Truncated Power Series

Alan Barnes
Dept. of Computer Science and Applied Mathematics
Aston University, Aston Triangle,
Birmingham B4 7ET
GREAT BRITAIN
Email: barnesa@aston.ac.uk

and

Julian Padget
School of Mathematics, University of Bath
Claverton Down, Bath, BA2 7AY
GREAT BRITAIN
Email: jap@maths.bath.ac.uk

1 Introduction

This package implements formal power series expansions in one variable using the domain mechanism of REDUCE. This means that power series objects can be added, multiplied, differentiated etc. like other first class objects in the system. A lazy evaluation scheme is used in the package and thus terms of the series are not evaluated until they are required for printing or for use in calculating terms in other power series. The series are extendible giving the user the impression that the full infinite series is being manipulated. The errors that can sometimes occur using series that are truncated at some fixed depth (for example when a term in the required series depends on terms of an intermediate series beyond the truncation depth) are thus avoided.

Below we give a brief description of the operators available in the power series package together with some examples of their use.

1.1 PS Operator

Syntax:

`PS`(EXPRN:*algebraic*,DEPVAR:*kernel*, ABOUT:*algebraic*):*ps object*

The `PS` operator returns a power series object (a tagged domain element) representing the univariate formal power series expansion of EXPRN with respect to the dependent variable DEPVAR about the expansion point ABOUT. EXPRN may itself contain power series objects.

The algebraic expression ABOUT should simplify to an expression which is independent of the dependent variable DEPVAR, otherwise an error will result. If ABOUT is the identifier `INFINITY` then the power series expansion about DEPVAR = ∞ is obtained in ascending powers of 1/DEPVAR.

If the command is terminated by a semi-colon, a power series object representing EXPRN is compiled and then a number of terms of the power series expansion are evaluated and printed. The expansion is carried out as far as the value specified by `PSEXPLIM`. If, subsequently, the value of `PSEXPLIM` is increased, sufficient information is stored in the power series object to enable the additional terms to be calculated without recalculating the terms already obtained.

If the command is terminated by a dollar symbol, a power series object is compiled, but at most one term is calculated at this stage.

If the function has a pole at the expansion point then the correct Laurent series expansion will be produced.

The following examples are valid uses of `PS`:

```
psexplim 6;
ps(log x,x,1);
ps(e**(sin x),x,0);
ps(x/(1+x),x,infinity);
ps(sin x/(1-cos x),x,0);
```

New user-defined functions may be expanded provided the user provides LET rules giving

1. the value of the function at the expansion point
2. a differentiation rule for the new function.

For example

```
operator sech;
forall x let df(sech x,x)= - sech x * tanh x;
let sech 0 = 1;
ps(sech(x**2),x,0);
```

The power series expansion of an integral may also be obtained (even if REDUCE cannot evaluate the integral in closed form). An example of this is

```
ps(int(e**x/x,x),x,1);
```

Note that if the integration variable is the same as the expansion variable then REDUCE's integration package is not called; if on the other hand the two variables are different then the integrator is called to integrate each of the coefficients in the power series expansion of the integrand. The constant of integration is zero by default.

1.2 PSEXPLIM Operator

Syntax:

`PSEXPLIM`(UPTO:*integer*):*integer*

or

`PSEXPLIM`():*integer*

Calling this operator sets an internal variable of the TPS package to the value of UPTO (which should evaluate to an integer). The value returned is the previous value of this variable. The default value is six.

If `PSEXPLIM` is called with no argument, the current value for the expansion limit is returned.

1.3 PSORDLIM Operator

Syntax:

`PSORDLIM`(UPTO:*integer*):*integer*

or

`PSORDLIM`():*integer*

An internal variable is set to the value of `UPTO` (which should evaluate to an integer). The value returned is the previous value of the variable. The default value is 15.

If `PSORDLIM` is called with no argument, the current value is returned.

The significance of this control is that the system attempts to find the order of the power series required, that is the order is the degree of the first non-zero term in the power series. If the order is greater than the value of this variable an error message is given and the computation aborts. This prevents infinite loops in examples such as

```
ps(1 - (sin x)**2 - (cos x)**2,x,0);
```

where the expression being expanded is identically zero, but is not recognized as such by REDUCE.

1.4 PSTERM Operator

Syntax:

`PSTERM`(TPS:*power series object*, NTH:*integer*):*algebraic*

The operator `PSTERM` returns the NTH term of the existing power series object TPS. If NTH does not evaluate to an integer or TPS to a power series object an error results. It should be noted that an integer is treated as a power series.

