	**7**	$\mathbf{2, 3, 1, 2}$
2	$1, 3, 2, 2$	8	$2, 3, 2, 1$
3	$2, 1, 2, 3$	9	$3, 1, 2, 2$
4	$2, 1, 3, 2$	10	$3, 2, 1, 2$
5	$2, 2, 1, 3$	11	$3, 2, 2, 1$

This is an $O(n)$ algorithm in the number of indices, and is therefore constant over the number of terms in the projection. Applying this algorithm to a sequence where names of repeated elements are irrelevant leaves some redundancy as can be seen be considering the unique permutations of the sequence $[a, a, b, b]$:

Index	Permutation	Adj. rep.
0	a, a, b, b	$[1, 0, 3, 2]$
1	a, b, a, b	$[2, 3, 0, 1]$
2	a, b, b, a	$[3, 2, 1, 0]$
3	b, a, a, b	$[3, 2, 1, 0]$
4	b, a, b, a	$[2, 3, 0, 1]$
5	b, b, a, a	$[1, 0, 3, 2]$

As can be seen from the adjacency representations on the right only three of these six permutations are unique. The interchangeability of dummy labels as a symmetry means that all permutations of dummy index names are equivalent, so the number of unique permutations is reduced by a factor of $d!$

$$P_{\text{indis}}(n, d) = \frac{n!}{d!\, 2^d} \tag{4.13}$$

which should be familiar from (4.11). The number of unique permutations of a sequence consisting purely of d dummy index pairs is a special case of this formula:

$$\begin{aligned} P_{\text{indis}}(2d, d) = \frac{(2d)!}{d!\, 2^d} &= \frac{1}{2^d} \frac{(2d) \times (2d-1) \times \ldots \times 1}{d \times (d-1) \times \ldots \times 1} \\ &= \frac{\cancel{(2d)}\,(2d-1)\,\cancel{(2d-2)}\,(2d-3)\,\cancel{(2d-4)}\ldots\,\cancel{(2)}\,(1)}{2\cancel{(d)} \quad \times \quad 2\cancel{(d-1)} \quad \times \quad 2\cancel{(d-2)}\ldots 2\cancel{(1)}} \\ &= (2d-1)(2d-3)\ldots 1 \\ &= (2d-1)!! \end{aligned} \tag{4.14}$$

where the second line is true as there are the same number of powers of 2 as terms in the factorial. $n!!$ is known as the double factorial[7] and we can see that $d = 2 \to 3!! = 3$ which agrees with the explicit calculation of the unique permutations in the table.

[7] `https://mathworld.wolfram.com/DoubleFactorial.html`

In `Adjform.cc`, a novel variation on the Lehmer code algorithm which maps an `Adjform` object onto the range $0 < i < P_{\text{indis}}$ is implemented. The algorithm calculates the index by calculating two separate values, i_{perm} which is an index in the range $0 \leq i_{\text{perm}} < n!\,/((2d)!\,)$ of the permutation where all dummy indices are given the same label, and i_{dummy} in the range $0 \leq i_{\text{dummy}} < (2d-1)!\,!$ which is the index of the permutation of the dummy indices. The final index is then calculated as $i = ((2d-1)!\,!\,i_{\text{perm}}) + i_{\text{dummy}}$ which has a maximum value of

$$\begin{aligned}(2d-1)!\,!\,(\overbrace{\frac{n!}{(2d)!}-1}^{\max(i_{\text{perm}})}) + \overbrace{((2d-1)!\,!-1)}^{\max(i_{\text{dummy}})} &= (2d-1)!\,!\,\frac{n!}{(2d)!}-1 \\ &= \frac{(2d)!}{d!\,2^d}\frac{n!}{(2d)!}-1 \\ &= \frac{n!}{d!\,2^d}-1\end{aligned} \tag{4.15}$$

as desired.

The algorithm begins by constructing two auxiliary arrays, `perm` which stores the permutation with all dummy indices relabelled to 0 and `counts` which is a mapping from a label to the number of times it is repeated in `perm`. The entries in `count` can be greater than one for free indices if the index name matters, e.g. for a index labelling a specific coordinate.

During this process, the index i_{dummy} is calculated. It begins with an initial value of 0, and similarly to the algorithm described above for every dummy index D encountered the number of preceding permutations is added, where the number of preceding permutations is given by the total number of permutations of dummy indices excluding the pair to which D belongs multiplied by number of dummy indices separating the pair. The logic here can be thought of as the other dummy index in the pair belonging to D being treated as a free index and all dummy indices having a lexicographic ordering greater than the free index. Therefore, every time it is moved one index further from D the index is increased by the number of permutations of remaining dummy indices. A small example helps to illustrate the idea: consider $[a, b, a, c, c, b] = [2, 5, 0, 4, 3, 1]$:

1. The first index is 2, and after stripping this index off the arrangement is $[4, f, 3, 2, 0]$ where f represents the now free index and for consistency the values of each dummy index have been decremented by one so they point to valid locations in the new list. As all free indices order less than dummy indices, the smallest possible permutation would be $[f, 4, 3, 2, 1]$ and so we must traverse all permutations of the dummy indices to arrive at $[4, f, 3, 2, 0]$. The number of permutations is given by $(2 \times 2 - 1)!\,! = 3$, so we set $i_{\text{dummy}} = 1 \times 3$. We also remove the free index f from the list as its position is now forced

2. We strip off the first index again to get $[1, 0, f]$. Here, as f is in the third position, we must count through two full sets of permutations of the remaining dummy indices. There is only one way to arrange a single dummy index pair so we set $i_{\text{dummy}} \to i_{\text{dummy}} + 2 \times 1 = 5$

3. We are left with a single dummy pair whose position is forced, so the final value is $i_{\text{dummy}} = 5$

We can again confirm this by looking at a table of all 15 possible orderings:

Index	Sequence	Index	Sequence
0	$[1, 0, 3, 2, 5, 4]$	8	$[3, 5, 4, 0, 2, 1]$
1	$[1, 0, 4, 5, 2, 3]$	9	$[4, 2, 1, 5, 0, 3]$
2	$[1, 0, 5, 4, 3, 2]$	10	$[4, 3, 5, 1, 0, 2]$
3	$[2, 3, 0, 1, 5, 4]$	11	$[4, 5, 3, 2, 0, 1]$
4	$[2, 4, 0, 5, 1, 3]$	12	$[5, 2, 1, 4, 3, 0]$
5	$[\mathbf{2}, \mathbf{5}, \mathbf{0}, \mathbf{4}, \mathbf{3}, \mathbf{1}]$	13	$[5, 3, 4, 1, 2, 0]$
6	$[3, 2, 1, 0, 5, 4]$	14	$[5, 4, 3, 2, 1, 0]$
7	$[3, 4, 5, 0, 1, 2]$		

Before the second part of the algorithm, a normalisation of the values in `perm` is performed: free indices are represented by negative integers in the `Adjform` class, but it is preferable for the algorithm to work on a consecutive range of integers starting at 0. Dummy indices are already replaced with the label 0, so free indices f are given a label $-f$, and then packed by checking the `counts` array for entries that are zero, removing these and shifting the labels.

In the second half of the algorithm, the `perm` array is traversed to calculate the value of i_{perm} which again is initially given the value 0. For each element i in `perm`, the number of permutations which appear before it is calculated using the modified Lehmer algorithm for distinguishable repeated indices above. The number of preceding permutations is calculated by iterating over all indices less than the current element, decreasing its value in the `count` array by one and then calculating

$$\sum_{j<i}\left[\frac{((\sum_k \texttt{count}[k])-1)!}{\sum_k\left((\texttt{count}[k]-\delta_{jk})!\right)}\right] \tag{4.16}$$

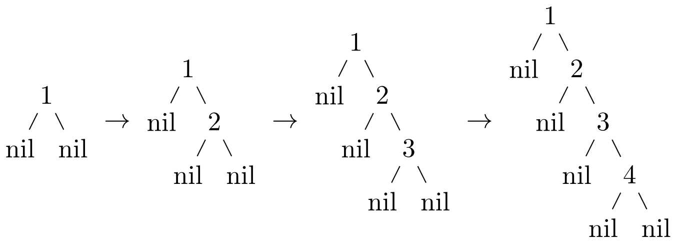
i.e. for each index less than i, its count is decremented and the number of permutations of the sequence now represented by `count` is added to i_{perm}.

We now have the ability to create a minimal dense representation of the projections of a `Adjform` object with constant time read and write access — the caveat which comes with this constant access is that the conversion from a `Adjform` to an index is very expensive as the algorithm must process many instructions and requires the use of auxiliary storage.

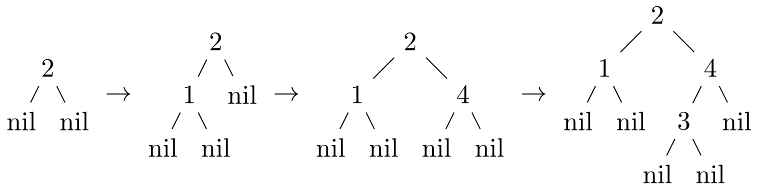
4.3.4 Binary search tree storage solution

The other viable solution is to store the projection in a binary search tree. In this format, instead of a contiguous block of memory each entry is stored as either a left or right child of another entry, depending on whether it is compares less than or not less than the parent entry. To maintain the binary structure of the tree, any nodes which do not have a left or right child point to a 'nil' value instead. Thus, in order to find an element in the tree the search starts with a *root* element and iterates through the children of this node choosing at each stage whether to take the left or right branch allowing entire branches of the tree to be eliminated from the search at each step. The worst case behaviour for tree operations is $O(h)$, where h is the height of the tree.

The order of insertion of elements can affect how high the tree is and therefore how efficiently its elements can be accessed: compare how a tree is constructed from the sequence $[1, 2, 3, 4]$



and the sequence $[2, 1, 4, 3]$



The worst case, as demonstrated here, is when elements are inserted already sorted which results in a tree with the same height as number of children, and thus element access becomes linear.

In order to resolve this issue, most implementation of binary search trees nowadays implement *balancing* trees, which ensure that when elements are inserted and deleted from the tree both sides of the tree remain approximately the same size. One of the most popular implementations of this is known as a *red-black* tree, first described in 1978 [76] using the ideas developed by Rudolf Bayer in 1972 [77]. The construction colours each node either 'red' or 'black' subject to the following constraints:

- All nil values are black
- All children of red nodes are black
- The number of black nodes passed when travelling from a node to any of its nil descendants is the same

By ensuring these are met every time a node is inserted or deleted from the tree, the balanced nature can be assured ensuring that the height, and therefore search complexity, of a tree with N elements is bounded by $\log_2(N)$. Rebalancing the tree increases the cost of many insertions and deletions, however these operations are not expensive and recolourings in particular on average can run in constant time [78].

The worst case element access behaviour for a tree storing all permutations of n indices can be approximated using a first order expansion of Stirling's approximation [79]:

$$\ln(n!\,) = n(\ln n - 1) \tag{4.17}$$

which gives us

$$\log_2(N) = log_2(n!\,) = \log_2(e)\ln(n!\,) \approx 1.44n(\ln n - 1) \tag{4.18}$$

which for scale is 12 comparisons at 8 indices and 25 comparisons at 12 indices. These sizes are of course only reached if considering totally symmetric objects with no dummy indices, and on average the search will not have to perform this many operations as the value may not be stored at the lowest depth of the tree. Red-black tree searching algorithms are also highly optimised with very low overheads, and so for indices around these values the binary search may be a more efficient way of finding an element than computing the Lehmer code as described above.

The major advantage that this representation has is of course that it permits a sparse representation of the projection, with elements that are zero omitted instead of explicitly stored. The compromise here is that extra information must be stored against each element to allow it to be found. This can either be the Adjform object itself, or using the algorithm from the previous section a single integer can be attached to each element. The elements of a `Adjform` object are stored in single bytes, the numeric coefficient (as will be discussed at the end of section 4.4.1) by a 4 byte integer type and the per-element overhead of a map type which, while dependent on the implementation, is around 32 bytes[8] , so the ratio between S_{sparse} and S_{dense}, the per-node size of elements representing a permutation of n indices is given by

$$\frac{S_{\text{sparse}}}{S_{\text{dense}}} \approx \frac{4+32+n}{4} \tag{4.19}$$

In order for the space required to be smaller in the sparse representation, the number of elements stored must be greater than this factor:

$$S_{\text{sparse}}N_{\text{sparse}} < S_{\text{dense}}N_{\text{dense}} \tag{4.20}$$

For permutations of only free indices where $N_{\text{sparse}} = N(Y)$ and $N_{\text{dense}} = n!$ this places the constraint:

$$N(Y) < \frac{4n!}{36+n} \tag{4.21}$$

The value of N(Y) is of course dependent on the shape of the tableau, but it is possible to gain an appreciation of the magnitude of this number by looking at square tableaux. For a perfect square n, the tableau consists of $\sqrt{n}$ rows each with $\sqrt{n}$ cells and similarly for the columns. Using these values, (4.9) then becomes

$$N(Y) = \prod_{j=1}^{\sqrt{n}} (\sqrt{n}!\,) \prod_{j=1}^{\sqrt{n}} (\sqrt{n}!\,) = (\sqrt{n}!\,)^{2\sqrt{n}} \tag{4.22}$$

For values where n is not a perfect square, we can interpolate this function using the Gamma function to give an approximation for a non trivial tableau with n cells:

$$N_{\text{approx}}(n) = \Gamma(1+\sqrt{n})^{2\sqrt{n}} \tag{4.23}$$

While this is an arbitrary choice of shape to use as a representative tableau, it generally produces a tableau whose complexity is representative of tableaux of mixed symmetries, and their projection will help to illustrate how tableau shape can alter the complexity of the algorithm. Using this as a value for $N(Y)$, the two sides of (4.21) are displayed in Fig. 4.1 which shows that the crossover point where it uses less memory to store the projections in a binary tree is around $n = 10$ indices.

The choice that has been made in implementing `meld` is to use a binary search tree. The choice is made based on the following observations:

1. In most real world applications, the number of terms in the projection is reasonable enough that the $O(log(N))$ accesses will often outperform the overhead of the constant lookup for an array storage solution

2. In the more esoteric cases where the number of terms is very large, the sparse storage

[8] See for example the definition of `_Rb_tree_node_base` in gcc: `https://github.com/gcc-mirror/gcc/blob/16e2427f50c208dfe07d07f18009969502c25dc8/libstdc%2B%2B-v3/include/bits/stl_tree.h#L101`

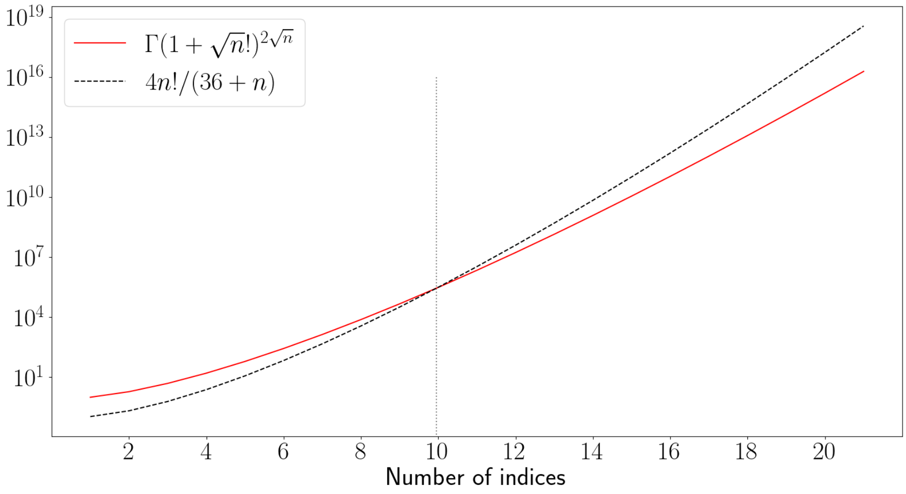


Figure 4.1: Plot showing the point where it becomes more efficient space-wise to use sparse storage for a projection

provided by the binary search tree will use less space

3. In some cases where an object projects to 0, this can be spotted early on in the projection as all the terms will have a coefficient of 0. By using a binary search tree, this is easy to detect as it will mean that the tree is empty, and avoids a large memory allocation of which almost all will never be used.

4. Some algorithms, such as symmetrisation, require iterating over all the non-zero terms in the projection. For a dense storage solution, this requires iterating over many zero terms in order to find the relevant terms which harms efficiency.

4.4 Symmetrisation algorithms

A central focus is, of course, the projection which makes the tensor symmetries manifest. The implementation is split across the `meld` class, which calculates the symmetrisers and antisymmetrisers corresponding to a Young tableau, and the `ProjectedAdjform` class which contains a `apply_young_symmetry` method which is called on each symmetriser in turn.

As well as the symmetrisation due to the Young projection, another symmetrisation required in order to ensure that the implicit symmetry due to identical objects is realised. Consider the simple expression

$$A_a A_b + A_b A_a \tag{4.24}$$

Although there is no explicitly defined symmetry between the index slots holding a and b, as long as A_a is not self non-commuting, then by swapping the order of the objects in the second term this expression can be collapsed to $2A_aA_b$. `meld` can recognise this symmetry by treating this symmetry as a projection operator too:

$$A_a A_b \to \frac{1}{2}(A_a A_b + A_b A_a) \tag{4.25}$$

This algorithm is implemented as a method of the `ProjectedAdjform` class which allows the symmetry to be applied to each element in the projection ensuring that the projection accumulates the new symmetry.

4.4.1 Applying a young symmetriser

The symmetrisation algorithm is defined in the `apply_young_symmetry` method of the class `ProjectedAdjform`. The algorithm accepts a list of the slots in the symmetriser and a flag indicating whether it should symmetrise or antisymmetrise.

The symmetriser is applied to each `Adjform` in the projection in turn, and the new set of terms produced by this are added to the projection. The terms in the projection before the symmetriser is applied need to be stored in a separate copy as otherwise, because the terms in the projection are modified as the symmetriser is being applied, this may result in new terms being erroneously symmetrised.

As noted above, swapping elements in the adjacency representation can be more expensive as swapping a dummy index involves also relabelling the other index in the pair. For this reason, it is desirable to minimise the number of swaps required when calculating each permutation. This is achieved by using an implementation of the Steinhaus-Johnson-Trotter (SJT) algorithm[80, 81] which iterates through all permutations requiring only one swap at each iteration. A further benefit of this is that it produces a sequence of permutations where the parity of the permutation switches between odd and even at each iteration making this trivial to keep track of.

For a given permutation P of the integers $1, 2, \ldots, n$, the algorithm works by generating two auxiliary sequences: X such that $X[P[i]] = i$, i.e. the ith element of X gives the position of i in P, and D[9] where

$$D[i] = \begin{cases} +1 & \text{if } P^{(i)} \text{ is an odd permutation} \\ -1 & \text{if } P^{(i)} \text{ is an even permutation} \end{cases} \tag{4.26}$$

where $P^{(i)}$ is the sequence P where all values $\geq i$ have been removed. The algorithm finds the largest value of i for which $P[X[i] + D[i]] < i$, and then swaps the values in positions $X[i]$ and $X[i] + D[i]$. For instance, starting with the sequence $[1, 2, 3]$ the set of permutations produced is

$$[1,2\overbrace{,3] \to [1,3}^{(2,3)}\underbrace{,2] \to [3,1}_{(1,3)}\overbrace{,2] \to [3,2}^{(1,2)}\underbrace{,1] \to [2,3}_{(2,3)}\overbrace{,1] \to [2,1}^{1,3},3]$$

The sequence is initially fully sorted, so $D[i] = -1$ and $X = [1, 2, 3]$. For $i = 3$ we find $P[X[3] + D[3]] = P[2] = 2 < 3$ so we swap the elements in P at positions $X[3] = 3$ and $X[3] + D[3] = 2$ giving the next permutation in the sequence $P \to [1, 3, 2]$. The subsequences $P^{(i)}$ are still all even permutations; note that in practice D can be updated without explicitly checking the parity of each subsequence as only sequences including the two elements which have been swapped will change parity and so flipping all entries in D after this value will generate the correct sequence. We also set $X = [1, 3, 2]$ and again find $P[X[3]+D[3]] = P[1] = 1 < 3$ and so swap the elements in P at positions $X[3] = 2$ and $X[3] + D[3] = 1$ and so $P \to [3, 1, 2]$. This continues until no suitable value of i can be found which satisfies the criteria $P[X[i] + D[i]] < i$

[9] The auxiliary sequence D is not mentioned in the description of the algorithm as first described by Johnson in [80], however its use is almost ubiquitous in reference implementations (c.f. `https://rosettacode.org/wiki/Permutations_by_swapping`, `https://www.npmjs.com/package/permutation-sjt?activeTab=readme` and `https://github.com/corentinway/permutation_way_rs`. The idea is credited to the computer scientist Shimon Even with all references to this pointing back to the Wikipedia entry for the algorithm `https://en.wikipedia.org/wiki/Steinhaus-Johnson-Trotter_algorithm#Even's_speedup`, although as I have been unable to find a copy of any work by Even mentioning the idea I cannot confirm this.

at which point the algorithm terminates.

Note that the algorithm requires a sequence of consecutive integers from 1 to n; the algorithm can still be used for other sequences by using the values in P as 'pointers' to ith element of the sequence, e.g. given $S = [b, n, g]$ and $P = [2, 3, 1]$, then the permutation of S corresponding to P is given by $[n, g, b]$.

Let us consider the application of the two symmetrisers $\begin{array}{|c|c|c|}\hline 0&1&2\\\hline\end{array}$ and $\begin{array}{|c|c|}\hline 0&3\\\hline\end{array}^{T}$ on the tensor T_{abac}. The adjacency representation of the tensor is $[2, b, 0, c]$ and the initial coefficient is 1, so the `ProjectedAdjform` object starts with the following data:

Slot				Weight
0	1	2	3	
2	b	0	c	1

Of course, in the actual object the indices b and c are represented by negative integers but for clarity they remain named here. The antisymmetriser $\begin{array}{|c|c|}\hline 0&3\\\hline\end{array}^{T}$ is applied to the first (and only) element in the projection so far. Starting from this element, each permutation of the elements in the symmetriser are calculated using the SJT algorithm. The complete set of adjacency representations in the projection are therefore incrementally calculated by applying the index swap calculated by SJT.

Permutation	Swapped	Sign	New term
$[0, 3]$	—	1	$[2, b, 0, c]$
$[3, 0]$	(0,3)	-1	$[c, b, 3, 2]$

The terms in the last column form the complete symmetrisation of the term, with the weights adjusted by the sign in the 'Sign' column to account for the antisymmetry. Thus, after the symmetrisation in these indices the object contains the data

Slot				Weight
0	1	2	3	
2	b	0	c	1
c	b	3	2	-1

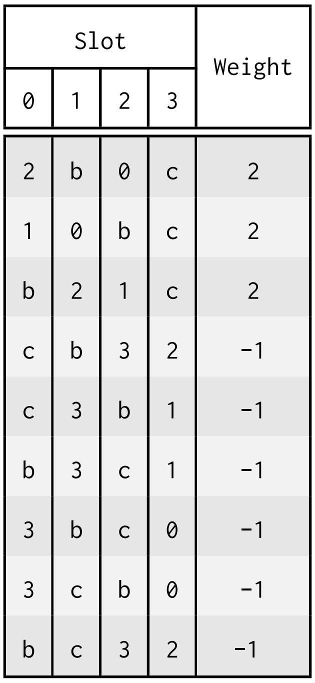
The symmetriser $\begin{array}{|c|c|c|}\hline 0&1&2\\\hline\end{array}$ is now applied to each term in succession: first $[2, b, 0, c]$:

Permutation	Swapped	Sign	New term
$[0, 1, 2]$	—	1	$[2, b, 0, c]$
$[0, 2, 1]$	(1,2)	-1	$[1, 0, b, c]$
$[2, 0, 1]$	(0,2)	1	$[b, 2, 1, c]$
$[2, 1, 0]$	(0,1)	-1	$[2, b, 0, c]$
$[1, 2, 0]$	(1,2)	1	$[1, 0, b, c]$
$[1, 0, 2]$	(0,2)	-1	$[b, 2, 1, c]$

and then $[c, b, 3, 2]$:

Permutation	Swapped	Sign	New term
$[0, 1, 2]$	—	1	$[c, b, 3, 2]$
$[0, 2, 1]$	(1,2)	-1	$[c, 3, b, 1]$
$[2, 0, 1]$	(0,2)	1	$[b, 3, c, 1]$
$[2, 1, 0]$	(0,1)	-1	$[3, b, c, 0]$
$[1, 2, 0]$	(1,2)	1	$[3, c, b, 0]$
$[1, 0, 2]$	(0,2)	-1	$[b, c, 3, 2]$

These terms are summed up to produce the final projection; as this symmetriser is not antisymmetric, the terms are summed up simply without taking into account the sign of the permutation:



Slot				Weight
0	1	2	3	
2	b	0	c	2
1	0	b	c	2
b	2	1	c	2
c	b	3	2	-1
c	3	b	1	-1
b	3	c	1	-1
3	b	c	0	-1
3	c	b	0	-1
b	c	3	2	-1

Note that after the symmetrisation, the weight should be divided through by a normalisation factor dependent on the shape of the tableaux to which the symmetrisers belong. The `meld` algorithm intentionally does not divide through by the normalisation factor in order to ensure that the weights can be stored as plain integer objects instead of using a rational number class which impacts performance severely. It is possible to do this as each term in the expression is acted upon by the same set of tableaux and therefore the normalisation factor picked up by each term is equal. As `meld` only compares terms to each other, this global normalisation cancels out and is therefore irrelevant.

