SACLIB 1.1 User's Guide[1]

Bruno Buchberger, George E. Collins, Mark J. Encarnación
Hoon Hong, Jeremy R. Johnson, Werner Krandick
Rüdiger Loos, Ana M. Mandache, Andreas Neubacher
Herbert Vielhaber

March 12, 1993

[1]RISC–Linz Report Series Technical Report Number 93-19 (Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz, Austria)

Abstract

This paper lists most of the algorithms provided by SACLIB and shows how to call them from C. There is also a brief explanation of the inner workings of the list processing and garbage collection facilities of SACLIB.

Contents

List of Figures

Chapter 1

Introduction

1.1 What is SACLIB?

SACLIB is a library of C programs for computer algebra derived from the SAC2 system. Hoon Hong was the main instigator. Sometime early in 1990 he proposed to translate SAC2 (which was written in the ALDES language) into C instead of Fortran (as it had been since 1976), and he quickly wrote the required translator. The results were rewarding in several ways. Hoon Hong, myself and Jeremy Johnson, working together at Ohio State University, observed a speedup by a factor of about two in most applications and the powerful debugging facilities associated with C became available.

Later that year Hoon finished his doctorate and moved to RISC, where Bruno Buchberger was writing a book on Gröbner bases and working on a set of programs to go with it. He found that SAC2 was the only computer algebra system in which he could write these programs without an unacceptable sacrifice in computational efficiency. It became apparent that for similar reasons other researchers would benefit greatly from the availability of a library of C programs derived from SAC2. Subsequently Bruno did much to promote and facilitate the preparation of the library for distribution.

Although the translated programs were correct, they needed to be reformatted for user consumption, a users guide was required, and we had compulsions to make some minor improvements. Jeremy Johnson made many recent improvements to the algebraic number algorithms and wrote the corresponding chapter of this Guide, among other things. Werner Krandick made improvements to the polynomial real root algorithms and wrote the corresponding chapter. Mark Encarnación wrote three chapters of the Guide and also converted the polynomial input and output algorithms to modern notation from the original "Fortran notation". Ana Mandache, Andreas Neubacher and Hoon Hong all toiled long hours editing and reformatting programs. Andreas deserves special recognition. He initiated the writing of the manual, wrote three chapters of the manual and two of the appendices, and did all the required system maintenance. To facilitate experimenting with the functions in the library, Herbert Vielhaber implemented ISAC, the interactive shell for SACLIB. He also wrote the corresponding appendix of the manual.

Besides the above it would be unthinkable not to mention, collectively, all of my former doctoral students, who contributed to the development of the SAC2 algorithms and the research on which they were founded over a period of 26 years. During the last 20 of those years Rüdiger Loos was a frequent collaborator. He proposed creation of an "ALgorithm DEScription language" for SAC1, the predecessor of SAC2, and wrote an ALDES-to-Fortran translator.

This initial version of SACLIB is just the beginning of what is to come. We know how to improve several of the programs in the current system and we will do it for subsequent versions. Some basic functionalities are largely undeveloped in the currrent system (e.g.

linear algebra) but they will be supplied in subsequent versions. Some more advanced functionalities (e.g. polynomial complex roots and quantifier elimination) are nearly ready and will be forthcoming soon. Also we expect that users of the system will write programs based on the ones we distribute and offer them to other users.

George E. Collins

1.2 About this Guide

The main goal in writing this guide was to enable the reader to quickly discover whether SACLIB provides a function for a given problem. The structure of the paper should facilitate searching for a function in the following way:

- Every chapter deals with functions operating over a certain domain (lists, numbers, polynomials, etc.) or with functions solving certain problems (GCD computation, factorization, real root calculation, etc.).
- Some chapters are split into sections covering more specific topics (integer arithmetic, rational number arithmetic, integral polynomial arithmetic, etc.)
- Inside a section, functions are divided into various areas (basic arithmetic, predicates, input/output, etc.).
- Inside these areas, closely related functions (a function and its inverse, functions solving essentially the same problem, a function and its auxiliary routines, etc.) are grouped.

This partitioning was done on a completely subjective basis. The intention always was that the neophyte user should be able to pinpoint a desired function by using simple heuristics. This approach may certainly fail in some cases, but with at most 50 functions per section browsing them sequentially should always succeed in an acceptable amount of time.

Another rather subjectively designed feature is the function descriptions. The lists were generated automatically from the headers of the SACLIB source files. For some functions additional remarks were added in *emphasized* type style.

Readers who want to use SACLIB functions in their C programs should read Appendix A, which describes how initialization and cleanup are done, which files have to be `#include`d, etc. A detailed description of the input/output specifications of a given function can be found in the comment block at the beginning of the corresponding source file. Read the "Addendum to the SACLIB User's Guide" for information on how to access these.

Those who want to know more about the inner workings of SACLIB should refer to Appendix C which gives an overview of the internal representation of lists, the garbage collector and the constants and global variables used internally. Descriptions of the high level data structures used for implementing the elements of domains like integers, polynomials, etc. can be found at the beginnings of the corresponding sections.

1.3 SACLIB Maintenance

The recommended way for reporting problems with SACLIB is sending e-mail to the maintenance account

`saclib@risc.uni-linz.ac.at`

or mail to

`SACLIB Maintenance`

`Research Institute for Symbolic Computation`
`Johannes Kepler University`
`4020 Linz`
`Austria`

Messages which might interest a greater audience should be sent to the mailing list

`saclib-l@risc.uni-linz.ac.at`

This list can be subscribed by sending a message with the body

subscribe saclib-l <first name> <last name>

to `listserv@risc.uni-linz.ac.at`.

Note that SACLIB is not sold for profit[1]. Therefore do not expect prompt service and extensive support. Nevertheless, SACLIB is continuously maintained and extended, so do not hesitate getting in contact with us.

[1] SACLIB maintenance is sponsored by the Research Institute for Symbolic Computation.

Chapter 2

List Processing

2.1 Mathematical Preliminaries

Let A be a finite domain and let C be the closure of A under the operation of finite sequence formation. Then

- the elements of C are called *objects*,
- the elements of A are called *atoms*, and
- the elements of $C \backslash A$ are called *lists*.

Note that the atom a and the list (a) containing a as its only element are distinct objects. Furthermore, the set of lists also encompasses the empty sequence, which we call the empty list.

2.2 Purpose

Lists are the basis for nearly all SACLIB internal representations of elements of domains like the integers, polynomials, algebraic numbers, etc. The SACLIB list processing package implements the abstract concept of lists described above.

2.3 Definitions of Terms

atom An integer `a` such that $-$`BETA` $<$ `a` $<$ `BETA`.

list (handle) An integer `L` such that `BETA` $\leq$ `L` $<$ `BETAp`, where `BETA` and `BETAp` are positive integer constants[1]. `L` is a reference to the memory location of the first cell of the list L.

The term *list* is used to denote the SACLIB internal representation of an element of $C \backslash A$ as given in Section 2.1. If emphasis is on the reference to memory, the term *list handle* is used.

(list) cell The memory space used to store a (reference to a) single list element and bookkeeping information needed to combine several cells into a list.

(list) element If L is the list $(l_1, l_2, \ldots, l_n)$, then l_1 is its *first element*, l_2 is its *second element*, etc.

empty list A list containing no elements, represented by the constant `NIL`.

[1] See Section C.2 for more information on `BETA` and `BETAp`.

object A term denoting both atoms and lists.

composition of an object l and a list $(l_1, l_2, \dots, l_n)$ is the list $(l, l_1, l_2, \dots, l_n)$.

reductum of a list $(l_1, l_2, \dots, l_n)$ is the list $(l_2, l_3, \dots, l_n)$. The reductum of the empty list is undefined.

concatenation of lists $(l_1, l_2, \dots, l_n)$ and $(m_1, m_2, \dots, m_k)$ is the list $(l_1, \dots, l_n, m_1, \dots, m_k)$.

inverse of a list $(l_1, l_2, \dots, l_n)$ is the list $(l_n, l_{n-1}, \dots, l_1)$.

length of a list $(l_1, l_2, \dots, l_n)$ is n. The length of the empty list is 0.

extent The number of cells used by an object. More precisely:

- $\texttt{EXTENT}(a) = 0$ if a is an atom.
- $\texttt{EXTENT(NIL)} = 0$.
- $\texttt{EXTENT}(L) = 1 + \texttt{EXTENT}(l_1) + \texttt{EXTENT}((l_2, \dots, l_n))$, where L is the non-empty list $(l_1, l_2, \dots, l_n)$.

order The depth of an object. More precisely:

- $\texttt{ORDER}(a) = 0$ if a is an atom.
- $\texttt{ORDER(NIL)} = 1$.
- $\texttt{ORDER}(L) = \texttt{MAX}(\texttt{ORDER}(l_1) + 1, \texttt{ORDER}((l_2, \dots, l_n)))$, where L is the non-empty list $(l_1, l_2, \dots, l_n)$.

side effects When a function modifies the content of one or more cells of the input list(s), it is said to cause *side effects*. This is always noted in the function specifications.

destructive An operation on lists causing side effects is called *destructive*.

(unordered) set An (unordered) list of atoms.

2.4 Functions

Constructors:

`M <- COMP(a,L)` Composition. *Prefixes an object to a list.*

`M <- COMP2(a,b,L)` Composition 2. *Prefixes 2 objects to a list.*

`M <- COMP3(a1,a2,a3,L)` Composition 3. *Prefixes 3 objects to a list.*

`M <- COMP4(a1,a2,a3,a4,L)` Composition 4. *Prefixes 4 objects to a list.*

`L <- LIST1(a)` List, 1 element. *Builds a list from one object.*

`L <- LIST2(a,b)` List, 2 elements. *Builds a list from 2 objects.*

`L <- LIST3(a1,a2,a3)` List, 3 elements. *Builds a list from 3 objects.*

`L <- LIST4(a1,a2,a3,a4)` List, 4 elements. *Builds a list from 4 objects.*

`L <- LIST5(a1,a2,a3,a4,a5)` List, 5 elements. *Builds a list from 5 objects.*

`L <- LIST10(a1,a2,a3,a4,a5,a6,a7,a8,a9,a10)` List, 10 elements. *Builds a list from 10 objects.*

Selectors:

`ADV(L; a,Lp)` Advance. *Returns the first element and the reductum of a list.*

`ADV2(L; a,b,Lp)` Advance 2. *Returns the first 2 elements and the 2nd reductum of a list.*

`ADV3(L; a1,a2,a3,Lp)` Advance 3. *Returns the first 3 elements and the 3rd reductum of a list.*

`ADV4(L; a1,a2,a3,a4,Lp)` Advance 4. *Returns the first 4 elements and the 4th reductum of a list.*

`AADV(L; a,Lp)` Arithmetic advance. *Returns the first element and the reductum of a non-empty list, returns 0 as the first element if the list is empty.*

`a <- FIRST(L)` First. *Returns the first element of a list.*

`FIRST2(L; a,b)` First 2. *Returns the first 2 elements of a list.*

`FIRST3(L; a1,a2,a3)` First 3. *Returns the first 3 elements of a list.*

`FIRST4(L; a1,a2,a3,a4)` First 4. *Returns the first 4 elements of a list.*

`a <- SECOND(L)` Second. *Returns the 2nd element of a list.*

`a <- THIRD(L)` Third. *Returns the 3rd element of a list.*

`a <- FOURTH(L)` Fourth. *Returns the 4th element of a list.*

`Lp <- LASTCELL(L)` Last cell. *Returns the list handle of the last cell of a list.*

`a <- LELTI(A,i)` List element. *Returns the i-th element of a list.*

`Lp <- RED(L)` Reductum. *Returns the reductum of a list.*

`Lp <- RED2(L)` Reductum 2. *Returns the 2nd reductum of a list.*

`M <- RED3(L)` Reductum 3. *Returns the 3rd reductum of a list.*

`M <- RED4(L)` Reductum 4. *Returns the 4th reductum of a list.*

`B <- REDI(A,i)` Reductum. *Returns the i-th reductum of a list.*

Information and Predicates:

`t <- ISOBJECT(a)` Test for object. *Tests whether the argument represents an object.*

`t <- ISATOM(a)` Test for atom. *Tests whether the argument represents an atom.*

`t <- ISLIST(a)` Test for list. *Tests whether the argument represents a list.*

`t <- ISNIL(L)` Test for empty list. *Tests whether the argument represents the empty list.*

`t <- EQUAL(a,b)` Equal. *Tests whether two objects are equal.*

`t <- MEMBER(a,L)` Membership test. *Tests whether an object is an element of a list.*

`i <- LSRCH(a,A)` List search. *Returns the index of an object in a list.*

`n <- EXTENT(a)` Extent.

`n <- LENGTH(L)` Length.

`n <- ORDER(a)` Order.

Concatenation:

`L <- CCONC(L1,L2)` Constructive concatenation. *Builds a list* $(l_1, \ldots, l_m, l_{m+1}, \ldots, l_n)$ *from lists* $(l_1, \ldots, l_m)$ *and* $(l_{m+1}, \ldots, l_n)$.

`L <- CONC(L1,L2)` Concatenation. *Concatenates two lists destructively.*

`M <- LCONC(L)` List concatenation. *Concatenates the elements of a list of lists destructively.*

Inversion:

`M <- CINV(L)` Constructive inverse. *Builds a list containing the elements of the argument in inverse order.*

`M <- INV(L)` Inverse. *Inverts a list destructively.*

Insertion:

`LINS(a,L)` List insertion. *Inserts an object after the first element of a list.*

`L <- LEINST(A,i,a)` List element insertion. *Inserts an object after the i-th element of a list.*

`Lp <- SUFFIX(L,b)` Suffix. *Appends an object after the last element of a list.*

`B <- LINSRT(a,A)` List insertion. *Inserts an atom into a sorted list of atoms.*

Combinatorial:

`M <- LEROT(L,i,j)` List element rotation. *Rotates some consecutive elements of a list.*

`Lp <- LPERM(L,P)` List permute. *Permutes the elements of a list.*

`Pp <- PERMCY(P)` Permutation, cyclic. *Rotates a list to the left.*

`L <- PERMR(n)` Permutation, random. *Builds a list of the first n integers in random order.*

`B <- LEXNEX(A)` Lexicographically next. *Computes the lexicographical successor of a permutation.*

Set Operations:

`b <- SEQUAL(A,B)` Set equality. *Tests whether two sets represented as unordered redundant lists are equal.*

`C <- SDIFF(A,B)` Set difference.

`B <- SFCS(A)` Set from characteristic set.

`C <- SINTER(A,B)` Set intersection.

`C <- SUNION(A,B)` Set union.

`C <- USDIFF(A,B)` Unordered set difference.

`C <- USINT(A,B)` Unordered set intersection.

`C <- USUN(A,B)` Unordered set union.

Sorting:

`M <- LBIBMS(L)` List of BETA-integers bubble-merge sort. *Sorts a list of atoms into non-descending order.*

`LBIBS(L)` List of BETA-integers bubble sort. *Sorts a list of atoms into non-descending order.*

`L <- LBIM(L1,L2)` List of BETA-integers merge. *Merges two sorted lists of atoms.*

`B <- LINSRT(a,A)` List insertion. *Inserts an atom into a sorted list of atoms.*

`C <- LMERGE(A,B)` List merge. *Constructively merges two lists avoiding duplicate elements.*

Input/Output:

`A <- AREAD()` Atom read.

`AWRITE(A)` Atom write.

`L <- LREAD()` List read.

`LWRITE(L)` List write.

`B <- OREAD()` Object read.

`OWRITE(B)` Object write.

Miscellaneous:

`C <- PAIR(A,B)` Pair. *Builds a list by interleaving the elements of two lists.*

`SFIRST(L,a)` Set first element. *Sets the first element of a list.*

`SLELTI(A,i,a)` Set list element. *Sets the i-th element of a list.*

`SRED(L,Lp)` Set reductum. *Sets the reductum of a list.*

Chapter 3

Arithmetic

3.1 Introduction

3.1.1 Purpose

The SACLIB arithmetic packages support computations with integers, modular numbers, and rational numbers whose sizes are only bounded by the amount of memory available.

3.1.2 Definitions of Terms

integer Integers to be entered into SACLIB must be of the following *external form.*

- <digit sequence> or
- + <digit sequence> or
- − <digit sequence> ,

where <digit sequence> designates any non-empty word over the alphabet $0, 1, ..., 9$. Note that there is no blank between the optional sign and the digit sequence; also note that leading zeros are allowed. Inputs of this form are interpreted in the usual way as decimal numbers.

SACLIB outputs the *canonical external representation* of integers. This is the integer in external form with both positive sign and leading zeros digits supressed.

The *internal representation* `I` of a number $n \in \mathbf{Z}$ is defined as follows:

- If $-\texttt{BETA} < n < \texttt{BETA}$ then `I` is the atom whose value is n.
- If $n \leq -\texttt{BETA}$ or $\texttt{BETA} \leq n$ then `I` is the list $(d_0, d_1, \ldots, d_k)$ with $d_k \neq 0$, $d_i \leq 0$ if $n < 0$ and $0 \leq d_i$ if $0 < n$ for $0 \leq i \leq k$, and $n = \sum_{i=0}^{k} d_i \texttt{BETA}^i$.

digit, `BETA`-digit, `BETA`-integer An atom n with $-\texttt{BETA} < n < \texttt{BETA}$.

`GAMMA`-digit, `GAMMA`-integer An atom n with $-\gamma < n < \gamma$, where γ is the largest integer which fits into a `Word`[1]. (E.g. if the size of a `Word` is 32 bits, then $\gamma = 2^{31} - 1$.)

modular digit An atom n with $0 \leq n < m$, where m is a positive `BETA`-digit.

modular integer An integer n with $0 \leq n < m$, where m is a positive integer.

symmetric modular An integer n with $-\lfloor \frac{m}{2} \rfloor + 1 \leq n \leq \lfloor \frac{m}{2} \rfloor$, where m is a positive integer. In the input/output specifications of the corresponding algorithms these are denoted as elements of `Z'_M`, as opposed to the notation `Z_M`, which is used for (non-symmetric) modular integers.

[1] See Section C.2 for details on the type `Word`.

rational number Rational numbers to be entered into SACLIB must be of the following *external form*.

- <integer N > or
- <integer N > / <integer D > ,

where <integer N > and <integer D > are external forms of relatively prime integers N and D, such that $D > 0$. Note that no blanks are permitted immediately before and after the /. Inputs of this form are interpreted in the usual way as rational numbers with numerator N and denominator D.

SACLIB outputs the *canonical external representation* of rational numbers $r \in \mathbf{Q}$. If $r \in \mathbf{Z}$, the canonical external representation of r is the canonical external representation of the integer r. Otherwise there are unique integers N and D such that $r = \frac{N}{D}$, $D > 1$, and $\gcd(N, D) = 1$. The canonical external representation of r in this case is the canonical external representation of the integer N followed by / followed by the canonical external representation of the integer D.

