

Polynomials; a Fortran 90 module

Per Håkan Lundow

May 21, 2001

Abstract

The author has written a Fortran 90 based package for multivariable polynomials. This paper gives an introduction to its usage and contains a reference of its procedures.

1 Introduction

If you, like the author, are working in combinatorics then you might need to compute some generating function explicitly. The first tool to try then is of course Mathematica (or Maple). However, when it comes to polynomials, these programs aren't very efficient. What we need then is the workhorse of mathematics, Fortran 90.

The package presented here supports several types of coefficients and allows the polynomials to grow dynamically in size. It also supports negative exponents of the variables so we are actually dealing with generalised polynomials. The output format can be controlled to a large extent allowing the polynomials to read using e.g. Mathematica. The module is especially efficient when your polynomials are sparse. For example, if the degree of the exponents in the polynomials vary between say, 0 and 1000, but less than half of the coefficients are non-zero, then you are probably using the right module.

The package is portable over a wide range of computers since it makes no assumptions on machine specifics.

2 An example

2.1 Choosing the coefficients

The package consists of several modules split up on several files. The files are `monoreal.f90`, `monmpi.f90` and `polynom.f90`. The first two files defines the monomials and these consist of a coefficient and an array of integer exponents. The polynomials are defined in the last file as a linked list of monomials.

First of all we need to settle what kind of coefficients you need for your polynomials. Currently there are two choices, real or multiprecision-integers (henceforth called mp-integers. For small integer (say less than 53 binary digits) coefficients or for numerical coefficients you use real coefficients. If your coefficients are large integers (signed or unsigned) then you use mp-integer coefficients. If you need mp-integers then the reader is first referred to the mp-integer module and its manual [1]. For the real coefficients we use the file `monoreal.f90` and for mp-integer coefficients we use the file `monmpi.f90`.

2.2 Setting the parameters

Let us first assume that we have settled for real coefficients. Open the file `monoreal.f90` with your text editor (emacs, vi, alpha,...) and locate the USER DEFINED section at the top of the file.

Now decide if you need your coefficients to be signed or unsigned. If your coefficients are always non-negative then set the constant `signed_coeff` to `.FALSE.`, otherwise you set it to `.TRUE.`. Next we set the number of variables in the polynomials with the constant `variables`. Finally set the kind type parameter of the real type, e.g. by setting the precision argument of the `SELECTED_REAL_KIND` function. Typically (but far from always!) setting the precision to 6 gives single-precision reals (24 binary digits), setting it to 15 gives double precision (53 binary digits) and 31 gives quadruple precision (106 binary digits). This holds for IBM computers but not for Cray computers where the only real type is of double precision.

Now we assume that we need coefficients of mp-integer type. For unsigned mp-integers we need the file `mpiunsgn.f90`, set its constants according to the instructions in [1]. Now we open the file `monompi.f90` and locate the USE-statements right above the `IMPLICIT NONE`-statement. Uncomment the line

```
USE UNSIGNED_MP_INTEGERS
```

and make sure the other USE-statements are commented. Then we locate the USER DEFINED section right below these statements. Set the constant `signed_coeff` to `.FALSE.` and set also the number of variables.

If we want signed mp-integers as coefficients then we also need the file `mpisgn.f90`. Set the USE-statement to

```
USE SIGNED_MP_INTEGERS
```

and the constant `signed_coeff` to `.TRUE.`.

For compactly stored mp-integers we need the file `mpicmp.f90` Set the USE-statement accordingly and set `signed_coeff` to `.FALSE.`.

The file `polynom.f90` requires no settings of constants.

