MPCR: An Efficient and Flexible Chains of Recurrences Server

Olaf Bachmann*
Department of Mathematics and Computer Science
Kent State University
Kent, OH - 44242, U.S.A.
obachman@mcs.kent.edu

July 26, 1996

Abstract

MPCR is a stand-alone, flexible, and efficient implementation of the Chains of Recurrences (CR) method. It can be used interactively, as a network-based server, or as a library. Using the CR method, MPCR evaluates closed-form expressions over regular grids with great efficiency – often hundreds of times faster than comparable programs. MPCR also features an MP (Multi Protocol) interface for data exchange, conditional CR simplification, use of intermediate evaluation arrays and generation of C source code. Evaluation timings are given and compared with other programs, and the algorithms which result in the high efficiency of MPCR are discussed.

1 Introduction

Chains of Recurrences (CRs) [3] can be used to greatly speed up the evaluation of closed-form elementary functions over regular grids. Consider evaluating an expression at equally spaced points. Instead of evaluating at each point afresh, the value at the next point can often be computed easily from values at previous points. The CR method specifies an algorithm to transform closed-form formulas into recurrence relations which reuse values from previous points and are suitable for fast evaluations. CR based executions use far fewer arithmetic operations and can usually take advantage of pipelining in modern CPU architectures. For example, a CR evaluation of a degree-n polynomial uses n

*Work reported herein has been supported in part by the National Science Foundation under Grant CCR-9503650.

additions in a tight loop for each evaluation point. It can be several hundred times faster than conventional evaluation.

The CR method has many practical applications including visualization of mathematical curves and surfaces, numeric computations of integrals, and automatic generation of optimized source code.

The CR method had been implemented on top of computer algebra systems (see e.g. [3], [4], [7]). This approach is all right for prototyping but has serious drawbacks for production systems. Among the problems are slow array and floating point operations, lack of portability, and difficulty to function as a component or server for other independent programs. MPCR, on the other hand, is a stand-alone implementation of the CR method written in C/C++. MPCR can be used as a library, an interactive program or a network-based server. For demonstrations of MPCR, access SymbolicNet (`http://SymbolicNet.mcs.kent.edu/`). For software distribution by public FTP, use `ftp.mcs.kent.edu` under `/pub/mpcr`.

Features, functionalities, and implementation techniques of MPCR will be described. Details of the CR method can be found in [2] or [3]. We begin with a top-level description of MPCR. The various features will then be presented in the following sections.

2 MPCR Overview

As mentioned, MPCR can be used interactively, as a network-based server, or as a library. Given closed-form formulas, MPCR applies the CR method to produce values at perscribed grid points or generates a C program that does the same. Input to MPCR consists of three parts: A command, a list of expressions, and evaluation range specifications. The main commands are `CREval` and `CRCodeGen`. `CREval` causes the server to compute point values whereas `CRCodeGen` tells the server to generate C codes. Valid closed-form input expressions are real elementary expressions (i.e. expressions composed of constants, variables, and elementary function symbols like `+`,`-`,`*`,`/`,`^`,`log`,`exp`,`sin`,`acos`,`tanh`, `acsch`, etc) in one or two evaluation variables.

To transfer the generated points efficiently to another compute engine which may plot or otherwise use the data, the server adopts the Multi Protocol (MP) [6] as a data exchange mechanism. Basically, MP uses binary encoded, annotated and linarized expression trees to transfer data. The name MPCR reflects the fact that the server has an MP interface. With this interface, MPCR can be used as a network-based compute server which uses MP for its I/O encoding and listens at a specified TCP/IP port for computation requests from other programs. Due to the binary format of the MP encoding, this mode provides very efficient means for fast communication of computation requests and results. For example, interactive visualization servers like IZIC [5] may very conveniently and efficiently use MPCR as a server through its MP interface.

For applications which do not have an MP interface, MPCR can also be

operated via an interactive ASCII interface or via command line arguments. In both cases, commands, expressions, and evaluation ranges are specified using a conventional ASCII infix notation. Output may be sent to files, pipes, or stdout in ASCII or binary format.

Finally, MPCR can also be used as a library which is linked to another program, thereby largely avoiding any communication overhead. In this mode, the linking program has to provide the "input" either via strings, or has to supply a "translation table" such that expressions can be transformed into MPCR's internal representation. Furthermore, "communications" are accomplished by C++ class member/access functions. The forthcoming version of the Macintosh Graphing Calculator [1] uses the linked library version of MPCR for its function evaluations.

