KASH: Recent Developments

Sebastian Freundt[1], Aneesh Karve[2], Anita Krahmann[1], and Sebastian Pauli[1]

[1] Institut für Mathematik, MA 8–1, Technische Universität Berlin
Straße des 17. Juni 136, 10623 Berlin, Germany
{freundt,krahmann,pauli}@math.tu-berlin.de
[2] Computer Sciences Department, University of Wisconsin - Madison
1210 W. Dayton St., Madison, WI 53706-1685, USA
karve@cs.wisc.edu

Abstract. In recent years the computer algebra system KASH/KANT for number theory has evolved considerably. We present its new features and introduce the related components, QaoS (Querying Algebraic Objects System) and GiANT (Graphical Algebraic Number Theory).

1 Introduction

KASH/KANT is a computer algebra system specialized for algebraic number theory and its applications. KANT stands for "Computational Algebraic Number Theory" with a slight hint of its German origin. It contains the following system components:

- The KANT C library with highly specialized algorithms for number theory
- KASH, the KANT shell, and its programming language
- The graphical user interface GiANT
- The QaoS databases for algebraic objects with access via the worldwide web, or from computer algebra systems

The KANT library for number theory has been in development since 1987 under the leadership of Michael Pohst. Development began in Düsseldorf, and since 1993, continues at Technische Universität Berlin. The following describes the evolution of the components of KASH/KANT.

1987 KANT V1 (Fortran library)
1992 KANT V2 (C library) built on the Cayley platform
1994 KANT V4 (C library) built on the Magma platform
1995 KASH 1.0 (KANT shell) based on GAP 3
1996 KASH 1.6, Database for number fields
1999 KASH 2.1
2004 KASH 2.4, WWW Database
2005 KASH 3, GiANT (GUI for KASH 2.x), QaoS Databases

The KANT V4 C library, built on the Magma [BCP97] platform, provides the core functionality of KASH. (KANT V4 functionality is also available in Magma.)

The current version uses the GNU multi-precision library [GMP] for long integer arithmetic, and [MPFR] for arbitrary precision real numbers. KASH consists of a modified version of GAP 3, which is linked to the KANT V4 library, a variety of utility functions to provide an interface to GAP 3, and numerous functions written directly in the native KASH shell language, that make up the KASH library. Some of the GAP 3 library functions are also available in KASH. An overview of KASH functionality is found in section 2.

In recent years we have redesigned key components of KASH. Our main purpose was to create a system that was easier to maintain and extend. To this end, we added a number of object-oriented features to KASH 3, although we were limited since a ground-up rewrite was not feasible. With these new features, users can now write *generic functions* which can be used, for example, with global fields *and* number fields alike. To support generic functions we have introduced identifier overloading for function names (see section 5) and a new type system, which also allows for *user-defined types* (section 3).

In KASH 3, documentation (section 4) is written directly into the source code making documenting new functions simpler, and general maintenance easier. Similarly, the online help system and downloadable manuals are automatically generated from in-line comments in the source code.

In section 6 we introduce the redesigned and expanded KANT database for number fields (now called QaoS – Querying Algebraic Objects System).

While most algebra is done by writing text and formulas, diagrams have always been used to present structural information clearly and concisely. Text shells are the *de facto* interface for computational algebraic number theory, but they are incapable of presenting structural information graphically. In section 7 we present GiANT, a newly developed graphical interface for working with number fields. GiANT offers interactive diagrams, drag-and-drop functionality, and typeset formulas.

KASH and Other Computer Algebra Systems As already mentioned, KASH/KANT shares parts of its C library with Magma. KASH also integrates easily with other computer algebra systems. The package Alnuth [AED05] allows access to the functionality of KASH 2.x from GAP 4. KASH 3 can be accessed through the Python based computer algebra system SAGE [St06].

The SCIEnce (Symbolic Computation Infrastructure for Europe) project, funded by the European Union, aims to link the computer algebra systems GAP 4 [GAP], KASH/KANT, MuPad [Mu05], and Maple [Ma05] in the context of Web and Grid services. In this scheme, each of these systems will run as a server or a client. The project will answer many questions about access/transport for complex objects. In particular the project team will develop an OpenMath [OM] XML format for the algebraic objects used by these systems.