2.3 Compiling

Now we compile the files. On an IBM we might use the following command (IBM requires the `.f` suffix)

```
> xlf90 -c -O3 -qstrict monoreal.f polynom.f
```

if we had settled for real coefficients. On a SUN or a Cray we use:

```
> f90 -c -O3 monoreal.f90 polynom.f90
```

If we chose, say, unsigned mp-integers as coefficients then we compile (if we continue with the SUN case) as follows

```
> f90 -c -O3 mpiunsgn.f90 monompi.f90 polynom.f90
```

For signed mp-integers we compile as in

```
> f90 -c -O3 mpiunsgn.f90 mpisgn.f90 monompi.f90 polynom.f90
```

and for compactly stored mp-integers we compile as in

```
> f90 -c -O3 mpiunsgn.f90 mpicmp.f90 monompi.f90 polynom.f90
```

2.4 A simple program

Next we need a main program. The following program is a (not very useful admittedly) simple example demonstrating the feel of the package. We assume

that the number of variables is set to 2. Your program must have a line

```
USE Polynomials
```

after the PROGRAM-statement and before the IMPLICIT NONE-statement. This gives the program access to the Monomial-type, the Polynomial-type, the constants and the procedures.

```
PROGRAM Main
USE Polynomials
IMPLICIT NONE

TYPE(Polynomial) :: p, q

CALL Init_Poly (p)      ! Always initialise the polynomials
CALL Init_Poly (q)      ! before they are used.

p = 0
CALL Add_Poly (p, 2*x_mono( 1 )**3) ! p = p + 2*x**3
CALL Add_Poly (p,  x_mono( 2 )**2) ! p = p + y**2
! Now p has value 2 * x**3 + y**2

q = p                    ! Copy p to q
CALL Mult_Poly (p, 9)     ! p = 9*p
CALL Power_Poly (p, p, 2) ! p = p ** 2
CALL Add_Poly (p, q, x_mono( 1 )) ! p = p + q*x

CALL Write_Poly (6, p) ! Write to standard output.

! The following is printed:
! 81 0 4           ! 81 * y**4 +
! 324 3 2          ! 324 * x**3 * y**2 +
! 1 1 2           ! 1 * x * y**2 +
! 324 6 0          ! 324 * x**6 +
! 2 4 0           ! 2 * x**4
! 0 0 0           ! END-OF-POLYNOMIAL

STOP
END PROGRAM Main
```

Compiling and running the program will print the polynomial on the screen monomial-wise as a sequence of lines in the following fashion:

```
coeff1 expx1 expy1
coeff2 expx2 expy2
coeff3 expx3 expy3
...
```

```
0 0 0
```

where coeff1 is the coefficient of $x^{**}expx1 * y^{**}expy1$, coeff2 is the coefficient of $x^{**}expx2 * y^{**}expy2$ etc.

References

[1] P.H. Lundow, Multiprecision integers; a Fortran 90 module

3 Reference

Throughout this section we assume that P is a polynomial, M is a monomial and I is a default integer. Also, let P1 denote a rank 1 array and P2 a rank 2 array of polynomials.

3.1 Construction and destruction

Before a polynomial is used it must be constructed. This is done with the procedure `Init_Poly`. To prevent memory leaks a polynomial should be destroyed when it is not needed anymore (though this is not required). This is done with the procedure `Clear_Poly`. Both procedures are extended for arrays of rank j . Calling `Clear_Poly` with no arguments destroys all polynomials used by the program.

```
TYPE(Polynomial) :: p, q( 10 )
CALL Init_Poly (p)
CALL Init_Poly (q)
CALL Clear_Poly (p)
CALL Clear_Poly (q)
CALL Clear_Poly () ! Destroy all polynomials.
```

Henceforth we assume that all polynomials have been constructed before they are used.

A monomial has two components, a coefficient (`coeff`) and a rank 1 array (`degree`) of length `variables` containing the exponents of the variables. Defining values of monomial can be done in the following way

```
TYPE(Monomial) :: u
u % coeff = 12
u % degree = (/ 3, 4 /)
! u has value 21* x**3 * y**4
u = Monomial (12*one_coeff, (/3,4/)) ! Alternative
```

3.2 Assignments

The following types of assignments are allowed: `P = P`, `P = M`, `P = I`, `M = M` and `M = I` as in the following examples:

```
TYPE(Polynomial) :: p, q
TYPE(Monomial) :: u
! ...
u = 3 ! u has value 3
p = u ! p has value 3
u = 7 ! u has value 7
p = 9 ! p has value 9
q = p ! q has value 9
```

