row vectors of the product of the matrices. In lines 28–31, the two arguments are migrated back to the current thread's thread-local region; the `AdoptObj()` calls effectively undo the `Shareobj()` calls in lines 19–20.

The actual multiplication operation for each row vector takes place in the `ParMatrixMultiplyRow` function in lines 1–15. This is the basic matrix multiplication algorithm for a given row vector. The only difference compared to the sequential version is the `atomic` statement in lines 4–13, which gives the task temporary shared ownership of the matrix regions. Note that this is necessary even though the main thread had already done the same: the tasks have no knowledge of what the main thread is currently doing and thus have to also claim shared ownership on their own. Because all tasks require only read access to the matrices, shared ownership is sufficient and all tasks can execute in parallel (as long as the hardware permits).

Conclusion

A public beta release of HPC-GAP is planned for the near future. If you would like access to the current pre-release version, please contact the author or the GAP Group.

References

[1] GAP – Groups, Algorithms, and Programming, Version 4.7.5; 2014 `http://www.gap-system.org`

[2] Hansen, P. B. Java's Insecure Parallelism. *SIGPLAN Notices*, v.34 n.4, p.38–45, April 1999.

[3] Friedman, D. P. and Wise, D. S. The Impact of Applicative Programming on Multiprocessing. *1976 International Conference on Parallel Processing*, p.263–272.

The GAP package `SingularInterface`

M. Barakat, M. Horn, F. Lübeck, O. Motsak, M. Neunhöffer, H. Schönemann
(TU Kaiserslautern, JLU Gießen, RWTH Aachen University, TU Kaiserslautern, triAGENS GmbH, TU Kaiserslautern)

`barakat@mathematik.uni-kl.de, max.horn@math.uni-giessen.de, frank.luebeck@math.rwth-aachen.de, motsak@mathematik.uni-kl.de, max@9hoeffer.de, hannes@mathematik.uni-kl.de`

What is `SingularInterface`?

The GAP package `SingularInterface` is a highly efficient and robust unidirectional low-level interface to SINGULAR [2, 3]. It is the outcome of an intensive collaboration between core developers of both systems.

The goal of this interface is to map all of SINGULAR's powerful functionality into GAP. To achieve this it automatically wraps *all* SINGULAR datatypes and exports *all* of SINGULAR's interface procedures to GAP.[1] Furthermore, all procedures of any contributed library can be loaded on demand.[2]

This package is a rather "faithful" image of SINGULAR; it does not make an extensive attempt for a better integration of SINGULAR into the GAP ecosystem. This is intentionally left to other packages, which are free to realize this in different ways.

The development of `SingularInterface` has reached a β-phase and is already actively used in some research projects. We hope to attract more users in the near future, whose feedback will be crucial for a successful further development.

How to get it?

To download and install `SingularInterface` please follow the instructions on

```
http://gap-system.github.io/
      SingularInterface/
```

If you are reading this article, say, more than one year in the future, and have a recent GAP installation, then hopefully you already have a working version of this package.

To check that the package has been successfully installed, start GAP and type:

```
gap> LoadPackage( "SingularInterface" );
true
```

To see all imported procedures type:

```
gap> SI_<press TAB twice>
```

The SINGULAR library "`standard.lib`" is loaded by default. To see all imported SINGULAR library procedures type:

```
gap> SIL_<press TAB twice>
```

To load any other library, e.g. "`matrix.lib`", type:

```
gap> SI_LIB( "matrix.lib" );
true
```

[1] With the prefix "`SI_`" prepended to their names.

[2] They appear in GAP with the prefix "`SIL_`" prepended to their names.

Goals

The motivation behind developing `SingularInterface` is the increasing interest of various research projects in combining the strength of both systems: GAP users get access to SINGULAR's polynomial arithmetic and highly optimized Gröbner basis engine. SINGULAR users gain a second front end language for this engine – in addition to the current SINGULAR language – with an advanced object model primarily designed for modeling higher mathematical structures, as well as access to GAP as an expert system for group and representation theory.

