Available online at www.sciencedirect.com

SCIENCE DIRECT

Computers and Mathematics with Applications 51 (2006) 51–72

An International Journal
computers & mathematics with applications

www.elsevier.com/locate/camwa

An Efficient Algorithm for the Evaluation of Convolution Integrals

K. DIETHELM
Institut Computational Mathematics
Technische Universität Braunschweig
Pockelsstraße 14, 38106 Braunschweig, Germany
and
GNS Gesellschaft für Numerische Simulation mbH
Am Gaußberg 2, 38114 Braunschweig, Germany
diethelm@gns-mbh.com

A. D. FREED*
Bio Science and Technology Branch, MS 49-3
NASA Glenn Research Center
21000 Brookpark Road, Cleveland, OH 44135, U.S.A.
Alan.D.Freed@nasa.gov

(Received August 2004; revised and accepted July 2005)

Abstract—We propose an algorithm for the numerical evaluation of convolution integrals of the form $\int_0^x k(x-y)f(y,x)\,\mathrm{d}y$, for $x \in [0, X]$. Our method is especially suitable in situations where the fundamental interval $[0, X]$ is very long and the kernel function k is expensive to calculate. Separate versions are provided where the forcing function f is known in advance, and where it must be determined step-by-step along the solution path. These methods are efficient with respect to both run time and memory requirements. © 2006 Elsevier Ltd. All rights reserved.

Keywords—Convolution integral, Numerical approximation, Quadrature, Logarithmic memory, Viscoelasticity, Mittag-Leffler function.

1. INTRODUCTION

The numerical evaluation of convolution integrals of the form

$$\int_0^x k(x-y)f(y,x)\,\mathrm{d}y, \qquad x \in [0, X], \tag{1}$$

with a given kernel function k and a given integrand (or forcing) function f is a standard problem arising in many areas of mathematics, physics, and engineering.

Partially supported by U.S. Army Medical Research and Material Command Grant No. DAMD17-01-1-0673 to The Cleveland Clinic (PI: Dr. Ivan Vesely).

*Author to whom all correspondence should be addressed.

Typical kernels are functions like monomials $k(t) = t^a$, for some $a > -1$, or exponentials $k(t) = \exp(-bt)$, for some $b > 0$. In classical cases like these, the kernels can be evaluated very cheaply, and hence, it is rather easy to construct fast, accurate, numerical algorithms for these types of convolution integrals.

In recent applications [1] concerning the modeling of soft biological tissues, with help from the fractional calculus, we encountered a need to deal with a kernel of the form

$$k(t) = -\frac{E_{\alpha,0}(-(t/\tau)^\alpha)}{t} \geq 0, \qquad t > 0, \tag{2}$$

with certain constants $\tau > 0$ and $\alpha \in (0,1)$, where $E_{\alpha,0}$ is the two-parameter Mittag-Leffler function defined by [2],

$$E_{\alpha,\beta}(z) := \sum_{\ell=0}^{\infty} \frac{z^\ell}{\Gamma(\beta + \alpha\ell)}, \qquad \alpha > 0, \quad \beta \in \mathbb{R}, \quad z \in \mathbb{C}.$$

(Note here that the summand for $\ell = 0$ vanishes if $\beta = 0$ in view of the Gamma function's pole at the origin.) To the best of our knowledge, the only reliable numerical algorithm presently available for the accurate evaluation of such functions is described in [3] and recalled in [4], and it has the disadvantage of being very slow. In a typical environment, the computation of a single function call to the Mittag-Leffler function may require more than 1000 times the amount of time necessary for a like calculation of an exponential or a monomial. Standard algorithms for convolution integrals that do not take such issues into account may not be able to produce results in acceptable time, especially if approximations are sought on a fine discretization over interval $[0, X]$ given a large value for X.

In view of the potentially high cost in evaluating such a kernel, it is prudent that one's algorithm stores kernel function values, and reuses them whenever possible, thus minimizing the total number of calls to the kernel evaluation routine. In an application like the one described in [1], the number of convolution integrals that need to be approximated can become very large, and as such, storage requirements may introduce new difficulties due to possible limitations in computer resources. This problem is taken into account in the construction of our numerical algorithms. Their storage requirements will be seen to be modest as well.

With this as our motivation, we present three different algorithms capable of solving this problem. Specifically, the fundamental approach is outlined in Section 2, where we also derive the first algorithm. This routine will be seen to have certain weaknesses in that its convergence order is not very high, but we still have decided to include it into this paper because it has some merits: it is fast and rather simple to implement, and it helps us to explain the fundamental concepts behind the construction of the other methods. We modify the method in Section 3, resulting in a technically somewhat more complex algorithm that remains fast but which now converges better.

Finally, in Section 4, we provide yet another variant of our method that is more appropriate for situations where there is a two-way communication between the integration routine and a second procedure used to compute the forcing function f. In such cases, the forcing function f is not known *a priori*, but rather, must be computed at each step along a solution path (e.g., in finite elements) and consequently needs to be stored in a special history array.

A section with numerical examples concludes the paper.

2. THE FUNDAMENTAL ALGORITHM

In order to develop a suitable algorithm, we first collect some essential properties of our problem.

THEOREM 1. *Let $0 < \alpha < 1$, and assume the kernel function k to be defined according to equation (2).*

(a) *The kernel function k has an integrable singularity at the origin.*
(b) *The kernel function k is monotonically decreasing, and its asymptotic behavior as $u \to \infty$ is described by $k(u) \sim u^{-\alpha-1}$.*

Note that Property (a) is a property that is shared by many other kernels arising in various applications. Property (b) indicates that the decay of $k(u)$ as $u \to \infty$ is only algebraic and not exponential (i.e., rather slow).

PROOF. As mentioned above, we are only interested in the case $0 < \alpha < 1$. The definition (2) of the kernel in combination with the power series expansion of $E_{\alpha,0}$ gives that $k(u) = cu^{\alpha-1}$ + higher-order terms near the origin, and this implies (a).

Property (b) follows from the fact that $k(t) = -\frac{d}{dt}E_{\alpha,1}(-(t/\tau)^\alpha)$ (see [1;5, p. 22]), and the properties of the derivative of the Mittag-Leffler function $E_{\alpha,1}$ described by Gorenflo and Mainardi [6]. ∎

In addition to the properties mentioned in Theorem 1, our original algorithm is based on the following features of the problem.

- The convolution integral in equation (1) is more general than is typically considered, in that the forcing function f in the integrand depends not only on the integration variable y, but also on the upper limit of integration x.
- If we had an exponential kernel function, $k(u) = \exp(-u)$, we could use its functional (recursive) equation to get rid of the nonlocal character of the convolution integral, decomposing it (as x grows) into the sum of a known term and a local component that can be evaluated quickly (e.g., see [7, Ch. 10]). In many cases, the Mittag-Leffler function included, no such recursive equation exists, and therefore, the integral retains its nonlocal structure, which necessarily leads to longer run times.
- Algorithms that evaluate the kernel k listed in equation (2) are available [3,4], but they are computationally expensive.

As a consequence of these properties, we aim to construct an algorithm that requires a minimal number of evaluations of k, and that attempts to store and reuse these function values whenever possible, but without consuming too much memory. Specifically, we seek to minimize the influence of the nonlocality of the integral on the algorithm's run time.

The key to our algorithm is that we introduce two parameters. The first is a characteristic time $T > 0$, say, that quantifies the decay behavior of the kernel k. This parameter is chosen at the beginning of the process and kept fixed from then on. For example, T could be chosen such that $\int_0^T k(u)\,du = C\int_0^\infty k(u)\,du$ with a certain value of C, so that the "energy" introduced during the time interval $[0, T]$ is a suitable fraction of the "total energy" introduced over $[0, \infty)$. In our case, described in equation (2) above, we would choose $T = \tau$.

Moreover, we select a quality parameter in the form of integer $Q \geq 2$ that we also keep fixed throughout the process. It is evident from our considerations below that a small value for Q will lead to a fast but not so accurate implementation of the algorithm; large choices for Q improve the accuracy but run slower (see the examples in Section 5). According to our experience, in many applications $Q \approx 5$ turned out to be a useful compromise between accuracy and speed. We explicitly draw the reader's attention to the fact that the parameter $Q \in \{2, 3, 4, \dots\}$ must be chosen in advance and kept fixed. Apart from this, there are no restrictions on the choice of Q for the first two algorithms, while the last algorithm additionally requires Q to be odd.

Having chosen the two numbers T and Q, we then define

$$\mu := \left\lceil \log_Q \frac{X}{T} \right\rceil,$$

with $\lceil\cdot\rceil$ denoting the usual ceiling function (round upwards to the nearest integer). As above, X denotes the upper limit of the interval where we need to calculate the integral. Note that, since Q, T, and X are fixed, the parameter μ does not vary either. This choice of μ asserts that

$$\bar{X} := Q^{\mu}T \geq X,$$

and hence, we may rewrite the convolution integral in question as

$$\int_0^x k(x-y)f(y,x)\,\mathrm{d}y = \int_0^x k(u)f(x-u,x)\,\mathrm{d}u = \int_0^{\bar{X}} k(u)f_x(u)\,\mathrm{d}u, \tag{3}$$

where the new function f_x is defined by

$$f_x(u) := \begin{cases} f(x-u,x), & \text{if } 0 \leq u \leq x, \\ 0, & \text{otherwise}, \end{cases} \tag{4}$$

which rearranges equation (1) into a standard integration problem: calculate a number of integrals over the same range of integration, where the integrands all have the form of products with identical first factors but varying second factors. Moreover, the first factor of each integral (i.e., the function k) exhibits the features mentioned in Theorem 1.

