


HECKE: A Number Theory Package

C. Fieker, C. Sircana (Kaiserslautern University)
T. Hofmann (University des Saarlands)

fieker@mathematik.uni-kl.de
sircana@mathematik.uni-kl.de
thofmann@math.uni-sb.de

HECKE & OSCAR

What is HECKE or OSCAR? OSCAR is a new computer algebra system developed through the SFB-TRR 195 "Symbolic Tools". The idea is to fuse the well established, mature projects GAP, POLYMAKE and SINGULARas well as the comparatively new project HECKE [2] into a comprehensive combined system (OSCAR) where users are free to utilize all parts of the underlying "cornerstones" through a consistent interface that is driven by mathematics only. In the near future we evision more articles detailing other components of OSCAR as they are developed. A core feature of OSCAR is the use of a new programming language: OSCAR, that is, the glueing layer and the mathematical modelling, is written in JULIA [1]. The programming language JULIA is a relatively new language, developed originally at the MIT mathematics department for fast numerical mathematics. It features a just-in-time compiler and a strongly typed language with multiple dispatch. We'll come back to what this means later.

HECKE itself started about 5 years ago with a small nucleous written in C: The ANTIC library for arithmetic in number fields [4]. This nucleous was then extended in JULIA and now comprises about 130,000 lines of code, covering classical algorithmic number theory, some geometry of numbers, class field theory and, more recently, also quadratic forms and associative algebras.

While developing HECKE, a strong emphasis was put on support of large degree fields: Most of the classical algorithms, as available in for example in MAGMA or PARI/GP were conceived and developed at a time when most computations applied to fields of degree 10 or less. On the other hand currently, driven partly by increase in processing power and partly by demands of cryptography (and other applications), computations in fields of degree larger than 100 are desirable, aiming for degree 1000.

The JULIA language through the strong typing and the JIT compiler allows for essentially the same execution speed as pure C code while at the same time being interactive, through its REPL. The downside of this is that the first time a function is executed, there is a delay while the compilation happens.

Installing

Installation of HECKE is made very easy through JULIA. First one needs to install JULIA in version at least 1.0[1]. Then, once JULIAis installed and started, we do

```
julia> using Pgk
julia> Pkg.add("Hecke")
```

Now Hecke is installed. If you want to use the latest, cutting edge version, do

```
julia> using Pgk
julia> Pkg.add("Hecke#master")
```

instead. Now in order to load HECKE, one needs to enter

```
julia> using Hecke
```

[1] available from https://julialang.org/

Note, that HECKE is build on and requires other packages as well, but they will be installed automatically.

Basics

Number fields are finite dimensional extensions of the rationals and this is the view taken by HECKE. Algorithmically we distinguish two cases depending on the base field

- the field K is given explicitly as an extension of $\mathbb{Q}$, that is, $K = \mathbb{Q}[t]/(f)$ for some suitable irreducible polynomial f,
- the field K is given as an extension of an already constructed number field k,

as well as two cases depending on the presentation

- the field K is given via some primitive element (or, more precisely, its minimal polynomial),
- the field K has multiple generators, that is, $K = \mathbb{Q}[\sqrt{2}, \sqrt{3}, \sqrt{5}]$, a so called non-simple field.

Internally, non-simple fields and their elements are represented using sparse multivariate polynomials, while fields given via primitive elements are densely represented. Of course, all fields can be converted to a simple extension of $\mathbb{Q}$, and while this is asymptotically the best presentation, many problems naturally require the other presentations.

```
julia> using Hecke
...
julia> Qt, t = QQ["t"];
julia> K, a =
  number_field(t^3+13t^2-13t+13)
(Number field over Rational Field
  with defining polynomial ...
julia> defining_polynomial(K)
t^3+13*t^2-13*t+13
julia> ans(a)
0
```

(Note that a `;` after a command suppresses the output, and `ans` refers to the return value of the previous command.)

But also

```
julia> E, g =
  number_field([t^2-2, t^2-3, t^2-5],
  ["s2", "s3", "s5"]);
julia> [x^2 for x in g]
Array{NfAbsNSElem, 1}: 2 3 5
```

Of course, this non-simple field can be converted to the dense presentation, that is, we can find a primitive element:

```
julia> Es, phi = simple_extension(W);
julia> phi(gen(Es))
s2 + s3 + s5
```

So, `phi` is a map: `Es` $\to$ `E`, the element `gen(Es)` is the primitive element of the "new" field `Es` and, as expected, in the "old" field the primitive element is the sum of the generators:

```
julia> sum(g)
s2 + s3 + s5
```

There are shortcuts available for common field constructions such as `quadratic_field` and `cyclotomic_field`.

