Binding-time analysis applied to mathematical algorithms

*Robert Glück**, *Ryo Nakashige**, *Robert Zöchling*[◇]
**DIKU, Dept. of Computer Science, University of Copenhagen*
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark.
◇*Inst. für Computersprachen, Vienna University of Technology,*
Argentinierstr. 8, A-1040 Vienna, Austria.
e-mail: {glueck,ryon}@diku.dk, e1802gab@vm.univie.ac.at

Abstract
Our goal is to incorporate state-of-the-art partial evaluation in a library of general-purpose algorithms – in particular, mathematical algorithms – in order to allow the automatic creation of efficient, special-purpose programs. The main goal is efficiency: a specialized program often runs significantly faster than its generic version.

This paper shows how a binding-time analysis can be used to identify potential sources for specialization in mathematical algorithms. The method is surprisingly simple and effective. To demonstrate the effectiveness of this approach we used an automatic partial evaluator for Fortran that we developed. Results for five well-known algorithms show that some remarkable speedup factors can be obtained on a uniprocessor architecture.

Keywords
Scientific computing, numerical algorithms, partial evaluation, binding-time analysis.

1 INTRODUCTION

The application of partial evaluation to mathematical algorithms seems especially promising for several reasons. A large body of general-purpose algorithms is available (*e.g.* the NAG library contains more than 1000 mathematical algorithms). Their motivation clearly comes from the practical world of scientific computing, which has been dictating their development over a long time. High performance of mathematical algorithms is a key issue in most scientific and engineering applications.

Our goal is to incorporate state-of-the-art partial evaluation in a library of general-purpose mathematical algorithms in order to allow the automatic generation of fast, special-purpose programs from generic algorithms. This work is an attempt to capitalize on partial evaluation's ability to identify and extract static computations automatically from mathematical algorithms (Berlin &Weise, 1990; Baier et al., 1994; Andersen, 1995). Early examples of specializing numerical algorithms are provided by (Gustavson et al., 1970) and (Goad, 1982). Interprocedural constant propagation was applied to scientific applications in (Metzger & Stroud, 1993). They do not associate themselves with the partial evaluation paradigm, however.

We demonstrate how a binding-time analysis can be used to identify sources for specialization in general-purpose mathematical algorithms. To demonstrate the effectiveness of this approach we used an automatic partial evaluator for a subset of Fortran 77 which we developed (Kleinrubatscher et al., 1995). Our results show that this approach is strong enough to improve the efficiency of a certain class of mathematical problems.

2 PARTIAL EVALUATION AND BINDING-TIME ANALYSIS

Program Specialization. Assume that P is a general program with two arguments and that its first argument x is known (*static*) while its second argument y is unknown (*dynamic*). A program specializer produces a specialized program P_x that returns the same result when applied to the remaining input y as the original program P when applied to the input x and y, but potentially much faster.

Partial Evaluation is an automatic method for program specialization. In *offline* partial evaluation the transformation process is guided by a *binding-time analysis* performed prior to the specialization phase (Jones et al., 1993). The result of the binding-time analysis is a program in which all expressions are annotated as either static or dynamic. Operations annotated as static are performed at specialization time, while operations annotated as dynamic are delayed until run time (*i.e.* residual code is generated). Partial evaluation differs from ordinary optimizing compilers since it takes the *static input* of programs into account. Optimizing compilers lack binding-time information, thus it is unreasonable to expect a compiler to execute static statements and generate specialized programs.

Binding-TimeAnalysis. The analysis computes a division B of all variables X in a program P given an initial classification of the input variables as either static or dynamic. Variables classified as static depend only on static input variables. Variables classified as dynamic may depend on dynamic input variables.