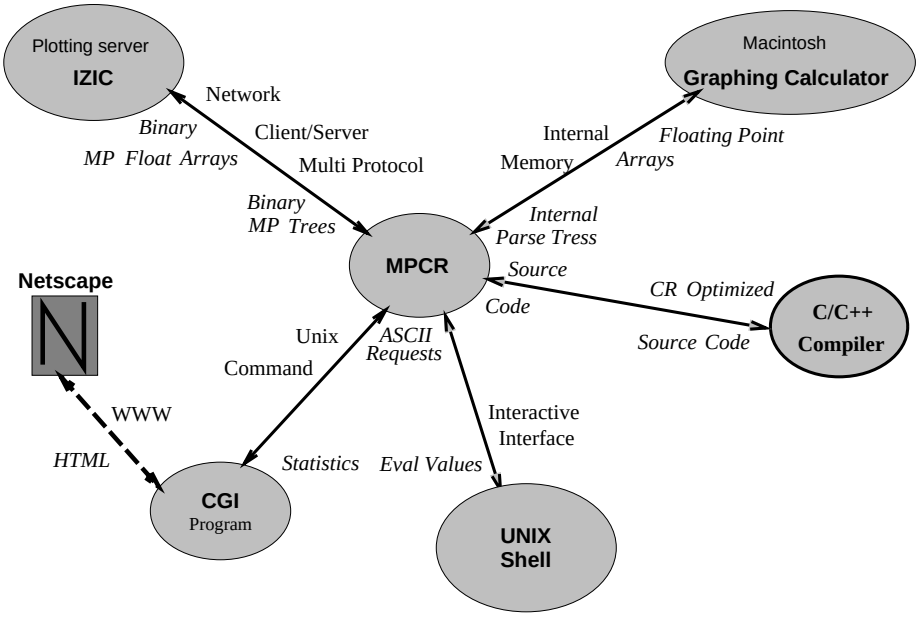

Figure 1: Scenarios for using MPCR

MPCR uses `double` floating point numbers as its default computational domain. However, this might be changed at compile-time into `float`'s or `long double`'s. Also, many of the other aforementioned features can be enabled or disabled at compile time, making it possible to configure MPCR into an application-specific, customized tool. See also figure 1, which further outlines different scenarios and modes of MPCR.

So far, MPCR has been ported to most UNIX architectures and to the Mac-

intosh running MacOS version 7.5. We anticipate that a port to an Intel-based architecture running a Windows operating system would impose little or no difficulty.

3 Efficiency of MPCR

One of the primary features of MPCR is its high evaluation speed. To illustrate the efficiency of MPCR and to compare it with the speed of other evaluation programs, let us look at some timings.

Table 3 shows various CPU timings in seconds obtained on a SparcStationII with 32MB of main memory running SunOS 4.1. The timings were obtained using the following evaluation programs:

`Maple`: Maple V.3 running a procedure which was taken from the evaluation routine used inside Maple's `plot` command.

`NormalEval`: An implementation of the conventional evaluation method within MPCR (which, for example, is equivalent to the implementation of the evaluation routine used inside the first version of the Macintosh Graphing Calculator [1]).

`CRStepEval`: An implementation of the CR method within MPCR which uses the "step evaluation" algorithm (see section 5.2).

`CREval`: The standard way of CR evaluations of MPCR (uses the "array evaluation" algorithm described in section 5.2).

`CompiledNormalEval` and `CompiledCREval`: The source code generated by MPCR for conventional and CR evaluations was compiled and executed. In other words, the results were computed by executing compiled expressions, instead of by interpreting the equivalent expression trees.

All timings include CR construction times (if applicable), but exclude I/O times and code-generation/compilation times (which, where applicable, were generally in the range of 4 seconds). All evaluations of two (resp. one) dimensional expressions were done using 100x100 (resp. 10,000) evaluation points with $x_0 = y_0 = 1, h_x = h_y = .01$. The following expressions were used:

1. $x(x(x(x - \frac{1}{2}) + 3) - \frac{3}{5}) + 5$: Because of its input representation, this example might be used to compare the CR evaluation with the Horner evaluation method of polynomials.