4.5 Projection of an independent symmetriser

A common case for a symmetry is when a particular tensor is completely symmetric or antisymmetric in some subset of indices. In these cases where there is no mixing of other symmetrisers the Young projection will of course permit detection of identical terms, however all that the projection is in practice doing is checking that those index slots contain the same set of index names: if they do, then the set of all permutations of index names in those slots will be the same. It is therefore much faster to sort the indices in each term lexicographically, which will detect any identical terms without requiring generating the $n!$ terms in the totally symmetric projection. In meld therefore, all symmetrisations of totally symmetric/antisymmetric index sets are replaced with a sort. Note that if the subset of indices contains a dummy index then it can no longer be treated as independent, as the dummy index's partner may be located in a slot where another symmetry is to be applied which would require the full projection.

4.5.1 Calculating symmetrisers for products of tableaux

Given a group of terms with the same tensorial structure, the set of symmetrisers corresponding to that structure must be calculated in order for the projection operation to be applied. Each term in the product of factors making up the tensor structure of the monomial is examined for a TableauSymmetry property and if this exists the tableaux represented by that property are collected. The symmetrisers are collected by performing a single pass over the collected tableaux, for each iterating over the columns and then the rows to construct a list of symmetrizer_t objects

```
struct symmetrizer_t {
    std::vector<size_t> indices;
    bool antisymmetric;
    bool independent;
};
```

The indices are collected as the entries in the row or column, and the antisymmetric flag set to true for columns and false for rows. The independent flag records if the indices in the symmetriser are unique, i.e. do not appear in any other symmetriser, which allows the projection to be optimised into a sort as described above. Of course, as the symmetrisers are collected from tableaux independently of the actual index structure of the object which they act upon it is not possible at this stage to determine if it contains any dummy indices which would cause the symmetriser to not be independent, however its independence assuming the indices are free is determined after all the symmetrisers have been collected by attempting to commute it through all the other symmetrisers. The actual symmetrisation algorithm checks both this flag and the index structure of the term it is symmetrising before deciding whether to apply the sorting optimisation or not. Commuting the symmetriser through its neighbours not only allows the detection of any independent symmetrisers, but also checks for two other optimisations which can be applied:

- If two symmetrisers $\mathcal{S}_1$ and $\mathcal{S}_2$ have the same parity (i.e. both symmetric or both antisymmetric) and $\mathcal{S}_1 \subseteq \mathcal{S}_2$ such that all slots in $\mathcal{S}_1$ also appear in $\mathcal{S}_2$, then the product $S_1 S_2 \equiv S_2 S_1 \equiv S_2$. To see this, consider a tensor $T_{i_1 i_2 \ldots i_n}$ and the symmetriser $\mathcal{S}_2$ which permutes a set of slots $\{s_1, s_2, \ldots, s_k\}$. The result of applying $\mathcal{S}_2$ is

$$\mathcal{S}_2 T_{i_1 \ldots i_n} = \frac{1}{k!}(T_{P_1(i_1 \ldots i_n)} + T_{P_2(i_1 \ldots i_n)} + \cdots + T_{P_{k!}(i_1 \ldots i_n)} \tag{4.27}$$

where the P_j are all the $k!$ permutations of the indices in $\mathcal{S}_2$. Now consider $\mathcal{S}_1 = \{s_1, s_2, \ldots, s_l\}$ where $l \leq k$ so that $\mathcal{S}_1 \subseteq \mathcal{S}_2$ and the permutations of indices generated by

$\mathcal{S}_1$ are given by P_j for $1 \leq j \leq l!$. Then the action of $\mathcal{S}_1$ on (4.27) is

$$\begin{aligned}\mathcal{S}_1\mathcal{S}_2 T_{i_1\dots i_n} = \frac{1}{l!\,k!}(&T_{P_1P_1(i_1\dots i_n)} + T_{P_1P_2(i_1\dots i_n)} + \dots + T_{P_1P_{k!}(i_1\dots i_n)} + \\ &T_{P_2P_1(i_1\dots i_n)} + T_{P_2P_2(i_1\dots i_n)} + \dots + T_{P_2P_{k!}(i_1\dots i_n)} + \\ &\dots \\ &T_{P_{l!}P_1(i_1\dots i_n)} + T_{P_{l!}P_2(i_1\dots i_n)} + \dots + T_{P_{l!}P_{k!}(i_1\dots i_n)})\end{aligned} \tag{4.28}$$

Each row contains all permutations $P_j(i_1 \dots i_n)$ for $1 \leq j \leq k!$ in a different order, as it is made up of the permutations P_j left-multiplied by a different $P_{j'}$ for each row. As there are $l!$ rows, this means each permutation appears $l!$ times, so

$$\begin{aligned}\mathcal{S}_1\mathcal{S}_2 T_{i_1\dots i_n} &= \frac{1}{l!\,k!}(l!\,T_{P_1(i_1\dots i_n)} + l!\,T_{P_2(i_1\dots i_n)} + \dots + l!\,T_{P_{k!}(i_1\dots i_n)}) \\ &= \mathcal{S}_2 T_{i_1\dots i_n}\end{aligned} \tag{4.29}$$

A proof for $S_2S_1 \equiv S_2$ can be constructed in the same manner. For antisymmetrisers $\mathcal{A}_1$ and $\mathcal{A}_2 \supseteq \mathcal{A}_1$ then the result $\mathcal{A}_1\mathcal{A}_2 \equiv \mathcal{A}_2\mathcal{A}_1 \equiv \mathcal{A}_2$ also holds as in each row of (4.28) the sign of the permutation P_j will be the same, depending only on the parity of P_j itself regardless of which combination of permutations are used to construct it.

- Two symmetrisers which have opposite parity (i.e. one symmetric and one antisymmetric) and share more than one slot cause the entire projection to be zero. Note that this is different to the zero given in (3.36) which occurs because the object obeys two independent symmetries of different parity which share a single index; when constructing a symmetriser this necessarily does not result in the projection equalling zero or mixed symmetries would be impossible to analyse by using Young projection operators. Consider

$$\begin{aligned}\mathcal{S} &= \begin{array}{|c|c|c|}\hline a & b & \dots \\ \hline\end{array} \\ \mathcal{A} &= \begin{array}{|c|c|c|}\hline a & b & \dots \\ \hline\end{array}^{\,T}\end{aligned} \tag{4.30}$$

where the other slots in $\mathcal{S}$ and $\mathcal{A}$ need not overlap. Then the expression

$$\left(\begin{array}{|c|}\hline a \\ \hline b \\ \hline \vdots \\ \hline\end{array} \otimes \begin{array}{|c|c|c|}\hline a & b & \dots \\ \hline\end{array}\right) T_{i_1 i_2 \dots i_a \dots i_b \dots i_n} \tag{4.31}$$

is equal to zero. To show this, we begin by using the inverse of the previous optimisation to rewrite this as

$$\left(\begin{array}{|c|}\hline a \\ \hline b \\ \hline \vdots \\ \hline\end{array} \otimes \begin{array}{|c|}\hline a \\ \hline b \\ \hline\end{array} \otimes \begin{array}{|c|c|}\hline a & b \\ \hline\end{array} \otimes \begin{array}{|c|c|c|}\hline a & b & \dots \\ \hline\end{array}\right) T_{i_1 i_2 \dots i_a \dots i_b \dots i_n} \tag{4.32}$$

Let

$$T_{k_1 k_2 \dots k_a \dots k_b \dots k_n} \tag{4.33}$$

be one of the permutations generated by the symmetriser $\begin{array}{|c|c|c|}\hline a & b & \ldots \\ \hline\end{array}$, then

$$\begin{aligned}\left(\begin{array}{|c|}\hline a\\ \hline b\\ \hline\end{array}\otimes\begin{array}{|c|c|}\hline a & b\\ \hline\end{array}\right)T_{k_1k_2\ldots k_a\ldots k_b\ldots k_n} &= \frac{1}{2}\begin{array}{|c|}\hline a\\ \hline b\\ \hline\end{array}(T_{k_1k_2\ldots k_a\ldots k_b\ldots k_n}+T_{k_1k_2\ldots k_b\ldots k_a\ldots k_n})\\ &= \frac{1}{4}(T_{k_1k_2\ldots k_a\ldots k_b\ldots k_n}-T_{k_1k_2\ldots k_b\ldots k_a\ldots k_n}\\ &\qquad +T_{k_1k_2\ldots k_b\ldots k_a\ldots k_n}-T_{k_1k_2\ldots k_a\ldots k_b\ldots k_n})\\ &= 0\end{aligned}\tag{4.34}$$

This is true of all the permutations generated by $\begin{array}{|c|c|c|}\hline a & b & \ldots \\ \hline\end{array}$ and so the whole expression (4.31) is equal to zero.

After these optimisations have been applied, the result is a list of symmetrisers which are applied in order.

4.5.2 Calculating symmetrisers for sums of tableaux

Tensors in Cadabra can be assigned multiple tableaux symmetries; ordinarily these are treated as though they were a tensor product of tableaux, but `meld` can also function as though they were terms in a direct sum of tableaux by being passed the `project_as_sum` flag, e.g. in order to obtain a projection which makes a cyclic symmetry $T_{abc} = T_{bca} = T_{cab}$ manifest it is possible to use the sum of a single-row and single-column tableau:

$$\begin{aligned}\left(H\left(\begin{array}{|c|c|c|}\hline 1 & 2 & 3\\ \hline\end{array}\right)\oplus H\left(\begin{array}{|c|}\hline 1\\ \hline 2\\ \hline 3\\ \hline\end{array}\right)\right)T_{abc} = &\ \tfrac{1}{6}(T_{abc}+T_{acb}+T_{bac}+T_{bca}+T_{cab}+T_{cba})+\\ &\ \tfrac{1}{6}(T_{abc}-T_{acb}-T_{bac}+T_{bca}+T_{cab}-T_{cba})\\ = &\ \frac{1}{3}(T_{abc}+T_{bca}+T_{cab})\end{aligned}\tag{4.35}$$

Note that the Hermitian projection operators are required when dealing with sums of tableau; in the case of pure rows and columns these are the same as the standard projectors.

In principle, other than needing to collect the set of Hermitian symmetrisers from each tableau, the collection of the symmetrisers is the same as for collecting the symmetrisers as a product with three main caveats:

- The symmetrisers are returned in a flat list, but it needs to be possible to detect the breakpoints between different tableaux as they are not applied onto the same projection, but are calculated as many projections and then summed
- It is no longer possible to perform the independent symmetriser optimisation, as the result of the projection is a sum of projectors so the terms in the projection of a fully symmetrised object will become mixed and can therefore not be treated as independent.
- The normalisation of the projector now matters, as the different projectors making up the sum will in general have different normalisations and so this can no longer be factored out as a global scaling.

Luckily these three problems are all able to work cooperatively towards a solution: `meld` introduces a dummy `symmetrizer_t` object before the first symmetriser in each tableau which is marked by the `independent` flag being set to `true` as the optimisation for independent symmetry subsets of indices cannot be applied when summing projections so this flag is always `false` for a real symmetriser. The object holds a single value in its data, which corresponds to the normalisation constant of the tableau.

In order to keep the integer arithmetic, instead of dividing through each projection by the normalisation factor of the associated tableau, each projector is first multiplied through by the product of all normalisations, e.g.

$$\frac{1}{N(Y_1)}P(Y_1)+\frac{1}{N(Y_2)}P(Y_2)+\frac{1}{N(Y_2)}P(Y_2)\rightarrow \begin{array}{c} N(Y_2)N(Y_3)P(Y_1)+ \\ N(Y_1)N(Y_3)P(Y_2)+ \\ N(Y_1)N(Y_2)P(Y_3) \end{array} \tag{4.36}$$

4.5.3 Applying a symmetriser in identical tensors

The final aspect of symmetrisation to be discussed is the symmetrisation of identical tensors. The process requires symmetrising in each pair of identical tensors so that any expressions which are equivalent up to swapped terms become manifestly identical:

$$\begin{aligned} A_aA_bA_c &\rightarrow A_aA_bA_c + A_bA_aA_c \\ &\rightarrow A_aA_bA_c + A_aA_cA_b + A_bA_aA_c + A_bA_cA_a \\ &\rightarrow A_aA_bA_c + A_cA_bA_a + A_aA_cA_b + A_bA_cA_a \\ &+ A_bA_aA_c + A_cA_aA_b + A_bA_cA_a + A_aA_cA_b \end{aligned} \tag{4.37}$$

This results in m identical tensors producing 2^m terms. The symmetrisation is performed in the `apply_ident_symmetry` method of the `ProjectedAdjform` class, and accepts three input parameters:

- `positions`: The list of positions in the adjacency representation where the identical tensors are located
- `n_indices`: The number of indices in the tensor
- `commutation_matrix`: A commutation matrix describing whether the terms can be swapped.

For instance, in the example above where the adjacency representation is $[a, b, c]$, the zero-indexed positions of the tensors are $[0, 1, 2]$ and the number of indices is 1. Assuming that $[A_a, A_b] = 0$ the commutation matrix would contain all ones indicating that the terms can be swapped and picks up no change of sign on doing so. The only other values which are accepted by the commutation matrix are -1 if the expression gains a sign change on the swap, or 0 for any commutation relation more complicated which indicates that it is not possible to symmetrise the two tensors. In the example above, the object initially starts with the data

Slot			Weight
0	1	2	
a	b	c	1

First, all pairs involving the first tensor, in this example slots $(0, 1)$ and $(0, 2)$ in the set are symmetrised:

Slot			Weight
0	1	2	
a	b	c	1
b	a	c	1
c	b	a	1

Next, all pairs involving the second slot are symmetrised. In this case this only requires symmetrising in $(1, 2)$ as we have already considered $(0, 1)$:

Slot			Weight
0	1	2	
a	b	c	1
a	c	b	1
b	a	c	1
b	c	a	1
c	b	a	1
c	a	b	1

4.6 Calculating linear dependence between terms

The final step of the algorithm involves detecting whether a projection can be written in terms of a linear combination of other projections. The other projections can all be assumed to be linearly independent from each other as they consist of terms which the algorithm has previous processed and not melded. Let $T_{\{i\}}$ for $1 \leq i \leq n!$ be the set of index permutations of a tensor T with n indices, then some projection $P_k(T)$ can be written

$$P_k(T) = c_{1,k}T_{\{1\}} + c_{2,k}T_{\{2\}} + \cdots + c_{n!,k}T_{\{n!\}} \tag{4.38}$$

where $c_{a,b}$ are the possibly zero coefficients in the projection. At the $(k+1)$th stage in the algorithm, we have k projections and wish to write the $(k+1)$th as a linear combination of these, i.e. the equation to be solved is

$$l_1 P_1(T) + l_2 P_2(T) + \cdots + P_k(T) = P_{k+1}(T) \tag{4.39}$$

where we wish to determine values for l_i. As each $T_{\{i\}}$ is independent, equations for each l_i can be determined by comparing the coefficients of each $T_{\{i\}}$ term:

$$\begin{aligned} l_1 c_{1,1} T_{\{1\}} + l_2 c_{1,2} T_{\{1\}} + \cdots + l_k c_{1,k} T_{\{1\}} &= c_{1,k+1} T_{\{1\}} \\ l_1 c_{2,1} T_{\{2\}} + l_2 c_{2,2} T_{\{2\}} + \cdots + l_k c_{2,k} T_{\{2\}} &= c_{2,k+1} T_{\{2\}} \\ \cdots \\ l_1 c_{n!,1} T_{\{n!\}} + l_2 c_{n!,2} T_{\{n!\}} + \cdots + l_k c_{n!,k} T_{\{n!\}} &= c_{n!,k+1} T_{\{n!\}} \end{aligned} \tag{4.40}$$

There are $n!$ such equations and can thus be represented by the $n! \times k$ matrix equation

$$C_k \vec{l} = \vec{c}_{k+1} \tag{4.41}$$

where we define C_k, $\vec{l}$ and $\vec{c}_{k+1}$ as

$$\begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,k} \\ c_{2,1} & \ddots & & \\ \vdots & & & \\ c_{n!,1} & & & c_{n!,k} \end{bmatrix} \begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_k \end{bmatrix} = \begin{bmatrix} c_{1,k+1} \\ c_{2,k+1} \\ \vdots \\ c_{n!,k+1} \end{bmatrix} \tag{4.42}$$

i.e. $C_k = [\vec{c}_1 \vec{c}_2 \ldots \vec{c}_k]$. As each column of C_k is linearly independent, its rank is k and there are thus three possibilities for the solution set

1. There are k non-zero rows in C_k and there is a single unique solution
2. There are more than k non-zero rows in C_k, which form a consistent system of linear equations with a single unique solution
3. There are more than k non-zero rows in C_k, which do not form a consistent system of linear equations and there is no solution.

In cases 2 and 3 standard linear algebra techniques, such as reducing the matrix C_k to row echelon form, can be used to determine the solution set, however there are several reasons why this is undesirable:

- Each projection may contain many thousands of terms, thus creating the explicit matrix increases storage requirements
- Even if a sparse matrix representation is used, as opposed to the `ProjectedAdjform` objects which can access elements using adjacency representations as keys matrices require integer indexing requiring the expensive calculation of Lehmer codes each time an entry is required
- Most matrix libraries do not provide overloads for matrix operations using integer data types

This last point is in some aspects unavoidable, as the linear combinations will in general consist of rational multiples of terms and thus requires rational arithmetic types, however we still seek to minimise the extent to which this is required. The explicit construction of the matrix C_k can be avoided by instead finding a $k \times k$ submatrix C'_k constructed by selecting linearly independent rows of the matrix and attempting to find a solution to the square system

$$C'_k \vec{l} = \vec{c'}_{k+1} \tag{4.43}$$

then substituting back into the full set of equations (4.40) to ensure that the solution is consistent, which is the approach `meld` takes. As k is typically a relatively small number, solving (4.43) is generally a very fast process with the expensive process being verifying the solution such that the overall complexity is linear in the total number of unique terms in the projections.

Alongside C'_k, a map between the rows of the matrix and the adjacency representations the row represents is kept. At the $(k+1)$th stage, if there is no solution to (4.43) then the matrix C'_k is grown by choosing a new term from $\vec{c}_{k+1}$ which ensures that the $(k+1)$th column is linearly independent of the first k columns. Consider a variant on the example in (4.1):

$$T_{abc} + T_{bca} + T_{cab} \tag{4.44}$$

where the symmetry is described by

$$\begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & \multicolumn{1}{c}{} \\ \cline{1-1} \end{array} \tag{4.45}$$

At the first stage, $k = 0$, we consider the first term T_{abc} with the projection

Slot			Weight
0	1	2	
a	b	c	1
b	a	c	1
b	c	a	-1
c	b	a	-1

As $C_0 = \mathbf{0}$ there are no solutions to (4.42). We form the matrix C'_1 by choosing an arbitrary element from the projection of T_{abc}

$$\begin{array}{c|c|} & T_{abc} \\ \text{abc} & 1 \end{array}$$

At the next stage $k = 1$ we consider the term T_{bca} with the projection

Slot			Weight
0	1	2	
a	c	b	-1
b	c	a	1
c	a	b	-1
c	b	a	1

The equation (4.42) is

$$[1][l_1] = [0] \tag{4.46}$$

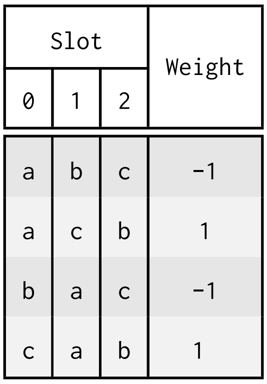
with the solution $l_1 = 0$. This immediately tells us that the two terms by themselves are linearly independent, but in general we would iterate through all the equations in (4.40) between these two terms to check for consistency:

$$\begin{array}{ll} \texttt{abc} & 0 \times 1 = 0 \\ \texttt{acb} & 0 \times 0 \neq -1 \end{array} \tag{4.47}$$

As soon as the inconsistent equation is found, the solution is invalid and we instead grow $C'_1 \to C'_2$ by choosing a new representative term. Again the choice is arbitrary up to the condition that the new column must be linearly independent from the first. The first term in the projection, acb, satisfies this criteria so we set C'_2 to

$$\begin{array}{c|cc|} & T_{abc} & T_{bca} \\ \texttt{abc} & 1 & 0 \\ \texttt{acb} & 0 & \text{-1} \end{array}$$

At the final stage $k = 2$ we examine the projection of T_{cab}:



Slot			Weight
0	1	2	
a	b	c	-1
a	c	b	1
b	a	c	-1
c	a	b	1

from which we derive the equation

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} l_1 \\ l_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \tag{4.48}$$

with the solutions $l_1 = l_2 = -1$. We again check the consistency of the equations

$$\begin{array}{llll} \texttt{abc} & -1 \times 1 & + -1 \times 0 & = -1 \\ \texttt{acb} & -1 \times 0 & + -1 \times -1 & = 1 \\ \texttt{bac} & -1 \times 1 & + -1 \times 0 & = -1 \\ \texttt{bca} & -1 \times -1 & + -1 \times 1 & = 0 \\ \texttt{cab} & -1 \times 0 & + -1 \times -1 & = 1 \\ \texttt{cba} & -1 \times -1 & + -1 \times 1 & = 0 \end{array} \tag{4.49}$$

The solution is consistent and it is determined that $-T_{abc} - T_{bca} = T_{cab}$ allowing these terms to be combined.

4.7 The complete algorithm

We have now discussed all the individual algorithms which make up the various components of meld. It is now time to zoom out from the low-level description of each component, and discuss at a higher level how each part fits in to the overall picture. The logic of the apply_tableau function will be examined line by line, with some whitespace and comments between logical sections omitted. Once a node where the algorithm can be applied has been determined using `can_apply_tableaux`, the algorithm calls `apply_tableaux`:

```
bool meld::apply_tableaux(iterator it)
{
```

The algorithm returns `true` if any melding took place, otherwise it returns `false`. The algorithm begins by checking if the node is an equality, in which case both sides must be called separately:

```
    if (*it->name == "\\equals") {
        bool res = false;
        Ex::sibling_iterator side = it.begin();
        res |= apply_tableaux(side);
        ++side;
        res |= apply_tableaux(side);
        return res;
    }
```

Next the matrix and vector classes are specified using the `mpq_class` which Cadabra uses for rational number types

```
    using namespace boost::numeric::ublas;
    using matrix_type = matrix<mpq_class>;
    using vector_type = vector<mpq_class>;
```

A variable so that we know whether any melding has taken place and which will serve as the return value for the algorithm is also defined before the main logic of the function

```
    bool applied = false;
```

The first true step of the algorithm is to sort the terms into groups of matching structure:

```
    std::vector<std::vector<ProjectedTerm>> patterns;
    for (const auto& term : split_it(it, "\\sum")) {
        ProjectedTerm projected_term(kernel, index_map, tr, term);
        if (projected_term.ident.size() == 0)
            continue;
        bool found = false;
        for (auto& pattern : patterns) {
            if (pattern[0].compare(kernel, projected_term)) {
                found = true;
                pattern.push_back(projected_term);
                break;
            }
        }
        if (!found)
            patterns.emplace_back(1, std::move(projected_term));
    }
```

`split_it` conditionally returns either an iterator by itself in a list, or a list of its children depending on its name, allowing us to treat a non-sum node as a sum of one element. Each term is converted to a `ProjectedTerm` object which is then compared against the first element of each ‘pattern’; if it matches then the term is added to that pattern group or else if it matches no pattern then a new pattern group is created to which it is appended. The algorithm then continues to iterate over each pattern group

```
    for (auto& terms : patterns) {
        ScopedProgressGroup group(pm,
            "Melding terms of form " + ex_to_string<DisplayTerminal>(kernel,
                terms[0].tensor),
            terms.size());
```

The progress monitor provides output to the GUI, allowing the user to track the progress of the algorithm which for large expressions with complicated symmetries might otherwise make Cadabra unresponsive for a long time. Next, the matrix and linear solver for this pattern group are initialised

```
        linear::Solver<mpq_class> solver;
        matrix_type coeffs;
        std::vector<Adjform> mapping;
```

The `linear::Solver` class implements the factorisation and solving algorithm for equations of the form $A\vec{x} = \vec{y}$ for the variable $\vec{x}$, as is the case for our problem. The matrix `coeffs` corresponds to C'_k above, and the ith element of `mapping` is the adjacency representation corresponding to the ith row of C'_k. We now collect the tableaux associated with the pattern group by querying the properties of the first element in the group, and then collect the corresponding symmetrisers

```
        auto tabs = collect_tableaux(terms[0].tensor);
        std::vector<symmetrizer_t> symmetrizers;
        bool is_zero = collect_symmetries(tabs, symmetrizers);
```

If the symmetries can only be satisfied if the object is identically zero, then `collect_symmetries` returns `true` and we delete all the objects in the group, indicate that the algorithm has applied some transformations and move on to the next pattern group

```
        if (is_zero) {
            // The term is identically zero due to its tableaux, delete all and move
            // onto next pattern
            for (auto& term : terms) {
                node_zero(term.it);
                applied = true;
            }
            continue;
        }
```

If the projection is not identically zero for any manifestly obvious reason, we begin iterating over each term in the projection and calculate its projection:

```
        for (size_t term_idx = 0; term_idx < terms.size(); ++term_idx) {
            group.progress();
            auto& term = terms[term_idx];
            symmetrize(term, symmetrizers);
```

