APU User Manual

Version 2.0

Ashutosh Rege
rege@cs.berkeley.edu
University of California
Berkeley, CA 94720

Contents

Chapter 1

Introduction

APU stands for *Algebraic Programming Utilities* and that is precisely what it provides - a collection of special-purpose C++ utilities which can be used for implementing algorithms involving polynomial equations and inequalities [1]. APU can be used in two ways : as a stand-alone developmental platform which can be used to develop C++ application code or in conjunction with a computer algebra package like Maple to solve specific problems without necessarily developing application code.

The kinds of problems that APU can be used for are :

- Basic algebraic algorithms. Though APU provides a collection of basic routines like Sylvester resultants, it is by no means a comprehensive collection of routines. The user however can use the elementary operations on polynomials to write application code for specific algebraic algorithms.
- Algorithms for real algebraic geometry. APU provides a collection of routines which can used to solve systems of equations using resultant-based techniques. It provides routines for counting the real roots of a univariate polynomial and for determining the sign sequences of a collection of polynomials at its roots.
- Stand-alone applications. APU can be used to develop C++ stand-alone applications which take advantage of the implementation of straight-line programs in APU. Two examples for this which have been implemented are : implicit point location in semi-algebraic sets and geometric theorem proving.

1.1 Writing Programs in APU

APU has been written (using the operator overloading facility in C++) to produce user code which looks like the routines written for a computer algebra system like Maple. Syntax is therefore simple

[1] The current version does not provide a full set of routines for inequalities

in APU : to add two polynomials one simply writes

$$p3 = p1 + p2;$$

Though knowledge of C++ would be nice, one is not required to know a whole lot about C++ to write a program in APU. In fact, one could use one of the example programs included in with the code as a starting point for writing their own application. If you feel you do not have a good enough knowledge of C++ the best way to get started would be to examine the two example programs provided with the toolkit. The first one counts the number of real intersections of (the algebraic sets defined by) two polynomials in two variables. The common roots are projected on to one axis by computing the Sylvester resultant of the two polynomials. A straightforward Sturm sequence computation gives the number of common real roots in the generic case. The second example program deals with implicit point location in semi-algebraic varieties.

1.2 Getting Started and Requirements

You can get the software for APU by anonymous ftp from **robotics.eecs.berkeley.edu** in the directory pub/rege/toolkit2.0. It can also be obtained on the World Wide Web using a net browser from the URL : **http://www.cs.berkeley.edu/~rege**.

APU requires the following software which is public domain :

- **Gnu C++** (*g++*), **Gnu make** (*gmake*).
- **LEDA** (Library of Efficient Data structures and Algorithms) which is the software APU uses currently for its arithmetic packages. LEDA is copyrighted by the Max-Planck-Institut fuer Informatik, Saarbruecken, Germany and can be obtained (for non-commercial use) by anonymous ftp from **ftp.mpi-sb.mpg.de** in pub/LEDA or on the World Wide Web from the URL : **http://www.mpi-sb.mpg.de/LEDA/leda.html**

For matrix utilities APU uses the **Numerical Recipes** library, copyrighted by the American Institute of Physics and available commercially. Users may replace the code with other matrix code. The two routines used from that library are **ludcmp()** and **lubksb()**. We intend to eliminate this dependence on non-public domain code by replacing these routines in the near future.

1.3 Overview of Data Structures and Operators

APU implements two primary levels of data structures :

Polynomials These represent the high level data structure which the user will typically use. A polynomial is represented in (sparse) recursive form, i.e. as a polynomial in one variable whose coefficients are either polynomials in another variable or field elements. Polynomials come with the standard arithmetic operators as well as other functions like Sylvester resultants, Sturm sequences etc.

Field elements These represent the lower level structures which can used by themselves for arithmetic or, usually, as structures over which polynomials are defined. A variety of different types are defined including integers modulo a prime, "big" integers, and straight-line programs. In addition to the standard operators provided within a sub-class, (binary) operators are also provided for certain pairs of classes (e.g. arithmetic operators are provided for two objects of types "small" integer and big integer).

In addition to the above two primary classes, APU provides subsidiary functions, operators and utilities such as reading and writing polynomials, interacting with polynomials in Maple format etc.

1.4 Important Things to Remember

This section details things that must be borne in mind while writing code using APU.

- Make sure to call the function **init_var()** in the main program before making other function calls. This function (located in the code file *init.c*) initializes the variables names that will used for polynomials and straight-line programs. A variable name that is not recognized as one in these lists will be treated as an error. You can customize the number and the names of the variables by making appropriate changes to the functions in *init.c*.

Chapter 2

Field Elements

All polynomial computations in APU are done over field elements of different types. APU supports different types of fields (some of which strictly speaking do not constitute fields in the mathematical sense) including integers modulo a prime, fixed-precision floating point, straight-line programs etc. In addition to these types, the type Variable (which represents variables such as "x1") is also a defined to be a field type. Additionally, the user can add his own field type if required.

2.1 General Field Elements

2.1.1 Definition

An instance f of the data type *Field_element* is a field element. A field element has a *type* and a *representation*, the latter being hidden from the user. Every type of field element has its own standard field operations such as $+,*,==$ (boolean equality check) etc. Additionally, operations are also allowed between two field elements of different types for certain combinations of field element types.

2.1.2 Types

Every field element is of a particular type which is stored in its data structure as an *int* . E.g. the data structure for *IntModp* has the type represented by the *int* value 3, which in turn is synonymous with INTMODP (by the command "#define INTMODP 3").

Type	Description	#define Synonym
-	No type	NOTYPE
Variable	Variables represented by strings	VAR
SmallInt	Fixed precision 1-word size integers	SMALLINT
IntModp	Integers modulo a prime	SMALLINT
BigInt	Arbitrary precision integers[a]	BIGINT
Float	Fixed-precision floating point	FLOAT
BigFloat	Multi-precision floating point[b]	BIGFLOAT
Mixed	Combination *IntModp* and float approximation	MIXED
Slp	Straight-line programs	SLP

[a] Currently implemented using the LEDA "integer" type
[b] Currently implemented using the LEDA "bigfloat" type

2.1.3 Constructors

Constructor	Usage	Explanation
Field_element ()	*Field_element* f ;	Empty field element.
Field_element (*int n*)	*Field_element* $f = 5$;	Set f to *SmallInt* with value 5
Field_element (*unsigned int n*)	*Field_element* $f =$ (*unsigned int*) 5;	Set f to *IntModp* with value 5
Field_element (*integer n*)	*Field_element* $f =$ *integer* (5);	Set f to *BigInt* with value 5
Field_element (*float f*)	*Field_element* $f = 5.0$;	Set f to *Float* with value 5.0
Field_element (*double d*)	*Field_element* $f =$ (double)(5.0);	Set f to *Float* with value 5.0
Field_element (*bigfloat F*)	*Field_element* $f =$ bigfloat(5.0);	Set f to *BigFloat* with value 5.0

2.1.4 Member Functions

Return Type	Usage	Explanation
int	f.type_of();	Returns the type of the field element
bool	f.is_number();	Returns true if the field element is not of no type or of type Variable
*char**	f.class_name();	Returns the class name of the given field element.
void	f.print();	Prints the field element.