The *internal representation* `R` of a number $r \in \mathbf{Q}$ is defined as follows:

- If $r = 0$ then `R` is the `BETA`-digit 0.
- Otherwise, `R` is the list (`N`,`D`), where `N` and `D` are the internal representations of the numerator and the denominator of r, i.e. the unique integers n and d such that $r = \frac{n}{d}$, $d > 0$, and $\gcd(n, d) = 1$.

ceiling of a number r is the smallest integer n such that $r \leq n$.

floor of a number r is the largest integer n such that $n \leq r$.

positive n is positive if $0 < n$.

non-negative n is non-negative if $0 \leq n$.

non-positive n is non-positive if $n \leq 0$.

negative n is negative if $n < 0$.

3.2 Integer Arithmetic

Basic Arithmetic:

`C <- ISUM(A,B)` Integer sum.

`C <- IDIF(A,B)` Integer difference.

`B <- INEG(A)` Integer negation.

`C <- IPROD(A,B)` Integer product.

`C <- IDPR(A,b)` Integer-digit product.

`DPR(a,b; c,d)` Digit Product.

`C <- IPRODK(A,B)` Integer product, Karatsuba algorithm.

`C <- IQ(A,B)` Integer quotient.

`C <- IDQ(A,b)` Integer-digit quotient.

`IQR(A,B; Q,R)` Integer quotient and remainder.

`IDQR(A,b; Q,r)` Integer-digit quotient and remainder.

`DQR(a1,a0,b; q,r)` Digit quotient and remainder.

`C <- IREM(A,B)` Integer remainder.

`r <- IDREM(A,b)` Integer-digit remainder.

`c <- IMAX(a,b)` Integer maximum. *Returns the greater of two integers.*

`c <- IMIN(a,b)` Integer minimum. *Returns the smaller of two integers.*

`s <- ISIGNF(A)` Integer sign function.

`B <- IABSF(A)` Integer absolute value function.

`s <- ICOMP(A,B)` Integer comparison. *Compares two integers and returns* $-1, 0,$ *and* $+1$ *in case of* $<, =,$ *and* $>$*, respectively.*

`t <- IEVEN(A)` Integer even. *Tests whether the argument is even.*

`t <- IODD(A)` Integer odd. *Tests whether the argument is odd.*

Exponentiation:

`B <- IEXP(A,n)` Integer exponentiation.

`IROOT(A,n; B,t)` Integer root.

`ISQRT(A; B,t)` Integer square root.

`DSQRTF(a; b,t)` Digit square root function.

`IPOWER(A,L; B,n)` Integer power. *If the argument can be expressed as* b^n*, such integers* b *and* n *are computed.*

Greatest Common Divisor:

`C <- IGCD(A,B)` Integer greatest common divisor.

`c <- DGCD(a,b)` Digit greatest common divisor.

`IGCDCF(A,B; C,Ab,Bb)` Integer greatest common divisor and cofactors.

`IEGCD(a,b; c,u1,v1)` Integer extended greatest common divisor algorithm.

`DEGCD(a,b; c,u,v)` Digit extended greatest common divisor.

`IDEGCD(a,b; c,u1,v1,u2,v2)` Integer doubly extended greatest common divisor algorithm.

`IHEGCD(A,B; C,V)` Integer half-extended greatest common divisor.

`C <- ILCM(A,B)` Integer least common multiple.

Factorization:

`F <- IFACT(n)` Integer factorization.

`s <- ISPT(m,mp,F)` Integer selfridge primality test. *Returns 1 if the argument is prime, -1 if it is not prime, and 0 if the primality could not be determined.*

`ILPDS(n,a,b; p,np)` Integer large prime divisor search.

`IMPDS(n,a,b; p,q)` Integer medium prime divisor search.

`ISPD(n; F,m)` Integer small prime divisors.

Prime Number Generation:

`L <- DPGEN(m,k)` Digit prime generator.

Random Number Generation:

`A <- IRAND(n)` Integer, random.

`a <- DRAN()` Digit, random.

`a <- DRANN()` Digit, random non-negative.

Combinatorial:

`A <- IFACTL(n)` Integer factorial.

`A <- IBCOEF(n,k)` Integer binomial coefficient. *Returns* $\binom{n}{k}$.

`B <- IBCIND(A,n,k)` Integer binomial coefficient induction. *Returns* $\binom{n}{k+1}$ *given* n, k, *and* $\binom{n}{k}$.

`A <- IBCPS(n,k)` Integer binomial coefficient partial sum. *Returns* $\sum_{i=0}^{k} \binom{n}{i}$.

Binary Arithmetic:

`n <- ILOG2(A)` Integer logarithm, base 2. *Returns 1 + (the floor of the base 2 logarithm of the argument).*

`n <- DLOG2(a)` Digit logarithm, base 2.

`IFCL2(a; m,n)` Integer, floor and ceiling, logarithm, base 2. *Returns the floor and the ceiling of the base 2 logarithm of the argument.*

`B <- IMP2(A,h)` Integer multiplication by power of 2. *Multiplies the argument by a non-negative power of 2.*

`B <- IDP2(A,k)` Integer division by power of 2. *Divides the argument by a non-negative power of 2.*

`B <- ITRUNC(A,n)` Integer truncation. *Divides the argument by a positive or negative power of 2.*

`n <- IORD2(a)` Integer, order of 2. *Returns the largest* n *such that* 2^n *divides the argument.*

Boolean:

`c <- DAND(a,b)` Digit and. *Returns the bit-wise* $\wedge$ *of two digits.*

`c <- DOR(a,b)` Digit or. *Returns the bit-wise* $\vee$ *of two digits.*

`b <- DNOT(a)` Digit not. *Returns the bit-wise* $\neg$ *of a digit.*

`c <- DNIMP(a,b)` Digit non-implication. *Returns the bit-wise* $\neg(a \Rightarrow b)$ *of digits* a *and* b.

Input/Output:

`A <- IREAD()` Integer read.

`IWRITE(A)` Integer write.

`ILWRITE(L)` Integer list write. *Writes a list of integers in the form* $(n_1, n_2, \ldots, n_k)$ *to the output stream.*

Auxiliary Functions:

`C <- ISSUM(n,L)` Integer shifted sum. *Computes* $\sum_{i=0}^{k} C_i \mathtt{BETA}^{in}$ *given* n *and the* C_i.

`ISEG(A,n; A1,A0)` Integer segmentation. *Splits an integer at a* `BETA`*-digit boundary.*

`C <- IDIPR2(A,B,a,b)` Integer digit inner product, length 2. *Computes* $Aa + Bb$ *for integers* A, B *and* `BETA`*-digits* a, b.

`C <- ILCOMB(A,B,u,v)` Integer linear combination. *Computes* $Aa + Bb$ *for integers* A, B *and* `BETA`*-digits* a, b *with* $Aa + Bb \geq 0$.

`DPCC(a1,a2; u,up,v,vp)` Digit partial cosequence calculation.

`AADV(L; a,Lp)` Arithmetic advance. *Returns the first element and the reductum of a non-empty list, returns 0 as the first element if the list is empty.*

3.3 Modular Number Arithmetic

3.3.1 Modular Digit Arithmetic

Basic Arithmetic:

`c <- MDSUM(m,a,b)` Modular digit sum.

`c <- MDDIF(m,a,b)` Modular digit difference.

`b <- MDNEG(m,a)` Modular digit negative.

`c <- MDPROD(m,a,b)` Modular digit product.

`c <- MDQ(m,a,b)` Modular digit quotient.

`b <- MDINV(m,a)` Modular digit inverse.

`b <- MDEXP(m,a,n)` Modular digit exponentiation.

Chinese Remainder Algorithm:

`a <- MDCRA(m1,m2,mp1,a1,a2)` Modular digit chinese remainder algorithm.

`L <- MDLCRA(m1,m2,L1,L2)` Modular digit list chinese remainder algorithm.

`b <- MDHOM(m,A)` Modular digit homomorphism. *Computes* $n \bmod m$.

Random Number Generation:

`a <- MDRAN(m)` Modular digit, random.

3.3.2 Modular Integer Arithmetic

Basic Arithmetic:

`C <- MISUM(M,A,B)` Modular integer sum.

`C <- MIDIF(M,A,B)` Modular integer difference.

`B <- MINEG(M,A)` Modular integer negation.

`C <- MIPROD(M,A,B)` Modular integer product.

`C <- MIQ(M,A,B)` Modular integer quotient.

`B <- MIINV(M,A)` Modular integer inverse.

`B <- MIEXP(M,A,N)` Modular integer exponentiation.

Chinese Remainder Algorithm:

`As <- MIDCRA(M,m,mp,A,a)` Modular integer digit chinese remainder algorithm.

`As <- MIHOM(M,A)` Modular integer homomorphism. *Computes* $n \bmod m$.

Random Number Generation:

`R <- MIRAN(M)` Modular integer, random.

Conversion:

`B <- SMFMI(M,A)` Symmetric modular from modular integer. *Computes the isomorphism from* $\mathbf{Z}_m$ *to* $\{-\lfloor\frac{m}{2}\rfloor+1,\ldots,\lfloor\frac{m}{2}\rfloor\}$.

3.4 Rational Number Arithmetic

Basic Arithmetic:

`T <- RNSUM(R,S)` Rational number sum.

`T <- RNDIF(R,S)` Rational number difference.

`S <- RNNEG(R)` Rational number negative.

`T <- RNPROD(R,S)` Rational number product.

`T <- RNQ(R,S)` Rational number quotient.

`S <- RNINV(R)` Rational number inverse.

`s <- RNSIGN(R)` Rational number sign.

`S <- RNABS(R)` Rational number absolute value.

`t <- RNCOMP(R,S)` Rational number comparison.

`c <- RNMIN(a,b)` Rational number min.

`c <- RNMAX(a,b)` Rational number max.

Constructors:

`R <- RNINT(A)` Rational number from integer. *Returns* $\frac{n}{1}$ *given an integer* n.

`R <- RNRED(A,B)` Rational number reduction to lowest terms. *Returns* $\frac{n}{d}$ *given two integers* n *and* d *with* $d \neq 0$.

Selectors:

`a <- RNNUM(R)` Rational number numerator.

`b <- RNDEN(R)` Rational number denominator.

Random Number Generation:

`R <- RNRAND(n)` Rational number, random.

Input/Output:

`R <- RNREAD()` Rational number read.

`RNWRITE(R)` Rational number write.

`RNDWRITE(R,n)` Rational number decimal write. *Approximates a rational number by a decimal fraction with a given accuracy and writes the approximation to the output stream.*

Miscellaneous:

`a <- RNCEIL(r)` Rational number, ceiling of.

`a <- RNFLOR(r)` Rational number, floor of.

`RNBCR(A,B; M,N,k)` Rational number binary common representation.

`RNFCL2(a; m,n)` Rational number floor and ceiling of logarithm, base 2.

`r <- RNP2(k)` Rational number power of 2. *Computes* 2^n *given a* `GAMMA`*-digit* n.

Chapter 4

Polynomial Arithmetic

4.1 Introduction

4.1.1 Purpose

The SACLIB polynomial arithmetic packages provide functions doing computations with multivariate polynomials over domains implemented by the SACLIB arithmetic packages.

Except for the functions listed in Section 4.7 and various conversion functions, only the *sparse recursive* representation is used.

4.1.2 Definitions of Terms

sparse recursive representation A polynomial $p \in \mathbf{D}[x_1, \ldots, x_r]$ is interpreted as an element of $(\ldots(\mathbf{D}[x_1])\ldots)[x_r]$, for some domain $\mathbf{D}$. The SACLIB *sparse recursive representation* `P` of a polynomial $p = \sum_{i=1}^{n} p_i x_r^{e_i}$ with $e_1 > \ldots > e_n$, $p_i \in (\ldots(\mathbf{D}[x_1])\ldots)[x_{r-1}]$, and $p_i \neq 0$ is defined recursively as follows:

- If $p = 0$ then `P` is the `BETA`-digit 0.
- If $r = 0$, then p is in $\mathbf{D}$ and its representation `P` is the representation of elements of the domain $\mathbf{D}$.
- If $r > 0$, then `P` is the list $(e_1, \mathtt{P}_1, \ldots, e_n, \mathtt{P}_n)$ where the e_i are `BETA`-digits and each $\mathtt{P}_i$ is the representation of p_i.

sparse distributive representation A polynomial $p \in \mathbf{D}[x_1, \ldots, x_r]$ is interpreted as $p = \sum_{i=1}^{n} d_i x^{e_i}$, where $d_i \in \mathbf{D}$, $d_i \neq 0$, and x^{e_i} stands for $x_1^{e_{i,1}} x_2^{e_{i,2}} \cdots x_r^{e_{i,r}}$ with $e_{i,j} \geq 0$. Furthermore, we assume that $e_1 > e_2 > \ldots > e_n$, where $e_k > e_i$ iff there exists a $\hat{\jmath}$ such that $e_{k,j} = e_{i,j}$ for $\hat{\jmath} < j \leq r$ and $e_{k,\hat{\jmath}} > e_{i,\hat{\jmath}}$.

The *sparse distributive representation* `P` of such a polynomial p is the list $(\mathtt{D}_1, \mathtt{E}_1, \mathtt{D}_2, \mathtt{E}_2, \ldots, \mathtt{D}_n, \mathtt{E}_n)$, where $\mathtt{D}_i$ is the SACLIB internal representation of d_i and $\mathtt{E}_i$ is the list $(e_{i,r}, e_{i,r-1}, \ldots, e_{i,1})$ with $e_{i,j}$ being `BETA`-digits.

As always in SACLIB, $\mathtt{P} = 0$ if $p = 0$.

dense recursive representation A polynomial $p \in \mathbf{D}[x_1, \ldots, x_r]$ is interpreted as an element of $(\ldots(\mathbf{D}[x_1])\ldots)[x_r]$, for some domain $\mathbf{D}$. The *dense recursive representation* `P` of a polynomial $p = \sum_{i=0}^{n} p_i x_r^i$ with $p_i \in (\ldots(\mathbf{D}[x_1])\ldots)[x_{r-1}]$ is defined recursively as follows:

- If $p = 0$ then `P` is the `BETA`-digit 0.
- If $r = 0$, then p is in $\mathbf{D}$ and its representation `P` is the representation of elements of the domain $\mathbf{D}$.

- If $r > 0$, then P is the list $(n, \mathtt{P}_n, \mathtt{P}_{n-1}, \ldots, \mathtt{P}_0)$ where the n is a BETA-digit and each $\mathtt{P}_i$ is the representation of p_i.

polynomial If this term appears in the parameter specifications of a function, this denotes a polynomial in the sparse recursive representation. Otherwise, it is used to denote a polynomial in arbitrary representation.

base domain, base ring If p is an element of $\mathbf{D}[x_1, \ldots, x_r]$, $\mathbf{D}$ is its base domain.

integral polynomial A polynomial whose base domain is $\mathbf{Z}$.

modular polynomial A polynomial whose base domain is $\mathbf{Z}_m$ with m a prime positive BETA-digit.

modular integral polynomial A polynomial whose base domain is $\mathbf{Z}_m$ with m a positive integer.

rational polynomial A polynomial whose base domain is $\mathbf{Q}$.

main variable of a polynomial in $\mathbf{D}[x_1, \ldots, x_r]$ is x_r.

degree The degree of a polynomial w.r.t. a given variable is the highest power of this variable appearing with non-zero coefficient in the polynomial. If no variable is specified, the degree is computed w.r.t. the main variable.

order The order of a polynomial $p = \sum_{i=0}^{n} p_i x_r^i$ is the smallest $k \geq 0$ such that $p_k \neq 0$.

constant polynomial A polynomial of degree 0 in every variable.

leading term of a polynomial is a polynomial equal to the term of highest degree w.r.t. the main variable.

reductum of a polynomial is the polynomial minus its leading term.

leading coefficient The leading coefficient of a polynomial is the coefficient of its leading term.

leading base coefficient An element of the base domain equal to the coefficient of the leading power product of a polynomial where the ordering on the power products is the lexicographic ordering with $x_1 < \cdots < x_r$.

trailing base coefficient An element of the base domain equal to the coefficient of the smallest power product of a polynomial where the ordering on the power products is the lexicographic ordering with $x_1 < \cdots < x_r$.

monic polynomial A polynomial, the leading coefficient of which is 1.

positive polynomial A polynomial, the leading base coefficient of which is positive.

sign An integer equal to 1 if the leading base coefficient of the polynomial is positive, -1 otherwise.

absolute value of a polynomial p is the positive polynomial q such that $p = \mathrm{sign}(p) \cdot q$.

content of a polynomial p is equal to the absolute value of the greatest common divisor of the coefficients of p.

integer content of an integral polynomial is an integer equal to the positive greatest common divisor of the integer coefficients of each power product of the polynomial.

primitive polynomial A polynomial, the content of which is 1.

squarefree polynomial A polynomial p is squarefree if each factor occurs only once. In other words, if $p = p_1^{e_1} \cdots p_k^{e_k}$ is a complete factorization of p then each of the e_i is equal to 1.

squarefree factorization The squarefree factorization of p is $p_1^{e_1} \cdots p_k^{e_k}$ where $1 \leq e_1 < \cdots < e_k$ and each of the p_i is a positive squarefree polynomial of positive degree. Note that if p is squarefree then p^1 is the squarefree factorization of p.

variable (name) A list $(c_1, \ldots, c_k)$, where the c_i are C characters. Example: the name "fubar" would be represented by the character list ('f','u','b','a','r').

list of variables A list $(n_1, \ldots, n_r)$ giving the names of the corresponding variables of an r-variate polynomial for input and output.

4.2 Polynomial Input and Output

In this section we will describe the polynomial input and output routines that are available in SACLIB. Before proceeding further, the reader should be familiar with the internal representations of polynomials which are discussed in Section 4.1.2.

4.2.1 Recursive polynomials over Z

The *external canonical representation* of sparse *recursive* polynomials over **Z** is defined by the following rules. First of all, each polynomial is enclosed in parentheses. A term is represented by the coefficient immediately followed by the variable (no space nor '*' in between). The coefficients $+1$ and -1 are suppressed unless the exponent of the variable is 0 in which case the variable is suppressed. The caret '^' is used to indicate exponentiation. Exponents with the value 1 are suppressed and if a variable has the exponent 0 then the variable is suppressed. These rules apply recursively to the coefficients which may themselves be polynomials. A few examples are in order.

recursive polynomial	external canonical form
$-x^4 + 2x^3 - x + 3$	`(-x^4+2x^3-x+3)`
$(x^2 + 1)y^3 + (x + 8)y - 5$	`((x^2+1)y^3+(x+8)y+(-5))`
$-(x^2 - 4)y^4 + y^2 - y - x$	`((-x^2+4)y^4+(1)y^2+(-1)y+(-x))`

Note that a constant polynomial in r variables will be enclosed in r sets of parentheses. For example, the constant polynomial 2 in 3 variables will be represented in external canonical form as `(((2)))`.