Availability KASH 3 is freely available. Binaries for Linux/x86, Mac OS X, and MS Windows can be downloaded from

`http://www.math.tu-berlin.de/~kant/kash.html`.

In addition KASH 2.x was ported to various UNIX versions (AIX, IRIX, OSF/1, Solaris).

2 Functionality

In the 1970s, Hans Zassenhaus postulated four principal tasks for computational algebraic number theory. Namely, the development of efficient algorithms for the computation of the maximal order, the unit group, the class group, and Galois group of algebraic number fields. We now have these algorithms, not only for number fields but also for global functions fields, and they are fast enough to make the computation of more complex objects possible, such as class fields.

KASH offers powerful functions for working with number fields, function fields, and local fields and for solving Diophantine equations. The only comparable systems in this regard are Magma [BCP97], which shares parts of its code with KASH, Pari-GP [BB$^+$05], and SAGE [St06], which contains Pari.

Number Fields

KASH allows base arithmetic and integral basis computation for extensions of the rationals and for relative extensions of number fields. Efficient algorithms for class groups and unit groups are applied in the computation of ray class groups [HPP03], which in turn enable the computation of class fields [Fi02]. As a further generalization of class groups, Picard groups of arbitrary orders can be computed. The computation of Galois groups is now possible for number fields of degree up to 23 [Ge05].

Example We determine a ray class field over a number field.

```
kash% K := NumberField(X^3+6*X+3);
Number Field with defining polynomial X^3+6*X+3 over Q
kash% O := MaximalOrder(K);
Maximal Equation Order with defining polynomial X^3+6*X+3 over Z
```

Next we pick the index 3 subgroup of the ray class group with conductor (3). The composed types such as `grp^abl` are explained in section 3.

```
kash% r := RayClassGroup(3*O);
Abelian Group isomorphic to Z/6, extended by:
  ext1 := Mapping from: grp^abl: r to ids/ord^num: _AF
kash% q := Quotient(r,3*r.1);
Abelian Group isomorphic to Z/3, extended by:
  ext1 := Mapping from: grp^abl: r to grp^abl: q
```

The corresponding ray class field is given as an abstract extension. A representation is computed if and only if the user requests it.

```
kash% L := RayClassField(Inverse(q.ext1)*r.ext1);
Abelian Field, defined by (<[3, 0, 0]>, []) of structure: Z/3
kash% EquationOrder(L);
Equation Order with defining polynomial X^3-3*X-1 over o
```

Function Fields

Numerous algorithms for number fields have been generalized to global fields. For example, maximal order algorithms have been generalized to algorithms for the computation of finite and infinite maximal orders. Many tasks in the theory of function fields rely on the computation of Riemann-Roch spaces, which are applied to the computation of divisor class groups [He02]. Galois groups for function fields up to degree 23 can also be determined [Ge05].

Future work will provide increased support for elliptic curves, automorphism groups of function fields or curves, and isogeny and isomorphy computations. Functions for computing endomorphism rings of hyper elliptic curves and Deuring correspondences are also planned, as well as support for function fields over the complex numbers.

Example We generate a function field `L` whose constant field is a function field `K`, and compute its finite maximal order.

```
kash% k := GF(7);; kx := FunctionField(k);;
kash% kxy := PolynomialAlgebra(kx);
Polynomial Ring over rational function field over GF(7)
kash% K := FunctionField(kxy.1^2+6*kx.1^7+2*kx.1^4+6*kx.1+3);
Algebraic function field defined over Univariate rational function
field over GF(7) by kxy.1^2 + 6*kx.1^7 + 2*kx.1^4 + 6*kx.1 + 3
kash% L := ConstantFieldExtension(K,K);;O := MaximalOrderFinite(L);
Maximal Equation Order of L over Polynomial Ring over K
```

