Computer Physics Communications ••• (••••) •••–•••

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

# A numerical differentiation library exploiting parallel architectures ☆

C. Voglis [a,*], P.E. Hadjidoukas [a], I.E. Lagaris [a], D.G. Papageorgiou [b]

[a] Department of Computer Science, University of Ioannina, P.O. Box 1186, 45110 Ioannina, Greece
[b] Department of Materials Science and Engineering, University of Ioannina, P.O. Box 1186, 45110 Ioannina, Greece

ARTICLE INFO

*Article history:*
Received 7 July 2008
Received in revised form 15 January 2009
Accepted 10 February 2009
Available online xxxx

*PACS:*
02.60.Jh
02.60.Pn
02.70.Bf

*Keywords:*
Numerical differentiation
Finite differences
Optimization
Nonlinear equations
OpenMP
MPI
Parallel processing
Distributed computing

ABSTRACT

We present a software library for numerically estimating first and second order partial derivatives of a function by finite differencing. Various truncation schemes are offered resulting in corresponding formulas that are accurate to order $O(h)$, $O(h^2)$, and $O(h^4)$, $h$ being the differencing step. The derivatives are calculated via forward, backward and central differences. Care has been taken that only feasible points are used in the case where bound constraints are imposed on the variables. The Hessian may be approximated either from function or from gradient values. There are three versions of the software: a sequential version, an OpenMP version for shared memory architectures and an MPI version for distributed systems (clusters). The parallel versions exploit the multiprocessing capability offered by computer clusters, as well as modern multi-core systems and due to the independent character of the derivative computation, the speedup scales almost linearly with the number of available processors/cores.

**Program summary**

*Program title:* NDL (Numerical Differentiation Library)
*Catalogue identifier:* AEDG_v1_0
*Program summary URL:* http://cpc.cs.qub.ac.uk/summaries/AEDG_v1_0.html
*Program obtainable from:* CPC Program Library, Queen's University, Belfast, N. Ireland
*Licensing provisions:* Standard CPC licence, http://cpc.cs.qub.ac.uk/licence/licence.html
*No. of lines in distributed program, including test data, etc.:* 73 030
*No. of bytes in distributed program, including test data, etc.:* 630 876
*Distribution format:* tar.gz
*Programming language:* ANSI FORTRAN-77, ANSI C, MPI, OPENMP
*Computer:* Distributed systems (clusters), shared memory systems
*Operating system:* Linux, Solaris
*Has the code been vectorised or parallelized?:* Yes
*RAM:* The library uses $O(N)$ internal storage, $N$ being the dimension of the problem
*Classification:* 4.9, 4.14, 6.5
*Nature of problem:* The numerical estimation of derivatives at several accuracy levels is a common requirement in many computational tasks, such as optimization, solution of nonlinear systems, etc. The parallel implementation that exploits systems with multiple CPUs is very important for large scale and computationally expensive problems.
*Solution method:* Finite differencing is used with carefully chosen step that minimizes the sum of the truncation and round-off errors. The parallel versions employ both OpenMP and MPI libraries.
*Restrictions:* The library uses only double precision arithmetic.
*Unusual features:* The software takes into account bound constraints, in the sense that only feasible points are used to evaluate the derivatives, and given the level of the desired accuracy, the proper formula is automatically employed.
*Running time:* Running time depends on the function's complexity. The test run took 15 ms for the serial distribution, 0.6 s for the OpenMP and 4.2 s for the MPI parallel distribution on 2 processors.



☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author.
*E-mail address:* voglis@cs.uoi.gr (C. Voglis).

0010-4655/$ – see front matter 
doi:10.1016/j.cpc.2009.02.004

## 1. Introduction

Estimating derivatives is a common subtask in many applications. For example, the majority of optimization methods employ the gradient and/or the Hessian of the objective function [1–5], while for the solution of nonlinear systems the Jacobian matrix [4,5] is required.

There are several methods for calculating derivatives. Methods that manipulate symbolically analytic expressions and provide the derivative in closed form [6,7], are exact but of limited applicability. Automatic differentiation (AD) [8] is a promising alternative that has been rather recently developed. Given the source code that implements a function, the AD software creates the code for its derivative exploiting repeated application of the chain rule. Although this is a powerful technique, the complexity of the process is considerable and sometimes the gains are marginal.

In a growing number of real world applications in science and engineering, the underlying functions are represented by large and complicated computer codes and the user may find it difficult or almost impossible to follow the original program (if it is available in source form) and develop the corresponding code for the derivative. An alternative is offered by finite differencing, where the derivatives are approximated from function values at suitably chosen points. The corresponding formulae may be derived in a number of ways, and a detailed classification is given in [9]. Recently Khan and Ohba [10] reported formulas based on Taylor expansion, suitable for highly oscillating functions and Li [11] extended this method by employing equally and unequally spaced points to estimate derivatives of arbitrary order.

Derivative estimation via finite differencing is simple but computationally expensive, since it requires a number of function evaluations. Moreover there are several applications where the time for a single function call is substantial. For example, the determination of stable molecular conformations via "molecular mechanics" [12,13], ab initio quantum mechanical structure calculations [13,14], construction of potentials for the atomistic simulation of materials [15,16], the construction of nuclear forces [17], phase-shift analysis from nucleon–nucleon scattering data [18], the solution of partial differential equations via Galerkin type of methods [19–21] and all cases where the function's value is a result of a simulation. In such cases parallelism could play a key role in accelerating the process. Assuming that we have an ample number of available processors, the cost for the derivative calculation may become almost equal to that of a single function call if parallel processing is employed.