It is still possible that the projection is zero, for example due to the arrangement of dummy indices, and if so the term is erased

```
            if (term.projection.empty()) {
                // Empty adjform means that the term is identically equal to 0
                node_zero(term.it);
                terms.erase(terms.begin() + term_idx);
                --term_idx;
                applied = true;
            }
```

If not, then we attempt to solve the equation (4.43). We declare the two vectors $\vec{l}$ and $\vec{c'}_{k+1}$, named `x` and `y` respectively and initially assume there is no solution.

```
            else {
                // We need to try and express the current YP as a linear combination of previous YPs
                // by solving "coeffs * x = y"
                vector_type x, y;
                bool has_solution = false;
```

If the matrix `coeffs` has some dimensions, i.e. $k > 0$, then we fill the vector $\vec{y}$ with the corresponding elements and solve the square system:

```
                if (coeffs.size1() > 0) {
                    y.resize(coeffs.size1());
                    for (size_t i = 0; i < mapping.size(); ++i)
                        y(i) = term.projection.get(mapping[i]);
                    x = solver.solve(y);
                    has_solution = true;
```

We assume that this solution is consistent, and now iterate through the already projected terms in the pattern group and check for inconsistencies. To do this, we hold iterators to the first term in each previously calculated projection (the `lhs_its`) as well as the first term in the current projection (`rhs_it`). The term corresponding to the equation we are currently checking is held in `cur_term`

```
                    std::vector<ProjectedAdjform::const_iterator> lhs_its;
                    ProjectedAdjform::const_iterator rhs_it = term.projection.
                        begin();
                    Adjform cur_term = rhs_it->first;
```

`lhs_its` is populated, and if the first term of any of these projections sorts lexicographically less than the first term of the current projection we update `cur_term`

```
                    for (size_t i = 0; i < term_idx; ++i) {
                        auto it = terms[i].projection.begin();
                        if (it->first < cur_term)
                            cur_term = it->first;
                        lhs_its.push_back(it);
                    }
```

We now iterate through all the terms calculating each expression. As the `ProjectedTerm` storage is sparse, instead of incrementing all iterators at the same time, they are only incremented once we have calculated the term for the value they currently point to. This continues until all the iterators are expired, i.e. all equations have been calculated:

```
                    size_t n_finished = 0;
                    while (n_finished < lhs_its.size()) {
```

Any iterators which point to `cur_term` are included in the sum and incremented, otherwise they are left alone. The next term is concurrently calculated, starting as a value lexicographically greater than any other adjacency representation can be, and then being compared to each iterator after any increments have taken place:

```
                        Adjform next_term;
                        next_term.push_coordinate(std::numeric_limits<Adjform::
                            value_type>::max());
                        mpq_class sum = 0;
                        for (size_t i = 0; i < term_idx; ++i) {
                            if (lhs_its[i] != terms[i].projection.end() &&
                                lhs_its[i]->first == cur_term) {
                                sum += x(i) * lhs_its[i]->second;
                                ++lhs_its[i];
                                if (lhs_its[i] == terms[i].projection.end())
                                    ++n_finished;
                            }
                            if (lhs_its[i] != terms[i].projection.end() &&
                                lhs_its[i]->first < next_term)
                                next_term = lhs_its[i]->first;
                        }
```

A similar process takes place for the right hand side of the equation

```
                        mpq_class rhs_sum;
                        if (rhs_it == term.projection.end() || rhs_it->first !=
                             cur_term) {
                            rhs_sum = 0;
                        }
                        else {
                            rhs_sum = rhs_it->second;
                            ++rhs_it;
                        }
```

If the two sums do not agree, then has_solution is set to false and the consistency check can end

```
                        if (sum != rhs_sum) {
                            has_solution = false;
                            break;
                        }
```

Finally, we check if the adjacency representation rhs_it points to compares less than the current value of next_term and set cur_term ready for the next iteration

```
                        if (rhs_it != term.projection.end() && rhs_it->first <
                            next_term)
                            next_term = rhs_it->first;
                        cur_term = next_term;
                    }
```

Once all iterators on the left-hand side have expired, if there are still terms to process on the right hand side then the set of equations is inconsistent

```
                    if (rhs_it != term.projection.end())
                        has_solution = false;
                }
```

The logic now depends on whether we found a solution or not

```
                if (has_solution) {
```

If the solution exists then the scalar component of the current term multiplied by the associated weight in the vector $\vec{l}$ is added onto the scalar components of each term of which it is a linear combination, and the changed flag for these terms set so that the algorithm knows to modify these in the actual expression at the end of the process

```
                    for (size_t i = 0; i < term_idx; ++i) {
                        if (x(i) != 0) {
                            terms[i].changed = true;
                            Ex::iterator scalar_head = term.scalar.begin();
                            for (Ex::sibling_iterator beg = scalar_head.begin()
                                , end = scalar_head.end(); beg != end; ++beg) {
                                auto new_term = terms[i].scalar.append_child(
                                    terms[i].scalar.begin(), (Ex::iterator)beg)
                                    ;
                                multiply(new_term->multiplier, x(i) * (*
                                    scalar_head->multiplier));
                            }
                        }
                    }
```

The applied flag is then set and the current term removed from the expression

```
                    applied = true;
                    node_zero(term.it);
                    terms.erase(terms.begin() + term_idx);
                    --term_idx;
```

If no solution was found, then the matrix C_k' is grown

```
                else {
                    // Expand the dimensions of the matrix by 1
                    coeffs.resize(coeffs.size1() + 1, coeffs.size2() + 1);
```

We then find an adjacency representation which isn't already included in the matrix and use this as the next row of the matrix, filling in the new row and column

```
                    bool found = false;
                    for (const auto& kv : term.projection) {
                        auto pos = std::find(mapping.begin(), mapping.end(), kv.first);
                        if (pos == mapping.end()) {
                            // Fill in bottom row
                            for (size_t i = 0; i < term_idx; ++i)
                                coeffs(coeffs.size1() - 1, i) = terms[i].projection.get(kv.first);
                            // Fill in the righthand column
                            for (size_t i = 0; i < mapping.size(); ++i)
                                coeffs(i, coeffs.size2() - 1) = term.projection.get(mapping[i]);
                            // Fill in the bottom right element
                            coeffs(coeffs.size1() - 1, coeffs.size2() - 1) =
                                term.projection.get(kv.first);
                            if (solver.factorize(coeffs)) {
                                mapping.push_back(kv.first);
                                found = true;
                                break;
                            }
                        }
                    }
```

A check to ensure that the matrix was grown is performed, to ensure that any bugs are caught and the process doesn't continue with junk values in the matrix

```
                    if (!found)
                        throw std::runtime_error("Could not find a suitable element to add to the matrix");
                } // if (has_solution) {} else {}
            } // if (term.projection.empty()) {} else {}
        } //for (size_t term_idx = 0; term_idx < terms.size(); ++term_idx)
```

Finally, once all terms in the pattern have been processed any terms which have had their scalar components modified are updated in the actual expression

```
        for (auto& term : terms) {
            if (term.changed) {
                // Replace the node with a product of the scalar and tensor parts, then cleanup
                tr.erase_children(term.it);
                term.it = tr.replace(term.it, str_node("\\prod"));
                tr.append_child(term.it, term.scalar.begin());
                tr.append_child(term.it, term.tensor.begin());
                cleanup_dispatch(kernel, tr, term.it);
            }
        }
    } // for (auto& terms : patterns)
```

The algorithm returns the applied flag to indicate if any changes took place

```
    return applied;
}
```

4.8 Example usage

One of the motivating examples for the construction of the `meld` algorithm is the ability to generally canonicalise expressions involving Riemann tensors where there may be hidden Bianchi identities.A canonicalisation algorithm based on mono-term canonicalisation techniques, such as the Butler-Portugal algorithm, would not be able to detect this type of symmetry, and many are not listed in lookup tables used by pattern detection algorithms. An illustrative example is the polynomial

$$R_{abcd}R_{abcd} + R_{abcd}R_{acbd} \tag{4.50}$$

The second term is in fact equal to half the first term due to a Bianchi identity, so although the simplification seems to be due to a mono-term symmetry as only two terms are involved `canonicalise` has no effect on the expression

```
R_{a b c d}::RiemannTensor.
canonicalise($R_{a b c d}R_{a b c d} + R_{a b c d}R_{a c b d}$);
```

$R_{abcd}R_{abcd} + R_{abcd}R_{acbd}$

however `meld` is able to simplify the expression

```
meld($R_{a b c d}R_{a b c d} + R_{a b c d}R_{a c b d}$);
```

$\frac{3}{2}R_{abcd}R_{abcd}$

Of course, as meld only combines terms and makes no attempt to canonicalise the order of the indices, if the expression were instead

$$R_{abcd}R_{acbd} + R_{abcd}R_{abcd} \tag{4.51}$$

then `meld` calculates the projections in the inverse order and produces the output in terms of $R_{abcd}R_{acbd}$:

```
meld($R_{a b c d}R_{a c b d} + R_{a b c d}R_{a b c d}$);
```

$3R_{abcd}R_{acbd}$

A simple polynomial such as this may be hard-coded into some computer algebra systems, but the generality of `meld` can be shown by combining the expression with an antisymmetric tensor

```
R_{a b c d}::RiemannTensor.
\epsilon{#}::AntiSymmetric.
ex := R_{a b e f}R_{c d g h}\epsilon_{f e h g} + R_{a b e f}R_{c g d h}\
    epsilon_{g e f h};
meld(ex);
```

$$R_{abef}R_{cdgh}\epsilon_{fehg} + R_{abef}R_{cgdh}\epsilon_{gefh}$$

$$\frac{3}{2}R_{abef}R_{cdgh}\epsilon_{fehg}$$

where the way in which the dummy indices interact may make lookup table techniques fail. Another example is identity 3.1 given in [82]:

$$R_{(abc}{}^{m}R_{d)me}{}^{f} - R_{ace}{}^{m}R_{bdm}{}^{f} + R_{acm}{}^{f}R_{bde}{}^{m} = 0 \tag{4.52}$$

```
exA = asym($R_{a b c}^{m}R_{d m e}^{f}$, $_{a}, _{b}, _{c}, _{d}$,
    antisymmetric=False)
ex2 := @(exA) - R_{a c e}^{m}R_{b d m}^{f} + R_{a c m}^{f}R_{b d e}^{m}.
meld(ex);
```

$$0$$

A more computationally expensive example involves a fourth order polynomial of the Weyl tensor, which is symmetrically identical to the Riemann tensor

```
W_{m n p q}::WeylTensor;
ex:= W_{p q r s} W_{p t r u} W_{t v q w} W_{u v s w}
     - W_{p q r s} W_{p q t u} W_{r v t w} W_{s v u w}
     - W_{m n a b} W_{n p b c} W_{m s c d} W_{s p d a}
     + (1/4) W_{m n a b} W_{p s b a} W_{m p c d} W_{n s d c};
meld(ex);
```

$$W_{pqrs}W_{ptru}W_{tvqw}W_{uvsw} - W_{pqrs}W_{pqtu}W_{rvtw}W_{svuw} - W_{mnab}W_{npbc}W_{mscd}W_{spda} + (1/4)W_{mnab}W_{psba}W_{mpcd}W_{nsdc}$$

$$0$$

The algorithm over a series of runs averaged a runtime of 2.68s on a standard home machine for this calculation.

Derivatives of Riemann tensors with a metric connection also satisfy higher order Bianchi identities, with the standard example of the first derivative being

$$\nabla_e R_{abcd} + \nabla_c R_{abde} + \nabla_d R_{abec} \tag{4.53}$$

The symmetry of the expression $T_e R_{abcd}$ for some arbitrary tensor T_e is given in general by the Littlewood-Richardson decomposition of the direct product of the two tableaux

$$\begin{array}{|c|}\hline e\\\hline\end{array} \otimes \begin{array}{|c|c|}\hline a & c\\\hline b & d\\\hline\end{array} = \begin{array}{|c|c|c|}\hline a & c & e\\\hline b & d & \multicolumn{1}{c}{}\\\cline{1-2}\end{array} \oplus \begin{array}{|c|c|}\hline a & c\\\hline b & d\\\hline e & \multicolumn{1}{c}{}\\\cline{1-1}\end{array} \tag{4.54}$$

Only the tableau

$$\begin{array}{|c|c|c|}\hline a & c & e\\\hline b & d & \multicolumn{1}{c}{}\\\cline{1-2}\end{array} \tag{4.55}$$

satisfies the Bianchi identity and so the symmetry of the object is described by this 'hook' tableau. The process can be repeated [83, 84] for higher derivatives to show that the symmetry of $\nabla_{e_1}\nabla_{e_2}\ldots\nabla_{e_n}R_{abcd}$ is

$$\begin{array}{l}\overbrace{\begin{array}{|c|c|c|c|c|c|}\hline a & b & e_1 & e_2 & \cdots & e_n \\ \hline c & d \\ \cline{1-2}\end{array}}^{n+2\text{ boxes}}\end{array} \tag{4.56}$$

Expressions involving derivatives of Riemann tensors can therefore be simplified by defining objects with properties associating them with these hook tableaux and substituting them into the equation before calling `meld`, however as this is such as common case the hook-tableau is automatically calculated by `meld` when a derivative of a Riemann tensor is encountered

```
\nabla{#}::Derivative.
R_{a b c d}::RiemannTensor.
ex := \nabla_{e}{R_{a b c d}} + \nabla_{c}{R_{a b d e}} +
      \nabla_{d}{R_{a b e c}};
meld(ex);
```

$\nabla_e R_{abcd} + \nabla_c R_{abde} + \nabla_d R_{abec}$

0

and similarly for higher order identities, such as the following from [85]

$$R^{abcd;e}{}_{a}R_{be}{}^{fg;hi}R_{cfgi;dh} = \frac{1}{8}R^{abcd;e}{}_{e}R_{ab}{}^{fg;hi}R_{cdfg;ih} \tag{4.57}$$

```
\nabla{#}::Derivative.
R^{a b c d}::RiemannTensor.
R_{a b}^{c d}::RiemannTensor.
R_{a b c d}::RiemannTensor.
ex := \nabla^{e}{\nabla_{a}{R^{a b c d}}}
      \nabla^{h}{\nabla^{i}{R_{b e}{}^{f g}}}
      \nabla_{d}{\nabla_{h}{R_{c f g i}}} -
      1/8 \nabla^{e}{\nabla_{e}{R^{a b c d}}}
      \nabla^{h}{\nabla^{i}{R_{a b}^{f g}}}
      \nabla_{i}{\nabla_{h}{R_{c d f g}}};
meld(ex);
```

0

Two more identities from [82] serve to demonstrate this point:

$$\nabla_a R_{bcd}{}^{e} - \nabla_b R_{adc}{}^{e} + \nabla_c R_{adb}{}^{e} - \nabla_d R_{bca}{}^{e} = 0 \tag{4.58}$$

```
\nabla{#}::Derivative.
R_{a b c}^{d}::RiemannTensor.
ex := \nabla_{a}{R_{b c d}^{e}} - \nabla_{b}{R_{a d c}^{e}} +
      \nabla_{c}{R_{a d b}^{e}} - \nabla_{d}{R_{b c a}^{e}};
meld(ex);
```

0

and

$$[\nabla_a,\nabla_b]R_{cdef} + [\nabla_c,\nabla_d]R_{abef} + [\nabla_e,\nabla_f]R_{abcd} = 0 \tag{4.59}$$

```
\nabla{#}::Derivative.
R_{a b c d}::RiemannTensor.
ex :=
    \nabla_{a}{\nabla_{b}{R_{c d e f}}} - \nabla_{b}{\nabla_{a}{R_{c d e f}}} +
    \nabla_{c}{\nabla_{d}{R_{a b e f}}} - \nabla_{d}{\nabla_{c}{R_{a b e f}}} +
    \nabla_{e}{\nabla_{f}{R_{a b c d}}} - \nabla_{f}{\nabla_{e}{R_{a b c d}}};
meld(ex);
```

0

The separation of scalar factors from projections in `meld` also means that even a Bianchi identity with unknown ratios between the terms can be simplified

```
R_{a b c d}::RiemannTensor.
ex := \alpha R_{a b c d} + \beta R_{a c d b} + \gamma R_{a d b c};
meld(ex);
```

$\alpha R_{abcd} + \beta R_{acdb} + \gamma R_{adbc}$

$(\alpha - \gamma) R_{abcd} + (\beta - \gamma) R_{acdb}$

4.9 Complexity

As with all algorithms, one of the most important considerations is whether they run in a practical amount of time. In section 3.2.4, the complexity of mono-term canonicalisation algorithms was discussed, and although `meld` is not directly analogous to classic canonicalisation algorithms in its behaviour there are certainly many instances where their motivations are aligned and in these cases comparisons between their complexities also becomes relevant. The complexity of the `meld` algorithm is dependent on three main factors which are discussed in detail here. The graphs used in this chapter were produced using the same Cadabra notebooks used in section 2.8 of [72].

4.9.1 Number of terms in the expression

For each term in a sum, `meld` must calculate the projection of the term and attempt to solve the set of linear equations for the term. The cost of calculating the projection is constant inside a particular pattern group, however solving the linear system depends on the number of independent representations which have been discovered, with each equation to be solved containing as many terms as there are independent representations and so the complexity of the equation grows linearly with respect to this. The maximum number of independent representations is determined by the symmetry of the object, with the ith stage of the algorithm having at most $i - 1$ independent representations discovered (i.e. the total number of terms so far processed) which leads to a worst case behaviour which is quadratic in the total number of stages the algorithm must make, i.e. the number of terms N, and so the overall complexity is given by

$$O(N + \epsilon N^2) \tag{4.60}$$

where the factor ϵ, related the difference in computation power required to calculate the projection and solve the linear system, is in practice very small making this term irrelevant especially as the number of independent representations is almost always far smaller than N so the quadratic behaviour almost never manifests in real world calculations.

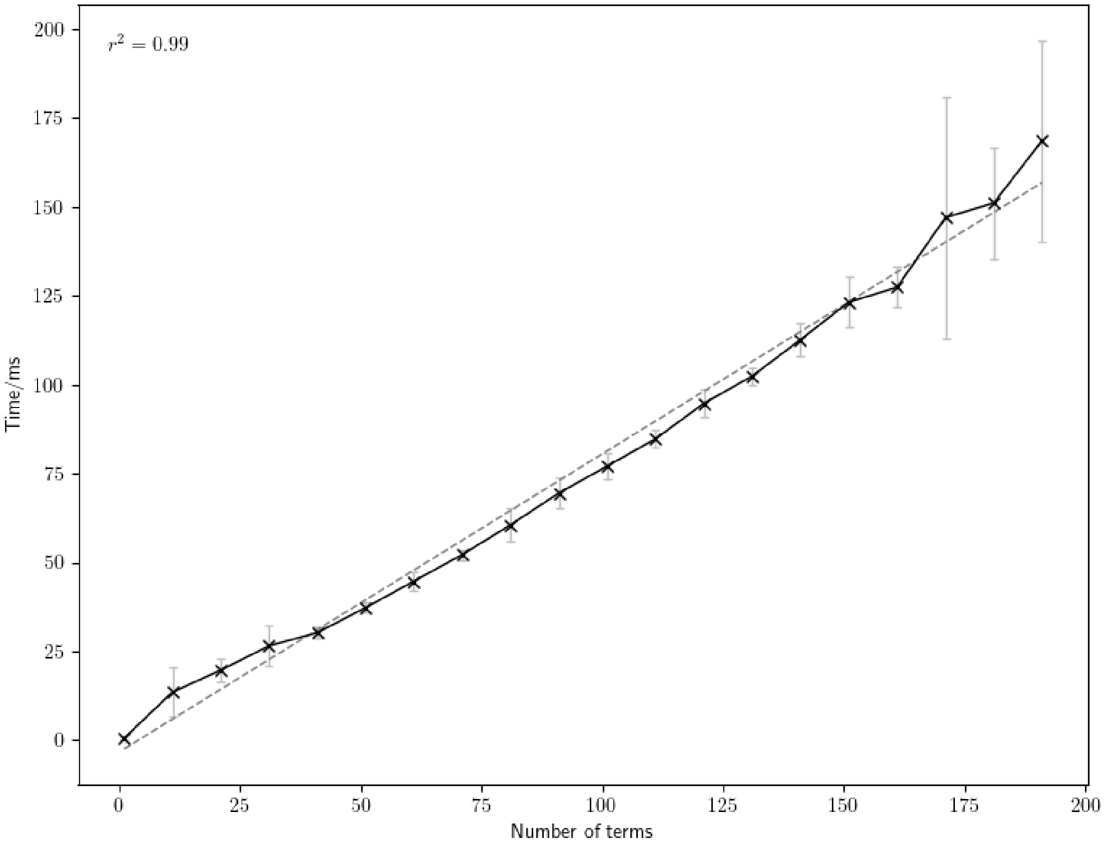


Figure 4.2: Behaviour of meld as the number of terms in the expression increases

The behaviour of melding randomly generated sums of Riemann monomials is shown in Fig. 4.2, with a line showing a linear regression fit superimposed.

4.9.2 Number of dummy indices in each term

Dummy indices introduce a redundancy in the number of permutations of a set of indices, reducing the total number of independent terms which are possible in a projection. This reduces the complexity of the algorithm by modifying two factors: the number of memory allocations which must be made in order to generate new terms, and the number of equations to be checked when solving the linear system. We therefore expect a reduction in the runtime of the algorithm for expressions which differ only in having more pairs of dummy indices, i.e. each expression in the chain

$$R_{abcd}R_{efgh} \to R_{aabc}R_{defg} \to R_{aabb}R_{cdef} \to R_{aabb}R_{ccde} \to R_{aabb}R_{ccdd} \tag{4.61}$$

The exact effect of this behaviour, shown in Fig. 4.3, is harder to quantify explicitly especially for relatively small number of indices where the ratio cannot be treated as a continuous quantity, and for many tableau shapes (including the Riemann symmetries) certain combinations of dummy indices lead to expressions which are identically zero meaning that testing by generating random expressions is heavily biased towards smaller runtimes for higher numbers of dummy indices. The graph of the behaviour generally supports the reduction in complexity as the ratio of dummy indices increases.

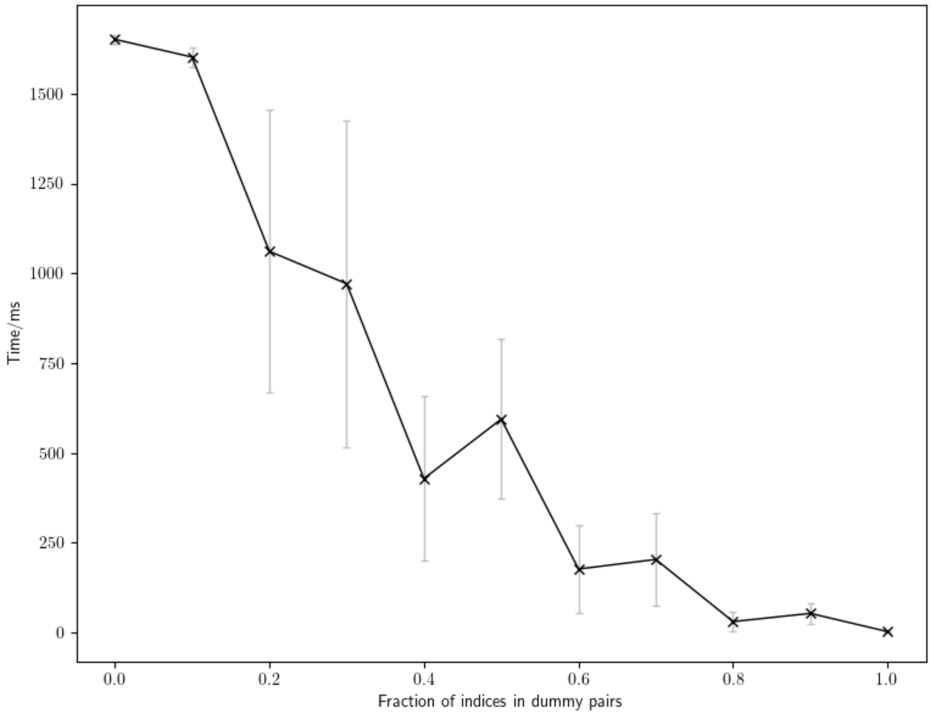


Figure 4.3: Behaviour of meld as the ratio of dummy indices to free indices increases

4.10 Composition of tableaux

The tableau associated with the symmetry effects the complexity of the algorithm in numerous ways:

1. The worst-case running time for the algorithm is proportional to $n!$ for a tableau with n cells, so tableaux which act over a smaller set of indices are more efficient to compute
2. The shape of the tableau itself effects the total number of terms which must be computed during the projection
3. Terms whose symmetry is a combination of products of tableaux, for instance polynomials of tensors, generally perform better than tensors described by a large tableau.

The effect the shape of the tableau has on the performance is shown in Fig. 4.4[10] for a collection of 9 celled tableaux, which shows that the shape of the tableau can have a huge impact on the efficiency.