We generate a non-trivial Deuring correspondence of the function field `K` represented by the ideal `V`. We determine its Riemann-Roch space, which then could be applied to show that the correspondence is not trivial.

```
kash% BindNames_(CoefficientRing(O),["X"]);
kash% V := Ideal(O, [L.1-K.1,X^2+kx.1^2-2*kx.1*X-2*X-2*kx.1+1]);
Ideal of O, Basis: [X^2 +(5*kx.1+5)*X+kx.1^2+5*kx.1+1  0] [6*K.1  1]
kash% M := RiemannRochSpace(2*Divisor(V));
KModule M of dimension 2 over K, extended by:
  ext1 := Mapping from: mdl/fld: M to fld^fun: L
```

Local Fields

KASH supports p-adic fields and their extensions as well as fields of Laurent series. Polynomials over local fields can also be factored; [Pa01]. Applications

include the computation of integral bases, ideal decomposition over global fields, and completions of global fields. Support for computing unit groups and class fields over p-adic fields is scheduled for the next KASH release.

Example We use an Eisenstein polynomial to generate a ramified extension over $\mathbb{Q}_3$. It is the completion of the number field `K`, from the number field example above, with respect to the ideal over 3.

```
kash% Q3 := pAdicField(3,10);
3-adic field mod 3^10
kash% V := TotallyRamifiedExtension(Q3,X^3+6*X+3);
Totally ramified extension defined by X^3+6*X+3 over Q3
```

We factor the generating polynomial of the class field `L/K` from the number field example over the field `V`.

```
kash% S := PolynomialRing(V);
Univariate Polynomial Ring over V
kash% Factorization(S.1^3-3*S.1-1,rec(Certificates:=TRUE));
[<S.1^3+((4*V.1^2-2*V.1 + 25)*V.1^3+O(V.1^30))*S.1-1+O(V.1^30),1>]
extended by certificates := [rec(F := 1, Rho := 1, E := 3,
Pi := (V.1^-1+O(F.1^29))*S.1+(V.1+2)*V.1^-1+O(F.1^29))]
```

Note that `V.1` denotes the uniformizer of the field `V` and that `S.1` is the indeterminate of the polynomial ring `S`. The polynomial $\texttt{S.1}^3 - 3 \cdot \texttt{S.1} - 1$ is irreducible, as expected. The certificates tell us that it generates a totally ramified extension of degree `E`=3 over `V`. Thus the ideal over 3 is totally ramified in `L` with ramification index 9.

Diophantine Equations

KASH contains efficient functions for solving absolute and relative norm equations [Fi97], Thue equations [BH96], and unit and index form equations [Wi00].

Example We solve the Thue equation $X^3 + X^2Y - 2XY^2 - Y^3 = 7$.

```
kash% T := Thue(X^3+X^2-2*X-1);
Thue object with form:  X^3 + X^2 Y - 2 X Y^2 - Y^3
kash% Solutions(T,7);
[ [ -3, 2 ], [ 1, -3 ], [ 2, 1 ] ]
```

3 Type System

The type system of KASH 2.x was simple. When asked for the type of an object, KASH returned a string containing a description of the type. Most functions contained type information for their main argument and return value. Type checking was typically done using predicates, like `IsOrder()`. For KASH 3 we wanted to create a type system that met the following criteria:

– Represent mathematical categories in an intuitive way
– Allow efficient comparison of types (to facilitate overloading, inheritance)
– Allow the creation of user-defined types

That said, we needed to maintain backwards compatibility with the type system provided by the underlying KANT C libraries. We also needed to address the existence of data structures unique to the GAP language, such as lists. In creating the KASH 3 type system we studied the GAP and MAGMA type systems.

In the transition from GAP 3 [Sc+93] to GAP 4 [GAP], the overloading of functions based on the predicate-driven type system was introduced. A function is installed as a method with a list of predicates that are required for the function to be called. This generalization of the predicate driven approach would have been difficult to combine with the types in the KANT C library.

Magma [BCP97] has a categorical type system. Types are internally organized in a category inclusion graph. A type is said to be related to another type if there is a path of inclusions from the one type to the other. Types in function signatures are matched using this relation. Recently, extended types have been introduced that make it possible, for instance, to distinguish the types of polynomial algebras over various rings. Magma does not support user-defined types.

Taking all this into consideration, we designed a new type system for KASH 3. Three kinds of types exist: simple types, typed aggregates, and composed types.