In the literature there exist several software packages for estimating derivatives numerically. In the GSL library [22] the authors provide a five-point rule for central differences and four-point rules for forward/backward differences using an adaptive scheme to achieve the desired accuracy. NAG library [23] provides subroutine `D04AAF` that calculates derivatives up to 14th order for a univariate function using an extended Neville's algorithm [24,25]. In the book of Mathews [26] three subroutines are offered for differentiating univariate functions, one of which takes in account upper and lower variable bounds. The `numDeriv` package [27] differentiates multivariate functions using Richardson extrapolation and calculates the Jacobian and Hessian matrices. Package `DIFF` [28] differentiates univariate functions up to the third order using Neville's process. Mathematica [6] provides a command (`ND`) for differentiation up to any order, using Richardson's extrapolation to the limit. Package `LNIDIF` [29] calculates first and second order derivatives having non-uniformly spaced points, out to three dimensions. IMSL library provides subroutine `DERIV` [30] for calculating up to third order derivatives of univariate functions. From the above only [27] supports multivariate functions, while [29] is limited to three-dimensional functions. Note that the implementation of these packages is sequential, hence they cannot exploit the advantage offered by the architecture of distributed or parallel systems. This capability is important and that is why even automatic differentiation packages, such as [31], have followed the parallel implementation path.

The software described in this article, supports multivariate functions, respects variable bounds, offers several prescribed accuracy levels, and is implemented using state of the art parallel techniques in the framework of MPI [32] and OpenMP [33] platforms. The rest of the paper is organized as follows. In Section 2 we provide derivative formulas. In Section 3 we briefly sketch the parallelization strategy followed, and in Section 4 we describe the interface of the subroutines included in the library. Installation instructions are given in Section 5. Finally in Section 6 we present and analyze numerical experiments that concern both the accuracy and the efficiency of the software. In the software's distribution extensive test runs are included.

## 2. Derivative formulae

In the following, we present derivative formulae using forward differences (FD), backward differences (BD), and central differences (CD), for several levels of accuracy. The formulae for FD and BD are common. FD use positive while BD use negative differencing step. The reason for this variety is to handle the bound constraints case, i.e. when $x_i \in [a_i, b_i]$. Consider the case where the derivative at the bound $x_i = a_i$ is desired. Then the proper formula to use is from the forward difference class, since otherwise infeasible $x$-values will be used, which may lead to numerical errors. (Imagine, for example, a quantity under the square root sign that becomes negative when $x$ is infeasible.) The software takes into account bound constraints, in the sense that only feasible points are used to evaluate the derivatives, and given the level of the desired accuracy, the proper formula is automatically employed. In the case where there are no bounds or when the bounds are not violated, the default selection is the FD formula for $O(h)$ and the CD formulae for $O(h^2)$ and $O(h^4)$.

### 2.1. First order derivatives

The formulae of this section are used to evaluate the gradient of a scalar function and the Jacobian of a vector function. Each formula uses a different stepsize that is determined by approximately minimizing the total (truncation and roundoff) error [3]. In what follows we denote by $\eta$ the relative error in the calculation of $f$, that defaults to the machine precision in absence of such information.

(1) Accurate to order $O(h)$
$h = \pm\sqrt{\eta}\max\{1, |x|\}$.
FD/BD formula:

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x)}{h}. \tag{1}$$

Please cite this article in press as: C. Voglis et al., A numerical differentiation library exploiting parallel architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.02.004

(2) Accurate to order $O(h^2)$
$h = \pm\eta^{1/3}\max\{1, |x|\}$.
(a) CD formula:

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x-h)}{2h}. \tag{2}$$

(b) FD/BD formula:

$$\frac{df(x)}{dx} \approx \frac{4f(x+h) - 3f(x) - f(x+2h)}{2h}. \tag{3}$$

(3) Accurate to order $O(h^4)$
$h = \pm\eta^{1/5}\max\{1, |x|\}$.
(a) CD formula:

$$\frac{df(x)}{dx} \approx \frac{1}{3}\left(4\frac{f(x+h) - f(x-h)}{2h} - \frac{f(x+2h) - f(x-2h)}{4h}\right). \tag{4}$$

(b) FD/BD formula:

$$\frac{df(x)}{dx} \approx \frac{1}{21}\left(64\frac{f(x+h) - f(x)}{h} - 56\frac{f(x+2h) - f(x)}{2h} + 14\frac{f(x+4h) - f(x)}{4h} - \frac{f(x+8h) - f(x)}{8h}\right). \tag{5}$$

The gradient of a function $f(x)$, $x \in R^N$ is calculated as:

$$\frac{\partial f(x)}{\partial x_i} = \left.\frac{df(x+\alpha e_i)}{d\alpha}\right|_{\alpha=0}, \quad \forall i = 1, 2, \ldots, N,$$

where $e_i$ is the unit vector in the $i$th direction. The gradient is coded using all formulae (1) to (5).

The Jacobian of a vector function $F^T(x) = (f_1(x), f_2(x), \ldots, f_M(x))$ is given by:

$$J_{ti} = \frac{\partial f_t(x)}{\partial x_i} = \left.\frac{df_t(x+\alpha e_i)}{d\alpha}\right|_{\alpha=0}$$

with $t = 1, 2, \ldots, M$ and $i = 1, 2, \ldots, N$. Formulae up to $O(h^2)$ are implemented (only formulae (1), (2) and (3)).

### 2.2. Second order derivatives

The library offers two options for the estimation of the Hessian elements. If the gradient $g_i(x) = \frac{\partial f(x)}{\partial x_i}$ is available, the Hessian may be estimated by differencing the gradient, otherwise it is estimated using function values.