2.1.5 Other Functions

Some of the member functions have been rewritten to make the syntax easier :

Return Type	Usage	Explanation
int	type_of(f);	Returns the type of f
bool	is_number(f);	Returns true if the field element is not of no type or of type Variable

2.1.6 Operators

The field element operators do the obvious thing with the operands.

Arithmetic Operators

Return Type	Usage	Explanation
Field_element	$a+b$	Addition.
Field_element	$a-b$	Subtraction.
Field_element	$a*b$	Multiplicaton.
Field_element	a/b	Division.
Field_element	$-a$	Negation.
Field_element	$a\hat{}b$	Raise a to power b (b is *Field_element* type) [a].
Field_element	$a\hat{}n$	Raise a to power n ($n \geq 0$ is *int* type).

[a] b must be of type SmallInt

Relational Operators

Return Type	Usage	Explanation
bool	if $(a == b)\ldots$	Equals.
bool	if $(a! = b)\ldots$	Not equals.
bool	if $(a < b)\ldots$	Less than.
bool	if $(a <= b)\ldots$	Less than or equal to.
bool	if $(a > b)\ldots$	Greater than.
bool	if $(a >= b)\ldots$	Greater than or equal to.
bool	if is_zero$(a)\ldots$	Equal to 0.

Other Operators

Return Type	Usage	Explanation
ostream&	cout $<< a$	Print a to given output stream.

2.1.7 Header Files

- **Primary header file** *field.h* (Use this file with the #include directive)
- **Other headers file** *field_arithop.h, field_aux.h, field_def.h, field_otherop.h, field_relop.h, field_types.h, field_unaryop.h*

2.1.8 Usage Examples

```
#include "field.h"

int
main(int argc, char** argv)
{
  Field_element fe1 = 5;            // Creates a SmallInt type with value 5
  Field_element fe2 = 10.0          // Creates a Float type with value 10.0
  Field_element result = fe1 + fe2; // Add 5 and 10.0
  cout << result;                   // Prints result to standard output.
  Field_element fe4 = integer(4);   // Creates a BigInt type with value 4
  Field_element result = fe4 / fe1; // Operand with different types -
                                    // allowed for some combinations
  cout << result;                   // Prints result to standard output.
  if (is_zero(result))              // Test for zero
    cout << "Result is 0\n";
}
```

2.2 Fixed Precision Integers

2.2.1 Definition

An instance f of the data type *SmallInt* is a fixed-precision integer of size one word, and is essentially equivalent to the *int* type. The *type* of f is SMALLINT.

SmallInt is a class derived from the base class *Field_element* and as such inherits various functions such as *print()* and type_of().

2.2.2 Global variables

These variables must be set in the user code. Currently they are set in the file *init.c*.

Variable	Usage	Explanation
SmallInt::Class_name	SmallInt::Class_name = "SmallInt";	Set only once.

2.2.3 Constructors

Constructor	Usage	Explanation
SmallInt ()	*SmallInt* a ;	Creates an instance of type *SmallInt* and initializes it with zero
SmallInt (*int* n)	*SmallInt* $a = 5$;	Creates an instance of *SmallInt* and initializes it with value n

2.2.4 Member Functions

Return Type	Usage	Explanation
int	a.i_value();	Returns the integer value of a
void	a.print();	Prints the integer value of a

2.2.5 Other Functions

Some of the member functions have been rewritten to make the syntax easier :

Return Type	Usage	Explanation

2.2.6 Operators

Arithmetic Operators

Return Type	Usage	Explanation
SmallInt	$a + b$	Addition.
SmallInt	$a - b$	Subtraction.
SmallInt	$a * b$	Multiplicaton.
SmallInt	a/b	Division.
SmallInt	$-a$	Negation.
SmallInt	$a\hat{\ }b$	Raise a to power b (b is of type *SmallInt*)
SmallInt	$a\hat{\ }n$	Raise a to power n ($n \geq 0$ is *int* type).

Relational Operators

Return Type	Usage	Explanation
bool	if $(a == b) \ldots$	Equals.
bool	if $(a! = b) \ldots$	Not equals.
bool	if $(a < b) \ldots$	Not equals.
bool	if $(a <= b) \ldots$	Not equals.
bool	if $(a > b) \ldots$	Not equals.
bool	if $(a >= b) \ldots$	Not equals.
bool	if is_zero$(a) \ldots$	Equal to 0.

Other Operators

Return Type	Usage	Explanation
int	sign(a)	Returns, $+1$, -1 or 0 according to sign of a

2.2.7 Header Files

- **Primary header file** *smallint.h* (Use this file with the #include directive)

2.2.8 Usage Examples

```
#include "smallint.h"

SmallInt::Class_name = "SmallInt";  // Set the global name (once-only)
int
main(int argc, char** argv)
{
  SmallInt a = 5;                   // Creates a SmallInt type with value 5
  SmallInt b = 8;                   // Creates a SmallInt type with value 8
  SmallInt c = a+b ;                // Add a + b and store in c.
  cout << c;                        // Prints result to standard output.
  if (is_zero(c))                   // Test for zero
    cout << "Result is 0\n";
}
```

2.3 Arbitrary Precision Integers

2.3.1 Definition

An instance f of the data type *BigInt* is an arbitrary-precision integer, currently implemented in APU using the LEDA *integer* type. The *type* of f is BIGINT.

2.3.2 Global variables

These variables must be set in the user code. Currently they are set in the file *init.c*.

Variable	Usage	Explanation
BigInt::Class_name	BigInt::Class_name = "BigInt";	Set only once.

2.3.3 Constructors

Constructor	Usage	Explanation
BigInt ()	*BigInt* a ;	Creates an instance of type *BigInt* and initializes it with zero
BigInt (*int* n)	*BigInt* $a = 5$;	Creates an instance of *BigInt* and initializes it with value n
BigInt (*integer* n)	*BigInt* $a =$ integer(5);	Creates an instance of *BigInt* and initializes it with value n

2.3.4 Member Functions

Return Type	Usage	Explanation
int	a.i_value();	Returns the integer value of a
void	a.print();	Prints the integer value of a

2.3.5 Other Functions

Some of the member functions have been rewritten to make the syntax easier :

Return Type	Usage	Explanation

2.3.6 Operators

Arithmetic Operators

Return Type	Usage	Explanation
BigInt	$a + b$	Addition.
BigInt	$a - b$	Subtraction.
BigInt	$a * b$	Multiplicaton.
BigInt	a/b	Division.
BigInt	$-a$	Negation.
BigInt	$a\hat{\ }n$	Raise a to power n ($n \geq 0$ is *int* type).