Simple types include basic types like `char` and `type`, or more specialized types such as `thue` for Thue equations. Simple types also include data structures (aggregates) such as `list`, `dry` (lists with no duplicate entries), `set` (sorted lists with no duplicate entries), `record` , and `alist` (associative lists).

Typed aggregates extend aggregates to include one or more additional type specifications. They are similar to parameterized types found in modern programming languages. The syntax for typed aggregates is *aggtype*`(`[`,`*type*]`)`. Typed aggregates can be created from aggregates: sequences (`seq()`), maps (`map()`), tuples (`tup()`), and `nof()`. The latter stands for "n of". It can be used in defining functions with a variable number of arguments. For example, a function that takes an arbitrary number of string arguments would have argument type `nof(string)`.

Composed types represent complex mathematical objects. Composed types are composed of atoms. The first atom is either `elt-` or `str-`, which stand for element and structure respectively (elements are contained in structures). The first atom is followed by a *structure* atom, a caret (`^`), and a *specifier* atom. To take an example, `elt-fld^num` is the type of elements of number fields. The *structure* atoms describe the general algebraic structure, such as `fld` (field), `alg` (algebra), `grp` (group), etc. The *specifier* atoms give a more detailed description `fin` (finite), `pol` (polynomial), `abl` (abelian), etc.

Still more complex types can be constructed by appending a slash (`/`) followed by further *structure^specifier* pairs. The atom `str` can be left out unless it stands alone. In summary, the syntax for composed types is as follows: [`elt-`]*structure^specifier*[`/`...]. For examples `grp^abl` denotes the type of abelian groups, `fld^fun` the type of function fields, `ord^num` the type of orders of number fields, `elt-ord^num` the type of algebraic integers, and `alg^pol/fld^num` is the type of polynomial algebras over number fields.

Given the type of an element within a mathematical structure, we obtain the type of the parent structure by prepending `str-`, and vice versa by prepending `elt-` to the structure's type. Users can create new composed types by combining existing atoms, or by defining new atoms and combining those.

User-Defined Types Objects of user-defined types are represented as records whose `.type` field contains the type of the object. If a record `r` contains a `.base` component then `r` inherits most of the functionality of `r.base`. The functionality can be overwritten by methods installed for the type assigned in the `.type` field or by the `.operations` field which, as in GAP, can be used for overloading the function `Print` and the infix operations `+`, `-`, `*`, `/`, `^`, `in`, and `mod`.

Further special record fields are `.`n (where n is a positive integer) which contains the n-th generator of a structure and `.parent` for the parent structure of an object.

4 Documentation and Help

We designed the new documentation system for KASH with the following goals in mind.

- All documentation should be prepared from a single source. (This provides ease of maintenance, consistency, and up-to-date accuracy.)
- Documentation for new functions should be available to the user as soon as new functions are loaded into the system.
- The help system should support complex search patterns, including full text search (similar to Internet search engines).
- Function documentation should, whenever possible, be accompanied by examples written in working code.
- Output formats should be as follows: ASCII for help within the shell environment (we call this *online help*), XML/HTML for worldwide web documentation, and LaTeX/PDF for printed documentation.

Experience with previous releases of KASH, as well as other computer algebra systems, has taught us that users benefit tremendously from illustrative examples. We have therefore made an effort to provide an example for every standard KASH function. The system also includes special features to credit the authors of each example, and to provide cross-references to other parts of the system and the mathematical literature at large.

Implementation

We decided to represent entries in the documentation system as KASH records. As a result, KASH itself can be used to automatically process and generate documentation. All functions for converting the documentation to any of the desired formats are written in KASH. In this way future developers will not have to find their way through scripts written in other languages. A further advantage is that the documentation system can be easily extended.

The following is a list of the key fields present in a documentation record.

`kind` defines what is being documented.
`name` contains the name of the documented object.
`sin` is the input signature of a function as a list of the input-types.
`sou` is a list of the output-types of a function or constant.
`short` is a text describing the documented object.
`author` contains a list of authors.
`ex` contains examples illustrating how to use a function or generate an object of the given type.
`see` contains a list of references to related documentation records.

Each help entry is identified by a unique hash value computed from the `name` entry and, where applicable, the `sin` entry. The function `DocHash` returns this hash value. The hash values are used in the cross references in the `see` entry for example.

