The Scratchpad II Newsletter

Volume 1 Number 1 September 1, 1985

In this issue ...

Welcome! 1
A Brief Introduction to Scratchpad II 1
Elliptic Curve Calculations in Scratchpad II 3
The LISP/VM foundation of Scratchpad II 4
Scratchpad II System Status 5
  The Interpreter 5
  The Compiler 7
  Algebraic Facilities 7
  New Remote User Interface 8
Algebra Snapshot: Eigenvalues and Eigenvectors 8
Visitors to the Scratchpad II Group in 1985 10
1985 Demonstrations of Scratchpad II 10

The Scratchpad II Newsletter

An informal quarterly publication of the Computer Algebra Group, Knowledge Systems, Computing Technology Department, IBM Thomas J. Watson Research Center, Box 218 Yorktown Heights, New York 10598.

Editor: Robert S. Sutor

Volume 1, Number 1 September 1, 1985

The Scratchpad II Computer Algebra Group

Manager
Richard D. Jenks
IBM VNET: JENKS at YKTVMZ
CSNET: jenks.yktvmx@ibm
914-945-1233

Algebra
Barry M. Trager
IBM VNET: BMT at YKTVMZ
CSNET: bmt.yktvmz@ibm
914-945-1868

Patrizia Gianni
IBM VNET: GIANNI at YKTVMZ

Compiler
Stephen Watt
IBM VNET: SMWATT at YKTVMZ
CSNET: smwatt.yktvmz@ibm
BITNET: smwatt@yktvmz
914-945-3405

Interface
Michael Lucks
IBM VNET: LUCKS at YKTVMZ
914-945-4060

Interpreter
Albrecht Fortenbacher
IBM VNET: ALBI at YKTVMZ
CSNET: fort@germany
914-945-2065 (until 9/85)

Scott C. Morrison
IBM VNET: SCM at YKTVMZ
ARPANET: morrison@berkeley.arpa
914-945-2065 (until 9/85)

Robert S. Sutor
IBM VNET: SUTOR at YKTVMZ
CSNET: sutor.yktvmz@ibm
BITNET: sutor@yktvmz
914-945-2360

System
Martin L. Brock
IBM VNET: MBROCK at YKTVMZ
914-945-2471

Consultants
David and Gregory Chudnovsky
Professors of Mathematics,
Columbia University.
IBM VNET: SPAD at YKTVMZ

Victor Miller
IBM Research
IBM VNET: VICTOR at YKTVMZ

Secretary
Evelyn Zoernack
IBM VNET: EVELYNZ at YKTVMT
914-945-1187

Welcome!

Welcome to the first issue of the ***The Scratchpad II Newsletter***. It is our intention to provide this newsletter on a quarterly basis to the growing user community of our computer algebra system. Articles contained in the issue are generally of an expository nature and should serve to inform you of the current state of **Scratchpad II** and some details of its implementation. In future issues we will solicit and welcome contributions from our users.

This issue will be somewhat large in that it will serve to introduce many of you to our system.

Please feel free to contact any **Scratchpad II** group member if you have questions or want information about our system. The names and network addresses of the group appear inside the front cover of this newsletter. To get on our mailing list for future issues, please fill out and return the form on the last page. We're looking forward to hearing from you.

Robert S. Sutor

A Brief Introduction to Scratchpad II

by
Richard D. Jenks

The **Scratchpad II** system represents a new generation of extensible computer algebra systems with a general-purpose programming language based on parameterized and dynamically constructible abstract datatypes with generic operations. Here we describe some of the more important aspects of its design.

Representation

Most algebra systems use a small, fixed number of basic algebraic structures such as rational functions and expression trees to represent user data. In **Scratchpad II**, algebraic structures are built by "domain constructors." Some constructors such as lists, polynomials, and matrices require other domains as parameters. Others, such as the integers, rational numbers, and floats, have no parameters. Since constructors can be composed, **Scratchpad II** can build arbitrary towers of constructors and thus create an unlimited number of computational domains.

Modularity

Scratchpad II is a modular system based on abstract data types. There is no fundamental distinction between data structures (e.g. lists and tables) and algebraic structures (e.g. polynomials and matrices): all are defined by domain constructors. Domain constructors are independently defined and compiled. Constructor definitions are rather concise, typically 1 or 2 pages of code. As a result of this modularity, **Scratchpad II**'s complexity grows only linearly with the number of extensions.