#### 2.2.1. Using the gradient

(1) Accurate to order $O(h_i) + O(h_j)$
$h_k = \pm\sqrt{\eta}\max\{1, |x_k|\}$ for $k = i, j$.
FD/BD formula:

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} \approx \frac{1}{2}\left(\frac{g_i(x+h_j e_j) - g_i(x)}{h_j} + \frac{g_j(x+h_i e_i) - g_j(x)}{h_i}\right). \tag{6}$$

(2) Accurate to order $O(h_i^2) + O(h_j^2)$
$h_k = \pm\eta^{1/3}\max\{1, |x_k|\}$ for $k = i, j$.
(a) FD/BD formula:

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} \approx \frac{1}{2}\left(\frac{4g_i(x+h_j e_j) - g_i(x+2h_j e_j) - 3g_i(x)}{2h_j} + \frac{4g_j(x+h_i e_i) - g_j(x+2h_i e_i) - 3g_j(x)}{2h_i}\right). \tag{7}$$

(b) CD formula:

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} \approx \frac{1}{2}\left(\frac{g_i(x+h_j e_j) - g_i(x-h_j e_j)}{2h_j} + \frac{g_j(x+h_i e_i) - g_j(x-h_i e_i)}{2h_i}\right). \tag{8}$$

#### 2.2.2. Using function values

(1) Accurate to order $O(h_i) + O(h_j)$
$h_k = \pm\eta^{1/3}\max\{1, |x_k|\}$ for $k = i, j$.
FD/BD formula:

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} \approx \frac{1}{h_i h_j}\big(f(x+h_i e_i + h_j e_j) - f(x+h_i e_i) - f(x+h_j e_j) + f(x)\big). \tag{9}$$

(2) Accurate to order $O(h_i h_j)$
$h_k = \pm\eta^{1/4}\max\{1, |x_k|\}$ for $k = i, j$.

Please cite this article in press as: C. Voglis et al., A numerical differentiation library exploiting parallel architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.02.004

(a) FD/BD formula:
Off-diagonal elements:

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} \approx \frac{1}{4h_i h_j}\big(9f(x) + 16f(x+h_i e_i + h_j e_j) + f(x+2h_i e_i + 2h_j e_j) - 4f(x+h_i e_i + 2h_j e_j) - 4f(x+2h_i e_i + h_j e_j) - 12f(x+h_i e_i) - 12f(x+h_j e_j) + 3f(x+2h_i e_i) + 3f(x+2h_j e_j)\big). \tag{10}$$

Diagonal elements:

$$\frac{\partial^2 f(x)}{\partial x_i^2} \approx \frac{1}{h_i^2}\big(2f(x) - f(x+3h_i e_i) + 4f(x+2h_i e_i) - 5f(x+h_i e_i)\big). \tag{11}$$

(b) CD formula:
Off-diagonal elements:

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} \approx \frac{1}{4h_i h_j}\big(f(x+h_i e_i + h_j e_j) + f(x-h_i e_i - h_j e_j) - f(x+h_i e_i - h_j e_j) - f(x-h_i e_i + h_j e_j)\big). \tag{12}$$

Diagonal elements:

$$\frac{\partial^2 f(x)}{\partial x_i^2} \approx \frac{1}{h_i^2}\big(f(x+h_i e_i) + f(x-h_i e_i) - 2f(x)\big). \tag{13}$$

(c) Mixed CD and FD(BD) formula:

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} \approx \frac{1}{4h_i h_j}\big(4\big(f(x+h_i e_i + h_j e_j) - f(x+h_i e_i - h_j e_j)\big) - 3\big(f(x+h_j e_j) - f(x-h_j e_j)\big) - \big(f(x+2h_i e_i + h_j e_j) - f(x+2h_i e_i - h_j e_j)\big)\big). \tag{14}$$

## 3. Parallelization strategy

For the shared-memory parallelization of the Numerical Differentiation Library (NDL) we have used OpenMP, the standard programming model for a wide range of parallel platforms including small-scale SMPs and emerging multi-core processors. OpenMP defines a portable programming interface based on directives, i.e. annotations that enclose loops and sections of code. In addition, it provides a means of seamless parallelization of NDL, as it allows the construction of a parallel program as a natural extension of its sequential counterpart. To achieve optimal load balance and speedup in NDL, we exploit its inherent nested parallelism [34], that results from the multiple function evaluations performed at each coordinate direction. Nested parallelism is a major feature of OpenMP that allows multiple levels of parallelism to be active simultaneously. Nowadays, several research and commercial OpenMP compilers support more than one level of parallelism.

The parallelization of the software library on multiprocessor clusters has been based on the master–worker programming paradigm, a fundamental approach for parallel and distributed computing. In NDL, task parallelism is possible due to the independent function evaluations assigned by the master to the workers. For each function evaluation, the master provides the input vector $x$ to the worker and receives the computed function value $f(x)$. The parallel version of NDL has been coded using the LWRPC library [35], a runtime environment which is part of the software and provides a lightweight framework for executing task parallelism on top of MPI. LWRPC is a flexible and portable runtime library that implements a two-level thread model (where user-level threads run on top of kernel-level threads [36]) on clusters of multiprocessors, trying to exploit the shared-memory hardware whenever this is available.

It provides transparent data movement and load balancing and allows for static and dynamic scheduling of tasks. Furthermore, it supports multiple levels of parallelism and enables the same code to run efficiently on both distributed and shared memory multiprocessors.

In LWRPC, a task is represented with a data structure, called *work descriptor*. Tasks are distributed to the available nodes and eventually executed on top of user-level threads. The same approach has also been followed for the master which is the primary task. An MPI process runs on each cluster node and utilizes one or more kernel threads that execute these tasks. Moreover, task submission and management is performed completely asynchronously by means of a special per-node server thread. There are ready queues where tasks are submitted for execution. The submission of a work descriptor to a local queue is always performed through hardware shared memory, otherwise appropriate messages are sent to the server thread of the remote node. Each work descriptor (i.e. task) is associated with an owner node. If a task finishes on its owner node, its parent is notified directly through shared memory. Otherwise, a message is sent to the owner node and this notification is performed by the server thread of that node.

When the application is executed on shared memory machines, the runtime library, and accordingly the application, operates exclusively through the available hardware. However, whenever a task is inserted on a remote node, its data has to be sent explicitly. In this case, we associate each work descriptor with the arguments (data) of its corresponding function, similarly to the Remote Procedure Call protocol [37]. For each argument, the user specifies its MPI data type and number of elements, an intent attribute, and optionally a reduction operator. These MPI-specific details are the only references made to message passing programming at the user level. Note that, the explicit data movement is performed transparently to the user.

