Sage: Open Source Mathematical Software:
Symbolic Computation, Combinatorial Species, Backtracking Algorithms, and Distributed Computation

William Stein, Gary Furnish, Mike Hansen, Robert Miller, and Yi Qiang

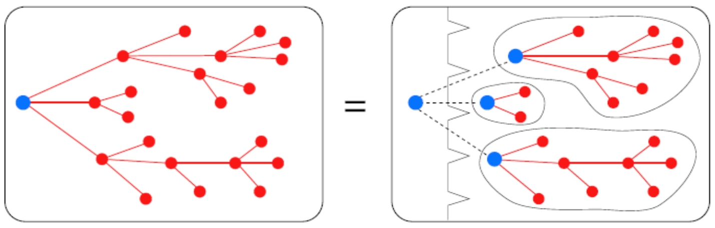
March 21, 2008

1 Introduction

Sage brings together Python and the best existing open source mathematical libraries and software to create a powerful alternative to commercial mathematics software without reinventing the wheel. This is a proposal for \$18,000 (=\$4,500 × 4) addressed to Chris DiBona at Google to fund work by four students on Sage. Each of the four students (two graduate and two undergraduate) have already contributed extensively to the Sage project, and are very well respected in the Sage community.

2 Combinatorial Species – Mike Hansen (UCSD Graduate Student in Mathematics)

Trees and tree-like structures play an important role in both computer science and in mathematics. One example of a commonly used tree is a binary search tree. As a simple example, we may consider rooted trees, which can be defined recursively as follows: a rooted tree consists of a root node attached to a (possibley empty) set of rooted trees. This is illustrated in the diagram below.



Typical questions that one might ask about rooted trees include how many labeled, rooted trees there are, how many labeled, unrooted trees there are, and what are all of the distinct unlabeled rooted trees. These questions often arise while considering the complexity analysis of algorithms.

The theory of *combinatorial species* provides an elegant algebraic framework for answering these questions and is well-suited for computer implementation. When translated into the language of species, the recursive definition of rooted trees X becomes

$$X = A \cdot E(X)$$

where A is the species of singletons, and E is the species of sets. From this one equation, one can efficiently count and generate all labeled and unlabeled rooted trees.

The goal of this project is to provide an implementation of combinatorial species in Sage. Once the species code is in place, one could run the following Sage session:

```
sage: X = CombinatorialSpecies()
sage: A = SingletonSpecies()
sage: E = SetSpecies()
sage: X.set( A*E(X) )
sage: X.count_labeled(range(1,10))
[1, 2, 9, 64, 625, 7776, 117649, 2097152, 43046721]
sage: X.count_unlabeled(range(1,10))
[1, 1, 2, 4, 9, 20, 48, 115, 286]
sage: l3 = X.list_labeled(3)
sage: u3 = X.list_unlabeled(3)
```

It would provide a backend to handle a wide range of new combinatorial objects with minimal effort. Additionally, the core of the module could be written in pure Python with no dependencies on Sage to allow it to be used in a standalone manner from other Python programs.

3 Rewrite and Vastly Optimize Symbolic Computation – Gary Furnish (Utah, Undergraduate in Physics)

Symbolic manipulations involving differentiation and integration are a key part of computational calculus. Sage currently uses Maxima, which is the best free symbolic computation package available. Maxima was created over thirty years ago in Lisp before the advent of object oriented programming, and also lacks useful documentation of its internals. Thus progress on open source symbolic manipulation has largely stagnated.

The primary reason that Maxima has not been replaced before now was the extensive work that has gone into it to perfect its ability to perform integration. However, compared to the modern capabilities of products such as Mathematica, it is lacking significant features. It is not uncommon for Maxima to be unable to perform relatively simple calculations due to flaws in the way it treats variables. The most commonly used features of Mathematica are its integration and differential equation solvers, and yet these are precisely where open source software is most lacking. Thus for the last twenty years those who have wanted to perform symbolic manipulations have been forced to pay over $2000 per seat, or $99 if they were a student. It is critical that this gap in open source mathematics software be closed if it is to be usable by most students, let alone professionals.

However, Sage does not just have a chance to equal commercial software for symbolic manipulations. Because it was written using object oriented Python, it is not limited to just numbers like Mathematica or Maple. By writing the code to consider general classes (in the Python sense) of objects, one may consider significantly more general forms of symbolic manipulation that are used extensively in physics. In general relativity, for instance, one often talks about space-time as a four dimensional object that exists in its own right. One commonly wants to find the curvature of space (in two dimensions, this would be the analogue of how far from flat a sheet is). Current methods all involve complicated, non-intuitive additions to commercial software, or the involve the creation of software explicitly for the task being solved. However, by having Sage consider Python classes instead of numbers, solving this problem becomes as easy as overriding a few function calls. In quantum physics one often considers particles, such as electrons, that can exist in two states (spin-up or spin-down). Problems that involve such particles are normally solved with specialized software, or worse, by hand. However, in this case the situation is even better. By leveraging existing code for groups in Sage, it will be possible to quickly write the code necessary to consider these problems. A new symbolics implementation using Python will not only benefit simple calculus calculations but has the potential to dramatically decrease the effort required to solve many physics problems.