Dynamic Types

As described in the section "The Interpreter" on page 5, it is part of the responsibility of the system interpreter to build a domain appropriate for given user input. If the user enters the expression:

```
u := 2*x + 7 + 3*%i
```

where *%i* denotes the square root of -1, the interpreter will choose the domain tower **P G I**, meaning "polynomials of Gaussians over the integers," to represent the expression. Unlike other abstract data type languages such as **Ada** and **Modula 2** where types must be specially pre-generated for each application, the domains of **Scratchpad II** are created dynamically, specifically tailored for user input. More generally, domains are first class objects: they can be passed to, manipulated, and returned from **Scratchpad II** functions. The full set of domains needed in a computation often cannot be predicted by a user before he starts. For example, the algorithm for symbolic integration will generally need to build extensions of an initially given domain in order to introduce new algebraic numbers and then new logarithms involving these numbers. Even simpler tasks such as factorization cannot predict in advance which finite fields they will wish to construct, or which variables will be evaluated away and which will remain.

Consistency

Scratchpad II also goes beyond most other abstract data type languages by providing mechanisms to check the algebraic consistency of constructs encountered. This allows the system to *know* that **P G I** is algebraically consistent. However **P L I**, "polynomials of lists of integers," is inconsistent since the polynomial constructor requires its coefficient domain to have operations such as "+" and

"*"; lists of integers do not. Because of the dynamic nature of the system, consistency cannot be left to some "linker" to resolve but must be solved as users interact with the system, defining and loading new data types.

Generality

Consistency is provided by an algebraic knowledge base founded on the use of *categories*. A category describes an abstract set of operations together with explicit or implicit axioms (e.g. associativity and commutativity) associated with those operations. New categories can be defined as extensions or arbitrary combinations of existing ones. Categories thus provide a hierarchical organization of abstract operations similar to that of SMALLTALK classes except with multiple inheritance.

Through categories, the system is able to provide compiled "polymorphic functions," that is, functions defined so as to apply over a variety of domains. A function in a domain is parameterized by domain parameters passed to its constructor. For example, in computing *m***2, where *m* denotes a "matrix of matrices of integers," the same matrix multiplication function is used at both matrix levels of the domain tower even though it is calling different routines to multiply the elements of the matrix: in one case itself recursively, in the other, the integer multiplication routine. More generally, it is possible to define algorithms parameterized entirely by domains characterized only by categories, such as an algorithm for sorting any aggregate of elements on which some total-ordering is defined.

Efficiency

We are currently experiencing performance problems with the interpreter. Trivial computations often require non-trivial amounts of computing time if they require the consideration of domains not previously used. However, this problem is only associated with the type analysis of user expressions, not with the run-time efficiency of compiled algebra code. Some benchmarks have shown **Scratchpad II** to be comparable to **Scratchpad** for similar algebraic computations.

All functions in **Scratchpad II** are compiled and take an additional parameter, a structure containing the domain run-time parameters. Domains are implemented as vectors which can be transformed so that they can be viewed in alternate categorical ways, such as an ordered set, group, or ring. The design of domain towers makes function calling from level-to-level in the domain tree essentially equivalent to that of a normal **LISP/VM** function call. As a result, the run-time efficiency of algebra code running in the dynamically constructed towers of **Scratchpad II** is comparable to that of similar code running in specially pre-generated and compiled towers in **LISP/VM**.

Extensibility

In **Scratchpad II**, user and implementor alike use the same programming language. Programs are always compiled so that there is no loss of efficiency for user extension. With the exception of functions from data types provided by **LISP/VM** (such as large integers (bignums) and floating point arithmetic or the operations on basic data structures such as lists or vectors), all algebra code is written in the high-level **Scratchpad II** language. Existing constructors can be edited, modified, and recompiled. A database of dependencies is maintained dynamically to ensure that all information (such as sets of operations) is kept current.

Size

Scratchpad II's generality comes at some cost. Its current implementation of domains makes this system require more space than other computer algebra systems. The current implementation of **Scratchpad II** in **LISP/VM** requires users to have at least 4 megabytes of virtual memory allocation. In addition, a 3 megabyte segment is shared among all users.

Comparison with Scratchpad I

Interactive users of **Scratchpad** will notice a great deal of similarity between the old and new systems. As before, programs can be written as sets of mutually recursive rewrite rules with computation of values delayed until requested. In **Scratchpad**, the display format of expression could be altered by use of a "format" statement. In **Scratchpad II**, however, this is done by altering the domain of computation. The domain of a computed result in **Scratchpad II** is displayed as the "Type" (see "Algebra Snapshot: Eigenvalues and Eigenvectors" on 8). An on-line database is provided for telling the user what functions are applicable to a computed result.