The algorithm `IPREAD` reads an r-variate recursive polynomial over **Z** in external canonical form from the input stream. The polynomial that is read is stored in internal canonical form and the number of variables is also recorded. The variables are *not* stored. Integer coefficients may be of arbitrary length but exponents must be `BETA`-digits. Since no sorting is performed on the terms, they must be given in order of descending degree. This is an important remark since almost all algorithms that manipulate polynomials require that the terms be ordered and violating this rule will undoubtedly cause incorrect results to be computed and may even crash the system. Another important remark is that terms whose coefficients are 0 should not be given as these terms will be stored and may cause problems, for example in equality testing.

Although `IPREAD` is happiest when a polynomial is given in external canonical form as exemplified by the previous examples, the user is allowed some freedom. An arbitrary number of spaces may interspersed between the coefficients, the variables, the exponents and the symbols '+', '-' and '^'. Spaces may not be inserted within a variable nor within an integer. Coefficients with magnitude 1 as well as the exponents 0 and 1 may be explicitly given. Thus, for example, `((x ^ 2+1) y^3+(1x+8) y^1-(5x^0) y^0)` is perfectly valid and is equivalent to the second example given in the table above.

Since `IPREAD` was intended to be used mainly for reading output produced by previous computations, it is designed to be fast and, consequently, very little error checking is performed on the input. Among other things, `IPREAD` does not check for consistency among the variables, e.g. `((y)x^2+(z)y)` will be accepted as valid input and would be identical to `((u)v^2+(u)v)` in internal representation. Also, `IPREAD` does not check for consistency among terms, i.e. each term is processed separately and it is not checked whether all terms have the same number of recursive nestings. For example, `(y^3+(x-1)y)` will be accepted although the first term, `y^3`, is a univariate polynomial whereas the second, `(x-1)y`, is a bivariate polynomial. *It is therefore the responsibility of the user to see that polynomials are input properly.*

The algorithm `IPWRITE` takes as inputs an r-variate recursive polynomial `A` over $\mathbf{Z}$ and a list `V = (`$v_1, \ldots, v_r$`)` of r variables and writes `A` to the output stream using the variables specified with v_r as the main variable and v_1 as the most minor variable. The list `V` may be initialized using `VLREAD` which reads a variable list from the input stream. For generating a list with a fixed number of variables one could also use an expression such as `LIST3(LFS("X"),LFS("Y"),LFS("Z"))`. Here the functions `LFS` is used for converting a C string to a SACLIB variable. It is possible to use the algorithm `IUPWRITE` to write univariate recursive polynomials but this algorithm was intended mainly as a subroutine to be called by `IPWRITE`, which also handles univariate polynomials, and the user need not even be aware of its existence.

There is an additional set of input functions of which the top level function is `IPEXPREAD`. The format accepted by this function is a bit more convenient as expressions may be of the form `(3 X Y^2 + X)^3 - (Y X + Y) (X - 1)^2 + 5`. Note that `IPEXPREAD` also takes a variable list as input and therefore can detect the order of the variables without requiring the recursive structure made explicit by parentheses.

To be more precise, `IPEXPREAD` accepts any polynomial expression built from integers and variables using +, -, blanks for multiplication, ^ for exponentiation, and parenthesis for grouping. The expression may be terminated by any character not being part of the legal input set (e.g. a period, a semicolon, etc.). This terminating character is not removed from the input stream.

The function `IPEXPREADR` has the same specification as `IPEXPREAD`, with the difference that it *does* remove the terminating character.

4.2.2 Recursive polynomials over Q

For r-variate recursive polynomials over $\mathbf{Q}$ the algorithms `RPREAD` and `RPWRITE` are the corresponding input and output routines. The situation for rational polynomials is essentially the same as that for integral polynomials with the exception that the base coefficients may be rational numbers. The same freedoms on valid input apply and an arbitrary number of spaces may be inserted before and after '/'. If the denominator of a base coefficient is 1 then only the numerator is in the external canonical representation. As an example, the external canonical representation of $\frac{2}{7}x^3 - 65x^2 + 5x + \frac{12}{4}$ is `(2/7x^3-65x^2+5x+12/4)` which, among many other possible variations, may be input as `(2/7x^ 3- 65 x^2 + 5/1x+12/ 4)`. It should be noted that the rational base coefficients are not reduced to lowest terms when converted to internal representation. Corresponding to `IUPWRITE` is `RUPWRITE` which, again, need not concern the user.

For rational polynomials there are also input functions for reading polynomial expressions. Here the name of the top level function is `RPEXPREAD`. The input format here is the same as in the integral case, except that numbers may be rational.

4.2.3 Distributive polynomials over Z

The external canonical representation of sparse *distributive* polynomials over **Z** is as follows. The entire polynomial is enclosed in parentheses. Each term is made up of the integer coefficient followed by the variables having positive exponents with each variable separated from its corresponding exponent by a caret. The coefficient and each variable-exponent pair is separated by a single space. As was the case for recursive polynomials, coefficients and exponents with a magnitude of 1 are suppressed as are variables with exponent 0. For example, the polynomial $2x^3y^5 - xy^3 - 4y + x + 1$ will be expressed in external canonical form as `( 2 x^3 y^5 - x y^3 -4 y + x +1 )`.

The algorithms `DIIPREAD` and `DIIPWRITE` are the input and output routines for distributive polynomials over **Z**. `DIIPREAD` takes as input a variable list `V` $= (v_1, \ldots, v_r)$ and reads a distributive polynomial in external canonical form from the input stream. The ordering of the variables in `V` is significant and the variables in each term of the polynomial that is read must appear in the same order that they appear in `V` and the terms must be ordered in descending degree in v_r. For example, if `V` = `(x,y,z)` then `(4 z^5 - y^2 z^4 + 9 x y z)` is valid but `(4 z^5 + 9 y x z - y^2 z^4)` is not for two reasons—first, `y` appears before `x` in the term `9 y x z` and second, the term `9 y x z` appears before `- y^2 z^4` which violates the rule that terms must appear in order of descending degree in `z`. Additionally, if there are two terms with the same degree in v_r then they should be ordered according to descending degree in v_{r-1} and so on. Coefficients may be separated from the variables by an arbitrary number of spaces (including no space at all). Variables must be separated by at least one space if there is no exponent explicitly given, otherwise an arbitrary number of spaces may separate them. For example `(4z^5 - y^2z^4 + 9x y z)` is valid but `(4z^5 - y^2z^4 + 9xyz)` is not since `xyz` will be treated as a single variable.

4.2.4 Distributive polynomials over Q

Distributive polynomials over **Q** may be read in and written out using the algorithms `DIRPREAD` and `DIRPWRITE`. The only difference between rational distributive polynomials and integral distributive polynomials is that the base coefficients may be rational numbers and not just integers. It should be clear after reading the preceding subsections what constitutes valid input and we will not discuss this matter further.

4.2.5 Conversion Between Recursive and Distributive Representation

Converting recursive polynomials to distributive polynomials can be achieved by using `DIPFP` which, given a polynomial in recursive internal representation, computes an equivalent one in distributive internal representation. In the other direction, namely to convert from distributive to recursive representation, the algorithm `PFDIP` is provided. Both `DIPFP` and `PFDIP` work for polynomials over either **Z** or **Q** but the number of variables must be specified. For example, if `A` is a polynomial over **Q** in internal recursive representation and the user wants to display `A` in external distributive representation then the code

```
DIRPWRITE(r,DIPFP(r,A),V);
```

where `r` is equal to the number of variables and `V` is a list of `r` variables, will suffice.

4.2.6 Polynomials over $\mathbf{Z}_m$

The input and output routines for polynomials over **Z** work equally well for polynomials over $\mathbf{Z}_m$.

4.3 Domain Independent Polynomial Arithmetic

Constructors:

`A <- PFBRE(r,a)` Polynomial from Base Ring Element. *Builds an r-variate polynomial from an element of some domain.*

`A <- PMON(a,e)` Polynomial monomial. *Builds ax^e from a and e.*

`A <- PBIN(a1,e1,a2,e2)` Polynomial binomial. *Builds $a_1x^{e_1}+a_2x^{e_2}$ from a_1, a_2, e_1, and e_2.*

Selectors:

`a <- PLDCF(A)` Polynomial leading coefficient. *Returns the leading coefficient w.r.t. the main variable.*

`B <- PRED(A)` Polynomial reductum. *Returns the reductum (the polynomial minus its leading term) w.r.t. the main variable.*

`a <- PLBCF(r,A)` Polynomial leading base coefficient. *Returns the coefficient of the term of the highest degree w.r.t. all variables (an element of the base domain).*

`a <- PTBCF(r,A)` Polynomial trailing base coefficient. *Returns the coefficient of the term of the lowest degree w.r.t. all variables (an element of the base domain).*

Information and Predicates:

`n <- PDEG(A)` Polynomial degree. *Returns the degree of the argument w.r.t. the main variable.*

`n <- PMDEG(A)` Polynomial modified degree. *Returns the degree of the argument, -1 if the argument is 0.*

`n <- PDEGSV(r,A,i)` Polynomial degree, specified variable. *Returns the degree of the argument w.r.t. the i-th variable.*

`V <- PDEGV(r,A)` Polynomial degree vector. *Returns a list $(d_1, \ldots, d_r)$ where d_i is the degree of argument w.r.t. the i-th variable.*

`b <- PCONST(r,A)` Polynomial constant. *Tests whether the argument is a constant polynomial.*

`b <- PUNT(r,A)` Polynomial univariate test. *Tests whether the argument is a univariate polynomial.*

`k <- PORD(A)` Polynomial order. *Returns the smallest exponent appearing in the argument polynomial (w.r.t. the main variable).*

Transformation:

`B <- PSDSV(r,A,i,n)` Polynomial special decomposition, specified variable. *Computes $p(x_1, \ldots, x_i^{1/n}, \ldots, x_r)$ given p, i, n, and r.*

`B <- PDPV(r,A,i,n)` Polynomial division by power of variable. *Computes $x_i^{-n}p$ given p, i, and n.*

`B <- PMPMV(A,k)` Polynomial multiplication by power of main variable. *Computes $x^n p$ given p and n, with x being the main variable of p.*

`B <- PRT(A)` Polynomial reciprocal transformation. *Computes $x^n p(x^{-1})$ with $n = \deg(p)$.*

`B <- PDBORD(A)` Polynomial divided by order. *Computes $x^{-n}p$ where n is the order of p.*

Conversion[1]**:**

`B <- PFDIP(r,A)` Polynomial from distributive polynomial. *Computes a polynomial in the sparse recursive representation from a polynomial in the sparse distributive representation.*

`B <- PFDP(r,A)` Polynomial from dense polynomial. *Computes a polynomial in the sparse recursive representation from a polynomial in the dense recursive representation.*

Miscellaneous:

`B <- PINV(r,A,k)` Polynomial introduction of new variables. *Computes a polynomial in* $R[y_1,\ldots,y_s,x_1,\ldots,x_r]$ *from a polynomial in* $R[x_1,\ldots,x_r]$.

`B <- PPERMV(r,A,P)` Polynomial permutation of variables. *Computes a polynomial in* $R[x_{p_1},\ldots,x_{p_r}]$ *from a polynomial in* $R[x_1,\ldots,x_r]$, *where* $(p_1,\ldots,p_r)$ *is a permutation of* $(1,\ldots,r)$.

`B <- PCPV(r,A,i,j)` Polynomial cyclic permutation of variables.

`B <- PICPV(r,A,i,j)` Polynomial inverse cyclic permutation of variables.

`B <- PTV(r,A,i)` Polynomial transpose variables.

`B <- PTMV(r,A)` Polynomial transpose main variables.

`B <- PUFP(r,A)` Polynomial, univariate, from polynomial. *Computes a univariate polynomial from an r-variate polynomial by substituting 0 for all variables except the main variable* x_r.

`L <- PCL(A)` Polynomial coefficient list. *Returns a list* $(p_n,\ldots,p_0)$ *where* n *is the degree of* p *and the* p_i *are the coefficients of* p.

4.4 Integral Polynomial Arithmetic

Basic arithmetic:

`C <- IPSUM(r,A,B)` Integral polynomial sum.

`C <- IPDIF(r,A,B)` Integral polynomial difference.

`B <- IPNEG(r,A)` Integral polynomial negative.

`C <- IPPROD(r,A,B)` Integral polynomial product.

`C <- IPIP(r,a,B)` Integral polynomial integer product. *Computes* $c * p$ *given an integer* c *and an integral polynomial* p.

`C <- IPP2P(r,B,m)` Integral polynomial power of 2 product.

`IPQR(r,A,B; Q,R)` Integral polynomial quotient and remainder.

`C <- IPQ(r,A,B)` Integral polynomial quotient.

`C <- IPIQ(r,A,b)` Integral polynomial integer quotient. *Computes* p/c *given an integral polynomial* p *and an integer* c.

`C <- IPPSR(r,A,B)` Integral polynomial pseudo-remainder.

`IUPSR(A,B; ab,bb,C)` Integral univariate polynomial semi-remainder.

`B <- IPEXP(r,A,n)` Integral polynomial exponentiation.

`s <- IPSIGN(r,A)` Integral polynomial sign.

`B <- IPABS(r,A)` Integral polynomial absolute value.

[1] See Section 4.7 for a description of the sparse distributive and the dense recursive representations.

Differentiation and Integration:

`B <- IPDMV(r,A)` Integral polynomial derivative, main variable.

`B <- IPDER(r,A,i)` Integral polynomial derivative. *Computes the derivative of the argument w.r.t. the i-th variable.*

`B <- IPHDMV(r,A,k)` Integral polynomial higher derivative, main variable. *Computes the k-th derivative of the argument w.r.t. the main variable.*

`B <- IPINT(r,A,b)` Integral polynomial integration. *Computes the integral of the argument w.r.t. the main variable.*

Substitution and Evaluation:

`C <- IPSMV(r,A,B)` Integral polynomial substitution for main variable. *Substitutes an integral polynomial for the main variable of an integral polynomial.*

`C <- IPSUB(r,A,i,B)` Integral polynomial substitution. *Substitutes an integral polynomial for the i-th variable of an integral polynomial.*

`C <- IPGSUB(r,A,s,L)` Integral polynomial general substitution. *Substitutes an integral polynomials for all variables of an integral polynomial.*

`B <- IUPQS(A)` Integral univariate polynomial quotient substitution.

`B <- IPEMV(r,A,a)` Integral polynomial evaluation of main variable. *Substitutes a constant for the main variable of an integral polynomial.*

`B <- IPEVAL(r,A,i,a)` Integral polynomial evaluation. *Substitutes a constant for the i-th variable of an integral polynomial.*

`b <- IUPBEI(A,c,m)` Integral univariate polynomial binary rational evaluation, integer output.

`s <- IUPBES(A,a)` Integral univariate polynomial binary rational evaluation of sign.

`b <- IUPBRE(A,a)` Integral univariate polynomial binary rational evaluation.

`B <- IPBEILV(r,A,c,k,m)` Integral polynomial binary rational evaluation, integral polynomial result, leading variable.

`B <- IPBREI(r,A,i,c)` Integral polynomial binary rational evaluation, integral polynomial result.

Transformation:

`B <- IPTRMV(r,A,h)` Integral polynomial translation, main variable. *Computes $p(x+h)$ given p and h, where x is the main variable of p.*

`B <- IPTRAN(r,A,T)` Integral polynomial translation. *Computes $p(x_1+t_1,\ldots,x_r+t_r)$ given p and the t_i.*

`B <- IPBHT(r,A,i,k)` Integral polynomial binary homothetic transformation.

`B <- IPBHTLV(r,A,k)` Integral polynomial binary homothetic transformation, leading variable.

`B <- IPBHTMV(r,A,k)` Integral polynomial binary homothetic transformation, main variable.

`B <- IUPBHT(A,k)` Integral univariate polynomial binary homothetic transformation.

`B <- IUPIHT(A,n)` Integral univariate polynomial integer homothetic transformation.

`B <- IPNT(r,A,i)` Integral polynomial negative transformation.

`B <- IUPNT(A)` Integral univariate polynomial negative transformation.

`B <- IPTR(r,A,i,h)` Integral polynomial translation, specified variable.

`B <- IUPTR(A,h)` Integral univariate polynomial translation.

`B <- IPTR1(r,A,i)` Integral polynomial translation by 1, specified variable. specified variable.

`B <- IPTRLV(r,A)` Integral polynomial translation, leading variable.

`B <- IPTR1LV(r,A)` Integral polynomial translation by 1, leading variable.

`B <- IUPTR1(A)` Integral univariate polynomial translation by 1.

Predicates:

`t <- IPCONST(r,A)` Integral polynomial constant. *Tests whether the argument is a constant.*

`t <- IPONE(r,A)` Integral polynomial one. *Tests whether the argument is 1.*

Random Polynomial Generation:

`A <- IPRAN(r,k,q,N)` Integral polynomial, random.

Conversion:

`IPSRP(r,A; a,Ab)` Integral polynomial similiar to rational polynomial. *Given a rational polynomial q, computes a rational number c and an integral polynomial p with $cp = q$.*

`B <- IPFRP(r,A)` Integral polynomial from rational polynomial. *Computes an integral polynomial from a rational polynomial whose base coefficients are integers.*

Input/Output:

`IPREAD(; r,A)` Integral polynomial read.

`IPEXPREAD(r,V; A,t)` Integral polynomial expression read.

`IPWRITE(r,A,V)` Integral polynomial write.

`IPDWRITE(r,A,V)` Integral Polynomial Distributive Write. *Writes an integral recursive polynomial in distributive form.*

Contents and Primitive Parts:

`IPICPP(r,A; a,Ab)` Integral polynomial integer content and primitive part.

`c <- IPIC(r,A)` Integral polynomial integer content.

`Ab <- IPIPP(r,A)` Integral polynomial integer primitive part.

`d <- IPICS(r,A,c)` Integral polynomial integer content subroutine.

`IPSCPP(r,A; s,C,Ab)` Integral polynomial sign, content, and primitive part. *Computes the sign, content and primitive part of the argument w.r.t. the main variable.*

`IPCPP(r,A; C,Ab)` Integral polynomial content and primitive part.

`C <- IPC(r,A)` Integral polynomial content.

`Ab <- IPPP(r,A)` Integral polynomial primitive part.