Assuming that we want to calculate the integral on the right-hand side of equation (3), for some $x \in [0, X]$, we then choose a mesh size $h > 0$, such that $h = T/S$ for some integer $S \geq 4$, and thereby introduce an equispaced grid, $x_n := nh$, $n = 0, 1, \dots, N$ where $N := Q^{\mu}S$. Parameter S designates the number of integration steps per unit characteristic time T, and as such, is a second field that affects algorithm accuracy, Q being the other field.

We now construct a method to calculate the integral at any desired grid point(s) x_n. The basic idea is to use the so-called logarithmic memory concept of Ford and Simpson [8], *viz.*, we decompose the integral in question as

$$\int_0^{\bar{X}} k(u)f_x(u)\,\mathrm{d}u = \left(\int_0^{QT} + \int_{QT}^{Q^2T} + \dots + \int_{Q^{\mu-1}T}^{Q^{\mu}T}\right) k(u)f_x(u)\,\mathrm{d}u. \tag{5}$$

On each subinterval we propose using a quadrature formula with equispaced nodes to approximate the corresponding integral. Bearing in mind the monotonic decay properties of the kernel k (see Theorem 1 (b)), we suggest to use a step size of h on the first subinterval $[0, QT]$, a step size of Qh on the second subinterval $[QT, Q^2T]$, a step size of Q^2h on the third subinterval $[Q^2T, Q^3T]$, etc.

In most applications one is likely to have a forcing function f that is in $C^{\ell}([0,X]^2)$ with some moderately large value of ℓ. Thus, there is no reason to assume a high degree of smoothness of the full integrand $f_x \cdot k$, and hence, it does not make sense to employ a highly accurate but complicated quadrature formula like, e.g., the Gaussian method. Rather, a useful choice for the quadrature method is the midpoint rule with a Laplace end correction. This formula, which exhibits $O(h^5)$ convergence for smooth integrands, is given by (see [9, Section V.8])

$$\begin{aligned}\int_a^b g(x)\,\mathrm{d}x \approx h\Bigg\{&\sum_{j=1}^{J} g\left(a+\frac{2j-1}{2}h\right) + \frac{703}{5760}\left[g\left(a+\frac{1}{2}h\right)+g\left(b-\frac{1}{2}h\right)\right] - \frac{463}{1920}\left[g\left(a+\frac{3}{2}h\right)\right.\\ &\left.+g\left(b-\frac{3}{2}h\right)\right] + \frac{101}{640}\left[g\left(a+\frac{5}{2}h\right)+g\left(b-\frac{5}{2}h\right)\right] - \frac{223}{5760}\left[g\left(a+\frac{7}{2}h\right)+g\left(b-\frac{7}{2}h\right)\right]\Bigg\},\end{aligned} \tag{6}$$

where $J = (b-a)/h \geq 4$. There are two fundamental reasons behind selecting this construction.

- The structure of the quadrature formula is rather simple. Specifically, the functions k and f are sampled on a highly regular grid, and the function values of k at these sampling points are used again and again, so the overall process is rather economical. This is

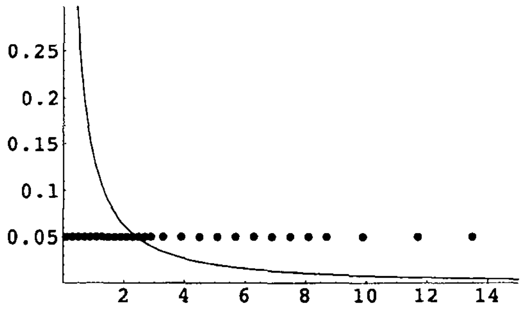

Figure 1. Kernel function k from equation (2) for $\alpha = 1/2$ and $\tau = 1$, and integration points for $T = 1$, $Q = 3$, and $S = 5$ (each dot represents one integration point).

important in light of the expense in computing k. In spite of this simplicity, the method still gives a reasonably quick convergence if the integrand happens to be smooth.

- We follow a concept known in numerical integration as "avoiding the singularity" [10,11] by not using 0 as a sampling point; in view of the fact that the precise behavior of the integrand as $y \to x$ (i.e., as $u \to 0$) may be problematic (and is not always known *a priori*), the convergence analysis of Braß [10] and the observations of Rabinowitz [11] indicate that avoiding this locale is an approach that is superior compared to methods using 0 as a quadrature node point as far as the order of magnitude of the error is concerned.

Consequently, we only need to evaluate the kernel k at the points $(j - 1/2)h$, $j = 1, 2, \ldots, QS$, and at $[T + (j - 1/2)h]Q^{i-1}$, $j = 1, 2, \ldots, (Q-1)S$, $i = 2, 3, \ldots, \mu$. Thus, the leftmost point used here is $h/2$, which is strictly greater than 0, which shows that we indeed avoid the use of the singularity of the kernel as an integration node. Figure 1 shows a typical kernel function and the distribution of the integration points for a concrete version of this approach. It is clearly seen that the spacing of the nodes is very fine near 0 (where the singularity of the kernel, i.e., the "difficult" part of the integral, is located) and becomes more and more coarse as we move away from the origin. Thus, the way in which the points are distributed is such that it counteracts the growth of k. The total number of evaluations of the function k can easily be counted, the result being $(\mu - 1)(Q-1)S + QS \leq \mu QS = O(\log X)$ for increasing X (if all other parameters remain fixed), and it is these values that need to be stored throughout the calculation. This observation shows that this integration algorithm is indeed memory efficient.

Moreover, the run time is also fast ($O(X \log X)$ to be precise) since each of the $O(X)$ steps requires $O(\log X)$ time units, because it involves a summation of terms, and the number of summands is identical to the number of evaluation points for k that we have seen to be $O(\log X)$.

Investigating the behavior of the algorithm as the step size $h \to 0$ (or equivalently, as $S \to \infty$) with the other parameters X, T, and Q being fixed, we find the following result.

THEOREM 2.

(a) *If the integrand* $f_x \cdot k \in C^5[0, X]$ *function then the error of the above scheme is* $O(h^5)$ *as* $h \to 0$.

(b) *If* $f \in C^1([0, X]^2)$ *then the error of the above scheme converges to* 0 *as* $h \to 0$.

PROOF. It is well known [9, Section V.8] that the midpoint formula with the Laplace end correction described in equation (6) converges as $O(h^5)$ if the integrand (denoted by g in equation (6)) is a C^5 function. Thus, proceeding as in the proof of [8, Theorem 1], we find (a).

The proof of (b) follows exactly the same lines as indicated in [10]. ∎

We note that some additional speedup may be achieved from the observation that some of the integrals over the subintervals indicated in the decomposition of equation (5) are zero because the function f_x in equation (4) vanishes on those intervals. Therefore, we can simply omit these integrals from the approximation process.

The following algorithm is well suited for implementing on a vector or parallel computer in that it is possible to distribute the work involved in computing the required function values of k over all available processors. This information can then be exchanged in a communication phase, and finally we can ask every processor to calculate the actual approximations of the convolution integrals $\int_0^{x_n} k(x_n - y) f(y, x_n)\,dy$ for some $n \in \{0, 1, 2, \dots, N\}$.

A pseudo-code description of this method is outlined in Algorithm 1 below. In this algorithm and in the following, $\lfloor\cdot\rfloor$ denotes the usual floor function (round down to nearest integer). Numerical examples follow in Section 5.

ALGORITHM 1. FUNDAMENTAL VERSION OF FAST CONVOLUTION QUADRATURE.