Relational Operators

Return Type	Usage	Explanation
bool	if $(a == b) \ldots$	Equals.
bool	if $(a! = b) \ldots$	Not equals.
bool	if $(a < b) \ldots$	Not equals.
bool	if $(a <= b) \ldots$	Not equals.
bool	if $(a > b) \ldots$	Not equals.
bool	if $(a >= b) \ldots$	Not equals.
bool	if is_zero$(a) \ldots$	Equal to 0.

Other Operators

Return Type	Usage	Explanation
int	sign(a)	Returns, $+1$, -1 or 0 according to sign of a

2.3.7 Header Files

- **Primary header file** *bigint.h* (Use this file with the #include directive)

2.3.8 Usage Examples

```
#include "bigint.h"

BigInt::Class_name = "BigInt";      // Set the global name (once-only)
int
main(int argc, char** argv)
{
  BigInt a = 5;                     // Creates a BigInt type with value 5
  BigInt b = 8;                     // Creates a BigInt type with value 8
  BigInt c = a+b ;                  // Add a + b and store in c.
  cout << c;                        // Prints result to standard output.
  if (is_zero(c))                   // Test for zero
    cout << "Result is 0\n";
}
```

2.4 Integers Modulo a Prime

2.4.1 Definition

An instance f of the data type *IntModp* is an integer mod a (global) prime. The *type* of f is INTMODP.

IntModp is a class derived from the base class *Field_element* and as such inherits various functions such as *print()* and type_of().

2.4.2 Global variables

These variables must be set in the user code. Currently they are set in the file *init.c*.

Variable	Usage	Explanation
IntModp::Class_name	IntModp::Class_name = "IntModp";	Set only once.
IntModp::prime	IntModp::prime = 44497;	Set the *common* prime for *all* instances of *IntModp* to given number.

2.4.3 Constructors

Constructor	Usage	Explanation
IntModp ()	*IntModp* a ;	Creates an instance of type *IntModp* and initializes it with zero
IntModp (*int* n)	*IntModp* $a = 5$;	Creates an instance of *IntModp* and initializes it with value n mod prime

2.4.4 Member Functions

Return Type	Usage	Explanation
int	a.i_value();	Returns the integer (mod prime) value of a
int	a.prime_of();	Returns the global prime of all *IntModp* elements

2.4.5 Other Functions

Some of the member functions have been rewritten to make the syntax easier :

Return Type	Usage	Explanation

2.4.6 Operators

Arithmetic Operators

Return Type	Usage	Explanation
IntModp	$a + b$	Addition.
IntModp	$a - b$	Subtraction.
IntModp	$a * b$	Multiplicaton.
IntModp	a/b	Division.
IntModp	$-a$	Negation.
IntModp	$a\hat{\ }n$	Raise a to power n ($n \geq 0$ is *int* type).

Relational Operators

Return Type	Usage	Explanation
bool	if $(a == b)\ldots$	Equals.
bool	if $(a! = b)\ldots$	Not equals.
bool	if is_zero$(a)\ldots$	Equal to 0.

Other Operators

Return Type	Usage	Explanation

2.4.7 Header Files

- **Primary header file** *intmodp.h* (Use this file with the #include directive)

2.4.8 Usage Examples

```
#include "intmodp.h"

IntModp::prime = 13;                 // Set the global prime to 13
int
main(int argc, char** argv)
{
  IntModp a = 5;                      // Creates a IntModp type with value 5
  IntModp b = 8;                      // Creates a IntModp type with value 8
  IntModp c = a+b ;                   // Add a + b % 13 and store in c.
  cout << c;                          // Prints result to standard output.
  if (is_zero(c))                     // Test for zero
    cout << "Result is 0\n";
}
```

2.5 Fixed Precision Floating Point

2.5.1 Definition

An instance f of the data type *Float* is a fixed-precision floating point of size corresponding to *double* on the machine on which the code is compiled, and is essentially equivalent to the *double* type. The *type* of f is FLOAT.

Float is a class derived from the base class *Field_element* and as such inherits various functions such as *print()* and type_of().

2.5.2 Global variables

These variables must be set in the user code. Currently they are set in the file *init.c*.

Variable	Usage	Explanation
Float::Class_name	Float::Class_name = "Float";	Set only once.

2.5.3 Constructors

Constructor	Usage	Explanation
Float ()	*Float* a ;	Creates an instance of type *Float* and initializes it with 0.0
Float (*int* n)	*Float* $a = 5$;	Creates an instance of *Float* and initializes it with value n.0
Float (*float* n)	*Float* $a = 5.0$;	Creates an instance of *Float* and initializes it with value n

2.5.4 Member Functions

Return Type	Usage	Explanation
double	a.f_value();	Returns the floating-point value of a
void	a.print();	Prints the floating-point value of a

2.5.5 Other Functions

Some of the member functions have been rewritten to make the syntax easier :

Return Type	Usage	Explanation

2.5.6 Operators

Arithmetic Operators

Return Type	Usage	Explanation
Float	$a + b$	Addition.
Float	$a - b$	Subtraction.
Float	$a * b$	Multiplicaton.
Float	a/b	Division.
Float	$-a$	Negation.
Float	$a\hat{}b$	Raise a to power b (b is of type *Float*)
Float	$a\hat{}n$	Raise a to power n ($n \geq 0$ is *int* type).

Relational Operators

Return Type	Usage	Explanation
bool	if $(a == b) \ldots$	Equals.
bool	if $(a! = b) \ldots$	Not equals.
bool	if $(a < b) \ldots$	Not equals.
bool	if $(a <= b) \ldots$	Not equals.
bool	if $(a > b) \ldots$	Not equals.
bool	if $(a >= b) \ldots$	Not equals.
bool	if is_zero$(a) \ldots$	Equal to 0.

Other Operators

Return Type	Usage	Explanation
int	sign(a)	Returns, $+1$, -1 or 0 according to sign of a

2.5.7 Header Files

- **Primary header file** *float.h* (Use this file with the #include directive)

2.5.8 Usage Examples

```
#include "float.h"

Float::Class_name = "Float";        // Set the global name (once-only)
int
main(int argc, char** argv)
{
  Float a = 5;                      // Creates a Float type with value 5
  Float b = 8;                      // Creates a Float type with value 8
  Float c = a+b ;                   // Add a + b and store in c.
  cout << c;                        // Prints result to standard output.
  if (is_zero(c))                   // Test for zero
    cout << "Result is 0\n";
}
```

2.6 Multi-Precision Floating Point

2.6.1 Definition

An instance f of the data type *BigFloat* is a multi-precision floating point and is implemented using the LEDA type *bigfloat* . The *type* of f is BIGFLOAT.

BigFloat is a class derived from the base class *Field_element* and as such inherits various functions such as *print()* and type_of().

2.6.2 Global variables

These variables must be set in the user code. Currently they are set in the file *init.c*.

Variable	Usage	Explanation
BigFloat::Class_name	BigFloat::Class_name = "BigFloat";	Set only once.

