DPL: A Decent Programming Language*

A. L. Davis       W. W. Wadge

1 Introduction

The effect of language upon thought processes is important and underestimated. The extreme example would be the “Newspeak” of Orwell’s 1984 [8]. Treasonous attitudes did not exist because “Newspeak” did not contain the words with which one could think of treason. It is true that this is a fictitious situation, but it is not far from being true in modern Computerdom. The majority of today’s users program and think of programs in a single language. In Canada, this language is probably FORTRAN, COBOL or PL/I. It is here that Orwell’s warning rings true. In any given language, there are generally an entire family of problems which are difficult to program in that language. When faced with this situation, many programmers feel that the problem is unsolvable. This type of restricted thinking greatly limits the ability and usefulness of the programmer. A small subset of programmers can think of the problem in less restricted ways and find a solution. Our goal is to educate a majority of programmers to think in less restricted ways and therefore avoid the “Newspeak” phenomenon.

Since initial ideas about anything seem to have a major effect on future attitudes, we feel that it is extremely important for the initial programming language to be a good one. We realize that the introductory language is but a small part of the programmer’s education, but since the initial language is the vehicle for learning initial programming skills, early algorithm design, and style development, it is an important part.

The remainder of this paper is devoted to the discussion of what this initial language should be, and the specification of DPL (Decent Programming Language), our effort at such a language.

Introductory programming languages are usually picked either for their historical importance or for their commercial popularity. This has resulted in 1) the acquisition of limited skills for academic puzzles, 2) confusion over fundamental concepts, and 3) restricted ability to use other languages in the future. The initial language should be viewed as a tool to provide a sound foundation of programming skills, rather than as the language. We take it as axiomatic that one of the most important of these skills is the ability to create a well-structured program (Dijkstra [5]).

A good initial language should have the following properties:

1. It should sufficiently aid the programmers’ power of abstraction (Dijkstra [5]).
2. It should tend to constrain the programmer to structured programming techniques.
3. It should be simple, natural to use, free from unnecessary specification, and allow programs to be composed in a straightforward and concise manner.
4. It should allow important programming techniques and concepts to be applied directly.

The implications of these properties are many, the spirit of which is well put by Dijkstra [5] in the following quotations. We assume familiarity with Dijkstra’s works.

... the art of programming is the art of organizing complexity, of mastering multitudes, and avoiding its bastard chaos as effectively as possible.

once a person has understood the way in which variables are used in programming, he has understood the quintessence of programming.

It is my firm belief that in each process of some complexity, the variables occurring in it admit analogous hierarchical orderings and that, when these hierarchies are clearly recognizable in the program text, the gain in clarity of the program and in efficiency of the implementation will be considerable.

let us be honestly humble and interpret the length of the proof as an urgent advice to restrict ourselves to simple structures whenever possible and to avoid in all intellectual modesty ‘clever constructions’ like the plague.

what bothers me is the length of the texts at the various levels. Therefore we may expect that notational technique will be one of our main concerns.

Our hope is that eventually we shall arrive at a program structure that is both nice to compose and nice to execute.

A structured programming language must allow the programmer to create program modules and to combine modules to form other modules. The effect of executing the module should remain local to that module. This restriction rules out global variables, call by name and call by reference, leaving local variables and call by value. There are some implementation efficiency problems surrounding call by value, but it is possible to reduce the effect of these problems. Implementation problems ideally should not affect the user environment. Moreover, one should be able to replace any module with a new module in a way that does not affect any of the other modules in the hierarchy. In order to guarantee non-interfering replacement, it is necessary that the modules have two important features: 1) two-terminal control, and 2) write locality. Write locality means that the variables for which the module is allowed write access must be local to that module.

Allowing only local variables and call by value guarantees write locality. These properties have strong implications on the nature of the DPL subroutine, i.e., it takes the appearance of a mathematical function. Two-terminal control means that the block can only be entered in one place and only be exited in one other place. If blocks did not have the two-terminal control property, and block A were to replace block B, then block A’s entries and exits would have to match those of block B; this would ruin the chance for A’s structure to be independent from the structure of B. Maintaining the independence between module structures is an important aspect of structured programming. DPL allows transfers to the head of a block via the LOOP command, and transfers to the end of a block via the EXIT command. Any less restrictive transfer, such as a general GOTO statement, would destroy this necessary two-terminal control property. It should be clear that the LOOP and EXIT commands do not destroy the two-terminal control structure of the block. The advantages of EXIT and LOOP transfers are less complex conditions for iterations, and a more natural and efficient control flow within the block.