`IPLCPP(r,A; C,P)` Integral polynomial list of contents and primitive parts.

Polynomial Norms:

`b <- IPSUMN(r,A)` Integral polynomial sum norm.

`b <- IPMAXN(r,A)` Integral polynomial maximum norm.

Chinese Remainder Algorithm:

`As <- IPCRA(M,m,mp,r,A,a)` Integral polynomial chinese remainder algorithm.

Squarefree Factorization:

`L <- IPSF(r,A)` Integral polynomial squarefree factorization.

`L <- IPFSD(r,A)` Integral polynomial factorization, second derivative.

`L <- IPSFSD(r,A)` Integral squarefree factorization, second derivative.

Computations in Ideals:

`B <- IPTRUN(r,D,A)` Integral polynomial truncation. *Computes* $p \bmod (x_1^{d_1}, \ldots, x_r^{d_r})$ *given* p *and the* d_i.

`C <- IPTPR(r,D,A,B)` Integral polynomial truncated product. *Computes* $pq \bmod (x_1^{d_1}, \ldots, x_r^{d_r})$ *given* p, q, *and the* d_i.

`B <- IPIHOM(r,D,A)` Integral polynomial mod ideal homomorphism. *Computes* $p \bmod$
$(x_1^{d_1}, \ldots, x_{r-1}^{d_{r-1}})$ *given an* r*-variate polynomial* p *and the* d_i.

`C <- IPIPR(r,D,A,B)` Integral polynomial mod ideal product. *Computes* $pq \bmod (x_1^{d_1}, \ldots, x_{r-1}^{d_{r-1}})$ *given* r*-variate polynomials* p *and* q *and the* d_i.

`C <- IUPTPR(n,A,B)` Integral univariate polynomial truncated product. *Computes* $pq \bmod x^n$ *given univariate polynomials* p *and* q *and a* `BETA`*-digit* n.

4.5 Modular Polynomial Arithmetic

Note that the functions whose names begin with MI are based upon modular integer arithmetic, while those beginning with MP and MUP are based upon modular digit arithmetic with a prime modulus[2].

Basic arithmetic:

`C <- MIPSUM(r,M,A,B)` Modular integral polynomial sum.

`C <- MPSUM(r,m,A,B)` Modular polynomial sum.

`C <- MIPDIF(r,M,A,B)` Modular integral polynomial difference.

`C <- MPDIF(r,m,A,B)` Modular polynomial difference.

`B <- MIPNEG(r,M,A)` Modular integral polynomial negation.

`B <- MPNEG(r,m,A)` Modular polynomial negative.

`C <- MIPPR(r,M,A,B)` Modular integral polynomial product.

`C <- MPPROD(r,m,A,B)` Modular polynomial product.

`B <- MPUP(r,m,c,A)` Modular polynomial univariate product.

`C <- MPMDP(r,p,a,B)` Modular polynomial modular digit product.

`C <- MIPIPR(r,M,D,A,B)` Modular integral polynomial mod ideal product.

`MIUPQR(M,A,B; Q,R)` Modular integral univariate polynomial quotient and remainder.

`MPQR(r,p,A,B; Q,R)` Modular polynomial quotient and remainder.

`C <- MPQ(r,p,A,B)` Modular polynomial quotient.

`C <- MPUQ(r,p,A,b)` Modular polynomial univariate quotient.

[2] See Section 3.3 for details on modular digit and integer arithmetic.

`C <- MPPSR(r,p,A,B)` Modular polynomial pseudo-remainder.

`MMPIQR(r,M,D,A,B; Q,R)` Modular monic polynomial mod ideal quotient and remainder.

`B <- MPEXP(r,m,A,n)` Modular polynomial exponentiation.

Differentiation and Integration:

`B <- MUPDER(m,A)` Modular univariate polynomial derivative.

Contents and Primitive Parts:

`MPUCPP(r,p,A; a,Ab)` Modular polynomial univariate content and primitive part.

`c <- MPUC(r,p,A)` Modular polynomial univariate content.

`Ab <- MPUPP(r,p,A)` Modular polynomial univariate primitive part.

`d <- MPUCS(r,p,A,c)` Modular polynomial univariate content subroutine.

Evaluation:

`B <- MPEMV(r,m,A,a)` Modular polynomial evaluation of main variable.

`B <- MPEVAL(r,m,A,i,a)` Modular polynomial evaluation.

Transformation:

`Ap <- MPMON(r,p,A)` Modular polynomial monic. *Computes the monic polynomial similar to a given modular polynomial.*

Chinese Remainder Algorithm:

`As <- MPINT(p,B,b,bp,r,A,A1)` Modular polynomial interpolation.

`B <- MIPHOM(r,M,A)` Modular integral polynomial homomorphism. *Computes the homomorphism from* $\mathbf{Z}[x_1,\ldots,x_r]$ *to* $\mathbf{Z}_m[x_1,\ldots,x_r]$.

`B <- MPHOM(r,m,A)` Modular polynomial homomorphism.

Squarefree Factorization:

`L <- MUPSFF(p,A)` Modular univariate polynomial squarefree factorization.

Random Polynomial Generation:

`A <- MIPRAN(r,M,q,N)` Modular integral polynomial, random.

`A <- MPRAN(r,m,q,N)` Modular polynomial, random.

`A <- MUPRAN(p,n)` Modular univariate polynomial, random.

Conversion:

`B <- MIPFSM(r,M,A)` Modular integral polynomial from symmetric modular.

`B <- SMFMIP(r,M,A)` Symmetric modular from modular integral polynomial.

4.6 Rational Polynomial Arithmetic

Basic arithmetic:

`C <- RPSUM(r,A,B)` Rational polynomial sum.

`C <- RPDIF(r,A,B)` Rational polynomial difference.

`B <- RPNEG(r,A)` Rational polynomial negative.

`C <- RPPROD(r,A,B)` Rational polynomial product.

`C <- RPRNP(r,a,B)` Rational polynomial rational number product.

`RPQR(r,A,B; Q,R)` Rational polynomial quotient and remainder.

Differentiation and Integration:

`B <- RPDMV(r,A)` Rational polynomial derivative, main variable.

`B <- RPIMV(r,A)` Rational polynomial integration, main variable.

Evaluation:

`C <- RPEMV(r,A,b)` Rational polynomial evaluation, main variable.

Conversion:

`B <- RPFIP(r,A)` Rational polynomial from integral polynomial.

`B <- RPMAIP(r,A)` Rational polynomial monic associate of integral polynomial.

Input/Output:

`RPREAD(; r,A)` Rational polynomial read.

`RPEXPREAD(r,V; A,t)` Rational polynomial expression read.

`RPWRITE(r,A,V)` Rational polynomial write.

`RPDWRITE(r,A,V)` Rational Polynomial Distributive Write.

Normalization:

`RPBLGS(r,A; a,b,s)` Rational polynomial base coefficients least common multiple, greatest common divisor, and sign.

4.7 Miscellaneous Representations

4.7.1 Sparse Distributive Representation

Conversion[3]:

`B <- DIPFP(r,A)` Distributive polynomial from polynomial. *Computes a polynomial in sparse distributive representation from a polynomial in the sparse recursive representation.*

`B <- PFDIP(r,A)` Polynomial from distributive polynomial. *Computes a polynomial in the sparse recursive representation from a polynomial in the sparse distributive representation.*

Input/Output:

`A <- DIIPREAD(V)` Distributive integral polynomial read.

[3] See Section 4.1 for a description of the sparse recursive representation.

`DIIPWRITE(r,A,V)` Distributive integral polynomial write.

`A <- DIRPREAD(V)` Distributive rational polynomial read.

`DIRPWRITE(r,A,V)` Distributive rational polynomial write.

Miscellaneous:

`n <- DIPDEG(r,A)` Distributive polynomial degree.

`DIPINS(a,d,A; t,B)` Distributive polynomial, insert term.

4.7.2 Dense Recursive Representation

Basic arithmetic:

`C <- DMPPRD(r,m,A,B)` Dense modular polynomial product.

`C <- DMPSUM(r,m,A,B)` Dense modular polynomial sum.

`C <- DMUPNR(p,A,B)` Dense modular univariate polynomial natural remainder.

Conversion[4]:

`B <- DPFP(r,A)` Dense polynomial from polynomial. *Computes a polynomial in dense recursive representation from a polynomial in the sparse recursive representation.*

`B <- PFDP(r,A)` Polynomial from dense polynomial. *Computes a polynomial in the sparse recursive representation from a polynomial in the dense recursive representation.*

[4] See Section 4.1 for a description of the sparse recursive representation.

Chapter 5

Linear Algebra

5.1 Mathematical Preliminaries

A *matrix* A of *order* $m \times n$ over a domain D is a rectangular array of elements of D of the form

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

which we will sometimes denote by $A = (a_{ij})$. If A has order $n \times n$ then we say that A is a *square* matrix. When appropriate, we will denote a matrix A by $A_{m\times n}$ to indicate that the order of A is $m \times n$.

If A is a square matrix then the *determinant* of A, denoted $\det(A)$, is defined to be $\det(A) = \sum \epsilon(\sigma) a_{\sigma(1),1} \cdots a_{\sigma(n),n}$, the sum being taken over all permutations σ of $\{1, \ldots, n\}$ with $\epsilon(\sigma)$ equal to the sign[1] of σ.

An *m-vector* V is a matrix of order $m \times 1$ and will be denoted by $V = (v_i)$. If A is a matrix of order $m \times n$, b is an m-vector and x is an n-vector then the equation $Ax = b$ can be viewed as representing the system of linear equations $\sum_{j=1}^{n} a_{ij} x_j = b_i$, $i = 1, \ldots, m$. If a solution to the system $Ax = b$ exists then we say that the system is *consistent*. The *null space* of a matrix $A_{m\times n}$ is the set of all n-vectors x that satisfy $Ax = 0$. A *basis* B for the null-space of A is a set of n-vectors such that each element of the null-space can be expressed as a linear combination of elements of B.

5.2 Purpose

The SACLIB linear algebra package provides algorithms for solving systems of linear diophantine equations, for computing null-space bases, for computing determinants and for matrix multiplication.

5.3 Methods and Algorithms

To solve a system of linear diophantine equations one may use either of the two algorithms `LDSMKB` and `LDSSBR`. Both algorithms take as inputs a matrix $A_{m\times n}$ and an m-vector b, with A represented column-wise, i.e. A is a list of n columns each of which is a list of m integers. Either algorithm returns an n-vector x^* and a list N where x^* is a particular solution of

[1] If σ is the product of m transpositions, then the *sign* of σ is $\epsilon(\sigma) = (-1)^m$.

the system of linear diophantine equations $Ax = b$ and N is a list of n-vectors that form a basis for the null space of A. In case the system $Ax = b$ is not consistent, both x^* and N are null lists. `LDSMKB` implements a modification of the Kannan-Bachem algorithm while `LDSSBR` implements an algorithm based on ideas of Rosser.

Determinants of matrices over $\mathbf{Z}[x_1, \ldots, x_r]$ may be computed by using either `MAIPDE` or `MAIPDM`. `MAIPDE` implements an algorithm which is based on exact division while `MAIPDM` is based on modular homomorphisms and Chinese remaindering.

`MMDDET` computes determinants of matrices over $\mathbf{Z}_p$ while `MMPDMA`, which is based on evaluation homomorphisms and interpolation, computes determinants of matrices over $\mathbf{Z}_p[x_1, \ldots, x_r]$.

5.4 Functions

Systems of linear equations:

`LDSMKB(A,b; xs,N)` Linear diophantine system solution, modified Kannan and Bachem algorithm. *Given an integral matrix $A_{m\times n}$, represented column-wise, and an integral m-vector b, uses a modification of the Kannan and Bachem algorithm to compute x^* and N where x^* is a particular solution of the system of linear equations $Ax = b$ and N is a list of vectors which form a basis for the solution module of $Ax = 0$. If $Ax = b$ is not consistent then both x^* and N are null lists.*

`LDSSBR(A,b; xs,N)` Linear diophantine system solution, based on Rosser ideas. *Similar to* LDSMKB *but the computations are performed using an algorithm based on ideas of Rosser.*

`B <- MMDNSB(p,M)` Matrix of modular digits null-space basis. *Given a matrix $A_{m\times n}$ over $\mathbf{Z}_p$, represented row-wise, computes a list $B = (B_1, \ldots, b_r)$ of m-vectors representing a basis for the null-space of A.*

Determinants:

`D <- MAIPDE(r,M)` Matrix of integral polynomials determinant, exact division algorithm. *Given a square matrix A over $\mathbf{Z}$ computes* $\det(A)$.

`D <- MAIPDM(r,M)` Matrix of integral polynomials determinant, modular algorithm. *Similar to* `MAIPDE` *but uses an algorithm based on modular homomorphisms and Chinese remaindering.*

`d <- MMDDET(p,M)` Matrix of modular digits determinant. *Given a square matrix A over $\mathbf{Z}_p$, computes* $\det(M)$.

`D <- MMPDMA(r,p,M)` Matrix of modular polynomials determinant, modular algorithm. *Given a matrix M over $\mathbf{Z}_p[x_1, \ldots, x_r]$, computes* $\det(M)$ *using a method based on evaluation homomorphisms and interpolation.*

Matrix arithmetic:

`C <- MAIPP(r,A,B)` Matrix of integral polynomials product.

Vector computations:

`B <- VIAZ(A,n)` Vector of integers, adjoin zeros.

`C <- VIDIF(A,B)` Vector of integers difference.

`W <- VIERED(U,V,i)` Vector of integers, element reduction.

`C <- VILCOM(a,b,A,B)` Vector of integers linear combination.

`B <- VINEG(A)` Vector of integers negation.

`C <- VISPR(a,A)` Vector of integers scalar product.

`C <- VISUM(A,B)` Vector of integers sum.

`VIUT(U,V,i; Up,Vp)` Vector of integers, unimodular transformation.

Miscellaneous functions:

`B <- MAIPHM(r,m,A)` Matrix of integral polynomials homomorphism.

`B <- MIAIM(A)` Matrix of integers, adjoin identity matrix.

`B <- MICINS(A,V)` Matrix of integers column insertion.

`B <- MICS(A)` Matrix of integers column sort.

`B <- MINNCT(A)` Matrix of integers, non-negative column transformation.

`B <- MMPEV(r,m,A,k,a)` Matrix of modular polynomials evaluation.

Chapter 6

Polynomial GCD and Resultants

6.1 Mathematical Preliminaries

Given polynomials A and B in $R[x_1, \ldots, x_r]$, R a unique factorization domain, a *greatest common divisor* (GCD) of A and B is a polynomial C in $R[x_1, \ldots, x_r]$ such that C divides both A and B and such that any other divisor of both A and B also divides C. GCDs of more than two polynomials are defined in a similar way. GCDs are not unique since any unit multiple of a GCD is itself a GCD. Polynomials A and B are *relatively prime* if 1 is a GCD of A and B.

If $A = \sum_{i=0}^{m} a_i x_r^i$ and $B = \sum_{i=0}^{n} b_i x_r^i$, then the *Sylvester matrix* of A and B is the $(m+n) \times (m+n)$ square matrix

$$\begin{pmatrix}
a_m & a_{m-1} & \cdots & & & & \cdots & a_0 & 0 & \cdots & 0 \\
0 & a_m & \cdots & & & & \cdots & a_1 & a_0 & \cdots & 0 \\
\vdots & & \ddots & & & & & & & \ddots & \vdots \\
0 & \cdots & 0 & a_m & \cdots & & & & \cdots & a_1 & a_0 \\
b_n & b_{n-1} & \cdots & & & \cdots & b_0 & 0 & \cdots & \cdots & 0 \\
0 & b_n & \cdots & & & \cdots & b_1 & b_0 & 0 & \cdots & 0 \\
\vdots & & \ddots & & & & & & \ddots & & \vdots \\
\vdots & & & \ddots & & & & & & \ddots & \vdots \\
0 & \cdots & \cdots & 0 & b_n & \cdots & & & \cdots & b_1 & b_0
\end{pmatrix}$$

in which there are n rows of A coefficients and m rows of B coefficients.

The *resultant* of two polynomials A and B, denoted by $\text{res}(A, B)$, is the determinant of the Sylvester matrix of A and B. The resultant will be an element of $R[x_1, \ldots, x_{r-1}]$ if A and B are elements of $R[x_1, \ldots, x_r]$. From a classic result we know that A and B are relatively prime just in case their resultant is nonzero.

Let $\deg_{x_r}(A) = m$ and $\deg_{x_r}(B) = n$, with $m \geq n > 0$. If M is the Sylvester matrix of A and B, then for $0 \leq i \leq j < n$ define M_{ij} to be the matrix obtained by deleting from M the last j rows of the A coefficients, the last j rows of the B coefficients and the last $2j+1$ columns except column $m+n-i-j$. The *j-th subresultant* of A and B is the polynomial $S_j(x_r) = \sum_{i=0}^{j} \det(M_{ij}) x_r^i$ for $0 \leq j < n$. Note that S_0 is simply $\text{res}(A, B)$. The *k-th principal subresultant coefficient* of A and B is the coefficient of x_r^k in S_k (which may be 0).

A and B are *similar*, denoted $A \sim B$, if there exist a and b, elements of $R[x_1, \ldots, x_{r-1}]$, such that $aA = bB$. If A and B are nonzero and $\deg_{x_r}(A) \geq \deg_{x_r}(B)$ then a *polynomial* *remainder sequence* (PRS) of A and B is a sequence $A_1, \ldots, A_n$ of nonzero polynomials such that $A_1 = A$, $A_2 = B$, $A_i \sim \mathrm{prem}(A_{i-2}, A_{i-1})$, for $i = 3, \ldots, n$, and $\deg_{x_r}(A_n) = 0$.[1]

Since there are many polynomials similar to a given one, there are many different PRSs $A_1, A_2, \ldots, A_n$ corresponding to A and B.

The *Euclidean* PRS is obtained by setting $A_i = \mathrm{prem}(A_{i-2}, A_{i-1})$ for $i = 3, \ldots, n$.

The *primitive* PRS is obtained by setting $A_i = \mathrm{prem}(A_{i-2}, A_{i-1})/g_i$, where g_i is the content of $\mathrm{prem}(A_{i-2}, A_{i-1})$. In other words, we set A_i to be equal to the primitive part of $\mathrm{prem}(A_{i-2}, A_{i-1})$.

The *subresultant* PRS *of the first kind* is obtained by setting $A_i = S_{d_{i-1}-1}$ where d_i is the degree of the i-th element of any PRS of A and B. [For each i, d_i is invariant over the set of PRSs of A and B.]

The *subresultant* PRS *of the second kind* is obtained by setting $A_i = S_{d_i}$ where d_i is as in the previous definition.