2.6.3 Constructors

Constructor	Usage	Explanation
BigFloat ()	*BigFloat* a ;	Creates a *BigFloat* and initializes it with 0.0
BigFloat (*int* n)	*BigFloat* $a = 5$;	Creates a *BigFloat* and initializes it with value $n.0$
BigFloat (*float* n)	*BigFloat* $a = 5.0$;	Creates a *BigFloat* and initializes it with value n
BigFloat (*integer* n)	*BigFloat* $a =$ integer(5);	Creates a *BigFloat* and initializes it with value $n.0$
BigFloat (*bigfloat* n)	*BigFloat* $a =$ bigfloat(5);	Creates a *BigFloat* and initializes it with value n

2.6.4 Member Functions

Return Type	Usage	Explanation

2.6.5 Other Functions

Some of the member functions have been rewritten to make the syntax easier :

Return Type	Usage	Explanation

2.6.6 Operators

Arithmetic Operators

Return Type	Usage	Explanation
BigFloat	$a + b$	Addition.
BigFloat	$a - b$	Subtraction.
BigFloat	$a * b$	Multiplicaton.
BigFloat	a/b	Division.
BigFloat	$-a$	Negation.
BigFloat	$a\hat{\ }b$	Raise a to power b (b is of type *BigFloat*)
BigFloat	$a\hat{\ }n$	Raise a to power n ($n \geq 0$ is *int* type).

Relational Operators

Return Type	Usage	Explanation
bool	if $(a == b) \ldots$	Equals.
bool	if $(a! = b) \ldots$	Not equals.
bool	if $(a < b) \ldots$	Not equals.
bool	if $(a <= b) \ldots$	Not equals.
bool	if $(a > b) \ldots$	Not equals.
bool	if $(a >= b) \ldots$	Not equals.
bool	if is_zero$(a) \ldots$	Equal to 0.

Other Operators

Return Type	Usage	Explanation
int	sign(a)	Returns, $+1$, -1 or 0 according to sign of a

2.6.7 Header Files

- **Primary header file** *bigfloatpt.h* (Use this file with the #include directive)

2.6.8 Usage Examples

```
#include "bigfloatpt.h"

BigFloat::Class_name = "BigFloat";  // Set the global name (once-only)
int
main(int argc, char** argv)
{
  BigFloat a = 5.1;                 // Creates a BigFloat type with value 5
  BigFloat b = 8;                   // Creates a BigFloat type with value 8
  BigFloat c = a+b ;                // Add a + b and store in c.
  cout << c;                        // Prints result to standard output.
  if (is_zero(c))                   // Test for zero
    cout << "Result is 0\n";
  a = bigfloat(5);                  // Use LEDA type bigfloat
  b = integer(4);                   // Use LEDA type integer
  c = a/b;
}
```

Chapter 3

Variables

Variables are defined to be a type of field element, though they do not (though could be extended to) support operations like addition. Variables represent strings such as "x1", "y" etc. over which polynomials and straight-line programs are defined. Variables are assumed to be *ordered*, i.e. each variable has a rank. For example, the user might define the variable set to be "x1", "x2" and "x3" which are ranked 1, 2 and 3 respectively (i.e. $x1 > x2 > x3$).
Note : The behaviour of variables is typically hidden from the user and unless one is interested in direct manipulations involving them it is not necessary to understand how they work in APU.

3.1 Definition

An instance v of the data type *Variable* has a *type* which is given by VAR, and a representation which has the following information : index or rank, a string which is the "name" of the variable (such as "x1"), a field element which stores a (random) value (such as an integer value or integer mod a prime) and a pointer to its straight-line program representation.

3.2 Constructors

Constructor	Usage	Explanation
Variable ()	*Variable* v;	Empty variable.
Variable (*char*$*$)	*Variable* v= "x1"	Variable "x1"

3.3 Member Functions

Return Type	Usage	Explanation
int	f.index();	Returns the index of the variable.
void	f.index(*int*);	Returns the index of the variable.
char∗	f.name();	Returns the string “name” of the variable.
void	f.name(*char∗*);	Sets the “name” of the variable.
Slp	f.slp(Slp);	Sets the Slp of the variable.
Slp	f.slp_of();	Returns the Slp of the variable.
Slp ∗	f.slpPtr_of();	Returns the pointer to Slp of the variable.
Field_element	f.value();	Returns the value of the variable.
void	f.value(*Field_element*);	Sets the value of the variable.

3.4 Other Functions

Some of the member functions have been rewritten to make the syntax easier :

Return Type	Function	Usage	Explanation
int	index(*Variable*)	index(f);	Returns the index of f
char∗	name(*Variable*)	index(f);	Returns the name of f
Slp ∗	slpPtr_of(*Variable*)	slpPtr_of(f);	Returns the pointer to Slp of f.

3.5 Operators

Return Type	Usage	Explanation
ostream&	cout << a	Print a to given output stream.

3.6 Header Files

- **Primary header file** *variable.h* (Use this file with the #include directive)

3.7 Usage Examples

```
#include "variable.h"

int
main(int argc, char** argv)
{
  Variable v1 = "x1";              // Creates a Variable with name "x1"
  v1.index(1);                     // Set its index to 1
  v1.slp(Slp(&v1));                // Makes its Slp be a terminal Slp with
                                   // itself as its leaf node.
  v1.value = 11;                 // Set its value.
}
```

Chapter 4

Straight-line Programs

A straight-line program (SLP) is a representation of a computation involving the operations : $+$, $-$, $*$, $-$ and ^. An SLP is a directed acyclic graph in which the interior nodes represent these arithmetic computations and the leaf nodes represent variables or numbers over which the computations are carried out. Each internal node has out-degree 2, i.e. points to two children which represent the operands of the operator represented by the node.
Evaluation of SLP's : A basic operation for an SLP is evaluating it at (random) values of the input variables. This operation is used, for example, when interpolating an SLP. To do this efficiently, an SLP node has to store the value during a given evaluation. To determine if the value stored in the SLP node is current, we need a flag (an *int* value) which will indicate when the last evaluation was done; this flag will be compared with a global variable *Slp::Eval_level* and if the flag is found to be not equal to it, the node will be re-evaluated and its flag updated to the value of *Slp::Eval_level*. Otherwise, the value of the node will be current.

4.1 Definition

An instance f of the data type *Slp* is an SLP with *type* SLP and a representation which contains the following information :

- Operator. This is one of NOP_FE,NOP_VAR,ADD,SUB, MULT, DIV, EXP, DIFF. The NOP_FE operator implies that the Slp has no operator, i.e. is a leaf in the d.a.g., representing a number. The NOP_VAR operator also represents a leaf node, but containing a variable. The EXP operator corresponds to the operation ^, that is, the operation a^n (n of type *int*) is represented by an EXP node with left child the Slp node corresponding to a and right child a node corresponding to the integer value n. The DIFF operator will be explained below.
- Pointers to the Slp's for the left and right child (i.e. the operands).
- Pointer to a *Field_element* which stores the current value of the node after an evaluation.