Array assignments of the following types are also allowed: P2 = P2, P2 = P, P1 = P1, P1 = P, P2 = M, P2 = I, P1 = M, P1 = I. These work as array assignments would for e.g. integers.

```

TYPE(Polynomial) :: p, q( 10 ), r( 10, 20 )
TYPE(Monomial) :: u
! ...
r = p
q = p
r = u
q = 2

```

Finally there is the assignment of type M1 = P, i.e. the monomials of the polynomial is copied to a rank 1 array of monomials. The array must be of size at least the number of monomials in the polynomial. Any remaining elements of the array are set to zero.

```

TYPE(Polynomial) :: p
TYPE(Monomial) :: u( 100 )
! ...
u = p

```

3.3 Operators

You can use the arithmetic operators on the monomials and they can also be combined with integers.

The arithmetic operators are -, *, / and ** and the following operations are allowed: -M, M * M, M * I, I * M, M / M, M / I, I / M and M ** I. The result of these operations is a monomial. Note that -M is not defined if the coefficient is of unsigned type and of positive value.

The relational operators == and /= are defined for the following types: P == P, P /= P, M == M and M /= M. The result of these operations are of type logical.

```

TYPE(Polynomial) :: p, q
TYPE(Monomial) :: u, v
! ...
IF ( p == q ) THEN
    u = 2*v
ELSE IF ( u == v ) THEN
    u = 0
END IF

```

3.4 Constants

There is a collection of useful constants available to the user.

`signed_coeff` is of type logical and is set to `.FALSE.` if coefficients are non-negative. Otherwise it is set to `.TRUE.`.

`variables` is of type integer and is set to the number of variables in the polynomials.

`rk_coeff` is of type integer and its value is the kind type parameter for coefficients. This is only available when using real coefficients.

`zero_coeff` is of the same type as the coefficients and has the value zero.

`one_coeff` is of the same type as the coefficients and has the value one.

`two_coeff` is of the same type as the coefficients and has the value two.

`ten_coeff` is of the same type as the coefficients and has the value ten.

`zero_mono` is of type monomial and has the value zero.

`one_mono` is of type monomial and has the value one.

`two_mono` is of type monomial and has the value two.

`ten_mono` is of type monomial and has the value ten.

`x_mono(1:variables)` is a rank 1 array of type monomial. If we have n variables and denote them by x_1, x_2, \dots, x_n , then `x_mono(1)` is x_1 , `x_mono(2)` is x_2 , etc.

3.5 Enumeration

The package also contains an enumeration type, `Enum_Poly`. This type allows you to iterate through the monomials of a polynomial. An example is probably best for describing its usage:

```
! Compute the sum of the coefficients of poly
TYPE(Polynomial) :: poly
TYPE(Monomial)   :: mono1, mono2
TYPE(Enum_Poly)  :: enum
mono2 = zero_mono
CALL Enumeration (enum, poly)
DO WHILE ( .NOT. LastQ (enum) )
    CALL Next (enum, mono1)
    mono2 % coeff = mono2 % coeff + mono1 % coeff
END DO
! mono2 % coeff is the sum of the coefficients.
```

3.6 Procedures

Throughout this section, let n be the number of variables and let x_1, \dots, x_n denote the variables. In the examples it is assumed that n is 2.

Init_Poly (p)

Subroutine. Constructs a polynomial.

Argument is of type polynomial and is INTENT(IN OUT). Argument can also be an array of rank at most two.

TYPE(Polynomial) :: p

CALL Init_Poly (p)

Clear_Poly (p)

Subroutine. Destroys a polynomial.

Argument is of type polynomial and is INTENT(IN OUT). Argument can also be an array of rank at most two .

Calling this subroutine without an argument destroys all the polynomials used by the program.

TYPE(Polynomial) :: p

CALL Clear_Poly (p)

Add_Poly (p, q, r)

Subroutine. p is set to the sum $p + qr$.

The first argument is of type polynomial and is INTENT(IN OUT). The second argument can be either an integer, an integer array of rank 1 and size n , a monomial or a polynomial and it is INTENT(IN). The third argument is optional and either of type integer, an integer array of rank 1 and length n , a monomial or a polynomial and it is INTENT(IN). The third argument can only be present if the second argument is a polynomial.