The *reduced* PRS is obtained by setting $A_i = \mathrm{prem}(A_{i-2}, A_{i-1})/c_i^{\delta_i+1}$, where $c_i = \mathrm{ldcf}(A_{i-2})$ and $\delta_i = \deg_{x_r}(A_{i-3}) - \deg_{x_r}(A_{i-2})$ for $3 \leq i \leq n$, with $\delta_3 = 0$.

Although it is not immediately clear from the definitions, both subresultant PRSs as well as the reduced PRS can be shown to be, in fact, PRSs.

For univariate polynomials over a field we may define what is known as the *natural* PRS defined by $A_i = A_{i-2} - Q_i A_{i-1}$, $\deg(A_i) < \deg(A_{i-1})$, for $i = 3, \ldots, n$. That is, we take A_i to be the remainder obtained from dividing A_{i-2} by A_{i-1}.

6.2 Purpose

The SACLIB polynomial GCD and resultant package provides algorithms for the calculation of GCDs of r-variate polynomials over $R = \mathbf{Z}$ or $R = \mathbf{Z}_p$. Since GCDs are not unique, we will need to specify a canonical form in which to express the results of the computations. Over $R = \mathbf{Z}$, the positive GCD is computed while over $R = \mathbf{Z}_p$ it is the monic GCD that is computed. Henceforth, if we refer to *the* GCD of A and B we will mean the GCD defined by the algorithms and this will be denoted by $\gcd(A, B)$.

Algorithms are also available for the computation of resultants of r-variate polynomials over $R = \mathbf{Z}$ or $R = \mathbf{Z}_p$. The package also provides algorithms for computing the subresultant PRS and the reduced PRS for r-variate polynomials over $R = \mathbf{Z}$ and the subresultant PRS for r-variate polynomials over $R = \mathbf{Z}_p$.

6.3 Definitions of Terms

coarsest squarefree basis If $A = (A_1, \ldots, A_n)$ is a list of r-variate polynomials, a coarsest squarefree basis for A is a list $B = (B_1, \ldots, B_m)$ of pairwise relatively prime squarefree r-variate polynomials such that each A_i in A can be expressed as the product of powers of elements of B.

discriminant If A is an r-variate polynomial of degree n in its main variable, $n \geq 2$, the discriminant of A is the $(r-1)$-variate polynomial equal to the quotient of $(-1)^{n(n-1)/2}\mathrm{res}(A, A')$ when divided by a, where A' is the derivative of A with respect to its main variable and a is the leading coefficient of A.

finest squarefree basis A finest squarefree basis $B = (B_1, \ldots, B_m)$ for a list A of r-variate polynomials is a coarsest squarefree basis for A with the additional condition that each B_i is irreducible.

cofactors If C is the GCD of two polynomials A and B then the cofactors of A and B, respectively, are A/C and B/C.

[1] $\mathrm{prem}(F, G)$ denotes the pseudo-remainder of F when pseudo-divided by G with respect to the main variable x_r.

content The content of a polynomial A in r variables is a polynomial in $r-1$ variables equal to the absolute value of the greatest common divisor of the coefficients of A.

greatest squarefree divisor A greatest squarefree divisor of a polynomial A is a squarefree polynomial C that divides A and is such that any other squarefree polynomial that divides A also divides C.

primitive part The primitive part of a polynomial A is the absolute value of A/c where c is the content of A.

primitive polynomial A polynomial, the content of which is 1.

squarefree factorization The squarefree factorization of A is $A = A_1^{e_1} \cdots A_k^{e_k}$ where $1 \leq e_1 < \cdots < e_k$ and each of the A_i is a positive squarefree polynomial of positive degree. Note that if A is squarefree then A^1 is the squarefree factorization of A.

squarefree polynomial A polynomial A is squarefree if each factor occurs only once. In other words, if $A = A_1^{e_1} \cdots A_k^{e_k}$ is a complete factorization of A then each of the e_i is equal to 1.

univariate content If A is an r-variate polynomial, $r \geq 2$, then the univariate content of A is a univariate polynomial in the most minor variable equal to the GCD of the coefficients of A, where A is considered as an element of $(R[x_1])[x_2, \ldots, x_r]$.

univariate primitive part Given an r-variate polynomial A, $r \geq 2$, the univariate primitive part of A is the r-variate polynomial A/a, where a is the univariate content of A.

6.4 Methods and Algorithms

In this section we briefly discuss the main algorithms that might be of interest to the user and give a sketch of the mathematical ideas behind these algorithms.

6.4.1 GCD Computations

To compute the GCD of two univariate polynomials over $R = \mathbf{Z}_p$, the algorithm `MUPGCD` may be used. Making use of the fact that if $A_1, \ldots, A_n$ is a PRS of two polynomials A and B then $\gcd(A, B) \sim A_n$, this algorithm simply computes the natural PRS of the two input polynomials and returns the monic GCD.

The GCD and cofactors of two r-variate polynomials over $\mathbf{Z}_p$ are computed by `MPGCDC` which employs evaluation homomorphisms and interpolation to reduce the problem to that of computing the GCDs of $(r-1)$-variate polynomials over $R = \mathbf{Z}_p$. `MPGCDC` proceeds recursively until it arrives at univariate polynomials whereupon `MUPGCD` is called. The GCD computed is monic.

To obtain the GCD of two r-variate integral polynomials A and B one would use the algorithm `IPGCDC` which also computes the cofactors of A and B. In this algorithm modular homomorphisms and Chinese remaindering are used to reduce the problem to GCD computations of r-variate polynomials over $R = \mathbf{Z}_p$, which is solved by `MPGCDC`.

6.4.2 Resultants

Using the algorithm suggested by the definition of the resultant, namely to construct the Sylvester matrix and compute its determinant, is not the most efficient way to proceed.

Instead, MUPRES computes the resultant of two univariate polynomials A and B over $R = \mathbf{Z}_p$ by computing the natural PRS of A and B and by using the identity

$$\operatorname{res}(A, B) = (-1)^{\nu} \left[\prod_{i=2}^{n-1} c_i^{d_{i-1}-d_{i+1}} \right] c_n^{d_{n-1}}$$

where $c_i = \operatorname{ldcf}(A_i)$, $d_i = \deg(A_i)$, $\nu = \sum_{i=1}^{k-2} d_i d_{i+1}$ and $A_1, \ldots, A_n$ is the natural PRS.

For calculating the resultant of r-variate polynomials over $R = \mathbf{Z}_p$, MPRES makes use of evaluation homomorphisms and interpolation to recursively reduce the problem to the calculation of resultants of univariate polynomials over $R = \mathbf{Z}_p$ which can be done by MUPRES.

IPRES computes the resultant of r-variate polynomials over $R = \mathbf{Z}$ by applying modular homomorphisms and Chinese remaindering to simplify the problem to resultant computations over $R = \mathbf{Z}_p$, computations which are performed by MPRES.

6.5 Functions

Integral polynomial GCDs:

`C <- IPC(r,A)` Integral polynomial content. *Given an r-variate polynomial A over $R = \mathbf{Z}$, computes the $(r-1)$-variate polynomial equal to the content of A.*

`IPCPP(r,A; C,Ab)` Integral polynomial content and primitive part. *Computes the content and the primitive part of a given polynomial over $R = \mathbf{Z}$.*

`IPGCDC(r,A,B; C,Ab,Bb)` Integral polynomial greatest common divisor and cofactors. *Given two r-variate polynomials A and B over $R = \mathbf{Z}$, computes the GCD and the cofactors of A and B.*

`IPLCPP(r,A; C,P)` Integral polynomial list of contents and primitive parts. *Given a list $(A_1, \ldots, A_n)$ of r-variate polynomials over $R = \mathbf{Z}$, computes two lists, one consisting of the contents of the A_i that have positive degree in at least one variable and another consisting of the primitive parts of the A_i that that have positive degree in the main variable.*

`Ab <- IPPP(r,A)` Integral polynomial primitive part. *Given a polynomial A over $R = \mathbf{Z}$, computes the primitive part of A.*

`IPSCPP(r,A; s,C,Ab)` Integral polynomial sign, content, and primitive part. *Computes the sign, the content and the primitive part of a given polynomial over $R = \mathbf{Z}$.*

Modular Polynomial GCDs:

`MPGCDC(r,p,A,B; C,Ab,Bb)` Modular polynomial greatest common divisor and cofactors. *Computes the GCD and cofactors of two given polynomials over $R = \mathbf{Z}_p$.*

`c <- MPUC(r,p,A)` Modular polynomial univariate content. *Computes the univariate content of an r-variate polynomial, $r \geq 2$, over $R = \mathbf{Z}_p$.*

`MPUCPP(r,p,A; a,Ab)` Modular polynomial univariate content and primitive part. *Giver an r-variate polynomial A, $r \geq 2$, computes the univariate content a of A and the univariate primitive part A/a.*

`d <- MPUCS(r,p,A,c)` Modular polynomial univariate content subroutine.

`Ab <- MPUPP(r,p,A)` Modular polynomial univariate primitive part. Given A, an r-variate polynomial over $R = \mathbf{Z}_p$, $r \geq 2$, computes the univariate primitive part of A.

`C <- MUPGCD(p,A,B)` Modular univariate polynomial greatest common divisor. *Computes the GCD of two given univariate polynomials over* $R = \mathbf{Z}_p$.

`L <- MUPSFF(p,A)` Modular univariate polynomial squarefree factorization. *Computes the squarefree factorization of a given univariate polynomial over* $R = \mathbf{Z}_p$.

Squarefree basis:

`B <- IPCSFB(r,A)` Integral polynomial coarsest squarefree basis. *Given a list* A *of positive and primitive* r*-variate polynomials over* $R = \mathbf{Z}$*, each of which is of positive degree in the main variable, computes a coarsest squarefree basis for* A.

`B <- IPFSFB(r,A)` Integral polynomial finest squarefree basis. *Given a list* A *of positive and primitive* r*-variate polynomials over* $R = \mathbf{Z}$*, each of which is of positive degree in the main variable, computes a finest squarefree basis for* A.

`B <- IPPGSD(r,A)` Integral polynomial primitive greatest squarefree divisor. *Given a polynomial* A *over* $R = \mathbf{Z}$*, computes the positive and primitive greatest squarefree divisor of the primitive part of* A.

`L <- IPSF(r,A)` Integral polynomial squarefree factorization. *Given a primitive polynomial* A*, of positive degree in the main variable, computes the squarefree factorization of* A.

`Bs <- IPSFBA(r,A,B)` Integral polynomial squarefree basis augmentation.

`B <- ISPSFB(r,A)` Integral squarefree polynomial squarefree basis.

Resultants:

`B <- IPDSCR(r,A)` Integral polynomial discriminant. *Computes the discriminant of an* r*-variate polynomial over* $R = \mathbf{Z}$*, the degree of which is greater than or equal to* 2 *in its main variable.*

`P <- IPPSC(r,A,B)` Integral polynomial principal subresultant coefficients. *Computes a list of the non-zero principal subresultant coefficients of two given* r*-variate polynomials over* $R = \mathbf{Z}$ *each of which is of positive degree in the main variable.*

`C <- IPRES(r,A,B)` Integral polynomial resultant. *Given two* r*-variate polynomials over* $R = \mathbf{Z}$*, each of which is of positive degree in the main variable, computes the* $(r-1)$*-variate polynomial over* $R = \mathbf{Z}$ *equal to their resultant.*

`IUPRC(A,B; C,R)` Integral univariate polynomial resultant and cofactor. *Given two univariate polynomials* A *and* B *over* $R = \mathbf{Z}$*, where both* A *and* B *are of positive degree, computes* $\mathrm{res}(A,B)$ *and the univariate polynomial* C *over* $R = \mathbf{Z}$ *such that for some* D*,* $AD + BC = \mathrm{res}(A,B)$ *and* $\deg(C) < \deg(A)$.

`C <- MPRES(r,p,A,B)` Modular polynomial resultant. *Given two* r*-variate polynomials over* $R = \mathbf{Z}_p$*, each of which is of positive degree in the main variable, computes the* $(r-1)$*-variate polynomial over* $R = \mathbf{Z}_p$ *equal to their resultant.*

`MUPRC(p,A,B; C,r)` Modular univariate polynomial resultant and cofactor. *Given two univariate polynomials* A *and* B *over* $R = \mathbf{Z}_p$*, where both* A *and* B *are of positive degree, computes* $\mathrm{res}(A,B)$ *and the univariate polynomial* C *over* $R = \mathbf{Z}_p$ *such that for some* D*,* $AD + BC = \mathrm{res}(A,B)$ *and* $\deg(C) < \deg(A)$.

`c <- MUPRES(p,A,B)` Modular univariate polynomial resultant. *Computes the resultant of two given univariate polynomials over* $R = \mathbf{Z}_p$*, each of which is of positive degree in the main variable.*

Polynomial Remainder Sequences:

`S <- IPRPRS(r,A,B)` Integral polynomial reduced polynomial remainder sequence. *Computes a list representing the reduced polynomial remainder sequence of two given nonzero r-variate polynomials over* $R = \mathbf{Z}$.

`S <- IPSPRS(r,A,B)` Integral polynomial subresultant polynomial remainder sequence. *Computes a list representing the subresultant polynomial remainder sequence of the first kind of two given nonzero r-variate polynomials over* $R = \mathbf{Z}$.

`S <- MPSPRS(r,p,A,B)` Modular polynomial subresultant polynomial remainder sequence. *Computes a list representing the subresultant polynomial remainder sequence of the first kind of two given nonzero r-variate polynomials over* $R = \mathbf{Z}_p$.

Extended GCDs:

`MUPEGC(p,A,B; C,U,V)` Modular univariate polynomial extended greatest common divisor. *Computes the GCD* C *of two univariate polynomials* A *and* B *over* $R = \mathbf{Z}_p$ *as well the univariate polynomials* U *and* V *such that* $AU + BV = C$.

`MUPHEG(p,A,B; C,V)` Modular univariate polynomial half-extended greatest common divisor. *Computes the GCD* C *of two univariate polynomials* A *and* B *over* $R = \mathbf{Z}_p$ *as well the univariate polynomial* V *such that* $AU + BV = C$ *for some* U.

Chapter 7

Polynomial Factorization

7.1 Mathematical Preliminaries

A non-constant polynomial $A(x_1, \ldots, x_r)$ in $R[x_1, \ldots, x_r]$, where R is a unique factorization domain, is said to be *irreducible* if A cannot be expressed as the product of two non-constant polynomials in $R[x_1, \ldots, x_r]$. The problem of factoring a polynomial $A(x_1, \ldots, x_r)$ is that of finding distinct irreducible polynomials $A_i(x_1, \ldots, x_r)$ and integers e_i, $i = 1, \ldots, k$, such that $A = A_1^{e_1} \cdots A_k^{e_k}$. Such an expression is called a *complete factorization* of A. The polynomials A_i are called the *irreducible factors* of A and the integer e_i is called the *multiplicity* of A_i.

7.2 Purpose

The SACLIB polynomial factorization package provides factorization algorithms for $R = \mathbf{Z}_p$, p a single-precision prime and $r = 1$, and for $R = \mathbf{Z}$ for $r \geq 1$. For $R = \mathbf{Z}$ one obtains the sign, integer content and positive primitive irreducible factors of A, as well as the multiplicity of each irreducible factor. The integer content is not factored. For $R = \mathbf{Z}_p$ the irreducible factors obtained are monic.

7.3 Methods and Algorithms

To factor an arbitrary univariate polynomial modulo a prime, one should first obtain a similar monic polynomial by using the algorithm `MPMON`. Having done this, one then computes the squarefree factors of the monic polynomial by using the algorithm `MUPSFF`. In order to factor each squarefree factor one would use `MUPFBL`, which implements Berlekamp's algorithm. The irreducible factors returned by `MUPFBL` are monic.

For factoring a univariate integral polynomial, `IUPFAC` first computes the squarefree factorization using the algorithm `IPSF`. The squarefree factors are in turn factored using `IUSFPF` which first computes a factorization modulo a prime and the modular factors thus obtained are then lifted by the quadratic version of the Hensel construction. `IUPFAC` returns the sign, the integer content and a list of irreducible factors, with multiplicities, of the input polynomial. The irreducible factors returned by `IUPFAC` are positive and primitive.

Multivariate integral polynomials are factored by using `IPFAC`. This algorithm first computes the content as well as the squarefree factors of the primitive part of the input polynomial and subsequently factors each of these separately. The factorization of a squarefree primitive polynomial is performed by the algorithm `ISFPF` which implements a multivariate lifting technique based on the Hensel lemma. The lifting is done one variable at a time as opposed to lifting several variables simultaneously.

If the polynomial A to be factored has rational base coefficients then it must first be converted to an integral polynomial by multiplying A by the least common multiple of the denominators of the base coefficients and then converting the polynomial thus obtained to integral representation. This can be achieved by using `IPSRP` which computes the primitive and positive integral polynomial A' as well as the rational number a such that $A = aA'$.

7.4 Functions

Factorization:

`IPFAC(r,A; s,c,L)` Integral polynomial factorization. *Factors r-variate polynomials over $\mathbf{Z}$.*

`IUPFAC(A; s,c,L)` Integral univariate polynomial factorization. *Factors univariate polynomials over $\mathbf{Z}$.*

`L <- MUPFBL(p,A)` Modular univariate polynomial factorization-Berlekamp algorithm. *Factors monic squarefree univariate polynomials over $\mathbf{Z}_p$.*

Auxiliary Functions for Factorization:

`IPCEVP(r,A; B,L)` Integral polynomial, choice of evaluation points. *Given an integral polynomial A that is squarefree in its main variable, computes integers that, when substituted for the minor variables, maintain the degree of A in the main variable and its squarefreeness.*

`b <- IPFCB(V)` Integral polynomial factor coefficient bound. *Given the degree vector of an integral polynomial A, computes an integer b such the product of the infinity norms of any divisors of A is less than or equal to 2^b times the infinity norm of A.*

`Lp <- IPFLC(r,M,I,A,L,D)` Integral polynomial factor list combine.

`B <- IPFSFB(r,A)` Integral polynomial finest squarefree basis.

`a <- IPGFCB(r,A)` Integral polynomial Gelfond factor coefficient bound.

`IPIQH(r,p,D,Ab,Bb,Sb,Tb,M,C; A,B)` Integral polynomial mod ideal quadratic Hensel lemma.

`L <- ISFPF(r,A)` Integral squarefree polynomial factorization. *Given a positive, primitive integral polynomial A that is squarefree with respect to the main variable, computes a list of the distinct positive irreducible factors of A.*

`IUPFDS(A; p,F,C)` Integral univariate polynomial factor degree set.

`IUPQH(p,Ab,Bb,Sb,Tb,M,C; A,B)` Integral univariate polynomial quadratic Hensel lemma.