An example

Here is a short example in SINGULAR 4.0.1 demonstrating some basic procedures. On the left you see the SINGULAR code, on the right the corresponding GAP code. SINGULAR uses "=" for assignments and suppresses any output while GAP uses "`:=`" for assignments and triggers the so-called `View`-method, which gives a very brief description of the object (unless suppressed by a trailing "`;;`"). Basically, SINGULAR's `print` procedure is mapped to the so-called `Display`-method in GAP. The current version of `SingularInterface` is `2014.09.23`.

Start by loading `SingularInterface` in GAP.

```
> // standard.lib is automatically preloaded
> // this example needs no further libraries
```

```
gap> LoadPackage( "SingularInterface" );
true
```

Define the ring $R := \mathbb{Q}[x_0, x_1, x_2, x_3]$ (with the monomial ordering `degrevlex`):

```
> ring R=0,(x0,x1,x2,x3),dp;

> short=0;

> option(redTail);
```

```
gap> R := SI_ring( 0, "x0..3", [["dp",4]] );
<singular ring, 4 indeterminates>
gap> ## short=0 is the default, disable by:
gap> ## Singular( "short=1" );
gap> SI_option( "redTail" );
true
```

Define the polynomial $(x_1 + x_3)^2$:

```
> poly p=(x1+x3)^2; p;
x1^2+2*x1*x3+x3^2
```

```
gap> AssignGeneratorVariables( R );
#I  Assigned the global variables
#I  [ x0, x1, x2, x3 ]
gap> p := (x1+x3)^2;
x1^2+2*x1*x3+x3^2
gap> IsSingularPoly( p );
true
```

Define the ideal $I := \langle x_0^2 - x_1x_3, x_0x_1 - x_2x_3 \rangle \lhd R$:

```
> ideal I=x0^2-x1*x3,x0*x1-x2*x3;

> print(I);
x0^2-x1*x3,
x0*x1-x2*x3
```

```
gap> I := SI_ideal([x0^2-x1*x3, x0*x1-x2*x3]);
<singular ideal, 2 gens>
gap> Display( I );
x0^2-x1*x3,
x0*x1-x2*x3
```

The corresponding matrix i:

```
> def i=matrix(I);

> print(i);
x0^2-x1*x3,x0*x1-x2*x3
```

```
gap> i := SI_matrix( I );
<singular matrix, 1x2>
gap> Display( i );
x0^2-x1*x3,x0*x1-x2*x3
```

The sum $I + I$ means the sum of ideals:

```
> J=I+I;

> print(J);
x0^2-x1*x3,
x0*x1-x2*x3
```

```
gap> J := I + I;
<singular ideal, 2 gens>
gap> Display( J );
x0^2-x1*x3,
x0*x1-x2*x3
```

Whereas $i + i$ means the sum of matrices:

```
> print(i+i);
2*x0^2-2*x1*x3,2*x0*x1-2*x2*x3
```

```
gap> Display( i + i );
2*x0^2-2*x1*x3,2*x0*x1-2*x2*x3
```

The squared ideal $I^2 \lhd R$:

```
> def I2=I^2;

> print(I2);
x0^4-2*x0^2*x1*x3+x1^2*x3^2,
x0^3*x1-x0*x1^2*x3-x0^2*x2*x3+x1*x2*x3^2,
x0^2*x1^2-2*x0*x1*x2*x3+x2^2*x3^2
```

```
gap> I2 := I^2;
<singular ideal, 3 gens>
gap> Display( I2 );
x0^4-2*x0^2*x1*x3+x1^2*x3^2,
x0^3*x1-x0*x1^2*x3-x0^2*x2*x3+x1*x2*x3^2,
x0^2*x1^2-2*x0*x1*x2*x3+x2^2*x3^2
```

The Gröbner basis of the ideal I is returned as a new different (but mathematically equal) ideal G:

```
> def G=std(I);

> print(G);
x0*x1-x2*x3
x0^2-x1*x3
x1^2*x3-x0*x2*x3
```

```
gap> G := SI_std( I );
<singular ideal, 3 gens>
gap> Display( G );
x0*x1-x2*x3,
x0^2-x1*x3,
x1^2*x3-x0*x2*x3
```