2. The expanded polynomial $(3y^2 - \frac{xy^2}{2} + \frac{3}{5}x + \frac{4}{3})^7$ containing 85 monomials.

3. The unexpanded polynomial $(3y^2 - \frac{xy^2}{2} + \frac{3}{5}x + \frac{4}{3})^7$.

4. $[\frac{u\cos v}{2} - \frac{u^3\cos 3v}{6}, \frac{u\sin v}{2} - \frac{u^3\sin 3v}{6}, \frac{u^2\cos 2v}{2}]$: An Enneper surface.

5. $1.3^{1.2x-1}\cos(1.5x)\sin(1.5y)$: A slightly changed version from an example given in Maple's `animate3d` package.

6. $\log x + \sqrt{x}$: For this example, the CR method does not result in computations which use less arithmetic operations. Therefore, these timings are interesting for comparing the overheads of the various evaluation programs.

`EvalMode` / Example	1	2	3	4	5	6
`Maple`	3.72	125.01	4.78	19.32	12.87	3.27
`NormalEval`	0.52	76.88	1.43	3.73	1.38	0.27
`CRStepEval`	0.06	0.30	0.18	0.85	0.16	0.27
`CREval`	<0.01	0.12	0.07	0.07	<0.01	0.15
`CompiledNormalEval`	0.02	42.4	0.47	1.08	0.51	0.13
`CompiledCREval`	<0.01	0.07	0.02	0.05	<0.01	0.13

Table 1: Timings in seconds for various evaluations modes and examples

The timings illustrate the impressive overall speed of MPCR and show that `CREval` (MPCR's standard evaluation routine) can be hundreds of times faster than comparable programs. It should also be mentioned that CR evaluations usually result in bigger numeric inaccuracies than conventional evaluations. However, these errors are generally not too large (e.g. the maximum precision loss in the results of the examples above are 3 of the 16 significant digits of a `double` floating point number) and should be tolerable for most applications. In [2], rigorous bounds on these errors are given and CR evaluation techniques are described which assure a user-specified numeric accuracy and, at the same time, do not significantly slow down the evaluations.

4 Implementation of MPCR

The first version of MPCR was written in the summer of 1995 as part of a project implementing a successor of the Macintosh Graphing Calculator. This version included most of the kernel routines and the Macintosh library version of MPCR. Major additions of the current version of MPCR are a UNIX port, the MP interface and the code-generation feature. The current version is 1.0 and consists of approximately 15,000 lines of C/C++ code.

The major concerns in the design and implementation of MPCR were efficiency, portability and flexibility. As a consequence, the implementation of MPCR

- includes its own memory management in order to exploit the regular memory requirement patterns and in order to be independent of some system-provided memory manager
- exploits common subexpressions in evaluations of CR expressions (i.e. after CR expressions are constructed, they are searched for common subexpressions which are then used to increase the evaluation efficiency)
- does not use recursive procedures (e.g. tree traversals are implemented using the equivalent of iteratively transformed tail recursions)
- may be run in a cooperative multi-tasking mode (i.e. after a certain amount of elapsed time, the state of the computation is internally saved and control is given to the operating system)[1]

5 Algorithms of MPCR

The two major algorithms underlying MPCR are those for constructing and evaluating CR expressions. MPCR incorporates new and important improvements to both algorithms: the conditional simplification and array evaluation technique. In fact, large parts of MPCR's efficiency are due to these novel techniques. Therefore, they are presented in some more detail in the next two subsection thereby assuming a familiarity of the reader with the main concepts of the CR method as described in [2] or [3].

5.1 Conditional simplification of CR expressions

Given a two-dimensional expression $F(x, y)$, let us briefly recall the basic algorithm used for constructing a two-dimensional CR expression Φ, such that $\Phi(i, j) = F(x_0 + i\,h_x, y_0 + j\,h_y)$:

(1) On the parse tree of F, replace each occurrence of the variables x and y by the primitive CRs $\{x_0, +, h_x\}_x$ and $\{y_0, +, h_y\}_y$, respectively.

(2) From the bottom to the top, recursively simplify each inner node using the simplification rules of CR expressions given in [2], [3].

By also recalling that we defined the Cost Index $CI(\Phi)$ as a measure of the evaluation cost of Φ, the idea behind this construction algorithm is that after step (1), $CI(\Phi) = CI(F)$ and that $CI(\Phi)$ is reduced by applying the simplification rules during step (2). One problem with this algorithm is that there are simplification rules which may or may not lead to a reduction of $CI(\Phi)$, depending on the "surrounding context" of the node and on the values of n_x, n_y.