Unlike its predecessor, **Scratchpad II** has assignments and declarations. The assignment

```
x := e
```

causes the expression *e* to be immediately evaluated and then assigned as the value of *x*. Unlike rewrite rules where values are produced by effectively rewriting the left hand side by the right, assignments capture the value produced by evaluation at the time of assignment. It is possible to "declare" the type of a variable, e.g.

```
u : P[x]P I
```

means "*u* is declared to be a polynomial in *x* whose coefficients are polynomials (in any other variables) over the integers." A declaration forces any expression assigned to the variable to be converted to the declared type.

Elliptic Curve Calculations in Scratchpad II

by
David and Gregory Chudnovsky

Various models of elliptic curves and expressions for laws of additions on them were studied, programmed and executed in **Scratchpad II**. The calculations were performed over polynomial rings and algebraic number fields, and mod N. The aim was to realize various new primality tests and factorization algorithms based on elliptic curves and Abelian varieties (see [1]). Particular attention was devoted to finding the algebraic laws of addition with minimal complexity, made possible by the computer algebra facilities of **Scratchpad II**. This led to a substantial reduction in the amount of programming: most of the programs written in **Scratchpad II** or its BOOT mode (directly **LISP/VM** oriented) take a dozen lines. We present a few examples of a probabilistic factorization algorithm (Lenstra's generalization of the p-1 method [3]) that are based on a random choice of a point X on an elliptic curve E mod N, and the computation of the gcd of N with the z-coordinate of a multiple M*X on E for an appropriate seed number M (= lcm{1,...,L}). In the examples below X was taken as X = (x,y,z) for a curve E:

$$x^3 + y^3 + z^3 = dxyz$$

uniquely determined by X.

To compare the performance of the algorithm we chose as a benchmark the factorizations of [2] of the "most wanted" numbers. We list below 7 numbers, the time needed for the complete factorization, and the corresponding points X. The "seed" number L was chosen as L <= 100000.

1. N = 2**211 - 1. Total time for factorization was T = 19953.445 sec. The points X were (694805341, 172477448, 1) and (376692538, 289694810, 1) (for the second largest prime factor).
2. N = 10**67 - 1. Total time for factorization was T = 11202.92 sec. The points X were (983555142, 1429640635, 1) and (1049941477, 493276540, 1) (for the second largest prime factor).
3. N = 3**124 + 1. Total time for factorization was T = 17.711 sec. The point X was (1868613011, 978022149, 1) for the second largest prime factor.
4. N = 3**128 + 1. Total time for factorization was T = 5819.496 sec. The points X were (1357825771, 1816500175, 1) and (1503779485, 286762852, 1) (for the second largest prime factor of N).
5. N = 11**64 + 1. Total time for factorization was T = 13479.041 sec. The point X was (1181636285, 1979758186, 1) (for the second largest prime factor of N).
6. N = 5**79 - 1. Total time for factorization was T = 4820.886 sec. The points X were (170856428, 398349357, 1) and (623733768, 1225757769, 1) (for the second largest prime factor of N).
7. N = 2**212 + 1. Total time for factorization was T = 1720.114 sec. The point X was (1330048984, 975992465, 1) and, for the second largest prime factor of N it was a point (68628, -3705912,1) on a curve

$$x^3 + axz^4 + bz^6 = y^2$$

for (a,b) = (0, -309490534257408).

All **Scratchpad II** calculations were performed on an IBM 3081 Model K. The benchmark factorizations in [2] were performed on a Cray I-S. The following table summarizes the timing differences in the two algorithms:

	CPU Hours	
N	New Algorithm (in [1]) on IBM 3081	Algorithm (in [2]) on Cray I-S
2**211 - 1	5.543	22.25
10**67 - 1	3.112	1.22
3**124 + 1	0.005	1.8
3**128 + 1	1.617	6.05
11**64 + 1	3.744	15.34
5**79 - 1	1.339	1
2**212 + 1	0.478	1

[1] D.V. Chudnovsky, G.V. Chudnovsky, Research Report RC 11262, 7/12/85, IBM Research.

[2] J.A. Davis, D.B. Holdridge, Sandia Report 1658, August 1984.

[3] Lenstra, H.W., Letter to A.M. Odlyzko, February 14, 1985.