Algorithm. (Monovariant Binding-Time Analysis) *Call the program variables $X_1, \ldots, X_N$ and assume that the input variables are $X_1, \ldots, X_n$, where $1 \leq n \leq N$. Assume that the binding-times $\overline{b}_1, \ldots, \overline{b}_n$ for the input variables are given, where $\overline{b}_i$ is either* S *(static) or* D *(dynamic). The task is to compute a congruent division for all program variables: $B = (b_1, \ldots, b_N)$ which satisfies $\overline{b}_i = \mathrm{D} \Rightarrow b_i = \mathrm{D}$ for the input variables. The analysis is done by the following algorithm:*

1. *Construct the initial division $\overline{B} = (\overline{b}_1, \ldots, \overline{b}_n, \mathrm{S}, \ldots, \mathrm{S})$ and set $B = \overline{B}$.*
2. *If the program contains an assignment $X_k \leftarrow \mathrm{exp}$ where the variable X_j appears in* exp *and $b_j = \mathrm{D}$ then set $b_k = \mathrm{D}$ in B.*
3. *Repeat step 2 until B does not change any longer. Then the algorithm terminates with congruent division B.*

3 PARTIAL EVALUATION OF MATHEMATICAL ALGORITHMS

The algorithms we studied can be classified roughly into one of the following categories (for a given S/D classifications of the input variables). A sequence of operations is *data-independent* if the control flow can be determined at specialization time and does not depend on numeric data. If the control flow is dynamic, then only few computations will be static in a loop since many of the operations will depend on the iteration variable. If the entire control flow and all computations are static, then specialization reduces to ordinary computation.

- Dynamic control flow / dynamic computations; *e.g.* Newton iteration.
- Partially static control flow / dynamic computations; *e.g.* PEQ (Section 4.3).
- Static control flow / dynamic computations; *e.g.* CSI with n static (Section 4.2).
- Static control flow / partially static computations; *e.g.* FFT, CC (Section 4.4, 4.5).

Effects and Limitations. The specialization effects we observed are typically due to *unfolding* (unrolling) of loops and *elimination* of conditionals, *interprocedural constant propagation* (as opposed to intraprocedural), *procedure specialization* (cloning), *precomputation* of *indices* and *coefficients* (*e.g.* involving trigonometric functions). Our experience shows that specialized programs often enable further compiler optimizations (*e.g.* when array indices become known); thus, the choice of the compiler optimization level affects the speedup.

The gain in efficiency has its price (program size) and it is not always desirable to fully unfold loops since the number of iteration may be extremely large; *e.g.* due to the stability condition for solving parabolic equations the number of time steps M must be very large and unfolding the M-bound loop is unacceptable (Section 4.3). Other numerical methods, such as the Romberg integration (Section 4.1) or the Chebyshev approximation (Section 4.5), converge much faster; hence unfolding may be practical.

Partial evaluation is not very effective when computations are extremely data-dependent. For example, in techniques for linear programming the choice of the pivot is not known before run-time, but depending on this choice different computations have to be performed. However, specialization may still remove boundary checks and precompute indices.

4 BINDING-TIME ANALYSIS OF MATHEMATICAL ALGORITHMS

To demonstrate the effectiveness of the approach we chose five well-known algorithms from different subject areas: numerical integration, partial differential equations, function approximation and interpolation.[1] We refer to the literature for a description of the mathematical methods; the algorithms were taken from (Kincaid & Cheney, 1991), (Press et al., 1993). Details about the Fortran partial evaluator can be found in (Kleinrubatscher et al., 1995); a similar system exists for C (Andersen, 1994).

4.1 Romberg integration

The Romberg integration (RI) approximates the integral of a function f in an interval $[a,b]$ using trapezoidal estimates. The input of the RI is the lower and upper limit a,b of the interval, the number of iterations M and values for the function f.

[1]The algorithms were written in Fortran 77. The run times (= user time + system time) are given in seconds using the HP-UX Fortran 77 compiler (optimization option: +O2 for CSI and FFT, +O1 otherwise) and a HP workstation (Apollo 9000, 99 MHz PA-RISC 7100). The size of the programs is lines of 'pretty-printed' text and KBytes of executable code.

Analysis. We consider the S/D classification where M is static. The control-flow is entirely data-independent where all index computations and the weight factor $(4^m - 1)$ are static. The numerical computations are dominated by the computation of the quantities $R(n, 0)$ from f. In the remainder of the algorithm additional quantities $R(n, m)$ are to be computed iteratively $(1 \leq n \leq M, 1 \leq m \leq n)$. The interval boundaries a,b are only required for computing the initial quantities $R(n, 0)$.