When the projection of a product of N identical tableau is calculated, the computation of the ith tableau requires projecting each term in the projection calculated for the $(i-1)$ previous terms. Each projection does not produce an identical term to a previous projection, as the projection is calculated over a different set of indices (assuming there are no dummy indices). If each projection produces k terms, then a quadratic polynomial produces k^2 terms,

[10] The optimisations for fully (anti-)symmetric tensors is disabled to make the mathematical structure clearer, in reality the tableau shapes $\square^{\otimes 9}$ and $(\square^{\otimes 9})^T$ would be by far the most efficient

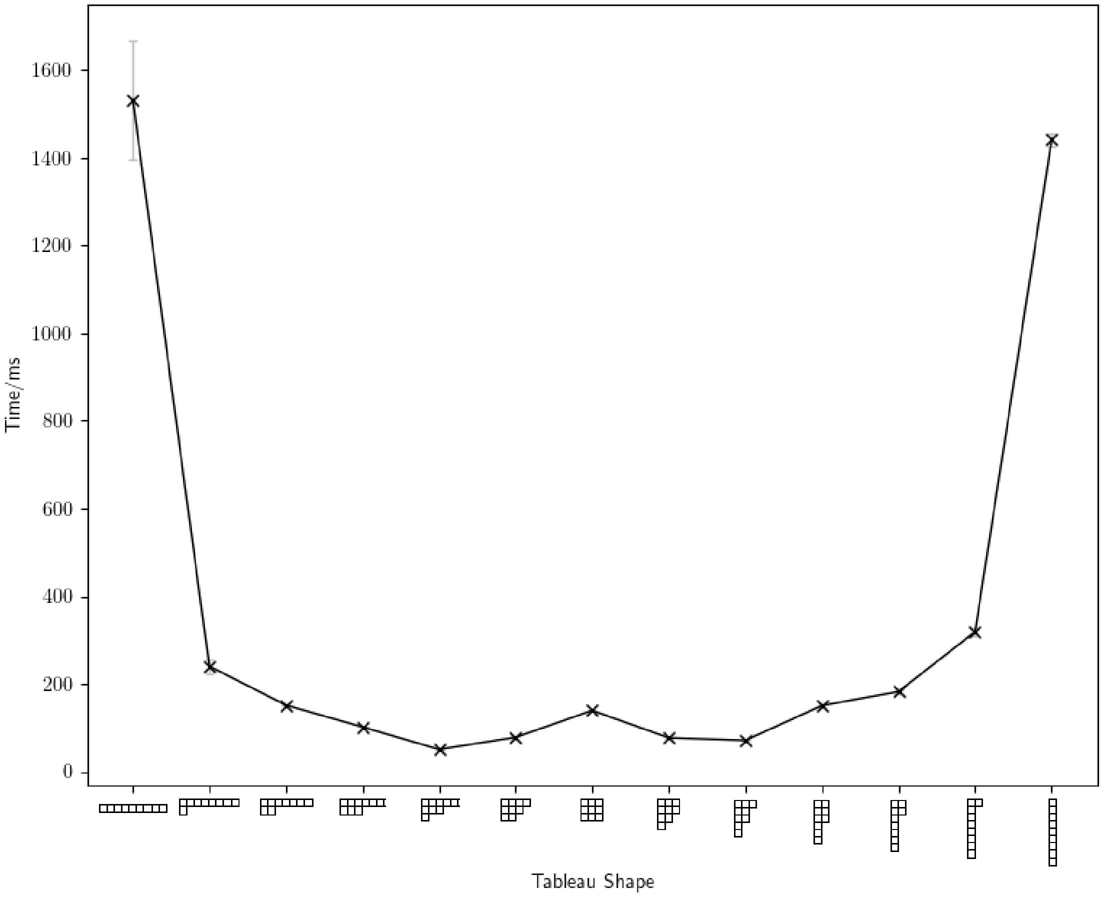


Figure 4.4: Behaviour of meld as the ratio of dummy indices to free indices increases

and similarly a Nth order polynomial produces k^N terms, and so the behaviour can be seen to be exponential in the order of the polynomial for expressions containing only free indices; dummy indices will of course reduce the complexity of the algorithm as discussed above. This behaviour is shown in Fig. 4.5 where the runtime of melding expressions containing Riemann polynomials is shown is shown to asymptotically approach an exponential curve. The effect by which lower orders are above the expected line, but have a smaller growth rate, is at least in part due to the constant overhead associated with calling `meld` and initialising the data structures which does not vary with the polynomial order.

4.10.1 Comparison to `canonicalise` and practical limits

In general, the `meld` algorithm is far slower than canonicalisation routines based on e.g. the Butler-Portugal algorithm due to the large expense which comes with calculating the Young projection of a term. The performance becomes markedly worse as the number of indices increases, with the worst-case factorial growth of `meld` unable to compete with the polynomial complexity of the Butler-Portugal algorithm in these cases. Even in cases where the Butler-Portugal algorithm also suffers from factorial growth, the overheads are generally smaller as far fewer memory allocations and arithmetic operations are required per term compared to `meld`.

In many practical and real-world cases, this is not a problem. Many problems in physics rarely require more than ten free indices to be expressed, and Fig. 4.5 demonstrates fifth-order Riemann polynomials can be processed on the order of seconds on home computers which for

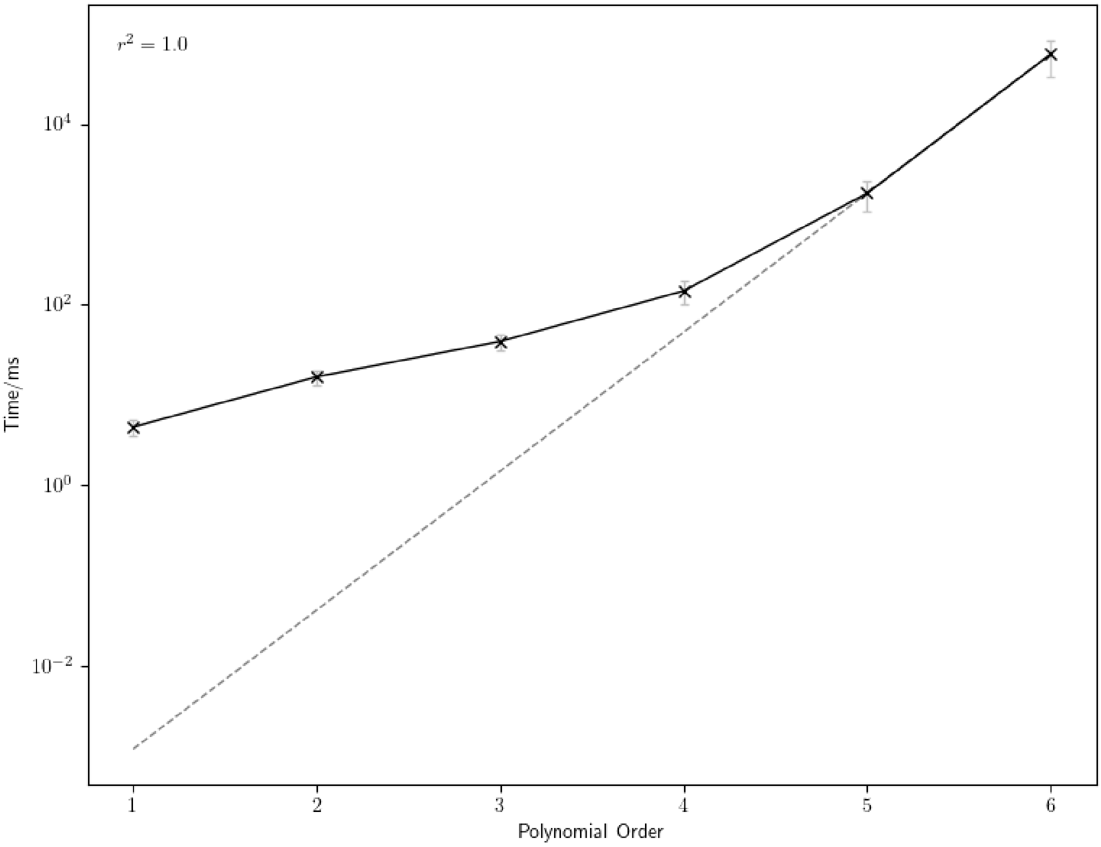


Figure 4.5: Behaviour of meld as the ratio of dummy indices to free indices increases

many applications is perfectly acceptable behaviour. For many classes of calculations, even these limits are far larger than will be encountered meaning that the ability to use `meld` at regular intervals in a calculation to ensure that expressions are reduced to their minimum forms is very viable.

In cases when very large polynomials might be considered, for instance in perturbation theory, the main limiting factor of `meld` is the amount of memory available on the computer. On modern computers, if the physical memory is exhausted then a region of virtual memory on the hard disk is allocated however as `meld` must read concurrently from areas of memory which are fragmented across the memory space[11] the performance suffers considerably due to the high overhead of performing the swap between physical and virtual memory spaces.

[11] The use of a binary search tree means that terms inside a projection are likely to be fragmented, however even in the storage was contiguous solving the linear system requires reading terms from each projection, which would be separated by the size of the projection and therefore likely very far removed in the situation where the size of the projection has exhausted the memory

Chapter 5

Extension to **meld**

As alluded to earlier, canonicalisation through Young projection operators makes up a large part of the overall logic of `meld`, but as an algorithm designed to provide the most general possible tool for simplification there are other routines which it calls. Each sub-algorithm has its own `can_apply` and `apply` methods, which `meld` delegates to; the definition of `meld::can_apply`[1] is

```
bool meld::can_apply(iterator it)
{
    return
        can_apply_diagonals(it) ||
        can_apply_traceless(it) ||
        can_apply_cycle_traces(it) ||
        can_apply_tableaux(it);
}
```

and similarly the `meld::apply` method checks the `can_apply_*` for the particular sub-algorithm before calling it. For completeness, the behaviour of these routines is described here.

5.1 Cyclic traces

The most substantial extra routine handles traces of expressions which are identical up to cyclic permutations of their indices. It is a common identity [86, Chapter 3.9] that

$$\mathrm{Tr}(A_1 A_2 \ldots A_{n-1} A_n) \equiv \mathrm{Tr}(A_2 A_3 \ldots A_n A_1) \equiv \mathrm{Tr}(A_3 A_4 \ldots A_1 A_2) \equiv \ldots \tag{5.1}$$

even if $[A_i, A_j] \neq 0$. If the objects are tensorial, then calculating equivalence using this identity should be done independently of the names of any dummy indices, which is well suited to the adjacency representation. In fact, it is possible to go further and express the problem in terms of a cyclic projection of the indices: given an expression such as

$$\mathrm{Tr}(T_{i_1 i_2} S_{j_1} U_{k_1 k_2 k_3}) \tag{5.2}$$

this can be expressed in an equivalent fashion which makes the cyclic symmetry manifest

$$\frac{1}{3}\mathrm{Tr}(T_{i_1 i_2} S_{j_1} U_{k_1 k_2 k_3} + S_{j_1} U_{k_1 k_2 k_3} T_{i_1 i_2} + U_{k_1 k_2 k_3} T_{i_1 i_2} S_{j_1}) \tag{5.3}$$

which can be represented by two `AdjformEx` objects, one for the indices

[1] Given in `https://github.com/kpeeters/cadabra2/blob/c91481ac0024dd320ccdd94422dd5b3603136908/core/algorithms/meld.cc`

Slot						Weight
0	1	2	3	4	5	
i1	i2	j1	k1	k2	k3	1
j1	k1	k2	k3	i1	i1	1
k1	k2	k3	i1	i2	j1	1

and a second for the object names

Slot			Weight
0	1	2	
T2	S1	U3	1
S1	U3	T2	1
U3	T2	S1	1

where extra information must be kept alongside the name to ensure that it uniquely identifies an index set, to avoid the incorrect matching of e.g. P_iQ_j and P_jQ_i. As the cyclic symmetry has been made manifest, it is now trivial to check if two terms in a trace are identical as they will have matching adjacency projections.

This technique, whilst mathematically pleasing, is more computationally expensive than calculating each cyclic permutation of a term and comparing it to another trace term, as not only does this only require one of the two terms to be cycled, but also allows in the average case for not all cyclic permutations of the other to need be calculated as a match may be found before all permutations have been calculated. In the case where there are more than two terms, the projection method must for each term calculate the projection and then solve a linear system involving all the previous terms as for the Young projection linear problem, with a complexity given by $O(kN + \epsilon kN^2)$ for N terms of k factors. The 'term-by-term' method on the other hand requires for each term to compare against each unseen term for each cyclic permutation, and thus the complexity is given by

$$\begin{aligned}\sum_{i=1}^{N}(k(\alpha+\beta(N-i))) &= \alpha k\sum_{i=1}^{N}1+\beta kN\sum_{i=1}^{N}1-\beta k\sum_{i=1}^{N}i \\ &= \alpha kN+\beta kN^2-\frac{1}{2}\beta kN(N+1) \\ &\propto O(kN+\delta kN^2)\end{aligned} \tag{5.4}$$

i.e. the complexity scales similarly, and the implementation in `meld` therefore takes this approach in the `apply_cycle_traces` function. Similarly to how `apply_tableaux` uses a `ProjectedTerm` class to hold information about each term it processes, the `apply_cycle_traces` uses a `CycledTerm` class

```
struct CycledTerm {
    Ex commuting, noncommuting;
    Adjform indices;
    std::vector<size_t> index_groups;
    Ex::iterator it;
    size_t n_terms;
    bool changed;
}
```

As well as holding `it`, a reference to the object in the original expression, in the same way that `apply_tableaux` splits an expression into scalar and tensor components when considering traces the expression is split into commuting and non-commuting components; only the non-commuting components are cycled as it is always possible to find equality between the commuting parts by sorting the terms in the expression and thus no new simplification can be discovered by cycling these terms. If two terms are discovered to be meldable, then one is removed from the tree and its commuting components combined into the `commuting` expression for the other term and its `changed` flag set. After the algorithm completes, any objects marked as changed have their forms in the original expression updated.

The `indices` adjacency object carries the index structure of the current permutation. After each cycle, which moves the object at the end of the expression to the front, the indices must be cycled by the number of indices held by the last object. To avoid recalculating this each time, during the initial parsing of the expression the number of indices held by each object is calculated and stored in the `index_groups` object and the total number of non-commuting objects in the `n_terms` member. Checking that the number of terms in two expressions is the same is a quick way to exit the algorithm early if there is no match.

The algorithm itself begins by constructing a list of `CycledTerm` objects for the trace object under consideration

```
bool meld::apply_cycle_traces(iterator it)
{
    assert(*it.begin()->name == "\\sum");
    bool applied = false;
    std::vector<CycledTerm> terms;
    for (const auto& term : split_it(it.begin(), "\\sum"))
        terms.emplace_back(term, index_map, kernel);
```

The main part of the algorithm is contained in a nested loop, the outer loop iterating over each term under consideration and the inner loop over all the terms which come after it. As mentioned above, any two terms which have a different number of terms are immediately skipped

```
    for (size_t i = 0; i < terms.size(); ++i) {
        for (size_t j = i + 1; j < terms.size(); ++j) {
            if (terms[i].n_terms != terms[j].n_terms)
                continue;
```

The algorithm then loops over all cyclic permutations of the object, comparing the non-commuting structures and combining if they match and setting the `applied` flag

```
            for (size_t k = 0; k <= terms[j].n_terms; ++k) {
                if (terms[i].compare(kernel, terms[j])) {
                    Ex::iterator head = terms[j].commuting.begin();
                    for (Ex::sibling_iterator beg = head.begin(), end = head.
                        end(); beg != end; ++beg)
                        terms[i].commuting.append_child(terms[i].commuting.
                            begin(), (Ex::iterator)beg);
                    node_zero(terms[j].it);
                    applied = true;
                    terms[i].changed = true;
                    terms.erase(terms.begin() + j);
                    --j;
                    break;
                }
                terms[j].cycle(kernel);
            }
        }
    }
```

Any modified terms are then updated and the `applied` flag returned to indicate if any changes to the tree were made

```
    for (const auto& term : terms) {
        if (term.changed) {
            tr.erase_children(term.it);
            it = tr.replace(term.it, str_node("\\prod"));
            tr.append_child(it, term.commuting.begin());
            tr.append_child(it, term.noncommuting.begin());
            cleanup_dispatch(kernel, tr, it);
        }
    }
    return applied;
}
```

To show how this handles commuting factors, consider the expression

$$\mathrm{Tr}(\alpha u_i u_j v_i v_j + \beta u_k v_i v_k u_i); \tag{5.5}$$

Clearly by naïvely cycling the factors the two expressions will never compare as equal as the α and β factors will never match, however by only considering the non-commuting parts `meld` manages to combine these two terms:

```
{u_{i}, v_{i}}::NonCommuting.
Tr{#}::Trace.
ex := Tr(\alpha u_{i} u_{j} v_{i} v_{j} + \beta u_{k} v_{i} v_{k} u_{i});
meld(ex);
```

$$\mathrm{Tr}\left(\alpha u_{i} u_{j} v_{i} v_{j} + \beta u_{k} v_{i} v_{k} u_{i}\right)$$

$$\mathrm{Tr}\left(\left(\alpha + \beta\right) u_{i} u_{j} v_{i} v_{j}\right)$$

As with the `apply_tableaux` algorithm, `meld` does not try to find a *canonical* order of the trace arguments, and if the order of terms were reversed then the output would be $\mathrm{Tr}\left(\left(\beta + \alpha\right) u_k v_i v_k u_i\right)$ instead .

5.2 Traceless and diagonal objects

Two more types of indexed objects which `meld` deals with are traces of traceless object and non-diagonal terms of diagonal objects. These algorithms are also implemented in the `canonicalise` algorithm and are thus carried over with few changes into `meld` as these simplifications do not benefit from using an adjacency representation as an intermediate data type but do fit into the motivation of `meld`.

As the algorithms are not novel implementations their operation will not be discussed[2] other than to mention that they are the first sub-algorithms to be called from the master `apply`

[2] Although can be viewed at `https://github.com/kpeeters/cadabra2/blob/c91481ac0024dd320ccdd94422dd5b3603136908/core/algorithms/meld.cc#L88` and `https://github.com/kpeeters/cadabra2/blob/c91481ac0024dd320ccdd94422dd5b3603136908/core/algorithms/meld.cc#L119`

method of `meld`. This is because they are far cheaper to run than the `apply_tableaux` method, and as they solely remove terms which are identically zero it is more efficient to call these first and potentially remove terms rather than canonicalise them using `apply_tableaux` only to have them removed.

5.3 Future considerations

Before moving on, it is worth mentioning some of the simplifications which `meld` is currently unable to process but which fit naturally into its purview.

5.3.1 Side relations as projections

One natural question to ask is whether `meld` is also capable of applying side relations as projection operators to find simplifications. For example, given that

$$A \equiv B + C + D \tag{5.6}$$

then this identity could be treated as a projection of A. An expression such as

$$A + B + C + D \tag{5.7}$$

could then be treated in a similar fashion to the `apply_tableaux` algorithm with each term written in terms of the basis $[B, C, D]$ to build up the system of linear equations:

Term Component	A	B	C	D
B	1	1	0	0
C	1	0	1	0
D	1	0	0	1

Attempting to write the last column in terms of the other three leads to a system of linear equations

$$\begin{aligned} 1 \times c_1 + 1 \times c_2 + 0 \times c_3 &= 0 \\ 1 \times c_1 + 0 \times c_2 + 1 \times c_3 &= 0 \\ 1 \times c_1 + 0 \times c_2 + 0 \times c_3 &= 1 \end{aligned} \tag{5.8}$$

which has the solutions

$$\begin{aligned} c_1 &= 1 \\ c_2 &= -1 \\ c_3 &= -1 \end{aligned} \tag{5.9}$$

The expression can therefore be simplified by replacing D with contributions from the other terms

$$D = c_1 A + c_2 B + c_3 C = A - B - C \tag{5.10}$$

and so

$$\begin{aligned} A + B + C + D &\to A + B + C + (A - B - C) \\ &= 2A \end{aligned} \tag{5.11}$$

Of course, in this instance a far easier way to achieve the same goal is to simply swap the sides of (5.6) to convert it into a substitution rule and substitute this into (5.7). However, computing it as a projection makes this a completely general process. The similarity of the process to `apply_tableaux` is evident, however as each term is an expression object rather than an index permutation a new data structure would need to be constructed although the implementation of `meld` as it is would provide most of the tools required and the same general structure of the algorithm could be kept.

In the case where the terms are composed of factors instead of just simple terms, the idea can be extended with the basis extended from $[B, C, D]$ to the complete set of products $[B^2, BC, BD, C^2, CD, D^2]$ (assuming that $[B, C] = [C, D] = [D, B] = 0$). Then each term must be distributed and added to the data structure as before for instance multiplying (5.7) through by A we get

$$A^2 + AB + AC + AD \tag{5.12}$$

which, after decomposing each term according to (5.6), leads to the series of projections

Component \ Term	A^2	AB	AC	AD
B^2	1	1	0	0
BC	2	1	1	0
BD	2	1	0	1
C^2	1	0	1	0
CD	2	0	1	1
D^2	1	0	0	1

with the series of equations

$$\begin{aligned} c_1 + c_2 &= 0 \\ 2c_1 + c_2 + c_3 &= 0 \\ 2c_1 + c_2 &= 1 \\ c_1 + c_3 &= 0 \\ 2c_1 + c_3 &= 1 \\ c_1 &= 1 \end{aligned} \tag{5.13}$$

with the same solution set

$$\begin{aligned} c_1 &= 1 \\ c_2 &= -1 \\ c_3 &= -1 \end{aligned} \tag{5.14}$$

from which the reduction to $2A^2$ is calculated as above.

5.3.2 Dimension dependent identities

As well as index symmetries which can be expressed as Young tableaux, there are also an entire class of symmetries which result from antisymmetrising a rank-n tensor in d dimensions where $d < n$. As there are only d ways of labelling each index, this ensures that each term in the antisymmetrisation is balanced by one of opposite sign which must be labelled in the same way so that the expression is zero:

$$T_{[i_1 i_2 \ldots i_n]} = 0 \tag{5.15}$$

This is most clearly seen for a rank-2 tensor which lives in one dimension, the antisymmetrisation is

$$T_{[ab]} = T_{ab} - T_{ba} \tag{5.16}$$

and as there is only one dimension each index must be labelled with the same coordinate, so the two terms cancel out:

$$T_{xx} - T_{xx} = 0 \tag{5.17}$$

This also applies to any tensor in d dimensions which has a subset of $n > d$ indices in which it is completely antisymmetric as this will lead to the same labelling problem. Identities involving dimension dependence have been known for a long time (at least as far back as 1918 when Weyl uses the identity ${}^*R^i{}_{klm} \equiv 0$ in $n < 4$ dimensions [87]) but it wasn't until 1970 when Lovelock [88] showed how these can be derived as a consequence of antisymmetrisation that a systematic method for verifying and constructing such identities was developed. Another big advancement was the publishing of the paper by Edgar and Hoglund [89] which shows how more identities which were thought to not be a consequence of any algebraic property of the objects could also be derived through antisymmetrisation.

Cadabra allows for index sets to be defined over a set of coordinates or an integer range, which makes simple applications of the principle possible e.g. in the simple case of a completely antisymmetric tensor defined in a dimension which is less than its rank. With the literature now containing general methods for constructing such identities, it is certainly possible to envisage that it would be possible to extend this concept to more complicated situations.

Part II

Software in Academia

Chapter 6

Reusing Algorithms Through Packaging

The discussion thus far has been on algorithms implemented in Cadabra using the underlying C++ codebase. As Cadabra is intended to be used through the Python interface, the C++ layer is mostly hidden to the user although it is possible to build and use Cadabra as a C++ library which may be desirable in some circumstances, for example where it is necessary to interface with a different C/C++ library which does not have Python bindings. This combination of using C++ as a backend language with a interface to Python is used to differing degrees by many other tools used in academic research, such as NumPy [90] and Pillow [91] which are written entirely in C with the classes and functions exposed to Python, as well as SciPy [92] which uses C for some of the more computationally expensive routines but also contains a large proportion of pure Python code. There are also many components of Python's standard library which are written in C, such as the `math` library which simply exposes the functionality of the C `math.h` library [1].

Structuring projects in this manner allows one to take advantage of the strengths of both languages, allowing the native machine-code compiled routines written in C/C++ to run without sacrificing efficiency whilst also allowing them to be accessed from the much simpler and interactive Python environment. The prevalence of Python in the scientific community also means that it provides a common interface from which many different tools can be used and combined, not unlike how programs can be redirected and piped into each other using shell environments like bash; in fact tools such as NumPy are intended for use as components for constructing other packages and can be found in other Python libraries such as the data analysis tool pandas [93] as well as Pillow and SciPy mentioned above. There are also numerous different interactive environments available for Python which Cadabra can interact with, including a kernel for Jupyter notebooks (the interface also used by SageMath) and the option to use IPython or the default Python interpreter.