[1] The cooperative multi-tasking and no-recursion features were mainly motivated by the requirements of the Macintosh Graphing Calculator [1].

Let us illustrate these problems by two examples. First, let $\Phi = \{\varphi_0, +, \varphi_1\}_x$, $l > 0, l \in \mathbf{Z}$ and consider Φ^l. By expanding Φ^l, we would obtain a polynomial CR, say, Φ_e, of length l, i.e. $CI(\Phi_e) = n_x l$. On the other hand, by not applying this simplification, $CI(\Phi^l) = n_x(\lceil \log l \rceil + 1)$.[2] Hence, at this point, the cost index of the unsimplified expression is less then the cost index of the simplified expression. However, what if Φ^l is just a subexpression of, say, the expression $\Psi = 3\Phi^l + \{\psi_0, +, \psi_1\}^l$. Then after expanding simplification we still have $CI(\Psi_e) = n_x l$ and by not expanding we have $CI(\Psi) = n_x(2\lceil \log l \rceil + 4)$. (which, depending on l, may or may not be smaller than $n_x l$). Hence, the decision on whether or not to expand expressions like Φ^l should not be determined "locally", but should take the "surrounding context" into consideration.

Secondly, consider the multiplication of one-dimensional polynomial CRs of different ranks. For example, let $\Phi = \{\varphi_0, +, \varphi_1\}_x, \Psi = \{\psi_0, +, \psi_1, +, \psi_2\}_y$. Then we would have three different possibilities of simplifying $\Phi * \Psi$

1. Do not simplify $\Phi * \Psi$ at all. In this case, $CI(\Phi * \Psi) = n_x n_y + 2n_x + 3n_y$.
2. Simplify $\Phi * \Psi$ into a CR of rank $x < y$, i.e. into $\{\{\psi_0\varphi_0, +, \psi_0\varphi_1\}_x, +, \{\psi_1\varphi_0, +, \psi_1\varphi_1\}_x, +, \{\psi_2\varphi_0, +, \psi_2\varphi_1\}_x\}_{xy}$. In this case, $CI(\Phi * \Psi) = 2n_x n_y + 3n_x$
3. Simplify $\Phi * \Psi$ into a CR of rank $y < x$, i.e. into $\{\{\varphi_0\psi_0, +, \varphi_0\psi_1, +, \varphi_0\psi_2\}_y, +, \{\varphi_1\psi_0, +, \varphi_1\psi_1, +, \varphi_1\psi_2\}_y\}_{yx}$. In this case, $CI(\Phi * \Psi) = n_x n_y + 6n_y$.

As we can see from this example, the simplification which yields the minimum cost index may not only depend on the surrounding context, but also on the actual values of n_x and n_y. It is also easy to see that an algorithm which always constructs a CR expression with a minimum cost index would have to pursue each possible simplification path which involves extensive backtracking. Unfortunately, this might be rather time-expensive. Therefore, such a backtracking simplification approach should only be used for applications where the construction time is of little or no significance (e.g. for code-optimization applications).

For applications where the construction time counts towards the overall evaluation time, we suggest to use a heuristic approach which balances the construction and evaluation cost. To accomplish this, we first classify all simplification rules into three categories:

Unconditional rules Simplification rules which always lead to a reduction of the cost index. All simplifications involving constants fall into this category, as well as the addition of polynomial CRs of the same rank.

Local rules Simplification rules for which we decide "locally" whether to apply them or not. All simplifications involving trigonometric functions fall into this category.

[2] By using a binary exponentiation method to compute Φ^l and by assuming that multiplication has the same cost as addition.

Conditional rules Simplification rules for which we decide on the basis of the cost index for the effected subtrees whether to apply them or not. The simplification rules from the examples above fall into this category.

Based on this classification, we advise changing step 2 of the CR construction algorithm as follows:

(2.1) Find independent subtrees $\Phi_1, \ldots, \Phi_k$ such that each Φ_j contains at least one node which could be simplified with a conditional simplification rule and such that an application of this rule would only effect the subtree Φ_j.