1.5 PSORDER Operator

Syntax:

`PSORDER`(TPS:*power series object*):*integer*

The operator `PSORDER` returns the order, that is the degree of the first non-zero term, of the power series object TPS. TPS should evaluate to a power series object or an error results. If TPS is zero, the identifier `UNDEFINED` is returned.

1.6 PSSETORDER Operator

Syntax:

`PSSETORDER`(TPS:*power series object*, ORD:*integer*):*integer*

The operator `PSSETORDER` sets the order of the power series TPS to the value ORD, which should evaluate to an integer. If TPS does not evaluate to a power series object, then an error occurs. The value returned by this operator is the previous order of TPS, or 0 if the order of TPS was undefined. This operator is useful for setting the order of the power series of a function defined by a differential equation in cases where the power series package is inadequate to determine the order automatically.

1.7 PSDEPVAR Operator

Syntax:

`PSDEPVAR`(TPS:*power series object*) :*identifier*

The operator `PSDEPVAR` returns the expansion variable of the power series object TPS. TPS should evaluate to a power series object or an integer, otherwise an error results. If TPS is an integer, the identifier `UNDEFINED` is returned.

1.8 PSEXPANSIONPT operator

Syntax:

`PSEXPANSIONPT`(TPS:*power series object*):*algebraic*

The operator `PSEXPANSIONPT` returns the expansion point of the power series object TPS. TPS should evaluate to a power series object or an integer, otherwise an error results. If TPS is integer, the identifier `UNDEFINED` is returned. If the expansion is about infinity, the identifier `INFINITY` is returned.

1.9 PSFUNCTION Operator

Syntax:

`PSFUNCTION`(TPS:*power series object*):*algebraic*

The operator `PSFUNCTION` returns the function whose expansion gave rise to the power series object TPS. TPS should evaluate to a power series object or an integer, otherwise an error results.

1.10 PSCHANGEVAR Operator

Syntax:

`PSCHANGEVAR`(TPS:*power series object*, X:*kernel*):*power series object*

The operator `PSCHANGEVAR` changes the dependent variable of the power series object TPS to the variable X. TPS should evaluate to a power series object and X to a kernel, otherwise an error results. Also X should not appear as a parameter in TPS. The power series with the new dependent variable is returned.

1.11 PSREVERSE Operator

Syntax:

`PSREVERSE`(TPS:*power series object*):*power series*

Power series reversion. The power series TPS is functionally inverted. Four cases arise:

1. If the order of the series is 1, then the expansion point of the inverted series is 0.
2. If the order is 0 *and* if the first order term in TPS is non-zero, then the expansion point of the inverted series is taken to be the coefficient of the zeroth order term in TPS.
3. If the order is -1 the expansion point of the inverted series is the point at infinity. In all other cases a REDUCE error is reported because the series cannot be inverted as a power series. Puiseux expansion would be required to handle these cases.
4. If the expansion point of TPS is finite it becomes the zeroth order term in the inverted series. For expansion about 0 or the point at infinity the order of the inverted series is one.

If TPS is not a power series object after evaluation an error results.

Here are some examples:

```
ps(sin x,x,0);
psreverse(ws); % produces series for asin x about x=0.
ps(exp x,x,0);
psreverse ws; % produces series for log x about x=1.
ps(sin(1/x),x,infinity);
psreverse(ws); % series for 1/asin(x) about x=0.
```

1.12 PSCOMPOSE Operator

Syntax:

PSCOMPOSE(TPS1:*power series*, TPS2:*power series*):*power series*

PSCOMPOSE performs power series composition. The power series TPS1 and TPS2 are functionally composed. That is to say that TPS2 is substituted for the expansion variable in TPS1 and the result expressed as a power series. The dependent variable and expansion point of the result coincide with those of TPS2. The following conditions apply to power series composition:

1. If the expansion point of TPS1 is 0 then the order of the TPS2 must be at least 1.
2. If the expansion point of TPS1 is finite, it should coincide with the coefficient of the zeroth order term in TPS2. The order of TPS2 should also be non-negative in this case.
3. If the expansion point of TPS1 is the point at infinity then the order of TPS2 must be less than or equal to -1.

If these conditions do not hold the series cannot be composed (with the current algorithm terms of the inverted series would involve infinite sums) and a REDUCE error occurs.