4 Backtracking Algorithms and Permutation Groups – Robert Miller (UW Graduate Student in Mathematics)

Permutation groups provide a convenient representation for many abstract groups, and they arise as the symmetry groups of many combinatorial and geometric objects. Backtracking algorithms are a common technique, but less known are the techniques involving successive refinement of ordered partitions. This technique makes many computations involving permutation groups feasible. It has been employed in the classification of graphs and error-correcting codes, and in the searches for balanced incomplete block designs on certain parameters. Recently, the classification of certain matrix algebras has reached the point where these techniques could be put to good use. They can also be employed to exhaustively generate isomorphism class representatives of a wide class of mathematical objects.

The concept of base and strong generating set was originally developed by Sims (1971) to prune the tree searched during backtrack. The degree of a permutation group, i.e. the number of points it permutes, is often restrictively large, and computing a base and strong generating set can greatly reduce this restriction. For example, the Janko sporadic simple groups in their representations of degrees 276, 100, and 6156 have bases of degree 3, 4 and 3 points respectively. The action of a permutation group element is determined by its action on the base, so a reduction like this makes it much more feasible to do computations in the group.

This notion is an implicit consequence of the work of McKay (1978), which develops the idea of refinements of ordered partitions. This allows for more extensive pruning of the search tree, resulting in a highly efficient algorithm for, in McKay's case, computing the automorphism group of a graph, and producing a canonical representative for its isomorphism class. Unfortunately, the software written by McKay to implement these ideas was never released under a sufficiently open license, and those that have examined the source code have reported that it is very difficult to understand. This was the main motivation for my implementation of the algorithm (2007) in Sage, which was released under the GPL, and is designed to be much more understandable at a source code level.

Leon observed (1991) that these techniques could be substantially generalized to many general permutation group questions. These questions include the graph-isomorphism complete questions of computing set stabilizers, centralizers and normalizers of elements, upper central series and intersections of groups. He very recently released the software he wrote to implement these computations under GPL. During his development of these programs, in his own words, he "didn't always remember to change the comments accordingly, so in some places comments appropriate to one algorithm appear in the code for another." Further, the code written was written strictly for use in a standalone application that terminates on finish, so memory management was essentially abandoned halfway through development. Finally, again in his own words, by the time the code was signed over to the Magma group, "the partition backtrack code had been patched up many times by then, and I felt it needed to be reorganized and rewritten from scratch. I started to do this on several occasions, but never had the time to complete more than a small fraction of the task."

The algorithms for graphs and binary codes have already been implemented in Sage, and recently the last known bugs in the programs were eliminated. The binary code case has tested faster than the analogous code in Magma, which shows promise for the future work to be done. The goals for this project are to generalize the algorithms to at least the level of generality of Leon's work, to be well documented and understandable at the source code level, and to revive the literature and code of partition backtrack algorithm to a modern setting. As in the case of symbolic calculus, the object oriented, *mathematically* oriented setting of Sage is optimal for implementing a general partition backtrack algorithm, so that even questions for which this may be an effective technique of which we are not yet aware can be eventually implemented efficiently.

5 Distributed Computing with DSage – Yi Qiang (UW Undergraduate in Mathematics)

Distributed computing is a method of computer processing in which different parts of a program run simultaneously on two or more computers that are communicating with each other over a network. Google is certainly no stranger to distributed computing and probably understands the power and utility of it better than most. For Sage to be a viable competitor to commercial offerings such as *gridMathematica* and Matlab's *Parallel Computing Toolbox*, we need to provide an easy to use, innovative and robust alternative.

Today, the distributed computing options that are available for mathematicians to use are either overly complex or are not integrated with any mathematical software. While commercial solutions such as *gridMathemat-*

ica and Matlab's *Parallel Computing Toolbox* exist, they are closed systems and often times are prohibitively expensive for the average user. Because DSage is written in Python and is distributed with Sage, it greatly reduces the barrier of entry for mathematicians who want to use distributed computing. Furthermore, it works just as well on a multi-core computer as it does on a cluster, and thus provides one way to overcome the limitations of Python's Global Interpreter Lock.

While there is not a lack of problems for which DSage is the right choice, namely those that can take advantage of coarse grained distributed computing, there are two main concerns which are not currently adaquately answered by DSage:

1. How does one easily split up the problem into discrete chunks?
2. How can the processing power of the limited computing resources available best be managed?

To effectively replace specialized systems currently used in parallel processing, DSage must better address these question. First, DSage needs to provide better documentation and examples that enmulate those found in the real world. For common scenerios, someone, someone should be able to copy/paste existing example code and make minor modifications to adapt it for usage. For more specialized scenerios, the examples may not be directly related to the situation but should provide enough insight into how to use DSage such that the person can easily write their own solution. Secondly, DSage must be made easier to deploy on ad-hoc cluster of computers which the person has access to. Possible scenarios such as using a departments computer lab as a cluster overnight should be made possible and easy to implement.

To achieve these goals, there are several specific areas where DSage needs improvement:

1. Make DSage more robust by improving code coverage and doctests
2. Improve the web interface to DSage
3. Provide real world examples on how to use DSage
4. Develop and document deployment strategies for DSage workers which include, among other features, automatic updating.

Each of these work items is achievable over the course of the summer, and their completion is crucial in ensuring the success of DSage and Sage as a viable alternative to specialized codebases and expensive commerical offerings