Structured operators and operands were introduced by Iverson [6] and find their best example in the APL programming language. Using APL easily demonstrates how effective structured operators are as an aid to abstraction. For this reason, structured operators exist in DPL much as they do in APL, the only difference being a slight change in the conformability rules. In DPL there is no difference between a one-cell array and a one-cell vector. Namely, 1’s in the rank vector of arrays are ignored for the purposes of conformability checking. A detailed description of the DPL conformability rules can be found in Davis [4]. A further aid to abstraction would be simple flexible data structures that admit to hierarchical organization. The records of Hoare [3, 1] using array primitives seem to be sufficient.

The above properties imply the following language features:

1. Simple syntax free from unnecessary rules and side effects.
2. English-like notation to aid readability and facilitate learning.
3. Default and simple format I/O.
4. Simple yet flexible data structures (records and arrays).
5. Structured operators and operands.
6. Recursion.
7. A concise control structure that encourages modularity.
8. Modules exhibiting two-terminal control and write locality (i.e., function-like subroutines, only local variables and only call by value).
9. Notational advances where possible.

DPL was designed to incorporate these features into a single language in a sensible way. Many of the features of DPL exist separately in existing languages and we therefore make no claims of origination.

2 DPL Language Specifications

2.1 Blocks

The basic unit of DPL code is the block. A block consists of a BEGIN statement, followed by declarations of local variables, followed by any number of executable statements and/or nested blocks, followed by an END statement. The BEGIN and END are treated as separate statements, not statement brackets as in ALGOL [1].

A block may be entered sequentially at the BEGIN in the usual manner. Once a block is entered, the executable statements are executed in the order they appear, unless a LOOP or EXIT statement is encountered. LOOP causes control to be transferred to the first executable statement of the block. EXIT causes control to leave the block via the END statement. Note that LOOP and EXIT may be qualified (see below). All storage allocated local to a block is lost upon leaving the block (as in ALGOL [1]).

2.2 DPL Data Structure

A DPL Data Structure is a collection of simple data items organized in a hierarchy. Since words such as “structure” and “data” can commonly mean different things, we will give definitions for our terms.

Definition 1. A DPL atom is one of
1. an integer;
2. a real number;
3. a character;
4. a logical value (true or false);
5. a function; or
6. null (undefined atom).

DPL also has two non-atomic structures. They are the array and the record [3, 1]. Both are structures of homogeneous cells (the structure of all the cells is the same), while the record may contain non-homogeneous cells.

Definition 2. A cell is any DPL atom or any DPL structure.

Definition 3. Any set of cells is homogeneous iff
1. they all contain the same type of atom;
2. they all contain the same size and type of array; or
3. they all contain records of the same structure.

Definition 4. A DPL array is any n-dimensional array of one or more cells.

Definition 5. A DPL data structure is a DPL atom, a DPL record or a DPL structure.

2.3 Identifiers, Declarations, and Names

A DPL name is any string of digits and alphabetic characters beginning with a character. DPL has two kinds of names: 1) function names, and 2) identifier names. Identifiers denote storage areas for DPL data structures or substructures. An identifier remains undefined until:

1. it is assigned a value in an assignment statement;
2. a structure which has it as a substructure is assigned a value; or
3. it is a record and all its components are assigned values.

Any attempt to assign to a defined structure a structure of a different type results in an error. The same is true of assignments to substructures. All identifiers that are declared local to a block become undefined upon leaving that block.

Identifiers may be declared. The declarations give partial or complete information about the type of the structures an identifier will represent. Any attempt to assign to an identifier a structure whose type conflicts with this information results in an error. All declarations must appear at the head of some block and apply only to occurrences of the identifier within that block. All identifiers denote structures local to the block in which they first occur.