Passing a documentation record to the function `InstallDocumentation` automatically enters it in the online help system and notes the source of the documentation, which can be a file or the interactive shell session.

Online help is accessed by typing `?` into a KASH shell. The `?` may be followed by a variety of modifiers in order to generate more specific results for example `?*` for a full text search, `?(`*type*`)` for finding functions with *type* in the input signature, or `?->`*type* for functions returning objects of type *type*. If more than one result is found, a list is displayed with each item labeled with a four-digit integer, *n*. Their documentation is accessed with `?`*n*.

While online and worldwide web help are accessed in a random order, printed documentation is comprehensive in scope. Printed documentation is assembled from the documentation records described above. The hierarchical structure of the document is determined by a recursive structure of records and lists, where documentation records are referenced by their hash values. The result is translated into LaTeX by a KASH function.

5 Overloading and Type Matching

Like many modern programming languages, KASH 3 supports overloading of functions. Overloading allows users to define multiple functions with the same name but different parameter types. We call such overloaded functions *methods*. Overloading is made possible by KASH 3's strongly typed object system (see section 3) that replaces the predicate driven object system of prior releases.

Calling a method is syntactically equivalent to a function call. Precisely which method to execute is determined at run-time by the method's *input signature* (i.e. the type and order of the method's parameters).

Closely related is the concept of *type matching*. Type matching is the process by which individual function or method calls are matched against known signatures. As with other object-oriented languages, KASH 3 types form a transitive hierarchy. KASH 3 includes a handful of wild card atoms which can be used as simple types, or as parts of composed types. The wild card `any` matches any type or atom, `loc` (local) matches `ser` (power series) or `pad` (p-adic), and `rng` (ring) matches `ord` (order) or `fld` (field). As an example, to write a function that takes an algebraic integer or an algebraic number, we would use the type `elt-any^num` in its input signature.

The overloading mechanism is connected with the documentation system. The `name` entry of a documentation record specifies the method name which the signature `sin` should be added to. The installation of new methods resembles the installion of documentation with a function as a second parameter.

6 The QaoS Databases

Databases facilitate the discovery of constructive examples and broad, heuristic observations. The KANT database is one of the world's largest databases for algebraic number fields. In KASH 3, we wanted to generalise the database design and to improve accessibility. The result is QaoS (Querying Algebraic Objects System), an easily integrated, stand-alone solution for access to the KANT databases. QaoS is more versatile and extensible than its predecessors. An improved representation for elements and polynomials makes it possible to represent (relative) algebraic and transcendental extensions.

QaoS incorporates data from the KANT database [DW96] for number fields, and tables of transitive groups from [Hul05]. The latter are linked to the Galois group entries in the tables of field extensions. Currently QaoS provides the following information

- Algebraic Extensions of $\mathbb{Q}$ up to degree 9 (over 1.3 million number fields)
- All 40226 Transitive Groups up to degree 30
- Extensions of $\mathbb{F}_p(t)$ and $\mathbb{Q}(t)$ (experimental)

The number field records contain generating polynomials, signatures, discriminants, regulators, structure of class groups, and Galois groups.

Accessing the QaoS databases Designed as a client-server application, QaoS transfers information via the hypertext transport protocol (HTTP). Query formats are identical across clients. On the server side, a script translates the client query into SQL (structured query language) for use by the PostgreSQL [PSQL] database. Another script translates the result back into the format requested by the client. A web browser, for example, would select hypertext markup language

(HTML); whereas a computer algebra system, would choose a format easily read by the system. In computer algebra systems the client functions are written in the system's native shell language. They call a non-interactive HTTP client for communicating with the server. The results are read using shell or string pipes, or a temporary file. If a new client can evaluate strings, porting the database interface is straightforward. Currently, client functions are available for GAP 4, KASH 2.56, KASH 3, Maple, and SAGE (contributed by Steven Simek).