(2.2) For each subtree Φ_j, determine the cost index $\begin{bmatrix} c_n \\ c_{xy} \\ c_{yx} \end{bmatrix}$ of Φ_j if $\begin{bmatrix} \text{no} \\ \text{all} \\ \text{all} \end{bmatrix}$ conditional rules are applied using rank $\begin{bmatrix} \text{n/a} \\ \text{x<y} \\ \text{y<x} \end{bmatrix}$ and mark all nodes nodes with conditional simplification rules according to $\min\{c_n, c_{xy}, c_{yx}\}$.

(2.3) Simplify Φ w.r.t. the marks and categorizations of simplification rules.

Notice that the algorithm might produce a CR expression which has subexpressions of different ranks. While this results in a decrease of the cost index it imposes a serious problem on conventional evaluation methods. Therefore, this feature might either have to be disabled for certain evaluation methods (see also section 5.2 for more details).

What remains to be discussed is how the cost indices c_n, c_{xy}, c_{yx} in step (2.2) can be determined. c_n can be obtained straightforwardly by simply counting operations, and so could c_{xy}, c_{yx}, after the respective simplification rules had been applied. However, this might be a rather expensive procedure by itself, since an application of these rules might involve extensive "CR arithmetic" (which is at least as expensive as polynomial arithmetic). Instead, we can obtain good estimates of c_{xy}, c_{yx} without having to apply any of the simplifcation rules based on the following lemma:

Lemma 1 *Let $p(x,y)$ be a polynomial, $C_{i,j}$ be the coefficient of the monomial $x^i y^j$ of p,*

$$\begin{array}{rclrcl} d_x &=& \max\{i : C_{i,j} \neq 0, j \geq 0\} & d_y &=& \max\{j : C_{i,j} \neq 0, i \geq 0\} \\ d_{x,yx} &=& \max\{i : C_{i,j} \neq 0, j > 0\} & d_{y,xy} &=& \max\{j : C_{i,j} \neq 0, i > 0\} \\ d_{x,xy} &=& \max\{i : C_{i,j} \neq 0, j = d_{y,xy}\} & d_{y,yx} &=& \max\{j : C_{i,j} \neq 0, i = d_{x,yx}\} \end{array}$$

and c_{xy} (resp. c_{yx}) be the cost index of the (fully expanded) polynomial CR of rank x, y (resp. y, x) when evaluated over $n_x * n_y$ (resp. $n_y * n_x$) points. Then,

$$d_{x,xy} \leq \frac{c_{xy} - n_x n_y \min\{d_{y,xy}+1, d_y\} - n_y \max\{0, d_y - d_{y,xy} - 1\} - n_x d_x}{n_x \max\{0, d_{y,xy} - 1\}} \leq d_{x,yx}$$

$$d_{y,yx} \leq \frac{c_{yx} - n_y n_x \min\{d_{x,yx}+1, d_x\} - n_x \max\{0, d_x - d_{x,yx} - 1\} - n_y d_y}{n_y \max\{0, d_{x,yx} - 1\}} \leq d_{y,xy}$$

For example, if $p(x,y) = x^5 + x^3y^2 + xy^4$ then $d_x = 5, d_y = 4, d_{x,yx} = 3, d_{y,xy} = 4, d_{x,xy} = 1, d_{y,yx} = 2$ and $3n_x \leq c_{xy} - 4n_xn_y - 5nx \leq 12n_x$, $4n_y \leq c_{yx} - 3n_xn_y - n_x - 4n_y \leq 8n_y$.

In other words, lemma 1 reduces the problem of finding the cost indicies c_{xy}, c_{yx} to determening the degree values $d_x, \ldots, d_{y,yx}$. If a polynomial or polynomial CR were in an expanded representation, then all degree values could be read directly. However, we do not need to expand a polynomial in order to obtain its degree values. Instead, we can determine these degree values based on the following observation: If $p(x,y) = x$ (resp. y), then $d_x = 1$ and $d_y = \ldots = d_{y,yx} = 0$ (resp. $d_y = 1$ and $d_x = \ldots = d_{y,yx} = 0$). Furthermore, it is tedious but straightforward to determine the degree values of the sum, product, and natural power of polynomials, given the degree values of the argument polynomial(s). For example, $d_x(p+q) = \max\{d_x(p), d_x(q)\}, d_x(p*q) = d_x(p) + d_x(q), d_x(p^n) = n * d_x(p)$, etc.