The syzygies of the generators of G are the columns of the SINGULAR datatype `module`[3]:

```
> def S=syz(G); S;
S[1]=x0*gen(1)-x1*gen(2)-gen(3)
S[2]=x1*x3*gen(1)-x2*x3*gen(2)-x0*gen(3)
> print(S);
x0, x1*x3,
-x1,-x2*x3,
-1, -x0
```

```
gap> S := SI_syz( G );
<singular module, 2 vectors in free module of
 rank 3>
gap> Display( S );
x0, x1*x3,
-x1,-x2*x3,
-1, -x0
```

To access the second column of `S` use:

```
> S[2];
x1*x3*gen(1)-x2*x3*gen(2)-x0*gen(3)
> print(S[2]);
[x1*x3,-x2*x3,-x0]
```

```
gap> S[2];
<singular vector, 3 entries>
gap> Display( S[2] );
[x1*x3,-x2*x3,-x0]
```

To access the first entry of the second column of `S` use:

```
> S[2][1];
x1*x3
> p-S[2][1];
x1^2+x1*x3+x3^2
```

```
gap> S[2][1];
x1*x3
gap> p - S[2][1];
x1^2+x1*x3+x3^2
```

To create a matrix use:

```
> matrix m[3][2]=x0,x3, x1,x2, x3,x0;

> print(m);
x0,x3,
x1,x2,
x3,x0
```

```
gap> m := SI_matrix(R,3,2,"x0,x3,x1,x2,x3,x0");
<singular matrix, 3x2>
gap> Display( m );
x0,x3,
x1,x2,
x3,x0
```

To extract the (2,1)-entry from the matrix use:

```
> m[2,1];
x1
```

```
gap> m[[2,1]];
x1
```

The sum of the `module S` and the `matrix m` is their augmentation:

```


> print(S+m);
x0, x1*x3, x0,x3,
-x1,-x2*x3,x1,x2,
-1, -x0,   x3,x0
```

```
gap> S + m;
<singular module, 4 vectors in free module of
 rank 3>
gap> Display( S + m );
x0, x1*x3, x0,x3,
-x1,-x2*x3,x1,x2,
-1, -x0,   x3,x0
```

The development

How does it work?
`SingularInterface` aims to be a comprehensive bridge between GAP and SINGULAR with as little overhead as possible, both in terms of speed and memory. To achieve this goal, some key design decisions were crucial:

(1) Avoid converting data between the two systems, as conversions are expensive.

(2) Automate generating function bindings as much as possible.

(3) We stay relatively low-level with the interface, mostly refraining from trying to change SINGULAR behaviour to be more GAP like (with one exception, see below).

Regarding (1), on the GAP side we use "wrapper objects" around SINGULAR objects. That is, tiny GAP objects which essentially consist of a pointer to the actual SINGULAR object (such as a polynomial) plus some meta information (types, attributes). With the exceptions of small integers and strings, we never automatically convert SINGULAR objects into "native" GAP

[3]The datatype `module` in SINGULAR 4.0.1 is in first approximation a specialized sparse data structure for column oriented matrices with compressed columns, where each column has the datatype `vector`. For more details see the `SingularInterface` manual [1].

objects. Indeed, in most cases that would be pointless, as for further operations with the object, we would have to convert it back into a SINGULAR object anyway. So when you have a SINGULAR polynomial p and call SI_deg(p), in the background, SingularInterface extracts the pointer to the SINGULAR object from it, then invokes the SINGULAR interpreter C code for deg, and returns the result to the user (since the result is a small integer, it is not wrapped).

Of course when necessary, you can still convert from and to "native" GAP objects (although this is one of the areas where there is still work to be done, in order to cover all possible SINGULAR coefficient ring types).

A major complicating factor for this approach is that GAP and SINGULAR use very different memory management systems (GAP uses a so-called "generational moving garbage collector", in which objects change their position in memory over time, while SINGULAR use a more traditional "malloc" system plus reference counting, and expects objects to stay at a fixed position in memory). With some clever tricks, aided by a few small but helpful changes on the SINGULAR side, this now works extremely well.