- Input values:
  - · univariate kernel function k,
  - · bivariate forcing function f,
  - · positive real numbers X and T,
  - · integers $S > 3$ and $Q > 1$,
- Output values:
  - · array $\mathbf{q} = \{q_0, q_1, q_2, \dots, q_N\}^T$ where $q_n \approx \int_0^{x_n} k(x_n - y) f(y, x_n)\,dy$.
- Body of the algorithm:

```
(* initialization of auxiliary variables *)
$h := T/S$;
$N := \lceil X/h \rceil$;
$\mu := \max\{1, \lceil \log_Q(X/T) \rceil\}$;
(* create table of required kernel values *)
FOR $j := 1$ TO $QS$ DO
  $\kappa_{1,j} := k((j - 1/2)h)$;
FOR $i := 2$ TO $\mu$ DO
  FOR $j := 1$ TO $(Q-1)S$ DO
    $\kappa_{i,j} := k((T + (j - 1/2)h)Q^{i-1})$;
(* determine weights of Laplace quadrature for first interval *)
FOR $i := 1$ TO $QS$ DO
  $\ell_i := 1$;
$\ell_1 := \ell_1 + 703/5760$;
$\ell_2 := \ell_2 - 463/1920$;
$\ell_3 := \ell_3 + 101/640$;
$\ell_4 := \ell_4 - 223/5760$;
$\ell_{QS} := \ell_{QS} + 703/5760$;
$\ell_{QS-1} := \ell_{QS-1} - 463/1920$;
$\ell_{QS-2} := \ell_{QS-2} + 101/640$;
$\ell_{QS-3} := \ell_{QS-3} - 223/5760$;
(* determine weights of Laplace quadrature for remaining intervals *)
FOR $i := 1$ TO $(Q-1)S$ DO
  $\tilde{\ell}_i := 1$;
$\tilde{\ell}_1 := \tilde{\ell}_1 + 703/5760$;
$\tilde{\ell}_2 := \tilde{\ell}_2 - 463/1920$;
$\tilde{\ell}_3 := \tilde{\ell}_3 + 101/640$;
$\tilde{\ell}_4 := \tilde{\ell}_4 - 223/5760$;
$\tilde{\ell}_{(Q-1)S} := \tilde{\ell}_{(Q-1)S} + 703/5760$;
$\tilde{\ell}_{(Q-1)S-1} := \tilde{\ell}_{(Q-1)S-1} - 463/1920$;
$\tilde{\ell}_{(Q-1)S-2} := \tilde{\ell}_{(Q-1)S-2} + 101/640$;
$\tilde{\ell}_{(Q-1)S-3} := \tilde{\ell}_{(Q-1)S-3} - 223/5760$;
```

```
(* Main part of algorithm *)
$q_0 := 0$;
(* subinterval from 0 to $QT$ *)
IF $\mu = 1$ THEN
  $m := N$;
ELSE
  $m := QS$;
(* calculate approximations $\int_0^{x_n} k(x_n - y) f(y, x_n)\,\mathrm{d}y$, $n = 1, 2, \dots, N$ *)
FOR $n := 1$ TO $m$ DO
  (* first subinterval *)
  $q_n := h \sum_{j=1}^{n} \ell_j \kappa_{1,j} f((n - j + 1/2)h, nh)$;
FOR $n := m + 1$ TO $N$ DO
  (* remaining subintervals *)
  $\nu := \lceil \log_Q (n/S) \rceil$;
  $q_n := h \sum_{j=1}^{QS} \ell_j \kappa_{1,j} f((n - j + 1/2)h, nh)$
        $+ h \sum_{i=2}^{\nu-1} Q^{i-1} \sum_{j=1}^{(Q-1)S} \tilde{\ell}_j \kappa_{i,j} f(nh - (T + (j - 1/2)h)Q^{i-1}, nh)$
        $+ hQ^{\nu-1} \sum_{j=1}^{\lfloor nQ^{1-\nu} - S - 1/2 \rfloor} \tilde{\ell}_j \kappa_{\nu,j} f(nh - (T + (j - 1/2)h)Q^{\nu-1}, nh)$;
RETURN $\mathbf{q}$.
```

3. AN IMPROVED VERSION OF THE ALGORITHM

It is evident from the simple numerical examples given in Section 5 that Algorithm 1 gives a fast, rough and ready scheme for our problem, but its accuracy is not very high, even for nicely behaved functions like in our first example. This apparent contradiction to the predicted $O(h^5)$ convergence behavior can be explained as follows. The algorithm is based on a numerical integration of the product $k \cdot f_x$ over the interval $[0, \bar{X}]$, see equation (3). But the function f_x is defined on this interval in a piecewise manner, see equation (4). As a consequence, given that f is smooth on $[0, X]$, then f_x will be smooth on the intervals $[0, X]$ and $[X, \bar{X}]$, but the transition between these two branches need not be smooth (typically, it will be continuous but not differentiable). Therefore, we cannot expect the Laplace quadrature formula (or, indeed, any other quadrature method) to achieve a high convergence order when it is applied over the complete interval $[0, \bar{X}]$. To overcome this deficiency, we construct a somewhat more sophisticated algorithm that uses similar concepts to the one listed above, but that now avoids integration across this critical point.

To this end, we use the same grid points $x_n = nh$ as before, but we introduce a slightly different scheme to approximate the integrals $\int_0^{x_n} k(u) f_{x_n}(u)\,\mathrm{d}u$. Here we distinguish between two cases, namely $x_n \le QT$ and $x_n > QT$.

In the first case, $x_n \le QT$, we have two subcases. For the subcase $n \ge 4$, we use the Laplace quadrature formula with step size h (i.e., with n nodes) to approximate the integral. This asserts the $O(h^5)$ convergence. However, the idea behind the construction of Laplace's method necessarily requires $n \ge 4$, which is why we constrain $S \ge 4$, and so we have to use a different concept for the other sub-case, $n \le 3$. Specifically, we choose the four-point MacLaurin rule [9, Section IV.1] to use on intervals $[0, x_n]$, $n = 1, 2, 3$, *viz.*,

$$\int_0^{x_n} k(u) f_{x_n}(u)\,\mathrm{d}u$$
$$\approx \frac{nh}{48} \left\{ 13 \left[k\left(\frac{7}{8}x_n\right) f\left(\frac{1}{8}x_n, x_n\right) + k\left(\frac{1}{8}x_n\right) f\left(\frac{7}{8}x_n, x_n\right) \right] \right.$$
$$\left. + 11 \left[k\left(\frac{5}{8}x_n\right) f\left(\frac{3}{8}x_n, x_n\right) + k\left(\frac{3}{8}x_n\right) f\left(\frac{5}{8}x_n, x_n\right) \right] \right\}.$$

Thus, we can stick to the idea of avoiding the singularity, and we can keep the $O(h^5)$ error estimate, but we are not able to use the precomputed and stored values of the kernel function. Instead, we have to perform twelve additional evaluations of k. Because this number remains fixed, independent of X, the overall arithmetic complexity remains unchanged.

In the second case, where $x_n > QT$, we now choose an integer σ, such that $Q^\sigma T < x_n \leq Q^{\sigma+1}T$, i.e., $\sigma = \lceil \log_Q(x_n/T) \rceil - 1 = \lceil \log_Q(n/S) \rceil - 1$. Hence, we decompose the integral as

$$\int_0^{x_n} k(u) f_{x_n}(u)\, du = \left(\int_0^{QT} + \int_{QT}^{Q^2T} + \cdots + \int_{Q^{\sigma-1}T}^{Q^\sigma T} + \int_{Q^\sigma T}^{x_n} \right) k(u) f_{x_n}(u)\, du. \qquad (7)$$

All integrals on the right-hand side of equation (7), except for the last one, can be approximated in exactly the same way as is done in Algorithm 1, because there are no problems with the critical point x_n associated with them. So, for these integrals we use the Laplace method with step sizes $h, Qh, Q^2h, \dots, Q^{\sigma-1}h$, respectively. Only the integral over $[Q^\sigma T, x_n]$ requires special attention, and we proceed as follows. By assumption, $T = Sh$ given some integer S, and hence, $Q^\sigma T = Q^\sigma Sh$; thus, the length of the interval of integration is $x_n - Q^\sigma T = nh - Q^\sigma Sh = \lambda h$ with $\lambda := n - Q^\sigma S \in \mathbb{N}$. Algorithm 1 would discretize this interval with a step size of $Q^\sigma h$, and this is what we shall try as well. The length of the interval is λh and we once again have two subcases, $\lambda < 4Q^\sigma$ and $\lambda \geq 4Q^\sigma$.

In the former case, the discretization will give rise to less than four subintervals, and therefore, we cannot use the Laplace method. Thus, we discard this idea and use the four-point MacLaurin method again, which for this subinterval reads as

$$\begin{aligned} &\int_{Q^\sigma T}^{x_n} k(u) f_{x_n}(u)\, du \\ &\approx \frac{\lambda h}{48} \left\{ 13 \left[k\left(\left(n - \frac{1}{8}\lambda\right) h\right) f\left(\frac{1}{8}\lambda h, x_n\right) + k\left(\left(n - \frac{7}{8}\lambda\right) h\right) f\left(\frac{7}{8}\lambda h, x_n\right) \right] \right. \\ &\left. +11 \left[k\left(\left(n - \frac{3}{8}\lambda\right) h\right) f\left(\frac{3}{8}\lambda h, x_n\right) + k\left(\left(n - \frac{5}{8}\lambda\right) h\right) f\left(\frac{5}{8}\lambda h, x_n\right) \right] \right\}. \end{aligned}$$

This again gives an error bound of the form $O((\lambda h)^5) = O(h^5)$, because λ is uniformly bounded if we let $h \to 0$ with Q and X remaining fixed. The additional complexity introduced by the necessity to calculate more function values of k remains bounded by $O(1)$ for each n, and hence, by $O(X)$ for the entire interval $[0, X]$, which does not change the complexity of the full algorithm.