Cadabra makes extensive use of the reflection abilities of Python as well as a preprocessor which allow it to hide much of the complexity associated with maintaining the state which is required for `Property` declarations to be accessible to algorithms. Several of the components Cadabra's Python interface hides can be seen by comparing a simple example of making a property declaration, constructing an expression and applying an algorithm to it

```
{ D, C, B, A }::SortOrder.
```

[1] `https://docs.python.org/3/library/math.html`

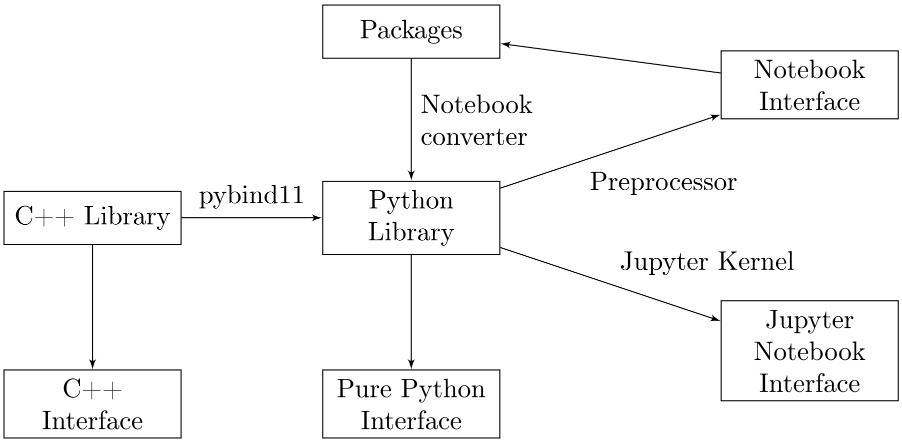


Figure 6.1: Components of Cadabra

```
ex := B A C D.
sort_product(ex); # prints 'DABC'
```

which using the pure C++ interface becomes

```
#include "cadabra2++.hh"

int main()
{
    using namespace cadabra;

    Kernel k;
    k.inject_property(new SortOrder(), k.ex_from_string("{ D, C, B, A }"),
        nullptr);

    auto ex = k.ex_from_string("B A C D");

    sort_product sp(k, *ex);
    sp.apply_generic();

    DisplayTerminal dt(k, *ex);
    dt.output(std::cout); // print 'DABC'
}
```

When using the Python interface knowledge of the `Kernel` object, the need to instantiate algorithms as class instances and use a `Display*` class in order to produce output are all unnecessary. There is an auxiliary `cpplib` namespace (developed over the course of this thesis) which contains some convenience functions to simplify the C++ experience

```
#include <cadabra2++.h>

int main() {
    using namespace cadabra;
    using namespace cadabra::cpplib;

    Kernel k;
    auto ex = "BACD"_ex(k);
    apply_algo<sort_product>(k, ex);
    std::cout << pprint(k, ex) << `\n';
}
```

but this only simplifies the constructions rather that allowing one to be totally agnostic of them. The flexibility of Python is one of the key factors in making Cadabra so powerful and expressive.

The structure does however also come with some drawbacks, the first one which a developer will come across is the added layer of complexity which arises from exposing the C++ interface to Python due to both the interfacing code which must be kept in sync with changes to the underlying library as well as ABI breaking changes between Python versions which can make distribution of binaries more complicated. The challenges to accessibility that result from this will be briefly examined later in section 8.1, however for now we will focus on a different problem which comes about from the difficulties introduced for encouraging the community to contribute improvements and fixes as although users are able to use Cadabra with only knowledge of Python, development requires knowledge of C++ which is less common in the sciences.

Community participation in open source software projects is very valuable, for as well as improving the range of functionality through user submissions, which almost all pieces of software including commercial products make use of, the smaller development teams and more limited time which they have to work with on their projects means that some features are simply infeasible to implement without the added production capability the community brings. It must however be kept in mind that only a small fraction of users will become involved enough in a project to contribute to the codebase, and as even 'small' projects can consist of many thousands of lines of code the barrier to entry is very large. This is of course neglecting the other useful ways in which people can contribute including documentation, writing examples and tutorials, as well as more passive modes of contribution such as submitting bug reports and promoting use of the software amongst their colleagues. The scope of this chapter however is in exploring how the Python import system can be used to make user contributed libraries possible.

6.1 Packages in Cadabra

6.1.1 The Python import system

Pythons allows code from external sources to be used inside a script through the use of its import system. This consists of a two stage process, first searching for the relevant module and then binding it to a name in the user code. The native import system[2] allows both modules written in python and compiled modules compiled written in another language, most commonly C/C++, but other languages which have bindings for the Python C API such as Rust with the rust-cpython project, Go by making use of the magic C* namespace to access the Python C API and even Haskell via C using its foreign function interface.

Python exposes imported modules as module objects, which are the same type regardless of whether the source is a Python object or a binary file, and allows the contents of the module to be accessed as members of the module object. Python also allows the module name to be aliased, or for certain parts of the module to be inserted into the current scope

```python
# Import matplotlib module and the pyplot submodule
import matplotlib.pyplot
# Import the pyplot submodule and assign to the name plt in the current scope.
# The matplotlib module is implicitly run but not bound to any name
import matplotlib.pyplot as plt
# Import all names in the matplotlib.pyplot module into the local scope
```

[2] This discussion is valid for CPython (used by Cadabra), but may be inaccurate for other Python implementations such as PyPy

```python
from matplotlib.pyplot import *
```

Note that the second two import statements skip the first step of loading the module, binding the existing reference to new names in the scope instead of creating new references to the module:

```python
assert matplotlib.pyplot is plt
assert plt.plot is plot
```

This is because after the first import statement an entry for `matplotlib.pyplot` along with all other dependencies it loads is inserted into `sys.modules` and this is searched before any subsequent imports.

Python allows the import behaviour to be customised using the `importlib` library which lets custom import machinery to be developed and ‘hooked’ into the import system. The hooks are made of two parts: finders, which take a fully qualified package name and returns information on how to import the package if it is able to, and loaders which use this information to import the package into the current Python session.

6.1.2 Implementation

As Cadabra scripts and notebooks are not native Python, the import system does not work with Cadabra code by default. However, as it is preprocessed into pure Python, extending the import system to support Cadabra is natural and can take advantage of most of the native mechanisms. Developing this packaging system was an early goal when starting work on this thesis and has been the source of many pieces of auxiliary work related to both expanding the number of algorithms distributed with Cadabra and producing papers on computer algebra techniques for general relativity which will be discussed later in section 6.5.

Packages in Cadabra are implemented as notebooks which provides several benefits; development and testing can be done in a single environment, packages can be written in Cadabra instead of pure Python and documentation can be written using LaTeX cells which is consistent with the documentation of the core algorithms.

The Cadabra startup script `cadabra2_defaults.py` implements a custom finder, a class which derives from `importlib.abc.MetaPathFinder` [3] which takes precedence over the default finder, by being placed before the default finder instance in the `sys.meta_path` variable, and checks if the name to import corresponds to a Cadabra `.cnb` notebook file, which it then preprocesses into a pure Python `.py` file which is returned to the default Python loader.

The relevant method of `MetaPathFinder` is the `find_spec` module which returns a `ModuleSpec` object describing where the module should be loaded from:

```python
@classmethod
def find_spec(cls, fullname, path, target=None):
```

The method first sets the `path` argument to the default system path if not provided, and converts the import target name into a list of its components, so that a directory can be constructed from them:

```python
    if '.' in fullname:
        parents = fullname.split('.')
        name = parents.pop()
    else:
        name = fullname
        parents = []
```

[3] `https://github.com/kpeeters/cadabra2/blob/27ba8de7dff03ab16470ec240f892dd14deb6689/core/cadabra2_defaults.py.in#L36`

Next, each entry in the path variable is checked to see if it contains a file which Cadabra recognises and can convert into a pure Python script:

```python
    for entry in path:
        have_cnb = os.path.isfile(os.path.join(entry, name + ".cnb"))
        have_cdb = os.path.isfile(os.path.join(entry, name + ".cdb"))
        have_ipynb = os.path.isfile(os.path.join(entry, name + ".ipynb"))
        if have_cnb or have_cdb or have_ipynb:
```

The file is created in a directory *cadabra_packages* which contains the pure Python versions of the modules. If the package name contains multiple components then it is placed in a subdirectory corresponding to this name, which must be made if it doesn't exist

```python
            pkg_path = os.path.join(user_config_dir(), "cadabra_packages", *parents)
            # Create the path if it doesnt exist
            if not os.path.exists(pkg_path):
                os.makedirs(pkg_path)
```

The compile_package__ function, defined in the C++ codebase, is then called which performs an appropriate preprocessing to convert from the Cadabra format to pure Python

```python
            if have_cnb:
                compile_package__(os.path.join(entry, name + ".cnb"), os.path.join(pkg_path, name + ".py"))
            elif have_cdb:
                compile_package__(os.path.join(entry, name + ".cdb"), os.path.join(pkg_path, name + ".py"))
            else:
                compile_package__(os.path.join(entry, name + ".ipynb"), os.path.join(pkg_path, name + ".py"))
```

The ModuleSpec object is then created and returned.

```python
            return ModuleSpec(
                fullname,
                SourceFileLoader(fullname, os.path.join(pkg_path, name + ".py")),
                origin=os.path.join(pkg_path, name + ".py"))
```

At the end of the function, if no entry in path contained a suitable module file, None is returned which allows Python to continue through the finders in sys.meta_path in order to locate the requested import

```python
    # Return none if no notebook was found
    return None
```

This allows Python to efficiently handle the actual loading and execution of the package and create a byte compiled .pyc binary. If the time stamp of the preprocessed Python file is newer that that of the original notebook the preprocessing step is skipped and a handle to the existing .py file is returned. As the startup file is also pure Python, this allows Cadabra packages to be imported into any Python environment, not only the Cadabra CLI and notebook interface.

Although being able to write Cadabra packages as notebooks allows for a rapid and intuitive development environment, for efficiency purposes it is sometimes necessary to implement routines in C++. Of course the vast majority of Cadabra's functionality is implemented as the set of core algorithms which are written in C++ but it may be preferable to release some algorithms as parts of packages instead of the core system:

- To replace algorithms which already exist in packages with more performant versions without breaking backwards compatibility

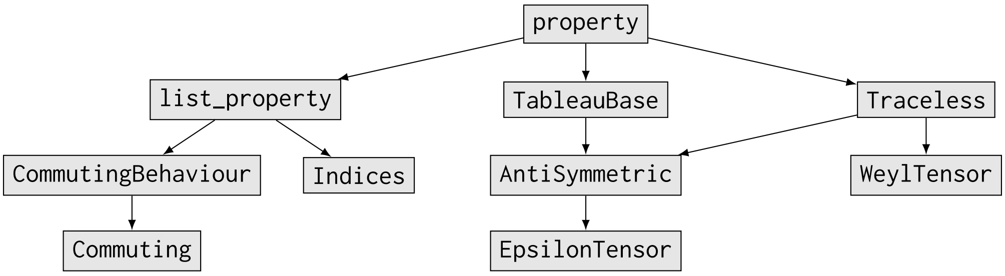


Figure 6.2: Example of some of the inheritance relationships between property types. The full tree has been abridged and many relationships of the classes displayed have been omitted.

- To provide consistent interfaces by releasing packages which expose the same functions but in different contexts, e.g. in the Python standard library the `json` and `pickle` libraries contain functions with the same names but which provide interfaces to different serialization formats
- To provide specialised/niche algorithms with a low demand without cluttering the main namespace

In the current implementation, the build system is only aware of how to build C++ packages which are part of the main set of packages distributed with Cadabra, but the concept is easily extendable and can be modularised into a standalone build system for creating binary packages, although binary incompatibilities between Python versions and even C++ compilers means that distribution has added complexity.

6.2 Interfacing properties in Python

As well as ensuring that the notebook format can be imported, is it also necessary to provide some of the lower level functionality of the C++ library such as traversing the expression tree and accessing methods of the various `Property` classes in order for users to develop more advanced and 'smarter' algorithms. An interface for the expression objects has been previously implemented with the `ExNode` class but a full property interface was still lacking.

Properties in Cadabra are implemented as class instances which are bound to 'patterns', i.e. expressions which may contain wildcard symbols, which define the object type which describe. This relational mapping is stored in the Cadabra kernel, and algorithms which need to be aware of the existence (or lack of existence) of a certain property definition in order to act correctly can query the kernel to see if a particular expression matches a pattern bound to a property of that type. As shown above, when one wants to associate a pattern with a property on the C++ side it is injected into the kernel using the `inject_property` method.

The property classes follow a natural inheritance hierarchy as shown in Fig. 6.2 which serves two major purposes: simplifying the construction of properties by allowing them to inherit behaviour, and allowing algorithms to check for higher-level properties for general behaviours e.g. symmetry or commuting behaviour, without having to know or check for specific implementations.

From the Python interface, the kernel is mostly hidden from the user, with an alternative interface used for declaring properties either using the Cadabra notation

```
expression::Property
```

or by calling the property constructor directly

```
Property(expression)
```

In both cases the kernel is inferred from the current scope rather than explicitly passed. This makes the Cadabra user interface much more intuitive for a user, but means that the Python interface for properties is quite different from the C++ interface. Originally, other than calling the class constructors to register a property there was no way to retrieve properties or query any stored information inside the property. As this ability is useful in a wide range of situations, and in particular for package developers, this functionality has been added to Cadabra (in `core/pythoncdb/py_properties.hh`[4] and `core/pythoncdb/py_properties.cc`[5]) whilst maintaining backwards compatibility to prevent code in existing projects from breaking.

The library used to create the Python bindings is pybind11 [40] which makes use of C++11 templates to hide a lot of the complexity in interfacing types - in particular for our usage the ability to specify parent classes using variadic templates minimises the amount of reduplication of code required to replicate the inheritance relationships of each property. Each Python type is a specialisation of a templated `BoundProperty` class which contains runtime type information about the C++ class for the property it represents as well as inheriting from the `BoundProperty` specialisations corresponding to the parent classes of the C++ property. In order to keep the kernel abstracted away from the user on the Python side, accessing properties from patterns is implemented a a static member function of each `BoundProperty` class.

6.3 Package structure

One of the main motivations for using the notebook format for packages is in order to allow them to be self-documenting through the use of LaTeX cells. As Cadabra already provides a mechanism for exporting notebooks to various different formats including pure LaTeX and HTML, this means that generating the documentation for a particular package can be automated as is the case for the packages included in the standard distribution of Cadabra (discussed below) and which can be found on the main Cadabra homepage[6].

Various special LaTeX commands have been introduced in order to make this as standardised as possible with the aim of allowing a natural structure which makes understanding distributed packages as simple as possible. Packages being with a `package` directive which contains the name of the package as well as a brief description, and below this a more detailed description. The exposed functions or classes of the package are placed below this with three cells devoted to each piece of exposed functionality:

- A LaTeX cell which uses the `algorithm` directive with the signature of the function or class and a description of what it does

- A Python cell with the implementation

- An ‘ignore on import’ Python cell, which the `compile_package__` function ignores but which can be viewed and run inside the notebook, containing some tests to ensure the function behaves correctly and which also provides some examples of how it can be used

[4] `https://github.com/kpeeters/cadabra2/blob/dc44eaaf7c1b5e0e0271bc9cad14f0ccc2ed37f0/core/pythoncdb/py_properties.hh`

[5] `https://github.com/kpeeters/cadabra2/blob/dc44eaaf7c1b5e0e0271bc9cad14f0ccc2ed37f0/core/pythoncdb/py_properties.cc`

[6] See for example `https://cadabra.science/manual/cdb/core/component.html`

If a package needs to run any code automatically on import then it can also contain cells containing this code, as the package is converted to a Python script before import these cells will be run as though they appeared directly inside the imported code.

Although not enforced it is also expected that the majority of algorithms in a package follow the standard Cadabra signature for algorithms by taking an input expression which is modified in-place and also returned by the algorithm.

6.4 The `cdb` library

One of the primary uses of the packaging system is to provide a set of standard packages distributed with Cadabra to extend the functionality with functions and classes which are either do not fit within the scope of the core functionality or are more naturally implemented in pure Python such as functionality which relies on other Python libraries such as SciPy. A direct was follow on from the development of the packaging logic was the construction of a standard library which are now distributed with Cadabra as the `cdb.*` set of packages and cover a range of different areas.

The library is currently split across six modules: `core` for commonly used functionality which isn't built in to Cadabra, `gauge_theory` and `relativity` as proof of concept packages for providing dedicated environments for specific fields, `numeric` for numerically evaluating Cadabra expressions with SymPy, `sympy` for interfacing with the symbolic and in particular differential equation solving capabilities of SymPy, and `utils` for lower level functionality which is useful for people developing packages or manually modifying expression trees.

6.4.1 `cdb.core`

Fundamental operations which do not exist within the core of Cadabra, or which extend some of the core algorithms, are implemented inside the `cdb.core` package. The different algorithms are split among several subpackages with the largest being the `manip` library which provides routines for manipulating equations in common ways such as multiplying through by a given term

```
from cdb.core.manip import multiply_through
multiply_through($x/a + 1 = 0$, $a$);
```

$$x + a = 0$$

isolating a particular sub-expression (if it appears linearly)

```
from cdb.core.manip import isolate
ex := 2a + 1/2 a * k - b = c * a + b * k + a - 4;
isolate(ex, $a$, auto_distribute=True);
```

$$a = (bk - 4 + b)(1 - c)^{-1}$$

and applying an operator through an expression

```
from cdb.core.manip import apply_through
apply_through($\partial{x_{\mu}} = 0$, $\partial{#}$);
```

$$\partial(\partial(x_{\mu})) = \partial(0)$$

along with many other similar routines.

Another major component of the `core` package is the `component` subpackage which provides ways to retrieve components from expressions which are the result of applying the `evaluate` algorithm which explicitly calculates components of a tensor:

```
{t,x}::Coordinate.
{i,j}::Indices(values={t,x}).
ex:= b_{i} = a_{i}.
evaluate(ex, $a_{t}=1, a_{x}=2$, rhsonly=True);
get_component(ex, $x$);
```

$$b_i = \square_i \begin{cases} \square_t = 1 \\ \square_x = 2 \end{cases}$$

$$b_x = 2$$

Another example of this can be found in 2.5.1 where it is used to extract a particular component of the connection and Ricci tensor, and again in 2.5.3 when selecting a particular component of the expression to represent on a 2D plot.

6.4.2 `cdb.numeric` and `cdb.sympy`

The `numeric` package is a wrapper around some of the SymPy functions for evaluating expressions numerically which can be used at later stages of calculations to get concrete figures from calculations, or in conjunction with plotting libraries such as `matplotlib` (which Cadabra notebooks support) to produce plots of field equations.

Cadabra has inbuilt support for converting expressions to SymPy objects using the `SympyBridge` which makes this a very natural route for implementing numeric calculations as SymPy already has numerous methods for substituting values into equations as well as more advanced functions such as numeric integration and differentiation.

As well as the ability to numerically evaluate expressions, SymPy also contains routines for more advanced manipulations of expressions which Cadabra can make use of using the bridge between the two libraries; in particular there are many techniques for dealing with differential equations in SymPy which it does not make sense to implement in Cadabra as they are in general highly non trivial and the large community working on SymPy ensures that the solvers they provide are already of high quality. The `cdb.sympy.solvers` library provides high level wrappers around these so that they work naturally on Cadabra expressions.

Examples of both these libraries being used can be found in 2.5.3 where the second order differential equations for geodesics calculated are solved as a system of first order equations using the `cdb.sympy.solvers.linsolve` function, and the equations are are numerically integrated to produce the data points to be plotted using `cdb.numeric.integrate.integrate_ode`.

6.4.3 `cdb.utils`

The final part of the standard library which contains code worked on as part of this thesis is the `utils` library which is aimed primarily at providing tools for helping people develop other libraries; indeed much of the functionality is used in other places within the other parts of the `cdb.*` libraries.

These include some helper functions to aid with navigating the expression tree, functions to manipulate index names and positions and additional tools to help with timing and testing functions.

6.5 Structuring projects with packages

Although Cadabra can be used for performing short calculations, and even more complicated calculations which fit logically inside a single notebook, it is also possible to perform far more extensive and complicated tasks potentially including several different calculations which need to be processed independently but at the end combined in order to produce a final result. Attempting to perform such a calculation in a single notebook comes with several drawbacks:

1. The notebook can become difficult to keep track of from a logical perspective

2. Different calculations may attempt to define conflicting properties which can cause algorithms to act incorrectly

3. It makes it difficult to extract particular parts of the calculation for reuse if they depend on the particular state of the notebook where they are performed

In order to solve these problems, splitting a calculation up between multiple notebooks is generally a sensible idea however this can make it difficult to combine the calculations at the end without manually copying results from each intermediate result into a final notebook which is more error prone and makes it difficult to rerun a calculation with e.g. different parameters.

The use of packages to structure a research project in order is one potential solution to this problem, and due to the power of the underlying Python which Cadabra runs on it is very easy to abstract certain calculations into packages which can help not only solve specific problems but can be made general and distributed in order to aid other people in their calculations. This is not a new concept, with Mathematica and other CAS programs offering packages hosting a wide variety of user contributed libraries which stem from research projects, however the import system of Python allows a lot more flexibility in the way in which it is possible to implement this.

To showcase the viability of using Cadabra packages to construct a full research project, we have worked on a paper, *Cadabra and Python algorithms in General Relativity and Cosmology II: Gravitational Waves* [7] which uses Cadabra to perform calculations on first and higher order gravitational waves. Perturbative solutions to General Relativity, as mentioned in section 2.5 in relation to the Schwarzschild metric, are very important as they allow us to analytically examine complicated systems to a high degree of accuracy. A review of different approaches taken in perturbative examinations of General Relativity can be found in [94].

As well as displaying some of the interesting ways in which Cadabra can be used in these calculations, the relevant part of the project for the current discussion is the way in which it has been structured using packages to highlight the reusability of much of the code as well as how some of the results from earlier chapters in the paper are used in later chapters by importing results as packages; for instance by using the decompositions of the Riemann tensor, Ricci tensor and Ricci scalar calculated in section 4.4.1 in section 5 in order to calculate the equations for first-order gravitational waves. The paper is designed to be read alongside the set of notebooks which go along with it and which contain the calculations used to generate the results inside the paper.

6.5.1 Header packages

One of the main notebooks which is used throughout the project is the *header* package which we developed in order to provide a set of functions which set up the environment for general relativity to perform further calculations. The main function inside the header package is `init_properties` which inserts all the property declarations for the spacetime and conventions used into the kernel. In order to increase reusability, it takes a set of coordinates and metrics which should be defined:

```
def init_properties(*, coordinates, metrics=[$g_{\mu\nu}$], signature=-1):
```

To use the provided coordinates in a property declaration the pull in syntax `@(...)` is used

```
    @(coordinates)::Coordinate.
    index_list := {\mu,\nu,\rho,\sigma,\alpha,\beta,\gamma,\tau,\chi,\psi,\
        lambda,\lambda#}.
    ncoords = Ex(len(coordinates))
    @(index_list)::Indices(position=independent, values=@(coordinates)).
    @(index_list)::Integer(1..@(ncoords))
```

Next the metrics are defined:

```
    sig = Ex(signature)
    for metric in metrics:
```

As they need to be defined with the `Metric`, `InverseMetric` and `KroneckerDelta` properties several different loops are used, first the indices are lowered

```
        for index in metric.top().indices():
            index.parent_rel = parent_rel_t.sub
        @(metric)::Metric(signature=@(sig)).
```

then raised in a similar manner

```
        for index in metric.top().indices():
            index.parent_rel = parent_rel_t.super
        @(metric)::InverseMetric(signature=@(sig)).
```

before being each in turn lowered and raised with the property definition for a Kronecker delta in between to ensure all index contractions are taken into account:

```
        for index in metric.top().indices():
            index.parent_rel = parent_rel_t.sub
            @(metric)::KroneckerDelta.
            index.parent_rel = parent_rel_t.super
```

Next a series of conventional symbols are inserted into the kernel

```
    i::ImaginaryI.
    \delta{#}::KroneckerDelta.
    \partial{#}::PartialDerivative.
    \nabla{#}::Derivative.
    d{#}::Derivative.
```

The notebooks use a variety of objects some of which are notated using the same symbols which can lead to ambiguities when entering the expressions into Cadabra. We therefore define some symbol names using the `LaTeXForm` property so that we can use unique names inside the code but have them display in a natural fashion in the output

```
    ch{#}::LaTeXForm("\Gamma").
    {rm{#},rc{#},sc}::LaTeXForm("R").
    ei{#}::LaTeXForm("G").
    Lmb{#}::LaTeXForm("\Lambda").
```

Finally we define the symmetries of the objects which will be used in the notebooks

```
        ch^{\rho}_{\mu\nu}::TableauSymmetry(shape={2},indices={1,2}).
        rm^{\rho}_{\sigma\mu\nu}::TableauSymmetry(shape={1,1}, indices={2,3}).
        rc_{\mu\nu}::Symmetric.
        ei_{\mu\nu}::Symmetric.
```

Having such a function means that at the top of every notebook where we perform calculations we can simply import `init_properties` and call it using the set of coordinates pertinent to the calculation ensuring which not only reduces the amount of typing but also reduces errors from having forgotten a particular definition.

At the end of the header package is a series of functions which provide definitions for some of the commonly used objects in terms of more basic objects, e.g.

```
def rs():
    return $sc = g^{\mu\nu} rc_{\mu\nu}$
def ei():
    return $ei_{\mu\nu} = rc_{\mu\nu} - \frac{1}{2} g_{\mu\nu} sc$
```

for the Ricci scalar and Einstein tensor. These are implemented as functions instead of simple variables (e.g. `rs = $sc = g^{\mu\nu} rc_{\mu\nu}$` for two reasons:

- It makes it impossible to accidentally alter the definitions which may cause errors if the original definitions is required later on in the notebook
- They may depend on property definition for symbols such as derivatives which are not available at the time of import as the `init_properties` function will not yet have been called leading to a consistency error being thrown on import.

Although the header packages is specific to the calculations which we were performing, it is possible to write a more general version and it is expected that various templates for these header files will be included in the Cadabra standard library in the future to cover a wide variety of different space-times and conventions.