As a summary, the conditional simplification heuristic avoids any additional CR manipulations and backtracking, is not very costly to apply and yields "close-to-optimal" results for many common cases. For example, the CR construction times for all examples of section 3 were smaller than 0.01 seconds, except for the 85 monomial example 2, which had a construction time of 0.05 seconds, and the constructed CR expressions were all optimal, except for example 3[3] Of course, further refinements of the conditional simplification heuristic are possible, but major improvements would most likely have to be paid for by more complicated construction algortihms and/or longer construction times. Furthermore, we should mention that the main ideas of this heuristic can easily be extended to CR expressions in more than two variables. The only problem might be that the formalisms needed to describe these ideas for the n-dimensional case are considerably more complicated (and, admittedly less comprehensible).

5.2 Array evaluation of CR expressions

Before describing the array evaluation technique, let us first outline what we call "step evaluation", which is basically equivalent to conventional evaluation techniques.[4]

StepEval(F, x_0, h, n)
for $i = 1$; $\mathrm{v}_x = x_0$ **to** n **do** v[i] = Eval(F, v_x); $v_x = v_x + h$ **od**
end

[3]The optimal CR-expression for this example is a CR of the form Φ^7 with Φ being a polynomial CR of rank $y < x$. The heuristic CR simplification yields a CR expression of the form $(\Phi_1 - \Phi_2 + \Phi_3)^7$ with Φ_1, Φ_2, Φ_3 being CR expressions corresponding to the inner terms of the polynomial.

[4]By evaluation we mean here evaluation by interpreting expression trees, which should be distinguished from evaluation by executing compiled expressions.

```
Eval(F, v_x)
    if F is a constant then  v = F
    elif F == x then v = v_x
    elif NumberOfArgs(F) == 1 then
            v = Eval(FirstArg(F), v_x); op = Operator(F);
            if op == sin then v = sin(v)
            elif op == cos then v = cos(v)
            elif /* handle all operators with one argument */ fi
    else /* similarly, handle cases with NumberOfArgs(F) > 1 */ fi
    return v
end
```

The main idea behind the array evaluation technique is to exchange the outer loop with the inner evaluation, i.e. to "push" the outer loop into the evaluation routine and to work on arrays of values, instead of single values. The basic algorithm can be outlined as follows:

```
ArrayEval(F, x_0, h, n)
    if F is a constant then  for i = 1; to n do v[i] = F od
    elif F == x then for i = 1; v[0] = x_0 to n do v[i] = v[i-1]+h od
    elif NumberOfArgs(F) == 1 then
            v = ArrayEval(FirstArg(F)); op = Operator(F);
            if op == sin then for i = 1 to n do v[i] = sin(v[i]) od
            elif op == cos then for i = 1 to n do v[i] = cos(v[i]) od
            elif/* handle all operators with one argument */ fi
    else /* similarly, handle cases with NumberOfArgs(F) > 1 */ fi
    return v
end
```

With some care, this basic algorithm can be extended to two- and n-dimensional expressions, and modified such that for subexpressions of F whose dimension is less than that of F, only arrays of the respective sub-dimension are used, instead of arrays of the full dimension of F.[5]

Although the array evaluation technique does not decrease the cost index of the evaluated expression, the timings of section 3 impressively illustrate the efficiency gained in comparison with the step evaluation technique. The main reasons for the efficiency gains are (i) array evaluations have considerable less parsing overhead compared with step evaluations; (ii) the inner loops of array evaluations can effectively be optimized by the compiler (especially for pipelined RISC CPUs).

[5] For example, if F contains a constant, then the value of this constant is returned, instead of an array whose elements all have the value of this constant. Or, if, $F_x(x)$ is a subexpression of $F(x, y)$, then an array containing the n_x values $F_x(x_0 + ih_x)$ is returned, instead of an array of dimension $n_x * n_y$.

Using array evaluations for CR expressions offers further efficiency advantages: First, the array evaluation of the various types of CRs (i.e. polynomial, exponential, and complex) may all be hand-coded so that only a few local variables (for CRs of small lengths) or an array containing the CR coefficients (for CRs of bigger lengths) are manipulated in order to obtain the consecutive values of a CR. Secondly, by using a proper indexing scheme, array evaluations can easily evaluate CR expressions which contain subexpressions of different ranks. On the other hand, this imposes a much more serious problem for step evaluations and solutions would basically require to mix up step and array evaluations locally.