In the last subcase still open, when $\lambda \geq 4Q^\sigma$, we decompose the remaining integral once again and write

$$\int_{Q^\sigma T}^{x_n} k(u) f_{x_n}(u)\, du = \left(\int_{x_{Q^\sigma S}}^{x_{Q^\sigma(S+\gamma)}} + \int_{x_{Q^\sigma(S+\gamma)}}^{x_n} \right) k(u) f_{x_n}(u)\, du,$$

where γ is that integer defined by $Q^\sigma(S+\gamma) \leq n < Q^\sigma(S+\gamma+1)$, i.e., $\gamma = \lfloor Q^{-\sigma}n - S \rfloor = \lfloor Q^{-\sigma}\lambda \rfloor \geq 4$. Here the length of the first integration interval $[x_{Q^\sigma S}, x_{Q^\sigma(S+\gamma)}]$ is $\gamma Q^\sigma h$, *viz.*, an integer multiple of $Q^\sigma h$, and so we use the Laplace method with step size $Q^\sigma h$ (γ nodes) to approximate it. This is the concept already used in Algorithm 1, and it turns out that no additional kernel evaluations are required here. For the last interval $[x_{Q^\sigma(S+\gamma)}, x_n]$, we once again resort to the four-point MacLaurin rule which, as above, affects neither the overall complexity nor the error bound.

The complete scheme is detailed in Algorithm 2 below. It is evident from our description that the memory requirements of the new version are not larger than those of the old version. The numerical experiments reported in Section 5 confirm our statements made above regarding the performance of Algorithm 2 with respect to run time and accuracy. Note also that the previous remark concerning a possible parallelization of Algorithm 1 applies to Algorithm 2 as well.

ALGORITHM 2. IMPROVED VERSION OF FAST CONVOLUTION QUADRATURE.

- Input values:
  - univariate kernel function k,
  - bivariate forcing function f,
  - positive real numbers X and T,
  - integers $S > 3$ and $Q > 1$,
- Output values:
  - array $\mathbf{q} = \{q_0, q_1, q_2, \ldots, q_N\}^T$ where $q_n \approx \int_0^{x_n} k(x_n - y) f(y, x_n)\,dy$.
- Body of the algorithm:

```
(* initialization of auxiliary variables *)
$h := T/S$;
$N := \lceil X/h \rceil$;
$\mu := \max\{1, \lceil \log_Q(X/T) \rceil\}$;
(* create table of required kernel values *)
FOR $j := 1$ TO $QS$ DO
  $\kappa_{1,j} := k((j - 1/2)h)$;
FOR $i := 2$ TO $\mu$ DO
  FOR $j := 1$ TO $(Q - 1)S$ DO
    $\kappa_{i,j} := k((T + (j - 1/2)h)Q^{i-1})$;
(* determine weights of Laplace quadratures with 4, 5, 6 or 7 nodes *)
FOR $j := 4$ TO 7 DO
  FOR $i := 1$ TO $j$ DO
    $\ell_{i,j} := 1$;
  $\ell_{1,j} := \ell_{1,j} + 703/5760$;
  $\ell_{2,j} := \ell_{2,j} - 463/1920$;
  $\ell_{3,j} := \ell_{3,j} + 101/640$;
  $\ell_{4,j} := \ell_{4,j} - 223/5760$;
  $\ell_{j,j} := \ell_{j,j} + 703/5760$;
  $\ell_{j-1,j} := \ell_{j-1,j} - 463/1920$;
  $\ell_{j-2,j} := \ell_{j-2,j} + 101/640$;
  $\ell_{j-3,j} := \ell_{j-3,j} - 223/5760$;
(* determine boundary weights of Laplace quadratures, nodes $\geq 8$ *)
$\tilde{\ell}_1 := 1 + 703/5760$;
$\tilde{\ell}_2 := 1 - 463/1920$;
$\tilde{\ell}_3 := 1 + 101/640$;
$\tilde{\ell}_4 := 1 - 223/5760$;

(* Main part of algorithm *)
$q_0 := 0$;
(* subinterval from 0 to $QT$ *)
IF $\mu = 1$ THEN
  $m := N$;
ELSE
  $m := QS$;
(* MacLaurin approximations for $\int_0^{x_n} k(x_n - y) f(y, x_n)\,dy$, $n = 1, 2, 3$ *)
FOR $n := 1$ TO $\min\{3, m\}$ DO
  $q_n := nh\,\{13[k(7nh/8) f(nh/8, nh) + k(nh/8) f(7nh/8, nh)]$
        $+11[k(5nh/8) f(3nh/8, nh) + k(3nh/8) f(5nh/8, nh)]\}/48$;
(* Laplace approximations for $\int_0^{x_n} k(x_n - y) f(y, x_n)\,dy$, $n = 4, 5, \ldots, m$ *)
FOR $n := 4$ TO $\min\{7, m\}$ DO
```

```
    $q_n := h\sum_{j=1}^{n} \ell_{j,n}\kappa_{1,j} f((n-j+1/2)h, nh)$;
FOR $n := 8$ TO $m$ DO
    $q_n := h\sum_{j=1}^{4} \tilde{\ell}_j\kappa_{1,j} f((n-j+1/2)h, nh)$
                $+h\sum_{j=5}^{n-4} \kappa_{1,j} f((n-j+1/2)h, nh)$
                $+h\sum_{j=n-3}^{n} \tilde{\ell}_{n+1-j}\kappa_{1,j} f((n-j+1/2)h, nh)$;
(* remaining subintervals, $n = m+1, m+2, \dots, N$ *)
FOR $n := m+1$ TO $N$ DO
    $\sigma := \lceil \log_Q(n/S) \rceil - 1$;
    (* determine contributions from $\int_0^{Q^\sigma T} k(x_n - y) f(y, x_n)\,\mathrm{d}y$ *)
    $q_n := h\sum_{j=1}^{4} \tilde{\ell}_j\kappa_{1,j} f((n-j+1/2)h, nh)$
      $+h\sum_{j=5}^{QS-4} \kappa_{1,j} f((n-j+1/2)h, nh)$
      $+h\sum_{j=QS-3}^{QS} \tilde{\ell}_{QS+1-j}\kappa_{1,j} f((n-j+1/2)h, nh)$
      $+h\sum_{i=1}^{\sigma-1} Q^i \left\{\sum_{j=1}^{4} \tilde{\ell}_j\kappa_{i+1,j} f(nh - (T + (j-1/2)h)Q^i, nh)\right.$
        $+\sum_{j=5}^{(Q-1)S-4} \kappa_{i+1,j} f(nh - (T + (j-1/2)h)Q^i, nh)$
        $\left.+\sum_{j=(Q-1)S-3}^{(Q-1)S} \tilde{\ell}_{(Q-1)S+1-j}\kappa_{i+1,j} f(nh - (T + (j-1/2)h)Q^i, nh)\right\}$;
(* add contributions from $\int_{Q^\sigma T}^{x_n} k(x_n - y) f(y, x_n)\,\mathrm{d}y$ *)
$\lambda := n - Q^\sigma S$;
IF $\lambda < 4Q^\sigma$ THEN
    (* MacLaurin approximation *)
    $q_n := q_n + \lambda h\,\{13[k((n-\lambda/8)h) f(\lambda h/8, nh)$
      $+k((n-7\lambda/8)h) f(7\lambda h/8, nh)]$
      $+11[k((n-3\lambda/8)h) f(3\lambda h/8, nh)$
      $+k((n-5\lambda/8)h) f(5\lambda h/8, nh)]\}\,/48$;
ELSE
    (* Laplace approximation for $\int_{Q^\sigma T}^{x_{Q^\sigma(S+\gamma)}} k(x_n - y) f(y, x_n)\,\mathrm{d}y$ *)
    $\gamma := \lfloor \lambda / Q^\sigma \rfloor$;
    IF $\gamma \le 7$ THEN
      $q_n := q_n + hQ^\sigma \sum_{j=1}^{\gamma} \ell_{j,\gamma}\kappa_{\sigma+1,j} f(nh - (T + (j-1/2)h)Q^\sigma, nh)$;
    ELSE
      $q_n := q_n + hQ^\sigma \sum_{j=1}^{4} \tilde{\ell}_j\kappa_{\sigma+1,j} f(nh - (T + (j-1/2)h)Q^\sigma, nh)$
        $+hQ^\sigma \sum_{j=5}^{\gamma-4} \kappa_{\sigma+1,j} f(nh - (T + (j-1/2)h)Q^\sigma, nh)$
        $+hQ^\sigma \sum_{j=\gamma-3}^{\gamma} \tilde{\ell}_{\gamma+1-j}\kappa_{\sigma+1,j} f(nh - (T + (j-1/2)h)Q^\sigma, nh)$;
    (* MacLaurin approximation for $\int_{x_{Q^\sigma(S+\gamma)}}^{x_n} k(x_n - y) f(y, x_n)\,\mathrm{d}y$ *)
    $\omega := n - Q^\sigma(S+\gamma)$;
    $q_n := q_n + \omega h\,\{13[k((n-\omega/8)h) f(\omega h/8, nh)$
        $+k((n-7\omega/8)h) f(7\omega h/8, nh)]$
        $+11[k((n-3\omega/8)h) f(3\omega h/8, nh)$
        $+k((n-5\omega/8)h) f(5\omega h/8, nh)]\}\,/48$;
RETURN $\mathbf{q}$.
```

4. A REVERSE COMMUNICATION VERSION

The two versions of our methodology described above return the entire solution history over an uniform grid. This addresses one of two situations that a typical user is likely to be faced with, *viz.*, the forcing function is known over the entire history *a priori*. This is the case when, for example, one solves boundary value problems associated with a material characterization, which are intentionally kept simple so as to be able to solve them readily. In this case, it is appropriate to integrate once (in the sense that one procedure call is made) and to ask for a solution over the entire history, which is what Algorithms 1 and 2 provide.

