A REDUCE package for manipulation of Taylor series

Rainer Schöpf
Zentrum für Datenverarbeitung der Universität Mainz
Anselm-Franz-von-Bentzel-Weg 12
D-55099 Mainz
Germany
E–mail: Schoepf@Uni-Mainz.DE

This short note describes a package of REDUCE procedures that allow Taylor expansion in one or several variables, and efficient manipulation of the resulting Taylor series. Capabilities include basic operations (addition, subtraction, multiplication and division), and also application of certain algebraic and transcendental functions. To a certain extent, Laurent and Puiseux expansions can be performed as well. In many cases, separable singularities are detected and factored out.

1 Introduction

The Taylor package was written to provide REDUCE with some of the facilities that MACSYMA's `TAYLOR` function offers, but most of all I needed it to be faster and more space-efficient. Especially I wanted procedures that would return the logarithm or arc tangent of a Taylor series, again as a Taylor series. This turned out be more work than expected. The features absolutely required were (as usual) those that were hardest to implement, e.g., arc tangent applied to a Taylor expansion in more than one variable.

This package is still undergoing development. I'll be happy if it is of any use for you. Tell me if you think that there is something missing. I invite everybody to criticize and comment and will eagerly try to correct any errors found.

2 How to use it

The most important operator is '`TAYLOR`'. It is used as follows:

`TAYLOR`(EXP:*exprn*[,VAR:*kernel*, VAR_0:*exprn*,ORDER:*integer*]...):*exprn*

where EXP is the expression to be expanded. It can be any REDUCE object, even an expression containing other Taylor kernels. VAR is the kernel with respect to which EXP is to be expanded. VAR_0 denotes the point about which and ORDER the order up to which expansion is to take place. If more than one (VAR, VAR0, ORDER) triple is specified `TAYLOR` will expand its first argument independently with respect to each variable in turn. For example,

```
taylor(e^(x^2+y^2),x,0,2,y,0,2);
```

will calculate the Taylor expansion up to order $X^2 * Y^2$. Note that once the expansion has been done it is not possible to calculate higher orders. Instead of a kernel, VAR may also be a list of kernels. In this case expansion will take place in a way so that the *sum* of the degrees of the kernels does not exceed ORDER. If VAR_0 evaluates to the special identifier `INFINITY` `TAYLOR` tries to expand EXP in a series in 1/VAR.

The expansion is performed variable per variable, i.e. in the example above by first expanding $\exp(x^2 + y^2)$ with respect to x and then expanding every coefficient with respect to y.

There are two extra operators to compute the Taylor expansions of implicit and inverse functions:

`IMPLICIT_TAYLOR`(F:*exprn*,VAR1,VAR2:*kernel*,
$\mathrm{VAR1}_0$,$\mathrm{VAR2}_0$:*exprn*, ORDER:*integer*):*exprn*

takes a function F depending on two variables VAR1 and VAR2 and computes the Taylor series of the implicit function VAR2(VAR1) given by the equation F(VAR1,VAR2) = 0. For example,

```
implicit_taylor(x^2 + y^2 - 1,x,y,0,1,5);
```

`INVERSE_TAYLOR`(F:*exprn*,VAR1,VAR2:*kernel*,
$\mathrm{VAR1}_0$:*exprn*, ORDER:*integer*):*exprn*

takes a function F depending on VAR1 and computes the Taylor series of the inverse of F with respect to VAR2. For example,

```
inverse_taylor(exp(x)-1,x,y,0,8);
```

When a Taylor kernel is printed, only a certain number of (non-zero) coefficients are shown. If there are more, an expression of the form `(`*n* `terms)` is printed to indicate how many non-zero terms have been suppressed. The number of terms printed is given by the value of the shared algebraic variable `TAYLORPRINTTERMS`. Allowed values are integers and the special identifier `ALL`. The latter setting specifies that all terms are to be printed. The default setting is 5.

If the switch `TAYLORKEEPORIGINAL` is set to `ON` the original expression EXP is kept for later reference. It can be recovered by means of the operator

`TAYLORORIGINAL`(EXP:*exprn*):*exprn*

An error is signalled if EXP is not a Taylor kernel or if the original expression was not kept, i.e. if `TAYLORKEEPORIGINAL` was `OFF` during expansion. The template of a Taylor kernel, i.e. the list of all variables with respect to which expansion took place together with expansion point and order can be extracted using

`TAYLORTEMPLATE`(EXP:*exprn*):*list*

This returns a list of lists with the three elements (VAR,VAR0,ORDER). As with `TAYLORORIGINAL`, an error is signalled if EXP is not a Taylor kernel.