The LISP/VM foundation of Scratchpad II

by
James H. Davenport

Scratchpad II, like most computer algebra systems, is implemented in LISP, in this case **LISP/VM** (IBM Program Product 5798-DQZ). While most aspects of the implementation should not concern the user, there are some aspects of **LISP/VM** that affect **Scratchpad II**'s external appearance.

Storage of the System

The **Scratchpad II** system kernel is stored internally in two files. One, whose current filename is SPAD7500, and whose filetype is always SEGMENT, contains the read-only portion of the system held in common among all **Scratchpad II** users on a given CP system. This file constitutes a CMS shared segment and currently contains about 3 Megabytes of compiled code and read-only names, etc.. The sharing of this data between several users can greatly decrease the paging load on the system, as well as improving an individual user's response time. This "shared segment" feature is, of course, available to all **LISP/VM** applications, and is, indeed, used by the standard **LISP/VM** system.

The other, whose current name is COURSE SHLISPWS, contains the read/write part of the system kernel. This contains essentially a complete copy of the data initially loaded when the user starts a **Scratchpad II** session. This file will contain new algebra code loaded or recompiled, the compiled version of rules, etc.. These files are large (typically one megabyte), but the saving and loading processes are relatively fast. Note that the SEGMENT file, which is much larger, is not saved or copied, so that a user working from a non-standard SHLISPWS still gets all the advantages of the shared segment.

The Garbage Collector

LISP/VM divides space into four sections, known as the *stack* (for program control), *heap* (for most user data), *bpi space* (for Binary Program Images, i.e., compiled programs) and *nilsec* (for holding various system pointers). This division is transparent to the user, since the process of garbage collection will re-allocate space among the various areas as necessary. Hence, unlike many algebra systems, there are no magic space allocation options to quote to the system, and, conversely, if the system says that you have run out of space, then you really have run out, and there is none lurking in other spaces, or under other type-codes, such as can occur in many LISP systems, where a vector, once freed, can only be re-used by a vector and not by a large number or by general LISP data structures.

The garbage collection algorithm used in **LISP/VM** is a "two-space" method, a variant of that described by Cheney, C.J. in "A Nonrecursive List Compacting Algorithm" [Comm. ACM 13(1970) pp. 677-678]. This algorithm has the advantage that it only processes data that are in use, and also often arranges to gather connected data together, thus reducing the working set required for accessing a data structure.

Garbage collection is normally invisible to the user, showing only as a small (.25 seconds in the initial system, rising to .8 seconds in a system that almost completely fills 16Mb of memory on an IBM 3081 model K) increase in CPU time.

The Compiler

Unlike many LISP systems, the **LISP/VM** compiler is an integral part of the system, and can be called

"on the fly" from within any application, such as **Scratchpad II**. This is used to compile a user's rules as they are translated, thus improving the efficiency of their execution. Similarly, a user who wishes to write or change a piece of the algebra can do so from within his system, without needing to call a separate program to perform the compilation. Again, many LISP-based algebra systems will require the user to run a separate program to translate rules into machine code.

Though the casual user will not notice this, the **Scratchpad II** group have augmented the standard compiler with many macros to improve the compilation of algebra code. Further macros can be added in order to optimise particular constructs.

The compiler is also capable of producing files called LISPLIBs, which contain the external form of function definitions, and other information that the system may require. The majority of the algebra in the **Scratchpad II** system is contained in LISPLIBs rather than in the kernel, so that it can be changed independently of other programs, and so that it does not occupy space except when needed. However, parts that most users will require are pre-loaded into the SEGMENT so that they are shared among users. These LISPLIBs are capable of containing more data than just the compiled code, and **Scratchpad II** also uses them for storing consistency checking information and type information that is required by the algebra compiler.

The Trace System

Scratchpad II has a trace package for system developers which is implemented via the **LISP/VM** debugging facilities. Unlike many LISPs, no special incantation (such as (SSTATUS UUOLINKS) of MACLISP) is needed before functions can be traced. However, some LISP functions can not be traced since they are part of the trace package, and tracing them will cause recursion. The most obvious example of this class is EQUAL, but there are others.

The output from the trace package may contain symbols such as *%L1* or *%L1=*. These are the LISP system's method of printing circular data structures, such as are quite common in **Scratchpad II**. For example, multivariate polynomials are defined as univariate polynomials over themselves, and this structure translates into a recursive LISP data structure.

