Mathematically Informed Linear Algebra Codes Through Term Rewriting

Matthew Rocklin

August 2013

Contents

Abstract

To motivate modularity in scientific software development we build and study a system to generate mathematically informed linear algebra codes as a case study. We stress the effects of modularity on verification, flexibility, extensibility, and distributed development, each of which are particularly important in scientific contexts.

Development in computational science is both accelerated and burdened by changing hardware and diffusion into new disciplines. Hardware development expands the scale of feasible problems. This same development also brings challenging programming models that are both unfamiliar and reflect complex memory and communication architectures. The adoption of computational methods by new fields multiplies both the potential and the burden of this growth. Old techniques can be reapplied to fresh problems in new fields such as biology or within smaller scale research groups. Unfortunately these new communities bring a population of novice scientific programmers without a strong tradition of software engineering. The progress of scientific computing is limited by scientists' ability to develop software solutions in these new fields for this new hardware.

This dissertation discusses the health of the current scientific computing ecosystem and the resulting costs and benefits on scientific discovery. It promotes software modularity within the scientific context for the optimization of global efficiency. To support this argument it considers a case study in automated linear algebra, a well studied problem with mature practitioners. We produce and analyze a prototype software system that adheres strictly to the principles of modularity.

This system automatically generates numerical linear algebra programs from mathematical inputs. It consists of loosely coupled modules which draw from computer algebra, compilers, logic programming, and static scheduling. Each domain is implemented in isolation. We find that this separation eases development by single-field experts, is robust to obsolescence, enables reuse, and is easily extensible.

1 Introduction

The development and execution of scientific codes is both critical to society and challenging to scientific software developers. When critical applications require faster solutions the development community often turns to more powerful computational hardware. Unfortunately the full use of high performance hardware by scientific software developers has become more challenging in recent years due to changing hardware models. In particular, power constraints have favored increased parallelism over increased clock speeds, triggering a paradigm shift in programming and algorithm models. As a result squeezing progress out of continued hardware development is becoming increasingly challenging.

An alternative approach to reduce solution times is to rely on sophisticated methods over sophisticated hardware. One strives to reduce rather than to accelerate the computation necessary to obtain the desired scientific result. Computationally challenging problems can often be made trivial by exposing and leveraging special structure of the problem at hand. Problems that previously required weeks on a super-computer may only require hours or minutes on a standard workstation once the appropriate method is found to match the structure of the problem.

Unfortunately the use of expert methods suffers from many of the same drawbacks as the use of large scale parallelism. In particular the use of expert methods depends on comprehensive and deep understanding of expertise in the fields relevant to the particular computation. Optimizations requiring this expertise can be equally inaccessible as the optimizations of high performance parallel hardware. Expert solutions may depend on theories requiring several years of advanced study. As a result universal deep expertise is as inconceivable as universal knowledge of advanced low-level parallel programming techniques.

Historically we solve this problem by having experts build automated systems to apply their methods more universally. Just as we develop systems like Hadoop or parallel languages to automate the use of parallel hardware we may also develop systems to automate the selection and implementation of expert solutions. This approach enables the work of the expert few to benefit the applications of the naive many. A small group of domain experts bent on automation may accelerate the work of thousands.

The traditional approach of automation is to have domain experts build automated systems themselves. For domains such as parallel programming this approach is feasible because a significant subset of the parallel programming community is also well versed in the advanced programming techniques necessary to construct automated systems; it is easy to find simultaneous experts in parallelism and automation. Unfortunately this situation may not be the case for various mathematical or scientific theories. It may be difficult to find an individual or group of researchers with simultaneous deep expertise both in a particular mathematical domain and in the practice of automation and dissemination through software engineering.

Multi-disciplinary work compounds the problem for the need of simultaneous expertise. When many domains of expertise are simultaneously required it often becomes impossible to find an individual or working group that can competently satisfy all domains. In the rare case where such a team is formed, such as in the case of critical operations (e.g. defense, meteorology), the resulting work is rarely transferable to even slightly different configurations of domains. Tragically, important applications for which all expertise to solve the problem

efficiently is known often go unsolved because that expertise is never collected in the same research unit. While elements of expertise may be trivially accessible to particular specialists, localizing the complete set of necessary expertise may be challenging. The only novelty and challenges to many of today's applications is the challenge of coordination.

The problem of many-domain expertise distribution aligns well with the software principle of modularity, particularly when the interfaces between modules align well with the demographic distribution of expertise. The principle of modularity proposes the separation of software systems into separable components such that each component solves exactly one concern. Relationships between elements within a module should be dense while relationships between inter-module elements should be sparse. Software engineering communities support this principle due to its benefits in testing, extensibility, reuse, and interoperation.

In the context of communities like scientific software engineering we add the following to the list of benefits for modularity; *demographically aligned modules increase the qualified development pool*. This claim is particularly true when the development pool consists largely of specialists, as is the case in the academic research environment. Developers may only easily contribute to a software package when the scope of that package is within their expertise. When the developer pool consists largely of specialists then even a moderately interdisciplinary module may exclude most developers. This situation is particularly unfortunate because those same excluded specialists may have valuable expertise for critical subparts of the problem. The separation of a software package into demographically aligned modules enables contributions from single-field experts. In fields where expertise is largely siloed (as in many scientific fields) this benefit can be substantial.

In summary:

- We should look towards sophisticated methods alongside sophisticated hardware.
- We should disseminate these methods to applications through automation.
- Interdisciplinary applications and a lack of software engineering tradition in the sciences encourage the use of a modular design that separates different domains into different software packages.
- In particular, the practice of automation should itself be separated from the formal description of mathematical expertise.

The rest of this chapter discusses traits of the scientific computing ecosystem in more detail and isolation. Sections 1.1 and 1.2 discuss the value and cost of computing to society at large. Sections 1.3 and 1.4 outline the challenges of the distribution of expertise and the benefits of modular software design. This dissertation investigates these concepts through linear algebra, a paragon application; this will application be introduced in section 1.5. Finally in Sections 1.6 and 1.7 we concretely outline the contributions and outline of the work that follows.

1.1 Value

Scientific software is of great value to society. Computational software enables scientists to approximately simulate complex physical systems in a computer, providing insight into exotic systems, or foresight to developing ones.

Consider the benefits of numerical weather prediction. Meteorological forecasting is able to reduce the cost of food (by warning farmers), decrease the frequency of power outages (by anticipating spikes in temperature and climate control usage), inform the location of renewable energy resources, and reduce the fatality of extreme weather events (e.g. hurricanes.) Traumatic events like the recent Hurricane Sandy (2012) were predicted several days in advance, giving the coastal population time to secure valuables and vacate immobile residents inland. Computation surrounds society and provides a thousand unnoticed efficiencies.

Scientific research in particular looks to computation. Computation has branched out from the physical sciences to the life and social sciences, bringing a greater and broader population of academic and industrial researchers online. Novel research fields may not yet have established software solutions, forcing this broad population of researchers to develop new software solutions within their domain. This broadening of computational science is a striking feature of the last decade. It is the author's opinion that the shift in demographics presents a challenge equivalent in scale to the challenge of parallelism.

1.2 Cost

Scientific software is also of great cost to society. Scientific researchers spend more and more of their time engineering software.

Scientific software is costly because it is difficult. The computational solution of scientific problems often require expertise from a variety of scientific, mathematical, and computational fields; in addition they must be formally encoded into software. This task is intrinsically difficult and therefore costly.

Scientific software is costly also because it is done by researchers with poorly matched training. The cost is magnified because it is often done by a non-computational but academic researcher. Such a researcher is often highly trained in another field (e.g. biology, physics) with only moderate training in software development. This mismatch of training and task means that the task occupies many working hours from some of society's highly trained citizens. It is like hiring a lawyer to fix a plumbing problem. The lawyer is only somewhat competent at the task but still charges high rates.

The cost of scientific software could be mitigated through reuse Fortunately scientific problems often share structure. Large parts of the solutions for one scientific problem may apply to a large number of other relevant problems. This is more likely within the same field but can occur even in strikingly different domains. The simulation of galaxies within a globular cluster is surprisingly similar to from the simulation of molecules within a liquid.

The cost of scientific software could be mitigated through mathematics These problems also often benefit from rich and mature mathematical theory built up over centuries. This theory can inform the efficient solution of scientific problems, often transforming very challenging computational tasks into mathematically equivalent but computationally trivial ones.

And yet scientists often start from scratch Existing code may not be applied to novel problems for any of the following reasons

1. It is not released to the public or is not sufficiently documented
2. The new researcher may not know to look for it
3. It integrates too many details about previous application or architecture

Coding practices, package managers, and general software support/infrastructure from the general programming community have alleviated many of these issues recently. Tool support and dependency systems have enabled the widespread propagation of small general purpose utilities. A culture of open source scientific code sharing around accessible scripting languages has drastically lowered the bar to obtaining, building, and integrating foreign code.

Unfortunately this development rarely extends to more sophisticated solutions. Particularly, this dissertation is concerned with the last issue in the context of sophisticated algorithms; existing code often integrates too many details about the previous application or architecture.

Older scientific software often assumes too much about its original application. E.g. codes for the interaction of many particles may be specialized to molecular dynamics, limiting their applicability to similar problems in astronomy, such as the simulation of stellar systems within globular clusters. Broadly useful code elements are often tightly and unnecessarily integrated into application-specific code elements within a single codebase. Extracting relevant components from irrelevant ones may be more difficult than simply rewriting the relevant components from scratch. Unfortunately, this rewritten work may continue to suffer from the original flaw of integrating general components into domain specific ones. As a result the same algorithm ends up being implemented again and again in several marginally different scientific projects, each at a substantial cost to society.

Incentives Unfortunately scientists have little incentive to generalize their codes to other domains. Existing incentives drive progress within a narrow scientific field, not within the broader field of scientific software. Producing computational components applicable to foreign fields generally has only marginal value within any individual scientist's career. Unfortunately this situation produces a prisoner's dilemma type situation with a globally suboptimal result.

It is the position of this dissertation that the construction of a base of modular software can shift incentives to tip the prisoner's dilemma situation towards the global optimum.

1.3 Modularity

The software principle of modularity may provide a path towards reducing this problem. Modularity supports the separation of a software project into modules, each of which performs an individual and independent task. This approach confers the following benefits:

Specialization of Labor When separation between modules aligns with separation between disciplines of expertise, modularity enables the specialization of labor, assigning each module to a different researcher with a different specialization. In general we assume that it

is easier to find a few experts in different fields than it is to find a single researcher who is an expert in all of those fields.

Evolution through Interchangeability When modules are combined with an established interface they become interchangeable with other simultaneous efforts by different groups in the same domain. Shared interfaces and interchangeability supports experimentation and eases evolution under changing contexts (such as changing hardware,) which gives users more choices and module developers an immediate audience of applications.

Reuse Scientific computing algorithms are often broadly shared across otherwise unrelated domains. E.g. the algorithms for the solution of sparse linear systems may be equally applicable both to the time evolution of partial differential equations and the optimization of machine learning problems. By separating out these components a larger community is able to both develop and benefit from the same software components. This approach yields a higher quality product within all domains.

Verification Smaller pieces are simpler to test and verify. Complex interactions between multiple fields within a single codebase may produce complex errors that are difficult to identify and resolve; leading potentially both to development frustration (at best), and incorrect science (at worst). Verification at the module level allows most issues to be resolved in an isolated and more controlled setting. Verification of modules and of interactions is often sufficient to ensure verification of a larger system.

Obsolescence Scientific software is often made obsolete either by the development of new methods, new hardware, new languages within the programming community, or even very rarely by new science or new mathematics. The separation of these projects into modules isolates the obsolescence to only the few modules that must be replaced. Because different elements of scientific computing evolve at different rates (e.g. hardware changes quickly while math changes slowly), this separation can avoid frequent rewrites of infrequently changing domains (e.g. mathematical elements may be allowed to persist between hardware generations.)

1.4 Expertise

Modularity allows single-field experts to meaningfully contribute their expertise broadly across applications, yielding immeasurable value. In this section we discuss the value, distribution, and demands of expertise in scientific applications. We use integration by numeric quadrature as a running example.

Skewed Distribution The distribution of expertise with a particular domain is highly skewed. Many practitioners understand naive solutions while very few understand the most mature solutions. In scientific and numerical domains mature solutions may require years of dedicated study. For example the rectangle and trapezoid rules are taught in introductory college calculus classes to the general engineering audience. Advanced techniques such as sparse grids and finite elements are substantially less well known.

Performance These expert methods can greatly improve performance. The relative cost between naive and mature solutions can vary by several orders of magnitude. It is common for a previously intractable problem to be made trivial by engaging the correct method. In quadrature for example adaptivity, higher order methods, and sparsity can each supply performance improvements of several orders of magnitude.

More Problems than Experts The number of scientific problems that engage a particular domain generally exceeds the number of experts. E.g. far more questions use integration than there are experts in numerical integration.

Broad Applicability of Single Domain A single domain may be used by a wide set of projects. This set is rarely known by the domain expert. E.g. numerical integration is used in several fields that are unfamiliar to numerical analysts.

Broad Demands from a Single Problem Conversely scientific computing problems touch many domains. A computational approach to a single research question may easily involve several scientific, mathematical, and computational domains of expertise. For example numerical weather prediction touches on meteorology, oceanography, statistics, partial differential equations, distributed linear algebra, and high performance array computation.

Analysis We need to compare, distribute, and interconnect expertise efficiently. An ideal software ecosystem selects and distributes the best implementation of a particular domain to all relevant problems. Multiple implementations of a domain in stable co-existence is a symptom of a poorly functioning ecosystem. It is a sign of poor reuse and fragments future development.

1.5 Numerical Linear Algebra

Numerical Linear Algebra is a long running scientific application that has been developed by experts for several decades. This dissertation investigates linear algebra as a case study and microcosm for modular scientific software development.

Linear algebra serves as an interface between computational scientists and computational hardware. Many scientific domains can express their problems as matrix computations. Scientific or mathematical programmers are often well trained in theoretical linear algebra. High performance and curated libraries for numerical linear algebra exist for most computational architectures. These libraries have a consistent and durable interface and are supported by a variety of software. As a result of this cross-familiarity, linear algebra serves as a de-facto computational language between computational researchers and low-level hardware. The majority of floating point operations within scientific computations occur within a linear algebra routine.

For full performance, linear algebra libraries are tightly coupled to hardware models. This organization forces a software rewrite when existing models become obsolete. Today this occurs both due to the break from homogeneity and the break from a simple memory-hierarchy (due to various forms of parallelism). Despite the paramount importance of numerical linear

algebra only a small number of groups at major universities and national labs seem able to contribute meaningfully to the software engineering endeavor. The simultaneous expertise required in linear algebra, numerical analysis, low-level hardware, parallel computation, and software engineering, is only found in relatively few groups specially geared for this task; each well-known group has decades of experience.

These groups produce very high quality software, capable of executing common matrix computations orders of magnitude faster than reasonably intelligent attempts. Still, despite their expertise it is difficult for them to respond to the rapidly changing hardware landscape, introducing a lag of several years between a new technology (e.g. CUDA) and a mature linear algebra implementation that makes use of that technology (e.g. CUDA-LAPACK).

1.6 Contributions

This dissertation presents a particular software system to transform high-level matrix expressions into high-performance code. This project supplies mathematical programmers access to powerful-yet-inaccessible computational libraries.

Additionally, this dissertation discusses the virtues of modularity within this domain. Care has been taken to separate the software system into a set of individual components, each of which retains value both independently and in other applications. These software contributions are the following:

- A computer algebra system for manipulation and inference over linear algebra expressions
- A high-level representation of common numeric libraries and the generation of low-level codes
- A composable Python implementation of miniKanren, a logic programming system
- A conglomerate compiler from matrix expressions to intelligently selected computational routines

These pieces are then assembled into a conglomerate project to provide easy access to intelligently selected computational routines.

1.7 Overview

This dissertation describes modularity in scientific computing in general and linear algebra in particular. It is separated into the following chapters:

First we discuss past and current efforts in modularity and numerical linear algebra in Chapter 2. We then separate the problem of automated generation of high-level codes for matrix computations into reusable components. In each section we discuss the history, problems, and a software solution to a single subproblem. In particular,

- Chapter 3 discusses computer algebra
- Chapter 4 discusses computations with BLAS/LAPACK and code generation
- Chapter 5 discusses term rewriting

- Chapter 6 assembles components from the previous sections to form a cohesive whole. It then exercises this conglomerate on common computational kernels.

We then demonstrate the extensibility of this system by adding components not present in the original design.

- Chapter 7 discusses static scheduling and the generation of parallel codes
- Chapter 8 further demonstrates extensibility using statistics and swappable back-ends.

Finally we conclude with remarks for future work in Chapter 9.

2 Background

Scientific software engineering is a deep topic with broad scope and decades of history. Even a reasonably comprehensive coverage is well beyond the scope of this document. Instead we pick and describe a few particularly relevant subtopics:

- In Section, 2.1 we give motivation for historical developments to make scientific computing accessible and to distribute work
- In Section, 2.2 we discuss numerical weather prediction as a representative of monolithic scientific software.
- In Section, 2.3 we discuss BLAS and LAPACK, widely reused static libraries for numerical linear algebra.
- In Sections 2.4 and 2.5, we discuss the BLAS/LAPACK/PETSc/FEniCS software stack and Trilinos as representatives of modern scientific software
- In Section 2.6 we discuss the use of automation to narrow the gap between domain users and low-level code.

2.1 Evolution

Computational science predates software. Numerical methods for the solution of critical problems have a rich history. Iterative methods for the solution of non-linear systems have roots dating back to French artillerymen when officers attempted to hit a military target by making small adjustments to cannon angles and gunpowder quantities. Because this target was often aiming back at them it was critical that the solution to the non-linear system be found with as few tries as possible. During World War II machinery was specifically developed to aid in cryptanalysis at Bletchley Park to intercept war messages. Modern scientific software often retains this same sense of urgency. All efforts are focused on building and pushing modern hardware to its limits for the solution of a critical problem.

While these efforts are both commendable and groundbreaking, they must often sacrifice general applicability in order to obtain peak performance. As a result future endeavors are unable to benefit from past efforts.

Static Libraries Fortunately as trends in scientific computing emerge computer science communities are encouraged to produce generally applicable codes for use across many simulations. Libraries like BLAS/LAPACK, FFTW, QUADPACK, ARPACK, etc., were created and refined early on and maintain relevance today. These battle-hardened codes form a set of high-level primitive operations on which much of the core community relies.

Scripting Languages As computers become more prevalent, the use of numeric methods expands into smaller and less computationally specialized disciplines and research groups. Groups without formal training in writing scientific code may find historical systems and interfaces challenging.

Scripting languages like Matlab, R, and Python address this growing user group by providing high-level dynamic languages with permissive syntax and interactive use. Unfortunately the lack of explicit types and a compilation step drastically reduces the performance of codes written within these languages. Linking low-level performant codes (e.g. tuned matrix multiplication) to high-level routines (e.g. Matlab's `*` operator) bridges this gap on a small but expressive set of array primitives. Scripting languages linked to low-level libraries are often sufficiently performant for many array programming tasks found in small research labs.

Open Source Scientific Ecosystems A broad userbase coupled with advances in online code sharing and relatively robust package managers has fostered a culture of open source scientific code publication. Often the choice of language is made due to the location of preexisting scientific codebases rather than the features of the language itself. Large scientific software ecosystems provide a scaffolding for several disciplines (R for Statistics, bioperl for Biology, SciPy for numerics).

Adaptation to Parallelism The rise of shared, distributed, and many-core parallelism forces the development community to reevaluate its implementation. Preexisting scientific software has substantial intellectual value. Unfortunately the adaptation of this software to take advantage of parallel hardware seems both arduous and necessary. Like old Fortran codes they are often called from other languages at great expense (e.g. using foreign function interfaces to call SciPy from Hadoop via Java.)

Return to Compilation Performance issues on modern hardware have increased interest in the compilation of these dynamic languages. At the time of writing, the scientific Python ecosystem supports the following active projects for compilation and interoperation of Python with low-level languages (largely for array programming.)

`Copperhead`	`Computations`	`Cython`	`HyperOpt`
`Ignition`	`Instant`	`Julia`	`PyOP2/Fenics`
`SymPy`	`Numba Pro / Lair`	`NumExpr`	`Theano`
`Seamless`	`ODIN`	`parakeet`	`PyTrillinos`
`fwrap`	`xdress`	`pydy`	`falcon`
`sparrow`	`blaze`	`dynd`	`PyCUDA / PyOpenCL`
`Loop.py`	`PyKit`	`Pythran`	`PyMC`
`PyOp2`			

We include this list mainly to stress the number of projects in this effort. This demonstrates both the community's commitment and a lack of organization.

2.2 Case Study - Numerical Weather Prediction

An alternative approach is to collect a group of experts for the long-term development and maintenance of code for a critical application. Numerical weather prediction is an example of such a monolithic code.

Numerical weather prediction benefits society. Major industries like agriculture and construction rely on short-term forecasts to determine day-to-day operation. The power grid relies on 12-24 hour forecasts to predict both load (due to climate control) and supply (due to weather dependent renewable energies) so that it can maintain a balanced resource without blackouts or burnouts. Severe weather events are substantially less fatal due to several day advanced warning. Food is substantially cheaper; agriculture insurance is a multi-billion dollar industry in the United States alone.

Numerical weather prediction is also computationally challenging. It requires substantial atmospheric modeling to simulate difficult sets of PDEs that represent an inherently chaotic system. These must be solved over a very large domain (the United States) and yet very finely resolved both in space (10km) and in time (minutes) to maintain numerical stability. Forecasts must be rerun frequently as a variety of new observations are recorded and assimilated and they must be run substantially faster than nature herself evolves.

WRF Because of these benefits and costs the federal government has supported the production and maintenance of high performance numerical codes for the short-term simulation and forecast of the weather. Along with several other federal and university groups the National Center for Atmospheric Research (NCAR) maintains the Weather Research Forecast model (WRF), which serves as a base for both research (ARW) and operational (NMW) codes. It is written in Fortran and MPI and maintained by a dedicated team of software developers.

It is used by a broad community of meteorologists and weather service professionals without computational expertise. External control is managed through a set of Fortran namelists that specify model parameters.

Code Example Internally the codebase is organized into several Fortran files that handle different physical processes. A representative code snippet is reproduced below:

```
! phys/module_mp_wsm5_accel.F:644 Version 3.4

do k = kte, kts, -1
      if(t(i,k,j).gt.t0c.and.qrs(i,k,2).gt.0.) then
!-------------------------------------------------------------
! psmlt: melting of snow [HL A33] [RH83 A25]
!       (T>T0: S->R)
!-------------------------------------------------------------
        xlf = xlf0
```

```
!               work2(i,k)= venfac(p(i,k),t(i,k,j),den(i,k,j))
        work2(i,k)= (exp(log(((1.496e-6*((t(i,k,j))*sqrt(t(i,k,j)))    &
                    /((t(i,k,j))+120.)/(den(i,k,j)))/(8.794e-5          &
                    *exp(log(t(i,k,j))*(1.81))/p(i,k,j))))              &
                    *((.3333333)))/sqrt((1.496e-6*((t(i,k,j))           &
                    *sqrt(t(i,k,j)))/((t(i,k,j))+120.)/(den(i,k,j))))  &
                    *sqrt(sqrt(den0/(den(i,k,j)))))
```

This snippet encodes the physics behind the melting of snow under certain conditions. It is a large mathematical expression iterated over arrays in a do-loop. This pattern is repeated in this routine for other physical processes such as "instantaneous melting of cloud ice", "homogeneous freezing of cloud water below -40c", "evaporation/condensation rate of rain", etc.

Adaptability to Hardware Like the code snippet above, much of the computational work required to forecast the weather is FLOP intensive and highly regular, making it amenable to GPU computing. In 2008 WRF developers investigated both the ease and utility of translating parts of WRF to CUDA[1]. They relate translating a 1500 line Fortran codebase to CUDA through a combination of hand coding, Perl scripts, and specialized language extensions. They include the following listing showing the original Fortran and their CUDA equivalent annotated with their custom memory layout macros

```
DO j = jts, jte
  DO k = kts, kte
    DO i = its, ite
      IF (t(i,k,j) .GT. t0c) THEN
        Q(i,k,j) = T(i,k,j) * DEN( i,k,j )
      ENDIF
    ENDDO
  ENDDO
ENDDO
```

(a) Fortran

```
//_def_ arg ikj:q,t,den
//_def_ copy_up_memory ikj:q
[...]
for (k = kps-1; k<=pe-1; k++){
  if (t[k] > t0c) {
    q[k] = t[k] * den[k] ;
  }
}
[...]
//_def_ copy_down_memory ikj:q
```

(b) CUDA C

They report a `5-20x` speedup in the translated kernel resulting in a `1.25-1.3x` speedup in total execution time of the entire program. They note the following:

- *a modest investment in programming effort for GPUs yields an order of magnitude performance improvement*
- *Only about one percent of GPU performance was realized but these are initial results; little optimization effort has been put into GPU code.*
- They later state that this project was *a few months effort.*

Use in Practice Two years later operational instructions were released to use this work for a particular version of WRF [2]. Today GPGPU is still not a standard option for operational users.

Later work Four years later Mielikainen et al[3] report increased substantially efficiency through exploiting more specialized GPU optimizations not often known by general researchers, some specific to the model of GPU.

These results represent a 60% improvement over the earlier GPU accelerated WSM5 module. The improvements over the previous GPU accelerated WSM5 module were numerous. Some minor improvements were that we scalarized more temporary arrays and compiler tricks specific to Fermi class GPU were used. The most important improvements were the use of coalesced memory access pattern for global memory and asynchronous memory transfers.

Analysis WRF software design is *embarrassingly* modular. This modularity separates routines representing physical processes from each other when they happen to be independent. It makes little effort at *vertical* modularity that might separate high and low level code.

In the listing above we see a high-level meteorological model implemented in a very low-level implementation alongside computational optimizations and array layouts. This problem is intrinsically simple; it is an algebraic expression on a few indexed arrays. And yet when external pressures (GPGPU) necessitated a code rewrite, that work took months of work from a researcher who was already familiar with this codebase. That work failed to implement several well known GPU specific optimizations; these optimizations were only implemented four years later, a significant gap.