Identifiers are declared in a DECLARE statement similar to PL/I's [2]. The format is DECLARE, or DCL, followed by a list of declaration items separated by commas. A declaration item is an identifier, possibly subscripted or followed by more information. This type information may be one of the words: INTEGER, REAL, CHARACTER, LOGICAL. For example,

\[
\text{DECLARE X(10) REAL, I INTEGER}
\]

Type information may also be the word LIKE followed by another identifier. This attributes to the first all information so far declared to the second. For example,

\[
\text{DECLARE X REAL, Y(10) LIKE X}
\]

is the same as

\[
\text{DECLARE X REAL, Y(10) REAL}
\]

Finally, the identifier may be followed by a colon, then a list of declaration items all enclosed in parentheses. For example,

\[
\text{DECLARE M INTEGER, PERSON: (NAME, AGE INTEGER, ADDRESS)}
\]

Each item in the sublist represents a substructure. The format for items in a sublist is as above, except the identifier may be omitted and any subscript follows the word denoting the type. An item may also be omitted completely. For example,

\[
\text{DECLARE STUDENT1: (NAME(10) CHAR, ID INTEGER, INTEGER)}
\]

\[
\text{DCL COURSEL : (CODE, DESC(250) CHAR, ENROLLMENT: (REG, HONORS), OTHER DATA)}
\]

\[
\text{DCL CODE: (LETTER CHAR, NUMBER), HONORS: (NUMBER, TYPE)}
\]
Structures are referenced as in PL/I, e.g., ID and STUDENT1.ID are equivalent.

Structures may not overlap, i.e., either they have no substructures in common or one is a substructure of the other. Thus in the above example, ‘NUMBER’ is ambiguous; it is necessary to distinguish between ‘code number’ and ‘honors number’. In the same way, if we added

DCL STUDENT2 LIKE STUDENT1

we would have to distinguish between STUDENT1.ID and STUDENT2.ID.

Attributes may be factored, as in

DCL (C1, C2, C3) : (REAL, REAL)

2.4 Operations

The simplest operations are operations on atoms. They are:

- arithmetic: +, -, *, /, **.
  They have the usual meanings, mixed mode allowed.

- relational: <, >, =, ne, ge, le.
  They have the standard meanings when operands are numerical; when both are characters, comparison is lexicographic, e.g., ‘A’ < ‘B’ is TRUE.

- logical: AND, OR, NOT.

The above operations are defined for more complex structures as follows:

- If the operands have the same type the operation is performed on corresponding atoms, e.g.,
  
  \[(3, (4, 5)) < (2, (7, -3))\]  is  \[(FALSE, (TRUE, FALSE))\]

- If one of the operands is an atom, it is applied to every atom in the other operand, e.g.,

  \[2 * (4, (1, 3))\]  is  \[(8, (2, 6))\]

- Otherwise the result is an error.

DPL also includes these operators:

same  A same B iff A and B currently represent identical data structures.

in  A in B iff B is an array, dimension \((n_1, \ldots, n_k)\), and A same B \((i_1, \ldots, i_k)\) for some sequence \((i_1, \ldots, i_k)\).

index  A index B is the first sequence \((i_1, \ldots, i_k)\) such that A same B \((i_1, \ldots, i_k)\), NIL if there is no such \((i_1, \ldots, i_k)\).

...N ... M, with N, M integers, is the array of integers N, N + 1, ... , M.

|  Array concatenation, e.g., if A and B have dimension 3 and 5 respectively, A|B  has dimension 8. If C has dimension \((5, 4)\), then B|C has dimension \((5, 5)\).

cross  A cross B gives the Cartesian product of A and B.

(A1, ..., An)  is a structure which has A1, ..., An as substructures. Each Ai may also be a substructure of another structure Aj, where i ≠ j. Ai and Aj may have no substructure in common.

