

# REDUCE Implementation of Primitives for Univariate Skew Polynomials and Linear Ordinary Differential Operators: A progress report

Francis J. Wright  
School of Mathematical Sciences  
Queen Mary and Westfield College  
University of London  
Mile End Road, London E1 4NS, UK.  
E-mail: F.J.Wright@QMW.ac.uk

5 January 1995

## 1 Summary

**Approach:** To port Thom Mulders' Maple code, which is recent and well documented. To preserve the user-interface as closely as possible to maximize portability, although not necessarily to retain all of the internal structure – that would be too restrictive.

### Results at end 1994:

- A port of the OrePoly package is complete, including documentation and test file. The test file runs successfully, but takes about  $1\frac{1}{2}$  times longer than the Maple version.
- A port of the Lodo package is about 75% complete. For this, I have written (and tested) a new nullspace function that has a better interface to symbolic-mode code (and seems to be faster) than the standard REDUCE version.

### Maple compatibility:

**Input:** The main differences are in escape conventions and procedure definition – see the example below.

**Output:** Maple and REDUCE have different canonical forms for polynomials – in REDUCE they are sorted and in Maple they are not. There is considerable freedom in the choice of a nullspace basis.

**GNU Emacs REDUCE mode:** A first release of this is now available from the REDUCE network library (or by ftp from QMW). It provides syntax-directed editing and automatic indentation of REDUCE source code, which proved very useful for the present project.

The user interface to the REDUCE OrePoly package, and how it compares with the Maple version, are illustrated by the following segments from the test files:

Extract from Maple OrePoly test file (by Thom Mulders):

=====

```
testrightlcm:=proc(a,b,cofa,cofb)
local l,r1,r2;
  l:='OrePoly/rightlcm'(a,b,cofa,cofb);
  r1:='OrePoly/-'(1,'OrePoly/*'(a,cofa));
  r2:='OrePoly/-'(1,'OrePoly/*'(b,cofb));
  if 'OrePoly/iszero'(r1) and 'OrePoly/iszero'(r2) then
    print('rightlcm: OK')
  else
    print('rightlcm: ERROR')
  fi
end:

'OrePoly/set_X'(X):
'OrePoly/set_sigma'((f,k)->subs(x=x+k,f)):
'OrePoly/set_delta'((f,k)->
  sum('(-1)^i*binomial(k,i)*subs(x=x+k-i,f)', 'i'=0..k)):

p2:='OrePoly/polynomial'((x+1)*X^2-x*(x+3)*X-x^4):
p3:='OrePoly/polynomial'(x*X-1):
p5:='OrePoly/polynomial'(x*X^2+x*X):

p10:='OrePoly/*'(p3,p2):
p11:='OrePoly/*'(p3,p5):

testrightlcm(p10,p11,'u','v'):
```

Extract from REDUCE OrePoly test file:

=====

```
procedure testrightlcm(a,b,cofa,cofb);
begin scalar l,r1,r2;
  l := OrePoly!/rightlcm(a,b,cofa,cofb);
  r1 := OrePoly!/!(l, OrePoly!/!(a,cofa));
  r2 := OrePoly!/!(l, OrePoly!/!(b,cofb));
  if OrePoly!/iszero(r1) and OrePoly!/iszero(r2) then
    write("rightlcm: OK")
  else
    write("rightlcm: ERROR")
end$

% Need to distinguish x from X in REDUCE, so replace X by XX.

OrePoly!/set_X(XX)$

OrePoly!/set_sigma(
  procedure my_sigma(f,k); sub(x=x+k, f) );

OrePoly!/set_delta(
  procedure my_delta(f,k);
    for i := 0 : k sum (-1)^i*binomial(k,i)*sub(x=x+k-i,f) );

p2 := OrePoly!/polynomial((x+1)*XX^2-x*(x+3)*XX-x^4)$
p3 := OrePoly!/polynomial(x*XX-1)$
p5 := OrePoly!/polynomial(x*XX^2+x*XX)$

p10 := OrePoly!/!(p3,p2)$
p11 := OrePoly!/!(p3,p5)$

testrightlcm(p10,p11,u,v)$
```

## 2 Design decisions and the structure of REDUCE

REDUCE has two programming modes, both with essentially the same Algol-like syntax:

**algebraic mode:** normal user mode – Maple-like semantics;

**symbolic mode:** implementation mode – Lisp-like semantics.

Symbolic-mode code potentially benefits much more from compilation, and most of REDUCE itself consists of compiled symbolic-mode code.