While this file encodes relatively high-level concepts it is difficult to perform sweeping high-level manipulations. As physics, numerical methods, and computational architecture change, flexibility is likely to become more important.

Other Codes WRF is an instance of a meteorological code written for a specific purpose. The surrounding ecosystem contains many variants and completely separate implementations. Each of these represented opportunities for reuse.

Independent Codes Other governments have produced similar codes for numerical weather prediction. The European Centre for Medium-Range Weather Forecasts (ECMWF) maintains the Integrated Forecasting System (IFS) [4], a similar code used by European countries. In many ways its architecture parallels that of WRF. It is a large Fortran/MPI codebase maintained by a dedicated group used by a similar population.

Despite these similarities the two codebases often produce substantially different forecasts. Each has strengths/weaknesses that arise in different situations.

Adjusted Codes NCAR has forked and adjusted `WRF` for specific situations. The Hurricane Weather Research Forecasting Model (`HWRF`) modifies `WRF` to be particularly suitable in the case of severe storms. Particular models have been developed to support more perturbed states.

`WRFDA` is an implementation of `WRF` for data assimilation. The latest version contains experimental algorithms for 4D-var, a new numerical technique that uses automatic derivatives to assimilate new observations more efficiently. This change was accomplished by applying automated AD compilers to a stripped down version of `WRF` with some troublesome modules

rewritten more simply. Unfortunately, the complete version of `WRF` was not amenable to automated transformation.

Climate Growing concern over global warming has spurred research into climate models. Meteorological codes like WRF are intended for short-term forecasts, rarely exceeding ten days. Climate models simulate the same physical processes but over decade or century timescales. Because of the difference in time scale, climate models must differ from meteorological models, both for computational efficiency and in order to conserve quantities that might not be of interest over the short term.

Analysis Computational atmospheric science is a large, and active field. The political and economic impact of weather and climate prediction have spurred research into new methods and applications. Unfortunately most developments seem to be either painful incremental improvements or are complete rewrites by large collaborations. These developments are more costly and development is slower than seems necessary.

2.3 BLAS/LAPACK

2.3.1 Design

Computational science often relies on computationally intensive dense linear algebra operations. This reliance is so pervasive that numerical linear algebra (NLA) libraries are among the most heavily optimized and studied algorithms in the field.

An early pair of software packages, BLAS and LAPACK[5], were sufficiently pervasive to establish a long-standing standard interface between users and developers of dense numerical linear algebra libraries (DLA). This particular set of algorithms has seen constant development over the last few decades due both to the importance of this problem and to the standard interface.

To optimize these operations fully the software solutions must be tightly coupled to hardware architecture. In particular the design of most BLAS/LAPACK implementations tightly integrates a model for the memory architecture. Because memory architectures continue to change, no long-standing solution has arisen and this field sees constant development. In this sense it is a self-contained microcosm of the larger scientific software development problem. The only difference is that the majority of practitioners in numerical linear algebra are highly trained experts.

Basic Linear Algebra Subroutines (BLAS) The Basic Linear Algebra Subroutines are a library of routines to perform common operations on dense matrices. They were originally implemented in FORTRAN-77 in 1979. They remain in wide use today.

The BLAS are organized into three levels

- Level-1: Vector-Vector operations, like element-wise addition
- Level-2: Matrix-Vector operations, like Matrix-vector multiply or solve
- Level-3: Matrix-Matrix operations, like Matrix-Matrix multiply or solve

Hardware Coupling of Level-3 As memory hierarchies have become more complex and as latencies from the upper levels have increased relative to clock cycle times the importance of keeping data local in cache for as many computations as possible has increased. This locality is of primary importance in the Level-3 BLAS, which are characterized by $O(n^3)$ computations on $O(n^2)$ data elements. Through clever organization of the computation into blocks, communication within the slower elements of the memory hierarchy can be hidden, resulting in order-of-magnitude performance gains.

In fact, Level-3 BLAS operations are one of the few cases where computational intensity can match the imbalance in CPU-Memory speeds, making them highly desirable operations on modern hardware. This benefit is even more significant in the case of many-core accelerators, such as graphics processing units (GPUs).

Linear Algebra Package (LAPACK) The Linear Algebra Package (LAPACK) is a library that builds on BLAS to solve more complex problems in dense numerical linear algebra. LAPACK includes routines for the solution of systems of linear equations, matrix factorizations, eigenvalue problems, etc.

Algorithms for the solution of these operations often require standard operations on dense matrices. Where possible LAPACK depends on BLAS for these operations. This isolates the majority of hardware-specific optimizations to the BLAS library, allowing LAPACK to remain relatively high-level. Optimizations to BLAS improve LAPACK without additional development.

Expert LAPACK Subroutines LAPACK retains the computationally intense characteristic of Level-3 BLAS and so can provide highly performant solutions. However the expert use of LAPACK requires several additional considerations including new storage formats and a selection between multiple valid subroutines.

LAPACK operations like matrix factorizations can often be solved by multiple algorithms. For example matrices can be factored into LU or QR decompositions. The Cholesky variant of LU can be used only if the left side is symmetric positive definite. These redundant algorithms are simultaneously included in LAPACK, yielding a large library with thousands of individual routines, a collection of which might be valid in any situation. Additionally LAPACK internally makes use of utility functions (like matrix permutation) and special storage formats (like banded matrices), further adding to a set of high-level matrix operations.

Interface These subroutines do not adhere to a hierarchically structured interface. Instead BLAS/LAPACK provides a flat, low-level interface. Parameters to BLAS/LAPACK routines include scalars (real, complex, integer) of varying precision, arrays of those types, and strings. These types are widely implemented in general purpose programming languages. As a result many numerical packages in other languages link to BLAS/LAPACK, extending their use beyond Fortran users. In particular array-oriented scripting languages like MatLab, R, and Python/NumPy rely on BLAS/LAPACK routines for their array operators.

However, simplicity of parameter types significantly increases their cardinality. In higher level languages array objects often contain fields for a data pointer, shape, and stride/access information. In BLAS/LAPACK these must be passed explicitly.

Many different algorithms exist for matrix problems with slightly different structure. BLAS and LAPACK implement these different algorithms in independent subroutines with very different subroutine headers. For example the routine GEMM performs a Matrix-Matrix multiply in the general case, `SYMM` performs a Matrix-Matrix multiplication when one of the matrices is symmetric, and `TRMM` performs a Matrix-Matrix Multiplication when one of the matrices is triangular. A combination of the quantity of different algorithms, multiple scalar types, and lack of polymorphism causes BLAS and LAPACK to contain over two thousand routines.

For concreteness, examples of the interfaces for `GEMM` and `SYMM` for double precision real numbers are included below:

- `DGEMM` - **D**ouble precision **GE**neral **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`
- `DSYMM` - **D**ouble precision **SY**mmetric **M**atrix **M**ultiply – $\alpha AB + \beta C$
  - `SUBROUTINE DSYMM(SIDE,UPLO,M,N,ALPHA,A,LDA,B,LDB,BETA,C,LDC)`

Challenges The interface to BLAS/LAPACK was standardized in 1979 within the scope of the Fortran-77 language. Memory locations, array sizes, strides, and transposition are all stated explicitly and independently. Modern language assistance like basic type checking or wrapping shape and stride information into array objects is unavailable.

The interface to BLAS/LAPACK appeals to a very low and common denominator. This design makes it trivial to interoperate with a broad set of languages. For example the popular Fortran to Python wrapper `f2py` handles most of the BLAS/LAPACK library without additional configuration. Unfortunately this same low and common denominator does not facilitate direct use by naive scientific users.

Analysis BLAS and LAPACK are sufficiently entrenched and widely supported to be a stable and de facto programming interface in numeric computing. This stability causes two notable attributes:

- Durability: Today BLAS/LAPACK are implemented *and optimized* for most relevant hardware. For example nVidia released `cuBLAS`, an implementation of the original `BLAS` interface in CUDA shortly after GPGPU gained popularity. We can be relatively confident that this support will continue for new architectures into the future.
- Age: The interface from 1979 is not appropriate for modern programmers.

2.3.2 Implementations

The BLAS/LAPACK interface has multiple implementations. These stress a variety of techniques. We list them both as a review of past work and also to demonstrate the wealth of techniques used to accelerate this set of important operations.

Reference BLAS A standard implementation of BLAS remains available in both Fortran and C. It implements the subroutines in a decent and human understandable manner.

Automatically Tuned Linear Algebra Software (ATLAS) The ATLAS[6] system produces a BLAS/LAPACK implementation specialized to a particular hardware architecture. It does this by empirically benchmarking BLAS/LAPACK directly against that target architecture. It tunes various parameters like block-size to fit the memory hierarchy and selects between various valid execution paths within LAPACK. ATLAS was the first successful use of automated methods in this domain and remains in widespread use today. It is the commonly installed software solution on standard Linux distributions.

GOTO BLAS Instead of searching a parameter space the BLAS can be optimized by hand. Kazushige Goto, a single developer, hand-tunes the generated assembly of BLAS for particular architectures. GOTO BLAS[7] is frequently among the fastest implementations available, routinely beating vendor supplied implementations. This implementation is an example of a single expert in low-level computation and memory hierarchy distributing expertise through software.

Formal Linear Algebra Methodology Environment (FLAME) The FLAME project[8, 9, 10] provides a language for the high-level description of block matrix algorithms. From these descriptions it generates low-level code. Through this approach it strives to lower entry barriers. Additionally this project ships a competitive BLAS/LAPACK library generated using their methods.

Math Kernel Library (MKL) The MKL is an industry standard. It is a professional implementation for multi-core Intel processors.

Distributed Memory BLAS/LAPACK Implementations The ubiquity of numerical linear algebra makes it an attractive candidate for mature parallel solutions. All computational kernels expressible as BLAS/LAPACK computations may be automatically parallelized if a robust solution can be found for distributed numerical linear algebra. Much of this work exists for sparse systems that are not part of the BLAS/LAPACK system. See notes on PETSc and Trilinos in 2.4 and 2.5 for more details.

In the case of dense linear algebra, data parallelism is most often achieved through blocking. Input arrays are blocked or tiled and then each operation distributes computation by managing sequential operations on blocks on different processors. A distributed GEMM may be achieved through many smaller sequential GEMMs on computational nodes. More sophisticated computations, like SYMM or POSV, may make use of a variety of sequential operations.

Occasionally communication is handled by a separate abstraction.

ScaLAPACK is the original widespread implementation of LAPACK for distributed memory architecture. ScaLAPACK[11] depends on BLACS[12], a library for the communication of blocks of memory, to coordinate computation of linear algebra routines across a parallel architecture. ScaLAPACK was the first major parallel implementation

Elemental Elemental[13] breaks the tie between algorithm and distribution block size, using element-sized distribution blocks (1×1). This improves load balancing at the cost of novel communication patterns. It also follows more modern software engineering practices than older ScaLAPACK style systems.

Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) Uses dynamic scheduling techniques to communicate tiles in a shared memory architecture. PLASMA is actively developed to support distributed memory[14].

DPLASMA The distributed memory variant of Plasma, depends on DAGuE[15], a "hardware aware" dynamic scheduler to manage its tile distribution.

Matrix Algebra on GPU and Multicore Architectures (MAGMA) is co-developed alongside PLASMA to support heterogeneous architectures with thought to their eventual merger[16].

Analysis The development history of BLAS/LAPACK closely mirrors the development history of computational hardware; hardware developments are closely followed by new BLAS/LAPACK implementations. We can expect BLAS/LAPACK development to continue aggressively into the future. This is particularly true as architecture development seems to have entered an age of experimentation as the need to ameliorate the effort of the power wall spurs the growth of parallel architectures; BLAS/LAPACK development seem to follow suit with a focus on distributing and scheduling tiled computations.

Robust BLAS/LAPACK implementations lag hardware development by several years. This lag propagates to scientific codes because so many link to BLAS/LAPACK libraries. Accelerating development and introducing hardware flexibility can meaningfully advance performance on mainstream scientific codes.

2.3.3 General Array Programming

It is worth noting that while BLAS/LAPACK cover many matrix operations they fail to address the entire space of multi-dimensional array operations. Array programming with high level abstractions can confer both numeric performance benefits (e.g. through loop fusion) and also programming performance, by providing a set of high-level primitives. By using array objects or terms rather than pointers to sequences of scalars we enable the mathematical programmer to more cleanly communicate his intent to the computer which is then able to more efficiently generate low-level code to solve the intended problem.

Array programming with high level abstractions is an old idea, going back as far as the 1960s with the Array Programming Language (APL)[17].

These ideas have been widely adopted in a variety of projects. Common scripting languages, like MatLAB, R, IDL, and Python-NumPy, implement array abstractions with hooks to low-level codes. Various C++ libraries like Armadillo[18], Eigen[19], Expression Templates[20], Blaze[21, 22], also provide high-level syntax to array operations. These sorts of projects often leverage C++'s templating system to perform extra optimizations like loop fusion or better use of in-place execution before the traditional compile time.

2.4 BLAS, LAPACK, PETSC, FEniCS

The numerical methods community has a good record of developing performant libraries in isolation from any particular application. These methods are developed by dedicated groups with mature software engineering practices. They are used in a broad set of applications.

In this section we examine a stack of numeric libraries for the solution of differential equations.

- BLAS
- LAPACK
- PETSc
- FEniCS

As just discussed, BLAS is a library for simple dense matrix operations (e.g. matrix multiply) and LAPACK is a similar library for more complex matrix operations (e.g. Cholesky Solve) that calls on BLAS for most manipulations. PETSc builds on MPI, BLAS, and LAPACK to implement numeric solvers for distributed and sparse problems. FEniCS is a high level domain specific language that transforms a high-level description of PDEs into C++ code that assembles matrices and then calls PETSc routines. Each of these libraries builds off of the layers beneath, making use of preexisting high-quality implementations when available.

In a sense this style of hierarchical modularity is like a Russian doll. Each new higher-level project can omit a substantial computational core by depending on generally applicable previous solutions. New higher level projects must create a shell rather than a complete solid.

Analysis The solution of numerical PDEs is a relevant problem in several applied domains. The ability to generate high performance low-level codes from high-level descriptions automatically is both encouraging and daunting. FEniCS development was greatly assisted by PETSc which was in turn greatly assisted by LAPACK, which was in turn greatly assisted by BLAS.

The different projects adhere to clear interfaces, enabling swappability of different implementations. This observation is particularly relevant for BLAS/LAPACK for which a healthy set of competing implementations exist and continues to develop. FEniCS also provides support for PETSc's peer, Trilinos which is discussed further in Section 2.5. Unlike BLAS/LAPACK, the relevant interface for the PETSc/Trilinos layer has not been standardized, requiring explicit pairwise support in the FEniCs codebase.

2.5 Trilinos

Trilinos is a success story for flat modular scientific design. Trilinos grew from an original three complementary projects into a loose federation of over fifty packages, each developed by an independent team. These packages interoperate through a set of C++ interfaces for generic solver types (e.g. Eigensolve). Trilinos has grown into a robust and powerful ecosystem for numeric computing.

The organization of Trilinos differs from the BLAS-LAPACK-PETSc-FEniCS stack. Trilinos packages operate largely as peers rather than in a strict hierarchy. Trilinos does not dominate its domain like BLAS/LAPACK, but it does demonstrate the value of prespecified complex interfaces in a higher level setting. A number of differently-abled groups are able to co-develop in the same space with relatively little communication.

Potential developers are enticed into this ecosystem with the following:

- A standardized testing and documentation framework
- A high-level distributed array data structure
- Functionality from other packages
- Name recognition and an established user-base

These incentives are essential for the creation of an active ecosystem. Unfortunately, Trilinos often suffers from significant software distribution and building overhead. The lack of centralized control and wide variety of dependencies required by various packages results in substantial startup cost for novice developers.

2.6 Automation

We often use compilers like `gcc` to automatically translate low-level human readable codes like C to machine code. In general a compiler transforms code from one language to another. Usually this transformation moves from high to low-level codes, moving from human comprehension to machine comprehension. During this transformation compilers often apply optimizations on the high-level code known to increase performance at the low level. For example `gcc` might perform optimizations like constant folding or loop unrolling if it is able to infer that they are applicable in the source code. Compilers from higher level languages may be able to better infer and encode higher-level optimizations.

Domain specific languages extend the idea of automated translation of code to application domains. These often translate between a high-level language intended for domain practitioners into a low-level language intended for efficient computation. Domain specific languages exist for a wide variety of domains ranging from hard science to web development. Below we highlight some cases in numerical computing relevant to this work.

The FEniCS project[23] is a collection of software packages for the automated solution of differential equations. It translates a mathematical description differential equations, meshes, boundary conditions, etc. into a sparse matrix problem that is solved with low-level and parallel code.

Spiral[24] compiles a high-level description of a signals processing system into low-level C code optimiezed for particular specification of computational hardware. Notably Spiral also uses internal rewrite rules, much in line with the methods that we will present in 5.

The Tensor Contraction Engine[25] accelerates quantum chemistry problems by encoding and leveraging tensor contractions inspired by physical laws. Like Sprial, TCE includes hardware specific optimizations alongside mathematical ones, fitting tile sizes comfortably within the computational hardware's memory hierarchy.

Built-to-Order BLAS[26] enables the fusion of high-level operations within the BLAS family, reducing the number of requisite passes over memory.

Work by Bientinesi and Fabregat-Traver[27, 28] automatically translates high level matrix expressions into a sequence of appropriate BLAS and LAPACK calls, selecting appropriate routines based on mathematical inference provided by rules encoded into Mathematica. This work is similar to the work we present in Sections 4, 6. They also deal with iterative algorithms and provide a search strategy specific to matrix compilation, a problem that we discuss in 5.3.

Automation has long been used in general purpose programming to facilitate the creation of efficient low-level code by those not sufficiently expert to create it by hand. When we elevate this idea to higher levels of abstraction it is well poised to assist with the training disparity in scientific computing.

In each of the cases above, the automated system was built more-or-less from scratch by mature teams that were simultaneously skilled both in a particular domain and in software engineering. While these projects demonstrate the value of this particular mixture of expertise, their stories also demonstrate the high requirements to accomplish such a project and put it into production to the assistance of that domain.

Just as these projects listed above strive to automate a particular domain there exists various metaprogramming projects which strive to automate the automation of domains. We discuss these efforts further in Section 5.4.

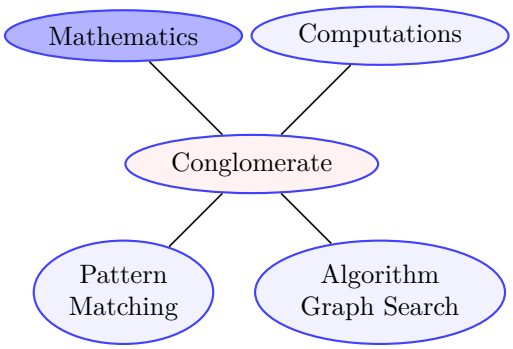
Much of the work that follows investigates the process of constructing domain specific automation, particularly in a linear algebra compiler similar to that of Bientinesi and Fabregat-Traver. We strive to separate this process into pieces that can be reused in other domains (see Section 8) and can be developed in isolation by single-domain experts.

2.7 Conclusion

Scientific software development has a rich history with a variety of approaches from which we can draw insight. High-level decisions in interfaces, data structures, and community have strong impacts on the practicality of the code, perhaps eclipsing the virtues of the low-level performance qualities. Interfaces supporting modular development can succeed where guaranteed federal funding and development fail.

3 Computer Algebra

This dissertation promotes the acceleration of scientific computing through the automated use of expert methods, many of which are mathematical in nature. It also supports the separation of different domains of expertise into different modules. This chapter discusses the design of computer algebra systems, the traditional home of automated mathematics and then builds an isolated module to encapsulate the particular domain of mathematical linear algebra.



In Sections 3.2, 3.1 we describe the basic design and history of computer algebra. In Sections 3.3 and 3.4 we describe SymPy, a particular computer algebra system on which the work of this dissertation depends. In Sections 3.5, 3.6 we present an extension of SymPy to linear algebra. Finally in 3.7 we motivate the use of computer algebra in the generation of numeric codes through a brief experiment.

3.1 Background

Computer algebra is the solution of mathematical problems via the manipulation of mathematical terms. This symbolic approach contrasts with the more popular numerical methods of computation. Computer algebra methods maintain the mathematical meaning of their arguments longer, enabling sophisticated analysis further along in the computation. Unfortunately, many real-world problems are intractable using symbolic methods, often due to irregular inputs. For example computing the heat flow through an engine requires a description of the shape of the engine, which may be difficult to describe symbolically. Even if purely symbolic inputs are available symbolic solutions often suffer from expensive combinatorial blowup. Symbolic methods are often preferable if analytic solutions exist because they retain substantially more information and may provide additional insight. Their lack of robustness has largely relegated them to pure mathematics, cryptography, and education.

History of Computer Algebra The first computer algebra system (CAS), Macsyma[29], was developed in 1962. The field grew slowly both in academic scope and in software. Commercial and open computer algebra systems like Maple and GAP began appearing in the 1980s. These were primarily used in pure mathematics research. Some systems were specialized to particular fields while others were general. For example GAP was specialized to algebraic group theory while Maple remained generally applicable. The community around a project often defined its function more than the language design.

The popular solution Mathematica was initially released in 1988 and grew, alongside Maple, to include numeric solvers, creating an "all in one" development environment. This trend was copied by the Sage project which serves as a fully featured mathematical development environment within the open software community.

Computation The majority of computer algebra research applies automated methods within pure mathematics. However, as early as 1988 computer algebra systems were also used to generate computational codes in Fortran[30]. The automated numerical computation subfield has remained minor but persistent within the CAS community.

Increasing disparity between FLOP and memory limits in modern architectures increasingly favor higher order computational methods. These methods perform more work on fewer node points, providing more accurate computations with less memory access at the cost both of additional computation and substantially increased development time. This development time is largely due to increased mathematical expression complexity and the increased demand of theoretical knowledge (e.g. knowing the attributes of a particular class of polynomials.)

For example Simpson's rule for numeric quadrature can replace the rectangle or trapezoidal methods. In the common case of fairly smooth functions Simpson's rule achieves $O(\delta x^3)$ errors rather than $O(\delta x)$ or $O(\delta x^2)$ for the rectangle and trapezoidal rules respectively. This comes at the following costs:

- Computation: Simpson's rule evaluates three points on each interval instead of two. It also uses extra scalar multiplications.
- Development: Simpson's rule is less intuitive. Parabolic fits to node points are significantly less intuitive both visually and symbolically than quadrilateral or trapezoidal approximations.

Increased FLOP/Memory ratios hide the cost of extra computation. Computer algebra can hide the cost of development and mathematical understanding. With sufficient automation this cost can be almost eliminated, encouraging the automated analysis and use of a wide range of higher order methods.

Computer algebra for automated computation remains a small but exciting field that is able to leverage decades of computer algebra and mathematical research to substantially mitigate the rising costs of mathematically complex scientific computation. This growth is orthogonal to contemporary developments in hardware.

3.2 Design

To describe and leverage mathematics to generate efficient programs we must first describe mathematics in a formal manner amenable to automated reasoning. In this section we describe the basic design of most computer algebra systems.

Data Structure Computer Algebra Systems enable the expression and manipulation of mathematical terms. In real analysis a mathematical term may be a literal like `5`, a variable like `x` or a compound term like `5 + x` composed of an operator like `Add` and a list of child terms `(5, x)`

We store mathematical terms in a tree data structure in which each node is either an operator or a leaf term. For example the expression $\log(3e^{x+2})$ can be stored as shown in Figure 1.

Manipulation A computer algebra system collects and applies functions to manipulate these tree data structures/terms. A common class of these functions perform mathematical simplifications returning mathematically equivalent but combinatorially simpler expression trees. Using the example $\log(3e^{x+2})$, we can expand the log and cancel the log/exp to form $x + 2 + \log(3)$; see Figure 2.

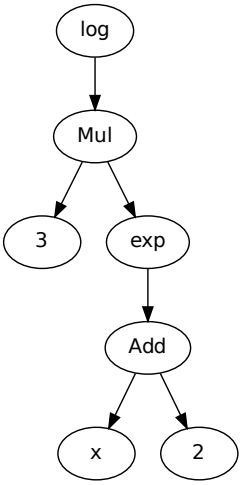


Figure 1: An expression tree for $\log(3e^{x+2})$

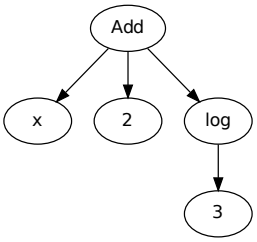


Figure 2: An expression tree for $x + 2 + \log(3)$

Extensions Systems exist for the automatic expression of several branches of mathematics. Extensive work has been done on traditional real and complex analysis including derivatives, integrals, simplification, equation solving, etc.... Other theories such as sets, groups, fields, polynomials, abstract and differential geometry, combinatorics and number theory all have similar treatments. The literals, variables, and manipulations change but the basic idea of automatic manipulation of terms remains constant.

3.3 SymPy Algebra

A computer algebra system is composed of a data structure to represent mathematical expressions in a computer, a syntax for the idiomatic construction of trees by mathematical users, and a collection of functions for common mathematical routines.

SymPy [31] is a computer algebra system embedded in the Python language. It implements these pieces as follows:

Operators Both SymPy operators and SymPy literal types are implemented as Python classes.

```
# Literal Types
class Symbol(Expr):
    ...
class Integer(Expr):
    ...

# Operators
class Add(Expr):
    ...
class Mul(Expr):
    ...
class Log(Expr):
    ...
```

Literal and variable terms are instantiated as Python objects. Standard Python variables are used for identifying information.

```
one = Integer(1)  # 1 is a Python int; one is a SymPy Integer
x = Symbol('x')   # 'x' is a Python string; x is a SymPy Symbol
```

Compound terms are also instantiated as Python objects. These contain SymPy terms as children.

```
y = Add(x, 1)
z = Mul(3, Add(x, 5), Log(y))
```

Every term can be fully described by its operation, stored as its type, and its children, stored as the instance variable `.args`

```
>>> type(y)
Add
>>> y.args
(x, 1)
```

At the lowest level SymPy manipulations are simply Python functions that inspect these terms, manipulate them with Python statements, and return the new versions.