6.5.2 The `perturbation` package

As our calculations concern various orders of gravitational waves a systematic way of defining perturbed objects at various orders and retrieving specific perturbative orders from an expression was required. Cadabra comes with a system for handling perturbative orders through the use of a `Weight` property representing the order of a symbol. The functions in this package were not written as part of this thesis, but many optimisations and restructurings were performed by us in order to make the package more useful

The first function in the package `defPertSymbol` accepts a symbol and an integer representing a perturbative order and produces a new symbol with the order appended to the symbol with the appropriate Weight property attached and a `LaTeXForm` property so that e.g. `a2_{\mu}` is typeset in the more aesthetic ${}^{(2)}a_{\mu}$. This is used to implement the next function `defPertList` which calls `defPertSymbol` on a symbol for all integers up to a maximum perturbative order.

The perturbation package also defines the `defPertSum` function which expands a symbol out into a sum of perturbative terms up to a particular order, e.g.

$$\Phi_{\mu} = {}^{(0)}\Phi_{\mu} + {}^{(1)}\Phi_{\mu} + {}^{(2)}\Phi_{\mu} + {}^{(3)}\Phi_{\mu} \tag{6.1}$$

This can be used as a substitution rule to expand expressions out into perturbative terms. A special function `subsPertSums` is also provided which performs this substitution and also accepts a maximum perturbative order; after the substitution it additionally distributes the result and detects any terms which combine to have a weight greater than the maximum order and drops these terms.

While far from constituting a complete package for general use, by abstracting away this functionality into a package it again makes the writing of the notebooks for each chapter much easier and more consistent. Providing such packages along with the paper, rather than just using them to generate a notebook of results, also allows readers to experiment with the calculations themselves by defining their own symbols and changing the parameters (such as the maximum perturbation order) which I believe to be an invaluable tool to both students and researchers alike.

6.5.3 Tensor Perturbations

Many of the chapters of the paper rely on using the definitions of tensors and gauges expanded out to a particular perturbative order. The first computational chapter of the paper deals with calculating these expansions and producing a general `perturb` algorithm which computes expansions of an expression in terms of other perturbed quantities.

The notebook is structured in such a way that after calling `init_properties` and computing various perturbative expansions the notebook is in a state where many of the definitions used in the remaining chapters are in memory and ready to be used. This allows the later chapters to simply import the notebook as a first step and then continue to explore different calculations based on these definitions.

The culmination of the perturbation package and the general perturb function presented in the notebook for this chapter will go on to be used as the basis for a perturbation package for Cadabra although the algorithms in their current state are still not general enough to be of use for calculations outside of this specific paper. This does however highlight one of the great merits of releasing all the code used to write a paper: even if it is in its 'raw' form not generally usable it can still provide a very useful springboard for other researchers to start from and potentially expand into a tool which can find many other uses. This is in effect what we as researchers are doing when releasing papers — as well as the final results an important part of any paper is a discussion of the process and method used to arrive at the results and a paper would be scarcely accepted if it omitted this important part of the discussion (perhaps save Conway's famously short paper [95]).

With an increasing number of published papers relying on computational methods to generate their results it is surprising how few also publish the complete code used to generate these results as well as an overview of the techniques used. This is not to say that there aren't plenty of papers published describing new releases in software and how it can be used [96, 97, 98, 99] and it is very encouraging that journals within the physics community are interested in publishing such papers which focus heavily on software development rather than pure physics, however demonstrations of usage often do not push the boundaries of research whereas being able to build upon the code used in a paper which is presenting new advances might provide opportunities for even more rapid advancement in these sectors.

Chapter 7

Cadabra and Teaching Computational Methods

We will now move on to a short aside to make some comments on the role computer algebra takes in the classroom and provide a short overview of work on a pedagogic paper we co-authored which aims to present Cadabra from a tutorial perspective. Something which became immediately obvious during the course of writing this thesis was that whilst Mathematica was known to many people, there was a general lack of awareness amongst many researchers in maths and physics on the wide array of software available for solving more niche problems and I have thus felt it necessary to make some general comments on this — in particular with regards to teaching Cadabra for students taking courses in subjects grounded on tensor field theory.

As computers become more and more prevalent in all areas of research, the need for teaching computational methods is also becoming ever greater. Many university courses nowadays provide courses which give students an introduction to programming, and introduce a variety of techniques related to numerical solutions to problems, however these courses rarely integrate with other module options meaning that there is still a separation between the theoretical understanding of a problem and the ability for a student to calculate the solutions to a particular problem. This is amplified in disciplines which traditionally use specialist languages such as GAP for solving problems, as these are rarely taught to students and are non trivial to learn.

This is not to say that there is no movement inside universities towards increasing the amount of information available to students when it comes to computational solutions and recently the amount of material available for teaching and studying CAS has increased, for instance the manuals developed by Daniel Schroeder who has advocated for more focus on this in the classroom for a long time and who has released freely available textbooks [100, 101] on techniques in various programming languages for students.

Of course this problem is not unique to students, as advances in computational methods occur rapidly and the need to communicate easier and more efficient ways of solving classes of problems among researchers is an essential part of the advancement of scientific ideas. These ideas are communicated in a variety of different ways:

- Journal articles presenting new software systems or packages

- Papers making references and citations to the software systems that were used during the course of their research

- Presentations at scientific and computing conferences
- Discussions between researchers and research groups, whether in person or on scientific forums

In addition to the Cadabra and Gravity paper demonstrating the use of packages and other advanced Cadabra techniques, we have also written another paper *Cadabra and Python algorithms in General Relativity and Cosmology I: Generalities* [102] aimed at introducing some more fundamental ideas about programming in Cadabra and Python using topics in General Relativity which are accessible to physicists at a postgraduate and higher level.

There is no shortage of different tools available for computational solutions to problems in physics, and nor is there a single tool which can be used in every situation. However, when first learning how to translate physics problems from pen-and-paper calculations to a computer assisted solution Cadabra comes through as a natural choice for several reasons:

- Using a mixture of Python and LaTeX the basic syntax should already be familiar to a good majority of students with Python being one of the most popular languages for scientific computing in general, and LaTeX almost ubiquitous in the scientific community for typesetting mathematics
- It is a relatively lightweight CAS tool suitable for home computing compared to many other tools which provide far more complete experiences for advanced use cases but most of which will remain unexplored for a student
- As it is open source there is no burden on the university or student to buy a licence
- The workflow is very natural consisting of a single expression undergoing a series of transformations as one would also act with a pen-and-paper calculation
- Tensor field problems translate verbatim into Cadabra without the need to handle tensor indices in a special manner

Many other tools also share these benefits, e.g SageMath is also based on Python and Redberry is a lightweight tool, and there are of course downsides to using Cadabra too:

- Cross-platform support (in particular Windows support) is limited
- The workflow is different to many other tools as the expression tree Cadabra uses maps very closely to the LaTeX representation of an expression rather than the , so skills in Cadabra do not necessarily translate well to using other CAS tools
- For very computationally intensive calculations it is not as performant as other tools

The main point however remains that there is a very low barrier of entry to using Cadabra and thus even if it does not solve all the problems that an introduction to symbolic algebra requires, giving students a simple way of putting problems from their courses into a compute to explore these situations further is still an invaluable resource for the classroom.

General relativity is an excellent candidate for a subject to introduce ideas in symbolic algebra, and in particular Cadabra, as it is a very well studied area with a large amount of literature in both journals and textbooks which offer opportunities to study problems which are easily solvable by hand as well as to explore more complicated techniques often requiring the calculation of many components of a tensor which is ideally suited to computational techniques.

There is also a large amount of scope for varying the conditions by either changing the space-time metric or the geometry of the situation. Manipulating these ideas provides a mutual benefit of having intuitive and concrete ways of adapting the code as well as also providing more insight into the underlying physics of relativity by exploring different situations. At a more advanced level, it provides a good framework for introducing how a particular CAS tool works as the field is both familiar to many physicists and also utilises many common techniques such as handling metric contractions, using tensor calculus and manipulating differential forms which form the basis for many other fields in physics too.

Much of the original work discussed in the following discussion is not my own, with my contributions mainly consisting of structural changes, optimisations to the code and clarifications in places where the code was unclear however an overview of the paper is provided here as I believe the way the paper is conceived is a useful tool for teaching computational methods in general.

In the *General Relativity and Cosmology I* paper we begin with an introduction to the formalism of general relativity in order to ensure that the reader has at least a fundamental understanding of the physics which is explored throughout the paper. Although the paper still assumes familiarity with general relativity and the underlying mathematics it can often be useful to a reader to being with going over assumed knowledge in order to get used to the conventions and style used by the author when the knowledge about to be used in an unfamiliar context, in this case in a computer algebra program.

The first piece of Cadabra programming to be introduced is a header notebook, similar to the one introduced in chapter 6, which both provides an introduction to how properties and expressions in Cadabra work as well as giving an example of a project structure which uses a shared notebook as a package in order to reduce duplication of code and incompatibilities between notebooks in a project.

The rest of the paper is split amongst small projects which explore different areas of general relativity, each of which highlight a different aspect of coding with Cadabra. The first project is a calculation of the explicit form of the Lanczos-Lovelock Lagrangian. This reinforces the concept of making Cadabra contextually aware of the objects inside expressions by defining properties related to the specific project rather than the more general properties defined in the header file. The chapter begins with a simple calculation of the linear curvature Lagrangian by starting with the general first order expression and applying various algorithms to develop the required form of the expression. It is then noted that the series of algorithms applied can be abstracted away into a Python function demonstrating how a user might begin to build up a collection of generic routines for dealing with expression in a particular project. As the section continues to calculate higher order corrections, the general routine which has been developed is used as a major simplification in the calculations.

The next project is a calculation of the field equations of general relativity from the Einstein-Hilbert action. This is a more involved calculation which is less ‘automatic’ than the previous project in that more thought must be given to the exact process which must be taken rather than developing a single function to solve a problem generally. The chapter walks through the calculation very slowly to ensure that there are no jumps in the physics which might distract the reader from understanding how the computer is being programming to solve the problem.

The final project consists of solving the field equations derived in the previous section for specific space-times. Not only does adding continuity between the sections help a learner follow the concepts more easily, but solving the field equations allows us to explore yet another area of Cadabra’s capabilities; namely component calculations and scalar manipulations.

Finally we have written an appendix which uses the projects in the paper as demonstrations on how to debug and profile Cadabra code. For many projects which beginner to computer algebra will attempt there are usually two scenarios: the problem is either fairly trivial and can be computed almost instantaneously, or is a seemingly innocuous problem but the algorithm used to compute the result has a catastrophic complexity which means it is unlikely to finish within the lifetime of the universe. Being able to predict the running time of a program, finding bottlenecks and optimising the order of steps in an algorithm to minimise the complexity is important and also not often discussed when teaching high level computing such as symbolic algebra software and we therefore considered it an important aspect to include inside the paper.

There are of course many different techniques which can be taken to teach symbolic algebra which depends heavily on the language being taught and the exact field for which it is being taught, but it is perhaps not as openly discussed as it should be and thus opportunities to share insights into how courses have been designed are useful resources. The overview given here of how we have structured our attempt at a tutorial on the use of Cadabra is given in the hopes that it is both useful and adds to the impetus of the discussion of pedagogic techniques.

Chapter 8

Software Development as Research

It is hard to deny the prevalence of computers in modern research and the impact which they have had in allowing us to explore theories through calculations which were simply not possible before. As well as being a useful tool for replacing the need to manipulate expressions by hand and numerical evaluation, the use of computers in research also drives new techniques and allows us to think about and visualise problems in a different way as noted in works such as [103, 104, 105] . This relationship between research and the available tools will be the focus of this chapter.

To a certain degree, this role played by computers is an extension of the notational paradigms which we use to think about and reason with expressions on paper. A classic example is the Leibniz notation of calculus where the object $\frac{df}{dx}$ is deliberately written to look like a fraction although many elementary operations which work for fractions will fail for this object, especially if partial derivatives are considered. However the notation allows us to intuitively think about the derivative as a fraction or ratio which allows us to think of other related concepts such as solving differential equations by separation of variables as multiplying through by the denominator (see e.g. [106] for more discussion on how notational conventions in calculus lead to more natural thinking in different contexts).

Even more fundamental concepts such as symbolic algebra and graph plotting are also merely useful human formalisms of the underlying mathematical logic which allows us to think about, reason and manipulate them in ways which are less natural or not possible without such tools. Similarly, access to computers opens up other ways of thinking about problems which was not feasible before their introductions.

Among the main techniques which the power modern computing has allowed us to explore include:

1. **Fast calculation over different models and configurations**

   Looking at how applying different models to fit types of data, or how different initial conditions and configurations of other parameters change the solution sets to a model, makes checking the validity of different models and configurations against a desired outcome very easy. This can be helpful in numerous ways, especially by providing an easy way to scan different models to see which are viable for a particular problem. This opens up the ‘trial-and-error’ method to a variety of different contexts is an invaluable tool in research which can quickly rule out dead-ends. This is of course still not viable if the number of configurations is insurmountably large but nevertheless opens up this technique for many every-day situations such as regression analysis which were impractical or at least inefficient to examine in this way.

2. **Simulations of complex systems**

   A large class of problems are in general only exactly solvable when there are a small number of entities in the system, in fact generally only *two-body problems* have general closed form solutions with higher n-order problems expected to be non-integrable [107] although solutions have been analytically calculated for some non trivial systems e.g. [108]. Therefore when calculating the dynamics of more complex systems approximations to two body systems are often made by isolating the object whose dynamics are of interest and collapsing all other objects into an overall effect with which it interacts. Computing allows for numerical solutions to arbitrary precision, which means that the individual orbits of entities in an n-body problem can be studied simultaneously and to a much higher degree of precision in general.

3. **Keeping track of large expressions**

   A limiting factor when carrying out calculations by hand is the size of the expression being manipulated, as if they span multiple lines on paper then they quickly become unwieldy and prone to copying or other accidental errors. This is of course a significantly smaller problem for computers as not only are they only limited by their amount of memory which can be made arbitrarily large, but are also very accurate in copying terms and so these large expressions are less prone to error.

   There are several types of problem which can cause expressions with extreme number of terms, two of the most common examples are systems of large numbers of particles, where the number of interaction terms scales quadratically with the number of particles, and perturbative solutions which can have an even worse exponential or factorial scaling with respect to the perturbative order.

4. **Proof by Exhaustion**

   One of the possibilities computational techniques allow is to explicitly check predicates over arbitrarily large sets allowing brute-force style proofs. The scale of these proofs has increased enormously since the first major effort in 1976 by Appel and Haken to prove the Four Colour Theorem [2, 109] which required checking 1482 configurations, through the 1989 proof by Lam and Swiercz to prove the non-existence of $n = 10$ finite projective planes [110] which they estimate to have required checking 2×10^{14} configurations of points and lines, to recent proofs including the Erdős Discrepancy Conjecture for which researchers at the University of Liverpool have solved for $C = 2$ [111] in 2014 and partial results for $C = 3$ [112] in 2015, with the proof of $C = 2$ totalling a file size of over 13GB.

   The whole concept of computer-assisted proofs, and in particular the use of computers in proofs by exhaustions, has come under criticism on multiple occasions for both their inelegance in only being able to state whether a particular result is true or not without prima facie revealing any deeper insight into the problem [113, Chapter 3.5], as well their 'black box' nature which reflects that as these proof are not in any practical sense able to be verified by hand requiring a level of trust in both the computation to be error-free which cannot be guaranteed and the programmer to have not introduced any bugs when writing the program [114, 115]. This second type of trust in the programmer can be seen as an extension of the trust we put in non-computer proofs as the program itself is verifiable by other people, however the trust in the computer to execute the program accurately is

> fundamentally different and in some senses it is not even in principle possible to verify that a computer will execute instructions faithfully, for example by the mechanisms Ken Thompson outlines in his paper *Reflections on trusting trust* [116].

We can of course point to the dedicated symbolic and numeric routines, backed by the large amounts of processing power available on modern hardware, as the essential tools which have been developed to make these techniques possible, however underlying many of these are techniques and tools which are developed not specifically for academic purposes but which serve just as important a role. This includes the development of the REPL (read-eval-print-loop) programming environment [117] which provides quick feedback and makes rerunning calculations with different inputs or parameters trivial and simple graphics interface wrappers which allow the visualisation of simulations. However, even other considerations such as the design of user interfaces can be influential in promoting researchers to be more productive and think about problems in useful ways. Various articles on the design of user-interfaces in academic design with regard to different contexts have been published; these include for dynamic geometry software [118],

It must also be acknowledged that many of these 'side-considerations' can actually pose serious technical challenges and many are not trivial to solve, and although the importance of these tasks has been written about on multiple occasions [119, 120, 121] it is still often not regarded on the same footing as the core scientific functionality. As the relationship between these two aspects is so closely-bound it is, in my opinion, as unuseful to draw such sharp contrast in the worth of one against the other as to argue on which begets which with a chicken and an egg. On the other hand, it must also be considered that such tools are intended to assist in performing other tasks, once a feature has been implemented on it further development can detract from the central focus of the research and so clear design goals should be set.

The implementation of the packaging system from Chapter 6 is an example of a non trivial feature which greatly impacts the usability of Cadabra and the ability of the community to contribute to it. We will now examine some of the non-scientific tools which Cadabra includes which both provide functionality required for Cadabra to run and help make using Cadabra an efficient way to solve problems and collaborate, and then also examine in more detail the specific implementation of some features.

8.1 Cross-platform compatibility and build systems

After reaching a certain critical mass, it becomes important for programming projects to implement a build system to take care of compiling the source files and piecing the various elements together to form a final packaged and self-contained output. The implementation of this is often highly influenced by the main language of the program, for example a Python project may use `setuptools`[1]. Cadabra uses CMake [122], an open-source cross-platform C/C++ build system which is used by many projects.

One of the very early goals when writing this thesis was to port Cadabra, which had previously been available only on Unix like systems, to Windows in order to increase availability as Windows is by far the most dominant operating system at the time of writing.

[1] https://docs.python.org/3/distributing/index.html

8.2 Notebook versioning

All but the most trivial of computing projects quickly find it necessary to use some form of version control system (VCS) in order to make development manageable. There are three main functions which a VCS provides: making experimentation easier by allowing easy reversion to a previous revision, allowing multiple developers to work simultaneously on the same codebase and the documentation of features as they are introduced [123]. The first public attempt to create a VCS was carried out, like many early computing projects, at Bell Labs where in 1972 Marc J. Rochkind wrote a C implementation of his system SCCS (Source Code Control System) [124]. This was a locking based approach where individual files could be checked out by a developer to work on preventing others from making concurrent changes until the file was unlocked; by keeping track of the revisions of each file this technically fulfilled the requirements given above, however the locking-based system did not allow for strict concurrency as although developers could work on separate parts of a codebase at the same time this made it impossible to work on the same part and if a developer forgot to unlock a file or spent a long time making changes then this could hinder the project workflow significantly.

This style of versioning was superseded by CVS (Concurrent Versions System) in 1986 which advanced the technology in two significant ways: it allowed developers to work on their own *branch*, which allowed them to work on the same file concurrently and then later compare and merge the changes, and it also worked on a client-server model which allowed developers to work from remote machine and work on a local branch only synchronizing with the main server when operations such as *pulling* other people's changes or *pushing* one's own local changes were required. Nowadays the most popular VCS is the open-source tool *Git*[2] which was developed in 2005 for developing the Linux kernel by Linus Torvalds as a reaction to the proprietary tool BitKeeper [125].

While these tools are invaluable for collaborative projects and make looking at revisions of plain-text source files very natural, one of the areas in which they are not performant is working with structured and binary files. Notebook files, based on the JSON format, are an example of a structured format which contains much information which is unimportant for most of the tasks commonly required of users of a VCS and thus the pertinent information is hidden amongst visual clutter. In order to provide functionality comparable to that of 'flat' source files therefore an external tool is required. The requirements of the tool as well as its implementation are dependent on the format of the file and therefore generic tools are not generally possible to produce (although attempts such as diffi [126] which can compare across different file formats are proposed); for example while there are many examples of tools which can *diff* (compute the differences between two files) JSON objects taking into account changes which have no real significance such as white space and the ordering of keys, these are still of little value for comparing notebooks where most of the interesting information is stored in long string values containing escaped characters (`\n`, `\t` etc. which represent non-printable characters) and which require a tool able to view deltas ('atoms' representing the individual changes required to transform one sequence into another) at a smaller, more local level.

The popular notebook format Jupyter [127] has a tool *nbdime* [128] for performing versioning related tasks. While the format of Cadabra's `.cnb` notebook files differs from the Jupyter `.ipynb` notebooks making the use of this tool directly for Cadabra impossible and the tool discussed below is designed independently of the nbdime implementation, the functionality and design

[2] Based on data from `https://www.openhub.net/repositories/compare`, accessed July 2021, which reports 72% of its repositories using Git

goals are of course very similar and where choices have differed this will be discussed.

8.2.1 Cadabra notebook format

Cadabra notebooks files are JSON documents representing the notebooks structure as a tree. Each cell is represented by a JSON object with some keys:

`cell_id` A unique 64-bit number identifying the cell

`cell_origin` String value of either `client` or `server`, detailing where the cell is constructed

`cell_type` String providing information on the type of data contained in the cell

`hidden`† Boolean option specifying whether the frontend should display the cell

`source` String contents of the cell

`cells`† Array of all direct children of this cell

The entries marked † are not required to be present. The tree starts at the `cells` entry of the root object element which also contains other metadata not pertinent to the discussion.

8.2.2 Diffing algorithm

The mechanism used for creating a diff between two Cadabra notebooks which we have developed is a two step process: the first to calculate where cells have been inserted and deleted, and the second to calculate differences between the contents of cells mutual to both notebooks. The algorithm only considers input cells (i.e. the raw LaTeX and Python code written by the user) and not the contents of the output. The primary reason for this is that the output is very commonly not of interest when examining the differences between notebooks, as for a deterministic notebook the output is directly dependent only on the input, and for non-deterministic notebooks, for instance making use of random number generators, the difference in the output is a by-product of random number generation and therefore not revealing of anything useful. This is not to say that there aren't occasions where it may be useful to compare outputs, but in order to comply with the earlier rule of not letting the development of auxiliary tools take over from the main focus of writing research code it was decided to omit for now the extra development and complexity this would entail.

Both steps use the same algorithm for calculating deltas between two sequences which is implemented in the `difflib.h` library[3] written specifically for Cadabra's versioning implementation, and which is an implementation based on the Python `difflib` library. There are other external C++ libraries which can perform sequence matching (for example `dtl`[4] and `diff-match-patch`[5]) but these fail to meet the design criteria, in particular not to introduce any heavy external dependencies (such as diff-match-patch) and to produce the same output as the Python `difflib` library to allow for better integration with Python code. Of course, of the many factors such as complexity, stability and integrability the most important factor is the visual appeal of the resulting diff; as this is generally dependent on the content being matched as well as personal preferences, the use of the algorithm used by Python for comparing Cadabra

[3] `https://github.com/kpeeters/cadabra2/blob/547d2efabccb79c66c07140a2108380dff8d3cb3/libs/internal/include/internal/difflib.h`

[4] `https://github.com/cubicdaiya/dtl`

[5] `https://github.com/google/diff-match-patch`

source code, itself very similar to Python, seemed a suitable compromise (c.f. the study conducted in [129], where the target language of Java which uses braces to in many places where Python uses white space forces different design requirements).

The algorithm (also described in the Python documentation[6]) is based on a matching algorithm known as *Gestalt Pattern Matching* first developed by Ratcliff and Obershelp (by whose names it also commonly referenced) [130]. This algorithm is centred around finding the longest matching subsequences between two sequences and then applying the same logic recursively to the parts of the sequence on either side in order to try and preserve as many unbroken subsequences as possible, although this does not produce the minimal set of deltas required to transform one sequence into the other.

The implementation centres around two classes: `SequenceMatcher` and `Differ`, both of which are templated on a 'string type', although this might be any type of sequence supporting the string interface. The `SequenceMatcher` class's main purpose is the `get_opcodes` method, which produces a set of `OpCode` objects

```
struct OpCode {
    tag_t tag;
    size_t i1, i2, j1, j2;
};
```

which describe how to transform one sequence into another by application of the `tag_t` tags `t_replace`, `t_delete`, `t_insert` and `t_equal` applied to the subsequences in the range $[i_1, i_2)$ for the first sequence and $[j_1, j_2)$ for the second. Therefore, for the strings `"good morning, C++"` and `"a good day, C"` the following opcodes `(tag, (i1, i2), (j1, j2))` are produced:

```
INPUT                                     "good␣morning,␣C++"
(t_insert,   (0, 0),    (0, 2)  )         "a␣good␣morning,␣C++"
(t_equal,    (0, 5),    (2, 7)  )         "a␣good␣morning,␣C++"
(t_replace,  (5, 12),   (7, 10) )         "a␣good␣morningday,␣ C++"
(t_equal,    (12, 15),  (10, 13))         "a␣good␣day,␣C++"
(t_delete,   (15, 17),  (13, 13))         "a␣good␣day,␣C++"
OUTPUT                                    "a␣good␣day,␣C"
```

and where the effect of the operations is displayed on the right hand side. As can be seen, the inclusion of equal subsequences allows the entire string to be constructed from the opcodes.