The first argument must not refer to the same variable as the second or third argument.

If the third argument is present then the second argument must be a polynomial. If the third argument is a polynomial, a monomial or an integer then p is set to $p + qr$ on return. If the third argument is an integer array a then it must be of size n and its elements are the exponents of a monomial with coefficient 1. Then p is set to $p + qx_1^{a_1} \dots x_n^{a_n}$ on return.

If the third argument is not present then the following holds. If the second argument is a polynomial, a monomial or an integer then p is set to $p + q$ on return. If the second argument is an integer array a then it must be of size n and its elements are the exponents of a monomial with coefficient 1. Then p is set to $p + x_1^{a_1} \dots x_n^{a_n}$ on return.

```

TYPE(Polynomial) :: p, q, r
TYPE(Monomial)   :: u
! ...
CALL Add_Poly (p, q)      ! p = p + q
CALL Add_Poly (p, u)      ! p = p + u
CALL Add_Poly (p, 3)      ! p = p + 3
CALL Add_Poly (p, (/2,3/)) ! p = p + x**2 * y**3
CALL Add_Poly (p, q, r)   ! p = p + q*r
CALL Add_Poly (p, q, u)   ! p = p + q*u
CALL Add_Poly (p, q, 3)   ! p = p + q*3
CALL Add_Poly (p, q, (/2,3/)) ! p = p + q * x**2 * y**3
CALL Add_Poly (p, q, q)   ! p = p + q*q

```

Mult_Poly (p, q, r)

Subroutine. Multiply polynomials.

The first argument is of type polynomial and is INTENT(IN OUT). The second argument can be either an integer, an integer array of rank 1 and size n , a monomial or a polynomial and it is INTENT(IN). The third argument is an optional polynomial with INTENT(IN). The third argument can only be present if the second argument is a polynomial.

If the third argument is present then the second argument must be a polynomial. Then p is set to qr on return. The first argument can refer to the same variable as the second and third, i.e. the first argument behaves as if it is INTENT(OUT).

If the third argument is not present then the following holds. If the second argument is a monomial or an integer then p is set to pq on return. If the second argument is an integer array a then it must be of size n and its elements are the exponents of a monomial with coefficient 1. Then p is set to $px_1^{a_1} \dots x_n^{a_n}$ on return.

```

TYPE(Polynomial) :: p, q, r
TYPE(Monomial)   :: u
! ...
CALL Mult_Poly (p, u)      ! p = p * u
CALL Mult_Poly (p, 3)      ! p = p * 3
CALL Mult_Poly (p, (/2,3/)) ! p = p * x**2 * y**3
CALL Mult_Poly (p, q, r)   ! p = q * r
CALL Mult_Poly (p, p, p)   ! p = p * p

```

Div_Poly (p, u)

Subroutine. Divide a polynomial by a monomial.

The first argument is of type polynomial and is INTENT(IN OUT). The second argument can be either a monomial, an integer or an integer array of rank 1 and size n .

On return p is set to p/u .

If the second argument is an integer array a then it must be of size n and its elements are the exponents of a monomial with coefficient 1. Then p is set to $px_1^{-a_1} \dots x_n^{-a_n}$ on return.

If the coefficients are of an integer type (e.g. mp-integers) then integer divisions are performed.

```
TYPE(Polynomial) :: p, q, r
TYPE(Monomial)   :: u
! ...
CALL Div_Poly (p, u)      ! p = p / u
CALL Div_Poly (p, 3)     ! p = p / 3
CALL Div_Poly (p, (/2,3/)) ! p = p / x**2 / y**3
```

Power_Poly (p, q, k)

Subroutine. Power of a polynomial.

The first argument is of type polynomial and is INTENT(OUT). The second argument is a polynomial of INTENT(IN) and the third argument is an integer.

On return p is set to q^k . The first argument can refer to the same polynomial as the second argument.

```
TYPE(Polynomial) :: p, q
! ...
CALL Power_Poly (p, q, 3) ! p = q ** 3
CALL Power_Poly (p, p, 3) ! p = p ** 3
```

Sum_Poly (p, q)

Subroutine. Computes the sum of an array.