In a second class of problems, as arise in finite elements, for example, the forcing function is not known *a priori*; instead, it is solved for step-by-step along the solution path. In this example, values for the forcing function are subject to a set of external conditions, *viz.*, a minimization of some potential function, e.g., work done. Here, the history of the forcing function is known up to the current step x_n, but its future values are not yet known. In such situations, it is no longer optimal to have a convolution integration algorithm that returns the entire history of the solution at each step; rather, it is sufficient, and much more economical, if the integrator just returns the solution at the current value for the upper limit of integration from which the total history is then assembled as N separate integrations along an uniform grid of $N+1$ points, where each integration only provides the solution at x_n, $n = 1, 2, \dots, N$.

The reason why we do not know future values of the forcing function is due to the fact that the forcing function is evaluated at Gauss (or quadrature) points in a finite-element grid; whereas, the boundary conditions are applied to a subset of nodal points belonging to that grid, which are different from its quadrature points. What the FE solver does is to suggest a new set of displacements for all nodal points in the grid, which are then interpolated to the quadrature points where stresses are computed. These stresses are then interpolated back to the nodal points as forces where, for example, the conservation of linear momentum is imposed and errors are obtained between the updated solution and the imposed boundary conditions at those nodes where they exist. The solver then perturbs the suggested set of displacements and this process continues until a convergence criterion is satisfied, at which time the global solver advances to the next step along its solution path.

What is known *a priori* is the overall interval of integration $[0, X]$, which can then be discretized into a set of grid points $\{x_0, x_1, x_2, \dots, x_N\}$ where $x_0 = 0$ and $x_{N-1} < X \leq x_N$. If a global finite-element solver is at step x_n, then the local solver (at the Gauss points), which is where the convolution quadrature algorithm would reside, will have a forcing function whose values are known up to point x_n, but are unknown at steps $x_{n+1}, \dots, x_N$; furthermore, the value of the forcing function at x_n will typically vary between iterations of the global solver. As we shall see, we can still apply the fundamental concepts of our original algorithm, but in a more sophisticated way, thus giving rise to a version that is optimized for returning an approximation of the convolution integral over interval $[0, x_n]$.

To state this in a more formal way, as we integrate—for $x_n = nh$, say—we know all values for $f(x_i, x_{n-1})$, $0 \leq x_i < x_{n-1}$, (they have been evaluated by some procedure external to the integrator); $f(x_j, x_j) = 0$, for all j. The forcing function is originally guessed at for the interval $[x_{n-1}, x_n]$, and then sequentially refined via a Newton-Raphson iteration, for example, done external to the integrator. No values for $f(x_i, x_j)$ are known that lie in the future, *viz.*, for $x_j > x_n$. For these reasons we cannot use the existing algorithms efficiently: if we were to use them, then $f(x_i, x_1)$ would be computed N times, $f(x_i, x_2)$ would be computed $N - 1$ times, etc., which does not lead to an efficient scheme.

To start our algorithm for the approximation of an integral with upper bound $x_n = nh$, we again need some special precautions. Specifically, for $n \leq 3$, we proceed as before and use the MacLaurin method. For $n \geq 4$, we again use the Laplace method, but instead of dividing the interval $[0, x_n]$ into n subintervals of length h, as in equation (6), we split it up (starting from the left, i.e., from 0) into $P_n[m]$ subintervals each of length $Q^{m-1}h$ (with m being an integer chosen as large as possible), $P_n[m-1]$ subintervals each of length $Q^{m-2}h$, ... , and P_n [9] subintervals each of length h, where in all cases we choose the coefficients $P_n[i] \geq 4$ so as to be compliant with our Laplace method for integration. How this partitioning is actually accomplished is detailed in Algorithm 3. This logarithmic segmentation of past states differs from that which is employed by Algorithms 1 and 2 in that their parameter T, which is fixed for all subintervals, is now being replaced by $\mathbf{P}_n$, which has distinct values for each subinterval.

ALGORITHM 3. $O(\log X)$ PARTITIONING OF INTEGRATION NODES.

```
• Initialized parameters:
    · integers $N$, $Q \in \{3,5,7,\dots\}$, $S > 3$ and $\ell := \lfloor \log_Q(N/S) \rfloor + 1$,
• Input value:
    · integer $n \in [0,N]$,
• Output values:
    · integer $L_n > 0$,
    · integer array $\mathbf{P}_n = \{P_n[1], P_n[2], \dots, P_n[\ell]\}^{\mathrm{T}}$.
• Body of the algorithm:
    $m := n$ DIV $S$;
    (* p-adic representation of $m$ *)
    IF $m = 0$ THEN
      $L_n := 1$;
      $P_n[1] := 0$;
    ELSE
      $L_n := 0$;
      WHILE $m > 0$ DO
        $L_n := L_n + 1$;
        $P_n[L_n] := m$ MOD $Q$;
        $m := m$ DIV $Q$;
      FOR $i := L_n + 1$ TO $\ell$ DO
        $P_n[i] := 0$;
      (* put into a non-standard p-adic form to ensure that *)
      (* no component $P_n[i]$ is zero for $i = 1,2,\dots,L_n$ *)
      FOR $i := 1$ TO $L_n - 1$ DO
        IF $P_n[i] < 1$ THEN
          $P_n[i] := P_n[i] + Q$;
          $P_n[i+1] := P_n[i+1] - 1$;
      IF $P_n[L_n] = 0$ THEN
        $L_n := L_n - 1$;
    (* convert to number of steps required per subinterval of integration *)
    FOR $i := 1$ TO $L_n$ DO
      $P_n[i] := SP_n[i]$;
    $P_n[1] := P_n[1] + n$ MOD $S$;
    RETURN $L_n$ and $\mathbf{P}_n$.
```

For viscoelastic models, a physical property that proves useful from the point of view of developing our algorithm is that the forcing function f (i.e., strain) can be expressed as a function of another function g (*viz.*, stretch), and as such, can be rewritten as $f(s,t) = F(g(t)/g(s))$ in the 1D case (where g is a scalar), or as $f(s,t) = F(g(t) \cdot g^{-1}(s))$ in the 3D case (with g being a positive-definite tensor). So really, what we need to store is g, not f; specifically, we need to store $g(x_i)$, $i = 0,1,2,\dots,n-1$. This can be done much more economically than a storage of $f(x_i, x_n)$ for relevant values of the arguments. Furthermore, it will be shown that as one moves further along the solution path one can delete some of the earlier $g(x_j)$, since all future samplings will only occur at points other than x_j. Actually, this was a fundamental design criterion for the ensuing method.

So, at each quadrature point in an element, we must store the history of stretch at that quadrature point. From this stored set one can then compute the appropriate strain function (or forcing function) for our viscoelastic convolution integral. This is why we spoke about f (strain) being a function of g (stretch).

The desired method therefore consists of the integration algorithm plus a driver that manages the vector of stored values $g(\cdot)$, external to the integrator, and from these stored data the forcing function f can then be constructed, again external to the integrator. These stored stretches are independent of stress, the dependent variable of integration. The integration algorithm in turn provides information as to which g values can be decimated at location x_i, say, since they will no longer be required of any future step in order to construct an f; consequently, the driver can perform a garbage collection. In this way, we are able to keep the storage requirements for the vector of stretches reasonably small. (Recall that in the versions described in the previous sections there was no need to store the forcing functions or related data at all, but now, due to the high cost of evaluating these functions, a means for their storage must be taken into account.)

A brief analysis reveals that the prior algorithms lead to a highly irregular and uneconomical sampling of the stretch function g that does not allow for an efficient memory management scheme for this data set. Therefore, we have to modify our basic approach in such a way that solves this problem without giving up a substantial amount of efficiency with respect to run time and, ideally, to the storage requirements of the kernel function.

The general idea behind this version of our algorithm is to calculate the integral at a point $x_n = nh$, use this result to compute $g(x_n)$, and then add it to the list of stored g values. After every Q^{th} step, this list will undergo a garbage collection where existing entries that are no longer needed are removed, thereby freeing up memory for future g values to be stored. At the same time, all indexed parameters are updated in preparation for the next integration step. This objective is accomplished in Algorithm 4, which manages the integrator's data base. This procedure of updating is repeated in a sequential manner for $n = 1, 2, \dots, N$, i.e., until we reach the end of the desired interval $[0, X]$.

ALGORITHM 4. DRIVER FOR MANAGING HISTORY VARIABLES.