- An *int* value which represents the the last time the node was evaluated. This is compared to the global value *Slp::Eval_level* to check if the value is current.
- An array of derivatives of the Slp node w.r.t. the variables over which the Slp is defined.
- An array of *int* flags which indicate which derivatives have been computed.
- The anti-derivative of the current Slp (required for avoiding exponential growth during the differentiation process)
- An *int* representing the degree of the Slp in a given variable.
- An *int* value representing the last time the degree was evaluated. (This works similar to the evaluation process described above, except we are interested in computing the degree of the Slp in a given variable.)

4.2 Constructors

Constructor	Usage	Explanation
Slp ()	*Slp* a ;	Empty *Slp* .
Slp (*int n*)	*Slp* $a = 5$;	Set a to terminal Slp with value *SmallInt* (5)
Slp (*unsigned int n*)	*Slp* $a =$ (*unsigned int*) 5;	Set a to terminal Slp with value *IntModp* (5)
Slp (*float f*)	*Slp* $a = 5.0$;	Set a to terminal Slp with value *Float* (5.0)
Slp (*double d*)	*Slp* $a =$ (double)(5.0);	Set a to terminal Slp with value *Float* (5.0)
Slp (*Field_element f*)	*Slp* $a =$ *Slp* (f) ;	Set a to terminal Slp with value f
Slp (*Variable v*)	*Slp* $a =$ *Slp* (v) ;	Set a to terminal Slp representing the variable v.
Slp (*Variable* $*$ *vp*)	*Slp* $a =$ *Slp* (vp) ;	Set a to terminal Slp representing the variable pointed to by vp.
Slp (*char*$*$ *s*)	*Slp* $a =$ *Slp* ("x1") ;	Set a to terminal Slp representing the variable "x1"

4.3 Member Functions

In the table below f is the Slp for which the member functions are called.

Return Type	Usage	Explanation
int	f.op()	Returns the operator.
Slp	f.lchild()	Returns the left child.
Slp	f.rchild()	Returns the right child.
Field_element	f.value()	Returns the current evaluated value.
void	f.set_value(*Field_element*)	Set the current evaluated value.
Slp	f.anti_deriv_of()	Returns the anti-derivative.
Array<Slp >	f.deriv_array()	Returns the array of derivatives of f w.r.t. the *Slp* variables.
Array<int >	f.deriv_computed_array()	Returns the array of flags which indicate which derivatives have been computed.
int	f.eval_level();	Returns the current evaluation level.
void	f.eval_level(*int* n);	Sets the current evaluation level to n.
int	f.degree()	Returns the currently evaluated degree.
void	f.degree(*int* n)	Sets the currently evaluated degree to n.
int	f.degree_eval_level();	Returns the current degree evaluation level.
void	f.degree_eval_level(*int* n);	Sets the current degree evaluation level to n.
bool	f.is_numerical()	Returns true if f is terminal and represents a number.
bool	f.is_variable()	Returns true if f is terminal and represents a variable.
Variable	f.variable_of()	Returns the variable representation of the Slp.

4.4 Other Functions

Some of the member functions have been rewritten to make the syntax easier :

Return Type	Usage	Explanation
int	op(f)	Returns the operator.
Slp	lchild(f)	Returns the left child.
Slp	rchild(f)	Returns the right child.
Field_element	value(f)	Returns the current evaluated value.
Array<Slp >	deriv_array(f)	Returns the array of derivatives of f w.r.t. the *Slp* variables.
Array<int >	deriv_computed_array(f)	Returns the array of flags which indicate
Slp	anti_deriv_of(f)	Returns the anti-derivative.
		which derivatives have been computed.
int	eval_level(f);	Returns the current evaluation level.
bool	is_numerical(f)	Returns true if f is terminal and represents a number.
bool	is_variable(f)	Returns true if f is terminal and represents a variable.
Variable	variable_of(f)	Returns the variable representation of the Slp.

4.5 Operators

The field element operators do the obvious thing with the operands.

4.5.1 Arithmetic Operators

Return Type	Usage	Explanation
Slp	$a+b$	Addition.
Slp	$a-b$	Subtraction.
Slp	$a*b$	Multiplicaton.
Slp	a/b	Division.
Slp	$-a$	Negation.
Slp	$a\hat{\ }n$	Raise a to power n ($n \geq 0$ is *int* type).

4.5.2 Relational Operators

Return Type	Usage	Explanation
bool	if is_zero(a)...	Equal to 0.

4.5.3 Utilities

Function Name diff()

Syntax *Slp* **diff**(*Slp* slp, *char** **var** = "",*bool* **symbolic** = false)

Description Computes the partial derivative of the given slp w.r.t. the given variable **var**. If the variable **symbolic** is set to true the derivative computed will be 'symbolic' in the sense that the derivatives of leaf nodes which represent variables are not computed - rather a special node called "DIFF" is added in front of them. This allows computation of the derivative just once, the particular derivative can then be obtained by specifying which variable w.r.t. which the derivative has been taken.

Function Name evaluate()

Syntax *Field_element* **evaluate**(*Slp* **slp**, *char** **var** = "", *int* * **non_sing** = 0, *char** **var2** = "", *Field_element* **fe** = 0)

Description Returns an evaluation of the *Slp* . The default call **evaluate(slp)** will simply evaluate the *Slp* at the random values assigned to its variable leaf nodes. Other types of evaluation can be performed depending on the arguments to **evaluate()** as follows :

1. The optional argument **var** is to be set to the variable wrt to which the DIFF operator nodes are to be applied, if the Slp contains such nodes. That is, if one wants to evaluate the *Slp* representing the *symbolic derivative* of some *Slp* one specifies the actual variable

w.r.t. which the derivative is to be taken. In other words, this corresponds to computing

$$\frac{df}{dx|_{x=a}}$$

. Default is no variable.

2. The optional argument **non_sing** is to be set to the address of an integer variable if you want the evaluation procedure to determine if the *Slp* evaluated to a singular value such as 0/0.
3. The optional argument **var2** is set to the variable which has to be substituted for with the next argument **fe**. That is when the variable represented by **var2** is encountered in the *Slp* , its value for evaluation will be **fe** rather than the random value which has been assigned to it. This is useful when you want to evaluate the *Slp* at different values of a variable such as when you want to interpolate the *Slp* into a polynomial in that variable.
4. The optional argument **fe** is used in conjunction with **var2**, as explained above. Set to the desired field element.