Syntax and Printing Reading and writing terms like `z = Mul(3, Add(x, 5), Log(y))` can quickly become cumbersome, particularly for mathematical users generating large complex equations.

Because SymPy is embedded in a pre-existing language it can not define its own grammar but must instead restrict itself to the expressiveness of the host language. To support mathematically idiomatic term construction, operator and literal classes overload hooks for Python operator syntax like `__add__`, `__mul__`, and `__call__`. To support mathematically idiomatic textual representation these classes also overload hooks for interactive printing

like `__str__`. SymPy also implements printers for LaTeX and unicode which can be called on by the ubiquitous `ipython` console and notebook tools. Most mathematical text in this document is automatically generated by SymPy. Together these hooks provide an intuitive interactive experience for mathematical users

```
>>> from sympy import Symbol, log, exp, simplify
>>> x = Symbol('x')
>>> y = log(3*exp(x + 2))
>>> print y
log(3*exp(x + 2))

>>> print simplify(y)
x + log(3) + 2
```

Relation to Other Computer Algebra Systems SymPy differs from existing computer algebra systems in the following ways

Application SymPy is focused more around scientific and computational applications and less around pure mathematics. This choice of focus is largely due to the focus of the developer community. SymPy is one of six core modules commonly associated with the scientific Python software stack.

Library Other dominant computer algebra systems today serve as monolithic fully-featured development environments. They surround a computer algebra system core with specific numeric, visual, and database components. Interoperation with other software systems (e.g. application specific scientific codes) is rare. In contrast SymPy is intended to be imported as a library within other applications. It aggressively limits its scope, providing interfaces to other popular systems for visualization, computation, etc.,instead of shipping with particular implementations.

Pure Python SymPy is written in Python, a common language for accessible scientific computing. Because SymPy restricts itself to pure Python rather than relying on C extension modules it sacrifices performance to enable trivial build and installation for ease of access and broad interoperation. This feature is often cited by users as a primary motivation for the choice of SymPy over other computer algebra systems.

3.4 SymPy Inference

Later work in this dissertation will require inference over mathematical terms. We discuss this element of SymPy now in preparation. We often want to test whether algebraic statements are true or not in a general case. For example,

Given that x is a natural number and that y is real, is $x + y^2$ positive?

To create a system capable of posing and answering these questions we need the following components:

1. A set of predicates:
   positive and *real*
2. A syntax to pose facts and queries:
   Given that x is positive, is $x + 1$ positive?
3. A set of relations between pairs of predicates:
   x is a natural number implies that x is positive.
4. A set of relations between predicates and operators:
   The addition of positive numbers is positive or
   The square of a real number is positive
5. A solver for satisfiability given the above relations:

These components exist in SymPy. We describe them below.

A set of predicates A set of predicates is collected inside the singleton object, `Q`

```
Q.positive
Q.real
....
```

These Python objects serve only as literal terms. They contain no logic on their own.

A syntax for posing queries Predicates may be applied to SymPy expressions.

```
context = Q.positive(x) & Q.positive(y)
query   = Q.positive(x + y)
```

The user interface for query is the `ask` function.

```
>>> ask(query, context)
True
```

Predicate-Predicate Relations A set of predicate relations is stated declaratively in the following manner

```
Implies(Q.natural, Q.integer)
Implies(Q.integer, Q.real)
Implies(Q.natural, Q.positive)
```

For efficiency, forward chaining from these axioms is done at code-distribution time and lives in a generated file in the source code, yielding a second set of generated implications that contains, for example

```
Implies(Q.natural, Q.real)
```

Predicate-Operator Relations The relationship between predicates and operators is described by low-level Python functions. These are organized into classes of static methods. Classes are indexed by predicate, and methods are indexed by operator name. Logical relations are encoded in straight Python. For example:

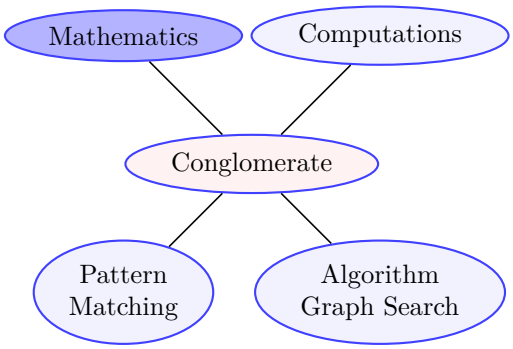
```
class AskPositiveHandler(...):
    @staticmethod
    def Add(expr, assumptions):
        """ An Add is positive if all of its arguments are positive """
        if all(ask(Q.positive(arg, assumptions) for arg in expr.args)):
            return True
```

Testing for Satisfiability SymPy assumptions relies on the Davis–Putnam–Logemann–Loveland algorithm for solving the CNF-SAT problem[32]. This algorithm is separable from the rest of the codebase. This solver accesses the predicate-predicate and predicate-operator relations defined above.

The separation of the SAT solver enables the mathematical code to be very declarative in nature. This system is trivial to extend.

3.5 Matrix Algebra

We now extend the SymPy computer algebra system to matrix algebra. Leaf variables in a matrix algebra are defined by an identifier (e.g. `'X'`) and a shape, two integers of rows and columns. These shape integers may themselves be symbolic. Common matrix algebra operators include Matrix Multiplication, Matrix Addition, Transposition, and Inversion. Each of these operators has its own logic about the shape of the term given the shapes of its inputs, validity, and possible simplifications.



In the end we enable the construction of expressions such as the following for least squares linear regression in which X is an $n \times m$ matrix and y an $n \times 1$ column vector.

$$\beta = (X^T X)^{-1} X^T y$$

Simplification Just as we simplify $\log(e^x) \to x$ we know trivial simplifications in matrix algebra. For example $(X^T)^T \to X$ or $\text{Trace}(X + Y) \to \text{Trace}(X) + \text{Trace}(Y)$.

Extension As with real analysis, matrix algebra has a rich and extensive theory. As a result this algebra can be extended to include a large set of additional operators including Trace, Determinant, Blocks, Slices, EigenVectors, Adjoints, Matrix Derivatives, etc. Each of these operators has its own rules about validity and propagation of shape, its own trivial simplifications, and its own special transformations.

Embedding in SymPy We implement this matrix algebra in the SymPy language. As shown Section 3.3 we implement the literals and operators as Python classes.

```
# Literals
class MatrixSymbol(MatrixExpr):
    ...
class Identity(MatrixExpr):
    ...
class ZeroMatrix(MatrixExpr):
    ...

# Operators
class MatAdd(MatrixExpr):
    ...
class MatMul(MatrixExpr):
    ...
class Inverse(MatrixExpr):
    ...
class Transpose(MatrixExpr):
    ...
```

In this case however matrix expression "literals" contain not only Python variables for identification, but also SymPy scalar expressions like `Symbol('n')` for shape information.

We can encode the least squares example above in the following way

```
>>> n = Symbol('n')
>>> m = Symbol('m')
>>> X = MatrixSymbol('X', n, m)
>>> y = MatrixSymbol('y', n, 1)
>>> beta = MatMul(Inverse(MatMul(Transpose(X), X)), Transpose(X), y)
```

The execution of these commands does not perform any specific numeric computation. Rather it builds an expression tree that can be analyzed and manipulated in the future.

Syntax As in Section 3.3 we overload Python operator methods `__add__`, `__mul__` to point to `MatAdd` and `MatMul` respectively. We use Python `properties` to encode `.T` as Transpose and `.I` as inverse. This approach follows the precedent of `NumPy`, a popular library for numerical linear algebra. These changes allow a more familiar syntax for mathematical users.

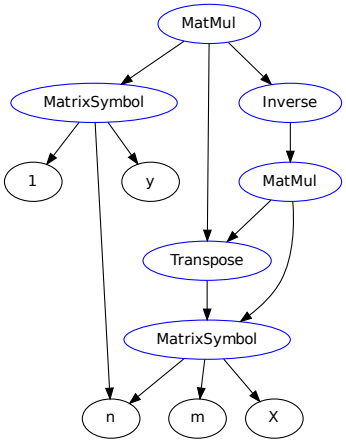


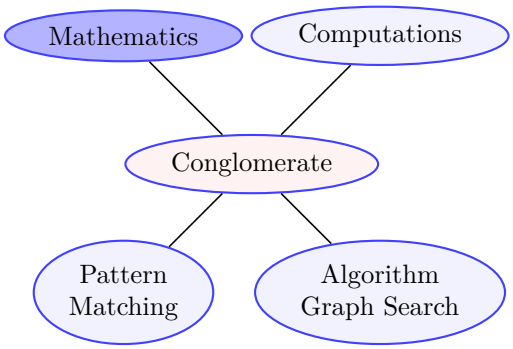
Figure 3: Expression tree for least squares linear regression

```
>>> # beta = MatMul(Inverse(MatMul(Transpose(X), X)), Transpose(X), y)
>>> beta = (X.T*X).I * X.T * y
```

Shape Checking and Trivial Simplification Shape checking and trivial simplifications, e.g. the removal of nested transposes, are done at object instantiation time. This task is accomplished by calling raw Python code within the class `__init__` constructors.

3.6 Matrix Inference

In Section 3.4 we saw that SymPy supports the expression and solution of logical queries on mathematical expressions. In this section we extend this system to matrix algebra.



Inference Problems Matrices can satisfy a rich set of predicates. A matrix can have structural attributes like symmetry, upper or lower triangularity, or bandedness. Matrices can also exhibit mathematical structure like invertibility, orthogonality, or positive definiteness. Matrices also have basic field types like real, complex, or integer valued elements. Rich interactions exist between these predicates and between predicate/operator pairs. For example positive definiteness implies invertibility (a predicate-predicate relation) and the product of invertible matrices is always invertible (a predicate-operator relation).

Example In Section 3.4 we posed the following example:

Given that x is a natural number and that y is real, is $x + y^2$ positive?

An analogous example in matrix algebra would be the following:

Given that $\mathbf{A}$ is symmetric positive-definite and $\mathbf{B}$ is fullrank, is $\mathbf{B} \cdot \mathbf{A} \cdot \mathbf{B}^\top$ symmetric and positive-definite?

To teach SymPy to answer this question we supply the same information as in the scalar case:

1. A set of predicates
2. A set of predicate-predicate relations
3. A set of predicate-operator relations

Fortunately the syntax and SAT solver may be reused. The only elements that need to be generated for this case are relatively declarative in nature.

Predicates We provide the following predicates:

```
positive_definite, invertible, singular, fullrank, symmetric,
orthogonal, unitary, normal, upper/lower triangular, diagonal, square,
complex_elements, real_elements, integer_elements
```

Predicate-Predicate Relations Many of these predicates have straightforward relationships. For example:

```
Implies(Q.orthogonal, Q.positive_definite)
Implies(Q.positive_definite, Q.invertible)
Implies(Q.invertible, Q.fullrank)
Equiavlent(Q.fullrank & Q.square, Q.invertible)  # & operator connotes "and"
Equivalent(Q.invertible, ~Q.singular) # ~ operator connotes "not"
...
```

From these a wider set of implications can be inferred at code generation time. Such a set would include trivial extensions such as the following:

```
Implies(Q.orthogonal, Q.fullrank)
```

Predicate-Operator Relations As in 3.4 the relationship between predicates and operators may be described by low-level Python functions. These are organized into classes of static methods where classes are indexed by predicate and methods are indexed by operator.

```
class AskInvertibleHandler(...):
    @staticmethod
    def MatMul(expr, assumptions):
        """ MatMul is invertible if all arguments are invertible """
        if all(ask(Q.invertible(arg, assumptions) for arg in expr.args)):
            return True
```

Example revisited We posed the following question above:

Given that $\mathbf{A}$ *is symmetric positive-definite and* $\mathbf{B}$ *is fullrank, is* $\mathbf{B} \cdot \mathbf{A} \cdot \mathbf{B}^\top$ *symmetric and positive-definite?*

We are now able to answer this question using SymPy.

```
>>> A = MatrixSymbol('A', n, n)
>>> B = MatrixSymbol('B', n, n)

>>> context = Q.symmetric(A) & Q.positive_definite(A) & Q.fullrank(B)
>>> query   = Q.symmetric(B*A*B.T) & Q.positive_definite(B*A*B.T)

>>> ask(query, context)
True
```

This particular question is computationally relevant. It arises frequently in scientific problems and significantly more efficient algorithms are applicable when it is true. Unfortunately relatively few scientific users are able to recognize this situation. Even when this situation is correctly identified developers may not be able to take advantage of the appropriate lower-level routines.

SymPy matrix expressions is the first computer algebra system that can answer such questions for abstract matrices. In Section 4 we describe a system to describe the desired subroutines. In Section 6.1 we describe a system to select the desired subroutine given the power of inference described here.

Refined Simplification This advanced inference enables a substantially larger set of optimizations that depend on logical information. For example, the inverse of a matrix can be simplified to its transpose if that matrix is orthogonal.

Linear algebra is a mature field with many such relations. The Matrix Cookbook [33] alone contains thousands of such relations. Formally describing each of these is challenging due both to their quantity and the limited population of practitioners. To address this issue we create a mechanism to describe them declaratively. This mechanism will be discussed further in Section 5.9 after the requisite technology has been developed. Declarative techniques reduce the extent of the code-base with which a mathematician must be familiar in order to encode mathematics. It also increases portability. This reduction in scope drastically increases the domain of qualified developers.

3.7 Mathematical Code Generation

We now give a brief experiment to support the general use of computer algebra in numeric code generation. This example is separate from the work in matrix algebra.

Numerical code is often used to evaluate and solve mathematical problems. Frequently human users translate high-level mathematics directly into low-level code. In this section we motivate the use of computer algebra systems to serve as an intermediate step. This approach confers the following benefits.

1. Automated systems can leverage mathematics deeper in the compilation system
2. Human error is reduced
3. Multiple backends can be used

We will demonstrate these benefits with interactions between SymPy as discussed in section 3.3 and Theano[34], a library for the generation of mathematical codes in C and CUDA.

As an aside we note that this example uses differentiation, an algorithmic transform of contemporary popularity. Automatic differentiation techniques are an application of a small section of computer algebra at the numeric level. Much of this dissertation argues for repeating this experience with other domains of computer algebra beyond differentiation.

Radial Wave Function Computer algebra systems often have strong communities in the physical sciences. We use SymPy to generate a radial wave-function corresponding to `n = 6` and `l = 2` for Carbon (`Z = 6`).

```
from sympy.physics.hydrogen import R_nl
n, l, Z = 6, 2, 6
expr = R_nl(n, l, x, Z)
```

$$\frac{1}{210}\sqrt{70}x^2\left(-\frac{4}{3}x^3+16x^2-56x+56\right)e^{-x}$$

As a case study, we generate code to simultaneously evaluate this expression and its derivative on an array of real values.

Simplification We show the expression, its derivative, and SymPy's simplification of that derivative. In each case we quantify the complexity of the expression by the number of algebraic operations.

The target expression

$$\frac{1}{210}\sqrt{70}x^2\left(-\frac{4}{3}x^3+16x^2-56x+56\right)e^{-x}$$

```
Operations:  17
```

It's derivative

$$\begin{aligned}
& \frac{1}{210}\sqrt{70}x^2\left(-4x^2+32x-56\right)e^{-x} \\
- \quad & \frac{1}{210}\sqrt{70}x^2\left(-\frac{4}{3}x^3+16x^2-56x+56\right)e^{-x} \\
+ \quad & \frac{1}{105}\sqrt{70}x\left(-\frac{4}{3}x^3+16x^2-56x+56\right)e^{-x}
\end{aligned}$$

```
Operations:  48
```

The result of `simplify` on the derivative

$$\frac{2}{315}\sqrt{70}x\left(x^4 - 17x^3 + 90x^2 - 168x + 84\right)e^{-x}$$

```
Operations:  18
```

Note the significant cancellation.

Bounds on the Cost of Differentiation Algorithmic scalar differentiation is a simple transformation. The system must know how to transform all of the elementary functions (`exp, log, sin, cos, polynomials, etc...`) as well as the chain rule; nothing else is required. Theorems behind automatic differentiation state that the cost of a derivative will be at most five times the cost of the original. In this case we're guaranteed to have at most `17*5 == 85` operations in the derivative computation; this bound holds in our case because `48 < 85`.

However, derivatives are often far simpler than this upper bound. We see that after simplification the operation count of the derivative is `18`, only one more than the original. This situation is common in practice but is rarely leverage fully.

Experiment We compute the derivative of our radial wavefunction and then simplify the result. Both SymPy and Theano are capable of these transformations. We perform these operations using both of the following methods:

- SymPy's symbolic derivative and simplify routines
- Theano's automatic derivative and computation optimization routines

We then compare and evaluate the two results by counting the number of algebraic operations.

In SymPy we create both an unevaluated derivative and a fully evaluated and SymPy-simplified version. We translate each to Theano, simplify within Theano, and then count the number of operations both before and after simplification. In this way we can see the value added by both SymPy's and Theano's optimizations.

Theano Only

$$\begin{aligned} & \frac{1}{210}\sqrt{70}x^2\left(-4x^2 + 32x - 56\right)e^{-x} \\ - \quad & \frac{1}{210}\sqrt{70}x^2\left(-\frac{4}{3}x^3 + 16x^2 - 56x + 56\right)e^{-x} \\ + \quad & \frac{1}{105}\sqrt{70}x\left(-\frac{4}{3}x^3 + 16x^2 - 56x + 56\right)e^{-x} \end{aligned}$$

```
Operations:                              40
Operations after Theano Simplification:  21
```

SymPy + Theano

$$\frac{2}{315}\sqrt{70}x\left(x^4 - 17x^3 + 90x^2 - 168x + 84\right)e^{-x}$$

```
Operations:                                13
Operations after Theano Simplification:  10
```

Analysis On its own Theano produces a derivative expression that is about as complex as the unsimplified SymPy version. Theano simplification then does a surprisingly good job, roughly halving the amount of work needed ($40 \to 21$) to compute the result. If you dig deeper however you find that this result arises not because it was able to algebraically simplify the computation (it was not), but rather because the computation contained several common sub-expressions. The Theano version looks a lot like the unsimplified SymPy version. Note the common sub-expressions like `56*x`.

The pure-SymPy simplified result is again substantially more efficient (`13` operations). Interestingly, Theano is still able to improve on this, again not because of additional algebraic simplification, but rather due to constant folding. The two projects simplify in orthogonal ways.

Simultaneous Computation When we compute both the expression and its derivative simultaneously we find substantial benefits from using the two projects together.

Theano Only

$$\frac{\partial}{\partial x}\left(\frac{1}{210}\sqrt{70}x^2\left(-\frac{4}{3}x^3 + 16x^2 - 56x + 56\right)e^{-x}\right), \frac{1}{210}\sqrt{70}x^2\left(-\frac{4}{3}x^3 + 16x^2 - 56x + 56\right)e^{-x}$$

```
Operations:                                57
Operations after Theano Simplification:  24
```

SymPy + Theano

$$\frac{2}{315}\sqrt{70}x\left(x^4 - 17x^3 + 90x^2 - 168x + 84\right)e^{-x}, \frac{1}{210}\sqrt{70}x^2\left(-\frac{4}{3}x^3 + 16x^2 - 56x + 56\right)e^{-x}$$

```
Operations:                                27
Operations after Theano Simplification:  17
```

The combination of SymPy's scalar simplification and Theano's common sub-expression optimization yields a significantly simpler computation than either project could do independently.

To summarize:

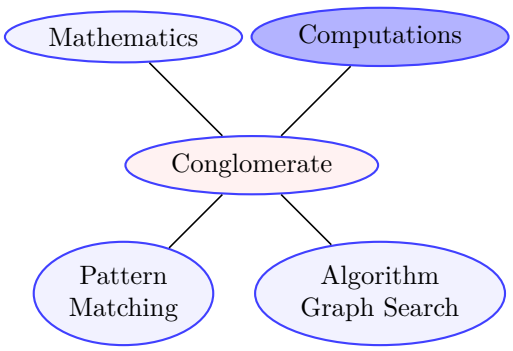
Project	operation count
Input	57
SymPy	27
Theano	24
SymPy+Theano	17

Conclusion Similarly to SymPy, Theano transforms graphs to mathematically equivalent but computationally more efficient representations. It provides standard compiler optimizations like constant folding, and common sub-expressions as well as array specific optimizations element-wise element-wise operation fusion.

Because users regularly handle mathematical terms, Theano also provides a set of optimizations to simplify some common scalar expressions. For example, Theano will convert expressions like `x*y/x` to `y`. In this sense it overlaps with SymPy's `simplify` functions. This section demonstrates that SymPy's scalar simplifications are more powerful than Theano's and that their use can result in significant improvements. This result should not be surprising. Sympians are devoted to scalar simplification to a degree that far exceeds the Theano community's devotion to this topic.

These experiments mostly contain polynomials and exponentials. In this sense they are trivial from a computer algebra perspective. Computer algebra systems are capable of substantially more sophisticated analyses.

4 Computations



In Chapter 3 we described a computer algebra system to express and manipulate mathematical expressions at a high, symbolic level. In Section 3.5 we specialized this to matrix expressions. This symbolic work is not appropriate for numeric computation. In this section we describe numeric routines for computation of matrix subproblems, in particular the BLAS/LAPACK routines. These libraries have old and unstructured interfaces that are difficult to target with high-level automated systems. So in section 4.2 we present a package for the symbolic description of these routines using SymPy matrix expressions so that it can be used as an intermediary. Subsequently we will discuss the use of standard compiler functionality to manage variables and generate Fortran code that calls down to curated libraries. This system

is extensible to support other low-level libraries. We believe that its separation makes it broadly applicable to applications beyond our own.

Specifically we present a small library to encode low-level computational routines that is amenable to manipulation by automated high-level tools. This library is extensible and broadly applicable. This library also supports low level code generation. Later in Chapter 6 we use logic programming to connect these two to build an automated system .

4.1 Background

Low-level Libraries BLAS/LAPACK, FFTW, and MPI are examples of efficient low level libraries/interfaces that serve as computational building blocks of mathematical codes. These codes are commonly linked to by higher level languages (e.g. `scipy.linalg.blas`, `numpy.fft`, `mpi4py`). The high-level packages magnify the utility of the low-level libraries both by providing a more modern interface for novice users and by interconnection with the scientific Python ecosystem. "High productivity" languages like Python, Matlab, and `R` are very popular within scientific communities. The dual needs of performance and accessibility are often met with such links to lower level refined code.

Compiled Python Occasionally this set of high-level operators is insufficient to describe a computational task, forcing users to revert to execution within the relatively slow CPython/Matlab/R runtimes. Also, even if the high-level operators can describe a computation the interpreted nature of the Python runtime may not be able to take advantage of transformations like operation fusion, in-place computation, or execution on a GPU.

Dozens of projects have attempted to address these issues in recent years by compiling user-friendly Python code into more performant low-level code (see list in Section 2.1.) These traditionally annotate Python functions with light type information, use this information to generate low-level C/C++ code, wrap this code with the appropriate CPython bindings and expose it to the Python level. Care is taken so that this transformation is hidden from the scientific user.

Problems with Current Approaches When these projects bind themselves to the CPython runtime they retain some of the less obvious inefficiencies of Python.

On a multicore machine, concurrency between simultaneous operations remains difficult due to Python's global interpreter lock (GIL). In a massively parallel context each process still needs to load the `python` runtime and import the necessary libraries. This import process routinely takes hours on a standard network file system. Resulting codes depend strongly on the Python ecosystem, eliminating opportunities for interaction with other software systems in the future. They are also written in exotic language variants with uncertain longevity and support.

These projects lack a common data structure or framework to share contributions. Dozens of implementations exist which reimplement roughly the same architecture in order to experiment with a relatively small novel optimization.

With regards to this last point this chapter is no different. We implement a small intermediate representation and code generation system in order to demonstrate the values of exclusively

using low-level libraries as primitives and using mathematical inference to inform matrix computation. Fortunately this redundant element is small and does not contain the majority of our logic. The intelligence of this system resides in a far more stable and dominant computer algebra system, SymPy.

4.2 Software

We describe a software system, `computations`, that serves both as both a rudimentary code generation system and as a repository for a high-level description of numeric libraries (particularly BLAS/LAPACK). In describing this system we claim that the high-level coordination of low-level numeric routines can succinctly describe a broad set of computational science.

Every BLAS/LAPACK routine can be logically identified by a set of inputs, outputs, conditions on the inputs, and inplace memory behavior. Additionally each routine can be imbued with code for generation of the inline call in a variety of languages. In our implementation we focus on Fortran but C or scipy could be added without substantial difficulty. CUDA code generation is a current work in progress.

Each routine is represented by a Python class. In this paragraph we describe `SYMM`, a routine for **SY**mmetric **M**atrix **M**ultiply, in prose; just below we describe this same routine in code. `SYMM` fuses a matrix multiply `A*B` with a scalar multiplication `alpha*A*B` and an extra matrix addition `alpha*A*B + beta*C`. This fusion is done for computational efficiency. `SYMM` is a specialized version that is only valid when one of the first two matrices `A` or `B` are symmetric. It exploits this special structure and performs only half of the normally required FLOPs. Like many BLAS/LAPACK routines `SYMM` operates *inplace*, storing the output in one of its inputs. In this particular case it stores the result of the zeroth output, `alpha*A*B + beta*C` in its fourth input, `C`.

```
class SYMM(BLAS):
    """ Symmetric Matrix Multiply """
    _inputs   = [alpha, A, B, beta, C]
    _outputs  = [alpha*A*B + beta*C]
    condition = Q.symmetric(A) | Q.symmetric(B)
    inplace   = {0: 4}
```

Composite Computations Multiple operations like `SYMM` can be joined together to form larger composite computations. Composite computations may be built up from many constituents. Edges between these constituents exist if the output of one computation is the input of the other. Treating computations as nodes and data dependencies as edges defines a directed acyclic graph (DAG) over the set of computations.