The parallel routines that NDL exports to MPI programs are designed to be called by all MPI processes that participate in the program execution, similarly to the MPI collective communication routines. This design adheres to the SPMD (Single Program Multiple Data) execution model that MPI supports by default. When an NDL parallel routine is invoked, the execution model switches to master–worker and the thread of the process with rank 0 becomes responsible for distributing the tasks to the workers and gathering the results. In NDL, each task corresponds to a function evaluation at a given point. Before returning from the library call to the user program, the execution model

Please cite this article in press as: C. Voglis et al., A numerical differentiation library exploiting parallel architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.02.004

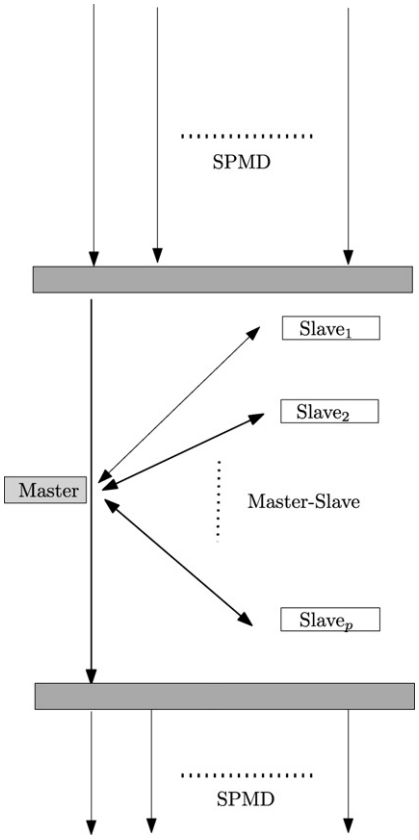


**Fig. 1.** Library's Programming Model.

switches back to SPMD and the results are broadcast from the master to the rest of the MPI processes. This is schematically illustrated in Fig. 1.

In the case of second order derivatives we use a nested parallelization model that is provided by the LWRPC library. By nested we mean that each element of the Hessian is calculated, as a first level job, by a single worker. That leads to $(N^2 + N)/2$ first level jobs. Every first level job is responsible to perform function/gradient evaluations, second level jobs, according to the desired accuracy and the bounds. By inspecting the formulae of the previous section, it is readily deduced that the required number of second level jobs ranges between two and nine.

## 4. User interface

We have implemented subroutines for calculating gradients, Hessians and Jacobians. In all cases we provide serial, OpenMP-parallel and MPI-parallel subroutines. Every subroutine has a standard and an advanced interface. The advanced interface allows the user to specify bounds on the variables and an estimate for the relative accuracy of the function evaluation. It also returns the number of function calls, error codes and optionally issues verbose output.

### *4.1. Naming conventions*

We use the following naming convention for the subroutines:

```
xNDLyz ( arguments )
```

where `x`, denotes the type of parallelization and can be:

*empty*: for the serial version
O: for the OpenMP-parallel version
P: for the MPI-parallel version

`y` denotes the order of the derivative and can be:

G: for the gradient
HF: for the Hessian using function values
HG: for the Hessian using analytic gradients
J: for the Jacobian

Please cite this article in press as: C. Voglis et al., A numerical differentiation library exploiting parallel architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.02.004

z denotes the interface type correspondingly and can be:

*empty*: for the standard interface
A: for the advanced interface

where by *empty* is meant that the symbol is missing.

For example, subroutine PNDLGA stands for the MPI-parallel code (P) for the gradient calculation (G) using the advanced interface (A). Also, NDLJ is the serial subroutine to calculate the Jacobian matrix using the standard interface.

### 4.2. Common arguments

The provided subroutines share a number of common arguments which are described bellow.

| | | |
|---|---|---|
| X | (input) | Array containing the point at which the calculation is desired. |
| N | (input) | The dimensionality of the function. |
| XL | (input) | Array containing the lower bounds on the variables. |
| XU | (input) | Array containing the upper bounds on the variables. |
| FEPS | (input) | The user's estimate for the relative precision of the function evaluation. If FEPS=0 it is reset to the machine's precision. |
| IORD | (input) | Requested order of accuracy. Possible values for gradients are 1, 2, 4 and for Hessians and Jacobians 1, 2. |
| IPRINT | (input) | Controls the amount of printout from the routine. Note that all output appears on the standard output device. Possible values are:<br>0 No printout at all.<br>1 Fatal error messages are printed.<br>2 Warning messages are printed.<br>3 Detailed information is printed (the formula that was used, differentiation steps and the resulting derivatives vector). |
| NOC | (output) | Number of calls to the function being differentiated. |
| IERR | (output) | Error indicator. Possible values are:<br>0 No errors at all.<br>1 Improper IORD value.<br>2 The supplied N is less than 1.<br>3 Some of the supplied upper bounds (XU) are less than the corresponding lower bounds (XL).<br>4 Some of the supplied variables are out of bounds.<br>5 The value of FEPS is incorrect (less than 0 or greater than 1).<br>6 The supplied IPRINT is incorrect.<br>7 N exceeds the maximum allowed value (MAXN). MAXN must be increased appropriately and the library must be reconfigured.<br>8 There is not enough internal storage. MAXN must be increased to match the number of squared terms (M) and the library must be reconfigured.<br>9 The number of squared terms (M) is less than 1. |

### 4.3. Gradient calculation

Given a multidimensional function (F), these routines return the gradient vector (G) by applying a numerical differentiation formula according to the desired order of accuracy (IORD). The user provided function F must be declared as:

```
FUNCTION F ( X, N )
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
DIMENSION X(N)
```