`Fp <- IUPQHL(p,F,M,C)` Integral univariate polynomial quadratic Hensel lemma, list.

`L <- IUSFPF(A)` Integral univariate squarefree polynomial factorization. *Given a univariate, positive, primitive, squarefree integral polynomial A, computes a list of the positive irreducible factors of A.*

`M <- MCPMV(n,L)` Matrix of coefficients of polynomials, with respect to main variable.

`MIPISE(r,M,D,A,B,S,T,C; U,V)` Modular integral polynomial mod ideal, solution of equation.

`MIUPSE(M,A,B,S,T,C; U,V)` Modular integral univariate polynomial, solution of equation.

`MPIQH(r,p,D,Ab,Bb,Sb,Tb,M,Dp,C; A,B)` Modular polynomial mod ideal, quadratic Hensel lemma.

`Fp <- MPIQHL(r,p,F,M,D,C)` Modular polynomial mod ideal, quadratic Hensel lemma, list.

`MPIQHS(r,M,D,Ab,Bb,Sb,Tb,s,n,C; A,B,S,T,Dp)` Modular polynomial mod ideal, quadratic Hensel lemma on a single variable.

`Q <- MUPBQP(p,A)` Modular univariate polynomial Berlekamp Q-polynomials construction.

`L <- MUPDDF(p,A)` Modular polynomial distinct degree factorization. *Given a monic, squarefree polynomial A over $R = \mathbf{Z}_p$, computes a list $((n_1, A_1), \ldots, (n_k, A_k))$, where the n_i are positive integers with $n_1 < \cdots < n_k$ and each A_i is the product of all monic irreducible factors of A of degree n_i.*

`L <- MUPFS(p,A,B,d)` Modular univariate polynomial factorization, special.

Chapter 8

Real Root Calculation

8.1 Mathematical Preliminaries

Let $A(x)$ be a univariate polynomial with integer coefficients. A real number x_0 with $A(x_0) = 0$ is called a *real root* of $A(x)$. A real number x_0 is a root of $A(x)$ if and only if $A(x)$ is divisible by $(x - x_0)$, i.e. if there is a polynomial $B(x)$ with real coefficients such that $A(x) = (x - x_0)B(x)$. For any real root x_0 of $A(x)$ there is a natural number k such that $A(x)$ is divisible by $(x - x_0)^k$ but not by $(x - x_0)^{k+1}$. This number k is called the *multiplicity* of the root x_0 of $A(x)$. Roots of multiplicity 1 are called *simple roots*. An interval I containing x_0 but no other real root of $A(x)$, is called an *isolating interval* for x_0. For example, if $A(x) = x^2 - 2$, the interval $(-2, 2)$ is not an isolating interval for a real root of $A(x)$, but $(0, 1000)$ is.

8.2 Purpose

The SACLIB real root calculation package solves non-linear equations in one variable: It computes isolating intervals for the real roots of univariate integral polynomials along with the multiplicity of each root, and it refines the isolating intervals to any specified size.

8.3 Methods and Algorithms

For root isolation three methods are available. The *coefficient sign variation method* (or: *modified Uspensky method*), is based on Descartes' rule of signs. The *Collins-Loos method* is based on Rolle's theorem. *Sturm's method* is based on Sturm sequences.

Generally, the coefficient sign variation method is many times faster than the other two methods. For the coefficient sign variation method various main programs are provided to accommodate details of input and output specifications.

For the refinement of isolating intervals to any specified precision a symbolic version of Newton's method is used.

Given an arbitrary integral polynomial `IPRCH` will calculate all its real roots to specified accuracy. The multiplicity of each root is also computed. The algorithm uses the coefficient sign variation method to isolate the roots from each other and then applies Newton's method to refine the isolating intervals to the desired width.

Given a squarefree integral polynomial `IPRIM` isolates all the real roots from each other. The roots inside a specified open interval are isolated by `IPRIMO`. Both `IPRIM` and `IPRIMO` use the coefficient sign variation method. Other main algorithms which use this method are `IPRIMS` and `IPRIMW`.

The Collins-Loos method is implemented in `IPRICL`: Given an arbitrary univariate integral polynomial `IPRICL` produces a list of isolating intervals for its real roots. These intervals have the additional property that the first derivative of A is monotone on each of them.

An implementation of Sturm's method is provided by `IPRIST`: Given a squarefree univariate integral polynomial `IPRIST` produces a list of isolating intervals for its real roots.

Roots of different polynomials can be isolated from each other using the program `IPLRRI`.

Reference: Jeremy R. Johnson: *Algorithms for polynomial real root isolation.* Technical Research Report OSU-CISRC-8/91-TR21, 1991. The Ohio State University, 2036 Neil Avenue Mall, Columbus, Ohio 43210, Phone: 614-292-5813.

8.4 Definitions of Terms

binary rational number A rational number whose denominator is a power of 2.

interval A list $I = (a, b)$ of rational numbers $a \le b$. If $a = b$ the interval is called a one-point interval and it designates the set consisting of the number a. If $a < b$ it is not evident from the representation whether the endpoints are thought to be part of I or not. Therefore the specifications of the algorithms have to state whether a particular interval is meant to be an open interval, a left-open and right closed interval, a left-closed and right open interval or a closed interval.

standard interval An interval whose endpoints are binary rational numbers a, b such that $a = m/2^k$, $b = (m+1)/2^k$, k and m being positive or negative integers, or zero.

(weakly) isolating interval An interval I is called a (weakly) isolating interval for a simple real root α of the polynomial A if I contains α but no other root of A.

strongly isolating interval An isolating interval for a root α of a polynomial A is said to be strongly isolating, if the closure of I is also an isolating interval for α.

disjoint intervals Intervals are called disjoint if the sets they designate are disjoint.

strongly disjoint intervals Disjoint intervals are called strongly disjoint if their closures are disjoint.

inflectionless isolating interval An interval I with binary rational endpoints which is an isolating interval for a real root x_0 of $A(x)$ is called inflectionless if the derivative $A'(x)$ is monotone in I, i.e. if $A''(x)$ does not have a root in I except possibly x_0.

inflectionless isolation list A list of inflectionless isolating intervals.

8.5 Functions

High Precision Calculation

`L <- IPRCH(A,I,k)` Integral polynomial real root calculation, high precision. Input: any polynomial. Output: all roots or all roots in an interval.

`L <- IPRCHS(A,I,k)` Integral polynomial real root calculation, high-precision special. Input: polynomial which does not have common roots with its first or second derivative. Output: all roots or all roots in an interval.

`IPRCNP(A,I; sp,spp,J)` Integral polynomial real root calculation, Newton method preparation.

`J <- IPRCN1(A,I,s,k)` Integral polynomial real root calculation, 1 root.

Coefficient Sign Variation Method

`L <- IPRIM(A)` Integral polynomial real root isolation, modified Uspensky method.

`L <- IPRIMO(A,Ap,I)` Integral polynomial real root isolation, modified Uspensky method, open interval.

`L <- IPRIMS(A,Ap,I)` Integral polynomial real root isolation, modified Uspensky method, standard interval.

`L <- IPRIMU(A)` Integral polynomial real root isolation, modified Uspensky method, unit interval.

`L <- IPRIMW(A)` Integral polynomial real root isolation, modified Uspensky method, weakly disjoint intervals.

`L <- IPSRM(A,I)` Integral polynomial strong real root isolation, modified Uspensky method. Input: an integral polynomial without multiple roots and no roots in common with its second derivative. Output: an inflectionless isolation list for all roots or all roots in an interval.

`L <- IPSRMS(A,I)` Integral polynomial strong real root isolation, modified Uspensky method, standard interval.

Rolle's Theorem

`L <- IPRICL(A)` Integral polynomial real root isolation, Collins-Loos algorithm.

Sturm's method

`IPRIST` Integral polynomial real root isolation using a Sturm sequence.

Special

`r <- IUPRLP(A)` Integral univariate polynomial, root of a linear polynomial.

`b <- IUPRB(A)` Integral univariate polynomial root bound. Input: a univariate integral polynomial A. Output: a binary rational number, power of 2, which is greater than the absolute value of any root of A.

`M <- IPLRRI(L)` Integral polynomial list real root isolation. Input: a list of integral polynomials without multiple roots and without common roots. Output: a list of strongly disjoint isolating intervals in ascending order for all the roots of all the input polynomials – each interval is listed with the polynomial of which it isolates a root.

`Is <- IUPIIR(A,I)` Integral univariate polynomial isolating interval refinement.

Low-Level Functions

`IPRRS(A1,A2,I1,I2; Is1,Is2,s)` Integral polynomial real root separation.

`IPRRLS(A1,A2,L1,L2; Ls1,Ls2)` Integral polynomial real root list separation.

`L <- IPRRII(A,Ap,d,Lp)` Integral polynomial real root isolation induction.

`Is <- IPRRRI(A,B,I,s1,t1)` Integral polynomial relative real root isolation.

`J <- IPSIFI(A,I)` Integral polynomial standard isolating interval from isolating interval.

`IPIIWS(A,L)` Integral polynomial isolating intervals weakly disjoint to strongly disjoint.

`IPPNPRS` Integral polynomial primitive negative polynomial remainder sequence.

`k <- IPVCHT(A)` Integral polynomial variations after circle to half-plane transformation.

`B <- IUPCHT(A)` Integral univariate polynomial circle to half-plane transformation.

`n <- IUPVAR(A)` Integral univariate polynomial variations.

`v <- IUPVOI(A,I)` Integral univariate polynomial, variations for open interval.

`v <- IUPVSI(A,I)` Integral univariate polynomial, variations for standard interval.

Chapter 9

Algebraic Number Arithmetic

9.1 Mathematical Preliminaries

An algebraic number is a number that satisfies a rational polynomial equation. An algebraic number α is represented by an irreducible polynomial, $A(x)$, such that $A(\alpha) = 0$. A real algebraic number, is a real number that is also an algebraic number, and it is represented by an irreducible polynomial and an isolating interval to distinguish it from its real conjugates. The collection of algebraic numbers forms a field containing the real algebraic numbers as a subfield. Since $A(x)$ is irreducible, the extension field $Q(\alpha)$ obtained by adjoining α to the rational number field is isomorphic to $Q[x]/(A(x))$ and elements of $Q(\alpha)$ are represented by polynomials whose degrees are less than the degree of $A(x)$. If α is real then $Q(\alpha)$ is an ordered field and sign computations can be performed using the isolating interval for α.

9.2 Purpose

The SACLIB algebraic number arithmetic package provides algorithms for performing arithmetic with algebraic numbers, with elements of an algebraic number field, and with polynomials whose coefficients belong to an algebraic number field. There are algorithms for computing the gcd of two polynomials with algebraic number coefficients and for factoring a polynomial with algebraic number coefficients. Algorithms are also provided for performing sign computations in a real algebraic number field and for isolating the real roots of a polynomial with real algebraic number coefficients.

9.3 Methods and Algorithms

Algorithms for performing algebraic number arithmetic use resultant computations. Let $A(x) = \sum_{i=0}^{m} a_i x^i = a_m \prod_{i=1}^{m}(x - \alpha_i)$ be the integral minimal polynomial for $\alpha = \alpha_1$ and let $B(y) = \sum_{j=0}^{n} b_j y^j = b_n \prod_{j=1}^{n}(y - \beta_j)$ be the integral minimal polynomial for $\beta = \beta_1$. The minimal polynomial for $\alpha + \beta$ is a factor of $\mathrm{res}_x(A(x), B(y - x))$ and the minimal polynomial for $\alpha \cdot \beta$ is a factor of $\mathrm{res}_x(A(x), x^n B(y/x))$. If α and β are real algebraic numbers, the particular factor can be found by using the isolating intervals for α and β. The algorithms `ANSUM` and `ANPROD` use these ideas to perform addition and multiplication in the field of real algebraic numbers. Subtraction and division can be performed by negating and adding and inverting and multiplying respectively. The minimal polynomial of $-\alpha$ is $A(-x)$ and the minimal polynomial of $1/\alpha$ is $x^m A(1/x)$. These operations are provided by the algorithms `IUPNT` and `PRT` in the polynomial arithmetic system.

Let $Q(\alpha)$ be the extension field of the rationals obtained by adjoining the algebraic number α. Arithmetic in $Q(\alpha)$ is performed using the isomorphism $Q(\alpha) \cong Q[x]/(A(x))$.

Elements of $Q(\alpha)$ are represented by polynomials whose degrees are less than the degree of the minimal polynomial of α and addition and multiplication are performed using polynomial multiplication and addition modulo the minimal polynomial. Inverses of elements of $Q(\alpha)$ are calculated by using a resultant computation. If $B(x)$ is the polynomial representing $\beta = B(\alpha)$ and $R = \mathrm{res}(A(x), B(x))$, then there exist polynomials $S(x)$ and $T(x)$ such that $A(x)S(x) + B(x)T(x) = R$. Since the minimal polynomial $A(x)$ is irreducible, the resultant does not equal zero and $B(\alpha)^{-1} = T(\alpha)/R$. The algorithm **AFINV** uses this approach to compute inverses of elements of $Q(\alpha)$.

If $\alpha \in I$ is a real algebraic number, then the field $Q(\alpha)$ can be ordered. The algorithm **AFSIGN** computes the sign of an element of $Q(\alpha)$. The sign of $\beta = B(\alpha)$ is determined by refining the isolating interval, I, for α until it can be shown that $B(y)$ does not contain any roots in I. If there are no roots of $B(y)$ in the isolating interval I, then the sign of $B(\alpha)$ is equal to the sign of $B(y)$ for any $y \in I$. The algorithm **AFSIGN** uses this approach and Descartes' rule of signs to determine how much to refine I.

SACLIB provides algorithms for computing with polynomials whose coefficients belong to an algebraic field $Q(\alpha)$. Besides basic arithmetic, there are algorithms for polynomial gcd computation, factorization, and real root isolation. The algorithm **AFUPGC** uses the monic PRS to compute the gcd of two univariate polynomials. The algorithms **AFUPGS**, **AFUPSF**, and **AFUPCSFB** use this algorithm to compute greatest squarefree divisors, squarefree factorization, and a squarefree basis respectively. Algorithms are also provided to isolate the real roots of a polynomial whose coefficients belong to a real algebraic number field. Both the Collins-Loos algorithm (**AFUPRICL**) and the coefficient sign variation method (**AFUPRICS**) are provided.

An algebraic number may arise as a solution of a polynomial with algebraic number coefficients. The norm can be used to find a defining polynomial with integral coefficients. Let $B(\alpha, y)$ be a polynomial with coefficients in $Q(\alpha)$. The norm of $B(\alpha, y)$ is the rational polynomial $\mathrm{Norm}(B\alpha, y)) = \prod_{i=1}^{m} B(\alpha_i, y)$. The norm can be computed with the resultant computation $res_x(A(x), B(x, y))$ which produces a polynomial similar to the norm. The algorithm **AFPNORM** uses this approach to compute the norm. The algorithm **AFPNIP** returns the list of irreducible factors of the norm. If α is a real algebraic number, the isolating interval for α can be used to select the appropriate irreducible factor of the norm. This is done by the algorithm **AFUPMPR**.

As a special case, the minimal polynomial of $\beta = B(\alpha)$ can be computed by calculating the norm of the linear polynomial $y - B(\alpha)$. Since $y - B(\alpha)$ is irreducible the norm is a power of an irreducible polynomial, and the minimal polynomial can be obtained with a greatest squarefree divisor computation. The algorithm **ANFAF** uses this approach to convert the representation of an element of an algebraic number field to its representation as an algebraic number.

The algorithm **AFUPFAC** uses the norm to factor a squarefree polynomial whose coefficients belong to an algebraic number field. Let $B^*(y) = \mathrm{Norm}(B(\alpha, y))$, and let $\prod_{i=1}^{t} B_i^*(y)$ be the irreducible factorization of $B^*(y)$. Provided the norm is squarefree the irreducible factorization of $B(\alpha, y) = \prod_{i=1}^{t} \gcd(B(\alpha, y), B_i^*(y))$. If $B^*(y)$ is not squarefree, a translation, $B(\alpha, y - s\alpha)$, is computed whose norm is squarefree. The factorization of $B(\alpha, y)$ can be recovered from the factorization of the translated polynomial.

SACLIB also provides an algorithm for computing a primitive element of a multiple extension field. Let α and β be algebraic numbers and consider the multiple extension field $Q(\alpha, \beta)$. The primitive element theorem states that there exists a primitive element γ such that $Q(\alpha, \beta) = Q(\gamma)$. The algorithms **ANPEDE** and **ANREPE** provide a constructive version of this theorem.

References: R. G. K. Loos: *Computing in Algebraic Extensions*, In "Computer Algebra, Symbolic and Algebraic Computation", pages 173–187.

Jeremy R. Johnson: *Algorithms for polynomial real root isolation.* Technical Research Report OSU-CISRC-8/91-TR21, 1991. The Ohio State University, 2036 Neil Avenue Mall, Columbus, Ohio 43210, Phone: 614-292-5813.

Barry Trager: *Algebraic Factoring and Rational Function Integration*, In "SYMSAC '81: Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation", pages 219–226.

9.4 Definitions of Terms

algebraic number A solution of a rational polynomial equation. An algebraic number α is represented either by a rational minimal polynomial or an integral minimal polynomial.

algebraic integer A solution of a monic integral polynomial equation.

real algebraic number A real number that is also an algebraic number. A real algebraic number is represented by an integral minimal polynomial and an acceptable isolating interval.

rational minimal polynomial The rational minimal polynomial for an algebraic number α is the unique monic, irreducible rational polynomial $A(x)$ such that $A(\alpha) = 0$.

integral minimal polynomial The integral minimal polynomial for an algebraic number α is the unique, positive, primitive, integral polynomial $A(x)$ such that $A(\alpha) = 0$.

acceptable isolating interval an isolating interval, I, for a real algebraic number α, where I is either a left-open and right-closed standard interval or a one-point interval.

algebraic field element an element of the extension field $Q(\alpha)$. $\beta \in Q(\alpha)$ is represented by a list $(r, B(y))$, where $\beta = rB(\alpha)$ and r is a rational number and $B(y)$ is a primitive integral polynomial whose degree is less than the degree of the minimal polynomial of α.

9.5 Representation

There are several different representations for elements of $Q(\alpha)$. Let $A(x)$ be the integral minimal polynomial for an algebraic number α with $\deg(A(x)) = m$. An element β of $Q(\alpha)$ can be uniquely represented by:

1. A rational polynomial, $B(x)$, whose degree is less than m and such that $B(\alpha) = \beta$.

2. A pair $(r, \overline{B}(x))$, where r is a rational number, $\overline{B}(x)$ is a positive primitive integral polynomial, and $\beta = B(\alpha) = r\overline{B}(\alpha)$.