4.6 Header Files

- **Primary header file** *slp.h* (Use this file with the #include directive)

4.7 Usage Examples

```
  #include "slp.h"

  int
  main(int argc, char** argv)
  {
    Slp slp1 = 5;                        // Creates a leaf Slp with value 5
    Slp slp2 = "x1";                     // Creates a leaf Slp for "x1".
    Slp slp3 = slp1 + slp2               // Adds the two to get Slp for 5 + x1
    Field_element fe = evaluate(slp3);  // Evaluates the slp for the random
// value that x1 has been set to.
  }
```

Chapter 5

Polynomials

Polynomials are represented in APU using a sparse recursive data structure as shown in Figure 5.1. A polynomial has a field element and a term list. A polynomial can either be a *terminal* polynomial (that is, one representing a numerical coefficient) in which case the field element will contain the numerical value (of the desired type) and its term list will be empty. Or it can be a non-terminal polynomial in which case the field element will be of type *Variable* and the term list will point to a collection of term nodes, each of which will have a degree and a coefficient polynomial.

The *variables* over which the polynomial is defined are stored in a linked list and it is assumed that in the sparse recursive formulation the variables will appear in the order the are stored in the linked list. (E.g. in Figure 5.1 the ordering of the variables is assumed to be "x_1'' followed by "x_2''.)

5.1 Definition

An instance p of the data type *Polynomial* is a polynomial with two members : a *Field_element* entry and a *TermList* entry used as explained above. For basic algorithms dealing with polynomials, the operators and functions below should suffice. To perform more complex operations (e.g., extract the coefficient of the term of a polynomial of the second highest degree) the user will have to work with the *TermList* and *TermNode* data structures which are explained in Chapter 6. Also see Section 5.8 for utilities for polynomials such as pseudo-remainder and resultant.

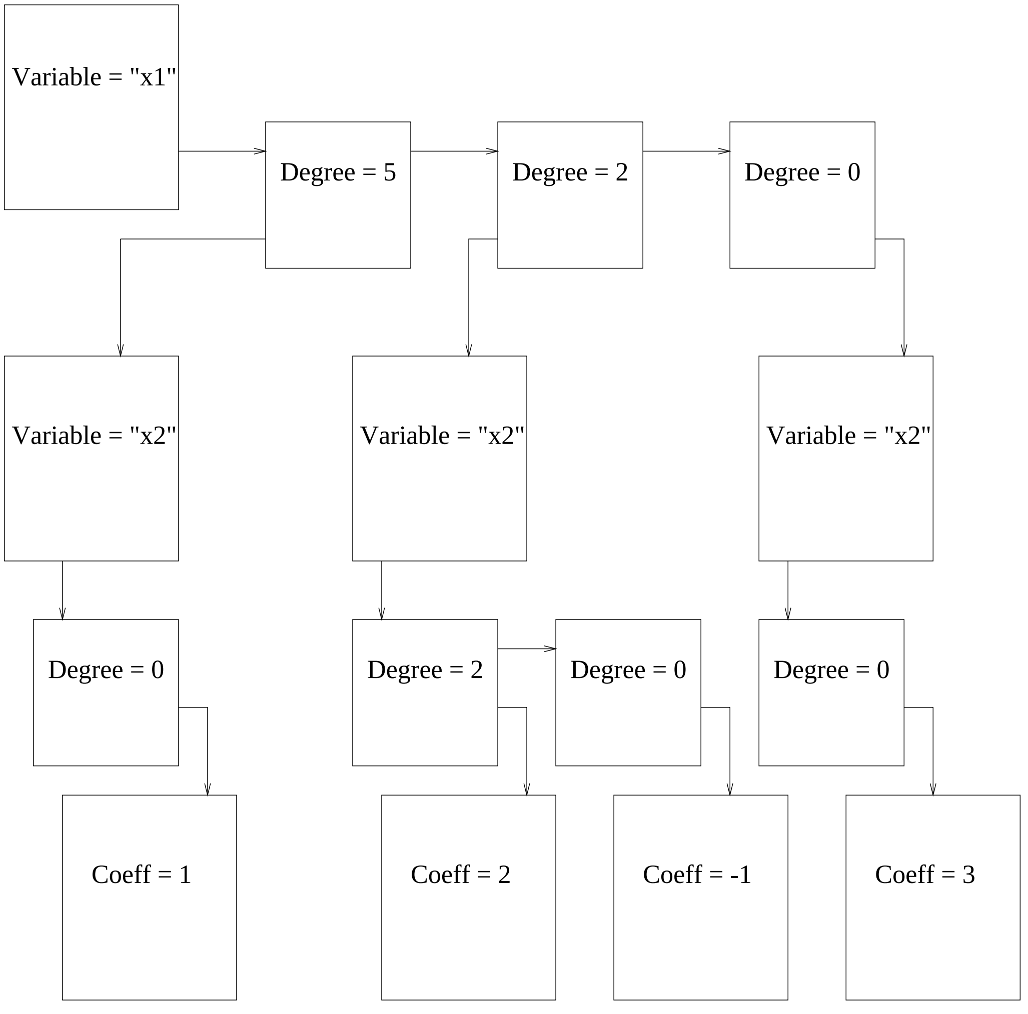

Figure 5.1: Polynomial representation for ${x_1}^5(x_2) + {x_1}^2(2{x_2}^2 - 1) + 1$

5.2 Constructors

Constructor	Usage	Explanation
Polynomial ()	*Polynomial* p ;	Empty *Polynomial* .
Polynomial (*int* n)	*Polynomial* $p = 5$;	Set p to constant poly with value *SmallInt* (5)
Polynomial (*unsigned int* n)	*Polynomial* $p =$ (*unsigned int*) 5;	Set p to constant poly with value *IntModp* (5)
Polynomial (*float* f)	*Polynomial* $p = 5.0$;	Set p to constant poly with value *Float* (5.0)
Polynomial (*double* d)	*Polynomial* $p =$ (double)(5.0);	Set p to constant poly with value *Float* (5.0)
Polynomial (*Field_element* f)	*Polynomial* $p =$ *Polynomial* (f) ;	Set p to constant poly with value f
Polynomial (*Variable* v)	*Polynomial* $p =$ *Polynomial* (v) ;	Set p to poly representing the variable v.
Polynomial (*Variable* $*$ vp)	*Polynomial* $p =$ *Polynomial* (vp) ;	Set p to poly representing the variable pointed to by vp.
Polynomial (*char*$*$ s)	*Polynomial* $p =$ *Polynomial* ("x1") ;	Set p to poly representing the variable $x1$
Polynomial (*char*$*$ s, *int* n)	*Polynomial* $p =$ *Polynomial* ("x1",4) ;	Set p to poly representing $x1^4$
Polynomial (*char*$*$ s, *TermList* t)	*Polynomial* $p =$ *Polynomial* ("x1",tl) ;	Set p to poly in $x1$ with termlist tl

5.3 Member Functions

In the table below f is the Poly for which the member functions are called.

Return Type	Usage	Explanation
Field_element	p.fe_of()	Returns the field element member.
Field_element *	p.fePtr_of()	Returns the field element member.
TermList	p.termlist_of()	Returns the term list member.
int	p.degree()	Returns the degree.
Polynomial	p.lcoeff()	Returns the leading coeff of the poly.
bool	p.is_terminal()	Returns true if the poly has no term list.
bool	p.is_zeropoly()	Returns true if the poly is identically zero.
Variable *	p.variable_of()	Returns a pointer to the variable.