It has three main data representations:

**prefix:** used mainly for I/O, e.g.

`(quotient x y);`

**standard quotient (SQ):** used for rational function arithmetic, e.g.

`((((x . 1) . 1)) ((y . 1) . 1));`

**pseudo-prefix (\*SQ):** used to pass data in algebraic mode, e.g.

`(*SQ <SQ-form>).`

One has to make a design decision about which mode and which representation to use when. A reasonable development strategy is to follow the path:

mode / representation	purpose
algebraic	fast prototyping
↓	
symbolic *SQ	intermediate testing
↓	
symbolic SQ	fast execution

For this project it was necessary to start from a hybrid of the first two levels although I have now moved to the bottom level.

By contrast, Maple has only one programming mode and one data representation.

### 3 Translation of Maple into REDUCE

This proceeded in several stages, dealing with successively more subtle differences between the two languages.

1. Many trivial syntactic differences can be translated semi-automatically. (With care this step could probably be made completely automatic.) Here are some examples:

Maple	REDUCE
#	%
:	\$
‘OrePoly/*‘	OrePoly!/*
foo := proc(a,b)	procedure foo(a,b)
local	begin scalar
print	write
ERROR	RedErr
‘string‘	"string"
elif	else if
then ... else ... fi	then <<...>> else <<...>>
from ... to	:= ... :
do ... od	do <<...>>

- The use of square brackets “[ ]” is heavily overloaded in Maple, and has the following three possible translations:

<b>operation</b>	<b>Maple</b>	<b>REDUCE</b>
list constructor	[a, b, c]	{a, b, c}
element selector	L[i]	part(L, i)
array/matrix indexing	A[i, j], A[i][j]	A(i, j)

Choosing the right translation requires either extensive parsing or user interaction; I use the latter.

- Maple is case-sensitive, whereas REDUCE is not and currently maps to lower-case. Case sensitivity is used only in the test file to distinguish `x` and `X`. The latter is easily renamed – I use `XX`.
- Maple allows variable numbers of procedure arguments. In REDUCE this is not allowed in algebraic mode and requires symbolic mode support. Since it is used a great deal in Maple, use of REDUCE symbolic mode even for the first working prototype was effectively unavoidable.
- It is standard Maple practice to use procedure arguments to return values. It is debatable whether this is good style; REDUCE moved away from this programming model some years ago (when algebraic-mode lists were introduced in version 3.3) and became more functional. Nevertheless, it is easy to support, although I have decreased the use of this technique internally.
- Arguments to user procedures are *always* evaluated in Maple. Hence Maple needs the unevaluation syntax ‘`x`’. This is neither necessary nor provided in REDUCE (algebraic mode), although symbolic-mode support is required to control argument evaluation. Unevaluated arguments are normally used for returning values. The REDUCE version of OrePoly does not evaluate arguments that are used to return values in all cases except one (`right/leftgcdex`), in which an argument may be used for input or output depending on its value.
- Maple has bizarre argument-passing semantics that effectively use textual substitution, which prevents procedure arguments from being used as local variables. Hence, it is often necessary to assign argument values to local variables in Maple, which is unnecessary in REDUCE.
- REDUCE algebraic mode does not support anonymous procedures. Hence procedure names must be used in place of anonymous procedure bodies, which is not a problem. REDUCE also provides algebraic-mode *operators*, which are quite different from procedures. They are handled by pattern matching (for which Maple is very poor) and cannot be compiled. Moreover, REDUCE functions written in symbolic mode can be implemented in several slightly different ways. Nevertheless, these internally

different facilities all serve much the same role as simple algebraic-mode procedures. They probably should be (and currently are) supported in the OrePoly package in the same transparent way that they are supported by the standard REDUCE algebraic processor. However, there may be a small execution overhead in doing this.