The default representation is (2). The algorithm `AFCR` converts representation (1) to (2), and the algorithm `AFICR` converts representation (2) to (1).

Let $Z[\alpha]$ denote the Z-module with basis $1, \alpha, \alpha^2, \ldots, \alpha^{m-1}$. Elements of $Z[\alpha]$ are represented by integral polynomials whose degree is less than m. If α is an algebraic integer, then $Z[\alpha]$ is a ring. If an algorithm does not require division or reduction by the minimal polynomial, operations in $Q(\alpha)$ can be replaced with operations in $Z[\alpha]$. When this is possible, efficiency is gained by using the integral representation $Z[\alpha]$. An important example is polynomial real root isolation. Let $P(\alpha, y)$ be a polynomial in $Q(\alpha)[y]$ and let d be the least common multiple of the denominators of the coefficients of $P(\alpha, y)$. Then $dP(\alpha, y)$ is in $Z[\alpha, y]$ and has the same roots as $P(\alpha, y)$. Moreover, all of the operations of the coefficient sign variation method for real root isolation can be performed in $Z[\alpha]$.

The name of algorithms which operate in $Z[\alpha]$ begin with the letters `AM`. The algorithm `AMPSAFP`(r, P) computes a polynomial $\overline{P} \in Z[\alpha, X1, \ldots, Xr]$ which is similar to the polynomial $P \in Q(\alpha)[X1, \ldots, Xr]$. The algorithm `AIFAN` computes an algebraic integer $\overline{\alpha}$ such that $Q(\alpha) = Q(\overline{\alpha})$.

9.6 Functions

Algebraic Field Arithmetic

`c <- AFDIF(a,b)` Algebraic number field element difference.

`b <- AFINV(M,a)` Algebraic number field element inverse.

`b <- AFNEG(a)` Algebraic number field negative.

`c <- AFPROD(P,a,b)` Algebraic number field element product.

`c <- AFQ(M,a,b)` Algebraic number field quotient.

`c <- AFSUM(a,b)` Algebraic number field element sum.

Algebraic Number Arithmetic

`ANIIPE(M,I,N,J,t,L; S,k,K)` Algebraic number isolating interval for a primitive element.

`ANPROD(A,I,B,J; C,K)` Algebraic number product.

`ANSUM(A,I,B,J;C,K)` Algebraic number sum.

`ANPEDE(A,B;C,t)` Algebraic number primitive element for a double extension.

`b <- ANREPE(M,A,B,t)` Algebraic number represent element of a primitive extension.

Algebraic Polynomial Arithmetic

`C <- AFPAFP(r,M,a,B)` Algebraic number field polynomial algebraic number field element product.

`C <- AFPAFQ(r,M,A,b)` Algebraic number field polynomial algebraic number field element quotient.

`C <- AFPDIF(r,A,B)` Algebraic number field polynomial difference.

`Ap <- AFPMON(r,M,A)` Algebraic number field polynomial monic.

`B <- AFPNEG(r,A)` Algebraic number field polynomial negative.

`C <- AFPPR(r,M,A,B)` Algebraic number field polynomial product.

`AFPQR(r,M,A,B; Q,R)` Algebraic number field polynomial quotient and remainder.

`C <- AFPSUM(r,A,B)` Algebraic number field polynomial sum.

Algebraic Polynomial Differentiation and Integration

`B <- AFPDMV(r,M,A)` Algebraic number field polynomial derivative, main variable.

`B <- AFPINT(r,M,A,b)` Algebraic number field polynomial integration.

`B <- AMPDMV(r,M,A)` Algebraic module polynomial derivative, main variable.

Algebraic Polynomial Factorization

`F <- AFUPFAC(M,B)` Algebraic number field univariate polynomial factorization.

`L <- AFUPSF(M,A)` Algebraic number field univariate polynomial squarefree factorization.

Algebraic Polynomial Greatest Common Divisors

`AFUPGC(M,A,B; C,Ab,Bb)` Algebraic number field univariate polynomial greatest common divisor and cofactors.

`B <- AFUPGS(M,A)` Algebraic number field polynomial greatest squarefree divisor.

Algebraic Polynomial Norm Computation

`L <- AFPNIP(M,A)` Algebraic number field polynomial normalize to integral polynomial.

`Bs <- AFPNORM(r,M,B)` Algebraic number field polynomial norm.

Algebraic Polynomial Squarefree Basis

`B <- AFUPCSFB(M,A)` Algebraic number field univariate polynomial coarsest squarefree basis.

`Bs <- AFUPSFBA(M,A,B)` Algebraic number field univariate polynomial squarefree basis augmentation.

`B <- AFUSPSFB(M,A)` Algebraic number field univariate squarefree polynomial squarefree basis.

Algebraic Polynomial Substitution and Evaluation

`C <- AFPCMV(r,M,A,B)` Algebraic number field polynomial composition in main variable.

`B <- AFPEMV(r,M,A,a)` Algebraic number field polynomial evaluation of main variable.

`B <- AFPEV(r,M,A,i,a)` Algebraic number field polynomial evaluation.

`B <- AFPME(r,M,A,b)` Algebraic number field polynomial multiple evaluation.

`s <- AFUPSR(M,I,A,c)` Algebraic number field univariate polynomial, sign at a rational point.

`s <- AMUPBES(M,I,A,c)` Algebraic module univariate polynomial, binary rational evaluation of sign.

`s <- AMUPSR(M,I,A,c)` Algebraic module univariate polynomial, sign at a rational point.

`B <- IPAFME(r,M,A,b)` Integral polynomial, algebraic number field multiple evaluation.

`B <- RPAFME(r,M,A,b)` Rational polynomial, algebraic number field multiple evaluation.

Algebraic Polynomial Transformations

`B <- AMUPBHT(A,k)` Algebraic module univariate polynomial binary homothetic transformation.

`B <- AMUPNT(A)` Algebraic module univariate polynomial negative transformation.

`B <- AMUPTR(A,h)` Algebraic module univariate polynomial translation.

`B <- AMUPTR1(A)` Algebraic module univariate polynomial translation by 1.

Conversion

`Ap <- AFCR(A)` Algebraic number field element convert representation.

`a <- AFFINT(M)` Algebraic number field element from integer.

`a <- AFFRN(R)` Algebraic number field element from rational number.

`Ap <- AFICR(A)` Algebraic number field element inverse convert representation.

`B <- AFPCR(r,A)` Algebraic number field polynomial convert representation.

`B <- AFPFIP(r,A)` Algebraic number field polynomial from integral polynomial.

`B <- AFPFRP(r,A)` Algebraic number field polynomial from rational polynomial.

`B <- AFPICR(r,A)` Algebraic number field polynomial inverse convert representation.

`AIFAN(M; mh,Mh)` Algebraic integer from algebraic number.

`B <- AMPSAFP(r,A)` Algebraic module polynomial similar to algebraic field polynomial.

`ANFAF(M,I,a; N,J)` Algebraic number from algebraic number field element.

Input/Output

`AFDWRITE(M,I,b,n)` Algebraic number field, decimal write.

`AFPWRITE(r,A,V,v)` Algebraic number field polynomial write.

`AFUPWRITE(A,vA,vc)` Algebraic number field univariate polynomial write.

`AFWRITE(A,v)` Algebraic field element write.

`ANDWRITE(M,I,n)` Algebraic number decimal write.

Real Algebraic Number Sign and Order Computation

`t <- AFCOMP(M,I,a,b)` Algebraic number field comparison.

`s <- AFSIGN(M,I,a)` Algebraic number field sign.

`s <- AMSIGN(M,I,a)` Algebraic module sign.

`AMSIGNIR(M,I,a;s,Is)` Algebraic module sign, interval refinement.

Real Algebraic Polynomial Real Root Isolation

`N <- AFUPBRI(M,I,L)` Algebraic number field univariate polynomial basis real root isolation.

`AFUPMPR(M,I,B,J,L; Js,j)` Algebraic number field polynomial minimal polynomial of a real root.

`b <- AFUPRB(M,I,A)` Algebraic number field univariate polynomial root bound.

`L <- AFUPRICL(M,I,A)` Algebraic number field univariate polynomial real root isolation, Collins-Loos algorithm.

`L <- AFUPRICS(M,I,A)` Algebraic number field univariate polynomial real root isolation, coefficient sign variation method.

`a <- AFUPRL(M,A)` Algebraic number field univariate polynomial, root of a linear polynomial.

`n <- AFUPVAR(M,I,A)` Algebraic number field univariate polynomial variations.

`AMUPMPR(M,I,B,J,L; Js,j)` Algebraic module univariate polynomial minimal polynomial of a real root.

`L <- AMUPRICS(M,I,A)` Algebraic module univariate polynomial real root isolation, coefficient sign variation method.

`AMUPRICSW(M,I,A;L,Is)` Algebraic module univariate polynomial real root isolation, coefficient sign variation method, weakly disjoint intervals.

`AMUPRINCS(M,I,A,a,b;L,Is)` Algebraic module univariate polynomial root isolation, normalized coefficient sign variation method.

`AMUPVARIR(M,I,A; n,J)` Algebraic module univariate polynomial variations, interval refinement.

Real Algebraic Polynomial Real Root Refinement

`Js <- AFUPIIR(M,I,B,J)` Algebraic number field polynomial isolating interval refinement.

`AFUPIIWS(M,I,A,L)` Algebraic number field univariate polynomial isolating intervals weakly disjoint to strongly disjoint.

`AFUPRLS(M,I,A1,A2,L1,L2; Ls1,Ls2)` Algebraic number field univariate polynomial real root list separation.

`Js <- AFUPRRI(M,I,A,B,J,s1,t1)` Algebraic number field univariate polynomial relative real root isolation.

`AFUPRRS(M,I,A1,A2,I1,I2; Is1,Is2,s)` Algebraic number field univariate polynomial real root separation.

`Js <- AMUPIIR(M,I,B,J)` Algebraic module polynomial isolating interval refinement.

`AMUPIIWS(M,I,A,L)` Algebraic module univariate polynomial isolating intervals weakly disjoint to strongly disjoint.

`AMUPRLS(M,I,A1,A2,L1,L2; Ls1,Ls2)` Algebraic module univariate polynomial real root list separation.

`AMUPRRS(M,I,A1,A2,I1,I2; Is1,Is2,s)` Algebraic module univariate polynomial real root separation.

Appendix A

Calling SACLIB Functions from C

This chapter describes how the SACLIB environment has to be set up for SACLIB functions to work correctly. We will start with a quick introduction to the basics using a sample program in Section A.1. In Section A.2 we describe the steps necessary for combining dynamic allocation with SACLIB list processing. Special care has to be taken with SACLIB data structures addressed by global variables. This is explained in Section A.3. Finally, Section A.4 describes how SACLIB can be initialized without using `sacMain()`, and Section A.5 gives some information on error handling in SACLIB.

A.1 A Sample Program

Figure A.1 shows the basic layout of a program using SACLIB functions.

Note that the only thing which is different from ordinary C programs are the `#include "saclib.h"` statement and the fact that the main routine is called `sacMain` instead of `main`.

One important point is that the `argc` and `argv` variables passed to `sacMain` will not contain all command line parameters. Parameters starting with "`+`" are filtered out and used for initializing some SACLIB global variables. Information on these parameters is written out when a program is called with the parameter "`+h`".

In Section A.4 we give some more details on the initializations done before `sacMain` is called.

A.2 Dynamic Memory Allocation in SACLIB

When one needs to *randomly* (as opposed to *sequentially*) access elements in a data structure, one may prefer to use arrays instead of lists. If the size and the number of these arrays is determined at runtime, they have to be dynamically allocated. Furthermore, one may need to mix them with lists, in which case the garbage collector must be able to handle them.

The concept of the GCA (Garbage Collected Array) handle provides this kind of dynamic data structure.

Nevertheless it is recommended to first check whether it might be possible to reformulate the algorithm so that lists can be used instead of arrays. In many cases one uses arrays only because one is more familiar with them, although lists may be better suited to the problem at hand.

The following functions are to be used for initializing GCA handles and for accessing the elements of the corresponding arrays:

```
#include "saclib.h"

int sacMain(argc, argv)
        int argc;
        char **argv;
{
        Word    I1,I2,I3,t;
        Word    i,n;

Step1:  /* Input. */
        SWRITE("Please enter the first integer: "); I1 = IREAD();
        SWRITE("Please enter the second integer: "); I2 = IREAD();
        SWRITE("How many iterations? "); n = GREAD();

Step2:  /* Processing. */
        t = CLOCK();
        for (i=0; i<n; i++)
          I3 = IPROD(I1,I2);
        t = CLOCK() - t;

Step3:  /* Output. */
        IWRITE(I1); SWRITE(" * "); IWRITE(I2); SWRITE(" =\n"); IWRITE(I3);
        SWRITE("\nRepeating the above computation "); GWRITE(n);
SWRITE(" times took\n"); GWRITE(t); SWRITE(" milliseconds.\n");

Return:
return(0);
}
```

Figure A.1: A sample program.

`A <- GCAMALLOC(s,f)` is used for memory allocation. It takes a `BETA`-digit giving the size of the array in `Word`s as input, uses `malloc()` to allocate the array, and returns a GCA handle (a `Word`). *This GCA handle is not a C pointer to the array so you cannot address the elements of the array in C-style using this handle.* Rather, it can be used to store a reference to the array in SACLIB lists.

The second parameter to `GCAMALLOC()` may take one of the following two values (which are constants defined in "saclib.h"):

- `GC_CHECK` ... This will cause the garbage collector to check the contents of the array for list or GCA handles.
- `GC_NO_CHECK` ... With this setting, the garbage collector will ignore the contents of the array. Therefore, `GC_NO_CHECK` should only be used if it is guaranteed that the array will never contain list or GCA handles (e.g. if it is used to store `BETA`-digits).

If you are not sure which one to choose, use `GC_CHECK`, as inappropriate use of `GC_NO_CHECK` may cause the program to crash.

`GCASET(A,i,a)` sets the `i`-th element of the array referenced by the GCA handle `A` to the value `a`. Here, `a` can be any `Word`.

`a <- GCAGET(A,i)` returns the value of the `i`-th element of the array referenced by the GCA handle `A`.

Figure A.2 shows how the mechanism of GCA handles is used.

```
        .
        .
        .
        Word   A, L, I,i;
        .
        .
        .
Step2:  /* Here we do some allocation. */
        L = NIL;
        do {
          SWRITE("Enter an integer (0 to quit): "); I = IREAD();
          A = GCAMALLOC(10,GC_CHECK);
          for (i=0; i<10; i++)
            GCASET(A,i,IDPR(I,i+1));
          L = COMP(A,L);
        }
        until (ISZERO(I));
        .
        .
        .
```

Figure A.2: Sample code using GCA handles.

The code inside the `do/until` loop reads an integer `I`, allocates an array `A` of 10 `Words`, stores the value $\mathtt{I} * (\mathtt{i} + 1)$ at position `i` in the array using `GCASET()`, and then appends a new element containing the GCA handle of the array `A` to the beginning of the list `L`.

Always remember that GCA handles must be used whenever you want to store references to SACLIB structures (i.e. lists) in dynamically allocated memory. Using the standard UNIX function `malloc()` may crash your program sometime after a garbage collection or at least cause some strange bugs.

Furthermore, GCA handles are also implemented in such a way that they can be used as input to list processing functions in places where objects are regarded as data. E.g. in the `COMP()` function, a GCA handle can be used as the first argument (the element to be appended to the list) but not as the second argument (the list being appended to). Nevertheless note that the functions `LWRITE()`, `EXTENT()`, and `ORDER()` are not defined for lists containing GCA handles.

There are two more functions taking GCA handles as input. *It is not recommended to call these functions directly.* They are listed here only for completeness.

`p <- GCA2PTR(A)` gives access to the array referenced by a GCA handle. It takes a GCA handle as input and returns a C pointer to the array of `Words` allocated by a previous call to `GCAMALLOC()`. *This C pointer must not be used as input to* SACLIB *functions or stored in* SACLIB *lists.* Rather, it can be used to address the elements of the array directly.

Note that this is *not* the recommended way of accessing array elements. If you overwrite the variable containing the GCA handle (e.g. an optimizing compiler might remove it if it is not used anymore), you can still access the array using the C pointer, but the garbage collector will free the allocated memory the next time it is invoked.

`GCAFREE(A)` can be used to explicitly free the memory allocated by `GCAMALLOC()`. It takes a GCA handle as input which becomes invalid after the call.

You should consider calling GCAFREE() only in cases where you are sure you will not need the memory referenced by a GCA handle any more and want to deallocate it immediately instead of putting this off until the next garbage collection or until the SACLIB cleanup.

A.3 Declaring Global Variables to SACLIB

If you are using global variables, arrays, or structures containing SACLIB list or GCA handles other than those defined within SACLIB (in "external.c"), you have to make them visible to the garbage collector. This is done by the function GCGLOBAL().

Figure A.3 shows how these macros are used:

```
#include "saclib.h"

Word      GL = NIL;
Word      GA = NIL;

char      buffer[81];
int       flag;


int sacMain(argc, argv)
        int argc;
        char **argv;
{
 ... /* Variable declarations. */

Step1:  /* Declare global variables. */
        GCGLOBAL(&GL);
        GCGLOBAL(&GA);

Step2:  /* Initialize global variables. */
        GA = GCAMALLOC(10,GC_CHECK);
        .
        .
        .
```

Figure A.3: Declaring global variables.

First two global variables `GL` and `GA` of type `Word`, a global array `buffer` of 81 characters, and a global variable `flag` of type `int` are declared.

The variables `GL` and `GA` are declared to the garbage collector by calls to `GCGLOBAL()` *before* they are initialized. Note that for the variables `buffer` and `flag` this is not necessary because they will not hold SACLIB list or GCA handles at any time during program execution.

Calling `GCGLOBAL` on a pointer to a global variable tells the garbage collector not to free cells or arrays referenced by the corresponding variable. You should be careful about not missing any global variables which ought to be declared: while declaring too much does not really matter, declaring too little will cause weird bugs and crashes ...

A.4 Initializing SACLIB by Hand

If it is desired to have complete control over command line parameters or if SACLIB is used only as part of some bigger application, then the necessary initializations can also be done directly without using `sacMain()`.

There are three functions which are of interest:

`ARGSACLIB(argc,argv;ac,av)` does argument processing for SACLIB command line arguments. These must start with a "+" and are used to set various global variables. The argument "+h" causes a usage message to be printed (by `INFOSACLIB()`). Then the program is aborted.

`ARGSACLIB` takes the `argc` and `argv` parameters of `main()` as input. It returns the number of non-SACLIB command line arguments in `ac` and a pointer to an array of non-SACLIB command line arguments in `av`. This means that the output of `ARGSACLIB()` is similar to `argc` and `argv` with the exception that all arguments starting with "+" have been removed.