Regarding (2), here is one example: From the data structures the SINGULAR interpreter uses to lookup function names (such as transpose), the build system of SingularInterface automatically generates bindings for all SINGULAR functions in GAP (SI_transpose). Thus when the SINGULAR team adds a new function, SingularInterface automatically supports it (after recompiling). This also covers SINGULAR library functions. Finally, in addition to interpreter and library functions, we also provide direct access to select SINGULAR C++ kernel functions such as p_Mult_mm (with _SI_ prepended) which can be used by experts to further optimize their code. This, too, is automatically generated and can thus easily be extended to expose more functionality, if requested.

Regarding (3), we mostly expose the SINGULAR interface faithfully and with no changes (as opposed to trying to make it "more GAP like", or trying to fix perceived design flaws etc.). We plan to eventually add a high-level layer built atop the existing interface which changes some of this; however, this may well be in a separate package. But providing this raw access has multiple advantages: It allows others to build alternative high-level front ends (as everybody will have a different idea about how to do that), and it also frees us from making complicated decisions on what is "right" and "wrong". Finally, it has the added benefit that the SINGULAR manual can be used as a reference manual for SingularInterface. There is one major exception: We try to hide the SINGULAR concept of a "global" or "active" ring from users as much as possible. In SINGULAR, the result of a command like var(1) implicitly depends on the "active" ring. Working with multiple rings thus becomes rather tedious. To avoid this, in SingularInterface, each GAP wrapper object for "ring dependent" SINGULAR objects (such as polynomials, ideals, modules, but not integers or rings themselves) carries a reference to the ring it belongs to. The user does not need to worry about "active" rings at all. As a side effect, a few SINGULAR functions need to be called with one additional parameter, namely the ring they refer to. For example, var(1) becomes SI_var(r,1), where r is the (now explicit) SINGULAR ring we refer to.

Note that all of this bypasses the SINGULAR language. However, to guarantee complete access to SINGULAR, one can still send arbitrary commands to the SINGULAR interpreter as a string passed to the function Singular(). For details, we refer the reader to the SingularInterface manual (which is still work in progress).

Activity

The development of SingularInterface started in May 2011. The project was generously supported by the University of St Andrews, the University of Kaiserslautern, and DFG priority program SPP-1489. The C/C++ code was contributed by Max Horn, Frank Lübeck, and Max Neunhöffer. To keep the interface slick and efficient Hans Schönemann and Oleksandr Motsak made several changes and improvements on the side of SINGULAR. The homalg project [4] heavily uses SINGULAR through its own IO-based interface. It was thus a natural candidate which helped to test and stabilize some parts of SingularInterface through its extensive test suite.

Outlook

While SingularInterface can already be used (and is used) for research work, there is still quite a lot to be done before we can call it a complete product. We need to add some more functionality, expand the manual, and add many test cases. Maybe the most urgent is to make convenient the construction of all types of rings supported by SINGULAR.[4]

For this, we would appreciate your help. For example, if you experience any issue with SingularInterface, please report it using our issue tracker at GitHub[5]. The same holds if you feel that some functionality is missing or not as easy to access as it should be. We cannot guarantee to fulfil every wish, but it helps us to prioritize our efforts.

Beyond this, we also welcome contributions to the code, the manual or the test suite. Ideally in the form of pull requests[6].

A future challenge would be to port SingularInterface to HPC-GAP[7] once the future multithreaded SINGULAR is available.

[4]At the moment, they are accessible via the function Singular().

[5]https://github.com/gap-system/SingularInterface/issues

[6]https://github.com/gap-system/SingularInterface/pulls

[7]HPC-GAP stands for "High Performance Computing GAP" and adds parallelization support to GAP(cf. article of R. Behrends, p. 27 in this issue of the CAR).

References

[1] M. Barakat, M. Horn, F. Lübeck, O. Motsak, M. Neunhöffer, H. Schönemann, *Singular-Interface – A* GAP *interface to* SINGULAR, (http://gap-system.github.io/SingularInterface/), 2011–2014.

[2] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann, SINGULAR *3-1-6 — A computer algebra system for polynomial computations*, (http://www.singular.uni-kl.de), 2013.

[3] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.7.5*, (http://www.gap-system.org), 2014.

[4] The homalg project authors, *The* homalg *project – Algorithmic Homological Algebra*, (http://homalg.math.rwth-aachen.de/), 2003–2014.