The normal setting for the system causes errors to result in a message and a return to the interpreter. System developers may issue a command which will cause a **LISP/VM** error break to be taken instead. This gives the full facilities of the read-eval-print loop of **LISP/VM**. Again, it is not necessary to set any special options before doing this in order to have the full debugging information available. The detailed **Scratchpad II** information available in this state is subject to change, and is therefore not yet documented.

The **Scratchpad II** group plans to provide trace facilities for users that will allow them to halt operations and view the computational environment. Output from such tracing will be in the usual **Scratchpad II** algebraic notation.

Scratchpad II System Status

As the title implies, this section will be used each issue to discuss the state of the several components of the developing **Scratchpad II** system.

The Interpreter

Unlike the strongly-typed **Scratchpad II** programming language, where variables and parameters to functions must be given explicit data types, type declarations are optional during interactive computation. It is the responsibility of the interpreter to read the user's input expression and select the appropriate domains of computation in which the functions are defined and implemented.

If the user enters the expression:

```
x**2 - %i*y
```

(where *%i* stands for the square root of -1) the interpreter creates an object of type **P G I** (i.e., polynomials whose coefficients are Gaussian integers). This is done by choosing the default types of **Integer** for *2*, **Polynomial** for *x* and *y*, and **Gaussian** for *%i*, and resolving these types to **P G I**. If the analysis does not produce a result of the desired type, the user may make explicit declarations to guide the interpreter. Note that one may abbreviate the type names, often to a single letter.

A user need not completely specify all information in a declaration. For example, one may want a **Gaussian** but not particularly care what datatype is chosen for the real and imaginary parts. Such an incomplete specification is called a *mode*. To regard the above result as a **Gaussian**, a variable can be declared to have that mode:

```
p : G := x**2 - %i*y.
```

The type created by the interpreter is now **G P I**.

There are two useful operations for giving the interpreter additional type information: "::" and "$". The operator "::" is *coercion*: the changing of the datatype of an object. The dollar sign is used to specify the computational domain in which an operator is to be found. To illustrate these we give three ways to perform the operation 5*4 so that the result is in the finite field with 7 elements:

5::(GF 7) * 4::(GF 7)	Both integers are coerced to the type **GaloisField(7)** and the multiplication is performed in the finite field.
5 *$(GF 7) 4	The operation "*" is specified as being in **GF 7** and so the numeric literals are taken to be elements of **GF 7**.
(5*4)::(GF 7)	"*" is performed in the domain Integer and the result is coerced to **GF 7**.

User functions are treated as rewrite rules and are compiled for efficiency. They may be written with a single function body or in a piecewise fashion. For example, the Fibonacci numbers can be generated by the functions in either of the following definitions:

```
fib 0 == 1
fib 1 == 1
fib n == fib(n-1) + fib(n-2) when n > 1
```

or,

```
fib n == if n=0 or n=1 then 1 else fib(n-1)+fib(n-2)
```

Moreover, the interpreter specially compiles the above examples as recurrence relations and they execute in *linear* time, rather than the exponential time implied by the recursive definitions.

The first time a user-written function is called, the interpreter type-analyzes its body based on the types of the arguments. As with variables, the user may explicitly declare the type he wishes the function to return. In the following examples, comments begin with double hyphens (i.e., "--").

```
-- Define the Legendre polynomials
p(0) == 1
p(1) == x
p(n) == ((2*n-1)*x*p(n-1)-(n-1)*p(n-2))/n when n > 1
```

```
-- compute the fifth Legendre polynomial
p 5
   compiling p with signature I -> RF I
   compiling p as a recurrence relation

                 5        3
          63x  - 70x  + 15x
   (4)  -------------------
                   8

Type: RF I
```

Given no type information, the analysis determined that the function result should be **RationalFunction Integer** (quotients of polynomials with integer coefficients). If one had wanted the result to a be univariate polynomial in x with rational number coefficients, the example could have been done this way:

```
p: I -> P[x]RN

-- Define the Legendre polynomials again.
p(0) == 1
p(1) == x
p(n) == ((2*n-1)*x*p(n-1)-(n-1)*p(n-2))*(1/n)
        when n > 1

-- compute the fifth Legendre polynomial
p 5
   compiling p with signature I -> P[x]RN
   compiling p as a recurrence relation

            63    5       - 35    3       15
   (5)   (----)x  + (------)x  + (----)x
            8              4              8

Type: P[x]RN
```