`BEGINSACLIB(p)` initializes SACLIB by allocating memory, setting the values of various global variables, etc. It must be passed the address of a variable located on the stack before any variable containing SACLIB structures such as lists or GCA handles. One variable which fulfills this requirement is `argc`, for example.

`ENDSACLIB(f)` frees the memory allocated by `BEGINSACLIB()`. It must be passed one of the following values (which are constants defined in "saclib.h"):

- `SAC_FREEMEM` ... This will cause it to also free all remaining memory allocated by `GCAMALLOC()`.
- `SAC_KEEPMEM` ... This will cause it not to free the remaining memory allocated by `GCAMALLOC()`. Nevertheless, all GCA handles become invalid after `ENDSACLIB()` has been called, so the memory can only be accessed by C pointers which were initialized by calls to `GCA2PTR()` *before* calling `ENDSACLIB()`. Furthermore, if any of the arrays contains list or GCA handles, these will also become invalid, so keeping the allocated memory only makes sense when the arrays contain `BETA`- or `GAMMA`-digits.

  Deallocation then has to be done by the standard UNIX function `free()`, because `GCAFREE()` only works when the SACLIB environment is valid.

Figure A.4 gives an example of how the SACLIB environment can be initialized, used, and removed inside a function.

The function `symbolic_computation()` in this example encapsulates SACLIB as part of some bigger application whose main routine is `main()` instead of `sacMain()`. From Step 2 on the SACLIB environment is initialized and any SACLIB function may be used. Outside the area enclosed by `BEGINSACLIB / ENDSACLIB`, calls to SACLIB functions may crash the program.

A.5 SACLIB Error Handling

SACLIB functions do not check whether the parameters passed to them are correct and fulfill their input specifications. Calling a function with invalid inputs will most probably cause the program to crash instead of aborting in a controlled way.

Nevertheless, there are situations where SACLIB functions may fail and exit the program cleanly with an error message. For example, this is the case when an input functions discovers a syntax error.

```
#include "saclib.h"

void symbolic_computation()
{
        Word    stack;

Step1:  /* Initialise SACLIB. */
        BEGINSACLIB(&stack);

Step2:  /* Use SACLIB. */
        .
        .
        .

Step3:  /* Remove SACLIB. */
        ENDSACLIB(SAC_FREEMEM);
}
```

Figure A.4: Sample code for initializing SACLIB by hand.

All error handling (i.e. writing a message and aborting the program) is done by the function `FAIL()`. If some more sophisticated error processing is desired, the simplest way is to replace it by a custom written function.

A.6 Compiling

The SACLIB header files must be visible to the compiler and the compiled SACLIB library must be linked. How this is done is explained in the "Addendum to the SACLIB User's Guide", which should be supplied by the person installing SACLIB.

A point worth mentioning is the fact that several SACLIB functions are also defined as macros. By default, the macro versions are used, but there is a constant for switching on the C function versions: `NO_SACLIB_MACROS` switches off all macros except for `FIRST`, `RED`, `SFIRST`, `SRED`, `ISNIL`, `GCASET`, `GCAGET`. These elementary list and GCA functions are always defined as macros.

If you want to use this constant, you must add the statement

```
#define NO_SACLIB_MACROS
```

before you include the file "saclib.h".

Alternatively, you can use the "`-D`" option of the C compiler (see the "Addendum to the SACLIB User's Guide" for more information).

Appendix B

ISAC: An Interactive Interface to SACLIB

B.1 What is ISAC?

ISAC is a small experimental interactive interface to SACLIB, allowing simple `read--eval--write` cycles of interaction.

The system is designed and implemented in the most straightforward way, so that its source code can be used as an example or a tutorial for those who want to quickly write an interactive test environment for their SACLIB based functions or intend to develop professional interfaces to SACLIB.

B.2 Supported SACLIB Algorithms

All the SACLIB library algorithms and macros are accessible. `NIL` and `BETA` are available as constants.

B.3 Command Line Options

ISAC takes the standard SACLIB command line options for initializing various global variables. In order to find out what is available, issue the command

```
isac +h
```

B.4 Interface Functionality

An ISAC session consists of one or more statements. Every statement must end with a semicolon '`;`'. A statement can be one of the three kinds:

- command
- call
- assignment

The commands supported in this version are:

`quit;`	For quitting the session.
`vars;`	For displaying the contents of the variables. Values are displayed in internal SACLIB format.

`help [algName];`	For displaying a general help or an algorithm. For example, in order to display the algorithm `IPROD`, issue the the command: `help IPROD;`
`view algName;`	For displaying an algorithm with the editor vi(1).
`save fileName;`	For saving the current state of the session (i.e. the variable binding) to a file.
`restore fileName;`	For restoring the state of a session from a file.

A call statement is a call to any procedures or functions in the SACLIB library. For example,

```
IPFAC(r,A; s,c,F);
IPWRITE(r,IPSUM(r,A,B),V);
```

An assignment statement is of the form:

```
var := expression;
```

For example,

```
A := IPROD(a,ISUM(b,c));
a := 2 * 3 + 4;
a := 3 % 2;
```

B.5 Interface Grammar

Below we give a context-free grammar for a session. We have followed the following conventions:

- Upper-case strings and quoted strings denote tokens,
- Lower-case strings denote non-terminals.

```
session
        : statement
        | session statement
        ;

statement
        : command ';'
        | proc_call ';'
        | assignment ';'
        ;

command
        : IDENT
        | IDENT CMDARGS
        ;

proc_call
        : IDENT '(' proc_arg_star ')'
        ;

assignment
        : IDENT ':=' expr
```

```
        ;

proc_arg_star
        : val_star
        | val_star ';' ref_star
        ;

val_star
        : /* empty */
        | val_plus
        ;

val_plus
        : expr
        | val_plus ',' expr
        ;

ref_star
        : /* empty */
        | ref_plus
        ;

ref_plus
        : ref
        | ref_plus ',' ref
        ;

ref
        : IDENT

expr
        : expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | expr '%' expr
        | '+' expr
        | '-' expr
        | '(' expr ')'
        | func_call
        | atom
        ;

func_call
        : IDENT '(' func_arg_star ')'
        ;

func_arg_star
        : val_star
        ;

atom
        : IDENT
        | INTEGER
```

;

Appendix C

Notes on the Internal Workings of SACLIB

C.1 Lists, GCA Handles, and Garbage Collection

C.1.1 Implementation of Lists

When SACLIB is initialised, the array `SPACE` containing `NU` + 1 `Words` is allocated from the heap. This array is used as the memory space for list processing. Lists are built from *cells*, which are pairs of consecutive `Words` the first of which is at an odd position in the `SPACE` array. *List handles* ("pointers" to lists) are defined to be `BETA` plus the index of the first cell of the list in the `SPACE` array with the handle of the empty list being `NIL` (which is equal to `BETA`). Figure C.1 shows the structure of the `SPACE` array.

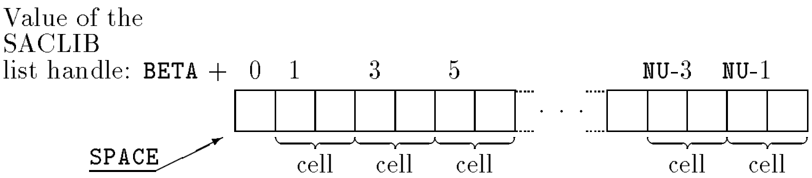

Figure C.1: The `SPACE` array.

The first `Word` of each cell is used to store the handle of the next cell in the list (i.e. the value returned by `RED()`), while the second `Word` contains the data of the list element represented by the cell (i.e. the value returned by `FIRST()`). Figure C.2 gives a graphical representation of the cell structure for a sample list. The arrows stand for list handles.

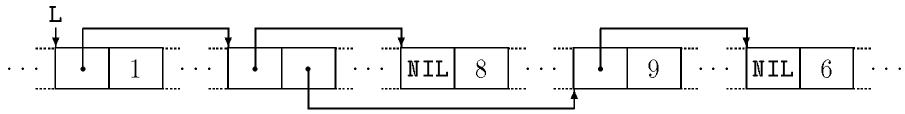

Figure C.2: The cell structure of the list `L` $= (1, (9, 6), 8)$.

As already mentioned in Chapter 2, atoms are required to be integers a with $-$`BETA` $< a <$ `BETA`. This allows the garbage collector and other functions operating on objects to decide whether a variable of type `Word` contains an atom or a list handle. Note that values less or equal $-$`BETA` are legal only during garbage collection while values greater than `BETA` + `NU` are used for referencing other garbage collected structures.

The `Words` of a cell adressed by a list handle `L` are `SPACE[L − BETA]` and `SPACE[L − BETA + 1]`. To simplify these computations, the C pointers `SPACEB` and `SPACEB1` are set to the memory addresses of SPACE[−BETA] and SPACE[−BETA + 1], respectively. This is used by the functions FIRST(L), which returns SPACEB1[L], and RED(L), which returns SPACEB[L].

C.1.2 Implementation of GCA Handles

When SACLIB is initialised, the array GCASPACE containing NUp + 1 structures of type GCArray is allocated. A *GCA handle* is defined to be BETApplus the index of the corresponding GCArray structure in the GCASPACE array, with the null handle being NIL.

The GCArray structure contains the following fields:

- next ... a Word, used for linking empty GCArrays to the GCAAVAIL list and for marking (see Section C.1.3).
- flag ... a Word, set to one of GC_CHECK and GC_NO_CHECK (see Section C.1.3).
- len ... a Word, the length of the array in Words.
- array ... a C pointer to an array of Words of size len.

When GCAMALLOC()(is called, it takes the first GCArray from the GCAAVAIL list and initializes its fields.

GCA2PTR() simply returns the C pointer in the array field.

GCAFREE() deallocates the memory addressed by the array field, sets all fields to zero, and links the GCArray to the beginning of the GCAAVAIL list.

C.1.3 The Garbage Collector

Garbage collection is invoked when COMP() or GCAMALLOC() call GC() in the case of AVAIL or GCAAVAIL being NIL. The garbage collector consists of two parts:

- The function GC() is system dependent. It must ensure that the contents of all processor registers are pushed onto the stack and pass alignment information and the address of the end of the stack to GCSI().
- GCSI() is the system independent part of the garbage collector. It uses a mark-and-sweep method for identifying unused cells:

  Marking: The processor registers, the system stack, and the variables and GCA arrays to which pointers are stored in the GCGLOBALS list are searched for non-NIL list and GCA handles. All the cells accessible from these handles are marked by a call to MARK().

  If a list handle is found, this function traverses the cells of the list, marking them by negating the contents of their first Word. If the second Word of a cell contains a list or GCA handle, MARK() calls itself recursively on this handle.

  In case of a GCA handle, the GCA cell adressed by the handle is marked by negating the contents of its next field. If the cell's flag field is not set to GC_NO_CHECK, the Words in the array pointed to by the array field are searched for list or GCA handles with MARK() calling itself recursively on valid handles.

  Sweeping: In the sweep step, the AVAIL and GCAAVAIL lists are built:

  Cells in SPACE whose first Word contains a positive value are linked to the AVAIL list. If the first Word of a cell contains a negative value, it is made positive again and the cell is not changed in any other way.

  Cells in GCASPACE whose next field contains a positive value are linked to the GCAAVAIL list and the array pointed to by the array field is deallocated. If the next field contains a negative value, it is made positive again and the cell is not changed in any other way.

If the `AVAIL` list contains no more than `NU/RHO` cells at the end of garbage collection, an error message is written to the output stream and the program is aborted.

C.2 Constants and Global Variables

This section lists SACLIB types, constants, and global variables. All types and constants are defined in "saclib.h", "sactypes.h", and "sacsys.h". External variables are defined in "external.c" and declared as `external` in "saclib.h".

The average user of SACLIB functions should not find it neccessary to deal directly with any of these values (except for `Word`, `BETA`, and `NIL`, which are also mentioned in other sections). If you modify any of the values listed below without knowing what you are doing, you may either crash SACLIB or cause it to produce false results, so please take care!

Notation: In the description below, *pointer* means a C pointer (i.e. an actual memory address) and *pointer to an array* means a C pointer to the first element of an array. *List handle* means a SACLIB list handle (i.e. an integer `L` with `BETA` $\leq$ `L` $<$ `BETAp` which is used as an index into the `SPACEB` and `SPACEB1` arrays), and *GCA handle* means a handle for a garbage collected array (i.e. an integer `A` with `BETAp` $\leq$ `A` $<$ `BETApp` which is used as an index into the `GCASPACEBp` array). *Cell* means a SACLIB list cell (i.e. two consecutive `Word`s in the `SPACE` array, the first one of which has an odd index) and *GCA cell* means a `GCArray` structure in the `GCASPACE` array.

In SACLIB only two low-level data structures are typechecked by the C compiler[1]. These two typedefs are:

- `Word` ... the basic type which in most installations of SACLIB will be the same as a C `int`. `Word` is defined in "sysdep.h".
- `GCArray` ... a `struct` containing information on garbage collected arrays. This is a SACLIB internal data structure defined in "sactypes.h".

The following constants are defined in "sacsys.h" except for `NIL`, which is defined in "saclib.h":

- `BETA` ... a `Word`, the value used to distinguish between atoms and lists. This is also the base for the internal representation of large integers. `BETA` must be a power of 2 such that $2^8 \leq$ `BETA` and $3 *$ `BETA` fits into a `Word`. In most implementations where a `Word` is a standard C `int` with n bits, the setting is `BETA` $= 2^{n-3}$.
- `BETA1` ... a `Word`, `BETA1` $=$ `BETA` $- 1$.
- `NIL` ... a `Word`, the empty list handle[2].
- `NU_, NUp_, NPRIME_, NSMPRM_, NPFDS_, RHO_, NPTR1_` ... `Word`s, the initial values for the corresponding global variables.

The following flags are defined in "saclib.h":

- `GC_CHECK / GC_NO_CHECK` ... `Word`s, used for telling the garbage collector whether an array allocated by `GCAMALLOC()` will contain list or GCA handles (and thus cannot be ignored in the mark phase).
- `SAC_KEEPMEM / SAC_FREEMEM` ... `Word`s, used when calling `ENDSACLIB()` directly for requesting memory allocated by `GCAMALLOC()` to be kept / deallocated.

[1] Lists, integers, polynomials, etc. are structures which are built at runtime. For these no type checking is done so that the programmer has to make sure that there are no conflicts.

[2] This is equal to `BETA`. For historical reasons, in some SACLIB functions `BETA` is explicitly used instead of `NIL`.

Below we give a list of the SACLIB global variables as defined in "external.c":

List processing and garbage collection:

- `AVAIL` ... a `Word`, the list handle of the free list.
- `GCGLOBALS` ... a `Word`, the list handle of the list of global variables.
- `BACSTACK` ... a pointer to the beginning of the system stack.
- `GCC` ... a `Word`, the number of garbage collections.
- `GCAC` ... a `Word`, the number of GCA cells collected in all garbage collections.
- `GCCC` ... a `Word`, the number of cells collected in all garbage collections.
- `GCM` ... a `Word`, if `GCM` is 1, a message is written to the output stream each time the garbage collector is called.
- `NU` ... a `Word`, one less than the size of the `SPACE` array in `Word`s, i.e. twice the number of cells in `SPACE`.
- `RHO` ... a `Word`, the garbage collector aborts the program if no more than `NU/RHO` cells were reclaimed.
- `SPACEB` ... a pointer to an array of words, $\texttt{SPACEB} = \texttt{SPACE} - \texttt{BETA}$.
- `SPACEB1` ... a pointer to an array of words, $\texttt{SPACEB1} = \texttt{SPACE} - \texttt{BETA1}$.
- `GCAAVAIL` ... a `Word`, the GCA handle of the free list of GCA cells.
- `GCASPACE` ... a pointer to an array of `GCArray` structures, the memory space for GCA cells.
- `GCASPACEBp` ... a pointer to an array of `GCArray` structures, $\texttt{GCASPACEBp} = \texttt{GCASPACE} - \texttt{BETAp}$.
- `NUp` ... a `Word`, one less than the number of `GCArray` structures in the `GCASPACE` array.
- `BETAp` ... a `Word`, the bound used to distinguish between list and GCA handles. $\texttt{BETAp} = \texttt{BETA} + \texttt{NU} + 1$.
- `BETApp` ... a `Word`, the upper bound on GCA GCA handles. $\texttt{BETApp} = \texttt{BETAp} + \texttt{NUp} + 1$.

Timing:

- `TAU` ... a `Word`, the time (in milliseconds) spent for garbage collections.
- `TAU0` ... a `Word`, the system time (in milliseconds) just before SACLIB initialization.
- `TAU1` ... a `Word`, the system time (in milliseconds) immediately after SACLIB initialization.

Integer arithmetic:

- `DELTA` ... a `Word`, $\texttt{DELTA} = 2^{\lfloor \texttt{ZETA}/2 \rfloor}$.
- `EPSIL` ... a `Word`, $\texttt{EPSIL} = 2^{\lceil \texttt{ZETA}/2 \rceil} = \texttt{BETA}/\texttt{DELTA}$.
- `ETA` ... a `Word`, $\texttt{ETA} = \lfloor \log_{10} \texttt{BETA} \rfloor$.
- `RINC` ... a `Word`, the increment for the random number generator.
- `RMULT` ... a `Word`, the multiplier for the random number generator.
- `RTERM` ... a `Word`, the last value produced by the random number generator.
- `TABP2` ... a pointer to an array of `Word`s, $\texttt{TABP2}[i] = 2^{i-1}$ for $1 \leq i \leq \texttt{ZETA}$.
- `THETA` ... a `Word`, $\texttt{THETA} = 10^{\texttt{ETA}}$.

- UZ210 ... a Word, the list handle of the list of units of $\mathbf{Z}_{210}$.
- ZETA ... a Word, ZETA $= \log_2$ BETA.

Prime numbers:

- NPFDS ... a Word, the number of primes used by the SACLIB function IUPFDS.
- NPRIME ... a Word controlling the number of primes in PRIME.
- PRIME ... a Word, the list handle of the list of primes between BETA $-$ NPRIME $*$ ZETA $*$ 7/5 and BETA.
- NSMPRM ... a Word, the upper bound on the size of primes in SMPRM.
- SMPRM ... a Word, the list handle of the list of primes $<$ NSMPRM.

Miscellaneous:

- NPTR1 ... a Word, the number of Words in the GCAPTR1 array.
- GCAPTR1 ... a Word, the GCA handle of the array used by the function IUPTR1.

Input/Output:

- LASTCHAR ... a Word, the last character read from the input stream.

Index