Static	Computations saved	Size estimation
M	test of 3 iterations index computations divisor $(4^m - 1)$	unfolding 3 iterations $O(2^M)$

Results. The results show speedup factors between 1.16 and 1.46 for static M (Table 1), where M ranges from 2 to 10 (run-times for 10000 repetitions). The code size grows exponentially $O(2^M)$, but the Romberg integration converges fast to the integral of f; hence only a moderate value of M isneeded.

4.2 Cubic splines interpolation

The cubic splines interpolation (CSI) approximates a function using a cubic polynomial. The input of the CSI is the number of points n, their x-coordinates $x[\,]$ and their y-coordinates $y[\,]$. The output is the second derivative for each point (the computation of the y-coordinate for an arbitrary point is then straightforward). The algorithm uses a natural cubic spline.

Analysis. We consider two S/D classifications of the input variables. Motivation: the number of points is often known beforehand and the x-coordinates may be fixed when regular measurements are taken. The control-flow is entirely data-independent and all iterations can be unfolded. If the x-coordinates $x[\,]$ are static, a series of coefficients can be precomputed.

Static	Computations saved	Size estimation
n	tests of 3 iterations index computations	unfolding 3 iterations $O(n) \approx n(\lvert\mathrm{for}_{i_1}\rvert + \lvert\mathrm{for}_{i_2}\rvert + \lvert\mathrm{for}_{i_3}\rvert)$
$n, x[\,]$	tests of 3 iteration index computations coefficient computations	unfolding 3 iterations $O(n) \approx n(\lvert\mathrm{for}_{i_1}\rvert + \lvert\mathrm{for}_{i_2}\rvert + \lvert\mathrm{for}_{i_3}\rvert)$

Results. The results for the CSI show speedup factor between 1.16 and 1.63 for n static (Table 3), and 1.63 and 2.19 for n and $x[\,]$ static (Table 4), where n ranges from 10 to 1000 (run-times for 100000 repetitions). Note how an additional static $x[\,]$ reduces the size of the residual programs while increasing the speedup. The code size grows linearly $O(n)$.

4.3 Partial differential equations of parabolic type

Partial differential equations are usually solved using the finite difference method. The approximative values of the solution function are computed at so-called *mesh points* (x_i, t_j) and the numerical solution is advanced step by step in the time-direction. The input for solving parabolic equations (PEQ) with the explicit method is the number of mesh points n (x-direction), the step size k and the number of steps M (t-direction), as well as the values $v_{i,0}$ of the initial profile and the boundary values $v_{0,j}$, $v_{n+1,j}$ $(1 \leq i \leq n, 0 \leq j \leq M)$. The output are the values $v_{i,M}$ of the last iteration.

For the algorithm to be stable, it is necessary to assume $k \leq \frac{1}{2}(\frac{1}{n+1})^2$. For example, if $n = 99$ then the largest permissible value for step size $k = \frac{1}{2}10^{-4}$. A solution for $0 \leq t \leq 10$ then requires $M \geq \frac{1}{5}10^6$. Unfolding the M-bound iteration is therefore unacceptable.

Analysis. We consider the S/D classification where the number n is static. The control-flow is entirely data-independent, all numerical operations are dynamic. Making more parameters static will not contribute to the speedup: the initial profile is used only in the first iteration and the boundary values only for computing two new values in each iteration (moreover, we do not want to unfold the M-bound iteration).

Static	Computations saved	Size estimation
n	tests of 3 iterations	unfolding 3 iterations
	index computations	$O(n) \approx n(\lvert\text{for}_\text{l}\rvert + \lvert\text{for}_{\text{i}_1}\rvert + \lvert\text{for}_{\text{i}_2}\rvert)$

Results. The results show speedup factors between 1.63 and 1.85 for static n (Table 2), where n ranges from 10 to 1000 (run-times for 10000 repetitions). The computation of new mesh points is straightforward involving only arithmetic operations; this is reflected in the speedup. The code size grows linearly $O(n)$.

4.4 Fast Fourier transformation

The fast Fourier transformation (FFT) approximates the Fourier transformation using N points. The FFT is the fastest known algorithm for calculating a discrete Fourier Transformation. The input is a sequence of points (given as a function f) and a variable N indicating the number of points. The output is the Fourier coefficient at each point.