5.4 Other Functions

Some of the member functions have been rewritten to make the syntax easier :

Return Type	Usage	Explanation
Field_element	fe_of(p)	Returns the field element member.
TermList	termlist_of(p)	Returns the term list member.
int	degree(p)	Returns the degree.
Polynomial	lcoeff(p)	Returns the leading coeff of the poly.
bool	is_terminal(p)	Returns true if the poly has no term list.
bool	is_zeropoly(p)	Returns true if the poly is identically zero.
Variable *	variable_of(p)	Returns a pointer to the variable.

5.5 Operators

The polynomial operators do the obvious thing with the operands.

5.5.1 Arithmetic Operators

The operands here can be combinations of *Polynomial* with *Field_element* and *int* .

Return Type	Usage	Explanation
Polynomial	$a + b$	Addition.
Polynomial	$a - b$	Subtraction.
Polynomial	$a * b$	Multiplicaton.
Polynomial	a/b	Division (returns quotient on division).
Polynomial	$-a$	Negation.
Polynomial	$a \% b$	Remainder (returns remainder on division).
Polynomial	$a\hat{\ }n$	Raise a to power n ($n \geq 0$ is *int* type).

5.5.2 Relational Operators

None.

5.5.3 Other Operators and Functions

Return Type	Usage	Explanation
istream&	inFile >> poly	Read in poly in text format from input text file with default arguments (see Section 5.8.5 for details).
ostream&	out_stream << poly	Print poly to the given output stream.

5.6 Header Files

- **Primary header file** *polynomial.h* (Use this file with the #include directive)

5.7 Usage Examples

```
#include "polynomial.h"

int
main(int argc, char** argv)
{
  Polynomial p1 = 5;                  // Creates a poly for 5
  Polynomial p2 = "x1";               // Creates a poly for "x1".
  Polynomial p3 = p1 + p2             // Adds the two to get poly x1 + 5
}
```

5.8 Polynomial Utilities

In addition to the standard polynomial operators, APU currently provides the following set of utilities.

5.8.1 Differentiation, Substitution and Interpolation

Syntax *Polynomial* **diff**(*Polynomial* **p**,*char** **s**,*int* **deriv_type** = NOTYPE).

Description Returns the derivative of the polynomial. The type of derivative returned depends on the variable **deriv_type**.

1. **deriv_type** = SLP, (e.g. diff(p,“l1”,SLP)), **p** is assumed to have *Slp* ’s as its coefficients and the derivative of the *coefficient Slp* ’s will be taken w.r.t. the variable **s**.
2. **deriv_type** = VAR, (e.g. diff(p,“x2”)), the derivative of **p** will be taken w.r.t. the variable **s**.
3. **deriv_type** is some other other field type, or is not specified (e.g. diff(p)), the derivative of **p** will be taken w.r.t. the leading variable of **p**.

Function Name subs()

Syntax *void* **subs** (*Polynomial* poly, *int* n, *Array*<*Polynomial* > numer_array, *Array*<*Polynomial* > denom_array, *Polynomial* *result_n, *Polynomial* *result_d)

Description Substitute variables in the given polynomial "poly" with the rational functions given by xnumer_array and xdenom_array. The ith element of each array gives the numerator and denominator respectively to be substituted for the ith variable, n is the bound on the number of variables.

Syntax *void* **subs** (*Polynomial* poly, *int* n, *Array*<Polynomial> numer_array, *Polynomial* denom, *Polynomial* *result_n, *Polynomial* *result_d)

Description Substitute variables in the given polynomial "poly" with the rational functions given by xnumer_array and common denominator given by denom.

Function Name vanderpolate()

Syntax *Polynomial* **vanderpolate**(*Array*<Polynomial> vals_array, *Array*<Field_element> eval_points, *int* **n**, *char** **var**, *int* **poly_type** = INTMODP)

Description Given an array of values, vals_array, of a polynomial (typically specified by an *Slp*) evaluated at an array of points (eval_points), **vanderpolate()** returns the polynomial after performing *univariate* Vandermonde interpolation. The degree of the polynomial is given by **n**, **var** specifies the variable w.r.t. which the interpolation is done and **poly_type** is the type of *Field_element* over which the polynomial is to be interpolated (default is INTMODP). Note again that even though the *Slp* representing the polynomial could be multivariate, the **vanderpolate()** will recover it only as a univariate polynomial. Typically used in conjunction with the interpolate routine for *Slp* 's

5.8.2 Pseudo-remainder

Syntax *Polynomial* **prem**(*Polynomial* **dividend**,*Polynomial* **divisor**)

Description Returns the pseudo-remainder on pseudo-division of **dividend** by **divisor**. The polynomials are assumed to be listed *ordered* according to the standard variable ordering.

5.8.3 Remainder Sequences and Resultants

1. **Syntax** *List*<*Polynomial* > **sturm_habicht**(*Polynomial* **p1**,*Polynomial* **p2**)

   Description Returns the Sturm-Habicht sequence of **p1** and **p2** as a linked list of *Polynomial* 's.

2. **Syntax** *List*<*Polynomial* > **subres_prs**(*Polynomial* **p1**,*Polynomial* **p2**)

   Description Returns the subresultant polynomial remainder sequence of **p1** and **p2** based on Collins algorithm.

3. **Syntax** *Polynomial* **sylvester_res**(*Polynomial* **p1**,*Polynomial* **p2**)

   Description Returns the Sylvester resultant of **p1** and **p2**. Uses **subres_prs** above.

5.8.4 Sign Determination

These routines various operations related to sign determination for polynomials.

1. **Syntax** *Matrix<int >* **det_sign**(*Polynomial* **p**, *List<Polynomial >* **qlist**)

   Description Returns a *Matrix*of *int* 's encoding the signs of the polynomials in the the polynomial list **qlist** at the roots of the polynomial **p**.

2. **Syntax** *int* **diff_in_alternations**(*List<Polynomial >* **qlist**)

   Description Returns the diference in sign alternations of the polynomials evaluated at $-\infty$ and $+\infty$. The polynomials should have *Field_element* 's as coefficients, not other polynomials.

5.8.5 Polynomial Input and Output

The **text format** for a polynomial in a (text) input or output file should be one of :

$$(var_1\ n_1\ (\cdots)\ n_2\ (\cdots)\ \ldots\ n_k\ (\cdots))$$

or

$$(k)$$

where var_1 is the string representing the first or leading variable of the polynomial; $n_1, \ldots, n_k$ are the degrees of the coefficients in var_1 ordered such that $n_1 > n_2 > \ldots > n_k$. The coefficients corresponding to each degree (which is a polynomial itself) are placed to the right of that degree; (k) represents a polynomial which is *terminal*, i.e. has its entry a number, which can be either an integer or a floating point number. For example the polynomial ${x_1}^5(x_2) + {x_1}^2(2{x_2}^2 - 1) + 1$ the text representation would be :

$$(x1\ 5\ (x2\ 1\ (1))\ 2\ (x2\ 2\ (2)\ 0\ (-1))\ 0\ (1))$$

The following routines allow reading in polynomials in various forms, as well as printing them in various forms.