2.5 Expressions

The simplest expressions are constants. Numerical constants are written in the standard way, e.g., -512 or 7.832E5, etc. The logical constants are TRUE and FALSE. Character constants consist of a character enclosed in single quotes, e.g., ‘t’. More than one character enclosed in quotes is a character array, e.g., ‘word’ is of dimension 4.
DPL expressions are formed by starting with constants and identifiers and allowing meta-operations, function evaluation and array subscription. Descriptions of functions and meta-operations appear in later sections. Arrays can be subscripted as in FORTRAN [7], e.g., $A(3,4)$. The subscripts can of course be arbitrary expressions. A subscript of a $k$-dimensional array is a structure consisting of $k$ integers: this form of subscripting can also be used. For example, if $S = (3,2,5)$, then $A(S)$ means $A(3,2,5)$. Arrays are subscripted as they are in APL [6]. If the ... notation is used, one of the operands can be omitted, e.g., $A(\ldots 8)$ indicates the first 8 components of $A$, $A(5\ldots)$ all but the first 4. If a subscript is omitted the full range is assumed.

2.6 Assignment Statements

The right-hand side of an assignment is an expression; the left-hand side must be a structure indicator which is either

- an identifier: possibly subscripted; possibly with substructures indicated $(A(3\ldots5).X)$
- an expression of the form $(I_1, I_2, \ldots, I_n)$, each $I$ a structure indicator.

For example

```plaintext
DCL L (3,3), M: (R,S)
X := Y+Z;
(A,B) := (C,D);
A(1\ldots3,2\ldots4) := L;
(X,|Y,Z(3))=(L,M)
```

Note that $(A,B) := (C,D)$ means $Z:=C$ and $B:=D$. Different substructures on the left-hand side may not overlap. This avoids ambiguities like $(X,X):=(0,1)$. The type of the structure on the left-hand side must be identical to the type of the structure of the expression on the right. When one array is assigned to another they must be of identical dimensions. This restriction can be avoided using the ... notation. If one of the operands of the ... is omitted, the result is the same as if the missing operand had been specified in such a way as to make the dimensions of the arrays identical. For example, if we have

```plaintext
DCL X(10), Y(5), Z(3) then
X(3\ldots) := Y is the same as X(3\ldots7) := Y
X(3\ldots) := Z is the same as X(3\ldots5) := Z
X(\ldots9) := Y is the same as X(5\ldots9) := Y
```

The other exception to the identical dimension rule occurs when the left-hand side is an array, and the right-hand side is a single structure of the same type as each array component. In this case the right-hand side is assigned to each component of the array. For example, if we have

```plaintext
DCL X(3) then
X:=8 is the same as X:=8|8|8.
```

2.7 Meta-Operators

DPL contains several meta-operators that correspond to standard mathematical meta-operations. The syntax of any expression containing a meta-operation is:

```plaintext
(meta-op)(identifier) IN (expression) ; (logical expression)
```

where the “;” means “such that”.

The meta-operations are (the first three have their usual meanings):

- FOR ALL
- EXIST
- FIRST
- **SIGMA** (summation)
- **PI** (product operation)
- **LAMBDA** (returns the vector of values of the expression)

**EXIST** and **FOR ALL** may return **NIL** values if their conditions are not met. Examples of meta-expressions and their results are:

- **EXIST I IN 1...5 : I>3** gives **TRUE**
- **SIGMA I IN 1...4: I** gives **30**
- **FIRST T IN .8|.6|.3|1.1|.2 : T<.5** gives **.3**
- **LAMBDA J IN 0...3 : 1 - J** gives **(1,0,-1,-2)**

### 2.8 Procedures

Any DPL block may be named and can have arguments as follows:

```
BEGIN PROCEDURENAME (ARG1, ARG2, ..., ARGN)
...
END
```

Any named block may be a procedure. All arguments are call by value. Arguments can also be specified to allow for returned results as follows:

```
BEGIN P NAME (RET1, RET2 : IN1, IN2)
```

**INPUT ARGUMENTS**

**OUTPUT ARGUMENTS**

The expression `(A,B) := PNAME (C,D)` causes `PNAME` to be invoked with `C` and `D` as the input parameters. The results of `PNAME` are stored in `A` and `B` respectively. Any length mismatch in the formal parameter list causes an error. If no output parameter is needed, it may be implicitly passed in the following two ways.

- **EXIT (C,D,)**
- **END (C,D,)**

### 2.9 Loops and Conditionals

DPL has no `'FOR'` or `'DO'` statements. Any executable statement can be qualified. The qualifiers are:

- **IF**
- **UNLESS**
- **UNTIL**
- **WHILE**
- **FOR**

A statement is qualified by writing a qualifier after it, followed by the details of the qualifications on the same line.