`TAYLORTOSTANDARD`(EXP:*exprn*):*exprn*

converts all Taylor kernels in EXP into standard form and resimplifies the result.

`TAYLORSERIESP`(EXP:*exprn*):*boolean*

may be used to determine if EXP is a Taylor kernel. Note that this operator is subject to the same restrictions as, e.g., ORDP or NUMBERP, i.e. it may only be used in boolean expressions in `IF` or `LET` statements. Finally there is

`TAYLORCOMBINE`(EXP:*exprn*):*exprn*

which tries to combine all Taylor kernels found in EXP into one. Operations currently possible are:

- Addition, subtraction, multiplication, and division.
- Roots, exponentials, and logarithms.

- Trigonometric and hyperbolic functions and their inverses.

Application of unary operators like `LOG` and `ATAN` will nearly always succeed. For binary operations their arguments have to be Taylor kernels with the same template. This means that the expansion variable and the expansion point must match. Expansion order is not so important, different order usually means that one of them is truncated before doing the operation.

If `TAYLORKEEPORIGINAL` is set to `ON` and if all Taylor kernels in `exp` have their original expressions kept `TAYLORCOMBINE` will also combine these and store the result as the original expression of the resulting Taylor kernel. There is also the switch `TAYLORAUTOEXPAND` (see below).

There are a few restrictions to avoid mathematically undefined expressions: it is not possible to take the logarithm of a Taylor kernel which has no terms (i.e. is zero), or to divide by such a beast. There are some provisions made to detect singularities during expansion: poles that arise because the denominator has zeros at the expansion point are detected and properly treated, i.e. the Taylor kernel will start with a negative power. (This is accomplished by expanding numerator and denominator separately and combining the results.) Essential singularities of the known functions (see above) are handled correctly.

Differentiation of a Taylor expression is possible. If you differentiate with respect to one of the Taylor variables the order will decrease by one.

Substitution is a bit restricted: Taylor variables can only be replaced by other kernels. There is one exception to this rule: you can always substitute a Taylor variable by an expression that evaluates to a constant. Note that REDUCE will not always be able to determine that an expression is constant.

Only simple taylor kernels can be integrated. More complicated expressions that contain Taylor kernels as parts of themselves are automatically converted into a standard representation by means of the TAYLORTOSTANDARD operator. In this case a suitable warning is printed.

It is possible to revert a Taylor series of a function f, i.e., to compute the first terms of the expansion of the inverse of f from the expansion of f. This is done by the operator

`TAYLORREVERT`(EXP:*exprn*,OLDVAR:*kernel*, NEWVAR:*kernel*):*exprn*

EXP must evaluate to a Taylor kernel with OLDVAR being one of its ex-

pansion variables. Example:

```
taylor (u - u**2, u, 0, 5);
taylorrevert (ws, u, x);
```

This package introduces a number of new switches:

- If you set TAYLORAUTOCOMBINE to ON REDUCE automatically combines Taylor expressions during the simplification process. This is equivalent to applying TAYLORCOMBINE to every expression that contains Taylor kernels. Default is ON.
- TAYLORAUTOEXPAND makes Taylor expressions "contagious" in the sense that TAYLORCOMBINE tries to Taylor expand all non-Taylor subexpressions and to combine the result with the rest. Default is OFF.
- TAYLORKEEPORIGINAL, if set to ON, forces the package to keep the original expression, i.e. the expression that was Taylor expanded. All operations performed on the Taylor kernels are also applied to this expression which can be recovered using the operator TAYLORORIGINAL. Default is OFF.
- TAYLORPRINTORDER, if set to ON, causes the remainder to be printed in big-O notation. Otherwise, three dots are printed. Default is ON.
- There is also the switch VERBOSELOAD. If it is set to ON REDUCE will print some information when the Taylor package is loaded. This switch is already present in PSL systems. Default is OFF.

3 Caveats

TAYLOR should now always detect non-analytical expressions in its first argument. As an example, consider the function $xy/(x+y)$ that is not analytical in the neighborhood of $(x, y) = (0, 0)$: Trying to calculate

```
taylor(x*y/(x+y),x,0,2,y,0,2);
```

causes an error

```
***** Not a unit in argument to QUOTTAYLOR
```

Note that it is not generally possible to apply the standard REDUCE operators to a Taylor kernel. For example, PART, COEFF, or COEFFN cannot be used. Instead, the expression at hand has to be converted to standard form

first using the `TAYLORTOSTANDARD` operator.