We can now extend the field K further. In order to do so we define a polynomial ring over K and adjoin a root of an irreducible polynomial.

```
julia> Ks, s = K["s"];
julia> F, f = number_field(s^3-2a+a);
```

Class Field Theory

One of the original goals for HECKE was to perform computations in constructive class field theory [3]. To continue with the fields from above:

```
julia> class_group(K)
(GrpAb: Z/6, ClassGroup map...)
```

So the class group is non-trivial, hence we should be able to find a non-trival Hilbert class field:

```
julia> H = hilbert_class_field(K)
Class field defined
  mod (<1, 1>, InfPlc[])
  of structure ... : Z/6

julia> number_field(H)
non-simple Relative number field over
 ...
with defining polynomials ...
[_$1^2+(2*_a^2 + 30*_a + 23),
 _$2^3+(39*_a^2 + 546*_a)*_$2
   +(-13*_a-182)]
```

As you can see (or notice when trying it out), the creation of the Hilbert class field was immediate—and it did nothing. The conversion to a number field however, then found defining equations. It is well known that in the Hilbert class field all ideals from the base field are principal. To check that, We first create an absolute field `Ha`, that is, a simple extension of $\mathbb{Q}$, which is isomorphic to the Hilbert class field. As a second step, we determine its class group and verify that ideals of K become principal.

```
julia> Ha, psi, inj =
             absolute_field(ans);
julia> CH, rho_H = class_group(Ha)
(GrpAb: Z/1, ...)
julia> CK, rho_K = class_group(K);
julia> for c in CK
         I = rho_K(c)
         J = inj(I)
         @show isprincipal(J)[1]
       end
(isprincipal(J))[1] = true
...
```

This shows, indeed, that all ideals become principal! In general, to compute class fields, we start by computing ray class groups:

```
julia> R, mR =
  ray_class_group(7*maximal_order(K))
```

The presentation returned makes it hard to see the structure (however this is very useful computationally), but we can compute the SNF of this abelian group:

```
julia> snf(R)[1]
GrpAb: Z/6 x Z/26
julia> ray_class_field(mR)
Class field defined mod (<7, 7>, ...
with structure: Z/6 x Z/36
```

Of course, while here conversion to a number field is possible (the degree is not too large), in general we want to study subfields which correspond to quotients of R:

```
julia> q, mq =
    quo(R, [3*g for g in gens(R)]);
julia> ray_class_field(mR, mq)
Class field defined mod (<7, 7>, ...
  ... with structure: Z/3^2

julia> for s in subgroups(R, quotype=[3])
         q, mq = quo(R, s[1])
         A = ray_class_field(mR, mq)
         println(number_field(A))
       end
non-simple Relative number field over
...
```

Given that K is a cubic field, the splitting field can at most be a quadratic extension, ramified at only the primes that already ramify in K. Let's see if we can find them!

```
julia> ZK = maximal_order(K)
julia> d = discriminant(ZK)
julia> R, mR = ray_class_group(d*ZK,
   real_places(K), n_quo = 2)

julia> for s in subgroups(R, quotype=[2])
         q, mq = quo(R, s[1])
         global A =
               ray_class_field(mR, mq)
         if isnormal(A)
           break
         end
       end
julia> con = conductor(A)
(<179, _$-8>
Norm: 179
Minimum: 179
..., InfPlc[...]
```

Since the Galois group of K is dihedral, there should be a quadratic extension B of $\mathbb{Q}$ contained in A. By Kronecker-Weber, B should be a subfield of a cyclotomic field:

```
julia> C =
  cyclotomic_field(ClassField,
                   minimum(con[1]))
julia> B = maximal_abelian_subfield(A, QQ)
julia> issubset(B, C)
true
```

This is obviously only showing a very small part of the functionality of HECKE, even of the class field theory. This is meant to give an indication or feeling about the use of both HECKE and, eventually OSCAR. This example, and a couple more are also available as Jupyter notebooks from

```
https://github.com/thofma/HeckeTutorials.jl
```

The source code of HECKE itself is also freely available from

```
https://github.com/thofma/Hecke.jl
```

References

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, *Julia: A fresh approach to numerical computing*, SIAM review **59** (2017), no. 1, 65–98.

[2] C. Fieker, W. Hart, T. Hofmann, and F. Johansson, *Nemo/Hecke: Computer Algebra and Number Theory Packages for the Julia Programming Language*, ISSAC'17—Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation, ACM, New York, 2017, pp. 157–164.