Analysis. We consider the S/D classification where $N(= 2^m)$ is static. The FFT is entirely data-independent with a significant number of numerical operations depending only on the number of points.

Static	Computations saved	Size estimation
$N(= 2^m)$	tests of 5 iterations	unfolding 5 iterations
	computations of w, Z	$O(N \log_2 N) \ll O(N^2)$
	index computations using j, k, n	
	index computations for $C[\,]$, $D[\,]$	

Results. The results for the FFT show speedup factors between 1.83 and 5.05 for static N (Table 5), where N ranges from 16 to 512 (run-times for 10000 repetitions). The computational costs of the FFT are mirrored in the growth of code size $O(N \log_2 N)$. Note the good speedup for larger N despite the growth in code size.

4.5 Chebyshev approximation

Chebyshev polynomials approximate continuous functions in a given interval. Once the Chebyshev coefficients (CC) are determined the approximation of $f(x)$ for arbitrary x is straightforward. They yield the 'most accurate' approximation of degree n. The input of the algorithm computing the CC is the lower and upper limit a,b of the interval, the maximum degree n of the Chebyshev polynomials, and f. The output are the Chebyshev coefficients.

Analysis. We consider the S/D classification where n is static. The CC is entirely data-independent with a significant number of numerical computations depending only on the degree n (*cf.* Section 4.4).

Static	Computations saved	Size estimation
n	tests of 3 iteration index computations coefficients (cos)	unfolding 3 iterations $O(n^2) \approx n^2(\lvert\mathrm{for_j}\rvert\lvert\mathrm{for_k}\rvert)$

Results. The results show remarkable speedup factors between 8.50 and 14.8 for static n (Table 6), where n ranges from 10 to 100 (run-time for 10000 repetitions). The speedup is mostly due to the precomputation of the trigonometric coefficients. To determine their influence, we removed their computation: the speedup was still 4.2. This shows that the specialization effect depends strongly on the efficiency of the standard functions in the mathematical library, but also gives a surprisingly good lower bound for the speedup. The code size grows $O(n^2)$, but the approximation error decreases rapidly; hence only a moderate degree n of the polynomial is needed (*e.g.* 30 or 50).

5 FURTHER WORK

Further work is desirable in several directions. Partial evaluators for numerical programs could take advantage of additional knowledge about mathematical and scientific functions, and exploit algebraic simplifications. Additional precision could be gained by online partial evaluation techniques and exploiting information about partially static data structures (*e.g.* sparse matrices). A combination of partial evaluation and traditional compiler optimizations seems promising; more should be known about their interaction. We expect that these techniques will improve the results presented in this paper. Another interesting direction is to exploit the parallelism exposed by partial evaluation on parallel computers.

REFERENCES

Andersen, L.O. (1994) *Program Analysis and Specialization for the C Programming Language.* DIKU, Department of Computer Science, University of Copenhagen. DIKU Report 94/19.

Andersen, P.H. (1995) *Partial Evaluation Applied to Ray Tracing.* DIKU, Department of Computer Science, University of Copenhagen. DIKU Report 95/2.

Baier, R., Glück, R., Zöchling, R. (1994) Partial evaluation of numerical programs in Fortran, in *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* University of Melbourne, Technical Report 94/9.

Berlin, A., Weise, D. (1990) Compiling scientific code using partial evaluation. *IEEE Computer,* **23**(12), 25–37.

Goad, C. (1982) Automatic construction of special purpose programs, in *6th Conference on Automated Deduction* (ed. D.W. Loveland), LNCS, vol. 138, Springer-Verlag.

Gustavson, F.G., Linige,r W., Willoughby, A.R. (1970) Symbolic generation of an optimal Crout algorithm for sparse systems of linear equations. *J. of the ACM,* **17**(1), 87–109.

Jones, N.D., Gomard, C.K., Sestoft, P. (1993) *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice Hall.

Kincaid, D., Cheney, W. (1991) *Numerical Analysis: Mathematics of Scientific Computing.* Brooks/Cole.

Kleinrubatscher, P., Kriegshaber, A., Zöchling, R., Glück, R. (1995) Fortran program specialization. *SIGPLAN Notices,* **30**(4), 61–70.

Metzger, R., Stroud, S. (1993) Interprocedural constant propagation: an empirical study. *ACM Letters on Programming Languages and Systems,* **2**(1–4), 213–32.

Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P. (1993) *Numerical Recipes in Fortran: the Art of Scientific Computing.* 2nd ed., Cambridge University Press.

APPENDIX 1 RESULTS

Table 1 Romberg (M static)

RI	Time	Ratio	Lines	KB
Src	0.16		99	20.5
Res, M=2	0.12	**1.33**	52	20.5
Src	0.44		99	20.5
Res, M=4	0.33	**1.33**	112	20.5
Src	1.24		99	20.5
Res, M=6	0.85	**1.46**	252	24.6
Src	3.98		99	20.5
Res, M=8	3.09	**1.29**	688	36.9
Src	14.37		99	20.5
Res, M=10	12.24	**1.17**	2284	86.0
Src	54.79		99	20.5
Res, M=12	47.42	**1.16**	8496	282.6

Table 2 Parabolic equation (n static)

PEQ	Time	Ratio	Lines	KB
Src	6.69		86	20.5
Res, n=10	4.11	**1.63**	115	20.5
Src	29.14		86	20.5
Res, n=50	16.76	**1.74**	315	28.7
Src	57.45		86	20.5
Res, n=100	32.44	**1.77**	565	36.9
Src	113.26		86	20.5
Res, n=200	63.98	**1.77**	1065	57.3
Src	169.98		86	20.5
Res, n=300	96.70	**1.76**	1565	77.8
Src	298.62		86	20.5
Res, n=500	161.82	**1.85**	2565	114.7
Src	587.82		86	20.5
Res, n=1000	319.02	**1.84**	5065	217.1

Table 3 Cubic splines (n static)

CSI	Time	Ratio	Lines	KB
Src	0.57		74	20.5
Res, $n = 10$	0.35	**1.63**	94	20.5
Src	3.16		74	20.5
Res, $n = 50$	2.15	**1.47**	457	28.7
Src	5.12		74	20.5
Res, $n = 100$	4.35	**1.18**	907	36.9
Src	10.25		74	20.5
Res, $n = 200$	8.66	**1.18**	1807	57.3
Src	15.46		74	20.5
Res, $n = 300$	13.28	**1.16**	2707	77.8
Src	25.60		74	20.5
Res, $n = 500$	22.03	**1.16**	4507	118.8
Src	52.28		74	20.5
Res, $n = 1000$	45.02	**1.16**	9007	262.1

Table 4 Cubic splines ($n, x[\,]$ static)

CSI	Time	Ratio	Lines	KB
Src	0.57		74	20.5
Res, $n = 10, x$	0.30	**1.90**	74	20.5
Src	3.16		74	20.5
Res, $n = 50, x$	1.44	**2.19**	354	24.6
Src	5.12		74	20.5
Res, $n = 100, x$	3.02	**1.70**	704	28.7
Src	10.25		74	20.5
Res, $n = 200, x$	6.28	**1.63**	1404	41.0
Src	15.46		74	20.5
Res, $n = 300, x$	9.29	**1.66**	2104	49.2
Src	25.60		74	20.5
Res, $n = 500, x$	15.73	**1.63**	3504	73.7
Src	52.28		74	20.5
Res, $n = 1000, x$	30.59	**1.71**	7004	122.9

Table 5 FFT (N static)

FFT	Time	Ratio	Lines	KB
Src	2.17		151	20.5
Res, N=16	0.43	**5.05**	701	24.6
Src	5.71		151	20.5
Res, N=32	1.37	**4.17**	1469	36.9
Src	15.27		151	20.5
Res, N=64	4.29	**3.56**	3069	57.3
Src	40.68		151	20.5
Res, N=128	14.22	**2.86**	6397	102.4
Src	115.45		151	20.5
Res, N=256	50.31	**2.29**	13309	192.5
Src	1183.78		151	20.5
Res, N=512	647.72	**1.83**	27645	720.9

Table 6 Chebyshev (n static)

CC	Time	Ratio	Lines	KB
Src	1.70		72	20.5
Res, n=10	0.20	**8.50**	170	20.5
Src	6.21		72	20.5
Res, n=20	0.55	**11.3**	520	28.7
Src	13.70		72	20.5
Res, n=30	1.22	**11.2**	1070	41.0
Src	23.96		72	20.5
Res, n=40	2.38	**10.1**	1820	57.3
Src	38.75		72	20.5
Res, n=50	2.62	**14.8**	2770	81.9
Src	145.14		72	20.5
Res, n=100	14.74	**9.85**	10520	335.8

APPENDIX 2 ANNOTATED ALGORITHMS

The algorithms are presented in a pseudo code containing additional details beyond the pure mathematical formula; see Kincaid & Cheney (1991), Press et al. (1993). The program annotations were determined automatically by the monovariant binding-time analysis (Section 2).