The `Differ` class acts one level above `SequenceMatcher`, with the main method `get_deltas` accepting two lists of strings and returning a list of `Delta` objects

```
struct Delta {
    tag_t tag;
    string_type a, b;
    std::vector<OpCode> opcodes;
};
```

`get_deltas` examines each string in the first input in turn, comparing it against each string in the second input to calculate a similarity score in the range $[0, 1]$, also known as the ratio between the two strings. If the score is above a supplied threshold, a `t_replace` (or `t_equal` if the strings are identical) `Delta` object is returned with the series of opcodes produced by a `SequenceMatcher` instance called with these two strings. Any strings appearing earlier in the second list generate a `t_insert` `Delta` object, and if no string similar enough is found a `t_delete` `Delta` object is generated.

The first step in the calculation of the notebook diff makes use of the `Differ` class with the two lists containing a list of the IDs of the input cells of each notebook and the threshold set to

[6] `https://docs.python.org/3/library/difflib.html`

1, as we are only interested in exact matches. The list of `Delta` objects this produces enables us to identify which cells have been inserted, deleted and retained. The output of this is an ordered list of changes as shown in the example below which enumerates the differences between the two cell lists in notebooks A and B, where sequential numbers representing creation order are used instead of UUIDs for clarity and greyed out cells are inserted for alignment:

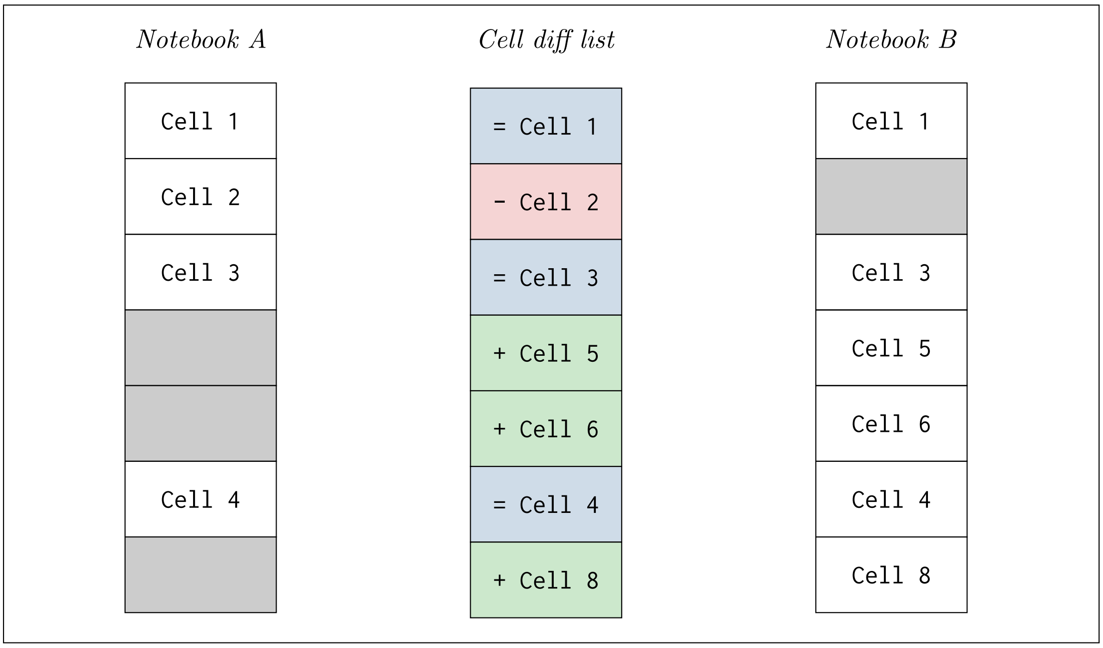


Figure 8.1: Illustration of how a list of diffs can be constructed from two lists of cells

The final step is to compare the contents of cells which appear in both notebooks. For this, the string contents of each cell is split into a list of lines and a `Differ` instance created to compare these two lists. For this, the value of the `threshold` is set to a value < 1 in order to allow lines to be marked with the `t_replace` tag instead of always a pair of `t_delete` and `t_insert` tags.

The effect of different thresholds can be seen in Fig. 8.2 where a code snippet where a piece of code has been refactored into a function is shown. A value which is able to accurately guess whether a line has been modified or replaced can not only be subjective but also depend on the use case; for the Cadabra diff tool a cut-off of 0.75 has been used. It would be possible to develop other metrics for determining the similarity between two lines by using semantic information about a line, for example if we consider trying to find a match for the string

```
message = "Hello, world"
```

in the lines

```
print_message("Hello, world")
msg = "Goodbye, world!"
```

then whereas the first line has a roughly 10% higher ratio with the input, the semantic information of the brackets and operators might encourage us to choose the second line as the match.

a.cdb

```
ex:=0.
for i in range(3):
  l=Ex(i)
  ex=ex+$k_{@(l)}$
ex;
```

b.cdb

```
def make_series(x, n):
  ex:=0.
  for i in range(n):
    l=Ex(i)
    ex=ex+$@(x)_{@(l)}$
  return ex
```

cutoff = 0.1

```
+ def␣make_series(x,␣n):
- ex:=0.
+ ␣␣ex:=0.
? ++

- for␣i␣in␣range(3):
?                ^

+   for␣i␣in␣range(n):
? ++               ^

- ␣␣l=Ex(i)
+ ␣␣␣␣l=Ex(i)
? ++

- ␣␣ex=ex+$k_{@(l)}$
?          ^

+ ␣␣␣␣ex=ex+$@(x)_{@(l)}$
? ++         ^^^^

- ex;
?   -

+ ␣␣return ex
? +++++++++
```

cutoff = 0.5

```
+ def␣make_series(x,␣n):
- ex:=0.
+ ␣␣ex:=0.
? ++

- for␣i␣in␣range(3):
?                ^

+ ␣␣for␣i␣in␣range(n):
? ++               ^

- ␣␣l=Ex(i)
+ ␣␣␣␣l=Ex(i)
? ++

- ␣␣ex=ex+$k_{@(l)}$
?          ^

+ ␣␣␣␣ex=ex+$@(x)_{@(l)}$
? ++         ^^^^

- ex;
+ ␣␣return ex
```

cutoff = 0.9

```
- ex:=0.
- for␣i␣in␣range(3):
+ def make_series(x, n):
+ ␣␣ex:=0.
+ ␣␣for i in range(n):
- ␣␣l=Ex(i)
+ ␣␣␣␣l=Ex(i)
? ++

- ␣␣ex=ex+$k_{@(l)}$
- ex;
+ ␣␣␣␣ex=ex+$@(x)_{@(l)}$
+ ␣␣return ex
```

Figure 8.2: Illustration of how the cut-off ratio changes the granularity of the resulting diff using some Cadabra code as an example.

8.2.3 Tool integration

We have developed three interfaces to the diffing functionality in order to ensure the tool can be used for a range of purposes. The primary tool is a GUI window integrated into the main Cadabra program which provides the two versions side-by-side with differences highlighted using red and green highlighting to indicate deletions and insertions. This can also be supplied with a Git executable from which it can automatically compare the current version of the file being edited to a previous commit.

The other two interfaces are included in an auxiliary package `cdb-nbtool` and provide a CLI interface which produces coloured output for the comparison between two files. Instead of calling the Git directly to produce a diff between two commits, instead `cdb-nbtool` can be passed an option allowing it to be called directly from the Git `diff` subcommand by setting up the Git configuration files.

8.2.4 Considerations for future development

The current implementation provides the core functionality required by a diffing tool, but there are further features which can be implemented. As 'non-core' features however, it must be stressed that particularly in an academic piece of software although there are *possible* routes for extension, actually implementing them should be primarily driven by need rather than convenience.

One large aspect of a notebook which is ignored by the current tool is the contents of the output cells. For the reasons described earlier there is often no relevant information contained in the outputs which doesn't derive from differences in the inputs, but this is not true of situations where data is read from an external source. In these cases, although the external data can also be diffed seeing its impact on the output of a calculation might be more useful.

There are two primary concerns when extending the tool to deal with outputs. The first of these is that the outputs might be in different formats e.g. LaTeX, textual output or a binary image format. These all require different approaches for comparison which may be significantly more complex than the text diffing algorithm described above and even more suited to specific needs.

Secondly, each time a notebook is rerun the output cells are destroyed and recreated with the output of the latest run. This means that differences in the structure of the output cells cannot be detected by using cell IDs to identify them. This problem is linked to a similar issue where, either by splitting or moving into a different location in the notebooks, a cell may have its ID changed even though it is 'semantically' the same cell. The primary solution for this problem is to create lists of cell IDs which are not common to both versions of the notebook. The method illustrated in Fig. 8.1 can then be used with the contents of the cell replacing the cell ID and the threshold set to some value < 1 which can be determined empirically. This is one method of heuristically determining whether two cells should be compared.

8.3 Creating a debugging environment

A common need amongst all programming environments is a need for a set of tools to debug the code which we write. Programming in an academic situation is no different, and the ability to examine calculations at every stage and perform introspection of what is going on inside more complicated routines is important for the workflow of an academic researcher.

Python provides a lot of tools for debugging with the `pdb` providing a very powerful interface for stepping through code at a very low level to understand exactly what the state of a program is as each instruction is performed. From the perspective of a user of Cadabra however this is probably too fine grained in detail as it is less often the actual underlying code which we wish to debug so much as how the expression we are working on is being changed at each step. Whilst the notebook interface provides a very simple way to structure and run calculations it is not an ideal interface for debugging as it is awkward to see the output of single commands, view intermediate results in a cell and as the output is formatted in LaTeX large amounts of output (which are often produced in the middle of a calculation before any canonicalisation, factoring or other simplifications have taken place) can take up large amounts of space, be difficult to suppress and increase the compilation time reducing the efficiency of the workflow.

It is far easier to perform these sorts of debugging steps in a REPL environment, such as the Cadabra CLI, however breaking up the workflow between both the notebook interface and the command line interface is awkward and in cases where a bug in a calculation is caused by the current state of the notebook it can be difficult to reproduce the error in the command line environment.

As a way of providing more sophisticated debugging tools in the notebook interface, a console which shares state with the kernel of the notebook but allows for a interactive REPL environment has been developed. As well as providing a way to quickly test out the result of certain expressions in the current state of the notebook it also allows redirecting output from a notebook cell to the console which mitigate the problems mentioned above and has the added benefit of providing a clear visual distinction between the debugging output and the output expected from the cell.

The notebook interface is implemented as a client-server: the GUI the user interacts with is a client instance which spawns a server which can be sent Cadabra code to interpret and return the result to the client to display back to the user. Each code block which is sent to the server is assigned a cell ID so that the data returned can be identified. Outputs are returned with their own cell ID as well as their parents with the expectation that the GUI will produce a corresponding visual cell attached to the input cell. In some cases output cells may also spawn children, i.e. LaTeX output will have a child containing the plaintext Cadabra input form so that the result of the output can be copied and pasted into an input cell.

As the console is designed to live 'outside' of the cell-like interface it does not fit naturally into this model and the outputs from the server requested by the console need to be handled separately. In order to do this, the console registers with the client a special 'interactive' cell ID through which it makes all its requests to the server. The client is then able to recognise any messages which return with this ID and also needs to keep track of the IDs of the returned cells in case more outputs are spawned from these. Instead of creating visual cells to display the output, the console keeps track of various marks inside the text buffer it uses which are labelled with the cell IDs allowing it to asynchronously insert outputs into the console below the input they were generated from asynchronously to any other inputs it might send.

The Cadabra script run at the startup of the notebook interface also defines a `console` object with a `log` method which can be called from within cells inside the notebook which sends a message to the server requesting the return ID to be the interactive cell ID providing a way of pushing output to the console from cells.

8.4 GUI Elements

The design of a user interface, in particular a graphical one, contains many different components which are designed to hopefully make a program intuitive, productive, responsive and discoverable for a user. Although not a primary focus of my work, contributing towards improving the GUI of Cadabra in these respects was nevertheless an important part of improving the software as a whole. The two main elements which were contributed were a hook to the cell buffers to implement syntax highlighting which can make common syntax errors stand out and help quickly identify the role of different elements of the code, and a hook between the algorithms and the status bar of the program to display information about the current progress which can make long running algorithms (such as `meld`) more responsive by providing feedback to the user.

8.4.1 Syntax highlighting

GTK, the GUI framework used to implement the notebook interface for Cadabra, supports an external library `gtksourceview`[7] for text buffers with syntax highlighting and many other code editing features. It was decided to not use this library to implement the syntax highlighting feature in Cadabra for a few reasons:

[7] `https://wiki.gnome.org/Projects/GtkSourceView`

1. It is a large extra dependency of which highlighting is only a small part

2. Although designed on top of the GTK toolkit it was not at the time of consideration fully cross-platform compatible

3. The Cadabra syntax is not fully compatible with Python's and so it would have been necessary to construct a lexer for Cadabra so that it would not be a drop-in replacement for the current text buffers

For more complicated languages such as C++ implementing a syntax highlighter would be very complex and in general would require a way to introspect the code using a parser such as LLVM, but for the Python/Cadabra and LaTeX syntax found within a Cadabra notebook it can be implemented in a single pass state machine. We have therefore developed a syntax highlighting algorithm for the Cadabra notebook frontend. The mechanism for parsing and highlighting the LaTeX cells is outlined here, a similar technique is used for the Cadabra code although there are more token types increasing the complexity slightly.

In the implementation, there are five separate states which need to be highlighted differently:

1. Normal text

2. Commands (words beginning with a backslash)

3. Parameters (following a command and enclosed in curly or square brackets)

4. Maths (enclosed inside dollar signs)

5. Comments (beginning with a percent sign until the end of a line)

The parser starts in normal mode and scans each character in turn. When it encounters a character which begins a new state, it marks the position of the character as the start of the region to be highlighted and updates an internal variable to indicate that it is in a new state. It then continues through the buffer character by character until encountering a character which indicates it needs to change state again. At this stage it applies the highlighting between the marked character and the current position, replaced the mark with a pointer to the current character and changes its internal state again. Fig. 8.3 gives an overview of the five different states the parser highlights and the characters which cause transitions between these states.

There are two extra cases which must also be taken into account related to highlighting parameters. The first is that inside a parameter there may be nesting of brackets, in this case instead of the first closing bracket to cause a state transition the parser should wait until all internal brackets have been closed. For this the parser holds counters containing the current bracket depth which is increased when an opening bracket is encountered and decremented when a closing bracket is found; only when the depth returns to zero is a state change triggered. The second is that immediately after a parameter another opening bracket will start a new parameter however this should not occur at any arbitrary point in normal mode and so an extra state to expect a parameter is required which returns to parameter mode if an opening bracket is encountered but returns to normal mode on encountering any other character.

Of course the number of things which can be done with a simple one-pass state machine such as this is limited, but for the purposes of simple highlighting these machines are efficient and perform well .

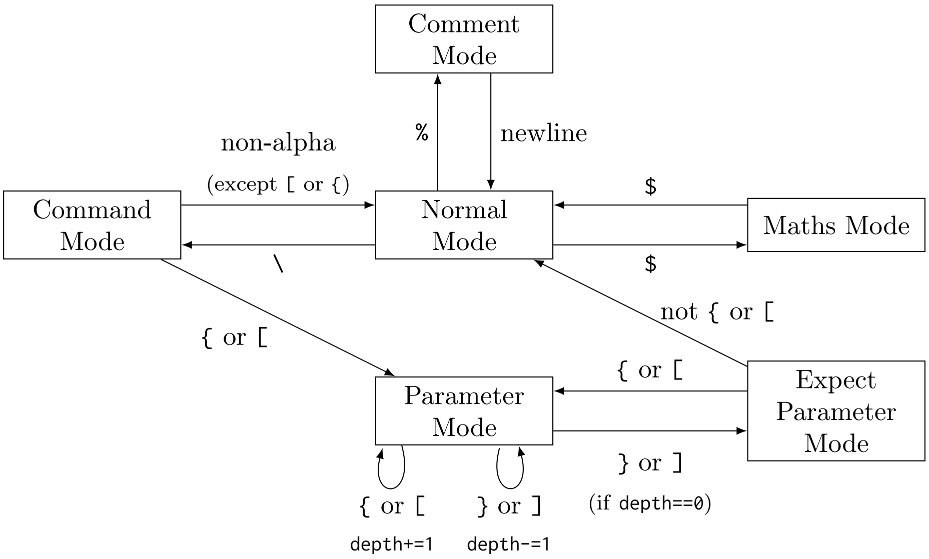


Figure 8.3: Flowchart of different states the LaTeX parser for syntax highlighting can be in and the transitions between states

8.4.2 Progress information

Long running algorithms can be a cause of uncertainty for users as it can be unclear whether an algorithm is working as intended, has entered an infinite loop or whether Cadabra has simply stopped responding. However reporting progressing from an algorithm to the GUI is complicated by the separation of the client and the server. The implementation of an extension to Cadabra which allows asynchronous conversation between the client and server allowing the frontend to dynamically inform the user of the current state of the underlying Python process was developed after we introduced the `meld` algorithm which is an example of a potentially long-running process.

For each kernel instance, a single `ProgressMonitor` object is constructed to which all progress information is reported. The object is created inside of Python and can be fetched from the C++ codebase in order for algorithms to talk to it. The `ProgressMonitor` handles a stack of activities and reports the progress of the highest activity on the stack; every time an algorithm is invocated new activity is added to the stack but algorithms (or other code) can add custom activities to give a more informative picture of what is going on. Each activity can assign a total number of tasks which it must perform and can push updates to report how many of the tasks which it has completed, this is presented to the user as a progress bar in the GUI.

In order for the progress monitor to report the progress to the client it can be provided with a callback function which is called every time some activity or progress is pushed to it. When Cadabra is spawned from the GUI client this callback is set to a Python function which causes the server to send a `progress_update` message which the client reads and uses to update the progress bar.

Chapter 9

Conclusion

Computing has become a core tool for almost everyone in scientific research, and has both rapidly increased the speed and efficiency at which we work as well as opening doors to new possibilities for broadening our understanding of many different fields. The ability for us to move at such a fast pace is due not only to advances in hardware which make computationally expensive calculations possible on even modest home hardware, but also advances in our ability to make tools which open up the world of computing to a wider audience. Developments in our understanding of how to design tools to make different types of workflow easier, such as notebooks, versioning control and interactive prompts, has opened up different channels for us to communicate scientific ideas by sharing programs which allow us to explore calculations in a freer way by replacing calculations which are tedious by hand with algorithms which can swiftly perform transformations to expressions. One of the most important facets of modern academic computing is the relation between the power of software to perform calculations and to communicate these ideas, but this link is easily overlooked resulting in reduplication of code and ideas, multiple competing libraries performing the same tasks and a lack of transparency of the code used to generate results.

In this thesis I have aimed to present as many different aspects of the software development cycle as possible in order to demonstrate the vast number of considerations that go into functionality we mainly take for granted in most of the applications we use. Many of these appear to be trivial problems on the surface, but have been developed slowly and with careful balancing of motivations which I believe makes the study of their history, usage and lower level details essential when it comes to writing any form of software; however instances of this being mentioned in relation to academic software are vanishingly few.

There are a variety of skills which go into writing software which are not directly related to scientific research or the computing skills which many researchers develop when using academic software, but which are nonetheless essential for the developers of this software; for example writing GUI interfaces, threading and concurrency, ideas related to different data structures, resource management and build systems to name but a few. Learning these skills is often not recognised within the academic community when it can in fact, as I hope has been displayed here, constitute a sizeable proportion of a project.

The first part of this thesis concentrated on what would at first sight seem to be the core aim of anyone writing software for research: constructing algorithms to solve and facilitate solving equations which arise in physics and mathematics. The `meld` algorithm in particular provides a different perspective on what the aim should actually be when 'canonicalising' an expression with the standard algorithms for many years attempting to turn each individual term into a canonical form, as opposed to `meld` which attempts to canonicalise on a more zoomed out level, considering the expression as a whole. This allows a more general approach, implemented in `meld` using the Young projection operator, to finding identical objects which means that general expressions with multi-term symmetries can be reduced to their smallest basis of independent objects.

The reuse of code is one of the central factors I have attempted to highlight throughout this thesis, and the packaging system developed for Cadabra allows the open source community to easily create and share libraries and contribute to the set of packages which make up the Cadabra 'standard library'. The use and utility of packages in writing papers and sharing code was displayed through the set of gravity papers [102, 7] which not only rely heavily on the packaging system to provide a consistent interface across the notebooks, but the development of these papers also inspired many very generic algorithms to be included into the standard library. This is a very good showcase of how open source projects provide a springboard whereupon developers and users can mutually benefit from each other in order to make not only the software more comprehensive but also allow for rapid feedback and developing which allows users to experiment with features quickly.

The discussion on the gravity papers also highlights the importance of being able to communicate through notebooks and how important it is to provide pedagogic materials for people to show them how to effectively use CAS in order to solve problems in their fields. Notebooks are one of the most natural ways in which we can communicate ideas using CAS as they emulate the physical idea of performing a calculating and annotating the ideas and concepts placing different stages into different logical sections. CAS notebooks have the major advantage over hard-copy papers of allowing viewers to modify and adapt parameters or forms of expressions in order to gain a better insight into the calculation without having to manually go through the algebraic manipulations. This concept is well presented in the second gravity paper, where variables such as the perturbation order are defined in a way which makes them easily modifiable to view the result of the calculation at different orders.

The second part also covers many of the other considerations which went into the development of Cadabra which make it easier and more efficient to use. The two most prominent of these are the development of the versioning control system for notebooks and the interactive console. Neither concept is original, even within academic software, however both can be invaluable tools at different stages when attempting to solve a problem using Cadabra, and the ideas have not (as far as I have been able to ascertain) been presented from the viewpoint of an academic writer before. For fear of having repeated the point too many times before, such ideas often go unnoticed in the broader context of academia and I therefore hope to have provided a starting point from which other developers, who are specifically writing for academic audiences, can turn to when approaching these tools.

The development of any piece of software is a multi-faceted project, and the development of a piece of *academic* software perhaps even more so due to the amount of knowledge of different fields required to build tools for researchers studying in many varied and niche areas. If I were to make any broad comment on what this thesis is about, and what I hope the reader will be able to have gained from it, then it is exactly this lack of a single overarching umbrella under which the task of writing academic software can be placed; indeed, far from being an 'edge case' the field stretches throughout and finds both inspiration and utility in all the scientific disciplines, natural, mathematical and computational.

Bibliography

[1] Michael Clarke. “Academic and Professional Publishing”. In: ed. by Robert Campbell, Ed Pentz, and Ian Borthwick. 1st Edition. Chandos Publishing, 2012. Chap. 4, pp. 79–98. ISBN: 9781780633091.

[2] K. I. Appel. “The Use of the Computer in the Proof of the Four Color Theorem”. In: *Proceedings of the American Philosophical Society* 128 (1984), pp. 35–39. ISSN: 0003049X. URL: http://www.jstor.org/stable/986491.

[3] *The GNU Operating System.* Web Page. URL: www.gnu.org/.

[4] Richard Stallman. *Free Software Foundation.* Web Page. URL: https://www.fsf.org.

[5] Martinus J. G. Veltman. *From Weak Interaction to Gravitation.* Generic. 1999. URL: https://www.nobelprize.org/uploads/2018/06/veltman-lecture.pdf.

[6] J. Ihm. “Total energy calculations in solid state physics”. In: *Reports on Progress in Physics* 51.1 (1988), pp. 105–142. ISSN: 0034-4885 1361-6633. DOI: 10.1088/0034-4885/51/1/003. URL: http://dx.doi.org/10.1088/0034-4885/51/1/003.

[7] O. Castillo-Felisola, D. T. Price, and M. Scomparin. “Cadabra and Python algorithms in General Relativity and Cosmology II: Gravitational Waves”. In: *Awaiting Publishing* (2021). DOI: 10.48550. URL: https://arxiv.org/abs/2210.00007.

[8] Carlos Palenzuela. “Introduction to Numerical Relativity”. In: *Frontiers in Astronomy and Space Sciences* (2020). DOI: 10.3389/fspas.2020.00058. URL: arXiv:2008.12931.

[9] M. Kreuzer and H. Skarke. “Complete classification of reflexive polyhedra in four-dimensions”. In: *Advances in Theoretical and Mathematical Physics* 4 (2000), pp. 1209–1230. DOI: 10.4310/ATMP.2000.v4.n6.a2. URL: https://arxiv.org/abs/hep-th/0002240.

[10] J. Carifio et al. “Vacuum Selection from Cosmology on Network of String Geometries”. In: *Physics Review Letters* 121 (2017), p. 101602. DOI: 10.1103/PhysRevLett.121.101602. URL: https://arxiv.org/abs/1711.06685.

[11] YH. He. “Machine-learning the string landscape”. In: *Physics Letters B* 774 (2017), pp. 564–568. ISSN: 0370-2693. DOI: 10.1016/j.physletb.2017.10.024. URL: https://arxiv.org/abs/1706.02714.

[12] Wolfram Research. *Mathematica.* Computer Program. 2020. URL: https://www.wolfram.com/wolfram-alpha-notebook-edition.

[13] *SageMath, the Sage Mathematics Software System.* Computer Program. 2021. URL: https://www.sagemath.org.

[14] *Maple.* Computer Program. 2021. URL: https://www.maplesoft.com.

[15] *Maxima, a Computer Algebra System.* Computer Program. 2021. URL: https://maxima.sourceforge.io.

[16] K. Peeters. "Introducing Cadabra: a symbolic computer algebra system for field theory problems". In: *arXiv* (2007), hep–th/0701238.

[17] K. Peeters. "Cadabra2: computer algebra for field theory revisited". In: *Journal of Open Source Software* 3.32 (2018), p. 1118. DOI: 10.21105/joss.01118.