[3] C. Fieker, T. Hofmann, and C. Sircana, *On the construction of class fields*, Proceedings of the Thirteenth Algorithmic Number Theory Symposium, Open Book Ser., vol. 2, Math. Sci. Publ., Berkeley, CA, 2019, pp. 239–255.

[4] W. Hart, *ANTIC: Algebraic Number Theory in C*, Computeralgebra Rundbrief **56** (2015), 10–11.

Mit einem CAS Termumformungen motivieren

J. H. Müller
(Rivius Gymnasium Attendorn)

mueller@rivius-gymnasium.de

Einführung

Im vorliegenden Artikel wird beschrieben, wie Termumformungen auf eine einfache Art mit Hilfe eines CAS motiviert werden können.

Termumformungen früh motivieren

Terme sollten in verschiedenen äquivalenten Darstellungsarten schon früh im Unterricht thematisiert werden. Im Kontext von Formeln, etwa zur Berechnung des Flächeninhalts eines Dreiecks mit bekannter Grundseite g und Höhe h, kann der Flächeninhalt auf verschiedene Arten in Form von

$$g \cdot h/2, \quad g/2 \cdot h, \quad \frac{1}{2} \cdot g \cdot h$$

oder

$$(g \cdot h)/2$$

geschrieben werden. Trotz der Äquivalenz dieser Terme, können sie herleitungsbedingt geometrisch unterschiedlich interpretiert werden. Zerteilt man ein Dreieck auf verschiedene Arten in flächeninhaltsgleiche Rechtecke, ergeben sich zerlegungsbedingt verschiedene äquivalente Flächeninhaltsformeln. Abb. 1 zeigt zwei Zerlegungsmöglichkeiten, die auf die Terme $g \cdot h/2$ und $g/2 \cdot h$ führen können, deren Äquivalenz anschließend mit Hilfe von Rechenregeln und Rechengesetzen nachgewiesen werden kann.

Warum hier ein CAS einsetzen?

Neben der Möglichkeit zur Vernetzung von Geometrie und Algebra existiert hier eine Möglichkeit zum Einsatz von CAS und kann anhand von Computergrafiken motiviert werden: Die Beschreibung von Oberflächen mit Hilfe von Dreiecken wird u.a. in Computerspielen genutzt und "Tessellation" genannt. Je leistungsfähiger die Grafikkarte in einem Computer ist, desto detaillierter kann diese Unterteilung sein und desto realistischer wirken die darzustellenden Figuren. Aktuelle Grafikkarten verarbeiten etwa 7 Mrd. Dreiecke pro Sekunde (Stand 2019, vgl. https://t1p.de/ujx0). Das entspricht bei einer Bildfrequenz von 60 Bildern pro Sekunde über 100 Millionen Dreiecken pro Bild.