Tokenized Computations When we transform DAGs of computations into executable Fortran code we find that the mathematical definition of our routines does not contain sufficient information to print consistent code, particularly due to the considerations of state. Because the atomic computations overwrite memory we must consider and preserve state

within our system. The consideration of inplace operations requires the introduction of `COPY` operations and a treatment of variable names. Consider `COPY` defined below

```
class COPY(BLAS):
    _inputs   = [X]
    _outputs  = [X]
    condition = True
    inplace   = {}
```

Mathematically this definition is correct. It consumes a variable, `X`, and produces a new variable whose *mathematical definition* is exactly `X`. While this definition is mathematically consistent, it lacks computational meaning; the output `X` exists in a new location in memory from the input.

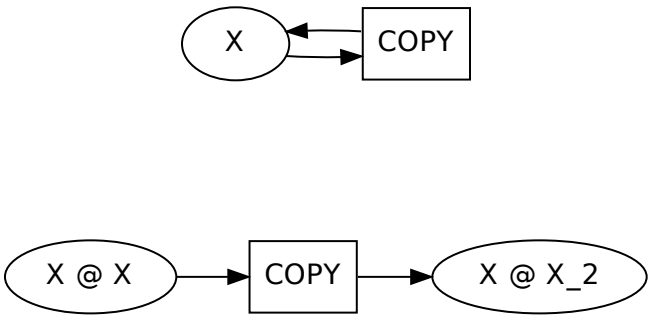


Figure 4: A meaningful 'Copy' operation with variables that contain both mathematical expressions and memory tokens

To encode this information about memory location we expand our model so that each variable is both a mathematical SymPy term and a unique token identifier, usually a Python string. This method supports a new class of transformations to manage the inplace computations common throughout BLAS/LAPACK.

Fortran Code Generation From such a directed acyclic graph we can generate readable low-level code; in particular we focus on Fortran 90. Each atomic computation contains a method to print a string to execute that computation given the correct parameter names. We traverse the directed acyclic graph to obtain variable names and a topological sort of atomic computations. Generating Fortran code from this stage is trivial.

Extensibility This model is not specific to BLAS/LAPACK. A range of scientific software can be constructed through the coordination of historic numeric libraries. Mature libraries exist for several fields of numerics. Our particular system has been extended to support `MPI` and `FFTW`.

Example use of `computations` We now demonstrate `computations` by constructing a simple program. The following example uses `SYMM` and `AXPY`, routines matrix multiplication and vector addition, to create a complex composite computation. It then introduces tokens and copy operations to generate human readable Fortran code.

Specific instances of each computation can be constructed by providing corresponding inputs, traditionally SymPy Expressions. We generate an instance of `SYMM`, here called `symm`, that computes `X*Y` and stores the result in `Y`.

```
>>> n = Symbol('n')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', n, n)
>>> symm = SYMM(1, X, Y, 0, Y)
>>> symm.inputs
[X, Y]
>>> symm.outputs
[X*Y]
```

We now want to take the result `X*Y`, multiply it by `5`, and add it again to `Y`. This can be done with a vector addition, accomplished by the routine `AXPY`. Notice that computations can take compound expressions like `X*Y` as inputs

```
>>> axpy = AXPY(5, X*Y, Y)
>>> axpy.inputs
[X*Y, Y]
>>> axpy.outputs
[5*X*Y + Y]
```

Computations like `symm` and `axpy` can be combined to form composite computations. These are assembled into a directed acyclic graph based on their inputs and outputs. For example, because `symm` produces `X*Y` and `axpy` consumes `X*Y` we infer that an edge much extend from `symm` to `axpy`.

```
>>> composite = CompositeComputation(axpy, symm)
>>> composite.inputs
[X, Y]
>>> composite.outputs
[5*X*Y + Y]
```

The computation above is purely mathematical in nature. We now consider inplace behavior and inject unique tokens as described in Section 4.2. We inject `COPY` operations where necessary to avoid loss of essential data.

```
>>> inplace = inplace_compile(composite)
```

Finally we declare the types of the matrices, specify orders for inputs and outputs, and print Fortran code

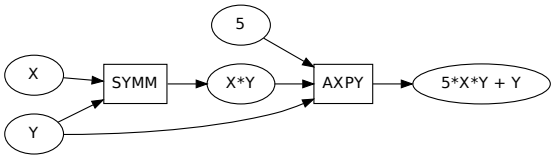


Figure 5: A computation graph to compute $5XY + Y$

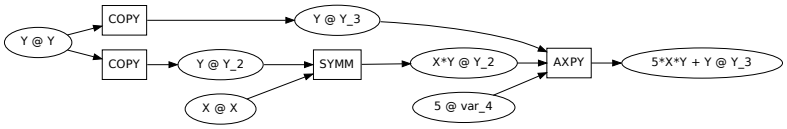


Figure 6: A tokenized computation graph to compute $5XY + Y$

```
>>> with assuming(Q.symmetric(X), Q.real_elements(X), Q.real_elements(Y)):
...     print generate(inplace, [X, Y], [5*X*Y + Y])
```

```
subroutine f(X, Y, Y_2)
    implicit none

    ! Argument Declarations !
    ! ====================== !
    real(kind=8), intent(in) :: X(:,:)
    real(kind=8), intent(in) :: Y(:,:)
    real(kind=8), intent(out) :: Y_2(:,:)

    ! Variable Declarations !
    ! ====================== !
    real(kind=8), allocatable :: Y_3(:,:)
    integer :: n

    ! Variable Initializations !
    ! ======================== !
    n = size(Y, 1)
    allocate(Y_3(n,n))

    ! Statements !
    ! ========== !
    call dcopy(n**2, Y, 1, Y_3, 1)
    call dsymm('L', 'U', n, n, 1, X, n, Y_3, n, 0, Y_3, n)
    call dcopy(n**2, Y, 1, Y_2, 1)
    call daxpy(n**2, 5, Y_3, 1, Y_2, 1)
```

```
    deallocate(Y_3)
    return
end subroutine f
```

5 Term Rewrite System

In Chapter 3 we saw that computer algebra systems represent and manipulate mathematical elements as terms or trees. In this chapter we discuss techniques for the manipulation of terms separately from mathematics. We first motivate the separation of term manipulation into a set of many small transformations and a system to coordinate those transformations in Section 5.1. Then in Section 5.2 we present the use of pattern matching to specify small transformations in mathematical syntax, enabling mathematical users to define transformations without knowledge of the underlying graph representations. In Section 5.3 we describe problems associated with coordinating many such transformations and pose the general topic as a graph search problem. We discuss background and existing solutions to these problems in Section 5.4. We then apply these ideas to the automated generation of expert solutions in matrix computations. We first implement a prototype matrix algebra language in one of these solutions in Section 5.5 and then discuss our final approach to these problems in Sections 5.6-5.10.

Later in Chapter 6 we demonstrate the utility of these tools by implementing a mathematically informed linear algebra compiler with minimal math/compilers expertise overlap. This system translates computer algebra (SymPy) expressions into directed acyclic `computations` graphs.

5.1 Motivation

The mathematical software ecosystem can best be served by the separation of mathematics from software.

The following arguments support this principle

Math changes more slowly than Software Software may change due to evolution in programming languages, radical shifts in hardware, or simply due to the adoption of an old technique by a new community. Conversely much of the mathematics used in computation is well established and changes relatively slowly. By separating the mathematics (a slowly changing component) from the software (a rapidly changing component) we reduce the frequency with with expertise must be rewritten.

Demographics Deep understanding of both computational mathematics and software engineering is held only by a small population of scientific software engineers. Separating mathematics from software reduces the demands of writing and verifying solutions. A larger body of mathematicians can work on the mathematics and a larger body of software engineers can work on the pure software components, rather than all developers needing to know both. The costly practice of collaboration can be avoided.

Definition We use term rewrite systems to enable the separation of mathematics from software. A term rewrite system is composed of the following:

1. A language of terms
2. A collection of isolated transformations on those terms
3. A system to coordinate the application of those transformations

In our case the terms are mathematical expressions, the transformations are known mathematical relations, and the system of coordination is abstracted as a graph search problem. This approach separates mathematics from software. The language and transformations are mathematical while the system for coordination is algorithmic. The isolated nature of the transformations limits the extent to which mathematical programmers need to understand the broader software context. The system for coordination need not depend on the transformations themselves, eliminating the need for mathematical understanding from an algorithmically centered task.

Specifically term rewrite systems confer the following benefits in the context of mathematical computing:

- Mathematical programmers can focus on much smaller units of software
- Algorithmic programmers without mathematical training can be enlisted
- Smaller transformations can be verified more effectively
- Isolated coordination systems can be verified more effectively
- Multiple independent coordination systems can interact with the same set of transformations
- Multiple sets of transformations can interact with the same coordination systems

Example First, a motivating example. Mathematical theories contain many small transformations on expressions. For example consider the cancellation of exponentials nested within logarithms, e.g.:

$$\log(\exp(x)) \to x \quad \forall x \in \mathbb{R}$$

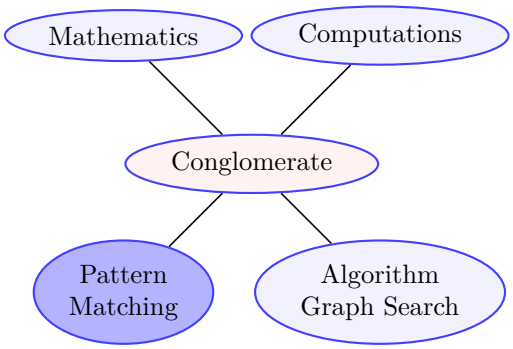
We encode this transformation into a computer algebra system like SymPy by manipulating the tree directly

```
def unpack_log_exp_if_real(term):
    if (isinstance(term, log) and isinstance(term.args[0], exp)
                              and ask(Q.real(x))):
        return term.args[0].args[0]  # unpack both `log` and `exp`
```

We appreciate that this transformation is isolated and compact. The function `unpack_log_exp_if_real` may be one of a large set of transformations, each of which transform terms to other, possibly better terms. This approach of many small `term -> term` functions isolates the mathematics from the coordination of the functions. A mathematical programmer may easily encode several such functions without thinking about how they are applied while an algorithmic programmer may develop sophisticated systems to coordinate these many functions without thinking about the math they represent.

5.2 Pattern Matching

The syntax of mathematics is both more widely understood and more stable than the syntax of programming



The last section argues that by separating mathematics from coordination we can more comfortably engage a wider development pool. This section repeats the same goal by separating math syntax from the term data structures. We consider the class of transformations that can be fully described by only source, target, and condition patterns. We instantiate these transformations using pattern matching. Pattern matching enables the definition of transformations using only the mathematical language of terms (e.g. SymPy) without relying on the implementation (e.g. Python). This separation compounds many of the previously mentioned benefits of term rewrite systems.

1. Rewrite patterns align more closely with the tradition of written mathematics than does general purpose code
2. Development of mathematical transformations is not tied to the implementation, freeing both to be developed at different time scales by different communities.

Motivation As an example we again consider the unpacking of logarithms of exponents.

$$\log(\exp(x)) \to x \quad \forall x \in \mathbb{R}$$

We noted that this transformation can be encoded as a manipulation of a tree within a computer algebra system, SymPy. We appreciated that this algorithmic code was isolated to just a few lines and does not affect the code for coordination. We do not simultaneously require any developer to understand both the mathematics and the coordination of transformations.

```
if isinstance(term, log) and isinstance(term.args[0], exp) and ask(Q.real(x)):
    return term.args[0].args[0]  # unpack both `log` and `exp`
```

However, this method of solution does simultaneously require the understanding of both the mathematics and the particular data structures within the computer algebra system. This approach has two flaws.

1. It restricts the development pool to simultaneous experts in mathematics and in the particular computer algebra system.
2. The solution is only valuable within this particular computer algebra system. It will need to be rewritten for future software solutions.

These flaws can be avoided by separating the mathematics from the details of term manipulation. We achieve this separation through the description and matching of patterns. We use the mathematical term language to describe the transformations directly, without referring to the particular data structures used in the computer algebra system.

Rewrite Patterns We define a rewrite pattern/rule as a source term, a target term, and a condition, each of which is a term in the mathematical language. For example, the $\log(\exp(\cdot))$ transformation can be decomposed into the following pieces:

$$\log(\exp(x)) \to x \quad \forall x \in \mathbb{R}$$

- Source: $\log(\exp(x))$
- Target: x
- Condition: $x \in \mathbb{R}$

Each of these elements may be encoded in the computer algebra system (SymPy) without additional support from the general purpose language (Python). We encode them below in a `(source, target, condition)` tuple.

```
( log(exp(x)),       x,      Q.real(x) )
```

Using these rewrite patterns we reduce the problem of transformation to matching incoming terms against the source pattern, checking the condition, and then reifying these values into the target pattern. These pattern matching operations can be dealt with outside the context of mathematics. Mature solutions already exist, largely stemming from other work in logic programming languages and theorem provers.

Mathematical theories differ somewhat from traditional pattern matching systems by introducing the following additional computational concerns to the pattern matching problem

Many Patterns Mathematical theories may contain thousands of rewrite patterns. For example `RUBI`[35], a system for the solution of indefinite integrals, requires a collection of thousands of patterns; none of which overlap. A matching process that scales linearly with the number of patterns can be computationally prohibitive.

Associative Commutative Matching Mathematical theories often contain associative and commutative operations (like scalar addition or set union). While associativity and commutativity could be defined in standard systems with the following identities

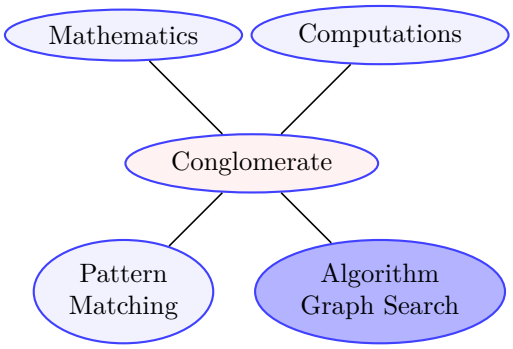
$$f(x, f(y, z)) = f(f(x, y), f(z))$$
$$f(x, y) = f(y, x)$$

Doing so may result in pathologically poor search patterns with a combinatorial number of options.

5.3 Algorithm Search

We reinforce that a Term Rewrite System consists of the following elements:

1. A language of terms
2. A collection of isolated transformations
3. A system to coordinate the application of those transformations.



In this section we discuss the third element, the coordination of transformations.

We iteratively evolve our input through repeated application of a collection of transformations. At each stage we select one among a set of several valid transformations. These repeated decisions form a decision tree.

We may arrive at the same state through multiple different decision paths. We consider a directed graph where nodes are states (terms) and edges are transitions between states (transformations). Macroscopic properties of this graph of possible states depend on properties of the set of transformations and terms.

Properties on Transformations A set of transformations is said to be *simply normalizing* if they are unable to return to visited states. In this case the graph of state transitions is a directed *acyclic* graph (DAG).

A set of terminating transformations is *confluent* if exhaustive application of the transformations can only lead to a single state, regardless of the order of application. I.e. the DAG has at most one node with out-degree zero.

Properties on States The information known on states affects the search problem. We consider the following properties:

- Validity: There may be a notion of validity at each final state. Only some leaves represent valid terminal points; others may be incompletely transformed dead-ends.
- Quality: There may be a notion of quality or cost both at each final state and at intermediate states. Such an objective function may be used as a local guide in a search process.

Algorithmic Challenges - Search Strategies Both multiple patterns and associative-commutative operators compound the possibility of multiple valid transformations at a single stage. If the collection of transformations are not confluent and if the set of possible states is large then a search strategy must be specified to find a good solution in a reasonable amount of time.

5.4 Background

Logic programming and term rewrite systems benefit from substantial theory and mature software packages.

Software Systems Several mature term rewrite systems exist. In spirit these descend from logic programming, first popularized by the Prolog language in the early seventies. Today systems like Maude and Elan from the OBJ family and the Stratego/XT toolset use term rewriting for the declarative definition of and optimization within languages. They serve as repositories for results from academic research.

OBJ: Maude, Elan The Maude system[36] uses term rewriting to support a meta-programming environment. The Maude language provides syntax to define syntax for other languages. It also enables the description of rewrite rules to transform terms in that language. Rewrite rules are separated into a set of confluent equations and a set of non-confluent rewrite rules. The non-confluent rules can be applied using two built-in strategies. Elan[37] differs from Maude in that it enables users to specify custom strategies.

Stratego/XT The Stratego/XT[38] toolset contains several separate tools to support the definition of syntax, language, transformations on terms within that language, and strategies to coordinate the application of those transformations. This orthogonal approach separates many of the ideas that are present in systems like Elan into distinct, clearly defined ideas.

Search Strategies Systems like Prolog hard-wire a single specific traversal into the transformation system. It includes backtracking to allow the search to walk out of dead ends and continuation to allow a user to lazily request additional results. Maude extends this system with the option of "fair rewrites" that samples from the applicable transformations with a round-robin policy.

While these strategies are useful in the common case it may be that a problem requires custom traversal for efficient computation. Systems like Elan enable developers to specify search strategies within their program. Elan includes terms like `repeat` to exhaustively evaluate a particular strategy or `dc one` to non-deterministically apply one of a set of strategies. Stratego/XT reinforces this idea by isolating it into it's own separate language Stratego. Stratego enables the description of complex traversals independent of any particular search problem.

Pattern Matching Challenges Pattern matching in some form is ubiquitous in modern programming languages. Unification of terms has long been a cornerstone of both logic programming languages and of theorem provers. Basic algorithms exist in standard texts on artificial intelligence[39].

However, as mentioned in Section 5.2, mathematical theories can be pathological both because they may require very many rewrite patterns and because they make heavy use of associative-commutative operators.

Many Patterns Because these patterns are used at every transformation step and because the collection changes infrequently it makes sense to store them in an indexed data structure that trades insertion time for query time. Discrimination nets are often used in practice [40]. These provide simultaneous matching of one input term to many stored rewrite patterns in logarithmic rather than linear time.

Associative-Commutative Matching Including the traditional definitions of associativity and commutativity in the rule set may lengthen compute times beyond a reasonable threshold.

Instead operators that follow one or both of the associative-commutative identities are often specifically handled within the implementation of the term rewrite system. For instance Maude requests that users mark such operators with special annotations.

In the simple case associativity may be handled by a round of flattening to n-ary trees (e.g. $f(x, f(y, z)) \to f(x, y, z)$) and commutativity by bipartite graph matching[41]. Because associative-commutative operators often occur in theories with many rewrite patterns, these two problems may be solved simultaneously for greatest efficiency. Discrimination nets can be extended (using multiple nets) to index many associative-commutative patterns[42, 43] efficiently, supporting many-to-one associative-commutative matching.

5.5 Matrix Rewriting in Maude

We implement a matrix language in Maude and use rewrite rules to declare mathematical transformations. Our goal is to demonstrate the simplicity with which mathematical theories can be constructed and the value of intuitive syntax. This discussion serves as proof of concept for a later implementation using SymPy and LogPy in Section 5.9.

Algebra The `matrix-algebra`[44] project defines a language for matrix expressions in Maude. First we define the sorts of terms:

```
sorts MatrixExpr MatrixSymbol Vector RowVector
subsort Vector RowVector MatrixSymbol < MatrixExpr
```

And a set of operators with associated precedences. A subset is included below:

```
op _+_   : MatrixExpr MatrixExpr -> MatrixExpr [ctor assoc comm prec 30] .
op __    : MatrixExpr MatrixExpr -> MatrixExpr [ctor assoc prec 25] .
op transpose : MatrixExpr            -> MatrixExpr [ctor] .
op inverse   : MatrixExpr            -> MatrixExpr [ctor] .
```

Note that operators are declared to be associative or commutative as keywords in the Maude system. These attributes are handled separately for the sake of efficiency. These operators define a language for expressions like the following expression for least squares linear regression. Note that a space connotes matrix multiplication.

```
inverse(transpose(X) X) transpose(X) y
```

We then provide a collection of equality transformations like the following:

```
eq inverse(inverse(A)) = A .
eq inverse(A) A = I .
eq A (B + C) = (A B) + (A C) [metadata "Distributive Law"] .
```

Inference This set of relations can be greatly increased with the ability to infer matrix properties on large expressions. In Maude we define a set of predicates:

```
sorts Predicate, AppliedPredicate, Context .
subsort AppliedPredicate < Context

ops symmetric orthogonal invertible positive-definite singular
    lower-triangular upper-triangular triangular unit-triangular
    diagaonal tridiagonal : -> Predicate .

op _is_ : MatrixExpr Predicate -> AppliedPredicate [prec 45].
op _,_  : Context    Context   -> Context [metadata "Conjoin two contexts"]
```

These lines provide the necessary infrastructure to declare a large set of matrix inference rules like the following example rules for symmetry:

```
var C     : Context .
vars X Y  : MatrixExpr .

ceq C => X Y is symmetric   = true if C => X is symmetric
                                   and C => Y is symmetric .
ceq C => transpose(X)     is symmetric = true if C => X is symmetric .
eq  C => transpose(X) X   is symmetric = true .
```

`eq` statements describe equality relations, `ceq` statements describe equality relations that are conditional on some expression, e.g. `C => X is symmetric`. `matrix-algebra` contains dozens of such statements.

Refinement The language and the inference can be combined automatically by a user to generate a rich set of simplification rules like the following:

```
ceq inverse(X) = transpose(X) if X is orthogonal
```

Statements of this form are clear to mathematical experts. More importantly the set of relations is sufficiently simple so that it can be extended by these same experts without teaching them the underlying system for their application to expression trees.

The meta-programming approach allows the specification of mathematical relations in a math-like syntax, drastically lowering the barrier of entry for potential mathematical developers. The term-rewrite infrastructure allows these relations to be applied automatically by mature and computationally efficient strategies.

Unfortunately the Maude system is an exotic dependency in the scientific community and interoperation with low-level computational codes was not a priority in its development. In Section 5.9 we will attain the ideals presented in this section by composing the Term, SymPy and LogPy packages.

5.6 Term

Term rewrite systems generally operate on a specific language of terms. In traditional logic programming languages like Prolog this term language is custom-built for and included within the logic programming system, enabling tight integration between terms and computational infrastructure. However a custom term language limits interoperation with other term-based systems (like computer algebra systems). Logic programming systems like miniKanren, written in Scheme, resolve this problem by describing terms with simple s-expressions, enabling broad interoperation with projects that use that same representation within its intended host language.

S-expressions are not idiomatic within the Python ecosystem and few projects define terms in this way. The intended object-oriented approach to this problem is to create an interface class and have client classes implement this interface if they want to interoperate with term manipulation codes. Unfortunately the Python ecosystem lacks a common interface for term representation in the standard sense. The lowest common shared denominator is the Python `object`.

Interface `Term` is a Python library to establish such an interface for terms across projects. It provides the following functions:

Function	Type	Description
`op`	`term -> operator`	The operator of a term
`args`	`term -> [term]`	The children or arguments of a term
`new`	`operator, [term] -> term`	Construct a new term
`isleaf`	`term -> bool`	Is this term a leaf?
`isvar`	`term -> bool`	Is this term a meta-variable?

These functions serve as a general interface. Client codes must somehow implement this interface for their objects. Utility codes can build functionality from these functions.

The `term` library also provides general utility functions for search and unification.

Composition In Python most systems that manipulate terms (like existing logic programming projects) create an interface which must be inherited by objects if they want to use the functionality of the system. This approach requires both foresight and coordination with the client projects. It is difficult to convince project organizers to modify their code to implement these interfaces, particularly if that code is pre-existing and well established.

Term was designed to interoperate with legacy systems where changing the client codebases to subclass from `term` classes is not an option. In particular, `term` was designed to simultaneously support two computer algebra systems, SymPy and Theano. Both of these projects are sufficiently entrenched to bar the possibility of changing the underlying data structures. This application constraint forced a design which makes minimal demands for interoperation; ease of composition is a core tenet.

To achieve interoperation we need to know how to do the following:

1. Implement the `new, op, args, isleaf` interface for a client object
2. Identify meta variables with `isvar`

`new, op, args, isleaf` To be useful in a client codebase we must specify how to interact with client types as terms. These can be added after code import time in two ways:

- Dispatch on global registries
- Dynamic manipulation of client classes

The functions `new, op, args, and isleaf` query appropriate global registries and search for the methods `_term_new`, `_term_op`, `_term_args`, `_term_isleaf` on their input objects. These method names are intended to be added onto client classes if they do not yet exist dynamically at runtime (so called "monkey patching"). This patching is possible after import time only due to Python's permissive and dynamic object model. This practice is dangerous in general only if other projects use the same names.

Because most Python objects can be completely defined by their type and attribute dictionary the following methods are usually sufficient for any Python object that does not use advanced features.

```
def _term_op(term):
    return type(term)

def _term_args(term):
    return term.__dict__

def _term_new(op, args):
    obj = object.__new__(op)
    obj.__dict__.update(args)
    return obj
```

These methods can then be attached after client code has been imported:

```
def termify(cls):
    cls._term_op    = _term_op
    cls._term_args  = _term_args
    cls._term_new   = classmethod(_term_new)

from client_code import ClientClass
termify(ClientClass)  # mutation
```

In this way any Python object may be regarded as a compound term. We provide a single `termify` function to mutate classes defined under the standard object model. Operationally we provide a more comprehensive function to handle common variations from the standard model (e.g. the use of `__slots__`)

Note that the Python objects themselves are usually traversed, not a translation (we do not call `_term_op, _term_args` exhaustively before execution.) Keeping the objects intact enables users to debug their code with familiar data structures and enables the use of client code *within* term traversals. This method will be useful later when we leverage SymPy's existing mathematical inference system within an external logic program.