- Initialized parameters:
  - · integers N, $Q \in \{3, 5, 7, \dots\}$, $S > 3$ and $\ell := S[1 + Q(1 + \lceil \log_Q(N/S) \rceil)]$,
- Imported values from Algorithm 3:
  - · integer array $\mathbf{P}_n$ of length L_n,
- Input values:
  - · field $g(x_n)$,
  - · field array $\mathbf{G} = \{G[0], G[1], G[2], \dots, G[\ell]\}^{\mathsf{T}}$ (* note $G[n] \neq g(x_n)$ *),
- Output value:
  - · field array $\mathbf{G} = \{G[0], G[1], G[2], \dots, G[\ell]\}^{\mathsf{T}}$,
- Managed values:
  - · integers I and n,
  - · integer array $\mathbf{P}_{n-1}$ of length L_{n-1},
  - · field $g(x_{n-1})$.
- Body of the algorithm:

```
(* attach current field to history array *)
IF $I = 1$ THEN
  $G[0] := g(x_{n-1})$;
$G[I] := (g(x_{n-1}) + g(x_n))/2$;
(* restructure history array - garbage collection *)
IF $(n - 1 > S)$ AND $((n - S)$ MOD $(QS) = 0)$ THEN
  FOR $j := 1$ TO $L_n$ DO
    IF $P_n[j] < P_{n-1}[j]$ THEN
      $a := 0$;
      FOR $i := j + 1$ TO $L_n$ DO
        $a := a + P_{n-1}[i]$;
      $b := S$;
```

```
        FOR $i := 1$ TO $j - 1$ DO
          $b := b + P_n[i]$;
        FOR $i := 1$ TO $S$ DO
          $G[a + i] := G[a + Q * i - (Q - 1)$ DIV $2]$;
        FOR $i := 1$ TO $b$ DO
          $G[a + S + i] := G[a + S + i + (Q - 1) * S]$;
RETURN $\mathbf{G}$.

(* Update integrator data base *)
$n := n + 1$;
$g(x_{n-1}) := g(x_n)$;
$L_{n-1} := L_n$;
FOR $i := 1$ TO $L_n$ DO
  $P_{n-1}[i] := P_n[i]$;
CALL Algorithm 3 to get $L_n$ and $\mathbf{P}_n$;
$I := 0$;
FOR $i := 1$ TO $L_n$ DO
  $I := I + P_n[i]$.
```

This relatively straightforward scheme for managing the history of the subordinate function g to forcing function f unfortunately does not have a counterpart for handling the kernel function k. Nevertheless, an efficient method for acquiring kernel function values from a pretabulated array is still achievable. One can devise a variety of methods that meet this objective; we present one such methodology below.

The main idea behind our approach is to return exact values for the kernel whenever function calls are made from the subinterval whose integration step size is h (i.e., where u is close to 0). For all remaining subintervals (where u is far away from 0) it is no longer practical to store exact values for the kernel at all possible locations; instead, we store an array of values captured in a logarithmic manner from which the requested value is then obtained via an interpolation. This becomes a reasonable approach whenever u is far from 0, and is in accordance with the monotonic decay properties that kernel functions possess. Because the interpolation scheme selected (a fourth-order Neville-Aitken interpolator) returns exact values whenever the requested location x coincides with an interpolation node x_j, Algorithm 5 returns exact values for k for all function calls made within the first subinterval of integration whose step size is h. We point out that creation of the array of kernel values (designated as κ in the algorithm below) only needs to be done once, i.e., during the initialization phase of the solver.

ALGORITHM 5. PROCEDURE FOR HANDLING KERNEL FUNCTION CALLS.

- Initialized parameters:
  - · integers N, $Q \in \{3, 5, 7, \dots\}$ and $S > 3$,
  - · real $h > 0$,
  - · univariate kernel function k,
- Input value:
  - · real $x > 0$,
- Output value:
  - · real K.
- Body of the algorithm:

```
    (* initialization of auxiliary variables *)
    $L := \lceil \log_Q(N/S) \rceil$;
    $\ell := LS(Q - 1) + S - 1$;
    (* create array of locations with logarithmic gait *)
    $m := 0$;
```

```
FOR $i := 1$ TO $SQ - 1$ DO
  $m := m + 1$;
  $X[m] := (i - 1/2)h$;
$y := (SQ - 1)h$;
FOR $i := 2$ TO $L$ DO
  FOR $j := 1$ TO $S(Q - 1)$ DO
    $m := m + 1$;
    $X[m] := y + (j - 1/2)Q^{i-1}h$;
  $y := y + S(Q - 1)Q^{i-1}h$;
(* create associated array of kernel values *)
FOR $i := 1$ TO $\ell$ DO
  $\boldsymbol{\kappa}[i] := k(X[i])$;

(* Main part of algorithm *)
(* locate $x$ in $\mathbf{X}$ such that $X[lo] < x \leq X[hi]$ *)
$lo := 1$;
$hi := \ell$;
REPEAT
  $mid := (lo + hi)$ DIV 2;
  IF $x > X[mid]$ THEN
    $lo := mid$;
  ELSE
    $hi := mid$;
UNTIL $lo = hi - 1$;
(* set indexer for interpolation *)
IF $lo < 3$ THEN
  $m := 1$;
ELSIF $lo < \ell - 2$ THEN
  $m := lo - 1$;
ELSE
  $m := \ell - 3$;
(* Neville-Aitken interpolation *)
$K_{12} := \{(x - X[m+1])\boldsymbol{\kappa}[m] - (x - X[m])\boldsymbol{\kappa}[m+1]\}/$
$(X[m] - X[m+1])$;
$K_{23} := \{(x - X[m+2])\boldsymbol{\kappa}[m+1] - (x - X[m+1])\boldsymbol{\kappa}[m+2]\}/$
$(X[m+1] - X[m+2])$;
$K_{34} := \{(x - X[m+3])\boldsymbol{\kappa}[m+2] - (x - X[m+2])\boldsymbol{\kappa}[m+3]\}/$
$(X[m+2] - X[m+3])$;
$K_{123} := \{(x - X[m+2])K_{12} - (x - X[m])K_{23}\}/$
$(X[m] - X[m+2])$;
$K_{234} := \{(x - X[m+3])K_{23} - (x - X[m+1])K_{34}\}/$
$(X[m+1] - X[m+3])$;
$K := \{(x - X[m+3])K_{123} - (x - X[m])K_{234}\}/$
$(X[m] - X[m+3])$;
RETURN $K$.
```

In a finite-element setting, all elements of the same material type will share a common kernel function, thereby allowing for an economy of memory resources; whereas, every Gauss point of every element of a given material type will need to store its own history array associated with the forcing function.

The final version of our algorithm uses a reverse communication concept in the sense that data are passed between the forcing function and the integrator in both directions; specifically, the forcing function $f(x_i, x_n)$ is assumed to take on the form $F(g(x_i), g(x_n))$ wherein the subordinate function $g(x_i)$ is managed by Algorithm 4. Like the prior algorithms, the Laplace quadratures are computed just once, during the initialization phase of the solver.