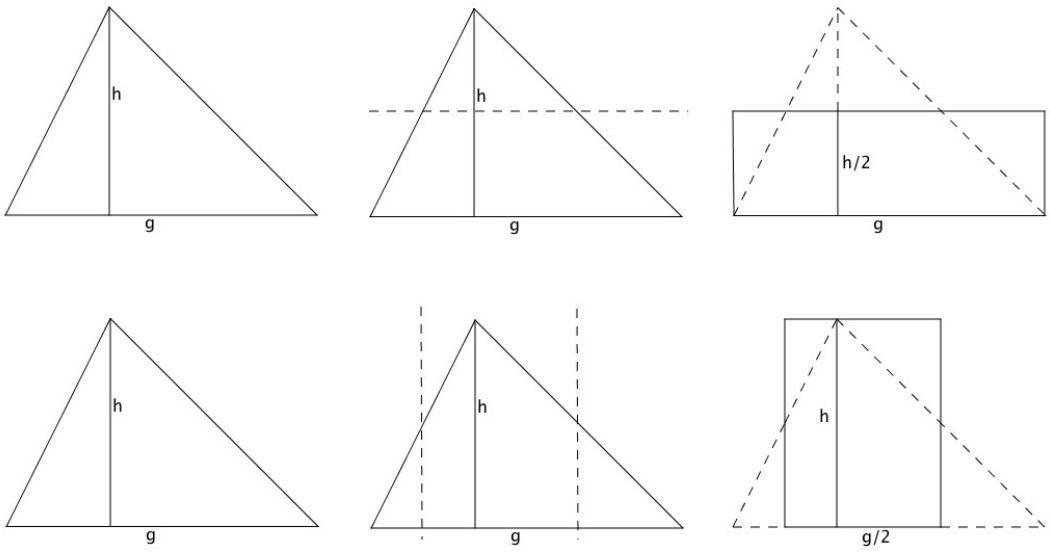

Abbildung 1: *Zerlegungsmöglichkeiten eines Dreiecks*

Lernenden kann man mit Hilfe eines CAS gut verdeutlichen, dass bei dieser enormen Anzahl an Dreiecken die Programmierung von Formeln einen enormen Einfluss auf die benötigte Rechenzeit hat, um etwa Spiele flüssig aussehen zu lassen. Dies kann vereinfacht mit Hilfe der gewonnenen Formeln und zwei zu programmierenden Schleifen verdeutlicht werden, um etwa die Oberfläche von einer Million Dreiecken mit Hilfe verschiedener Formeln zu berechnen und die dafür benötigte Rechenzeit zu vergleichen. Abb. 2 zeigt verschiedene mit dem CAS wxMaxima programmierte Schleifen und in eckigen Klammern die hierfür jeweils benötigte Rechenzeit in Sekunden.

```
for g:1 thru 1000 do
 for h:1 thru 1000 do
  g*h/2$
time(%);
[4.435]

for g:1 thru 1000 do
 for h:1 thru 1000 do
  g/2*h$
time(%);
[4.604]

for g:1 thru 1000 do
 for h:1 thru 1000 do
  0.5*g*h$
time(%);
[3.406]

for g:1 thru 1000 do
 for h:1 thru 1000 do
  g*0.5*h$
time(%);
[3.382]

for g:1 thru 1000 do
 for h:1 thru 1000 do
  g*h*0.5$
time(%);
[3.334]

for g:1 thru 1000 do
 for h:1 thru 1000 do
  (g*h)*0.5$
time(%);
[3.695]
```

Abbildung 2: *Laufzeitmessung von Schleifen mit wxMaxima*

Sehr auffällig - aber aufgrund der aufwändigeren Gleitpunktarithmetik weniger überraschend - ist die deutlich erhöhte Rechenzeit unter Nutzung von Division anstelle von Multiplikation. Auch Experimente zum (überflüssigen aber von Lernenden gern genutzten) Einsatz von Klammern in dieser Formel zeigen, dass dies die Rechenzeit negativ beeinflusst. Experimente zur Position von Konstanten in einer Formel liefern in diesem Beispiel wiederum keine konsistent besseren Laufzeit-Ergebnisse. Auf diese Art kann der Einsatz eines CAS auch als Motivation für weiterführende Fragen zur Vernetzung von (fachübergreifenden) Themengebieten dienen: Um wie viel Prozent kann die Rechenzeit verkürzt werden, wenn man das schlechteste mit dem besten Ergebnis vergleicht? Wie viele Dreiecke wurden mit den zwei Schleifen berechnet? Verdoppelt sich die Rechenzeit, wenn man die Anzahl der Dreiecke verdoppelt? Wie kann man die Anzahl der Dreiecke mit Hilfe der Schleifen verdoppeln? Welche Formeln könnte man noch ausprobieren und wie kann man geometrisch die Gültigkeit der Formeln veranschaulichen?

CAS in der analytischen Geometrie

Ob der vorgestellte Einsatz der Laufzeitmessung im Kontext zur Berechnung von Flächeninhalten von Dreiecken lohnenswert erscheint, kann kritisch diskutiert werden, soll die Idee lediglich erläutern und wurde vom Autor an dieser Stelle auch nur in Klassen eingesetzt und mit Schülern besprochen, die interessiert waren. Besonders gewinnbringend erscheint die Idee spätestens in der Oberstufe im Themengebiet Geometrie im Kontext von Computerspielen etwa zur Berechnung des Schnittpunktes von Geraden mit Ebenen, um räumliche Punkte so auf einem zweidimensionalen Bildschirm zu projizieren, dass ein dreidimensionaler Eindruck entsteht. Abb. 4 am Ende des Artikels zeigt die Idee der pseudo-dreidimensionalen Darstellung am Beispiel eines Tetraeders mit Hilfe der Zentralprojektion.

Im Unterricht können zunächst anhand konkreter räumlicher Koordinaten eines Fluchtpunktes

$$F(f1, f2, f3)$$

und eines Eckpunktes

$$E(e1, e2, e3)$$

des Tetraeders die Koordinaten der Projektion P_E von E etwa in die y-z-Ebene berechnet und visualisiert werden. Durch den zunehmenden Einsatz von Variablen anstelle konkreter Koordinatenwerte und einer Verallgemeinerung der Projektionsebene zu

$$e : ax + by + cz = d$$

nähert sich die Berechnung der Koordinaten P_E der Idee einer sogenannten Spiele-Engine an, also einem Programm, das für die visuelle Darstellung eines Computerspielablaufs genutzt wird.