- **IF** and **UNLESS** are followed by logical expressions and the statement is executed if or unless the expression evaluates as **TRUE**, e.g.,
  - `X = -X IF X < 0`
  - `I = I + 1 UNLESS I > 100`

- **UNLESS** and **WHILE** are also followed by logical expressions. They cause the statement to be executed repeatedly until or as long as the expression evaluates as **TRUE**, e.g.,
  - `X = ITER (X) UNTIL P(X) EQ 0`

The evaluation of the logical expression is performed before each execution; a statement qualified by **WHILE** or **UNTIL** might not be executed at all.

- **FOR** is followed by a scalar variable, then **IN**, then a vector expression of the same type as the variable. The statement is executed repeatedly while the variable takes on all values in the range of the vector, e.g.,
\( X(I) = I^2 \) for \( I \) in 1..N
\( SUM = SUM + T \) for \( T \) in \( X \)

An entire block may be qualified by qualifying the block’s BEGIN statement, e.g.,

\[
\text{BEGIN IF } X < 0 \\
\quad X = -X \\
\quad V = \text{TRUE} \\
\text{END}
\]

If a block is qualified by WHILE or UNTIL, then each execution causes reevaluation of the qualifying expressions. EXIT terminates further executions of the block. With a FOR qualifier, LOOP causes the control variable to be assigned a new value and EXIT ends the iteration, e.g.,

\[
\text{BEGIN FOR } I \text{ IN } 1..N \\
\quad \text{LOOP IF } X(I) > 0 \\
\quad \quad X(I) = -X(I) \\
\quad \quad K := K+1 \\
\text{END}
\]

If a FOR qualified block is not EXITed, the control variable will on completion of the iteration be assigned the value NIL.

3 Sample DPL Programs

In this section, we present some samples of DPL programs, comment on their function, and at times also show equivalent programs in ALGOL. All the ALGOL programs were found in published algorithms. It is not our intent to try to show conciseness (although in some cases it is evident), but to show that DPL is a more natural notation for programming than are other more commonly used languages.

3.1 Library Routines

The following DPL routines are used often enough to justify classing them as library routines for this paper.

L1. Calculate the minimum-valued element of an array

\[
\text{BEGIN MIN(X)} \\
\quad M=X(1) \\
\quad \text{BEGIN FOR } T \text{ IN } X \\
\quad \quad M:=T \text{ IF } T<M \\
\quad \text{END} \\
\quad \text{EXIT (T)} \\
\text{END}
\]

L2. Calculate the sum of the elements of a vector

\[
\text{BEGIN SUM(X)} \\
\quad \text{EXIT (SIGMA T IN X:T)} \\
\text{END}
\]

L3. Euclidean normalization

\[
\text{BEGIN LENGTH(X)} \\
\quad \text{EXIT (SQRT(SIGMA T IN X: T**2))} \\
\text{END}
\]
3.2 Example 1

The following is a program in B (an experimental programming language being developed at Bell Laboratories; algorithm courtesy B. Kernighan) to test permutations.

```plaintext
/* return false if new permutation is in ascending order, 
   true otherwise: all declarations omitted */

I:=N
WHILE (I>1 AND S(I-1) > S(I)) I:=I-1
J:=N
WHILE (I>1 AND S(I-1) > S(J)) J:=J-1
IF (TEST := (I>1)) S(I-1),S(J) := S(J),S(I-1)
   J:=N
WHILE (I>J) (S(I),S(J) := S(J),S(I); I,J := I+1,J-1;)
RETURN(TEST)
```

B was designed to help the programmer to write clear and understandable code. Note that no comments will appear in any of the programs in this section. We feel that for program comparisons the code should stand on its own merits rather than comments for comprehensibility.