*Standard interface.* `SUBROUTINE NDLG (F,X,N,IORD,G)`

*Advanced interface.* `SUBROUTINE NDLGA (F,X,N,XL,XU,FEPS,IORD,IPRINT,G,NOC,IERR)`

| | | |
|---|---|---|
| F | (input) | The function to be differentiated. |
| G | (output) | Array containing the resulting gradient vector |

### 4.4. Jacobian calculation

Given a multidimensional function that is written as a sum of squared terms (residuals):

$$F(x) = \sum_{i=1}^{M} f_i(x)^2$$

Please cite this article in press as: C. Voglis et al., A numerical differentiation library exploiting parallel architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.02.004

these routines return the Jacobian matrix (FJ) by applying a numerical differentiation formula according to the desired order of accuracy (IORD). The user provided subroutine must declared as:

```
SUBROUTINE RSD ( X, N, M, F )
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
DIMENSION X(N), F(M)
```

*Standard interface.* `SUBROUTINE NDLJ (RSD,X,N,M,IORD,FJ,LD)`

*Advanced interface.* `SUBROUTINE NDLJA (RSD,X,N,M,XL,XU,FEPS,IORD,IPRINT,FJ,LD, NOC,IERR)`

`RSD` (input) A subroutine that returns the residuals.
`M` (input) The number of squared terms.
`LD` (input) Leading dimension of matrix FJ.
`FJ` (output) The Jacobian matrix. $J_{ij}(x) = \partial f_i(x)/\partial x_j$.

### 4.5. Hessian calculation

#### 4.5.1. Using gradients

Given a routine (GRD) that evaluates analytically the first partial derivatives of a function, these routines return the Hessian matrix (HES) by applying a numerical differentiation formula according to the desired order of accuracy (IORD). The user provided subroutine GRD must be declared as:

```
SUBROUTINE GRD ( X, N, G )
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
DIMENSION X(N), G(N)
```

*Standard interface.* `SUBROUTINE NDLHG (GRD,X,N,IORD,HES,LD)`

*Advanced interface.* `SUBROUTINE NDLHGA (GRD,X,N,XL,XU,FEPS,IORD,IPRINT,HES,LD,NOC,IERR)`

`GRD` (input) A subroutine that returns the gradient vector (G), given the values of the variables (X).
`LD` (input) Leading dimension of matrix HES.
`HES` (output) Array containing the resulting Hessian matrix. Note that only the lower triangular part (plus the diagonal elements) is returned.

#### 4.5.2. Using function values

Given a multidimensional function (F), these routines return the Hessian matrix (HES) by applying a numerical differentiation formula according to the desired order of accuracy (IORD). The user provided function F must be declared as:

```
FUNCTION F ( X, N )
IMPLICIT DOUBLE PRECISION (A-H,O-Z)
DIMENSION X(N)
```

*Standard interface.* `SUBROUTINE NDLHF (F,X,N,IORD,HES,LD)`

*Advanced interface.* `SUBROUTINE NDLHFA (F,X,N,XL,XU,FEPS,IORD,IPRINT,HES,LD,NOC,IERR)`

`F` (input) The function to be differentiated.
`LD` (input) Leading dimension of matrix HES.
`HES` (output) Array containing the resulting Hessian matrix. Note that only the lower triangular part (plus the diagonal elements) is returned.

## 5. Installation instructions and sample program

In this section we describe in brief how to configure and install the software and we provide a basic sample program for the MPI-parallel case.

### 5.1. Installation instructions

The software is distributed as a `tar.gz` file and can be uncompressed and extracted by issuing

```
gunzip ndl-1.0.tar.gz
tar -xvf ndl-1.0.tar.gz
```

A directory called `ndl-1.0` will be created with three subdirectories `serial`, `openmp` and `mpi` containing the serial, OpenMP–parallel and MPI-parallel distributions respectively. Any of the three distributions can be installed by entering the corresponding directory and executing the following steps:

Please cite this article in press as: C. Voglis et al., A numerical differentiation library exploiting parallel architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.02.004

(1) Configure the package:

```
./configure
```

In the case of OpenMP one must specify the Fortran compiler with the appropriate options. For example:

```
./configure F77=gfortran FFLAGS=-fopenmp
./configure F77=ifort FFLAGS=-openmp
```

Other configuration choices include the specification of the installation directory:

```
./configure --prefix=<install-dir>
```

the definition of the desired C and Fortran compilers:

```
./configure CC=<path-to-mpicc> F77=<path-to-mpif77>
```

and the definition of the maximum problem dimension:

```
./configure --with-maxn=<num>
```

Further help for the configuration parameters can be obtained by entering:

```
./configure --help
```

(2) Build the library:

```
make
```

For each distribution a library file will be created, `libndl.a` for the serial version, `libpndl.a` for the OpenMP-parallel version and `libondl.a` for the MPI-parallel version. This step also compiles the provided test run code.

(3) Install the library:

```
make install
```

By default the NDL library files will be placed in `/usr/local/lib`.

### 5.2. Sample program

We present a sample program (file `osample.f`) that uses the OpenMP version of our library in order to calculate the gradient of the function $f(x_1, x_2) = x_1 \cos(x_2) + x_2 \cos(x_1)$ using order $O(h^2)$.

```
      program otest
      implicit double precision (a-h, o-z)
      parameter (n=2)
      dimension x(n), g(n)
      external f
C
      x(1) = 1.0d0
      x(2) = 1.1d0
      iord = 2
      call ondlg ( f, x, n, iord, g )
      print *, 'point ', (x(i), i=1, n)
      print *, 'gradient ', (g(i), i=1, n)
      end
c-------------------------------------------------------------
      function f(x,n)
      implicit double precision (a-h, o-z)
      dimension x(n)
      f = x(1)*cos(x(2))+x(2)*cos(x(1))
      end
```

The above sample program can be compiled and executed using:

```
$ gfortran -o osample osample.f -londl
$ export OMP_NUM_THREADS=2
$./osample
```