9. All REDUCE procedure identifiers are global, as are array and matrix identifiers. The former limitation is inherited from (Standard) Lisp and is unavoidable; it means that procedure definitions have to be copied to appropriate global identifiers in a way that can be done locally and more elegantly in Maple. However, provided matrices are manipulated in symbolic mode then there is no need to use algebraic-mode matrix (or array) identifiers and there is no difficulty making them local.
10. REDUCE does not support “abstract functions” (i.e. functions or mappings independent of arguments) and hence does not provide any analogue of the Maple function composition operators @ and @@. Only the self-composition operator @@ is used (occasionally) in the Maple OrePoly package, and it is trivial to replace these uses by recursively defined procedures. A user could alternatively use recursive rewrite (`let`) rules (although they cannot be compiled).
11. Maple provides a binomial coefficient function in its standard library. This is now provided in REDUCE as part of the special functions (`specfn`) library package, but it seems wasteful to load a large package for such a trivial function, so I have provided a separate version. This version is restricted to the case that the value is an integer, and is in fact a port of the Maple version. I wrote this when porting the Maple `randpoly` function (an enhanced version of which is available in the REDUCE network library) for another project.
12. Maple online help is a fundamental facility that is well documented and portable between implementations. Approximately half of the Maple OrePoly and Lodo source code is actually source for the help system – it is very well documented. Whilst most current REDUCE implementations provide online help, it is still not as extensive as in Maple, but in particular it is not supported by REDUCE itself but rather by the superstructure provided by the implementers, and so is not fully portable. Hence, at least as a temporary measure, I have converted the Maple help text to ordinary text in files that are separate from the REDUCE source code files, and edited them as necessary to reflect the small differences between the Maple and REDUCE user interfaces. Most aspects of the Maple documentation apply also to the REDUCE implementation, and certainly it was a design goal of the port that this should be the case.

## 4 OrePoly/Lodo development strategy

The initial translation was aimed at algebraic mode, but as explained above some use of symbolic mode was unavoidable to support variable numbers of procedure arguments. The

first working “hybrid” version consisted of the addition and multiplication functions and most of the right-division-based functions.

The Maple OrePoly representation consists of the list

```
[ coeff_list, X, sigma, delta ]
```

where `coeff_list` is a list of the OrePoly coefficients and `X` is the OrePoly variable. This variable is really a placeholder for the pair of operators `sigma` and `delta`, which are the Ore conjugation and derivation operators. These act on the OrePoly coefficients, which are rational functions. Hence, the OrePoly variable `X` is entirely characterized by `sigma` and `delta`. It has no particular significance in itself, and is required only as a convenience for input and output.

Within any one calculation, `X`, `sigma` and `delta` *must* be fixed, so that only the coefficient list varies. The coefficient list provides a dense representation of the OrePoly, in which the variable is implied. I have taken advantage of this structure and introduced a two-layer implementation. A user-layer procedure handles argument processing, extracts the coefficient list, passes it to an appropriate internal-layer procedure, and finally re-packages the result. The price of doing this is to lose the clear separation between procedures that depend on the representation and those that do not, and I have re-implemented many support routines (e.g. `OrePoly!/monomial`) at a much lower level. This *ought* to be faster. I have added user-layer code to check that arguments are OrePoly representations. The internal-layer procedures all have the symbol `@` in their names.