The equivalent DPL program is

```plaintext
BEGIN PERLE(S); INTEGER
N := DIM(S)
I := FIRST I IN N-1..(-1)..<1 : S(I) < S(I+1)
EXIT (NIL) IF I=2
J := FIRST J IN N..(-1)..<I : S(I) < S(J)
S(I|J) := S(J|I)
S(I..<N) := S(N..<(-1)..<I)
EXIT (S)
END
```

3.3 Example 2

A simple interchange sort. The list is shortened each pass.

```plaintext
BEGIN SORTI(A)
N := DIM(A)
BEGIN
   I := FIRST I IN 1..<N-1 : A(I) > A(I+1)
   EXIT IF I=NIL
   A(I | I+1) := A(I+1 | I)
   LOOP
END
EXIT (A)
END
```
3.4 Example 3

A binary sort.

\[
\text{BEGIN BSORT}(A) \\
N := \text{DIM}(A) \\
\text{BEGIN FOR } J\text{MAX } \text{IN } (N-1)\ldots(-1)\ldots2 \\
\quad J := \text{FIRST } J \text{ IN } 1\ldots J\text{MAX } : A(J) > A(J+1) \\
\quad \text{EXIT IF } J=\text{NIL} \\
\quad \text{BEGIN FOR } K \text{ IN } J\ldots J\text{MAX} \\
\quad\quad A(K|K+1) := A(K+1|K) \text{ IF } A(K) > A(K+1) \\
\quad \quad \text{END} \\
\quad \text{END} \\
\text{EXIT } (A) \\
\text{END}
\]

3.5 Example 4

A radix sort.

\[
\text{BEGIN RSORT}(A,K) \\
\text{EXIT } (A) \text{ IF } K=1 \\
\quad I := \text{A INDEX MAX } (A(1\ldots K)) \\
\quad A(I|K) := A(K|I) \\
\quad A := \text{RSORT}(A,K-1) \\
\quad \text{EXIT } (A) \\
\text{END}
\]

3.6 Example 5

The following is an example of a program that is commonly used as an example of an algorithm that is very difficult to code efficiently without \texttt{GOTO} statements. The problem is to maintain a table of items and the count of how many times each item was entered. On each call a trial entry is passed to the routine. If the entry already exists, the program is to increment the count field and return the location of the entry. If the entry does not exist, the program is to enter the entry into the table, set its count to 1, and return its location. We present two DPL versions, 1) without \texttt{INDEX}, and 2) with \texttt{INDEX}.

Version 1

\[
\text{BEGIN LOOKUP } (\text{TABLE, TARGET}) \\
\quad \text{DCL } \text{TABLE : (TAB: (DATA,N), LENGTH)} \\
\quad I := 1 \\
\quad \text{BEGIN} \\
\quad\quad \text{EXIT IF } \text{TARGET := TAB}(I).\text{DATA} \\
\quad\quad I := I+1 \\
\quad\quad \text{LOOP UNLESS } I>\text{LENGTH} \\
\quad\quad \text{TAB}(I) := (\text{TARGET},0) \\
\quad\quad \text{LENGTH} := I \\
\quad\quad \text{END} \\
\quad \text{TAB}(I).N := \text{TAB}(I).N + 1 \\
\quad \text{EXIT } (\text{TABLE,I}) \\
\text{END}
\]
Version 2
BEGIN LOOKUP (TABLE, TARGET)
DCL TABLE : (TAB: (DATA,N), LENGTH)
I := TAB.DATA INDEX TARGET
BEGIN IF I=NIL
   I := LENGTH := LENGTH+1
   TAB(LENGTH) := (TARGET,0)
END
TAB(I).N := TAB(I).N + 1
EXIT (TABLE,I)
END

3.7 Example 6
This example and the ALGOL version is CACM Algorithm #387.

algorithm 387
function minimization and linear search [e4]
K. Fielding (Reed, 23 Sept. 1969)
Computing Centre, University of Essex, Wivenhoe Park, Colchester, Essex, England
key words and phrases: function minimization, relative minimum, quasi-Newton method
CR categories: 5.15

[Editor’s Note: According to tests made by the referee this algorithm is slower than flepomin, Algorithm 251, Comm. ACM 8 (Mar. 1965), 169–170. However, in two out of six tests flepomin failed and bromin did not fail to find a minimum.—L.D.F.]