Variable identification – `isvar` Meta variables denote sub-terms that can match any other term. Within existing Python logic programming projects meta variables are traditionally identified by their type. Python objects of the class `term.Var` are considered to be meta-variables. However, interaction with user defined classes may require the injection of a meta-variable as an attribute into an arbitrary Python object. It is possible that that object will perform checks that reject the inclusion of a `term.Var` object. For example, in user-written code for an Account object it is feasible that a balance attribute will be checked to be of type `float`. To match against a balance we need some way to make a `float` a meta-variable.

To resolve this issue we rely on a carefully managed set of global variables. We make unique and rare values (e.g. `-9999.9`) and place them in a globally accessible collection. Membership in this collection connotes meta-variable-ness. To avoid the normal confusion caused by global collections we manage this set with Python context managers/coroutines.

```
_meta_variables = set()

@contextmanager
def variables(*variables):
    old = _meta_variables.copy()             # Save old set
    _meta_variables.update(set(variables))   # Inject new variables

    yield                                    # Yield control to with block

    _meta_variables.clear()                  # Delete current set
    _meta_variables.update(old)              # Load old set
```

In the example below we find the name of the account-holder with 100 dollars. The generic Python string `"NAME"` is used as a meta-variable. The `variables` context manager places

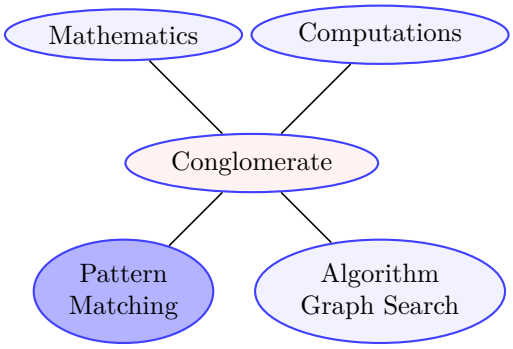
"NAME" into a global collection and then yields control to the code within the subsequent block. Code within that block is executed and queries this collection. Membership in the collection is equivalent to being a meta-variable. After the completion of the `with variables` block the global collection is reset to its original value, commonly the empty set. This approach allows the use of arbitrarily typed values as meta-variables, further enabling interoperation.

```
>>> from term import termify, variables, unify
>>> from bank import Account
>>> termify(Account)

>>> acct  = Account(name="Alice", balance=100)
>>> query = Account(name="NAME",  balance=100)
>>> vars  = ["NAME"]

>>> with variables(*vars):
...     print unify(acct, query, {})
{"NAME": "Alice"}

>>> print unify(acct, query, {})  # Does not unify outside the with block
False
```

Term is able to seamlessly interoperate with a generic client code with little setup.

5.7 LogPy

LogPy uses the `term` interface to build a general purpose logic programming library for Python. It implements a variant of `miniKanren`[45], a language originally implemented in a subset of Scheme. Comprehensive documentation for LogPy is available online[46].



The construction of LogPy was motivated by duplicated efforts in SymPy and Theano, two computer algebra systems available in Python. Both SymPy and Theano include special purpose modules to define and apply optimisations to their built-in mathematical and computational data structures. LogPy aims to replace these modules. The desire to deliver functionality to two inflexible codebases forced the creation of the `term` system described in Section 5.6. LogPy provides functionality on top of the `term` interface.

Basic Design - Goals LogPy programs are built up of *goals*. Goals produce and manage streams of substitutions.

```
goal :: substitution -> [substitution]
```

Example

```
>>> x = var('x')
>>> a_goal = eq(x, 1)
```

This goal uses `eq`, a goal constructor, to require that the logic variable `x` unifies to `1`. As previously mentioned goals are functions from single substitutions to sequences of substitutions. In the case of `eq` this stream has either one or zero elements:

```
>>> a_goal({})      # require no other constraints
({~x: 1},)
```

Basic Design - Goal Combinators LogPy provides logical goal combinators to manage the union and intersection of streams.

Example We specify that $x \in \{1, 2, 3\}$ and that $x \in \{2, 3, 4\}$ with the goals `g1` and `g2` respectively.

```
>>> g1 = membero(x, (1, 2, 3))
>>> g2 = membero(x, (2, 3, 4))

>>> for s in g1({}):
...     print s
{~x: 1}
{~x: 2}
{~x: 3}
```

To find all substitutions that satisfy *both* goals we can feed each element of one stream into the other.

```
>>> for s in g1({}):
...     for ss in g2(s):
...         print ss
{~x: 2}
{~x: 3}
```

Logic programs can have many goals in complex hierarchies. Writing explicit for loops quickly becomes tedious. Instead, LogPy provides functions to combine goals logically.

```
combinator :: [goals] -> goal
```

Two important logical goal combinators are logical all `lall` and logical any `lany`.

```
>>> for s in lall(g1, g2)({}):
...     print s
{~x: 2}
{~x: 3}

>>> for s in lany(g1, g2)({}):
...     print s
{~x: 1}
{~x: 2}
{~x: 3}
{~x: 4}
```

User Syntax These combinators and goals are accessed with the `run` function as in miniKanren. `run` has three arguments, the number of desired results, the desired target variable, and a set of goals to satisfy. The number of desired results can also take on the value `0` for "all results" or `None` for a lazy iterator. Examples are shown below:

```
>>> run(0, x, lall(membero(x, (1, 2, 3)),
...                membero(x, (2, 3, 4)))
(2, 3)

>>> run(0, x, lany(membero(x, (1, 2, 3)),
...                membero(x, (2, 3, 4)))
(1, 2, 3, 4)
```

LogPy is a collection of such goals and combinators. These will be useful for the matching of mathematical patterns in SymPy and their translation into DAGs in `computations`.

5.8 Mathematical Rewriting - LogPy and SymPy

We implement a rudimentary mathematical pattern matcher by composing LogPy, a general purpose logic programming library, and SymPy, a computer algebra system. We chose this approach instead of one of the mature systems mentioned in Section 5.4 in order to limit the number of dependencies that are uncommon within the scientific computing ecosystem and in order to leverage and expose existing mathematical expertise already within SymPy.

LogPy Manipulates SymPy Terms Recall that LogPy supports the `term` interface discussed in Section 5.6. We now impose the `term` interface on SymPy classes so that LogPy can manipulate SymPy terms. This process happens outside of the SymPy codebase. We do this with the following definitions of the `_term_xxx` methods:

```
from sympy import Basic
Basic._term_op      = lambda self: self.func
Basic._term_args    = lambda self: self.args
Basic._term_new     = classmethod(lambda op, args: op(*args))
Basic._term_isleaf  = lambda self: len(self.args) == 0
```

We do not invent a new term language for this term rewrite system. Rather, we reuse the existing language from the SymPy computer algebra system; mathematics is not reinvented within the logic programming system.

Storing Mathematical Patterns We use our old example unpacking logarithms of exponents. A rewrite rule can be specified by a source, target, and condition terms. These are specified with SymPy terms. For example the following transformation can be specified with the following tuple:

$$\log(\exp(x)) \to x \quad \forall x \in \mathbb{R}$$

```
( log(exp(x)),       x,      Q.real(x) )
```

For a particular theory we may store a large collection of these

```
patterns = [
    (Abs(x),            x,      Q.positive(x)),
    (exp(log(x)),       x,      Q.positive(x)),
    (log(exp(x)),       x,      Q.real(x)),
    (log(x**y),     y*log(x),   True),
    ...
          ]

vars = {x, y}
```

These are later indexed in a LogPy relation.

```
from logpy import TermIndexedRelation as Relation
from logpy import facts
rewrites = Relation('rewrites')
facts(rewrites, *patterns)
```

Note that the definition of the mathematical patterns depends only on SymPy. The injection into a LogPy relation is well isolated. In the future more mature implementations can replace the LogPy interaction easily without necessitating changes in the mathematical code. Removing such connections enables components to survive obsolescence of neighboring components. The `patterns` collection does not depend on the continued use of `LogPy`. By removing unnecessary connections between modules we avoid "weakest link in the chain" survivability.

LogPy Execution We transform SymPy's `ask` function for mathematical inference into a LogPy goal with `goalify` to form `asko`, a goal constructor. `asko` simply evaluates `ask` on the first parameter and filters a stream so that results match the second parameter.

```
from logpy import goalify
from sympy import ask
asko = goalify(ask)
```

We construct a function to perform a single term rewrite step. It creates and executes a brief LogPy program, discussed immediately afterwards.

```
from logpy import run, lall, variables

def rewrite_step(expr, rewrites):
    """ Possible rewrites of expr given relation of patterns """
    target, condition = var(), var()
    with variables(*vars):
        return run(None, target, rewrites(expr, target, condition),
                                 asko(condition, True))
```

The `run` function asks for a lazily evaluated iterator that returns reified values of the variable `target` that satisfy both of the following goals:

`rewrites(expr, target, condition)` The LogPy Relation `rewrites` stores facts, in this case our rewrite patterns. The facts are of the form `(source, target, condition)` and claim that an expression matching `source` can be rewritten as the `target` expression if the boolean expression `condition` holds true. For example rewrites might contain the following facts

```
(Abs(x),            x,      Q.positive(x)),
(exp(log(x)),       x,      Q.positive(x)),
(log(exp(x)),       x,      Q.real(x)),
(log(x**y),     y*log(x),   True),
```

By placing the input, `expr`, in the source position we mandate that `expr` must unify with the `source` of the pattern. The `rewrites` relation selects the set of potentially matching patterns and produces a stream of matching substitutions. The `target` and `condition` terms will be reified with these matchings during future computations.

For example if `expr` is the term `Abs(y**2)` then only the first pattern matches because the operations `exp` and `log` can not unify to `Abs`. The logic variables `target` and `condition` reify to `y**2` and `Q.positive(y**2)` respectively. In this case only one pattern in our collection yields a valid transformation.

`asko(condition, True)` The `asko` goal further constrains results to those for which the `condition` of the pattern evaluates to `True` under SymPy's `ask` routine. This process engages SymPy's logic system and the underlying SAT solver. Through interoperation we gain access to and interact with a large body of pre-existing logic code.

If as above `expr` is `Abs(y**2)` and `x` matches to `y**2` then we ask SymPy if the boolean expression `Q.positive(y**2)` is true. This might hold if, for example, we knew that the SymPy variable `y` was real and non-zero. If this is so then we yield the value of `target`, in this case `y**2`; otherwise this function returns an empty iterator.

Finally we return a lazy iterator of all target patterns such that the source pattern matches the input expression and that the condition of the pattern is satisfied.

Analysis Interactions between mathematical, logical, and algorithmic pieces of our solution are limited to a few lines of code. Simultaneous expertise is only rarely necessary.

Teaching LogPy to interact with SymPy is a simple exercise; the need for simultaneous expertise in both projects is brief. Using LogPy to construct a term rewrite system is similarly brief, only a few lines in the function `rewrite_step`. By supporting interoperation with preexisting data structures we were able to leverage the preexisting mathematical logic system in SymPy without significant hassle. The implementation of the `rewrites` Relation determines matching performance. Algorithmic code is a separate concern and not visible to the mathematical users.

5.9 Matrix Rewriting in SymPy

By composing SymPy.matrices.expressions with LogPy we obtain much of the same intuitive functionality presented in the `matrix-algebra` project in Maude discussed in Section 5.5.

We describe high-level mathematical transformations while restricting ourselves to the SymPy language. Unfortunately because our solution is embedded in Python we can not achieve the same convenient syntax support provided by Maude (e.g. the `_is_` operator.) Instead we encode a set of transformations in `(source, target, condition)` tuples of SymPy terms.

We suffer the following degradation in readability in order to remove Maude, an exotic dependency. We describe the content of the transformation without specialized syntax.

```
Wanted:      inverse(X) = transpose(X) if X is orthogonal
Delivered:  (inverse(X) , transpose(X) ,  Q.orthogonal(X))
```

We can then separately connect an external term rewrite system to transform these tuples into rewrite rules and use them to simplify matrix expressions. In this work we use LogPy but in principle any term rewrite system should be sufficient. As with the system in Maude we believe that extending the set of simplification relations is straightforward and approachable to a very broad community. Additionally, this declarative nature allows us to swap out the term rewrite system backend should future development produce more mature solutions.

Example – Determinants We present mathematical information about determinants taken from the Matrix Cookbook [33] and encoded in the manner described above.

```
# Original,      Result,          Condition
(det(A),         0,               Q.singular(A)),
```

```
(det(A),        1,              Q.orthogonal(A)),
(Abs(det(A)),   1,              Q.unitary(A)),
(det(A*B),      det(A)*det(B),  Q.square(A)),
(det(BlockMatrix([[A,B],[C,D]])), det(A)*det(D - C*A.I*B), Q.invertible(A))
...
```

5.10 Greedy Search with Backtracking

Pattern matching enables the declarative expression of a large number of transformations. To be used effectively these transformations must be coordinated intelligently. In this section we complement the previous discussion on pattern matching with a discussion of coordination strategies.

5.10.1 Problem Description

The projects within this dissertation match and apply one of a set of possible transformations to a term. This process is often repeated until no further transformations apply. This process is not deterministic; each step may engage multiple valid transformations. These in turn may yield multiple different transformation paths and multiple terminal results. These steps and options define a graph with a single root node. The root is the input expression and the nodes with zero out-degree (leaves) are terminal states on which no further transformation can be performed.

This section discusses this search problem abstractly. Section 6.1 discusses the search problem concretely in the context of BLAS/LAPACK computations.

We consider a sequence of decreasingly trivial traversal algorithms. These expose important considerations. We build up to our operational algorithm, greedy depth first search with backtracking.

Properties on Transformations The sets of transformations described within this dissertation have the following two important properties:

- They are *strongly normalizing*: The graph has no cycles. There is always a sense of constant progression to a final result. As a result the object of study is a directed acyclic graph (DAG).
- They are not *confluent* in general: There are potentially multiple valid outcomes; the DAG may have multiple leaves. The choice of final outcome depends on which path the system takes at intermediate stages.

Due to these two properties we can consider the set of all possible intermediate and final states as a directed acyclic graph (DAG) with a single input. Operationally this DAG can grow to be prohibitively large. In this section we discuss ways to traverse this DAG to find high quality nodes/computations quickly.

For simplicity this section will consider the simpler problem of searching a tree (without duplicates.) The full DAG search problem can be recovered through use of dynamic programming.

Properties on States Additionally, the states within this graph have two important properties:

- Quality: There is a notion of quality or cost both at each final state and at all intermediate states. This cost is provided by an objective function and can be used to guide our search.
- Validity: There is a notion of validity at each final state. Only some leaves represent valid terminal points; others are dead-ends.

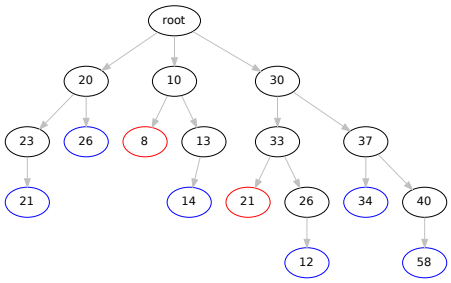


Figure 7: An example tree of possible computations. A score annotates each node.

Example Tree We reinforce the problem description above with the example in Figure 7.

This tree has a root at the top. Each of its children represent incremental improvements on that computation. Each node is labeled with a cost; costs of children correlate with the costs of their parents. The leaves of the tree are marked as either valid (blue) or invalid (red). Our goal is to quickly find a valid (blue) leaf with low cost without searching the entire tree.

Interface When exploring a tree to minimize an objective function, we depend on the following interface:

```
children  ::  node -> [node]
objective ::  node -> score
isvalid   ::  node -> bool
```

In Section 6.1 we provide implementations of these functions for the particular problem of matrix algorithm search. In this section we describe abstract search algorithms using this interface.

5.10.2 A Sequence of Algorithms

Leftmost A blind search may find sub-optimal solutions. For example consider the strategy that takes the left-most node at each step as in Figure 8. This process arrives at a node cost 21. In this particular case that node is scored relatively poorly. The search process was cheap but the result was poor.

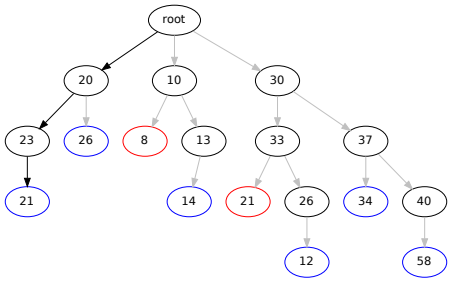


Figure 8: A naive strategy to traverse down the left branch yields a sub-optimal result.

```
def leftmost(children, objective, isvalid, node):
    if isvalid(node):
        return node
    kids = children(node):
    if kids:
        return leftmost(kids[0])
    else:
        raise Exception("Unable to find valid leaf")
```

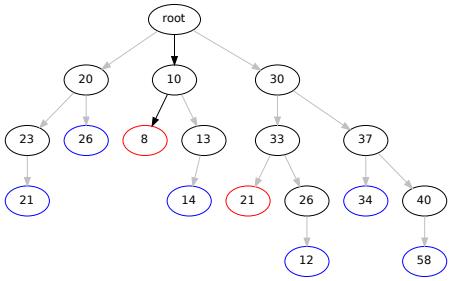


Figure 9: A greedy strategy selects the branch whose root has the best score.

Greedy Search If we can assume that the cost of intermediate nodes is indicative of the cost of their children then we can implement a greedy solution that always considers the subtree of the minimum cost child. See Figure 9.

```
def greedy(children, objective, isvalid, node):
    if isvalid(node):
        return node
    kids = children(node):
    if kids:
        best_subtree = min(kids, key=objective)
```

```
        return greedy(best_subtree)
    else:
        raise Exception("Unable to find valid leaf")
```

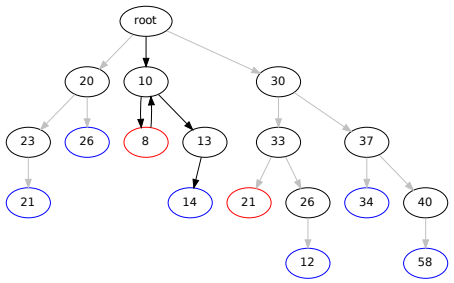


Figure 10: Backtracking allows us to avoid terminating in dead ends.

Greedy Search with Backtracking Greedy solutions like the one above can become trapped in a dead-end. Our example arrives at an invalid leaf with cost `8`. There is no further option to pursue in this case. The correct path to take at this stage is to regress backwards up the tree as in Figure 10 and consider other previously discarded options.

This process requires the storage and management of history of the traversal. By propagating streams of ordered solutions rather than a single optimum we implement a simple backtracking scheme.

```
from itertools import imap, chain

def greedy(children, objective, isvalid, node):
    """ Greedy guided depth first search.  Returns iterator """
    if isvalid(node):
        return iter([node])

    kids = sorted(children(node), key=objective)
    streams = (greedy(children, objective isvalid, kid) for kid in kids)

    return chain.from_iterator(streams)
```

We evaluate and multiplex streams of possibilities lazily, computing results as they are requested. Management of history, old state, and garbage collection is performed by the Python runtime and is localized to the generator mechanisms and `chain` functions found in the standard library. Similar elements are found within most functional or modern programming languages.

Continuation The greedy search with backtracking approach has the added benefit that a lazily evaluated stream of all leaves is returned. If the first result is not adequate then one

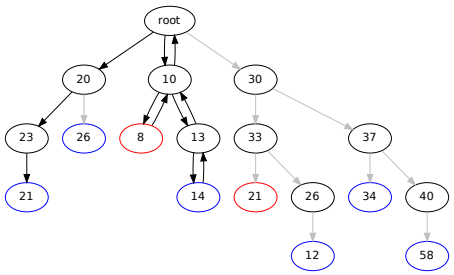


Figure 11: Continuation allows us to continue to search the tree even after a valid result has been found.

can ask the system to find subsequent solutions. These subsequent computations pick up where the previous search process ended, limiting redundant search. See Figure 11.

By exhaustively computing the iterator above we may also traverse the entire tree and can minimize over all valid leaves. This computation may be prohibitively expensive in some cases but remains possible when the size of the tree is small.

Repeated Nodes - Dynamic Programming If equivalent nodes are found in multiple locations then we can increase search efficiency by considering a DAG rather than tree search problem. This method is equivalent to dynamic programming and can be achieved by memoizing the intermediate shared results. The tree search functions presented above can be transformed into their DAG equivalents with a `memoize` function decorator.

5.10.3 Extensions

In this section we discuss a few generalizations of greedy search.

K-deep greedy Rather than compute and then minimize over the children of a node we could compute and minimize over the grandchildren. More generally we can compute and minimize over the descendants of depth `k`. This method increases foresight and computational cost.

Breadth first The greedy search above is *depth first*. It exhausts its current subtree before moving on to siblings. Alternatively, it could search subtrees more fairly, returning a single greedily optimal solution within each before moving on to the next. This cycles between early alternatives rather than late alternatives. Code for this algorithm can be obtained by replacing `chain` with `interleave` in the code above.

Expanding Frontier Both depth and breadth first are special cases of an expanding frontier, navigating the graph (evaluating `children`) at nodes adjacent to those just visited.

This restriction of adjacency is not essential. Instead we can maintain a set of accessible nodes and select a global optimum to evaluate.

5.11 Managing Rule Sets

The properties of the sets of rules impact the performance of compilation. The possibility of pathological properties like cycles or non-terminating sequences call into question the feasibility of these methods. A poor rule set combined with a naive rule coordination system has little value. Even when pathological graph motifs are absent the large numbers of redundant rules that occur in mathematical theories (e.g. integration) can result in very poor search times. These problems can be mitigated either directly by the domain practitioner or through automated systems.

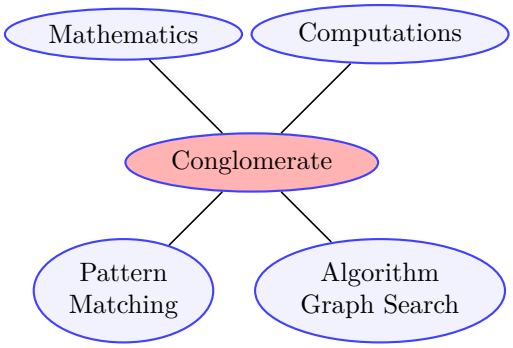
In many cases domain knowledge to mitigate this issue may be readily available to the rule specifier. Many mathematical theories outline a clear direction of simplicity. For as long as transformations proceed monotonically under such an objective function, problems like cycles may be easily avoided. Pre-existing rule systems like RUBI [35] for indefinite integration and Fu et. al's work[47] on trigonometric simplification both take care to outline such directions explicitly. In the case of Fu, they even go so far as to separate the rule set into separate stages which should be applied sequentially. The specification of control by domain experts is orthogonal to the design presented here but may have significant value for performance.

These problems may also be approached abstractly through automated methods. The identification of cycles is often possible by looking only at the structure of the terms without semantic understanding of the domain. This approach is orthogonal to the domain-specific approach and can supply valuable checks on domain practitioners solutions. These methods become increasingly valuable as these methods are used by a wider population of non-experts. The Maude Sufficient Completeness Checker[48] is such an automated analysis system.

6 Automated Matrix Computations

In this chapter we compose software components from the previous sections to create a compiler for the generation of linear algebra codes from matrix expressions.

- Chapter 3: Computer Algebra
- Chapter 4: Computations
- Chapter 5: Term Rewriting



We then compose these components to create a larger system to produce numeric codes for common matrix expressions called. These examples demonstrate both the ability and the extensibility of our system. We refer to this composition as the `conglomerate`

project. Specifically we will construct two computations that are common in scientific computing, least squares linear regression and the Kalman Filter. In each case we will highlight the added value of modular design.

6.1 Covering Matrix Expressions with Computations

We search for high quality computations to compute a set of matrix expressions. This task will require functionality from the following sections

- Matrix Language 3.5: extends SymPy to handle symbolic linear algebra
- Computations 4: describes BLAS/LAPACK at a high level and provides Fortran90 code generation
- Pattern Matching 5.2 and LogPy 5.7: provides functionality to match a current state to a set of valid next states
- Graph Search 5.10: traverses a potentially large tree of decisions to arrive at a "good" final state.

These projects are disjoint. In this section we describe the information necessary to compose them to solve our problem in automated generation of mathematically informed linear algebra routines.

A Graph of Computations Given a set of expressions-to-be-computed we consider a tree where:

- Each node is a computation whose outputs include those expressions
- An edge exists between two nodes if we know a transformation to produce one computation from the other

At the top of this tree is the trivial identity computation which computes the desired outputs given those same outputs as inputs. At the bottom of this tree are computations whose inputs are not decomposable by any of our patterns. In particular, some of these leaf computations have inputs that are all atoms; we call these leaves valid.

In principle this tree can be very large, negating the possibility of exhaustive search in the general case. Additionally some branches of this tree may contain dead-ends requiring back-tracking; we may not be able to find a valid all-inputs-are-atoms leaf within a subtree. We desire an algorithm to find a valid and high-quality leaf of this tree efficiently.

This problem matches the abstract version in Section 5.10 on algorithmic search. In that section we discussed the declarative definition and application of rewrite rules and algorithms to search a decision tree given the following interface:

```
children  ::  node -> [node]
objective ::  node -> score
isvalid   ::  node -> bool
```

In this section we implement a concrete version. We provide a set of transformation patterns and implementations of the search interface functions. We describe transformations declaratively in `SymPy` and `computations`, and then use these transformations and the `LogPy` project to define the `children` function. We implement and discusses a simple and effective objective functions on intermediate computations and build a simple validity function. We reproduce a function for greedy search with backtracking first encountered in Section 5.3 and finally produce our final code. We reinforce that this is the entirety of the solution for the particular problem of automated search of dense linear algebra algorithms. All other intelligence is distributed to the appropriate application agnostic package.