The first argument is of type polynomial and is INTENT(IN OUT). The second argument, of INTENT(IN), is a rank 1 array of polynomials or monomials.

The first argument must not refer to an element of the second argument.

On return p is set to $\sum q_i$.

```
TYPE(Polynomial) :: p, q( 10 )
TYPE(Monomial)   :: u( 100 )
! ...
CALL Sum_Poly (p, q)      ! p = q( 1 ) + q( 2 ) + ...
CALL Sum_Poly (p, u)     ! p = u( 1 ) + u( 2 ) + ...
```

Swap_Poly (p, q)

Subroutine. Efficient swap of contents of polynomials.

Both arguments are polynomials and of INTENT(IN OUT).

On return p is set to q and q is set to p .

This subroutine is considerably faster than the sequence $r = p, p = q, q = r$.

```
TYPE(Polynomial) :: p, q
! ...
CALL Swap_Poly (p, q) ! p has value q, q has value p.
```

Diff_Poly (p, i)

Subroutine. Derivative of polynomial.

The first argument is of type polynomial and is INTENT(IN OUT).
The second argument is an integer of INTENT(IN).

On return p is set to $\partial p/\partial x_i$.

TYPE(Polynomial) :: p

! ...

CALL Diff_Poly (p, 1) ! p = dp/dxi

Diff_Mono (u, i)

Function. Derivative of monomial.

The first argument is of type monomial and the second argument is
an integer. Both are of INTENT(IN).

Result is of type monomial.

The value of the result $\partial u/\partial x_i$.

TYPE(Monomial) :: u

u = Diff_Mono (3*x_mono(1) ** 4, 1)

! u has value 12*x**3

ZeroQ (p)

Function. Test for zero value of argument.

One argument, either a polynomial or a monomial.

Result is of type logical.

The value of the result is .TRUE. if the argument is zero, otherwise
the value of the result is .FALSE..

Note that a monomial is zero if either the coefficient is zero or an
exponent is less than or equal to -HUGE(0).

TYPE(Polynomial) :: p

TYPE(Monomial) :: u

LOGICAL :: a

! ...

a = ZeroQ (one_mono) ! a has value .FALSE.

p = one_mono

a = ZeroQ (p) ! a has value .FALSE.

u % coeff = two_coeff

u % degree = -HUGE (0)

a = ZeroQ (u) ! a has value .TRUE.

p = 0

a = ZeroQ (p) ! a has value .TRUE.

Length_Poly (p)

Function. Length of polynomial.

One argument of type polynomial. The argument can also be an array of rank at most two.

Result is of type integer. If the argument is an array then the result has the same shape as the argument.

The value of the result is the number of non-zero monomials in the polynomial p .

```
TYPE(Polynomial) :: p, q( 5, 10 )
INTEGER :: i, j( 5, 10 )
! ...
p = 1
CALL Add_Poly (p, x_mono( 1 ))
CALL Power_Poly (p, p, 4) ! p has value (1+x)**4
i = Length_Poly (p)      ! i has value 5
q = p
j = Length_Poly (q) ! all elements of j have value 5
```

Order (p, q)

Function. Returns 1, 0 or -1 depending on whether $p < q$, $p = q$ or $p > q$.

Two argument of the same type, either polynomial or monomial.

Result is of type integer. The value of the result is 1 if the first argument precedes the second argument in the canonical order. If they are equal then the result is 0, otherwise the result is -1. is an array then the result has the same shape as the argument.

In a polynomial, the monomials are stored in the canonical order.

The canonical order imposed on the polynomials and monomials allows them to be sorted.

```
TYPE(Polynomial) :: p
INTEGER :: i
i = Order (one_mono, x_mono( 1 )) ! i has value -1
i = Order (p, p)                  ! i has value 0
```

Enumeration (e, p)

Subroutine. Start an enumeration of the monomials of a polynomial.

Two arguments, the first of type `Enum_Poly` the second of type polynomial. The first argument is `INTENT(OUT)` and the second is `INTENT(IN)`.