The interpreter provides facilities similar to ***APL*** and **SETL** for collecting objects and reducing over these collections. For example, the statement

```
[i**2 for i in 1..10]
```

will produce a list of the squares of the integers from 1 to 10, and

```
+/[i**2 for i in 1..10]
```

will yield the sum of these numbers, i.e.,

$$\sum_{i=1}^{10} i^2$$

The infix operator "!" is used to distribute a function across a collection:

```
-- collect a list of the primes < 30
primes := [i for i in 1..30 | isPrime i]

   (3)  [2,3,5,7,11,13,17,19,23,29]

-- add 1 to each element in the list
1 + !primes

   (6)  [3,4,6,8,12,14,18,20,24,30]
```

```
-- make a list of the double of each
-- prime less than 30
!primes + !primes

   (7)  [4,6,10,14,22,26,34,38,46,58]
```

The interpreter can also collect over infinite index sets. The following demonstrates a collection of the factorials of the positive integers.

```
posInts == [i for i in 1..]

-- Define the factorial function
fact n == */[1..n]

fact!posInts
   compiling fact with signature I -> I

 (10)  [1, 2, 6, 24, 120, 720, 5040,
        40320, 362880, 3628800, ...]
```

Scott C. Morrison
Albrecht Fortenbacher
Victor Miller

The Compiler

One of the components of the **Scratchpad II** system is the programming language compiler. The compiler under which the existing algebra code was developed has not been made available to the **Scratchpad II** user community for a number of reasons. This compiler corresponds to an older dialect of the language and is fairly difficult to use, mainly because of shortcomings in error diagnostics and recovery.

Over the past year, the compiler has undergone complete redesign for a new implementation. The new compiler will be made available in the fall of 1985, allowing users to create their own categories, domains and packages. A complete description of the compiled language and the facilities it provides is given in **Scratchpad II Programming Language Reference Manual** (in preparation).

The new compiler is being written entirely in **Scratchpad II** and is self compiling. This has had a number of advantages, including the early identification of points for improvement and the addition of a rich set of data structures to the **Scratchpad II** library.

Stephen M. Watt

Algebraic Facilities

Basic Arithmetic Domains

The underlying **LISP/VM** system provides support for arbitrary precision integers and double-precision floating point numbers. In addition to these, **Scratchpad II** has implementations of **RationalNumber**, **BigFloat** and **GaloisField**. **BigFloat** is a port of the bigfloat package done by Tateaki Sasaki for the Reduce computer algebra system. It supports calculations done to a user specified precision and includes full precision computation of π, e and the elementary transcendental functions. **GaloisField** and **GaloisFieldExtension** provide support for finite field computations of prime and prime power order, respectively. The system makes no *a priori* restrictions on the size of the field with which it will deal.

Aggregates

Primitive constructors **List** and **Vector** support unlimited finite sequences. **DirectProduct** extends **Vector** to supply the usual direct product of rings, groups and fields. The user can thus create a vector space of a dimension n over any field F by invoking **DirectProduct(n,F)**. The additional primitive constructors **Record** and **Union** support heterogeneous aggregates, the former with selector names for each field, and the latter being a discriminated union of any finite set of constructors.

Polynomial Type Constructors

Prior experience with computer algebra systems has shown that no single polynomial representation is "best" for all algorithms, and so **Scratchpad II** provides the user with a choice among several. Dense and sparse univariate polynomials are supported and these are used to implement recursive multivariate polynomials. All sparse representations are extensions of **PolRing**, a very general polynomial constructor which permits the exponent domain to be any ordered abelian monoid. Thus "polynomials" with rational number exponents are easily created. By specializing the exponent domain to **DirectProduct** of a fixed number of copies of **NonNegativeInteger**, a distributed polynomial representation is created. The underlying system philosophy of making constructors as general as possible allows the creation of many complicated domains from a small number of very flexible components.

Domain Extensions

The **QuotientField** constructor creates a field of fractions from any underlying **IntegralDomain**. Greatest common divisors will be cancelled if a *gcd* operation is available. **FactoredRing** creates a domain which performs operations so as to preserve the factored form of an object as much as possible. This can be very helpful in maintaining or discovering structural properties in expressions. **Gaussian** gives the complex extension of any real ring.

RadicalExtension creates a domain which is closed under root extraction and performs operations by rationalizing denominators and reducing powers of radicals. In the particular case of **RationalRadicals**, where the arguments are positive rational numbers, a canonical form is generated by factoring the radicands into primes. **SimpleAlgebraicExtension** allows computations with algebraic extensions which cannot be expressed using radicals. **ElementaryFunction** supports rational operations and differentiation involving the elementary transcendental functions.