Berichte über Arbeitsgruppen

Arbeitsgruppe Algebra und diskrete Mathematik an der Universität Osnabrück

Winfried Bruns (Osnabrück)

Die Arbeitsgruppe Algebra und diskrete Mathematik des Instituts für Mathematik der Universität Osnabrück besteht zurzeit aus vier Professoren (H. Brenner, W. Bruns, T. Römer, H. Spindler), vier Postdocs und sieben Doktoranden. Die Professur von Bruns, der am 30.09.2014 in den Ruhestand tritt, wird ab April 2015 durch eine Juniorprofessur (M. Juhnke-Kubitzke) ersetzt.

Die Arbeitsgruppe war 2009–2012 Bestandteil des universitätseigenen Graduiertenkollegs „Kombinatorische Strukturen in Algebra und Topologie“ und ist seit Oktober 2013 am Osnabrücker DFG-GRK „Kombinatorische Methoden in der Geometrie“ beteiligt. Das seit 1997 bestehende Normaliz-Projekt wird aktuell durch das DFG-SPP „Experimentelle Methoden in Algebra, Geometrie und Zahlentheorie“ gefördert.

Zu den langjährigen Kooperationspartnern gehören Mathematiker in Ann Arbor, Bukarest, Essen, Genua, Hanoi, Lexington, San Francisco, Salt Lake City, Sheffield und Tokio.

Das zentrale Forschungsgebiet der Arbeitsgruppe ist die kommutative Algebra unter Einbeziehung der algebraischen Geometrie, algebraische Aspekte der Kombinatorik und der Darstellungstheorie. Über viele Jahre sind Beiträge zu determinantiellen Ringen und Idealen, affinen Monoiden und Monoid-Algebren, homologischen Invarianten graduierter Ringe, Hilbert-Kunz-Theorie, Tight closure, Grothendieck-Topologien, tropischer Geometrie und Arrangements von Hyperebenen hervorgegangen.

Die gegenwärtig und in den letzten Jahren bearbeiteten Themen sind insbesondere die (Nicht-) Lokalisierung von Tight Closure, Irrationalität von Hilbert-Kunz-Multiplizitäten, arithmetische und geometrische Deformationen von Vektorbündeln, symmetrische Asymptotik von Moduln, Gröbner-Basen und Initial-Ideale bzw. -Algebren determinantieller Ideale bzw. Algebren, Relationen von Minoren, Stanley- und Hilbert-Zerlegungen, algebraische Aspekte von Hyperebenen-Arrangements, generische tropische Varietäten.

Für die Computeralgebra ist das Projekt Normaliz sicherlich am interessantesten. Für ein normales affines Monoid berechnet Normaliz die Hilbert-Basis, die Hilbert-Reihe zu einer $\mathbb{Z}$-Graduierung sowie zusätzliche Daten, die bei der Berechnung von Hilbert-Basen und Hilbert-Reihen notwendig sind oder sich nebenbei ergeben. Dazu gehören insbesondere Triangulierungen und Stanley-Zerlegungen.

Ein normales affines Monoid entsteht als Durchschnitt eines rationalen Kegels mit einem Gitter. Deshalb ist es nichts anderes als die Lösungsmenge eines homogenen Systems linearer diophantischer Ungleichungen, Gleichungen und Kongruenzen. Die Hilbert-Basis ist das eindeutig bestimmte minimale Erzeugendensystem (bezüglich der Addition).

Seit Version 2.11 berechnet Normaliz aber auch Lösungen von entsprechenden inhomogenen Systemen, also Gitterpunkte in rationalen Polyedern. Die Erweiterung NmzIntegrate berechnet Hilbert-Reihen zu polynomialen Gewichtsfunktionen und Integrale von Polynomen über Polytope.

Normaliz ist in C++ geschrieben, nutzt GMP für Großzahl-Arithmetik und OpenMP für Parallelisierung. Es steht als ausführbares Programm für die gängigen Betriebssysteme zur Verfügung und ist über sein Bibliotheks-Interface in CoCoA, polymake, GAP, Regina und als optionales Paket in Sage eingebunden. Außerdem gibt es Interfaces zu Macaulay 2 und Singular.