Notation. The for $i = e_1, e_2, e_3$ do ... end denotes an iteration from e_1 to e_2 with a step of e_3 (if e_3 is omitted then 1 is assumed). Dynamic variables and operations are annotated as $\boldsymbol{op}$; all other operations op are static (*i.e.* they can be precomputed at specialization time).

Romberg integral (M static)

```
input $\boldsymbol{a}, \boldsymbol{b}, M, \boldsymbol{f}$
integer $M, n, m, i$
real $\boldsymbol{h}, \boldsymbol{s}$
real function $\boldsymbol{f}$
real array $\boldsymbol{r}[\,,]$
$\boldsymbol{h} \leftarrow \boldsymbol{b}\boldsymbol{-}\boldsymbol{a}$
$\boldsymbol{r}[0,0] \leftarrow (\boldsymbol{f}(\boldsymbol{a})\boldsymbol{+}\boldsymbol{f}(\boldsymbol{b}))\boldsymbol{*}\boldsymbol{h}\boldsymbol{/}2$
for $n = 1, M$ do
   $\boldsymbol{h} \leftarrow \boldsymbol{h}\boldsymbol{/}2$
   $\boldsymbol{s} \leftarrow 0$
   for $i = 1, 2**(n-1)$ do
      $\boldsymbol{s} \leftarrow \boldsymbol{s}\boldsymbol{+}\boldsymbol{f}(\boldsymbol{a}\boldsymbol{+}(2*i-1)\boldsymbol{*}\boldsymbol{h})$
   end
   $\boldsymbol{r}[n,0] \leftarrow \boldsymbol{r}[n-1,0]\boldsymbol{/}2\boldsymbol{+}\boldsymbol{h}\boldsymbol{*}\boldsymbol{s}$
   for $m = 1, n$ do
      $\boldsymbol{r}[n,m] \leftarrow \boldsymbol{r}[n,m-1]\boldsymbol{+}\frac{1}{(4**m-1)}\boldsymbol{*}$
               $(\boldsymbol{r}[n,m-1]\boldsymbol{-}\boldsymbol{r}[n-1,m-1])$
   end
end
output $\boldsymbol{r}[M,M]$
```

Cubic splines interpolation ($n, x[\,]$ static)

```
input $n, x[\,], \boldsymbol{y}[\,]$
integer $i, n$
real array $\boldsymbol{b}, h, x, \boldsymbol{u}, \boldsymbol{v}, \boldsymbol{y}, \boldsymbol{z}$
for $i = 0, n-1$ do
   $h[i] \leftarrow x[i+1] - x[i]$
   $\boldsymbol{b}[i] \leftarrow (6/h[i])\boldsymbol{*}(\boldsymbol{y}[i+1]\boldsymbol{-}\boldsymbol{y}[i])$
end
$\boldsymbol{u}[1] \leftarrow 2*(h[0]+h[1])$
$\boldsymbol{v}[1] \leftarrow \boldsymbol{b}[1]\boldsymbol{-}\boldsymbol{b}[0]$
for $i = 2, n-1$ do
   $\boldsymbol{u}[i] \leftarrow 2*(h[i]+h[i-1])\boldsymbol{-}h[i-1]**2\boldsymbol{/}\boldsymbol{u}[i-1]$
   $\boldsymbol{v}[i] \leftarrow \boldsymbol{b}[i]\boldsymbol{-}\boldsymbol{b}[i-1]\boldsymbol{-}h[i-1]\boldsymbol{*}\boldsymbol{v}[i-1]\boldsymbol{/}\boldsymbol{u}[i-1]$
end
$\boldsymbol{z}[n] \leftarrow 0$
for $i = n-1, 1, -1$ do
   $\boldsymbol{z}[i] \leftarrow (\boldsymbol{v}[i]\boldsymbol{-}h[i]\boldsymbol{*}\boldsymbol{z}[i+1])\boldsymbol{/}\boldsymbol{u}[i]$
end
$\boldsymbol{z}[0] \leftarrow 0$
output $\boldsymbol{z}[\,]$
```