ALGORITHM 6. FINAL VERSION OF FAST CONVOLUTION QUADRATURE.

```
• Input values:
    · integers $Q \in \{3, 5, 7, \dots\}$ and $S > 3$,
    · positive real numbers $T$ and $X$,
    · fields $g(0)$ and $g(x_n)$,
    · univariate kernel function $k$,
    · bivariate forcing function $F(g(x_i), g(x_n))$,
• Imported values:
    · integers $I$ and $n$ managed by Algorithm 4,
    · field $g(x_{n-1})$ managed by Algorithm 4,
    · p-adic like array $\mathbf{P}_{n-1}$ and its length $L_{n-1}$ managed by Algorithm 4,
    · history array $G$ from Algorithm 4,
    · kernel $K$ from Algorithm 5,
• Output value:
    · real $q_n \approx \int_0^{x_n} k(x_n - y) f(y, x_n) \mathrm{d}y$.
• Body of the algorithm:
        (* initialization of auxiliary variables *)
        $h := T/S$;
        $m := I$;
        $N := \lceil X/h \rceil$;
        (* determine weights of Laplace quadratures with 4, 5, 6 or 7 nodes *)
        FOR $j := 4$ TO 7 DO
          FOR $i := 1$ TO $j$ DO
            $\ell_{i,j} := 1$;
          $\ell_{1,j} := \ell_{1,j} + 703/5760$;
          $\ell_{2,j} := \ell_{2,j} - 463/1920$;
          $\ell_{3,j} := \ell_{3,j} + 101/640$;
          $\ell_{4,j} := \ell_{4,j} - 223/5760$;
          $\ell_{j,j} := \ell_{j,j} + 703/5760$;
          $\ell_{j-1,j} := \ell_{j-1,j} - 463/1920$;
          $\ell_{j-2,j} := \ell_{j-2,j} + 101/640$;
          $\ell_{j-3,j} := \ell_{j-3,j} - 223/5760$;
        (* determine boundary weights of Laplace quadratures, nodes $\geq 8$ *)
        $\tilde{\ell}_1 := 1 + 703/5760$;
        $\tilde{\ell}_2 := 1 - 463/1920$;
        $\tilde{\ell}_3 := 1 + 101/640$;
        $\tilde{\ell}_4 := 1 - 223/5760$;

        (* Main part of algorithm *)
        IF $n < 4$ THEN
          (* MacLaurin approximations for $\int_0^{x_n} k(x_n - y) f(y, x_n)\, \mathrm{d}y$, $n \leq 3$ *)
          IF $n = 1$ THEN
            $g_1 := g(x_{n-1}) + (g(x_n) - g(x_{n-1}))/8$;
            $g_2 := g(x_{n-1}) + 3(g(x_n) - g(x_{n-1}))/8$;
            $g_3 := g(x_{n-1}) + 5(g(x_n) - g(x_{n-1}))/8$;
            $g_4 := g(x_{n-1}) + 7(g(x_n) - g(x_{n-1}))/8$;
```

```
  ELSEIF $n = 2$ THEN
    $g_1 := G[0] + (G[1] - G[0])/2;$
    $g_2 := G[1] + (g(x_{n-1}) - G[1])/2;$
    $g_3 := g(x_{n-1}) + (g(x_n) - g(x_{n-1}))/4;$
    $g_4 := g(x_{n-1}) + 3(g(x_n) - g(x_{n-1}))/4;$
  ELSE
    $g_1 := G[0] + 3(G[1] - G[0])/4;$
    $g_2 := G[1] + 5(G[2] - G[1])/8;$
    $g_3 := G[2] + 3(g(x_{n-1}) - G[2])/4;$
    $g_4 := g(x_{n-1}) + 5(g(x_n) - g(x_{n-1}))/8;$
  $q_n := nh\,\{13[k(7nh/8)F(g_1, g(x_n)) + k(nh/8)F(g_4, g(x_n))]$
             $+11[k(5nh/8)F(g_2, g(x_n)) + k(3nh/8)F(g_3, g(x_n))]\}\,/48;$
ELSE
  (* Laplace approximations for $\int_0^{x_n} k(x_n - y)f(y, x_n)\,\mathrm{d}y$, $n > 3$ *)
  (* integrate over first subinterval with step size $h$ *)
  IF $P_{n-1}[1] < 7$ THEN
    $q_n := h\ell_{1,P_{n-1}[1]+1}K(h/2)F((g(x_{n-1}) + g(x_n))/2, g(x_n));$
  ELSE
    $q_n := h\tilde{\ell}_1 K(h/2)F((g(x_{n-1}) + g(x_n))/2, g(x_n));$
  IF $P_{n-1}[1] < 7$ THEN
    FOR $i := 2$ TO $P_{n-1}[1] + 1$ DO
      $q_n := q_n + h\ell_{i,P_{n-1}[1]+1}K((i - 1/2)h)F(G[m], g(x_n));$
      $m := m - 1;$
  ELSE
    FOR $i := 2$ TO 4 DO
      $q_n := q_n + h\tilde{\ell}_i K((i - 1/2)h)F(G[m], g(x_n));$
      $m := m - 1;$
    FOR $i := 5$ TO $P_{n-1}[1] - 3$ DO
      $q_n := q_n + hK((i - 1/2)h)F(G[m], g(x_n));$
      $m := m - 1;$
    FOR $i := P_{n-1}[1] - 2$ TO $P_{n-1}[1] + 1$ DO
      $q_n := q_n + h\tilde{\ell}_{P_{n-1}[1]+2-i}K((i - 1/2)h)F(G[m], g(x_n));$
      $m := m - 1;$
  (* integrate over remaining subintervals with step sizes $Q^{j-1}h$ *)
  $y := (P_{n-1}[1] + 1)h;$
  FOR $j := 2$ TO $L_{n-1}$ DO
    IF $P_{n-1}[j] < 8$ THEN
      FOR $i := 1$ TO $P_{n-1}[j]$ DO
        $q_n := q_n + Q^{j-1}h\ell_{i,P_{n-1}[j]}K(y + (i - 1/2)Q^{j-1}h)F(G[m], g(x_n));$
        $m := m - 1;$
    ELSE
      FOR $i := 1$ TO 4 DO
        $q_n := q_n + Q^{j-1}h\tilde{\ell}_i K(y + (i - 1/2)Q^{j-1}h)F(G[m], g(x_n));$
        $m := m - 1;$
      FOR $i := 5$ TO $P_{n-1}[j] - 4$ DO
        $q_n := q_n + Q^{j-1}hK(y + (i - 1/2)Q^{j-1}h)F(G[m], g(x_n));$
        $m := m - 1;$
      FOR $i := P_{n-1}[j] - 3$ TO $P_{n-1}[j]$ DO
        $q_n := q_n + Q^{j-1}h\tilde{\ell}_{P_{n-1}[j]+1-i}K(y + (i - 1/2)Q^{j-1}h)F(G[m], g(x_n));$
        $m := m - 1;$
  $y := y + P_{n-1}[j]Q^{j-1}h;$
RETURN $q_n$.
```

- Test for convergence of $g(x_n)$, if appropriate, and upon convergence call Algorithm 4 to update the history vector $\mathbf{G}$. Vector $\mathbf{G}$ must be updated before this integrator is called again for the next step, $n + 1$.

Some numerical examples are provided in the following section.

5. NUMERICAL EXAMPLES

In this last section, we provide some numerical examples illustrating the performance of the algorithms developed above. The first example has a known analytic solution, so it can be used to demonstrate the accuracy of the three algorithms. The second example is the application that motivated the development of our algorithms, *viz.*, fractional-order viscoelasticity (FOV) [1].

5.1. Analytic Result

Here, we investigate a simple but nevertheless typical problem where the exact value of a convolution integral is known. The problem is

$$k(x) = \exp(-x), \qquad f(t,x) = \sin(t - x), \tag{8}$$

whose solution is

$$\int_0^x k(x-t) f(t,x)\,dt = \frac{1}{2}\left(\exp(-x)(\sin x + \cos x) - 1\right).$$

The characteristic time was chosen to be $T = 1$. Some typical results for Algorithm 1 with various choices of the parameters are given in Tables 1–3.

Table 1. Results for Algorithm 1, applied to problem (8), with step size $h = 0.04$ and $Q = 4$.

X	Run Time	Max. |Error|
4	0.2 s	4.39(−4)
8	0.7 s	2.71(−3)
16	2.2 s	2.71(−3)
32	6.0 s	2.71(−3)
64	15.2 s	2.71(−3)

Table 2. Results for Algorithm 1, applied to problem (8), with step size $h = 0.02$ and $Q = 4$.

X	Run Time	Max. |Error|
4	0.8 s	1.15(−4)
8	2.7 s	1.43(−3)
16	8.5 s	1.43(−3)
32	23.0 s	1.43(−3)
64	58.7 s	1.43(−3)

Table 3. Results for Algorithm 1, applied to problem (8), with step size $h = 0.04$ and $Q = 6$.

X	Run Time	Max. |Error|
4	0.2 s	4.39(−4)
8	0.8 s	4.39(−4)
16	2.5 s	4.39(−4)
32	6.8 s	4.39(−4)
64	18.0 s	4.39(−4)

Table 4. Results for Algorithm 2, applied to problem (8), with step size $h = 0.04$ and $Q = 4$.

X	Run Time	Max. |Error|
4	0.2 s	5.58(−9)
8	0.8 s	3.66(−7)
16	2.2 s	3.66(−7)
32	6.2 s	3.66(−7)
64	15.7 s	3.66(−7)

Table 5. Results for Algorithm 2, applied to problem (8), with step size $h = 0.02$ and $Q = 4$.

X	Run Time	Max. |Error|
4	0.8 s	1.615(−10)
8	2.8 s	1.24(−8)
16	8.4 s	1.24(−8)
32	23.1 s	1.24(−8)
64	60.1 s	1.24(−8)

Table 6. Results for Algorithm 2, applied to problem (8), with step size $h = 0.04$ and $Q = 6$.

X	Run Time	Max. |Error|
4	0.2 s	5.57(−9)
8	0.8 s	8.35(−8)
16	2.4 s	8.35(−8)
32	6.7 s	8.35(−8)
64	18.1 s	8.35(−8)

Table 7. Results for Algorithm 6, applied to problem (8), with step size $h = 0.04$ and $Q = 5$.

X	Run Time	Max. |Error|	Mean |Error|
400	1 s	8.04(−5)	1.15(−7)
800	4 s	8.63(−5)	9.03(−8)
1600	7 s	1.00(−4)	8.17(−8)
3200	17 s	1.22(−4)	9.19(−8)
6400	36 s	1.71(−4)	1.21(−7)

The effect of the quality parameter Q on run time and accuracy is clearly exhibited. Moreover, we find that, for increasing X, the run time grows at a rate significantly slower than the $O(X^2)$, which a standard algorithm would exhibit; the theoretical rate $O(X \log X)$ is reproduced, as expected.

For the purpose of comparison, we have repeated the calculations for the same example using Algorithm 2. These results are presented in Tables 4–6. The accuracy is indeed significantly higher, yet the more complicated structure of the algorithm leads to only slightly longer run times.

The computations for Tables 1–6 were performed on a 1.4 GHz Duron based PC running MATHEMATICA 4.2 for Linux.

Algorithm 6 along with dependent Algorithms 3–5 were coded in the Active Oberon programming language and executed on a 2 GHz PC running the Bluebottle operating system. Conditions similar to those of Tables 3 and 6 were imposed, except that X was taken to be larger by two orders in magnitude. This means that there were between 10,000 and 160,000 integration steps for the cases reported on in Table 7, thereby clearly demonstrating the $O(X \log X)$ run-time characteristic of this method. The mean error was observed to be several orders in magnitude smaller than the maximum error. The accuracy of Algorithm 6 was found to be comparable to that of Algorithm 1.