4 Warnings and error messages

- `Branch point detected in ...`
  This occurs if you take a rational power of a Taylor kernel and raising the lowest order term of the kernel to this power yields a non analytical term (i.e. a fractional power).
- `Cannot expand further... truncation done`
  You will get this warning if you try to expand a Taylor kernel to a higher order.
- `Computation loops (recursive definition?): ...`
  Most probably the expression to be expanded contains an operator whose derivative involves the operator itself.
- `Converting Taylor kernels to standard representation`
  This warning appears if you try to integrate an expression that contains Taylor kernels.
- `Error during expansion (possible singularity)`
  The expression you are trying to expand caused an error. As far as I know this can only happen if it contains a function with a pole or an essential singularity at the expansion point. (But one can never be sure.)
- `Essential singularity in ...`
  An essential singularity was detected while applying a special function to a Taylor kernel.
- `Expansion point lies on branch cut in ...`
  The only functions with branch cuts this package knows of are (natural) logarithm, inverse circular and hyperbolic tangent and cotangent. The branch cut of the logarithm is assumed to lie on the negative real axis. Those of the arc tangent and arc cotangent functions are chosen to be compatible with this: both have essential singularities at the points $\pm i$. The branch cut of arc tangent is the straight line along the imaginary axis connecting $+1$ to -1 going through ∞ whereas that of arc cotangent goes through the origin. Consequently, the branch cut of the inverse hyperbolic tangent resp. cotangent lies on the real axis

and goes from -1 to $+1$, that of the latter across 0, the other across ∞.

The error message can currently only appear when you try to calculate the inverse tangent or cotangent of a Taylor kernel that starts with a negative degree. The case of a logarithm of a Taylor kernel whose constant term is a negative real number is not caught since it is difficult to detect this in general.

- `Invalid substitution in Taylor kernel: ...`
  You tried to substitute a variable that is already present in the Taylor kernel or on which one of the Taylor variables depend.
- `Not a unity in ...`
  This will happen if you try to divide by or take the logarithm of a Taylor series whose constant term vanishes.
- `Not implemented yet (...)`
  Sorry, but I haven't had the time to implement this feature. Tell me if you really need it, maybe I have already an improved version of the package.
- `Reversion of Taylor series not possible: ...`
  You tried to call the `TAYLORREVERT` operator with inappropriate arguments. The second half of this error message tells you why this operation is not possible.
- `Taylor kernel doesn't have an original part`
  The Taylor kernel upon which you try to use `TAYLORORIGINAL` was created with the switch `TAYLORKEEPORIGINAL` set to `OFF` and does therefore not keep the original expression.
- `Wrong number of arguments to TAYLOR`
  You try to use the operator `TAYLOR` with a wrong number of arguments.
- `Zero divisor in TAYLOREXPAND`
  A zero divisor was found while an expression was being expanded. This should not normally occur.
- `Zero divisor in Taylor substitution`
  That's exactly what the message says. As an example consider the case of a Taylor kernel containing the term `1/x` and you try to substitute `x` by `0`.
- `... invalid as kernel`
  You tried to expand with respect to an expression that is not a kernel.

- `... invalid as order of Taylor expansion`
  The order parameter you gave to `TAYLOR` is not an integer.
- `... invalid as Taylor kernel`
  You tried to apply `TAYLORORIGINAL` or `TAYLORTEMPLATE` to an expression that is not a Taylor kernel.
- `... invalid as Taylor variable`
  You tried to substitute a Taylor variable by an expression that is not a kernel.
- `... invalid as value of TaylorPrintTerms`
  You have assigned an invalid value to `TAYLORPRINTTERMS`. Allowed values are: an integer or the special identifier `ALL`.
- `TAYLOR PACKAGE (...): this can't happen ...`
  This message shows that an internal inconsistency was detected. This is not your fault, at least as long as you did not try to work with the internal data structures of REDUCE. Send input and output to me, together with the version information that is printed out.

5 Comparison to other packages

At the moment there is only one REDUCE package that I know of: the truncated power series package by Alan Barnes and Julian Padget. In my opinion there are two major differences:

- The interface. They use the domain mechanism for their power series, I decided to invent a special kind of kernel. Both approaches have advantages and disadvantages: with domain modes, it is easier to do certain things automatically, e.g., conversions.
- The concept of a truncated series. Their idea is to remember the original expression and to compute more coefficients when more of them are needed. My approach is to truncate at a certain order and forget how the unexpanded expression looked like. I think that their method is more widely usable, whereas mine is more efficient when you know in advance exactly how many terms you need.