Parabolic equation (n static)

```
input $n, k, \boldsymbol{M}$
integer $n, \boldsymbol{M}, l, j$
real $k, h, \boldsymbol{s}, \boldsymbol{ss}, \boldsymbol{t}$
real function $\boldsymbol{g}, \boldsymbol{a}, \boldsymbol{b}$
real array $\boldsymbol{v}, \boldsymbol{w}$
$h \leftarrow 1/(n+1)$
$\boldsymbol{s} \leftarrow k/(h**2)$
$\boldsymbol{ss} \leftarrow 1\boldsymbol{-}2\boldsymbol{*}\boldsymbol{s}$
for $l = 0, n+1$ do
   $\boldsymbol{w}[l] \leftarrow \boldsymbol{g}(l)$
end
$\boldsymbol{t} \leftarrow 0$
for $j = 1, \boldsymbol{M}$ do
   $\boldsymbol{t} \leftarrow \boldsymbol{t}\boldsymbol{+}k$
   $\boldsymbol{v}[0] \leftarrow \boldsymbol{a}(\boldsymbol{t})$
   $\boldsymbol{v}[n+1] \leftarrow \boldsymbol{b}(\boldsymbol{t})$
   for $i = 1, n$ do
      $\boldsymbol{v}[i] \leftarrow \boldsymbol{s}\boldsymbol{*}\boldsymbol{w}[i-1]\boldsymbol{+}\boldsymbol{ss}\boldsymbol{*}\boldsymbol{w}[i]\boldsymbol{+}\boldsymbol{s}\boldsymbol{*}\boldsymbol{w}[i+1]$
   end
   for $i = 0, n+1$ do
      $\boldsymbol{w}[i] \leftarrow \boldsymbol{v}[i]$
   end
end
output $\boldsymbol{v}[\,]$
```

FFT ($N = 2^m$ static)

```
input $m, f$
integer $j, k, m, n, N$
real const $\pi = 3.14...$
complex $u, v, w$
complex array $C, D, Z$
complex function $f$
$N \leftarrow 2{*}{*}m$
$w \leftarrow \exp(-2 * \pi * \sqrt{-1}/N)$
for $k = 0, N-1$ do
    $Z[k] \leftarrow w{*}{*}k$
    $C[k] \leftarrow f(2 * \pi * k/N)$
end
for $n = 0, m-1$ do
    for $k = 0, 2{*}{*}(m-n-1)-1$ do
        for $j = 0, 2{*}{*}n - 1$ do
            $u \leftarrow C[2{*}{*}n * k + j]$
            $v \leftarrow Z[j * 2{*}{*}(m-n-1)]*$
                    $C[2{*}{*}n * k + 2{*}{*}(m-1) + j]$
            $D[2{*}{*}(n+1) * k + j] \leftarrow (u+v)/2$
            $D[2{*}{*}(n+1) * k + j + 2{*}{*}n]$
                                $\leftarrow (u-v)/2$
        end
    end
    for $j = 0, N-1$ do
        $C[j] \leftarrow D[j]$
    end
end
output $C[\,]$
```

Chebyshev approximation (n static)

```
input $n, xa, xb, func$
integer $n, k, j$
real $xa, xb, xm, xp, sm$
real function $func$
real array $c, f$
real const $\pi = 3.14...$
$xp \leftarrow (xb+xa)/2$
$xm \leftarrow (xb-xa)/2$
for $k = 1, n$ do
    $f[k] \leftarrow func(xp+xm* \cos(\pi * (k-0.5)/n))$
end
for $j = 0, n-1$ do
    $sm \leftarrow 0$
    for $k = 1, n$ do
        $sm \leftarrow sm+f[k]* \cos(\pi * j * (k-0.5)/n)$
    end
    $c[j] \leftarrow (2/n)*sm$
end
output $c[\,]$
```