The layer structure and the way that variable numbers of arguments are handled is illustrated by the following code extract:

```
REDUCE OrePoly implementation: example 1
=====
External and internal representation, user-level varargs

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% OrePoly!/monomial
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
put('OrePoly!/monomial, 'psopfn, 'OrePoly!/monomial)$

symbolic procedure OrePoly!/monomial args; % (c,i,X,sigma,delta)
  %% Called with either 5 args (lcoeff, degree, X, sigma, delta)
  %% or 3 args (lcoeff, degree, type) -- type = {X, sigma, delta}
  %% or 2 args (lcoeff, degree) -- type defaults
  begin scalar c, i;
    c := simp!* car args; % poly
    if not(args := cdr args) then OrePoly!/FnErr
      "monomial requires at least 2 args (coeff, degree)";
```

```

        if not(fixp(i := reval car args) and i >= 0) then % degree
            OrePoly!/FnErr "monomial: invalid degree (second argument)";
            args := OrePoly!/get_type!@(revlis cdr args, "monomial"); % type
            return OrePoly!/make!@(OrePoly!/monomial!@(c, i), args)
        end$

symbolic procedure OrePoly!/get_type!@(args, fn);
    if args then <<
        if null cdr args then <<
            % should be a single algebraic list
            if eqcar(car args, 'list) then args := cdar args
        >>;
        if length args = 3 and idlistp args then args
        else OrePoly!/FnErr {fn, ": invalid type arguments"}
    >>
    else {!_X, !_sigma, !_delta}$

symbolic procedure OrePoly!/monomial!@(c, i);
    %% Build the normalized internal representation of  $cX^i$ 
    %% as a list of SQ forms:
    if numr c then OrePoly!/monomial!@1(c, i) else zero_OrePoly$

symbolic procedure OrePoly!/monomial!@1(c, i);
    if zerop i then {c}
    else (nil ./ 1) . OrePoly!/monomial!@1(c, sub1 i)$

```

I have eliminated much repeated code by “parametrizing” it, mainly at the user-interface level. The repeated code is mostly for argument handling, which requires more code in REDUCE than in Maple (to handle variable argument number and argument evaluation). Hence the incentive to save work translating it! For example, the internal procedure `OrePoly!/get_type!@` is called by both `OrePoly!/monomial` and `OrePoly!/polynomial` (which are closely related in my implementation), and the name of the calling routine is a parameter for use in error messages.

I have combined all right and left division-based routines, which are almost identical, except the right and left lcm routines which use different algorithms. This is illustrated by the following code extract. I avoid passing variable numbers of arguments internally by passing `nil` values for those that are missing. It is “REDUCE policy” to avoid internal functions that accept variable numbers of arguments (“fexprs”) because this facility does not compile well (as I understand it), although it is supported.

REDUCE OrePoly implementation: example 2

=====

Parametrization, arg handling, avoiding varargs internally

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% OrePoly!/rightgcd
% OrePoly!/leftgcd
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
put('OrePoly!/rightgcd, 'psopfn, 'OrePoly!/rightgcd)$
put('OrePoly!/leftgcd, 'psopfn, 'OrePoly!/leftgcd)$

symbolic procedure OrePoly!/rightgcd args;
  OrePoly!/rightleftgcd(t, args)$

symbolic procedure OrePoly!/leftgcd args;
  OrePoly!/rightleftgcd(nil, args)$

symbolic procedure OrePoly!/rightleftgcd(right, args); % (a,b,cofa,cofb)
  begin scalar a, b, type, cofa, cofb, aa, bb, g;
    a := reval car args; args := cdr args;
    b := reval car args; args := cdr args;
    type := OrePoly!/sametype_check(a,b);
    if args then
      if not idp(cofa := car args) then
        OrePoly!/RightLeftFnArgErr(right, "gcd", "third")
      else if (args := cdr args) and not idp(cofb := car args) then
        OrePoly!/RightLeftFnArgErr(right, "gcd", "fourth");
    OrePoly!/assign_sigma_delta a;
    aa := OrePoly!/p!@ a; bb := OrePoly!/p!@ b;
    if OrePoly!/iszero!@ aa and OrePoly!/iszero!@ bb then <<
      if cofa then <<
        OrePoly!/assign(cofa, a);
        if cofb then OrePoly!/assign(cofb, a)
      >>;
      return a
    >>;
  g := OrePoly!/rightleftgcdex0!@(right, aa, bb, nil, nil);
  if right then <<
    g := OrePoly!/leftmonic!@(g);
    if cofa then <<
      OrePoly!/assign!@(cofa, car OrePoly!/rightquorem!@(aa,g), type);
      if cofb then
        OrePoly!/assign!@(cofb, car OrePoly!/rightquorem!@(bb,g), type)
    >>
  >>

```