Command `export OMP_NUM_THREADS=<num>` sets the number of worker threads in the OpenMP runtime library [33]. The MPI version of the above a sample program is presented bellow (file `psample.f`).

```
      program ptest
      implicit double precision (a-h, o-z)
      parameter (n=2)
      dimension x(n), g(n)
      external f
      include 'mpif.h'
      call mpi_init(mpierror)
c ...
      x(1) = 1.0d0
      x(2) = 1.1d0
      iord = 2
      call pndlg ( f, x, n, iord, g )
c ...
      call mpi_comm_rank (mpi_comm_world,irank,ierr)
      if (irank.Eq.0) then
         print *, 'point ', (x(i), i=1, n)
         print *, 'gradient ', (g(i), i=1, n)
      endif
c ...
      call mpi_finalize(mpierror)
      end
c-------------------------------------------------------------
      function f(x,n)
      implicit double precision (a-h, o-z)
      dimension x(n)
      f = x(1)*cos(x(2))+x(2)*cos(x(1))
      end
```

The above sample program can be compiled and executed using:

```
$ mpif77 -o psample psample.f -lpndl
$ mpirun -n 2./psample
```

## 6. Performance results

In Table 1 we report the relative error defined as:

$$err^{(k)} = \frac{|f^{(k)}(x) - f_{fd}^{(k)}|}{\max(1, |f^{(k)}(x)|)}$$

where $f^{(k)}(x)$ is the exact $k$th order derivative and $f_{fd}^{(k)}$ is a finite difference approximation to it. We used the five test functions listed in the first column. The relative error has been calculated at eleven equidistant points in $[-1, 1]$. In columns 2–4 we report the maximum (among these points) $err^{(1)}$, for the $O(h)$, $O(h^2)$ and $O(h^4)$ formulae. Correspondingly in columns 5–6 we report the maximum $err^{(2)}$ for the $O(h)$ and $O(h^2)$ formulae.

In order to measure the performance of the parallel-NDL implementation we have conducted extensive tests for both OpenMP and MPI-parallel versions. We have performed two sets of experiments:

E1: We used a 500-dimensional test function (without imposing bounds on the variables) and we calculated the gradient with $O(h^4)$ precision. That leads to a total of 2000 function evaluations. We have arranged for function evaluation time to be 1, 10, and 100 ms, respectively, via appropriate artificial delays.

E2: We used a 20-dimensional test function (without imposing bounds on the variables) and we calculated the gradient with $O(h^4)$ precision. In this setting the total number of function evaluations is 80. The computational cost for each function call was again set to be 1, 10, and 100 ms, respectively.

We measure the speedup $s$ defined as $s = T_1/T_p$, where $T_1$ is the time required for execution on one processor and $T_p$ is the real time required when running on $p$ processors.

**Table 1**
Relative errors in several example functions.

| Function | First order | | | Second order | |
|---|---|---|---|---|---|
| | $O(h)$ | $O(h^2)$ | $O(h^4)$ | $O(h)$ | $O(h^2)$ |
| $\sin x$ | 1.5E–08 | 2.0E–11 | 1.1E–13 | 1.0E–05 | 5.3E–09 |
| $e^x$ | 2.0E–08 | 2.4E–11 | 1.9E–13 | 1.4E–05 | 5.4E–09 |
| $x^2 \sin x$ | 4.7E–08 | 1.0E–10 | 6.7E–13 | 6.0E–05 | 2.5E–08 |
| $xe^{-2x} + \sin 3x$ | 1.5E–07 | 2.2E–10 | 5.1E–12 | 2.2E–04 | 1.2E–07 |
| $x^7 + 2x^5 - 5x$ | 5.3E–07 | 2.7E–09 | 6.2E–11 | 8.2E–05 | 1.4E–07 |

Please cite this article in press as: C. Voglis et al., A numerical differentiation library exploiting parallel architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.02.004