Representation of Field Extensions An algebraic extension L/K can be represented by a generating polynomial $f(x) \in K[x]$ such that $L \cong K[x]/(f(x))$. As, in general, databases do not support robust polynomial records, the polynomial f is represented as a list of coefficients. This works well if the coefficients of f are of a type that is supported by the database - integers, for example. In general however, this is not the case. If we consider the extension $\mathbb{F}_p(t)[y]/(g(y))$ with $g(y) \in \mathbb{F}_p(t)[y]$ of the rational function field $\mathbb{F}_p(t)$ over $\mathbb{F}_p$, we see that the polynomial $g(y)$ would have to be represented as a two-dimensional array of integers, but PostgreSQL does not easily support lists of lists of varying dimensions. We solve this problem by storing all elements needed to represent the generating polynomials as lists that contain either references to lists or references to integers. This approach can be used for towers of extensions containing arbitrarily many algebraic and transcendental steps.

Future Developments We will allow registered users bidirectional access to the database. This will empower users to add data to the databases from within their client computer algebra systems. We also plan to provide QaoS clients for more computer algebra systems and support the OpenMath [OM] XML format for representing the algebraic objects developed in the SCIEnce project. Furthermore the tables of number fields will be extended and rechecked, and new tables of function fields and extensions of local fields will be created.

7 Graphical User Interface

Several general computer algebra systems, such as Maple [Ma05] and MuPad [Mu05], provide graphical interfaces. They offer typesetting and plotting, but they do not allow graphical manipulation of algebraic objects. For group theory the XGAP [CN04] package for GAP 4 [GAP] offers a tool for viewing and manipulating subgroup lattices and other structural information on UNIX systems running X Windows. It allows bidirectional communication between the interface and GAP. Working with elements of groups is not supported.

We introduce a graphical user interface for KASH called GiANT. GiANT differs from the preceding systems in that it offers direct, intuitive manipulation of algebraic structures, and it is specialized for working with number fields. In GiANT, number fields created by the user are incorporated into a tower of fields diagram that is updated in real time. The diagram is interactive and

allows users to work with the elements, polynomials, and ideals of a number field simply by clicking on the number field's icon in the diagram. A specialized window containing the elements, polynomials, and ideals of the number field appears. All mathematical information is typeset using the same symbols and special characters that appear in modern math texts. The result is output which is easier to interpret than raw shell output.

Field elements, polynomials, and ideals can be manipulated by drag-and-drop operations, which may, for instance, reveal the minimal polynomial of an element, create the ideal generated by an element, or even move it to an extension field. In a similar way, the user can create relative field extensions by dropping irreducible polynomials from the ground field onto the field diagram.

GiANT is written in Java, but uses KASH to perform its computations. GiANT users may choose to work directly with KASH via a console. Any variables created with the graphical interface are also available in the console. The user may use the console to access features of KASH which are not graphically available. Alternatively, GiANT can be used as a KASH script generator, since all graphically-driven activities generate KASH code.

In the next step, we will provide a more flexible graphical user interface that, for example, displays lattices of subgroups using the same methods as for lattices of subfields. Furthermore, a bidirectional integration of graphical manipulation and a classic text based shell is needed, such that objects and structural information about the objects are displayed graphically as they are created in the shell.

For a more detailed description of GiANT see [KP06]. GiANT is released under the GNU General Public License (GPL) and available from

`http://giantsystem.sourceforge.net`.

This site also contains screen shots and videos of GiANT.

8 Acknowledgements

We would like to thank John Cannon for the long and fruitful collaboration with his Magma group. Special thanks to Allan Steel and Claus Fieker. Thanks to Martin Schönert for helping extend GAP 3 so that it would work with the KANT V4 C-library and Joachim Neubüser for making this possible.

Finally, a warm thanks to the following contributors to KANT and KASH. Michael Pohst (since 1987), Ulrich Schröter (1988–1992), Johannes von Schmettow (1989–1993), Bettina Arenz (1989–1992), Nicole Schröter (1990–1992), Max Jüntgen (1992–1996), Mario Daberkow (1992–1995), Klaus Wildanger (1993–1997), Martin Schörnig (1993–1996), Claus Fieker (since 1993), Jürgen Klüners (since 1994), Florian Heß (since 1995), Andreas Hoppe (1995–1997), Georg Baier (1996), Katharina Geissler (1996–2003), Carsten Friedrichs (1996–2000), Harald Bartel (1997–1999), Maike Henningsen (1999–2002), Hartmut Bauer (1998–2000), José Mendez (since 2000), Robert Fraatz (2002–2005), Marcus Wagner (since 2003), Lea Lieder (2005), Carolin Just (2005), and Osmanbey Uzunkol (since 2006).