```

>> else <<
  g := OrePoly!/rightmonic!@(g);
  if cofa then <<
    OrePoly!/assign!@(cofa, car OrePoly!/leftquorem!@(aa,g), type);
    if cofb then
      OrePoly!/assign!@(cofb, car OrePoly!/leftquorem!@(bb,g), type)
    >>
  >>
>>;
return OrePoly!/make!@(g, type)
end$

```

The first complete port of the OrePoly package used an algebraic-mode list representation that mirrored that used in Maple, and used pseudo-prefix (\*SQ) representation for the OrePoly coefficients. This version ran the test file successfully, but took about 1.8 times longer than the Maple version (on the same PC under MS-Windows). In particular, the `rightlcm` test runs extremely slowly. Most of the time appears to be spent in the OrePoly multiplication used to make the result monic. This is true for the Maple version also. This suggests that improvements in the multiplication algorithm would be beneficial, if that is possible.

I have now introduced a new algebraic-mode data type for OrePolys, which is actually the same as before but with the outermost `list` tag replaced by `OrePoly`. REDUCE algebraic-mode data structures are all tagged lists, i.e. Lisp lists in which the first element is a type identifier, e.g.

**list:** (list ...)

**matrix:** (mat ...)

**OrePoly:** (OrePoly ...)

There is therefore no additional overhead in this new representation. Its advantages are that

- the undesirable automatic mapping of unary operators over the list structure is avoided (without needing to be explicitly turned off);
- type checking is easier and more reliable;
- it has allowed me to overload the standard operators `+`, `-` and `*` to apply automatically to OrePolys. REDUCE provides the flexibility to do this quite easily, and it never automatically changes operand order! I also easily could (and may yet) overload `^` (and `**`) to compute natural powers of OrePolys.

The final representation change that I made to the OrePoly package was to use standard quotient (SQ) representation internally for the coefficients rather than pseudo-prefix (\*SQ) representation. The external representation is still \*SQ. This makes the internal

representation slightly more compact and means that I can call internal arithmetic functions (`addsq`, `multsq`, etc) directly without any need for additional simplification. I feel that this representation is more elegant and more in keeping with REDUCE philosophy. In particular, the OrePoly package now follows the structure of the matrix arithmetic package. (I will return to this point in the context of the Lodo package.)

The change from `*SQ` to `SQ` internal representation appears to have increased the speed by about 10% – I had hoped for a better improvement! This leaves my REDUCE implementation still about 50% slower than the Maple version, despite a number of minor optimisations. I hope to decrease this speed difference in the future. However, REDUCE always does more work than Maple because its canonical polynomial representation is sorted, and to easily compare results it is necessary explicitly to sort the Maple results.

## 5 Progress with the Lodo package

This is built on top of the OrePoly package by defining the conjugation `sigma` to be the identity operation. Hence, the conjugation is dropped as an *explicit* component of the Lodo representation, which requires changes to all functions relating to OrePoly type information. It also requires a change to all error messages.

In the Maple implementation, all the code relating to the OrePoly type and error checking is explicitly re-written. In my REDUCE implementation, I have instead added another level of parametrization. The package name – OrePoly or Lodo – is a global parameter that is initially set to OrePoly when the OrePoly package is loaded, and is then reset to Lodo if the Lodo package is loaded on top of OrePoly. This design implies that OrePoly and Lodo function calls cannot be mixed explicitly. This could easily be allowed by making the package name local to each user-level procedure, although this would require marginally more code and be marginally slower. I may introduce this change later.

With the OrePoly package suitably parametrized, most Lodo functions can be simply (source) macros that expand to the appropriate OrePoly function. (With a local package name this would be slightly more complicated.) Hence most of the Lodo package is trivial provided OrePoly works properly. However, there remain essentially three functions that are specific to the Lodo package. One of these computes the adjoint operator and is essentially trivial. It is used to implement `Lodo!/rightlcm` in terms of `OrePoly!/leftlcm!@` rather than `OrePoly!/rightlcm!@`, presumably because the latter is slow as I remarked earlier. The other two functions compute the symmetric product and power of a Lodo, and are not trivial. I have translated all of the Lodo package, including the documentation and test file, *except* for these two functions, although I have not yet tested any of it.