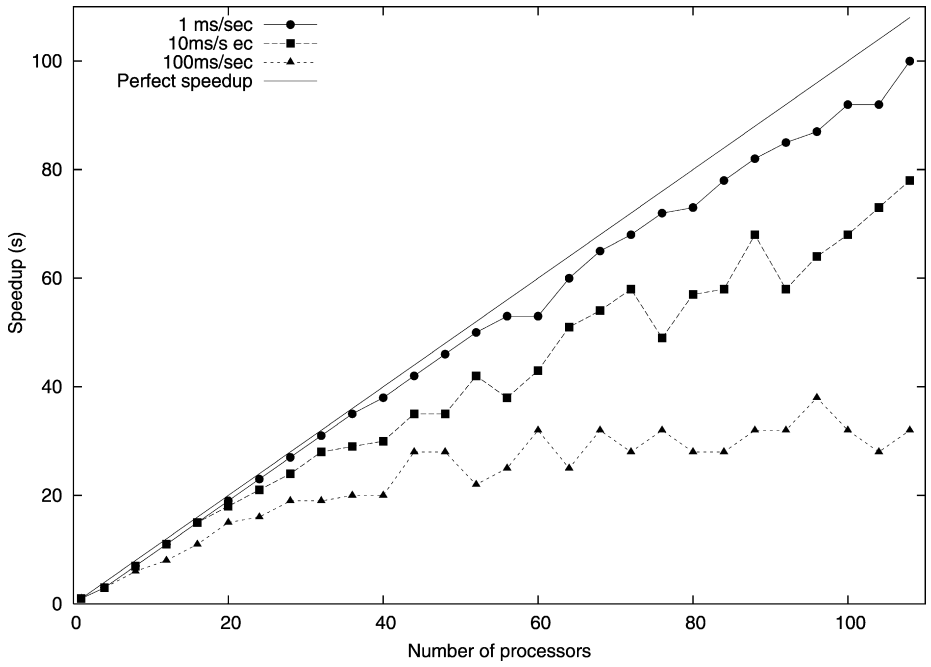


**Fig. 2.** Speedup for experiment E1 ($N = 500$).

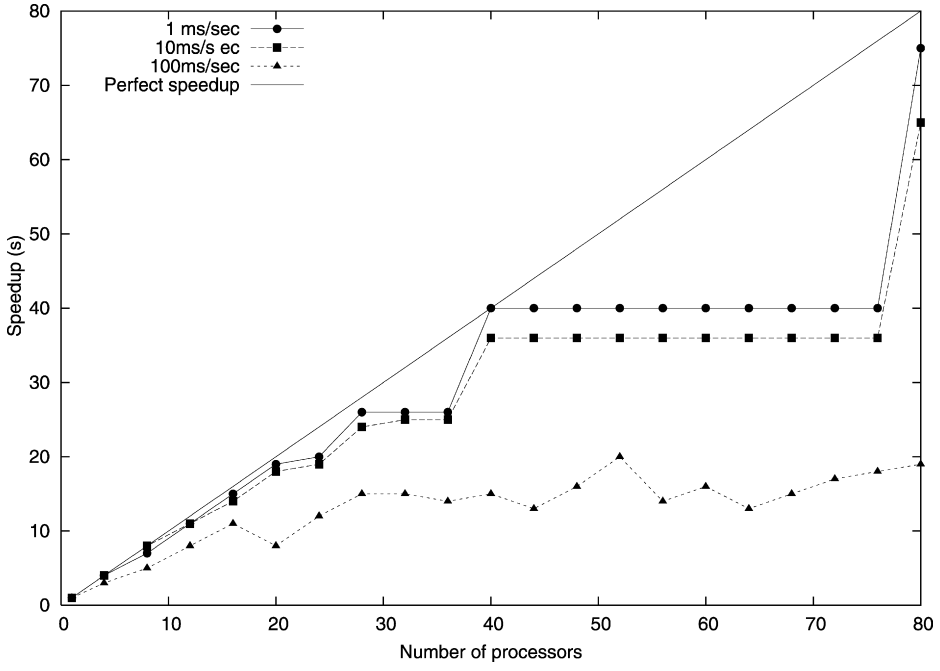


**Fig. 3.** Speedup for experiment E2 ($N = 20$).

### 6.1. MPI-parallel

Our experiments are performed on a 200 node Hewlett-Packard XC cluster system. Each node has 2 AMD Opteron-248 processors and 4 GB main memory, while the nodes are interconnected with Gigabit Ethernet.

In Fig. 2 we present the results from experiment E1. The solid line represents the ideal speedup. For the 1 and 10 ms test functions where the communication times are comparable to execution times, the speedup is reduced to approximately 35 and 70% away from the ideal, while the speedup for the 100 ms function almost coincides with the ideal as it was expected. The results for experiment E2 are presented in Fig. 3, where similar behavior is observed. The steps in the curves for the 10 and 100 ms functions are due to the way the required 80 function evaluations are distributed to the available processors. If the number of processors $p$ divides exactly the number of tasks $n_t$, then all $p$ processors are employed for $n_t/p$ cycles. Otherwise $p$ processors are fully utilized for $[n_t/p]$ cycles, and one more cycle is needed employing $n_t$ mod $p$ processors.

Please cite this article in press as: C. Voglis et al., A numerical differentiation library exploiting parallel architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.02.004

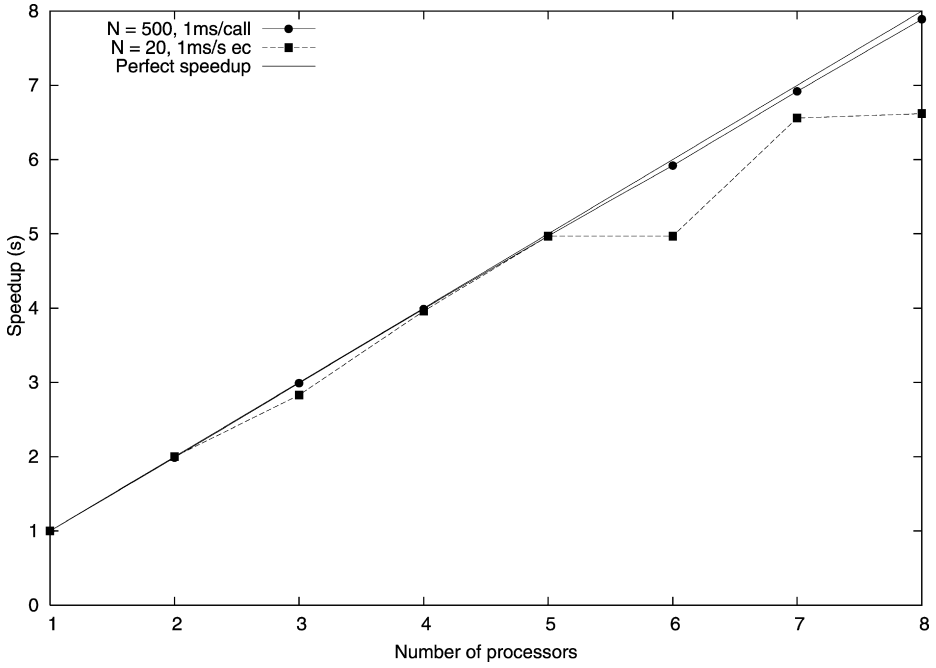


**Fig. 4.** OpenMP implementation speedup.

### 6.2. OpenMP-parallel

The same experiments were conducted on a shared-memory multiprocessor system equipped with 4 Dual-Core 3.0 GHz Intel Xeon processors and 4 GB main memory, running 64-bit Debian Linux. In Fig. 4 we present the results only for the 1 ms test function since the others (10, 100 ms test functions) almost coincide with the ideal speedup. We notice a 19% reduction from the ideal speedup when $N = 20$.