Compute Patterns Computations are used to break expressions into smaller pieces much in the way an enzyme breaks a protein into constituents. For example $\alpha AB + \beta C$ can be broken into the components α, A, B, β, C using the various Matrix Multiply routines (`GEMM`, `SYMM`, `TRMM`). To determine this automatically we create a set of patterns that match expressions to computations valid in that case. We encode this information in `(source expression, computation, condition)` patterns.

```
patterns = [
    (alpha*A*B + beta*C ,  GEMM(alpha, A, B, beta, C) ,
            True),
    (alpha*A*B + beta*C ,  SYMM(alpha, A, B, beta, C) ,
            Q.symmetric(A) | Q.symmetric(B)),
    (alpha*A*B + beta*C ,  TRMM(alpha, A, B, beta, C) ,
            Q.triangular(A) | Q.triangular(B)),
    ...]
```

These patterns can be encoded by computational experts and can be used by pattern matching systems such as LogPy.

```
from logpy import facts, Relation
computes = Relation('computes')
facts(computes, *patterns)
```

Children of a Computation Given a computation we compute a set of possible extensions with simpler inputs. We search the list of patterns for all computations which can break down one of the non-trivial inputs. Any of the resulting computations may be added into the current one.

Our solution with LogPy and `computations` depends on `rewrite_step` from 5.8 and looks like the following:

```
computations_for = partial(rewrite_step, rewrites=computes)
def children(comp):
    """ Compute next options in tree of possible algorithms """
    atomics = sum(map(computations_for, comp.inputs), ())
    return map(comp.__add__, atomics)
```

Validity When we build a computation we ask for the desired inputs of that computation. Our frontend interface will look like the following example:

```
#      compile(inputs,       outputs       ,   assumptions )
comp = compile([X, y],  [(X.T*X).I * X.T*y],  Q.fullrank(X))
```

We desire computations whose inputs are restricted to those requested.

```
def isvalid(comp):
    return set(comp.inputs).issubset(inputs)
```

Objective Function To guide our search we use an objective function to rank the overall quality of a computation. In general this function might include runtime, energy cost, or an easily accessible proxy like FLOPs.

Operationally we compute a much simpler objective function. We order atomic computations so that specialized operations like `SYMM` are preferred over mathematically equivalent but computationally general operations like `GEMM`. Less efficient operations like `AXPY` are deemphasized by placing them at the end. This function often produces results that match decisions made by individual experts when writing code by hand.

```
order = [FFTW, POSV, GESV, LASWP, SYRK, SYMM, GEMM, AXPY]
def objective(comp):
    """ Cost of a computation `comp` - lower is better """
    if isinstance(comp, CompositeComputation):
        return sum(map(objective, comp.computations))
    else:
        return order.index(type(comp))
```

The list `order` is trivially accessible by numeric experts. This solution is intuitive to extend and works surprisingly well in practice.

Search We re-present the tree search problem first defined in Section 5.3. Fortunately this problem is easily separable. For cohesion we restate our greedy solution below:

```
from itertools import imap, chain

def greedy(children, objective, isvalid, node):
    """ Greedy guided depth first search.  Returns iterator """
    if isvalid(node):
        return iter([node])

    kids = sorted(children(node), key=objective)
    streams = (greedy(children, objective isvalid, kid) for kid in kids)

    return chain.from_iterator(streams)
```

Note that this solution is ignorant of the application of matrix computations.

Compile We coordinate these functions in the following master function:

```
def compile(inputs, outputs, *assumptions):
    """ Compile math expressions to computation """
    c = Identity(*outputs)

    def isvalid(comp):
        return set(comp.inputs).issubset(inputs)

    with assuming(*assumptions):      # SymPy assumptions available
        stream = greedy(children, objective, isvalid, c)
        result = next(stream)

    return result
```

Analysis We chose to provide explicit code in this section both for completeness and to demonstrate the simplicity of this problem once the appropriate machinery is in place. We showed that once the generally applicable components exist the particular problem of automated matrix algorithm search can be reduced to around 40 lines of general purpose code (including comments and whitespace). The `conglomerate` project contains very little logic outside of what is present in the application agnostic and reusable packages (like LogPy). The information that is present is largely expert knowledge for this application (like the objective function or patterns.)

Finished Result

```
patterns = [
    (alpha*A*B + beta*C ,  GEMM(alpha, A, B, beta, C) ,
            True),
    (alpha*A*B + beta*C ,  SYMM(alpha, A, B, beta, C) ,
            Q.symmetric(A) | Q.symmetric(B)),
    (alpha*A*B + beta*C ,  TRMM(alpha, A, B, beta, C) ,
            Q.triangular(A) | Q.triangular(B)),
    ...]

from logpy import facts, Relation
computes = Relation('computes')
facts(computes, *patterns)


computations_for = partial(rewrite_step, rewrites=computes)
def children(comp):
    """ Compute next options in tree of possible algorithms """
    atomics = sum(map(computations_for, comp.inputs), ())
    return map(comp.__add__, atomics)
```

```
order = [FFTW, POSV, GESV, LASWP, SYRK, SYMM, GEMM, AXPY]
def objective(comp):
    """ Cost of a computation `comp` - lower is better """
    if isinstance(comp, CompositeComputation):
        return sum(map(objective, comp.computations))
    else:
        return order.index(type(comp))


def compile(inputs, outputs, *assumptions):
    """ Compile math expressions to computation """
    c = Identity(*outputs)

    def isvalid(comp):
        return set(comp.inputs).issubset(inputs)

    with assuming(*assumptions):      # SymPy assumptions available
        stream = greedy(children, objective, isvalid, c)
        result = next(stream)

    return result
```

Termination In Section 5.11 we discussed that the termination and performance properties of compilation may strongly depend on the set of transformations. In this case we are guaranteed termination because each of our transformations breaks down the inputs of the computation by a finite amount. Each rule application reduces the complexity of the inputs by an integer value, bounding the number of steps by the complexity of the inputs.

6.2 Relation to Other Work

Fabregat-Traver and Bientinesi approached this same problem with similar methods[27, 28]. They too transform matrix terms into BLAS/LAPACK call graphs by searching through a graph of possible transformations. They take this work further by providing matrix-specific heuristics to guide and to constrain the search space, and also by providing support for iterative algorithms.

While we were aware of their preliminary work early in the development of this project, we developed our work independently. Their work has since delved more deeply into efficient search for efficient matrix algorithms while ours has broadened out into the use of rewrite rules within other subfields of computer algebra as well as alternative control abstractions to support novice users.

6.3 Linear Regression

We automatically generate code to compute least squares linear regression, a common application first encountered in Section 3.5.

$$X\beta \cong y$$

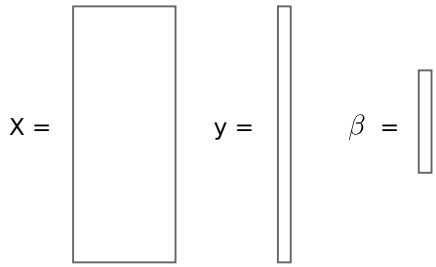


Figure 12: Array shapes for linear regression

The solution to this problem can be posed as the following matrix expression which will serve as the input to our compilation chain.

$$\beta = (X^T X)^{-1} X^T y$$

Naive Implementation Writing code to compute this expression given variables `X` and `y` can be challenging in a low-level language. Algorithms for multiplication and solution of matrices are not commonly known, even by practicing statisticians. Fortunately high-level languages like Matlab and Python/NumPy provide idiomatic solutions to these problems.

$$\beta = (X^T X)^{-1} X^T y$$

Python/NumPy	`beta = (X.T*X).I * X.T*y`
MatLab	`beta = inv(X'*X) * X'*y`

The code matches mathematical syntax almost exactly, enabling mathematical programmers.

Refined Implementations Unfortunately this implementation is also inefficient. A numerical expert would note that this code first computes an explicit inverse and then performs a matrix multiply rather than performing a direct matrix solve, an operation for which substantially cheaper and numerically robust methods exist. A slight change yields the following, improved implementation:

Python/NumPy	`beta = solve(X.T*X, X.T*y)`
MatLab	`beta = X'*X \ X'*y`

These implementations can again be refined. In the case when `X` is full rank (this is often

the case in linear regression) then the left hand side of the solve operation, $X^T X$, is both symmetric and positive definite. The symmetric positive definite case supports a more efficient solve routine based on the Cholesky decomposition.

The Matlab backslash operator performs dynamic checks for this property, while the Python/NumPy `solve` routine will not. The Matlab solution however still suffers from operation ordering issues as the backsolve will target the matrix `X'` rather than the vector `(X'*y)`.

And so a further refined solution looks like the following, using a specialized solve from the `scipy` Python library and explicitly parenthesizing operations in MatLab.

Python/NumPy	`beta = scipy.solve(X.T*X, X.T*y, sym_pos=True)`
MatLab	`beta = (X'*X) \ (X'*y)`

Connecting Math and Computation Languages like Matlab, Python, and R have demonstrated the utility of linking a "high productivity" syntax to low-level "high performance" routines like those within BLAS/LAPACK. While the process of designing efficient programs is notably simpler, it remains imperfect. Naive users are often incapable even of the simple optimizations at the high level language (e.g. using solve rather than computing explicit inverses); these optimizations require significant computational experience. Additionally, even moderately expert users are incapable of leveraging the full power of BLAS/LAPACK. This may be because they are unfamiliar with the low-level interface, or because their high-level language does not provide clean hooks to the full lower-level library.

Ideally we want to be given a naive input like the following expression and predicates:

```
(X.T*X).I * X.T*y
full_rank(X)
```

We produce the following sophisticated computation:

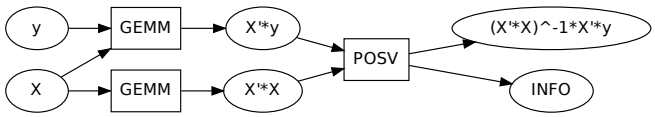


Figure 13: A computation graph for least squares linear regression

We perform this through a progression of small mathematically informed transformations.

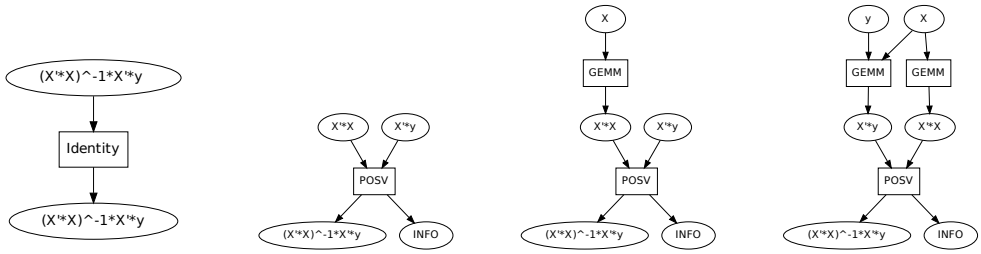


Figure 14: A progression of computations to evolve to the computation in Figure 13

We engage the pattern matching and search system described in Section 5 to transform the mathematical expression into the computational directed acyclic graph.

User Experience This search process and the final code emission is handled automatically. A scientific user has the following experience:

```
X = MatrixSymbol('X', n, m)
y = MatrixSymbol('y', n, 1)

inputs  = [X, y]
outputs = [(X.T*X).I*X.T*y]
facts   = Q.fullrank(X)
types   = Q.real_elements(X), Q.real_elements(y)

f = build(inputs, outputs, facts, *types)
```

This generates the following Fortran code.

```
subroutine f(X, y, var_7, m, n)
implicit none

integer, intent(in) :: m
integer, intent(in) :: n
real*8, intent(in) :: y(n)          !  y
real*8, intent(in) :: X(n, m)       !  X
real*8, intent(out) :: var_7(m)     !  0 -> X'*y -> (X'*X)^-1*X'*y
real*8 :: var_8(m, m)               !  0 -> X'*X
integer :: INFO                     !  INFO

call dgemm('N', 'N', m, 1, n, 1.0, X, n, y, n, 0.0, var_7, m)
call dgemm('N', 'N', m, m, n, 1.0, X, n, X, n, 0.0, var_8, m)
call dposv('U', m, 1, var_8, m, var_7, m, INFO)

RETURN
END
```

This code can be run in a separate context without the Python runtime environment. Alternatively for interactive convenience it can be linked in with Python's foreign function interface to a callable python function object that consumes the popular NumPy array data structure. This wrapping functionality is provided by the pre-existing and widely supported package `f2py`.

Numerical Result We provide timings for various implementations of least squares linear regression under a particular size. As we increase the sophistication of the method we decrease the runtime substantially.

```
>>> n, k = 1000, 500

>>> X = np.matrix(np.random.rand(n, k))
>>> y = np.matrix(np.random.rand(n, 1))

>>> timeit (X.T*X).I * X.T*y
```

```
10 loops, best of 3: 76.1 ms per loop

>>> timeit numpy.linalg.solve(X.T*X, X.T*y)
10 loops, best of 3: 55.4 ms per loop

>>> timeit scipy.linalg.solve(X.T*X, X.T*y, sym_pos=True)
10 loops, best of 3: 33.2 ms per loop
```

We now take the most naive user input from SymPy

```
>>> X = MatrixSymbol('X', n, k)
>>> y = MatrixSymbol('y', n, 1)
>>> beta = (X.T*X).I * X.T*y
```

And have our compiler build the computation:

```
>>> with assuming(Q.real_elements(X), Q.real_elements(y)):
...     comp = compile([X, y], [beta])
...     f = build(comp, [X, y], [beta])
```

Our computation originates from the naive user input $(X^T X)^{-1} X^T y$ but competes with the most sophisticated version that the `scipy` stack provides.

```
>>> timeit f(nX, ny)
10 loops, best of 3: 30.9 ms per loop
```

Disclaimer: These times are dependent on matrix size, architecture, and BLAS/LAPACK implementation. Results may vary. The relevant point is the comparable performance rather than the explicit numbers.

Development Result Our solution produces the numerically optimal result. It was generated by the most naive expression. We deliver high quality results to the majority of naive users.

This result is not isolated to the particular application of linear regression. SymPy supports the expression of a wide range of matrix computations ranging from simple multiplies to complex factorizations and solves.

Finally we mention that further room for improvement exists. Least squares problems can be solved with a single specialized LAPACK routine. This routine depends on the QR factorization for greater numerical stability.

6.4 SYRK - Extending Computations

The computation for linear regression can be further improved. In particular the computation `X -> GEMM -> X'*X`, while correct, actually fits a special pattern; it is a symmetric rank-k update and can be replaced by `X -> SYRK -> X'*X`.

This fact was discovered by a scientific programmer with extensive familiarity with BLAS/LAPACK. He was able to correct this inefficiency by adding an additional computation:

```
class SYRK(BLAS):
    """ Symmetric Rank-K Update 'alpha X' X + beta Y' """
    _inputs  = (alpha, A, beta, D)
    _outputs = (alpha * A * A.T + beta * D,)
    inplace  = {0: 3}
    fortran_template =  ...
```

And by adding the relevant patterns

```
  (alpha*A*A.T + beta*D, SYRK(alpha, A, beta, D),   True),
  (A*A.T,                SYRK(1.0, A, 0.0, 0),      True),
```

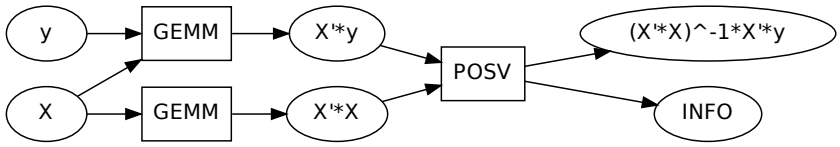


Figure 15: Least squares linear regression with 'GEMM' computation

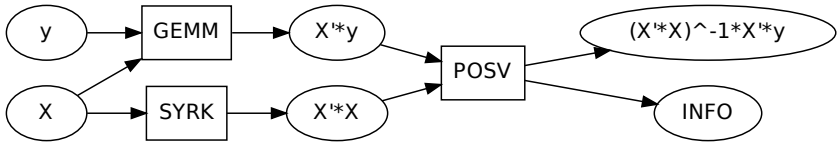


Figure 16: Least squares linear regression with 'SYRK' computation

Numeric Result This optimization is relevant within this application. The `SYRK` computation generally consumes about 50% as much compute time as the equivalent `GEMM`. It reads the input `X` only once and performs an symmetric multiply. The computation of $X^T X$ consumes a significant fraction of the cost within this computation.

Development Result This speedup was both found and implemented by a domain expert. He was able to identify the flaw in the current implementation because the intermediate representations (DAG, Fortran code) were clear and natural to someone in his domain. The code invited inspection. After identification he was able to implement the correct computation (`class SYRK`). The `computations` project for BLAS/LAPACK routines was simple enough for him to quickly engage and develop his contribution. Finally he was able to add a pattern `(A*A.T, SYRK(1.0, A, 0.0, 0), True)` into the compilation system so that his work could be automatically applied. The declarative inputs of the compiler are sufficiently approachable to be used by developers without a background in automated program development.

Numeric Result In section 6.3 we saw that our automated system was able to achieve the same performance as the expert implementation from naive results. The addition of SYRK pushes performance beyond what even specialized functions within the `scipy` stack allow:

```
>>> timeit scipy.linalg.solve(X.T*X, X.T*y, sym_pos=True)
10 loops, best of 3: 33.2 ms per loop

>>> comp = compile([X, y], [beta], Q.fullrank(X))
>>> with assuming(Q.real_elements(X), Q.real_elements(y)):
...     f = build(comp, [X, y], [beta])

>>> timeit f(nX, ny)
10 loops, best of 3: 23.4 ms per loop
```

6.5 Kalman Filter

The Kalman filter is an algorithm to compute the Bayesian update of a normal random variable given a linear observation with normal noise. It is commonly used when an uncertain quantity is updated with the results of noisy observations. Both the prior and the observation are assumed to be normally distributed. It is used in weather forecasting after weather stations report in with new measurements, in aircraft/car control to automatically adjust for changing external conditions, or in GPS navigation as the device updates position based on a variety of noisy GPS/cell tower signals. It is ubiquitous, it is important, and it needs to be computed quickly and continuously. It can also be completely defined with a pair of matrix expressions.

$$\Sigma' = \Sigma H^T \left(H\Sigma H^T + R\right)^{-1} \left(-data + H\mu\right) + \mu$$

$$\mu' = -\Sigma H^T \left(H\Sigma H^T + R\right)^{-1} H\Sigma + \Sigma$$

Math Expressions We define these expressions in SymPy:

```
from sympy import Symbol, MatrixSymbol, latex

n       = Symbol('n')                   # Number of variables in state
k       = Symbol('k')                   # Number of variables in observation
mu      = MatrixSymbol('mu',    n, 1)   # Mean of current state
Sigma   = MatrixSymbol('Sigma', n, n)   # Covariance of current state
H       = MatrixSymbol('H',     k, n)   # Measurement operator
R       = MatrixSymbol('R',     k, k)   # Covariance of measurement noise
data    = MatrixSymbol('data',  k, 1)   # Observed measurement data

# Updated mean
newmu   = mu + Sigma*H.T * (R + H*Sigma*H.T).I * (H*mu - data)
# Updated covariance
newSigma= Sigma - Sigma*H.T * (R + H*Sigma*H.T).I * H * Sigma

assumptions = (Q.positive_definite(Sigma), Q.symmetric(Sigma),
               Q.positive_definite(R), Q.symmetric(R), Q.fullrank(H))
```

Computation We compile these expressions into a computation.

```
comp = compile([mu, Sigma, H, R, data], [new_mu, new_Sigma], *assumptions)
```

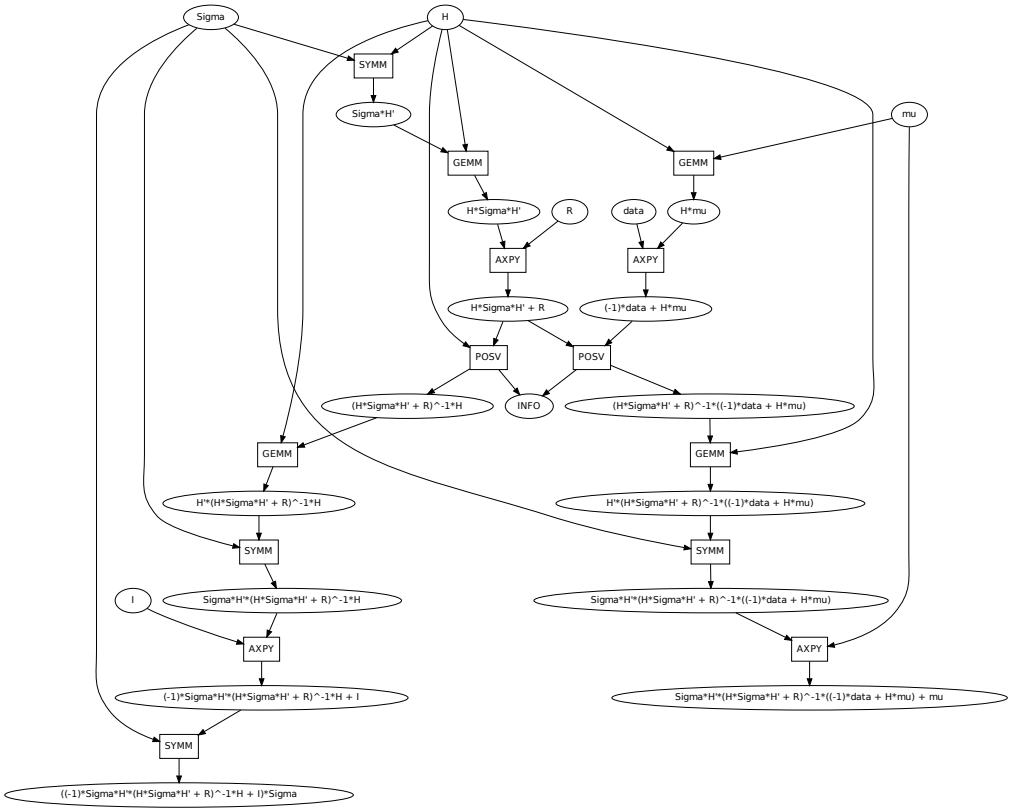


Figure 17: A computation graph for the Kalman Filter

Features We note two features of the computation:

1. Common-subexpressions are identified, computed once, and shared. In particular we can see that $H\Sigma H^T + R$ is shared between the two outputs.
2. This same subexpression is fed into a `POSV` routine for the solution of symmetric positive definite matrices. The inference system determined that because H is full rank, and Σ and R are symmetric positive definite that $H\Sigma H^T + R$ is symmetric positive definite.

The first benefit is trivial in traditional compiled systems but a substantial efficiency within scripting languages.

The second benefit is more substantial. Noticing that $H\Sigma H^T + R$ is symmetric positive definite requires both mathematical expertise and substantial attention to detail. This optimization can easily be missed, even by an expert mathematical developer. It is also numerically quite relevant.

6.6 Analysis

We have seen that the composition of computer algebra, numerical subroutines, and term rewriting can transform mathematical expressions into sophisticated high performance, human readable code. Here we motivate design decisions.

Small Components Enable Reuse Solutions to scientific problems often share structure. Large codes written for specific applications are often too specific to be reusable outside of their intended problem without substantial coding investment. Smaller components designed to solve common sub-problems may be more generally applied.

Smaller Scope Lowers Barriers to Development These components do not depend on each other for development. This isolated nature reduces the expertise requirements on potential developers. Mathematical developers can contribute to SymPy Matrix Expressions even if they are ignorant of Fortran. Computational developers familiar with BLAS/LAPACK can contribute to `computations` even if they are unfamiliar with compilers. Shared interfaces propagate these improvements to users at all levels. The demographics of expertise in scientific computing necessitate this decision.

Multiple Intermediate Representations Encourages Experimentation Broadly applicable software is unable to anticipate all possible use cases. In this situation it is important to provide clean intermediate representations at a variety of levels. This project allows users to manipulate representations at the math/term, DAG, and Fortran levels. Care has been taken so that each representation is human readable to specialists within the relevant domain.

This approach encourages future development within the project. For example to support multiple output languages we only need to translate from the DAG level, a relatively short conceptual distance relative to the size of the entire project. We currently support Fortran and DOT (for visualization), but adding other languages is a straightforward process.

This approach encourages development outside the project. In Section 7 we manipulate the DAG with an external composable static scheduler and then re-inject the transformed result into our compiler chain. Additionally scientific users can use the resulting Fortran90 code as a template for future by-hand development.

7 Heterogeneous Static Scheduling

7.1 Introduction

Recent developments in computations for dense linear algebra engage parallelism at the shared memory, distributed memory, and heterogeneous levels. These developments have required several significant software rewrites among many research teams. In this chapter we adapt our existing system to engage parallel computation by adding another composable component, a static scheduler. We show that this addition can be done separately, without rewriting existing code, which demonstrates both the extensibility of the existing system and durability of the individual components under changing external conditions.

Specifically we implement application-agnostic static scheduling algorithms in isolation. We compose these with our existing components to generate MPI programs to compute dense linear algebra on specific parallel architectures automatically.

Motivating Problem We want to develop high performance numerical algorithms on increasingly heterogeneous systems. This case is of high importance and requires substantial development time. We focus on systems with a few computational units (fewer than 10) such as might occur on a single board within a large high performance computer. Traditionally these kernels are written by hand using MPI. They are then tuned manually. We investigate the feasibility/utility of automation for these tasks.

7.2 Background

Task Scheduling is a broad topic under active development. Approaches in task scheduling can be separated along two different axes

1. The amount of assumed knowledge
2. When the scheduling is performed

The distribution along these axes is highly correlated. In general, systems with more knowledge perform sophisticated analyses that consume significant amounts of resources. These analyses are preferably done only once at compile-time rather than during execution when they may slow down the actual computation. Conversely systems about which little is known often use simple analyses and so can be done cheaply at runtime. While less is known a priori, these cheaper runtime systems can respond more dynamically to events as they occur.

Dynamic Scheduling In general, dynamic scheduling systems do not assume much knowledge about the computation. In the simplest case they blindly schedule operations from the task graph to computational workers as these tasks become available (data dependencies are met) and workers become available (no longer working on the previous job). More sophisticated analyses may try to schedule tasks onto machines more intelligently, for example by preferentially keeping data local to a single machine if possible.