## 6 Linear algebra in REDUCE

The symmetric product and power functions are similar, and are the only functions that use linear algebra. The functionality required is definition of matrices (and vectors), ac-

cess to elements, addition and multiplication, augmentation (concatenation) and nullspace computation. I expect there to be a linear algebra library package in the next release of REDUCE, but fortunately all the functionality required for this project is provided by the standard REDUCE matrix package (although it supports augmentation only internally).

However, matrix support in REDUCE is currently rather inconsistent. It appears to be intended almost entirely to support matrices in algebraic mode, and not to support efficient use from symbolic mode. Conventional matrix-element access syntax is supported only in algebraic mode for global matrix identifiers, and there is no other explicit support provided at all. The internal canonical matrix representation is a list of lists of SQ forms, which is one reason why I changed the internal OrePoly representation to a list of SQ forms. This is the form expected by the matrix addition and multiplication functions. It is clearly the best form for any algebraic operations, and therefore I intend to use it. Random access of the REDUCE list representation of matrices is inefficient, although easy enough to implement, and suitable code can easily be extracted from the algebraic-mode access functions.

The only potential difficulty is with the nullspace code, which does not seem to me to be suitably structured for the kind of symbolic-mode use that I want to make of it. It uses pseudo-prefix (\*SQ) form for input and output, and hence would require data conversion. The nullspace function is called in the main loops of the symmetric product and power functions together with other matrix arithmetic, and hence this data conversion could introduce a substantial overhead. Moreover, the code struck me as more complicated than necessary. I therefore decided to write my own nullspace function, designed to:

- use matrix canonical form for both input and output;
- use only SQ form;
- minimize random access.

The code consists of two (obvious) stages:

1. reduction to echelon form in which redundant zero rows are discarded. This uses Gaussian elimination that never introduces new denominators or unnecessary multipliers;
2. nullspace computation by modified back-substitution that introduces arbitrary constants as necessary when computing a basis vector element that does not correspond to a distinguished element in the echelon form. It does essentially no random access by first constructing a (reversed) access list of indices and pointers to the distinguished elements. It does not use any explicit indeterminates at all (unlike the standard REDUCE version).

I added some interface code to facilitate testing it in algebraic mode. The result appears to be faster than the standard REDUCE nullspace operator, increasingly so as the complexity increases. For example, on a  $5 \times 7$  integer matrix with rank 2 (nullity 5) it is

about 4 times faster. However, it produces a different basis, but I currently see no reason to prefer one basis over the other, and because it is used in a loop I consider that speed is more important. I have not tested it against the Maple nullspace function.

The echelon code could possibly be further optimised by using some destructive list operations (`rplaca/d`). I may also explore the possibility of using essentially the same code to solve linear equations, which also *might* be faster than the standard facilities.

It requires a little more work to finish porting and to test the Lodo package.

## 7 Possible future developments

It might be possible and worthwhile to link these packages with the REDUCE `ncpoly` package, which provides support for Gröbner bases and factorization of non-commuting polynomials. It may be useful to feed some of my modifications back to the Maple implementations of OrePoly and Lodo.

## Acknowledgements

I thank Malcolm MacCallum and Manuel Bronstein for suggesting this project, and Thom Mulders for providing an excellent starting point.