References

AED05. B. Assmann, B. Eick, and A. Distler, *GAP package Alnuth: an interface to KANT*, `http://www.gap-system.org/Packages/alnuth.html`.

BB$^+$05. C. Batut, K. Belabas, D. Benardi, H. Cohen, and M. Olivier, *User's Guide to PARI-GP*, 2005, `http://pari.math.u-bordeaux.fr`.

BH96. Y. Bilu and G. Hanrot. *Solving Thue equations of high degree*, J. Number Th., 60:373–392, 1996.

BCP97. W. Bosma, J. J. Canon, and K. Playoust, *The Magma algebra system. I. The user language*, J. Symbolic Comput. 24 (1997), no. 3-4, 235–265, `http://magma.maths.usyd.edu.au`.

CN04. F. Celler, M. Neunhöffer, *GAP package XGAP: a graphical user interface for GAP*, 2004, `http://www-gap.mcs.st-and.ac.uk/Packages/xgap.html`.

DF$^+$97. M. Daberkow, C. Fieker, J. Klüners, M. Pohst, K. Roegner, M. Schörnig, and K. Wildanger, KANT V4, J. Symb. Comp. **11** (1997), 267–283

DW96. M. Daberkow and A. Weber, *A Database for Number Fields* in Jacques Calmet and Carla Limongelli (editors), *Design and Implementation of Symbolic Computation Systems: DISCO'96*, LNCS **1128**, Springer, 1996, 320–330

Fi97. C. Fieker, *Über relative Normgleichungen in algebraischen Zahlkörpern*, Dissertation, TU Berlin, 1997,

Fi02. C. Fieker, *Computing class fields via the Artin map*, Math. Comp. **70** (2001), no. 235, 1293–1303

GAP. *GAP: Groups, Algorithms, Programming*, `http://www.gap-system.org`.

GMP. *GMP: GNU Multiple Precision Arithmetic Library*, `http://www.swox.com/gmp`.

Ge05. K. Geissler, Berechnung von Galoisgruppen über Zahl- und Funktionenkörpern, Dissertation, TU Berlin, 2003.

He02. F. Hess, *Computing Riemann-Roch spaces in algebraic function fields and related topics*, J. Symbolic Comput. **33** (2002), no. 4, 425–445

HPP03. F. Hess, S. Pauli, and M.E. Pohst, *Computing the Multiplicative Group of Residue Class Rings*, Mathematics of Computation **72** (2003)

Hul05. A. Hulpke, *Constructing Transitive Permutation Groups*, J. Symb. Comp. **39** (2005), 1-30.

KP06. A. Karve and S. Pauli, *GiANT: Graphical Algebraic Number Theory*, preprint, 2006, `http://giantsystem.sourceforge.net`.

Ma05. Maplesoft, *Maple*, 2005, `http://www.maplesoft.com`.

Mu05. *MuPad: Multi Processing Algebra Data Tool*, `http://www.mupad.de`.

MPFR. *MPFR library for multiple precision floating point computation*, `http://www.mpfr.org`.

OM. *OpenMath: an extensible standard for representing the semantics of mathematical objects*, `http://www.openmath.org`.

Pa01. S. Pauli, *Factoring polynomials over local fields*, J. Symb. Comp. **32** (2001).

PSQL. *PostgreSQL*, `http://www.postgresql.org`.

Sc$^+$93. M. Schönert et. al. *GAP: Groups, Algorithms, Programming – version 3.27*, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1993, `http://www-gap.mcs.st-and.ac.uk/Gap3`.

St06. W. Stein et al, SAGE: Software for Algebra and Geometry Experimentation, 2006, `http://sage.scipy.org/sage`.

Wi00. K. Wildanger, *Über das Lösen von Einheiten- und Indexformgleichungen in algebraischen Zahlkörpern*, J. Number Theory **82** (2000), no. 2, 188–224.