Examples of power series composition include the following.

```
a:=ps(exp y,y,0);  b:=ps(sin x,x,0);
pscompose(a,b);
% Produces the power series expansion of exp(sin x)
% about x=0.

a:=ps(exp z,z,1); b:=ps(cos x,x,0);
```

```
pscompose(a,b);
% Produces the power series expansion of exp(cos x)
% about x=0.

a:=ps(cos(1/x),x,infinity);  b:=ps(1/sin x,x,0);
pscompose(a,b);
% Produces the power series expansion of cos(sin x)
% about x=0.
```

1.13 PSSUM Operator

Syntax:

PSSUM(J:*kernel* = LOWLIM:*integer*, COEFF:*algebraic*, X:*kernel*,
ABOUT:*algebraic*, POWER:*algebraic*):*power series*

The formal power series sum for J from LOWLIM to INFINITY of

```
COEFF*(X-ABOUT)**POWER
```

or if ABOUT is given as INFINITY

```
COEFF*(1/X)**POWER
```

is constructed and returned. This enables power series whose general term is known to be constructed and manipulated using the other procedures of the power series package.

J and X should be distinct simple kernels. The algebraics ABOUT, COEFF and POWER should not depend on the expansion variable X, similarly the algebraic ABOUT should not depend on the summation variable J. The algebraic POWER should be a strictly increasing integer valued function of J for J in the range LOWLIM to INFINITY.

```
pssum(n=0,1,x,0,n*n);
% Produces the power series summation for n=0 to
% infinity of x**(n*n).

pssum(m=1,(-1)**(m-1)/(2m-1),y,1,2m-1);
% Produces the power series expansion of atan(y-1)
% about y=1.
```

```
pssum(j=1,-1/j,x,infinity,j);
% Produces the power series expansion of log(1-1/x)
% about the point at infinity.

pssum(n=0,1,x,0,2n**2+3n) + pssum(n=1,1,x,0,2n**2-3n);
% Produces the power series summation for n=-infinity
% to +infinity of x**(2n**2+3n).
```

1.14 PSCOPY Operator

Syntax:

PSCOPY(TPS:*power series*):*power series*

This procedure returns a copy of the power series TPS. The copy has no shared sub-structures in common with the original series. This enables substitutions to be performed on the series without side-effects on previously computed objects. For example:

```
clear a;
b := ps(sin(a*x)), x, 0);
b where a => 1;
```

will result in a being set to 1 in each of the terms of the power series and the resulting expressions being simplified. Owing to the way power series objects are implemented using Lisp vectors, this has the side-effect that the value of b is changed. This may be avoided by copying the series with PSCOPY before applying the substitution, thus:

```
b := ps(sin(a*x)), x, 0);
pscopy b where a => 1;
```

1.15 PSTRUNCATE Operator

Syntax:

PSTRUNCATE(TPS:*power series* POWER: *integer)* :*algebraic*

This procedure truncates the power series TPS discarding terms of order higher than POWER. The series is extended automtically if the value of POWER is greater than the order of last term calculated to date.

```
b := ps(sin x, x, 0);
a := pstruncate(b, 11);
```

will result in `a` being set to the eleventh order polynomial resulting in truncating the series for $sinx$ after the term involving x^{11}.

If `POWER` is less than the order of the series then 0 is returned. If `POWER` does not simplify to an integer or if `TPS` is not a power series object then Reduce errors result.

1.16 Arithmetic Operations

As power series objects are domain elements they may be combined together in algebraic expressions in algebraic mode of REDUCE in the normal way.

For example if A and B are power series objects then the commands such as:

```
a*b;
a/b;
a**2+b**2;
```

will produce power series objects representing the product,quotient and the sum of the squares of the power series objects A and B respectively.

1.17 Differentiation

If A is a power series object depending on X then the input `df(a, x);` will produce the power series expansion of the derivative of A with respect to X.

Note however that currently the input `int(a, x);` will not work as intended; instead one must input `ps(int(a, x),x,0);` in order to obtain the power series expansion of the integral of `a`.

2 Restrictions and Known Bugs

If A is a power series object and X is a variable which evaluates to itself then currently expressions such as `a*x` do not evaluate to a single power series object (although the result is formally valid). Instead use `ps(a*x,x,0)` *etc.*.

Similarly expressions such as `sin(A)` where `A` is a PS object currently will not be expanded. For example:

```
a:=ps(1/(1+x),x,0);
b:=sin a;
```

will not expand `sin(1/(1+x))` as a power series. In fact

```
SIN(1 - X + X**2 - X**3 + .....)
```

will be returned. However,

```
b:=ps(sin(a),x,0);
```

or

```
b:=ps(sin(1/(1+x)),x,0);
```

should work as intended.

The handling of functions with essential singularities is currently erratic: usually an error message

```
***** Essential Singularity
```

or

```
***** Logarithmic Singularity
```

occurs but occasionally a division by zero error or some drastic error like (for PSL) binding stack overflow may occur.

There is no simple way to write the results of power series calculation to a file and read them back into REDUCE at a later stage.