Systems like Condor, Pegasus, or Swift dynamically schedule a directed acyclic graph of tasks onto a pool of workers connected over a network. These systems enable users to define a task graph as a set of processes. They traditionally handle communication over a network file system. Hadoop, a common infrastructure for the MapReduce interface, bears mention. The MapReduce interface allows only a restricted form of dependency graph defined by one-to-one mapping functions and many-to-one reduction functions. This added restriction allows implementations like Hadoop to assume more about the problem and opens up various efficiencies. Hadoop, an implementation, allows substantially more control for reduced communication than MapReduce. For example, by controlling Partitioner objects data locality can be exploited to minimize network or even disk communication. In general, more restrictive models enable more sophisticated runtime analyses.

Static Scheduling The majority of static scheduling research assumes some knowledge both about the costs of tasks and, if the set of agents is heterogeneous, each agent's strengths and weaknesses. This situation has been historically rare in parallel programming but common in operations research. For example operations in the construction and assembly of an automobile or the distribution of goods is often well known ahead of time and automated workers often have widely varying but highly predictable task completion times. These problems are far more regular than a generic program and also far more dependent on the worker agents available (not all agents are equally suited to all tasks.)

In general, optimal scheduling is NP-hard; however algorithms, approximations, and heuristics exist. They differ by leveraging different theory, assuming different symmetries of the problem (e.g. homogeneous static scheduling where all agents are identical) or by assuming different amounts of the knowledge or symmetries about the tasks (all times known, all times known and equivalent, communication times known, communication times are all zero, etc....) Kwok and Ahmed[49] give a good review.

Finally we note that in practice most static scheduling at this level is written by hand. HPC software developers often explicitly encode a schedule statically into their code with MPI calls. This application is the target for this chapter.

Informed Dynamic Scheduling In special cases we may know something about the tasks and the architecture and also want to schedule dynamically for robustness or performance reasons. These situations tend to be fairly specialised. In the context of numerical linear algebra we can consider the communication of blocks or tiles (a term for a small block) around a network.

This approach is taken by systems like Supermatrix, Elemental, BLACS, and most recently, DAGuE[15]. These systems all move regularly sized blocks between connected nodes to perform relatively similarly timed operations. In the case of DAGuE, information about the network can be encoded to better support data locality.

Recent work with PLAMSA [16, 50] shows a trend towards hybrid schedulers where part of the communication is handled dynamically for robustness and part is handled statically for performance. As parallelism increases both sophisticated analyses and robustness are necessary. These can be added at different levels of granularity; for example operations on sets of neighboring nodes can be statically scheduled while calls to this neighborhood can be scheduled dynamically for robustness. Alternatively a top-down static schedule may exist over several high-granularity dynamic schedulers.

7.3 Scheduling as a Component

Numerical linear algebra is the ideal case for scheduling. The operations are well understood, access patterns are regular, and sufficient computation exists to mask communication costs. Numerical linear algebra supports stricter assumptions and the use of more sophisticated analyses. Recent hardware demand for exposing parallelism have pushed software development to adopt increasingly expensive and commonly static solutions. They have also pushed for dedicated and isolated scheduling software solutions.

In Sections 7.4, 7.5 we motivate the predictability of matrix computation and communication times. We then discuss algorithms for static scheduling in Section 7.6. Finally we integrate these software components to our existing family of modules and perform a numerical experiment; we validate its utility in Section 7.7.

Please note that the goal of this work is not to compete with professionally developed linear algebra solutions. Rather our goals are the following

1. To examine the applicability of static scheduling in numerical linear algebra
2. To demonstrate the extensibility of modular software

7.4 Predicting Array Computation Times

Challenges To create high performance task parallel programs at compile time we need to know the compute times of each task on each machine. This task is challenging in general.

Compute times can depend strongly on the inputs (known only at runtime), other processes running on the hardware, behavior of the operating system, and potentially even the hardware itself. Interactions with complex memory hierarchies introduce difficult-to-model durations. Even if estimates of tasks are available the uncertainty may be sufficient to ruin the accuracy of the overall schedule.

Array Programming is Easier Fortunately, scheduling in the context of array operations in a high performance computing context mitigates several of these concerns. Routines found in high performance libraries like BLAS/LAPACK are substantially more predictable. Often the action of the routine depends only on the size of the input, not the contents. Memory access patterns are often very regular and extend beyond the unpredictable lower levels of the cache. Because this computation occurs in a high performance context there are relatively few other processes sharing resources and our task is given relatively high priority.

The Predictability of BLAS Operations We profile the runtime of the `DGEMM` operation. We compute a $1000 \times 1000 \times 1000$ dense matrix multiply 1000 times on a workstation with an Intel i5-3320M running `OpenBLAS`. In Figure 18 we present a time series and in Figure 19 a histogram of the same data. While runtimes are not deterministic we do find that a tight distribution around a central peak with variations less than a percent.

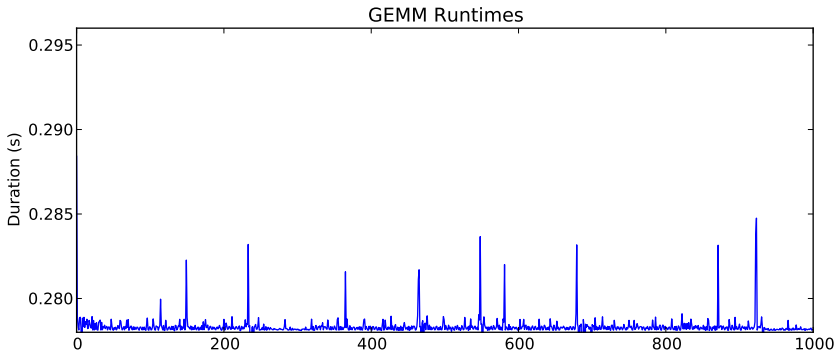


Figure 18: A time series of durations of $1000 \times 1000 \times 1000$ dense matrix matrix multiplies using `DGEMM`

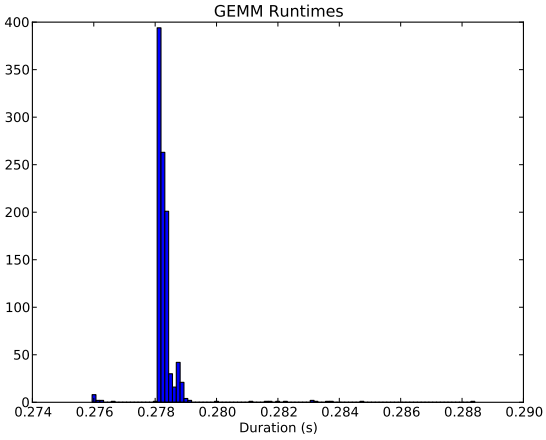


Figure 19: A histogram of durations of dense matrix matrix multiplies

The context in which computations are run is relevant. These times were computed on a workstation running a traditional operating system. To study the effects of runtime context we run this same computation within a Python environment. Compute times are computed strictly within Fortran subroutines but the memory is managed by the Python runtime. A time series and histogram are presented in Figures 20 and 21. These times have a marginally shifted central peak (the median value remains similar) but the distribution has widened in two ways. First, there is a larger population of outliers that require around substantially more time. Second, the central distribution is substantially wider with variations up to a

few percent.

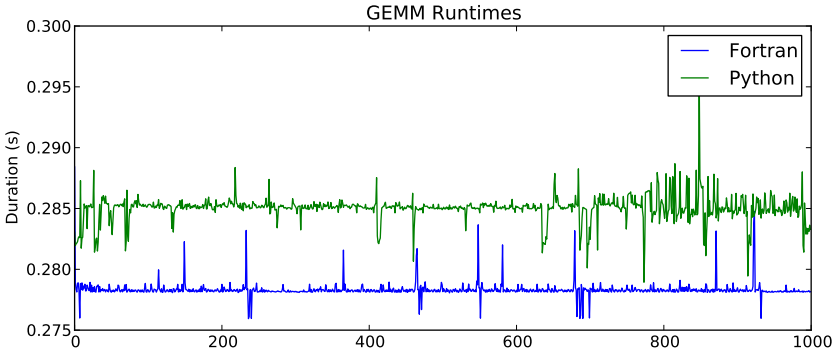


Figure 20: A time series of durations taken in a noisy environment

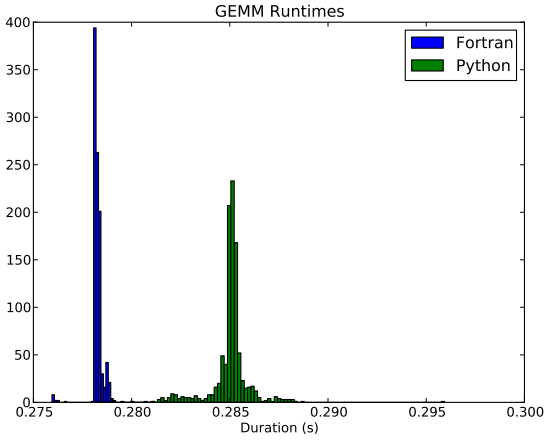


Figure 21: A histogram of durations taken in a noisy environment.

Presumably by running this same computation on a high performance machine with a quiet operating system the uncertainty could be further reduced.

Dynamic Routines These results on `GEMM` are representative of most but not all BLAS/LAPACK routines. Some routines, like `GESV` for general matrix solve do perform dynamic checks at runtime on the content of the array. In special cases, such as when the solving matrix is the identity, different execution paths are taken, drastically changing the execution time. Ideally such conditions are avoided beforehand at the mathematical level; if a matrix is known ahead-of-time to be the identity then SymPy should be able to reduce it before a `GESV` is ever generated. If this information is not known ahead of time then schedules may be invalid. In general we test with random matrices as they are, for most operations, representative of the general/worst case. Even this assumption breaks down under iterative methods like conjugate gradient solution, for which this approach is invalid.

7.5 Predicting Communication Times

In Section 7.4 we analyzed the predictability of computations on conventional hardware. We found that array computations on large data sets were generally predictable to an accuracy of one percent. In this section we perform a similar analysis on communication within a network. We find a similar result that communication times of bulk data transfer are fairly predictable within a local network, in particular a commodity gigabit switch.

We write an MPI computation to transfer arrays between two processes. We profile this computation, generate and run this code on two nodes within a conventional cluster on a wide range of data sizes. We plot the relationship between data size and communication time in Figure 22. We measure both the duration of the `MPI_Send` and `MPI_Recv` calls for data sizes ranging logarithmically from one eight byte float to 10^7 floats. We time each size coordinate multiple times to obtain an estimate of the variance.

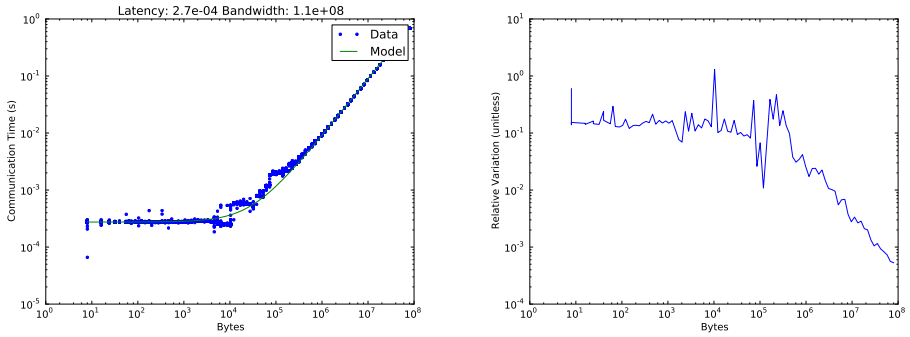


Figure 22: Communication time between two points in a cluster

The image demonstrates that there is a clear polynomial relationship above a few thousand bytes. Further inspection reveals that this relationship is linear, as is expected by a simple latency/bandwidth model. Below this size the linear model breaks down. Uncertainty varies with size but decreases steadily after a few thousand bytes to within a few percent.

We conclude that for this architecture communication times are both predictable and modelable above a few thousand bytes, at least for the sensitivity required for our applications.

We note that this result is for a particular communication architecture and a particular implementation of MPI. Our test cluster is a shared commodity cluster. We expect that results in a high performance setting would not be less predictable.

7.6 Static Scheduling Algorithms

Complexity Optimal scheduling is NP-Complete[49]; which limits the scale of optimally schedulable programs. We can get around this problem in a few ways:

1. Array programs can often be written with relatively few tasks, making the full NP-Complete problem feasible for interesting problems.

2. Robust approximation algorithms exist for common NP-Complete problems (e.g. integer linear programming.)
3. Heuristics for heterogeneous static scheduling exist.

In this section we connect existing work in static DAG scheduling to our linear algebra compilation system. We first describe an interface between linear algebra computations and schedulers, and then describe two schedulers that match this interface.

Interface We use the following interface for heterogeneous static scheduling:

Inputs:

- Task Directed Acyclic Graph
- Graph of Computational Agents
- Compute Time function : `Task` $\times$ `Agent` $\rightarrow$ `Time`
- Communication Time function : `Variable` $\times$ `Agent` $\times$ `Agent` $\rightarrow$ `Time`

Outputs:

- Mapping of `Agent` $\rightarrow$ `Task Directed Acyclic Graph`

That is we take information about a computation (a DAG), a network (a graph), and compute and communication times (functions) and produce a set of sub-computations (a set of DAGs) such that each sub-computation is assigned to one of the worker agents.

We implement two static scheduling algorithms that satisfy this interface.

Mixed Integer Linear Programming We pose the heterogeneous static scheduling problem as a mixed integer linear program as was done by Tompkins[51]. Integer programming is a standard description language with a rich theory and mature software solutions. It is an NP-Complete problem with a variety of approximation algorithms. It is a common intermediate representation to computational engines in operations research.

Dynamic List Scheduling Heuristic We also experiment with the Heterogeneous Earliest Finish Time (HEFT)[52] heuristic. This heuristic runs in polynomial time but is greedy and does not guarantee optimal solutions.

HEFT operates with two steps. It first assigns a very rough time-to-completion score to each task, based on the score of all of its dependencies and the average compute time across all of the heterogeneous workers. It then schedules each element of this list onto the computational resource that will execute that job with the earliest finish time. This second step takes into account both the known duration of that task on each machine and the communication time of all necessary variables from all other machines on which they might reside. It remains a greedy algorithm (it is unable to accept short-term losses for long-term optimality) but may still perform well in many situations.

7.7 Proof of Concept

To show how schedulers interoperate with our existing compilation chain we walk through a simple example. We build a parallel code for the matrix expression `(A*B).I * (C*D)` for execution on a two node system. The `conglomerate` project discussed in Chapter 6 transforms this expression into the computation shown in Figure 23.

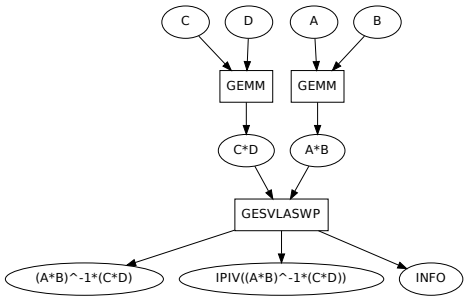


Figure 23: A simple computation for parallelization

Our two node system consists of two workstations (Intel Core i7-3770 with 8GB memory) over a gigabit switch. Profiling the network shows that connections can be well characterized by a latency of $270\mu s$ and a bandwidth of `1.1e8 Bytes/s`. With inputs of random 2000×2000 matrices computation times are as follows

Computation	Mean (s)	Standard Error (s)
GEMM	4.19	0.017
GEMM	4.19	0.012
GESV/LASWP	8.57	0.038

Feeding this information into either of our static schedulers (they produce identical results) we obtain the computations in Figure 24 with a total runtime of `13.03` seconds. Note that the graph has been split and asynchronous MPI computations have been injected to handle communication. In particular the matrix multiplications are done in parallel and then collected onto a single node to perform the final general matrix solve `GESV/LASWP`.

We ran this program ten times on our two-node system and record runtimes with mean of `13.17` and standard error of `0.017`. This measurement is different from the predicted time of `13.03`. The recorded uncertainty observed in both computation times and communication times is not sufficient to account for this discrepancy; clearly there exists some unaccounted factor. Still, on a macroscopic level the runtime is sufficiently close to the predicted time to be useful operationally. This speed-up demonstrates the feasibility of an end-to-end solution from high-level matrix expressions to hardware-specific MPI code.

7.8 Parallel Blocked Matrix Multiply

The above example is the minimum viable parallel matrix computation. In this section, we discuss a more relevant example and show both the potential power and drawbacks of this technique. Strong scaling can be achieved in matrix operation through blocking. As discussed in Section 8.2, SymPy is capable of providing all necessary blocked definitions for various operations. In this section, we create and execute a computation to perform a matrix-matrix multiply across two computational nodes.

As in Section 7.7, we form our problem in SymPy, pass it the `conglomerate` project to construct a task graph, profile each of these nodes on our workers and pass this information to the scheduler to partition the computation graph in two. We then rely on the code generation elements of `computations` to generate MPI code. This code blocks the two matrices and performs the various `GEMM` and `AXPY` calls while transmitting intermediate results across the network.

Positive Results For large matrices over a fast interconnect this problem can be parallelized effectively. Scheduled times are substantially shorter than the sequential time. Additionally, this technique can be used in the non-general matrix multiply case. Symmetric matrix multiply or even more exotic symmetric-symmetric or symmetric-triangular blocked variants can be generated and effectively scheduled. This approach allows for the construction of parallel routines specially tailored to a particular architecture and mathematical operation.

Negative Results Unfortunately on our test framework the executed runtimes do not match the predictions produced by the scheduler. Upon close inspection this mismatch is owed to a mismatch in assumptions made by the scheduler and the common MPI implementations. As they are written the schedulers assume perfect asynchronous communication/computation overlap. On our architecture with our MPI implementation (`openmpi-1.6.4`), this assumption is not the case and valid transactions for which both the `iRecv` and `iSend` calls have occurred are not guaranteed to transmit immediately, even if the network is open. The underlying problem is the lack of a separate thread for communication in the MPI implementation.

Details on the Communication Issue To better understand the communication failure we focus on a particular computation within our program, the multiplication of `X*Y` via a `GEMM` on Machine 1. Times have been rounded for presentation. General magnitudes and ordering have been preserved.

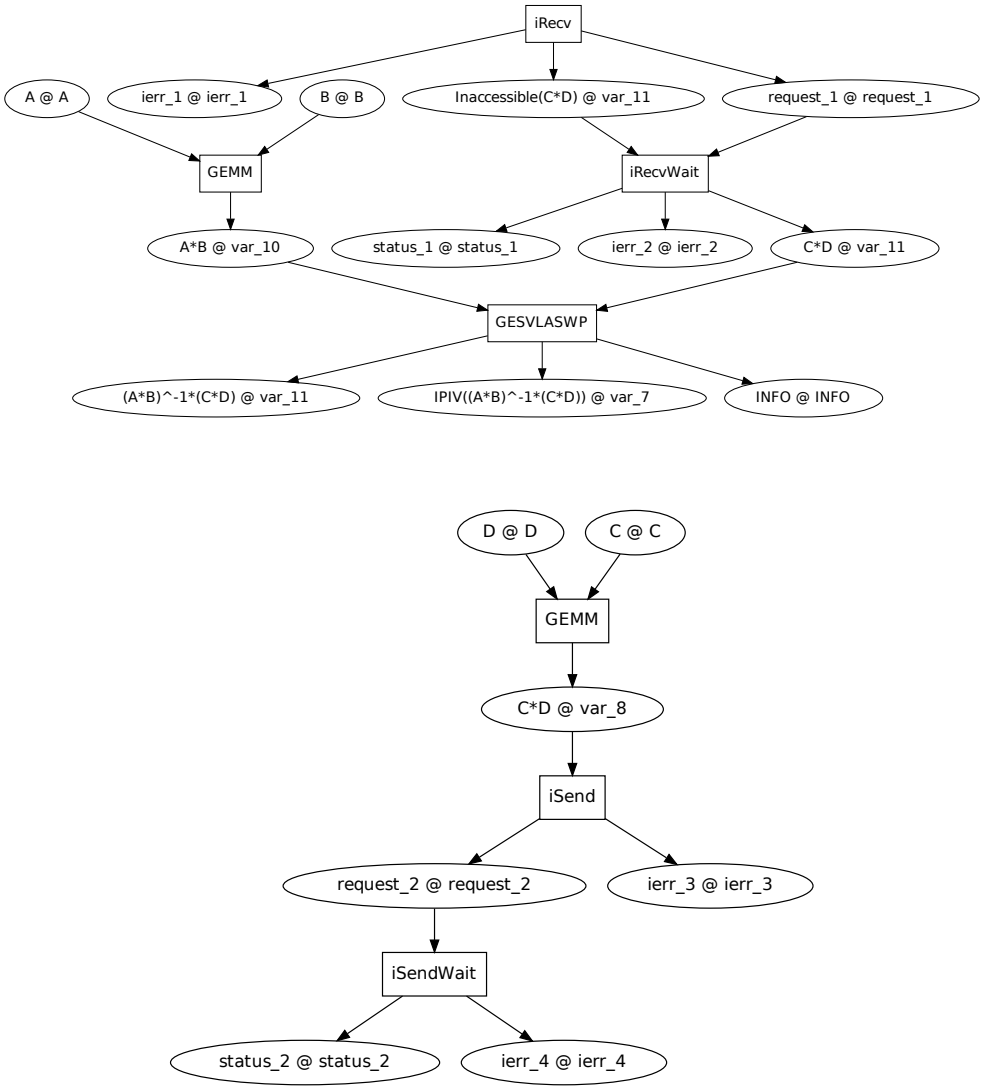


Figure 24: The computation from Fig. 23 scheduled onto two nodes

```
Machine 1                    Actual Start Time    Scheduled Start Time

iRecv(A from 2)                           0.00                    0.00
Y = ....                                  0.01                    0.01
iRecv(X from 2)                           0.02                    0.02
Wait(transfer of B from 2)                0.03                    0.03
[1.0, X, Y, 0, 0] -> GEMM -> [...]        3.10                    0.13

Machine 2

iSend(X to 1)                             0.00                    0.00
... Time consuming work ...
iSend(A to 1)                             3.00                    3.00
```

Note the discrepancy between the actual and scheduled start time of the `GEMM` operation on Machine 1. This operation depends on `X`, generated locally, and `Y`, transferred from Machine 2. Both the `iSend` and `iRecv` for this transfer started near the beginning of the program. Based on the bandwidth of `1e8 Bytes/s` and data size of $1000^2 * 8Bytes$, we expect a transfer time of around `0.1` seconds as is reflected in the schedule. Instead, from the perspective of Machine 1, the `Wait` call blocks on this transfer for three seconds. It appears that the transfer of `X` is implicitly blocked by the transfer of `A`, presumably by scheduling policies internal to the MPI implementation. As a result of this inability to coordinate asynchronous transfers across the machines at precise times, the computation effectively runs sequentially.

Approaches for Resolution This problem could be resolved in the following ways:

- The spawning of threads to handle simultaneous communication (so-called progress threads). Historically many MPI implementations have used threads to handle communication even when control is not explicitly given to the library. These are disabled by default for due to overhead concerns in common-case applications and development support for them has ceased (at least in our implementation). This deprecated feature suits our needs well.
- The careful generation of MPI calls that are mindful of the MPI scheduler in question. Currently `iRecv` several calls are dumped at the beginning of the routine without thought to how they will be interpreted by the internal MPI scheduler. By matching this order with the expected availability of data across machines implicit blocks caused by the scheduler may be avoided.
- The improvement of internal MPI schedulers. Rather than generate code to satisfy a scheduler work could be done to improve the schedulers themselves, making them more robust to situations with several open channels.
- The modification of schedulers for synchronous communication. These complications can be avoided by abstaining from asynchronous communication. This degrades performance, particularly when communication times are on par with computation times, but greatly simplifies the problem. Our current schedulers do not match this

model (they assume asynchronicity) but other schedulers could be found or developed and inserted into the compilation chain without worry.

We leave these approaches for future work.

8 Extensibility

A good tool can be applied to unforeseen problems. The hand held hammer can be applied beyond the application of driving nails. It can be extended to any activity that requires the delivery of percussive force. A good tool can be composed with other unanticipated tools. For example the hammer composes well with the end of a wrench to apply percussive torsion onto tough bolts.

In this same sense the computational tools discussed in this dissertation must be tested in novel contexts outside of the application for which they were originally designed. The following sections present examples using these software components in novel contexts. The following work demonstrates substantial results with trivial efforts.

In Section 8.1 we demonstrate interchangeability by swapping out our prototypical `computations` project for BLAS/LAPACK code generation with `Theano`, a similar project designed for array computations in machine learning. In Section 8.2 we show that improvements isolated to a single module can reverberate over the entire system by using the mathematical blocking known in SymPy to develop and execute blocked matrix algorithms. Finally in Section 8.3 we apply the ideas of term rewriting and modularity to the field of statistics to demonstrate applicability outside of linear algebra.

8.1 Theano Backend

Modular components allow interchangeability. This principle enables experimentation with alternative implementations of some of the components without rewriting the entire system. Fine software granularity encourages evolution and avoids global rewrites.

We demonstrate this virtue by swapping out our `computations` backend for BLAS/LAPACK routines with Theano, a Python package for array computing. Theano comes from the machine learning community. It was developed with different goals and so has a different set of strengths and weaknesses. It supports NDArrays, non-contiguous memory, and GPU operation but fails to make use of mathematical information like symmetry or positive-definiteness. It is also significantly more mature and its longevity extends beyond that of most research projects. In particular, it is the natural surrogate to the `computations` project, should it fail to persist.