But, of course, there is also an obvious down side to the array evaluation technique. The gains in efficiency have are paid for with (possibly much) more memory. Notice, however, that the memory usage can often be greatly reduced by transforming the (CR) expression trees into balanced binary trees thereby "grouping" together subexpressions of the same rank as much as possible and by judicious implementations which reuse evaluation arrays as much as possible. Nevertheless, in the worst case, we may need Depth(Φ) full evaluation arrays (where Φ is a balanced, binary (CR) expression) which, depending on various factors (e.g an operating system not providing virtual memory), may be problematic. Therefore, in addition to array evaluations, we recommend having the step evaluation technique implemented as a "back up" for use when memory is low.

As a summary, our experience shows that the array evaluation technique is very effective and comparable to compiled expression evaluations (see also the `CREval` and `CompileCREval` timings given in section 3) and that for many applications the price (more memory, more complex to implement) is well worth paying.

6 Summary and Future Work

MPCR is a stand-alone, flexible, and efficient implementation of the CR method. Its flexibility w.r.t. different configurations and I/O formats make it easily usable by other programs. It provides an unsurpassed efficiency for the dynamic evaluation of closed-form functions over regular grids and pushes the speed of such computations into new dimensions. This is especially suitable for the visualization of mathematical curves and surfaces. In addition to evaluating functions, MPCR can also generate C source code for CR evaluations, which makes it suitable to be used as a code optimization tool.

As with many other software packages, the TODO and wish list of MPCR is long. Most notably, we would like to implement the techniques for meeting strict numeric error bounds described in [2], as well as extensions to input expressions which may contain additional variables and extensions for the generation of Fortran or Lisp source code.

7 Availability

The newest binary version (currently 1.0) of MPCR is available for most UNIX architectures by anonymous ftp from `ftp.mcs.kent.edu/pub/mpcr`. The distribution contains various precompiled configurations of MPCR, as well as a manual page detailing the options, modes, and I/O formats. For further details, see the `README` file in the `/pub/MPCR` directory.

Furthermore, there is an interactive introduction to the CR method and a WWW interface to MPCR available at
`http://SymbolicNet.mcs.kent.edu/areas/cr/`. Using this WWW interface, all timings given in this paper can be reproduced, the generated source code for the CR evaluations can be inspected, evaluation points can be downloaded and plotted, and the numeric errors of the CR evaluations can be examined.

8 Acknowledgements

I would like to thank my supervisor, Paul S. Wang, for his invaluable help and suggestions for the MPCR work and for preparing this paper. I furthermore would like to thank Ron Avitzur for the support of and contributibtion to the implementation of MPCR, and Simon Gray, Norbert Kajler and Paul S. Wang for the design and implementaiton of MP.

References

[1] Ron Avitzur. Direct Manipulation in a Mathematics User Interface In Norbert Kajler, editor, *Human Interaction for Symbolic Computation*. Springer-Verlag, 1996.

[2] Olaf Bachmann. *Chains of Recurrences*. PhD thesis, Kent State University, Kent, OH - 44240, U.S.A., 1996.

[3] Olaf Bachmann. Chains of Recurrences for Functions of Two Variables and their Application to Surface Plotting. In Norbert Kajler, editor, *Human Interaction for Symbolic Computation*. Springer-Verlag, 1996.

[4] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of Recurrences - a method to expedite the evaluation of closed-form functions. In *Proceedings of the International Symposium on Symbolic an Algebraic Computation - ISSAC'94*, pages 242–249, Oxford, England, United Kingdom, July 1994. ACM Press.

[5] Robert Fournier, Norbert Kajler, and Bernard Mourrain. IZIC: a Portable Language-Driven Tool for Mathematical Surfaces Visualization. In A. Miola, editor, *Proc. of DISCO'93*, pages 341–353, Gmunden, Austria, September 1993. LNCS 722, Springer-Verlag.

[6] S. Gray, N. Kajler, and P. S. Wang. Design and Implementation of MP, a Protocol for Efficient Exchange of Mathematical Expressions. *Journal of Symbolic Computing*, To appear in 1996.

[7] E.V Zima. Simplification and optimization transformations of chains of recurrences. In A. H. M. Levelt, editor, *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'95), Montreal, Canada*, pages 42–50. ACM Press, 1995.