5.2. Fractional-Order Viscoelasticity

The linear theory of viscoelasticity, as it applies to a preconditioned soft tissue in a state of dynamic equilibrium, has a constitutive description of

$$\sigma(t) = E_\infty \epsilon(0,t) + (E_0 - E_\infty) \int_0^t M(t-s)\epsilon(s,t)\,\mathrm{d}s, \tag{9}$$

where stress σ responds to a history in strain ϵ. Strain is in turn a function of stretch $\lambda(t_1,t_2) = \ell(t_2)/\ell(t_1)$ in that $\epsilon(t_1,t_2) = \varepsilon(\lambda(t_1,t_2))$, wherein $\ell(t)$ is the length of a gage section at time t. We point out that $\lambda(s,t) = \lambda(0,t)/\lambda(0,s)$. The material parameters are: E_∞ (≥ 0) is the rubbery (or equilibrium) modulus, E_0 ($> E_\infty$) is the glassy (or dynamic) modulus, and M (≥ 0) is the so-called memory function, which must satisfy $M(t_2) < M(t_1)$, $\forall t_2 > t_1 \geq 0$ in order to be in accordance with the second law of thermodynamics.

FOV is a special case of equation (9) signified by the memory function in equation (2), which introduces two additional material constants via M: a characteristic time τ (> 0) for relaxation, and a fractional order α ($0 < \alpha < 1$) of evolution. For synthetic polymers undergoing infinitesimal deformations, strain can be quantified by the linear function $\epsilon(t_1,t_2) = \lambda(t_1,t_2) - 1$. However, in soft-tissue mechanics (specifically, in the modeling of collagenous tissues) the strain function needs to be nonlinear. Fung [12] calls such a theory quasilinear since the viscoelastic contribution is linear while the elastic contribution is nonlinear. The formula for $\epsilon(t_1,t_2) = \varepsilon(\lambda(t_1,t_2))$ that we employ is [1,13],

$$\epsilon(t_1,t_2) = \begin{cases} 0, & \text{if } \lambda(t_1,t_2) \leq 1, \\ [\lambda(t_1,t_2) - 1]^n / [n(\lambda_c - 1)^{n-1}], & \text{if } 1 \leq \lambda(t_1,t_2) \leq \lambda_c, \\ \lambda(t_1,t_2) - [(n-1)\lambda_c + 1]/n, & \text{if } \lambda_c \leq \lambda(t_1,t_2), \end{cases} \tag{10}$$

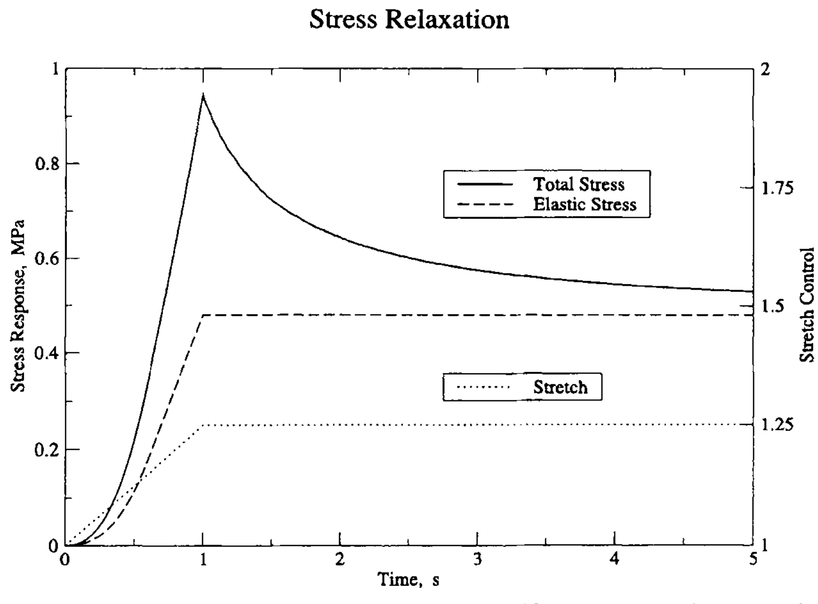

Figure 2. A typical stress-relaxation response of the FOV material model obtained from Algorithm 2 using $S = 4$ and $Q = 6$.

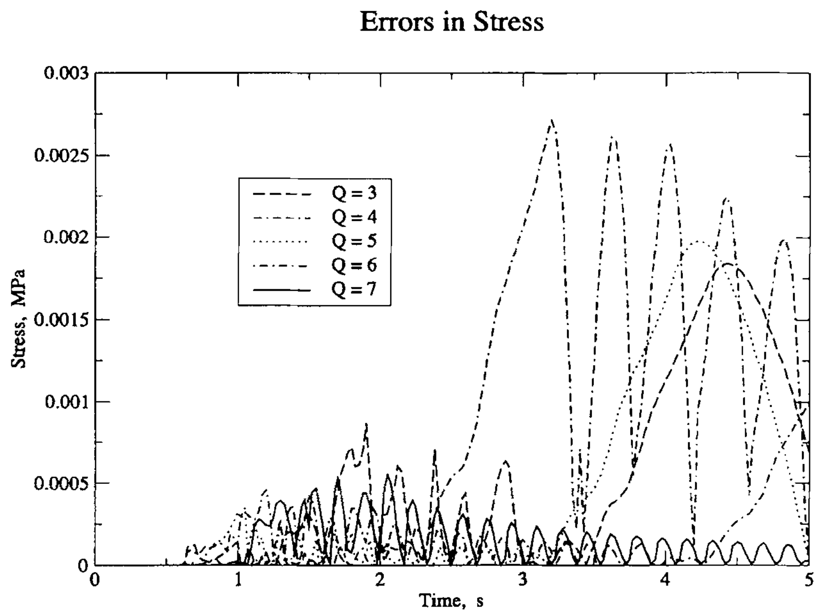

Figure 3. Absolute errors in the predicted stress-relaxation response of the FOV model obtained from Algorithm 2 using $S = 4$ for various values of Q.

which introduces constants n (≥ 1) and λ_c (≥ 1) into the material model. The first case above corresponds to compressive strains. Because collagen fibers buckle under compressive loads, the strain function is taken to be zero there. The second case models what is known as the toe region, which is highly nonlinear. The third case represents a linear response that occurs after the toe region. This expression for strain is a C^1 function of stretch.

Equations (2), (9), and (10) quantify FOV in 1D. Figure 2 presents a typical stress-relaxation plot obtained using Algorithm 2, where the material constants employed were: $E_\infty = 3\,\text{MPa}$, $E_0 = 100\,\text{MPa}$, $\lambda_c = 1.15$, $n = 2.5$, $\tau = 0.1\,\text{s}$ and $\alpha = 0.33$. The step size was $h = 0.025\,\text{s}$. The total response was obtained from equation (9), while the elastic response was obtained from just the first term on the right-hand side of this equation. Absolute errors for solutions acquired at given values of quality parameter Q are plotted in Figure 3. Because the analytic solution to this boundary-value problem is not known to us, the solution obtained with $Q = 200$ was used as the 'exact' result for purposes of constructing these errors. The influence of Q on accuracy is made clear in this figure.

REFERENCES

1. A.D. Freed and K. Diethelm, A K-BKZ formulation for soft-tissue viscoelasticity, *Biomechan. Model. Mechanobiol.* (to appear).
2. A. Erdélyi, W. Magnus, F. Oberhettinger and F.G. Tricomi, *Higher Transcendental Functions, Volume 3*, McGraw-Hill, New York, (1954).
3. R. Gorenflo, J. Loutchko and Yu. Luchko, Computation of the Mittag-Leffler function $E_{\alpha,\beta}(z)$ and its derivative, *Fract. Calc. Appl. Anal.* **5**, 491–518, (2002); *Fract. Calc. Appl. Anal.* **6**, 111–112 (Corrections), (2003).
4. K. Diethelm, N.J. Ford, A.D. Freed and Yu. Luchko, Algorithms for the fractional calculus: A selection of numerical methods, *Comput. Methods Appl. Mech. Eng.* **194**, 743–773, (2005).
5. I. Podlubny, *Fractional Differential Equations*, Academic Press, San Diego, CA, (1999).
6. R. Gorenflo and F. Mainardi, Fractional oscillations and Mittag-Leffler functions, In Proceedings of the *International Workshop on the Recent Advances in Applied Mathematics (RAAM '96)*, pp. 193–208, Kuwait University, Department of Mathematics and Computer Science, Kuwait, (1996).
7. J.C. Simo and T.J.R. Hughes, Computational inelasticity, In *Interdisciplinary Applied Mathematics, Volume 7*, Springer-Verlag, New York, (1998).
8. N.J. Ford and A.C. Simpson, The numerical solution of fractional differential equations: Speed versus accuracy, *Numer. Algorithms* **26**, 333–346, (2001).
9. H. Braß, *Quadraturverfahren*, Vandenhoeck and Ruprecht, Göttingen, (1977).

10. H. Braß, On the principle of avoiding the singularity in quadrature, *Z. Angew. Math. Mech.* **75**, S617–S618, (1995).
11. P. Rabinowitz, On avoiding the singularity in the numerical integration of proper integrals, *BIT* **19**, 104–110, (1979).
12. Y.-C. Fung, *Biomechanics: Mechanical properties of living tissues*, Second Edition, Springer, New York, (1993).
13. T.C. Doehring, A.D. Freed, E.O. Carew and I. Vesely, Fractional order viscoelasticity of the aortic valve cusp: An alternative to quasilinear viscoelasticity, *J. Biomech. Engr.* **127**, 700–708, (2005).