The first argument is set to point to the first monomial of polynomial.

See section 3.5

Next (e, u)

Subroutine. Update the second argument to the next monomial in an enumeration.

Two arguments, the first of type `Enum_Poly` the second of type monomial. The first argument is `INTENT(IN OUT)` and the second is `INTENT(OUT)`.

Update the monomial to the next monomial in an enumeration of the monomials in a polynomial.

See section 3.5

LastQ (e)

Function. Test if we have iterated through all monomials in an enumeration.

One argument of type `Enum_Poly`.

Result is of type logical.

The value of the result is `.TRUE.` if we have iterated through all monomials in an enumeration, otherwise it is `.FALSE.`.

See section 3.5

Write_Poly (unit, p, fmt, iostat)

Subroutine. Write a polynomial or an array of polynomials to a unit.

Four arguments, the last two are optional. The first and fourth argument are of type integer, the second of type polynomial, the third is a string. All arguments are `INTENT(IN)` except the fourth which is `INTENT(OUT)`.

Write polynomial p , one monomial on each line with a line of zeros after the last, to unit *unit*. The *iostat* variable works like for `WRITE`. The *fmt* variable can be used for editing the monomials.

If *fmt* is not present then a default integer format is used. If the unit is opened for unformatted output then *fmt* must not be present.

Argument p can also be an array of rank at most two. The elements are then written in array element order.

```
TYPE(Polynomial) :: p
! ...
! assume p has value 1 + 2*x**2 + 5*y**3
CALL Write_Poly (6, p)
! The following is printed on standard output:
! 5 0 3
! 2 2 0
! 1 0 0
! 0 0 0
```

Read_Poly (unit, p, iostat)

Subroutine. Read a polynomial or an array of polynomials from a unit.

Three arguments, the third is optional. The first and third argument are of type integer, the second of type polynomial. The second and third argument is `INTENT(OUT)`, the first is `INTENT(IN)`.

Read a polynomial p , one monomial at a time ending with a line of zeros after the last, from unit *unit*. The *iostat* variable works like for READ.

The subroutine detects if the unit is opened for unformatted input. Unformatted input should only be used if the polynomial has been written by `Write_Poly`.

Argument p can also be an array of rank at most two. The elements are then read in array element order.

The monomials need not be entered in any particular order, though for maximum efficiency they should be entered in the canonical order imposed by the `Order`-function, the same order as they are written by `Write_Poly`.

```
TYPE(Polynomial) :: p
! ...
CALL Read_Poly (5, p)
! Assume the following is typed on standard input
! with a return after each line:
! 5 0 3
! 2 2 0
! 1 0 0
! 0 0 0
! Now p has value 1 + 2 * x**2 + 5 * y**3
```

`Write_Mono (unit, p, fmt, iostat)`

Subroutine. Write a monomial to a unit.

Four arguments, the last two are optional. The first and fourth argument are of type integer, the second of type polynomial, the third is a string. All arguments are `INTENT(IN)` except the fourth which is `INTENT(OUT)`.

Write monomial p , on one line, to unit *unit*. The *iostat* variable works like for `WRITE`. The *fmt* variable can be used for editing the monomials.

If *fmt* is not present then a default integer format is used.

```
TYPE(Monomial) :: u
! ...
! assume u has value 2 * x**3 * y**4
CALL Write_Mono (6, u)
! The following is printed on standard output:
! 2 3 4
! For real coefficients:
! CALL Write_Mono (6, u, fmt='(F5.0,2I3)')
! 2.000 2 4
```

`Read_Mono (unit, p, iostat)`

Subroutine. Read a monomial from a unit.

Three arguments, the third is optional. The first and third argument are of type integer, the second of type monomial. The second and third argument is `INTENT(OUT)`, the first is `INTENT(IN)`.

Read a monomial p , as one line, from unit *unit*. The *iostat* variable works like for `READ`.

```
TYPE(Monomial) :: u
! ...
CALL Read_Mono (6, u)
! Assume the following is typed on standard input
! with a return at the end:
! 2 3 4
! u has value 2 * x**3 * y**4
```