[18] "GraviPy, Tensor Calculus Package for General Relativity". In: (2021). URL: https://github.com/wojciechczaja/GraviPy.

[19] Carl Engelman. *The legacy of MATHLAB 68*. Conference Paper. 1971. DOI: 10.1145/800204.806265. URL: https://doi.org/10.1145/800204.806265.

[20] Joel Moses. "Macsyma: A personal history". In: *Journal of Symbolic Computation* 47.2 (2012), pp. 123–130. ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2010.08.018. URL: https://www.sciencedirect.com/science/article/pii/S0747717110001483.

[21] Stefan Weinzierl. "Computer Algebra in Particle Physics". In: *arXiv* (2002). URL: arXiv:hep-ph/0209234.

[22] R. G. Tobey. "Experience with FORMAC Algorithm Design". In: *Communications of the ACM* 9 (1966), pp. 589–597. ISSN: 0001-0782. DOI: 10.1145/365758.365773.

[23] Richard J. Fateman. *Building Algebra System by Overloading Lisp*. Electronic Article. 2006. URL: https://people.eecs.berkeley.edu/~fateman/generic/overload-small.pdf.

[24] Jan Łukasiewicz. *Aristotle's Syllogistic: From the Standpoint of Modern Formal Logic*. Oxford University Press, 1951. ISBN: 978-0198241447.

[25] Arthur W. Burks, Don W. Warren, and Jesse B. Wright. "An Analysis of a Logical Machine Using Parenthesis-Free Notation". In: *Mathematical Tables and Other Aids to Computation* 8.46 (1954), pp. 53–57. ISSN: 08916837. DOI: 10.2307/2001990. URL: http://www.jstor.org/stable/2001990.

[26] C. L. Hamblin. "Translation to and from Polish Notation". In: *The Computer Journal* 5.3 (1962), pp. 210–213. ISSN: 0010-4620. DOI: 10.1093/comjnl/5.3.210. URL: https://doi.org/10.1093/comjnl/5.3.210.

[27] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison-Wesley Publishing Company, 1985. ISBN: 0-201-37922-8. URL: https://www.adobe.com/jp/print/postscript/pdfs/PLRM.pdf.

[28] John McCarthy. "Recursive functions of symbolic expressions and their computation by machine, Part I". In: *Commun. ACM* 3.4 (1960), 184–195. ISSN: 0001-0782. DOI: 10.1145/367177.367199. URL: https://doi.org/10.1145/367177.367199.

[29] S. Krogdahl. "The Birth of Simula". In: *History of Nordic Computing*. Ed. by J. Bubenko, J. Impagliazzo, and A. Solvberg. Boston, MA: Springer US, 2005, pp. 261–275.

[30] *Stephen Wolfram on Computer Langauge Design, SMP, Mathematica and Wolfram Language*. Podcast. 2020. URL: https://www.infoq.com/podcasts/wolfram-language-mathematica/.

[31] Henri Cohen. *PARI*. Computer Program. URL: https://pari.math.u-bordeaux.fr.

[32] John J. Cannon. *A draft description of the group theory language Cayley*. Conference Paper. 1976. DOI: 10.1145/800205.806325. URL: https://doi.org/10.1145/800205.806325.

[33] *GiNaC*. Computer Program. URL: https://www.ginac.de.

[34] Malcolm A. H. MacCallum. “Computer algebra in gravity research”. In: *Living Rev. Rel.* 21.1 (2018), p. 6. DOI: 10.1007/s41114-018-0015-6.

[35] *SymPy*. Computer Program. URL: https://www.sympy.org/en/index.html.

[36] *PyGiNaC*. Computer Program. URL: http://pyginac.sourceforge.net.

[37] *cypari2*. Computer Program. URL: https://pypi.org/project/cypari2/.

[38] *Pythonica*. Computer Program. URL: https://github.com/bjedwards/pythonica.

[39] *Boost.Python*. Computer Program. URL: https://www.boost.org/doc/libs/1_76_0/libs/python/doc/html/tutorial/index.html.

[40] *pybind11*. Computer Program. URL: https://github.com/pybind/pybind11.

[41] A. V. Korol’kova, D. S. Kulyabov, and L. A. Sevast’yanov. “Tensor computations in computer algebra systems”. In: *Programming and Computer Software* 39.3 (2013), pp. 135–142. ISSN: 1608-3261. DOI: 10.1134/S0361768813030031. URL: https://doi.org/10.1134/S0361768813030031.

[42] L. A. Sevastianov, D. S. Kulyabov, and M. G. Kokotchikova. “An application of computer algebra system Cadabra to scientific problems of physics”. In: *Physics of Particles and Nuclei Letters* 6.7 (2009), p. 530. ISSN: 1531-8567. DOI: 10.1134/S1547477109070073. URL: https://doi.org/10.1134/S1547477109070073.

[43] J. M. Martín-García. *xAct: Efficient tensor computer algebra for the Wolfram Language*. Computer Program. URL: http://www.xact.es.

[44] D. Bolotin and S. Poslavsky. “Introduction to Redberry: the computer algebra system designed for tensor manipulation”. In: *arXiv* (2013), p. 1302.1219. URL: arXiv:1302.1219.

[45] D. N. Vulcanov and V. D. Vulcanov. “The Use of Maple Platform for the Study of Geodesic Motion on Curved Spacetimes”. In: *2006 Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 55–62. DOI: 10.1109/SYNASC.2006.76.

[46] C. Sporea and D. Vulcanov. “Using Maple + GRTensorII in teaching basics of General Relativity and Cosmology”. In: *Romanian Reports in Physics* 68 (2014), p. 29.

[47] Thomas Müller and Jörg Frauendiener. “Studying null and time-like geodesics in the classroom”. In: *European Journal of Physics* 32.3 (2011), pp. 747–759. ISSN: 0143-0807 1361-6404. DOI: 10.1088/0143-0807/32/3/011. URL: http://dx.doi.org/10.1088/0143-0807/32/3/011.

[48] T. Birkandan et al. “Symbolic and numerical analysis in general relativity with open source computer algebra systems”. In: *General Relativity and Gravitation* 51 (2018), p. 4. DOI: 10.1007/s10714-018-2486-x. URL: arXiv:1703.09738.

[49] S. Weinberg. *Gravitation and cosmology*. New York: John Wiley and Sons, 1972.

[50] Hans Stephani et al. *Exact Solutions of Einstein’s Field Equations*. 2nd ed. Cambridge Monographs on Mathematical Physics. Cambridge University Press, 2003. DOI: 10.1017/CBO9780511535185.

[51] Yuki Kanai, Masaru Siino, and Akio Hosoya. “Gravitational Collapse in Painlevé-Gullstrand Coordinates”. In: *Progress of Theoretical Physics* 125.5 (2011), pp. 1053–1065. DOI: 10.1143/ptp.125.1053.

[52] R D Lehn, S S Chabysheva, and J R Hiller. “Klein–Gordon equation in curved spacetime”. In: *European Journal of Physics* 39.4 (2018), p. 045405. DOI: 10.1088/1361-6404/aabdde.

[53] N. D. Birrell and P. C. W. Davies. *Quantum Fields in Curved Space*. Cambridge Monographs on Mathematical Physics. Cambridge, UK: Cambridge Univ. Press, Feb. 1984. ISBN: 978-0-521-27858-4, 978-0-521-27858-4. DOI: 10.1017/CBO9780511622632.

[54] H.S. Vieira and V.B. Bezerra. “Confluent Heun functions and the physics of black holes: Resonant frequencies, Hawking radiation and scattering of scalar waves”. In: *Annals of Physics* 373 (2016), pp. 28–42. DOI: 10.1016/j.aop.2016.06.016.

[55] R. A. d’Inverno and R. A. Russell-Clark. “CLAM—Its function, structure and implementation”. In: *The Computer Journal* 17.3 (1974), pp. 229–233. ISSN: 0010-4620. DOI: 10.1093/comjnl/17.3.229. URL: https://doi.org/10.1093/comjnl/17.3.229.

[56] V. A. Ilyin and A. P. Kryukov. “ATENSOR - REDUCE program for tensor simplification”. In: *Computer Physics Communications* 96.1 (1996), pp. 36–52. DOI: 10.1016/0010-4655(96)00060-4.

[57] X. Jaén A. Balfagón. “TTC: Symbolic tensor calculus with indices”. In: *Computers in Physics* 12.3 (1997). DOI: https://doi.org/10.1063/1.168656.

[58] A. C. Balfagón and X. Jaén. “Simplifying Tensor Polynomials with Indices”. In: (1998). URL: arXiv:gr-qc/9809022.

[59] H. Weyl. *The Classical Groups: Their Invariants and Representations*. 2nd Revised ed. edition. Princeton University Press, 1939. ISBN: 978-0691057569.

[60] R. Portugal. “An Algorithm to Simplify Tensor Expressions”. In: *Computer Physics Communications* 115 (1998), pp. 215–230. DOI: 10.1016/S0010-4655(98)00117-9. URL: arXiv:gr-qc/9803023.

[61] L. R. U. Manssur, R. Portugal, and B. F. Svaiter. “Group-theoretic Approach for Symbolic Tensor Manipulation”. In: *International Journal of Modern Physics C* 13 (2002), pp. 859–879. DOI: 10.1142/S0129183102004571. URL: arXiv:math-ph/0107031arXiv:math-ph/0107032.

[62] Scott Murray. *The Schreier-Sims algorithm*. Master’s Thesis, Australian National University. 1993. URL: https://www.researchgate.net/publication/239215244_The_Schreier-Sims_algorithm.

[63] B. E. Niehoff. “Faster tensor canonicalization”. In: *Computer Physics Communications* 228 (2018), pp. 123–145. DOI: 10.1016/j.cpc.2018.02.014.

[64] A. Young. “On Quantitative Substitutional Analysis”. In: *Proceedings of the London Mathematical Society* s1-33.1 (1900). https://doi.org/10.1112/plms/s1-33.1.97, pp. 97–145. ISSN: 0024-6115. DOI: https://doi.org/10.1112/plms/s1-33.1.97. URL: https://doi.org/10.1112/plms/s1-33.1.97.

[65] Kunle Adegoke et al. “The Standard Representation of the Symmetric Group S_n over the Ring of Integers”. In: *Turkish Journal of Analysis and Number Theory* 3.5 (2015), pp. 126–127. ISSN: 2333-1232. DOI: 10.12691/tjant-3-5-3. URL: http://pubs.sciepub.com/tjant/3/5/3.

[66] Stephen D. Ellis. “Lecture Notes on Particles and Symmetries”. In: University of Washington, 2014. Chap. 11: Young Diagrams and SU(N) Representations. URL: http://staff.washington.edu/sdellis/Phys226.htm.

[67] J. Alcock-Zeilinger and H. Weigert. “Compact Hermitian Young Projection Operators”. In: *Journal of Mathematical Physics* 58.5 (2016). DOI: 10.1063/1.4983478.

[68] J. Alcock-Zeilinger and H. Weigert. “Simplification Rules for Birdtrack Operators”. In: *Journal of Mathematical Physics* 58.5 (2016). DOI: 10.1063/1.4983477.

[69] J. Alcock-Zeilinger and H. Weigert. “Transition Operators”. In: *Journal of Mathematical Physics* 58.5 (2016). DOI: 10.1063/1.4983479.

[70] Stefan Keppeler and Malin Sjödahl. “Hermitian Young Operators”. In: *Journal of Mathematical Physics* 55.2 (2014), p. 021702. ISSN: 1089-7658. DOI: 10.1063/1.4865177. URL: https://arxiv.org/abs/1307.6147.

[71] J. M. Martín-García. “xPerm: fast index canonicalization for tensor computer algebra”. In: *Computer Physics Communications* 179.8 (2008), pp. 597–603. DOI: 10.1016/j.cpc.2008.05.009. URL: arXiv:0803.0862.

[72] Dominic Price, Kasper Peeters, and Marija Zamaklar. “Hiding canonicalisation in tensor computer algebra”. In: *arXiv* (2022). URL: https://arxiv.org/abs/2208.11946.

[73] Dominic Price. “Writing Algorithms in Cadabra2”. Unpublished Work. 2021. URL: https://dominicprice.github.io/cadabra/cadabra-algorithms.pdf.

[74] D. H. Lehmer. “Teaching combinational tricks to a computer”. In: *Proc. Sympos. Appl. Math. Combinatorial Analysis, Amer. Math. Soc* 10 (1960), pp. 179–193.

[75] W. H. Ri and O. H. Song. “A Novel Representation for Permutations”. In: *IEEE Transactions on Information Theory* 67.3 (2021), pp. 1920–1927. ISSN: 1557-9654. DOI: 10.1109/TIT.2020.3048905.

[76] L. J. Guibas and R. Sedgewick. “A dichromatic framework for balanced trees”. In: *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pp. 8–21. ISBN: 0272-5428. DOI: 10.1109/SFCS.1978.3.

[77] Rudolf Bayer. “Symmetric binary B-Trees: Data structure and maintenance algorithms”. In: *Acta Informatica* 1.4 (1972), pp. 290–306. ISSN: 1432-0525. DOI: 10.1007/BF00289509. URL: https://doi.org/10.1007/BF00289509.

[78] Robert Endre Tarjan. “Amortized Computational Complexity”. In: *SIAM Journal on Algebraic Discrete Methods* 6.2 (1985), pp. 306–318. DOI: https://doi.org/10.1137/0606031.

[79] Herbert Robbins. “A Remark on Stirling’s Formula”. In: *The American Mathematical Monthly* 62.1 (1955), pp. 26–29. ISSN: 00029890, 19300972. DOI: 10.2307/2308012. URL: http://www.jstor.org/stable/2308012.

[80] Selmer M. Johnson. “Generation of Permutations by Adjacent Transposition”. In: *Mathematics of Computation* 17.83 (1963), pp. 282–285. ISSN: 00255718, 10886842. DOI: 10.2307/2003846. URL: http://www.jstor.org/stable/2003846.

[81] Robert Sedgewick. “Permutation Generation Methods”. In: *ACM Comput. Surv.* 9.2 (1977), 137–164. ISSN: 0360-0300. DOI: 10.1145/356689.356692. URL: https://doi.org/10.1145/356689.356692.

[82] Carlo Alberto Mantica and Luca Guido Molinari. “A second-order identity for the Riemann tensor and applications”. In: *Colloquium Mathematicum* 1 (2011), pp. 69–82. DOI: 10.4064/cm122-1-7.

[83] S. A. Fulling et al. “Normal forms for tensor polynomials. I. The Riemann tensor”. In: *Classical and Quantum Gravity* 9 (1992).

[84] B. Fiedler. “On the symmetry classes of the first covariant derivatives of tensor fields”. In: *Seminaire Lotharingien de Combinatoire* 49 (2003), Article B49f. URL: arXiv:math/0301042.

[85] J. M. Martin-Garcia, R. Portugal, and L. R. U. Manssur. “The Invar Tensor Package”. In: *Computer Physics Communications* 177 (2007), pp. 640–648. DOI: 10.1016/j.cpc.2007.05.015. URL: https://arxiv.org/abs/0802.1274.

[86] Mary L. Boas. *Mathematical Methods in The Physical Sciences.* Third Edition. John Wiley and Sons, 2006. ISBN: 0-471-19826-9.

[87] Hermann Weyl. “Reine Infinitesimalgeometrie”. In: *Mathematische Zeitschrift* 2.3 (1918), pp. 384–411. ISSN: 1432-1823. DOI: 10.1007/BF01199420. URL: https://doi.org/10.1007/BF01199420.

[88] David Lovelock. “Dimensionally dependent identities”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 68.2 (1970), pp. 345–350. ISSN: 0305-0041. DOI: 10.1017/S0305004100046144. URL: https://www.cambridge.org/core/article/dimensionally-dependent-identities/132CC39ACF23D53920F7475893F9C033.

[89] S. Brian Edgar and A. Höglund. “Dimensionally dependent tensor identities by double antisymmetrization”. In: *Journal of Mathematical Physics* 43 (2002). DOI: 10.1063/1.1425428.

[90] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362. ISSN: 1476-4687. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[91] *Python Pillow.* Computer Program. URL: https://python-pillow.org.

[92] Pauli Virtanen et al. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. In: *Nature Methods* 17.3 (2020), pp. 261–272. ISSN: 1548-7105. DOI: 10.1038/s41592-019-0686-2. URL: https://doi.org/10.1038/s41592-019-0686-2.

[93] Jeff Reback et al. *pandas-dev/pandas: Pandas.* Computer Program. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[94] Adam J Christopherson et al. “Comparing two different formulations of metric cosmological perturbation theory”. In: *Classical and Quantum Gravity* 28.22 (2011). DOI: 10.1088/0264-9381/28/22/225024.

[95] John H. Conway and Alexander Soifer. “Can $n^2 + 1$ unit equilateral triangles cover an equilateral triangle of side $>$ n, say $n + \epsilon$”. In: *The American Mathematical Monthly: The Official Journal of the Mathematical Association of America* (2005).

[96] Florian Staub. “SARAH 4: A tool for (not only SUSY) model builders”. In: *Computer Physics Communications* 185.6 (2014), pp. 1773–1790. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2014.02.018. URL: https://www.sciencedirect.com/science/article/pii/S0010465514000629.

[97] Florian Staub. “Exploring New Models in All Detail with SARAH”. In: *Advances in High Energy Physics* (2015). DOI: https://doi.org/10.1155/2015/840780.

[98] G. Bélanger et al. "SLHAplus: A library for implementing extensions of the standard model". In: *Computer Physics Communications* 182.3 (2011), pp. 763–774. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2010.10.025. URL: https://www.sciencedirect.com/science/article/pii/S0010465510004261.

[99] A. V. Semenov. "LanHEP—a package for the automatic generation of Feynman rules in field theory. Version 3.0". In: *Computer Physics Communications* 180.3 (2009), pp. 431–454. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2008.10.012. URL: https://www.sciencedirect.com/science/article/pii/S0010465508003718.

[100] Daniel V. Schroeder. "Physics Simulations in Java". Unpublished Work. 2011. URL: https://physics.weber.edu/schroeder/javacourse/javamanual.pdf.

[101] Daniel V. Schroeder. "Physics Simulations in Python". Unpublished Work. 2018. URL: https://physics.weber.edu/schroeder/scicomp/PythonManual.pdf.

[102] O. Castillo-Felisola, D. T. Price, and M. Scomparin. "Cadabra and Python algorithms in General Relativity and Cosmology I: Generalities". In: *Awaiting Publishing* (2021). DOI: 10.48550. URL: https://arxiv.org/abs/2210.00005.

[103] Jordi Vallverdú i Segura. "Computational Epistemology and e-Science: A New Way of Thinking". In: *Minds and Machines* 19.4 (2009), p. 557. ISSN: 1572-8641. DOI: 10.1007/s11023-009-9168-0. URL: https://doi.org/10.1007/s11023-009-9168-0.

[104] Paul Thagard. *Computational Philosophy of Science*. Electronic Book. 1988. DOI: 10.7551/mitpress/1968.001.0001. URL: https://doi.org/10.7551/mitpress/1968.001.0001.

[105] G. C. Levy. "How changes in computer technology are revolutionizing the practice of chemistry". In: *J Chem Inf Comput Sci* 28.4 (1988). Levy, G C RR-0317/RR/NCRR NIH HHS/United States Journal Article Research Support, Non-U.S. Gov't Research Support, U.S. Gov't, P.H.S. United States J Chem Inf Comput Sci. 1988 Nov;28(4):167-74. doi: 10.1021/ci00060a001., pp. 167–74. ISSN: 0095-2338 (Print) 0095-2338. DOI: 10.1021/ci00060a001.

[106] H. A. Thurston. "Leibniz's Notation". In: *The Mathematical Gazette* 57.401 (1973), pp. 189–191. ISSN: 00255572. DOI: 10.2307/3615564. URL: http://www.jstor.org.ezphost.dur.ac.uk/stable/3615564.

[107] Andrzej J. Maciejewski and Maria Przybylska. "Non-integrability of the three-body problem". In: *Celestial Mechanics and Dynamical Astronomy* 110.1 (2011), pp. 17–30. ISSN: 1572-9478. DOI: 10.1007/s10569-010-9333-z. URL: https://doi.org/10.1007/s10569-010-9333-z.

[108] Alain Chenciner and Richard Montgomery. "A Remarkable Periodic Solution of the Three-Body Problem in the Case of Equal Masses". In: *Annals of Mathematics* 152.3 (2000), pp. 881–901. ISSN: 0003486X. DOI: 10.2307/2661357. URL: http://www.jstor.org/stable/2661357.

[109] K. Appel and W. Haken. "Every planar map is four colorable". In: *Bull. Amer. Math. Soc.* 82.5 (1976). Cited By :10 Export Date: 25 June 2021, pp. 711–712.

[110] C. W. H. Lam, L. Thiel, and S. Swiercz. “The Non-Existence of Finite Projective Planes of Order 10”. In: *Canadian Journal of Mathematics* 41.6 (1989), pp. 1117–1123. ISSN: 0008-414X. DOI: 10.4153/CJM-1989-049-4. URL: https://www.cambridge.org/core/article/nonexistence-of-finite-projective-planes-of-order-10/EBD5E851090B1983978BBC6FE18D2A3Fhttps://www.cambridge.org/core/services/aop-cambridge-core/content/view/EBD5E851090B1983978BBC6FE18D2A3F/S0008414X00002534a.pdf/div-class-title-the-non-existence-of-finite-projective-planes-of-order-10-div.pdf.

[111] Boris Konev and Alexei Lisitsa. “A SAT Attack on the Erdős Discrepancy Conjecture”. In: *Theory and Applications of Satisfiability Testing – SAT 2014*. Ed. by Carsten Sinz and Uwe Egly. Springer International Publishing, pp. 219–226. ISBN: 978-3-319-09284-3.

[112] Boris Konev and Alexei Lisitsa. “Computer-aided proof of Erdős discrepancy properties”. In: *Artificial Intelligence* 224 (2015), pp. 103–118. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2015.03.004. URL: https://www.sciencedirect.com/science/article/pii/S0004370215000429.

[113] Joseph E. Fields. *A Gentle Introduction to the Art of Mathematics*. A. T. Still University, 2015. URL: https://open.umn.edu/opentextbooks/textbooks/177.

[114] Antonio J. Durán Guardeño, Mario Pérez, and J. L. Varona. “Misfortunes of a mathematicians’ trio using Computer Algebra Systems: Can we trust?” In: *ArXiv* abs/1312.3270 (2013).

[115] Donald MacKenzie. *Mechanizing proof: computing, risk, and trust*. MIT Press, 2001. ISBN: 0262133938.

[116] Ken Thompson. “Reflections on trusting trust”. In: *Commun. ACM* 27.8 (1984), 761–763. ISSN: 0001-0782. DOI: 10.1145/358198.358210. URL: https://doi.org/10.1145/358198.358210.

[117] L. Thomas van Binsbergen et al. *A principled approach to REPL interpreters*. Conference Paper. 2020. DOI: 10.1145/3426428.3426917. URL: https://doi.org/10.1145/3426428.3426917.

[118] Christian Dohrmann Ulrich Kortenkamp. “User Interface Design For Dynamic Geometry Software”. In: *Acta Didactica Napocensia* 3.2 (2010). ISSN: EISSN-2065-1430. URL: http://dppd.ubbcluj.ro/adn/article_3_2_6.pdf.

[119] B. L. Leong. *Iris: design of an user interface program for symbolic algebra*. Conference Paper. 1986. DOI: 10.1145/32439.32440. URL: https://doi.org/10.1145/32439.32440.

[120] Norbert Kajler and Neil Soiffer. “Some human interaction issues in computer algebra”. In: *SIGSAM Bull.* 28.1 (1994), 18–28. ISSN: 0163-5824. DOI: 10.1145/182130.182133. URL: https://doi.org/10.1145/182130.182133.

[121] Norbert Kajler. “Building a Computer Algebra environment by composition of collaborative tools”. In: Design and Implementation of Symbolic Computation Systems. Springer Berlin Heidelberg, pp. 85–94. ISBN: 978-3-540-48031-0.

[122] Kitware. *CMake*. Computer Program. URL: https://cmake.org.

[123] Eric Raymond. “Understanding Version-Control Systems (DRAFT)”. Unpublished Work. Accessed 2021. URL: http://www.catb.org/~esr/writings/version-control/version-control.html.

[124] Nayan B. Ruparelia. “The history of version control”. In: *SIGSOFT Softw. Eng. Notes* 35.1 (2010), 5–9. ISSN: 0163-5948. DOI: 10.1145/1668862.1668876. URL: https://doi.org/10.1145/1668862.1668876.

[125] Scott Chacon and Ben Straub. *Pro Git*. 2nd ed. edition. Apress, 2014. ISBN: 978-1484200773. URL: https://git-scm.com/book/en/v2.

[126] Gioele Barabucci. *diffi: diff improved; a preview.* 2018, pp. 1–4. DOI: 10.1145/3209280.3229084.

[127] Thomas Kluyver et al. *Jupyter Notebooks – a publishing format for reproducible computational workflows.* Conference Paper. 2016. URL: https://eprints.soton.ac.uk/403913/.

[128] Jupyter. *nbdime*. Computer Program. URL: https://github.com/jupyter/nbdime.

[129] Yusuf Nugroho, Hideaki Hata, and Kenichi Matsumoto. *How Different Are Different diff Algorithms in Git? Use –histogram for Code Changes.* 2019.

[130] John W. Ratcliff and David Metzener. “Pattern Matching: The Gestalt Approach”. In: *Dr. Dobb’s Journal* 46 (1988).