Kalman Filter In Section 6.5 we mathematically defined the Kalman Filter in SymPy and then implemented it automatically in `computations`. We do the same here with Theano.

```
from sympy import Symbol, MatrixSymbol, latex
```

```
n       = Symbol('n')                   # Number of variables in state
k       = Symbol('k')                   # Number of variables in observation
mu      = MatrixSymbol('mu',    n, 1)   # Mean of current state
Sigma   = MatrixSymbol('Sigma', n, n)   # Covariance of current state
H       = MatrixSymbol('H',     k, n)   # Measurement operator
R       = MatrixSymbol('R',     k, k)   # Covariance of measurement noise
data    = MatrixSymbol('data',  k, 1)   # Observed measurement data

# Updated mean
newmu   = mu + Sigma*H.T * (R + H*Sigma*H.T).I * (H*mu - data)
# Updated covariance
newSigma= Sigma - Sigma*H.T * (R + H*Sigma*H.T).I * H * Sigma

assumptions = (Q.positive_definite(Sigma), Q.symmetric(Sigma),
               Q.positive_definite(R), Q.symmetric(R), Q.fullrank(H))
```

We take this same mathematical definition and generate a Theano graph and runnable function.

```
from sympy.printing.theanocode import theano_function
inputs  = [mu, Sigma, H, R, data]
outputs = [newmu, newSigma]
dtypes  = {i: 'float64' for i in inputs}

f = theano_function(inputs, outputs, dtypes=dtypes)
```

Theano builds a Python function that calls down to a combination of low-level `C` code, `scipy` functions, and calls to static libraries. This function takes and produces `numpy` arrays corresponding to the symbolic `inputs` and `outputs`. Any SymPy matrix expression can be translated to and run by Theano in this manner.

This framework allows us to experiment with and evaluate features present in different systems. For example the value of adding GPU operations to `computations` may be evaluated first by viewing their value in Theano.

8.2 Blocked Kalman Filter

The addition of expertise to a single module may reverberate throughout the greater project. In this example we investigate the added value of mathematical matrix blocking, known by SymPy, across the larger application. We continue to use the Kalman filter as an example computation and Theano as a backend.

Blocked Execution If arrays are too large to fit comfortably in the fastest parts of the memory hierarchy then each sequential operation needs to move large chunks of memory in and out of cache during computation. After one operation completes the next operation moves around the same memory while it performs its task. This repeated memory shuffling impedes performance.

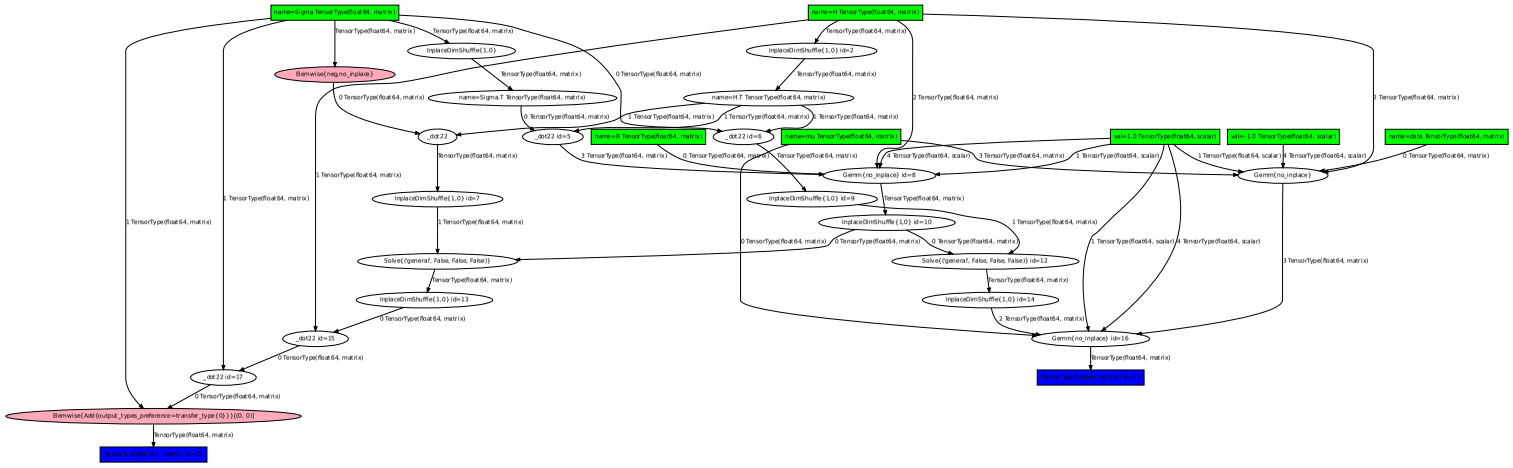


Figure 25: Theano computation graph for the Kalman Filter

A common approach to reduce memory shuffling is to cut the computation into smaller blocks and then perform as many computations as possible on a single block before moving on. This is a standard technique in matrix multiplication. The multiplication of two 2×2 blocked matrices can be expanded using the same logic that one uses to multiply matrices of scalar expressions

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & K \end{bmatrix} \rightarrow \begin{bmatrix} AE + BG & AF + BK \\ CE + DG & CF + DK \end{bmatrix}$$

We are now able to focus on substantially smaller chunks of the array that fit more comfortably in memory allowing us to improve memory locality during execution. For example we can choose to keep `A` in local memory and perform all computations that involve `A` (i.e. AE, AF) before releasing it permanently. We will still need to shuffle some memory around (this need is inevitable) but by organizing with blocks we're able to shuffle less. This idea extends beyond matrix multiplication. Matrix inverse expressions can also be expanded.

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} \rightarrow \begin{bmatrix} \left(-BD^{-1}C + A\right)^{-1} & -A^{-1}B\left(-CA^{-1}B + D\right)^{-1} \\ -\left(-CA^{-1}B + D\right)^{-1}CA^{-1} & \left(-CA^{-1}B + D\right)^{-1} \end{bmatrix}$$

High performance dense linear algebra libraries hard-code these tricks into individual routines. The call to the general matrix multiply routine `GEMM` performs blocked matrix multiply within the call. The call to the general matrix solve routine `GESV` can perform blocked matrix solve. Unfortunately these routines are unable to coordinate blocked computation *between* calls.

Fortunately, SymPy can generate these high-level blocked matrix mathematical expressions at compile time and Theano can generate code for them.

General Code to Block the Kalman Filter SymPy can define and reduce the blocked Kalman filter using matrix relations like those shown above for multiplication and inversion. The listing below shows all the code necessary to block the Kalman filter into blocks of size `n/2`. The code is dense and not particularly insightful but demonstrates that blocking, a general mathematical transformation can be transferred to a computational context with a small amount of general purpose glue code. SymPy is able to block a large computation with general purpose commands. No special computation-blocking library needs to be built.

The mathematical expression is then transformed to a computation with our traditional approach. No new interface is required for the increase in mathematical complexity. The two systems are well isolated. We can translate the expression into a Theano graph and compile it to low-level code with the same process as in Section 8.1. The resulting computation calls and organizes over a hundred operations; both the mathematics and the computation would be a difficult to coordinate by hand.

Numeric Results We measure performance by timing the standard and blocked variants of the Kalman filter

```
>>> timeit f(*ninputs)
1 loops, best of 3: 2.69 s per loop

>>> timeit fblocked(*ninputs)
1 loops, best of 3: 2.12 s per loop
```

This performance increase is substantial in this case but dependent on many factors, most notably the relationship between matrix size and memory hierarchy and the sequential BLAS implementation. Conditions such as with small matrices with generous memory conditions are unlikely to see such an improvement and may even see a performance degradation.

Blocked matrix multiply and blocked solve routines have long been established as a good idea. High level mathematical and array programming libraries like SymPy and Theano allow us to extend this good idea to *arbitrary* array computations composed of these operations. Moreover, experimentation with this idea is simple, requiring only a few lines of high-level general purpose code rather than a new software project.

High-level modular systems with mathematical components enable experimentation. Note that we have not introduced a new library for interoperation blocked dense linear algebra. Instead we compose pre-existing general purpose high-level tools to that effect. Block matrix manipulations were not developed for this application but are instead a commonly occurring mathematical sub-problem that is useful to have around.

Multiple Backends Because we invested in interfaces, we were able to trivially plug in a different backend. This ability is critical for the comparison and evaluation of components instead of systems. It also allows features to flow more smoothly between systems. A loose federation of components is less brittle than a monolithic system. Components with access to multiple clients encourage comparison, experimentation, and overall accelerate the evolution of scientific software.

8.3 SymPy Stats

The components and concepts discussed above are applicable to domains outside of numerical linear algebra. In this section we apply these same ideas to probability and uncertainty. In our past work[53] we support uncertainty modeling in computer algebra systems through the addition of a random variable type. The random variable abstraction allows existing computer algebra functionality to compose cleanly with the notions of probability and statistics. In this section we discuss the design in that system relevant for the discussion of modularity and show how work on term rewrite systems in this document is able to apply to that older project.

Composition in SymPy.stats We enable the expression of uncertain systems in computer algebra through the addition of a random variable type. A random variable is an algebraic symbol attached to a probability space with a defined probability distribution. Expressions containing random variables themselves become random. Expressions containing multiple random variables exist over joint probability distributions. The addition of conditions restricts

the space over which these distributions have support. Queries on random expressions generate deterministic computations.

SymPy.stats leverages existing SymPy objects to describe these concepts. Distributions are described with scalar expressions, joint and conditional probability spaces with SymPy sets, and results with SymPy integral expressions. SymPy.stats offloads difficult computations onto other, more heavily curated systems, which was the motivating design principle of this project. This organization enables the expression of statistical expertise in isolation from these computational concerns.

Example We generate a random variable distributed with a normal distribution with mean μ and variance σ^2.

```
>>> mu = Symbol('mu', real=True)
>>> sigma = Symbol('sigma', positive=True)
>>> X = Normal('X', mu, sigma)
```

We query for the probability that this variable is above some value y

```
>>> P(X > y)
```

$$\frac{1}{2}\operatorname{erf}\left(\frac{\sqrt{2}\,(\mu - y)}{2\sigma}\right) + \frac{1}{2}$$

Internally this operation produces a definite integral

```
>>> P(X > y, evaluate=False)
```

$$\int_0^\infty \frac{\sqrt{2} e^{-\frac{(z-\mu+y)^2}{2\sigma^2}}}{2\sqrt{\pi}\sigma}\, dz$$

SymPy.stats then relies on SymPy's internal integration routines to evaluate the integral.

For more complex queries SymPy.stats uses other SymPy utilities like equation solving and differentiation to manipulate queries on random expressions into the correct integrals

```
>>> P(X**2 + 1 > y, evaluate=False)
```

$$\int_0^\infty \frac{\sqrt{2}\left(e^{2\frac{\mu\sqrt{z+y-1}}{\sigma^2}} + 1\right) e^{\frac{-z-\mu^2-2\mu\sqrt{z+y-1}-y+1}{2\sigma^2}} \left|\frac{1}{\sqrt{z+y-1}}\right|}{4\sqrt{\pi}\sigma}\, dz$$

Benefits SymPy.stats tries to be as thin a layer as possible, transforming random expressions into integral expressions and no further. This transformation is simple and robust for a large class of expressions. It does not attempt to solve the entire computational problem on its own through, for example, the generation of Monte Carlo codes.

Fortunately its output language, integral expressions, are widely supported. Integration techniques benefit from a long and rich history including both analytic and numeric techniques. By robustly transforming queries on random expressions into integral expressions and then stepping aside, `sympy.stats` enables the entire integration ecosystem access to the domain of uncertainty.

Rewrite Rules We show how rewrite rules, applied by LogPy, can supply a valuable class of information not commonly found in computer algebra systems. Consider two standard normal random variables

```
>>> X = Normal('X', 0, 1)
>>> Y = Normal('Y', 0, 1)
```

SymPy is able to compute their densities trivially (these are known internally).

```
>>> pdf = density(X)
>>> pdf(z)
```

$$\frac{\sqrt{2}e^{-\frac{1}{2}z^2}}{2\sqrt{\pi}}$$

Using equation solving and differentiation SymPy.stats is able to compute densities of random expressions containing these random variables

```
>>> pdf = density(2*X)
>>> pdf(z)
```

$$\frac{\sqrt{2}e^{-\frac{1}{8}z^2}}{4\sqrt{\pi}}$$

SymPy.stats uses existing analytic integration code to marginalize over multivariate probability spaces, supporting interactions between different random variables.

```
>>> simplify(density(X+Y)(z))
```

$$\frac{e^{-\frac{1}{4}z^2}}{2\sqrt{\pi}}$$

This system is robust and can handle a wide variety of non-linear equations.

```
>>> density(X**2)(z)
```

$$\frac{\sqrt{2}e^{-\frac{1}{2}z}\left|\frac{1}{\sqrt{z}}\right|}{2\sqrt{\pi}}$$

The next expression however generates an integral that is too complex for the analytic integration routine. We display the unevaluated integral.

```
>>> density(X**2 + Y**2)(z)
```

$$\int_{-\infty}^{\infty} \frac{\sqrt{2}e^{-\frac{1}{2}X^2} \int_{-\infty}^{\infty} \frac{\sqrt{2}e^{-\frac{1}{2}Y^2}\delta\left(X^2+Y^2-z\right)}{2\sqrt{\pi}}\, dY}{2\sqrt{\pi}}\, dX$$

This result is to be expected. The integrals involved in analytic uncertainty modeling quickly become intractable. At this point we may send this integral to one of the many available numeric systems.

Applying Statistical Expertise A moderately well trained statistician can immediately supply the following solution to the previous, intractable problem.

$$\frac{1}{2}e^{-\frac{1}{2}z}$$

Statisticians are not more able to evaluate integrals, rather they apply domain knowledge at the higher statistical level. When X and Y are standard normal random variables the expression $X^2 + Y^2$ has a chi-squared distribution with two degrees of freedom. This distribution has the known form above. SymPy.stats actually knows the Chi squared distribution internally, but was unable to recognize that $X^2 + Y^2$ fell into this case.

This relation is just one of an extensive set of relations on univariate distributions. An extensive and well managed collection exists at `http://www.math.wm.edu/~leemis/chart/UDR/UDR.html`. Additional information has been generated by the group that maintains `APPL`, "An Probabilistic Programming Language" [54].

The application of high-level expertise simplifies this problem and avoids an intractable problem at a lower level.

Rewrite Rules Rules for the simplification of such expressions can be written down in SymPy as follows

```
patterns = [
  (Normal('X', 0, 1),                      StandardNormal('X'),    True),
  (StandardNormal('X')**2,                 ChiSquared('X', 1),     True),
  (ChiSquared('X', m) + ChiSquared('Y', n), ChiSquared('X', n + m), True),
  ...
  ]
```

Note that these rules are only valid within a `Density` operation when the user is querying for the distribution of the expression. They are not true in general because they destroy the notion of which distributions correspond to which random variables in the system (note the loss of `'Y'` in the last pattern).

These expressions are clear to a statistical user, even if that user is unfamiliar with computer algebra.

Conclusion The automated application of domain expertise at a high level simplifies the eventual computation required at lower-levels. This idea extends throughout many fields of mathematics and the sciences. Declarative techniques allow large pre-existing knowledgebases to be encoded by domain experts and facilitate the expression of this expertise. Systems like LogPy are generally applicable within computer algebra, not only within the scope of numerical linear algebra.

9 Conclusion

This dissertation promotes the value of modularity in scientific computing, primarily through the construction of a modular system for the generation of mathematically informed numerical linear algebra codes. We have highlighted cases where modularity is particularly relevant in scientific computing due to the abnormally high demands and opportunities from deep expertise. We hope that this work motivates the use of modularity even in tightly coupled domains. Additionally, software components included in this work are useful in general applications and are published online with liberal licenses.

9.1 Challenges to Modularity

We take a moment to point out the technical challenges to modular development within modern contexts. This is separate from what we see as the two primary challenges of lack of incentives and lack of training.

Coupled Testing and Source Testing code is of paramount importance to the development of robust and trusted software. Modern practices encourage the simultaneous development of tests alongside source code. I believe that this practice unnecessarily couples tests, which serve as a de facto interface for the functionality of code, to one particular implementation of source code. This promotes one implementation above others and stifles adoption of future attempts at implementing the same interface.

Additionally, as software increases in modularity packages become more granular. It is not clear how to test interactions between packages if tests are coupled to particular source repositories. Thus, we propose the separation of testing code into first class citizens of package ecosystems.

Package Management Software is often packaged together in order to reduce the cost of configuration, build, and installation. As modern ecosystems develop more robust package managers this cost decreases, thus enabling finer granularity and increased modularity. Multi-lingual ecosystems without good standards on versioning can complicate this issue substantially. Package management and installation tools such as pip/easy_install (Python) and CRAN (R) have alleviated these problems for many novice users. They continue to break under more complex situations. Further development into this field is necessary before wide-spread use of fine-grained modularity is possible.

9.2 Analytics

This dissertation argues for the value of modularity. Though it is difficult to quantitatively measure this value, it may be that such a quantitative measure would help to raise the issue among non-enthusiast programmer communities.

To this end I suggest the quantitative study of the scientific software ecosystem. Pervasive use of version control (e.g. `git`) and the recent popularity of online source control communities (e.g. `github`) provide a rich dataset by which we can quantify relationships among code and developers. Relationships between projects (which projects use what others) can be found from dependency managers (e.g. PyPI, Clojars). Relationships within code (which functions use which others) can be found by parsing the source. Relationships between developers and code (who built what and how much work did it take) can be found from commit logs. Relationships between developers (who talks to whom) can be found from online communities (e.g. `github`).

These relationships describe a large complex graph. The code elements of this graph can be analyzed for modularity as defined in a complex networks sense[55]. The commit logs can be analyzed to attribute cost to various elements of code.

This process has at least the following two benefits:

- By assigning a value to programmer time and identifying modular elements we may be able to attribute an added cost of tightly coupled, unmodular code.
- By looking at download counts and dependency graphs we can attribute impact factors to projects, teams, or individual developers. By publishing an impact factor that benefits from good software engineering we hope to encourage better practices in the future.

Understanding and expertise precede optimization.

9.3 Achievements

We summarize key achievements contained in this work. These contributions are either concrete software contributions or general demonstrations of principles.

9.3.1 Software

Concrete software contributions include the following:

SymPy Matrix Expressions An extension of a computer algebra system to matrix algebra including both a general language and a robust inference system. It serves as repository for commonly used theory of the style found in the popular Matrix Cookbook. We demonstrated the value of this theory in the creation of numerical computations.

Computations A high level encapsulation of popular low-level libraries, particularly BLAS/LAPACK, MPI, and FFTW. This system lifts the de-facto scientific programming primitives to a level that is more accessible to non-expert users. It also serves as a high-level target for automated systems, encapsulating many of the simple decisions made in traditional compilation.

Term, LogPy The `term` and `logpy` libraries support the composition of logic programming with pre-existing software projects within the Python ecosystem. In particular they enable the high-level description of small transformations of terms directly within the syntax of the host language.

Conglomerate We compose the above three elements to translate mathematical matrix expressions into low-level Fortran code that makes sophisticated use of low-level libraries. This conglomerate project brings the power of mature but aging libraries into new communities without a tradition in low-level software. It serves as a repository for the expertise of numerical analysis.

Static Schedulers We provide a high-level interface and two isolated implementations of static schedulers for heterogeneous parallel computing.

SymPy.stats An extension of SymPy enables the expression of uncertainty in mathematical models. The abstraction of random variables allows the concepts of uncertainty to compose cleanly with other elements of the computer algebra system. This module is the first such system within a general computer algebra system. By relying on and translating to other well used interfaces (like integral expressions) it is able to be broadly accessible while tightening its development scope enabling single-field statisticians to have broad impact in a range of applications.

9.3.2 Principles

This dissertation demonstrates the value of small composable software modules that align themselves with existing specialist communities. Experiments in this dissertation focused on the ease with which systems to select and implement sophisticated methods could be developed once the software system was properly separated. We showed how experts in linear algebra, numerical libraries, and statistics could each provide improvements that significantly impacted numerical performance. Each of these improvements was trivial for someone within that field and did not depend on simultaneous expertise in other fields.

As the scope of computational science continues to expand we believe that adaptation to developer demographics will increase in importance.

References

[1] John Michalakes and Manish Vachharajani. Gpu Acceleration of Numerical Weather Prediction. *Parallel Processing Letters*, 18(04):531–548, December 2008.

[2] John Michalakes and Manish Vachharajani. GPU acceleration of numerical weather prediction. *Parallel Processing Letters*, pages 1–18, 2008.

[3] Jarno Mielikainen and B Huang. Improved GPU/CUDA Based Parallel Weather and Research Forecast (WRF) Single Moment 5-Class (WSM5) Cloud Microphysics. *Selected Topics in Applied Earth Observations and Remote Sensing*, 5(4):1256–1265, 2012.

[4] SRM Barros, D Dent, and L Isaksen. The IFS model: A parallel production weather code. *Parallel Computing*, 8191(95), 1995.

[5] Edward Anderson. *LAPACK Users' guide*, volume 9. Siam, 1999.

[6] R Clint Whaley and Jack J Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1—-27. IEEE Computer Society, May 1998.

[7] Kazushige Goto and R Van De Geijn. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software (TOMS)*, pages 1–17, 2008.

[8] Paolo Bientinesi, John a. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert a. Van De Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[9] Field G. Van Zee. `libflame`*: The Complete Reference*. `lulu.com`, 2009.

[10] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations*. `lulu.com`, 2008.

[11] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK users' guide*, volume 4. Society for Industrial and Applied Mathematics, 1987.

[12] JJ Dongarra and RC Whaley. LAPACK Working Note 94 A User's Guide to the BLACS v1. Technical report, 1997.

[13] Jack Poulson, Bryan Marker, Robert A Van de Geijn, Jeff R Hammond, and Nichols A Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions in Math and Software*, 39(2), 2013.

[14] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1432–1441, May 2011.

[15] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 38(1-2):37–51, January 2012.

[16] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180:012037, July 2009.

[17] KE Iverson. *A Programming Language*. 1962.

[18] Conrad Sanderson. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. Technical report, NICTA, 2010.

[19] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[20] T. Veldhuizen. Expression templates. *C++-Report*, 7(5):26–31, jun 1995.

[21] K. Iglberger, G. Hager, J. Treibig, and U. Rüde. Expression templates revisited: A performance analysis of current methodologies. *SIAM Journal on Scientific Computing*, 34(2):C42–C69, 2012.

[22] K. Iglberger, G. Hager, J. Treibig, and U. Rude. High performance smart expression template math libraries. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 367–373, 2012.

[23] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[24] M Puschel and JMF Moura. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[25] So Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, (Ci):9887–9897, 2003.

[26] Geoffrey Belter, E. R. Jessup, Ian Karlin, and Jeremy G. Siek. Automating the generation of composed linear algebra kernels. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 59:1–59:12, New York, NY, USA, 2009. ACM.

[27] Diego Fabregat-Traver and Paolo Bientinesi. A domain-specific compiler for linear algebra operations. In *High Performance Computing for Computational Science – VECPAR 2012*, volume 7851 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2013.

[28] Diego Fabregat-Traver and Paolo Bientinesi. Application-tailored linear algebra algorithms: A search-based approach. *International Journal of High Performance Computing Applications*, 27(4):426–439, 2013.

[29] WA Martin and RJ Fateman. The MACSYMA system. *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, 1971.

[30] Megan Florence, S Steinberg, and P Roache. Generating subroutine codes with MACSYMA. *Mathematical and Computer Modeling*, I(4):1107–1111, 1988.

[31] SymPy Development Team. *SymPy: Python library for symbolic mathematics*, 2012.

[32] Martin Davis, G Logemann, and D Loveland. A machine program for theorem-proving. *Communications of the ACM*, 1962.

[33] Kaare Petersen and Michael Pedersen. The Matrix Cookbook. 2008.

[34] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: A CPU and GPU Math Compiler in Python. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, number Scipy, pages 1–7, Austin, TX, 2010.

[35] AD Rich and DJ Jeffrey. A knowledge repository for indefinite integration based on transformation rules. *Intelligent Computer Mathematics*, (1):2–5, 2009.

[36] M Clavel, S Eker, P Lincoln, and J Meseguer. Principles of maude. *Electronic Notes in Theoretical Computer Science*, 4:65–89, 1996.

[37] P Borovanský and C Kirchner. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.

[38] Eelco Visser. Program transformation with Stratego/XT. rules, strategies, tools, and systems in Stratego. *Domain-Specific Program Generation*, (February):315–349, 20024.

[39] Stuart Jonathan Russell, Peter Norvig, John F. Canny, Jitendra M. Malik, and Douglas D. Edwards. *Artificial Intelligence, a Modern Approach*. Prentice hall, Englewood Cliffs, 3 edition, 1995.

[40] Jim Christian. Flatterms, discrimination nets, and fast term rewriting. *Journal of Automated Reasoning*, 10(1):95–113, 1993.

[41] SM Eker. Associative-commutative matching via bipartite graph matching. *The Computer Journal*, 1995.

[42] Leo Bachmair, Ta Chen, and IV Ramakrishnan. Associative-commutative discrimination nets. *TAPSOFT'93: Theory and Practice of Software Development*, 1993.

[43] H Kirchner and PE Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 2001.

[44] Matthew Rocklin. Matrix algebra: Linear algebra in maude, 2012.

[45] William E Byrd. *Relational programming in miniKanren: Techniques, applications, and implementations*. PhD thesis, Indiana University, 2010.

[46] Matthew Rocklin. Logpy: Logic programming in python, 2013.

[47] Hongguang Fu, X Zhong, and Z Zeng. Automated and readable simplification of trigonometric expressions. *Mathematical and computer modelling*, pages 1–11, 2006.

[48] Joe Hendrix, Manuel Clavel, and J Meseguer. A sufficient completeness reasoning tool for partial specifications. *Term Rewriting and Applications*, (i):165–174, 2005.

[49] Y.K. Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[50] F Song and Jack Dongarra. A scalable framework for heterogeneous GPU-based clusters. *Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures*, (1), 2012.

[51] Mark F Tompkins. *Optimization Techniques for Task Allocation and Scheduling in Distributed Multi-Agent Operations*. Masters, Massachusetts Institute of Technology, 2003.

[52] Haluk Topcuoglu, Salim Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[53] Matthew Rocklin. Uncertainty Modeling with SymPy Stats. In *SciPy 2012*, Austin, TX, 2012.

[54] AG Glen, DL Evans, and LM Leemis. APPL: A probability programming language. *The American Statistician*, 55(2), 2001.

[55] Aaron Clauset, M E J Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 70(6 Pt 2):066111, December 2004.