Matrix

The matrix domain supports rational operations along with functions ***determinant, inverse, nullSpace*** and ***rowEchelon***. There is an eigenvector package which calculates eigenvalues and eigenvectors (see "Algebra Snapshot: Eigenvalues and Eigenvectors" on page 8). The user has the option of requesting non-rational eigenvalues to be expressed using radicals if possible or in terms of a defining minimal polynomial.

Solve Packages

Real zeros of polynomials can be determined to any user specified accuracy. Single polynomial equations can be solved by factoring and using quadratic, cubic, and quartic formulae. Systems of linear equations can be solved yielding a particular solution and a basis for the null space. A Grobner basis package finds canonical bases for polynomial ideals and can be used to "triangularize" a system of polynomial equations.

Factorization

Scratchpad II currently supports factorization of univariate polynomials over finite fields, and multivariate polynomials over the rational numbers. Factorization of integers is performed using a modified Pollard rho algorithm.

Integration

Only rational function integration is currently available though there is no restriction on the denominator. The answer returned can involve the necessary algebraic extensions to the ground field.

Packages in Progress

- Factoring over algebraic extensions
- Solving systems of polynomial equations
- Integration of elementary transcendental and algebraic functions
- Power series domain and solutions of differential equations in power series.

Barry M. Trager

New Remote User Interface

Scratchpad II users will no longer need to maintain local copies of the software or the relatively large virtual machines formerly necessary to effectively use the system. *RSPAD*, a program which runs on a user's virtual machine, and its partner program *SPADCOMM* running on a dedicated service machine, connect a user's console to a remote **Scratchpad II** session. All **Scratchpad II** overhead is carried at the service machine's end, thereby allowing the remote user to run the computer algebra system from within a small virtual machine.

In addition to the usual **Scratchpad II** features, an *RSPAD* user may run up to 4 simultaneous sessions. *RSPAD* also permits a remote user to substitute command files for console input. All I/O between the *RSPAD* user and the service machine is queued, thereby allowing the user to type ahead. The service machine can thus receive and process entire command files while the *RSPAD* user performs other tasks, either in another **Scratchpad II** session or in CMS subset.

A version of *RSPAD* for UNIX users with access to an ETHERNET link to VM is currently being implemented.

Mohamed Mobarak

Algebra Snapshot: Eigenvalues and Eigenvectors

Each issue this section will be used to highlight a recently added package to the growing library of **Scratchpad II** algebra code. The initial article demonstrates a package written by Patrizia Gianni for computing and manipulating eigenvalues and eigenvectors.

We start by setting *a* equal to a 3 by 3 matrix of rational numbers. The "$" in the assignment tells the interpreter that the structure is to be considered an M RN, that is, a matrix of rational numbers. (In the absence of the "$-qualification", the interpreter will assume the structure is a list of lists of integers.)

```
a := [[1,2,1],[2,1,-2],[1,-2,4]]$M RN
```

$$(2)\quad \begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & -2 \\ 1 & -2 & 4 \end{bmatrix}$$

```
Type: M RN
```

The function ***eigenvectors*** returns a list of pairs of values and vectors. When the eigenvalue is rational, it is listed explicitly. Otherwise, the minimal polynomial for the eigenvalues is given, along with a parametric representation of the eigenvector. Once the roots of the minimal polynomial are computed, they can be substituted into the parametric form and the corresponding eigenvectors found.

```
eigenvectors a
```

$$(3)\quad [[5,[\begin{bmatrix} 0 \\ -\frac{1}{2} \\ 1 \end{bmatrix}]],\ [\%A^2 - \%A - 5,[\begin{bmatrix} \%A \\ 2 \\ 1 \end{bmatrix}]]]$$

```
Type: L [algrel: P[%A]RN,algvec: L M P[%A]RN]
```

Here the minimal polynomials are given in terms of a generated symbol *%A*.