Abb. 3 zeigt, wie man syntaktisch mit Hilfe von wxMaxima die Koordinaten von P_E berechnen lassen kann. Die Rechenausdrücke der Koordinaten bieten vielfältige gewinnbringende Diskussionsanlässe für Vermutungen, wie die Rechenzeit verringert werden kann. Gute Beiträge oder Ideen wären etwa, dass gemeinsame Rechenausdrücke getrennt und nur einmalig berechnet werden (wie etwa den Faktor s in allen drei Koordinaten), ob es gewinnbringend ist im Nenner von s zu faktorisieren oder ob Subtraktionen in ihrer Anzahl minimiert werden können oder sollten.

```
gl:solve(a*(e1+s*(f1-e1))+b*(e2+s*(f2-e2))
   +c*(e3+s*(f3-e3))=d,s);
```

$$[\, s=-\frac{c\,e3+b\,e2+a\,e1-d}{c\,f3+b\,f2+a\,f1-c\,e3-b\,e2-a\,e1}\,]$$

```
lsg:map(rhs,gl);
```

$$[\,-\frac{c\,e3+b\,e2+a\,e1-d}{c\,f3+b\,f2+a\,f1-c\,e3-b\,e2-a\,e1}\,]$$

```
x:f1+lsg*(f1-e1);
```

$$[\,f1-\frac{(c\,e3+b\,e2+a\,e1-d)\,(f1-e1)}{c\,f3+b\,f2+a\,f1-c\,e3-b\,e2-a\,e1}\,]$$

```
y:f2+lsg*(f2-e2);
```

$$[\,f2-\frac{(c\,e3+b\,e2+a\,e1-d)\,(f2-e2)}{c\,f3+b\,f2+a\,f1-c\,e3-b\,e2-a\,e1}\,]$$

```
z:f3+lsg*(f3-e3);
```

$$[\,f3-\frac{(c\,e3+b\,e2+a\,e1-d)\,(f3-e3)}{c\,f3+b\,f2+a\,f1-c\,e3-b\,e2-a\,e1}\,]$$

Abbildung 3: *Geraden-Ebenschnittberechnung mit wxMaxima*

Zusammenfassung

Die Idee der Zeitmessung im Rahmen von Schleifenprogrammierung anhand der beiden Beispiele beschreibt eine einfach zu realisierende alternative Idee zur Motivierung von Termumformungen mit Hilfe eines CAS. Sie ist zwar nicht zwingend an ein CAS gebunden, ein CAS vereinfacht aber - wie im zweiten Beispiel gezeigt - die Berechnung, Darstellung und gemeinsame Analyse komplexer Termen auf schulischem Niveau. Zudem eröffnet die beschriebene Idee eine motivierende Möglichkeit Schüler(gruppen) mittels Termumformungen um eine möglichst geringe Laufzeit wetteifern zu lassen. Leider verfügen viele CAS-Handhelds nicht über die vorgestellte Laufzeitmessfunktion. Dies wäre zukünftig wünschenswert.

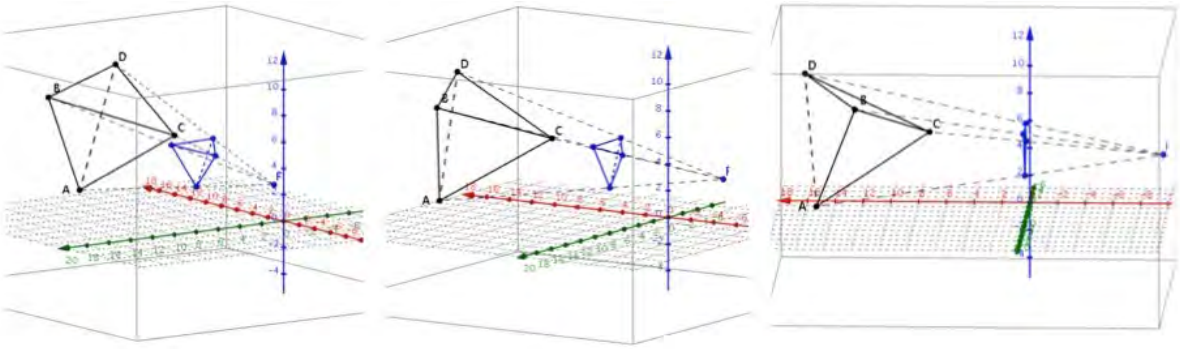

Abbildung 4: *Eine Tetraederprojektion in die y-z-Ebene mit Geogebra*