## 7. Test run description

Extensive test runs for the serial and parallel versions of the library were performed and are available with the distribution. The user is advised to repeat these runs in order to validate the installation. The relevant files are located in a subdirectory named `test`.

## Acknowledgements

The authors acknowledge the computational resources generously provided by the Center for Scientific Simulation of the University of Ioannina.

## References

[1] J. Nocedal, S.J. Wright, Numerical Optimization, Springer, New York, 1999.
[2] C.T. Kelley, Iterative Methods for Optimization, SIAM, Philadelphia, 1999.
[3] P.E. Gill, W. Murray, M.H. Wright, Practical Optimization, Academic Press, London, 1981.
[4] J.E. Dennis, R.B. Schnabel, Numerical Methods for Unconstrained Optimization and Nonlinear Equations, Englewood Cliffs, Prentice-Hall, 1983.
[5] R. Fletcher, Practical Methods of Optimization, 2nd edn., John Wiley & Sons, New York, 1987.
[6] Mathematica by Wolfram Research, http://www.wolfram.com/products/mathematica/index.html.
[7] Matlab by Mathworks, http://www.mathworks.com/products/matlab/.
[8] G. Corliss, C. Faure, A. Griewank, L. Hascoet, U. Naumann (Eds.), Automatic Differentiation of Algorithms, Springer, 2002.
[9] G. Dahlquist, A. Bjork, Numerical Methods, Prentice-Hall, New Jersey, 1974.
[10] I.R. Khan, R. Ohba, New finite difference formulas for numerical differentiation, Journal of Computational and Applied Mathematics 126 (2000) 269–276.
[11] J. Li, General explicit difference formulas for numerical differentiation, Journal of Computational and Applied Mathematics 183 (2005) 29–52.
[12] N.L. Allinger, U. Burkert, Molecular Mechanics, ACS Monograph, vol. 177, ACS Publication, 1982.
[13] K.I. Ramachadran, G. Deepa, K. Namboori, Computational Chemistry and Molecular Modeling: Principles and Applications, Springer, 2008.
[14] F. Jensen, Introduction to Computational Chemistry, Wiley, 2006.
[15] S. Yip (Ed.), Handbook of Materials Modelling, Springer, 2005.
[16] D.G. Papageorgiou, I.E. Lagaris, N.I. Papanicolaou, G. Petsos, H.M. Polatoglou, Merlin, a versatile optimization environment applied to the design of metallic alloys and intermetallic compounds, Computational Materials Science 28 (2003) 125–133.
[17] I.E. Lagaris, V.R. Pandharipande, Phenomenological two-nucleon interaction operator, Nucl. Phys. A 359 (1981) 331–348.
[18] Th.A. Rijken, et al., The Nijmegen NN phase shift analyses, Nucl. Phys. A 508 (1990) 173–183.
[19] I.E. Lagaris, A. Likas, D.I. Fotiadis, Artificial neural network methods in quantum mechanics, Comput. Phys. Comm. 104 (1997) 1–14.
[20] I.E. Lagaris, A. Likas, D.I. Fotiadis, Artificial neural networks for solving ordinary and partial differential equations, IEEE Trans. Neural Networks 9 (1998) 987–1000.
[21] I.E. Lagaris, A. Likas, D.G. Papageorgiou, Neural Network methods for boundary value problems with irregular boundaries, IEEE Trans. Neural Networks 11 (2000) 1041–1049.
[22] GSL, GNU Scientific Library, http://www.gnu.org/software/gsl/, 2008.
[23] NAG Fortran Library, D04 Numerical Differentiation, `subroutine D04AAF`.

[24] J.N. Lyness, G. Ande, Algorithm 413, ENTCAF and ENTCRE: Evaluation of normalised Taylor coefficients of an analytic function, Comm. ACM 14 (10) (1971) 669675.
[25] J.N. Lyness, C.B. Moler, Numerical differentiation of analytic functions, SIAM J. Numer. Anal. 4 (2) (1967) 202210.
[26] J.H. Mathews, Numerical Methods for Mathematics, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.
[27] P. Gilbert, R. Varadhan, The numDeriv Package, http://www.bank-banque-canada.ca/pgilbert, 2006.
[28] J. Oliver, An algorithm for numerical differentiation of a function of one real variable, Journal of Computational and Applied Mathematics 6 (2) (1980) 145–160.
[29] J. Waite, Routines for numerical interpolation, with first and second order differentiation, having non-uniformly spaced points, out to three dimensions, Comput. Phys. Comm. 46 (1987) 323.
[30] IMSLM Fortran Library, Module DERIV, http://gams.nist.gov/serve.cgi/Module/IMSLM/DERIV/3880/.
[31] H.M. Bucker, A. Rasch, A. Vehreschild, Automatic generation of parallel code for Hessian computations, in: IWOMP, in: LNCS, vol. 4315, 2008, pp. 372–381.
[32] Message Passing Interface Forum MPI: A message-passing interface standard, International Journal of Supercomputer Applications and High Performance Computing 8 (3–4) (1994) 159–416, http://www.mpi-forum.org/.
[33] OpenMP Architecture Review Board, OpenMP specifications, available at: http://www.openmp.org.
[34] X. Tian, J.P. Hoeflinger, G. Haab, Y.-K. Chen, M. Girkar, S. Shah, A compiler for exploiting nested parallelism in OpenMP programs, Parallel Computing 31 (2005) 963–980.
[35] P.E. Hadjidoukas, A lightweight framework for executing task parallelism on top of MPI, in: Proc. of the 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, 2004.
[36] D.R. Butenhof, Programming with POSIX Threads, Addison-Wesley, 1997.
[37] Sun Microsystems, Inc., RPC: Remote Procedure Call Protocol Specification Version 2, Request For Comments: 1057, 1988.

Please cite this article in press as: C. Voglis et al., A numerical differentiation library exploiting parallel architectures, Computer Physics Communications (2009), doi:10.1016/j.cpc.2009.02.004