1. **Syntax** *Polynomial* **read_poly**(*ifstream&* **inFile**, *int* **input_type** = SMALLINT, *int* **poly_type** = INTMODP, *int* **slp_type** = INTMODP)

   Description Reads a polynomial in sparse recursive format from the given input text file. The format of the polynomial should be the text format described above. The function has three optional arguments for controlling the input operation :

   (a) **input_type**. Specifies the type of the numerical coefficients of the input polynomial in the text file - can be one of SMALLINT (default) and FLOAT.

   (b) **poly_type**. Specifies the type of the numerical coefficients of the *Polynomial* type that is created after reading in the text polynomial - can be any of the *Field_element* sub-types. Default is INTMODP.

(c) **slp_type**. Specifies the type of the numerical coefficients of the *Slp* type that is formed (if any coefficients are to be read in as straight-line programs - read below for when this happens).

read_poly works recursively by processing the text file and based on the three type arguments reads in the text polynomial (ignoring white space) and creates a *Polynomial* object of the appropriate type with the appropriate coefficients. If the leading variable of the current polynomial is an *Slp* variable (i.e. a variable declared to be a *Slp* variable by insertion in the *Slp* variable list), it will read the rest of the current (coefficient) polynomial as an *Slp* . Otherwise it will read it as a polynomial. Example : Let $x1, x2$ be *Polynomial* variables and $s1$ an *Slp* variable. Then

$$(x1\ 2\ (s1\ 2\ (4)\ 1\ (1))\ 1\ (x2\ 1\ (1))\ 0\ (x2\ 1\ (s1\ 1\ (4))))$$

will be read as a polynomial in $x1$ with its leading coefficient (of degree 2) an *Slp* , its next coefficient (degree 1) a polynomial in $x2$ and its last coefficient (degree 0) a polynomial in $x2$ which in turn has its leading coefficient an *Slp* .

5.9 C++ Routines for Maple interface

APU can be used to interface with Maple by writing APU polynomials on to files in Maple format using :

Function name print_maple_poly()

Syntax *void* **print_maple_poly**(*ostream&*outFile, *Polynomial* poly);

Description This function will print the given polynomial to the specified output stream in Maple format. Thus one can dump the results of an APU computation onto a file which can then be read into Maple.

Chapter 6

TermLists and TermNodes

As explained in Chapter 5, a *Polynomial* has a *Field_element* entry and a *TermList* entry. This chapter deals with the classes for term lists and term nodes and should be read if the user is interested in implementing algorithms for polynomials which manipulate these items directly.

6.1 Definition

An instance *tl* of the data type *TermList* is a term list with one member : a pointer to a *TermNode* , which is the pointer to the first term node in the list. A instance *tn* of type *TermNode* is a term node which has three entries : an *int* entry representing the degree, a pointer to *Polynomial* entry representing the coefficient polynomial and a pointer to the next *TermNode* in the term list.

6.2 Constructors

Constructor	Usage	Explanation
TermList ()	*TermList tl* ;	Empty *TermList* .
TermList (*TermNode tn*)	*TermList tl* = *TermList* (*tn*);	Allocates memory for *tn* and copies its entries to that block
TermList (*TermNode* * *tn*)	*TermList tl* = *TermList* (*tn*);	Sets the pointer to *TermNode* in *tl* to *tn*
TermNode ()	*TermNode tn* ;	*TermNode* with deg, coeff and next = 0
TermNode (*int* ,*Polynomial* *)	*TermNode tn* = *TermNode* (*d*,*pp*)	*TermNode* with deg = n and coeff = *pp*.
TermNode (*int* ,*Polynomial*)	*TermNode tn* = *TermNode* (*d*,*p*)	*TermNode* with deg = n and allocate memory for coeff and set it to p.

6.3 Member Functions

In the tables below t is the *TermList* or *TermNode* for which the member functions are called. The first table is *TermList* member functions, the second gives the *TermNode* member functions.

Return Type	Usage	Explanation
bool	t.is_null()	Returns true if the entry is the 0 pointer.
int	t.lead_degree()	Returns the degree of the first term node.
Polynomial	t.lead_coeff()	Returns the coeff of the first term node.
TermNode *	t.tnode_of()	Returns the pointer to the first term node.
TermNode *	t.last()	Returns pointer to the last term node in the list.
TermList	t.rest_of()	Returns the remainder of the list excluding first *TermNode* . Returns it as a *TermList* object.
bool	t.is_zerotermlist()	Returns true if the entry is the 0 pointer.

Return Type	Usage	Explanation
TermNode *	t.next_of()	Returns pointer to the next *TermNode* .
void	t.set_next(*TermNode*)	Sets next *TermNode* to argument.
Polynomial *	t.coeff_of()	Returns the member for pointer to coeff poly.
Polynomial	t.coeff_poly()	Returns the coeff poly.

6.4 Other Functions

Some of the member functions have been rewritten to make the syntax easier :

Return Type	Usage	Explanation
TermNode *	tnode_of(*TermList*)	Returns the pointer to the first term node.
bool	is_zerotermlist(*TermList*)	Returns true if the entry is the 0 pointer.

6.5 Operators

The *TermList* operators do the obvious thing with the operands.

6.5.1 Arithmetic Operators

The operands here can be combinations of *TermList* with *Polynomial* and *Field_element* . (E.g. *TermList* + *Polynomial* will add the polynomial to the termlist. It will treat the polynomial as essential a *TermNode* with degree 0, and coefficient the given polynomial.)

Return Type	Usage	Explanation
TermList	$a + b$	Addition.
TermList	$a - b$	Subtraction.
TermList	$a * b$	Multiplicaton.
TermList	a/b	Division (returns quotient on division).
TermList	$a \% b$	Remainder (returns remainder on division).

6.5.2 Relational Operators

None.

6.5.3 Other Operators and Functions

Return Type	Usage	Explanation

6.6 Header Files

- **Primary header file** *polynomial.h* (Use this file with the #include directive)

6.7 Usage Examples

```
#include "polynomial.h"

int
main(int argc, char** argv)
{
  Polynomial p;
  ...

  /* Example 1 : Get the coeff of the term of the second highest
     degree in a given polynomial. (Remember that the polynomial is
     always maintained as a sorted list.) */

  Polynomial coeff;
  TermList t = termlist_of(p);
  TermNode* tn = tnode_of(t);
  if (tn != 0) tn = tn->next_of();
  if (tn != 0) coeff = tn->coeff_poly(); // coeff now contains the poly

  /* Example 2 : Output the degrees of every term node in the term list */
  for (tn=tnode_of(t);tn != 0; tn = tn->next_of()) {
    cout << tn->degree_of() << "\n";
  }

  /* For other examples see the file termlist.c which implements the
     various operators for the TermList class */
}
```