By making use of the root finding functions in Scratchpad II, the function ***eigenSolve*** will attempt to explicitly compute the eigenvalues and eigenvectors.

```
eigenSolve a
```

(4)

$$[[(\frac{1}{2})^{\frac{1}{2}} + \frac{1}{2},[\begin{bmatrix} (\frac{1}{2})^{\frac{1}{2}} + \frac{1}{2} \\ 2 \\ 1 \end{bmatrix}]],$$

$$[(\frac{-1}{2})^{\frac{1}{2}} + \frac{1}{2},$$

$$[\begin{bmatrix} (\frac{-1}{2})^{\frac{1}{2}} + \frac{1}{2} \\ 2 \\ 1 \end{bmatrix}],\ [5,[\begin{bmatrix} 0 \\ -\frac{1}{2} \\ 1 \end{bmatrix}]]]$$

```
Type: L [vval: RE RF I,valvect: L M RE RF I]
```

To get a list of the eigenvalues, the function ***eigenvalues*** is employed.

```
eigenvalues a
```

$$(6)\quad [(\frac{-1}{2})^{\frac{1}{2}} + \frac{1}{2},\ (\frac{1}{2})^{\frac{1}{2}} + \frac{1}{2},5]$$

```
Type: L RE RF I
```

Given an explicit eigenvalue, ***eigenvector*** computes the eigenvectors corresponding to it (note: % denotes the current expression).

```
eigenvector(%.0,a)
```

$$(7)\quad [\begin{bmatrix} (\frac{-1}{2})^{\frac{1}{2}} + \frac{1}{2} \\ 2 \\ 1 \end{bmatrix}]$$

```
Type: L M RE RF I
```

If the eigenvectors of a matrix form a basis for the vector space, the function ***eigenmatrix*** will return a matrix containing these vectors. If the matrix does not admit such a basis, ***eigenmatrix*** returns failed.

```
eigenmatrix a
```

$$(8)\quad \begin{bmatrix} (\frac{1}{2})^{\frac{1}{2}} + \frac{1}{2} & (\frac{-1}{2})^{\frac{1}{2}} + \frac{1}{2} & 0 \\ 2 & 2 & -\frac{1}{2} \\ 1 & 1 & 1 \end{bmatrix}$$

```
Type: M RE RF I
```

```
b := [[-5,-2],[18,7]]$M I
```

$$(9)\quad \begin{bmatrix} -5 & -2 \\ 18 & 7 \end{bmatrix}$$

```
Type: M I
```

```
eigenmatrix b
```

(10) failed

```
Type: S
```

If a symmetric matrix admits a basis of orthonormal eigenvectors, ***orthonormalBasis*** will compute a list of these vectors.

```
c := [[1,2],[2,1]]$M I
```

$$(11)\quad \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$$

```
Type: M I
```

```
orthonormalBasis c

                 1                  1
               ---                ---
                 2                  2
        1     - 1        1     - 1
      (---)2  (-----)2   (---)2  (-----)2
        2        2         2        2
(12) [                ,                  ]
                 1                  1
               ---                ---
                 2                  2
        1        1         1        1
      (---)2   (---)2    (---)2   (---)2
        2        2         2        2

Type: L M RE RF I
```

Patrizia Gianni

Visitors to the Scratchpad II Group in 1985

Albrecht Fortenbacher
Graduate student, University of Karlsruhe, West Germany. Interpreter: 9/84-9/85.

Scott Morrison
Graduate student, EECS, University of California, Berkeley. Interpreter: 6/85-9/85.

Michael Lucks
An M.S. graduate in Computer Science from the University of Hawaii. Interface: 2/85-2/86.

Martin Brock
A recent M.S. graduate in Mathematics from M.I.T.. System: 5/85-7/86

Dr. James H. Davenport
School of Mathematics, University of Bath, Bath, England. Algebra and system: 7/85-8/85.

Dr. Julian A. Padget
School of Mathematics, University of Bath, Bath, England. Algebra and system: 7/85-9/85.

Professor Patrizia Gianni
Dipartimento di Matematica, Universita Di Pisa, Pisa, Italy. Algebra: 7/85-11/85.

Dr. Jean Della Dora
Institut IMAG, Grenoble, France, 9/85-10/85

Professor Moss E. Sweedler
Department of Mathematics, Cornell University, Ithaca, New York, Algebra: 9/85-6/86.

Mohamed Mobarak
Undergraduate student, Department of Computer Science, Cornell University, Ithaca, New York. System: 5/85-8/85.

1985 Demonstrations of Scratchpad II

March 12-15	1985 ACM Computer Science Conference, New Orleans, Louisiana.
April 1-3	EUROCAL '85, Linz, Austria.
August 18-23	IJCAI '85, Los Angeles, California.
August 26-30	AMS Summer Conference in Computational Number Theory, Humboldt State University, Arcata, California.