procedure Bromin
   (n, iterations, number, maxiters, toleriter, tolerance, x, f, g, h, computef, computeq, converged);
value n, iterations, toleriter, tolerance, maxiters;
integer n, iterations, number, maxiters;
real toleriter, tolerance, f;
array x, g, h;
Boolean converged;
procedure computef, computeq;

comment This procedure minimizes a function using the method of Broyden [1]. The parameters are described as follows. n is the number of independent variables. iterations is an upper limit on the number of iterations allows. On exit number is the actual number of iterations taken. maxiters is the maximum number of function evaluations allowed on each linear search. toleriter is the convergence limit for Linmin2. tolerance is used as the convergence limit. A solution is assumed to have been reached if \( g(x)g'(x) < \) tolerance. \( x[1:n] \) is an estimate of the solution. On exit it is the best estimate of the solution found. f is the current function value \( f(x) \). \( g[1:n] \) is the current gradient vector of \( f(x) \). \( h[1:n,1:n] \) is the inverse Jacobian at the solution if number \( \geq n \) and if converged = true on exit. computef(x, f) is a procedure provided by the user to evaluate the function at any point. computeq(x, g) is a procedure provided by the user to evaluate the gradient vector at any point. converged is a Boolean variable used as follows:

   On entry converged = true implies that x, f, g, and h all have been assigned values, if converged = false however it is assumed that just x has been assigned a value and h will be set to a unit diagonal matrix.

   On exit converged = true means that a solution has been found, converged = false means that no solution has been found. However x is set to the best point found so far while the function value, gradient vector, and estimated inverse Jacobian corresponding to x are in f, g, and h.

   The procedure Linmin2 (n, maxiters, toleriter, x, f, computef, t, p) is used to find a linear minimum on each iteration.

begin
  integer i, j; real norm, t, ythy, pty, temp;
  array n, h, hy[1 : n];
  if ~converged then
    begin
      comment: Initialize g, f, h and converged;
      compute f(x, f); compute g(x, g);
      converged := true;
      for i := 1 step 1 until n do
        begin
          h[i, i] := 1.0;
          for j := i + 1 step 1 until n do
            h[i, j] := h[j, i] := 0.0
        end of loop on i to set up h
      end of initial set up
      start of main loop on number;
      for number := 1 step 1 until iterations do
        begin
          for i := 1 step 1 until n do
            begin
              comment: Evaluate the search vector p;
              p[i] := 0.0;
              for j := 1 step 1 until n do
                p[i] := p[i] - h[i, j] × g[j]
              end of loop on i to evaluate p;
              Linmin2 (n, mazters, toler, x, f, compute f, t, p);
              comment: Finds the optimum value of t and the values of x and f associated with it;
              for i := 1 step 1 until n do
                y[i] := g[i]
              comment: Use y as a temporary storage location for the old gradient before evaluating the new one as y = g_{new} - g_{old};
              compute g(x, g);
              norm := 0.0;
              for i := 1 step 1 until n do
                begin
                  norm := norm + g[i] × 2;
                  y[i] := g[i] - y[i]
                end of loop to calculate g'y'g and y;
              ythy := pty := 0;
              for i := 1 step 1 until n do
                begin
                  hy[i] := 0;
                  for j := 1 step 1 until n do
                    hy[i] := hy[i] + h[i, j] × y[j];
                  ythy := ythy + y[i] × hy[i];
                  pty := pty + p[i] × y[i]
                end of loop to evaluate hy, p'y'g and g'y hy;
              temp := ythy/pty + t;
              for i := 1 step 1 until n do
                begin
                  h[i, i] := h[i, j] + ((p[i] × temp - 2.0 × hy[i] × p[i]) / pty;
                  for j := i + 1 step 1 until n do
                  end of loop to update the matrix h;
                  if norm < tolerance then go to successful
                end of main loop on number;
              number := iterations;
              converged := false;
              successful :
            end of procedure Bromin;
          end of loop on i to evaluate p;
        end of loop on number;
      end of main loop on number;
      converged := false;
      successful :
procedure Linmin2 (n, maxiters, toliter, x, f, computef, t, p);
  value n, maxiters, toliter;
  integer n, maxiters;
  real toliter, f, t;
  array x, p;
  procedure computef;

comment This procedure carries out a linear search over \( t \). It considers \( f(x + p \times t) \) as a function of \( t \) alone. \( f \) is evaluated for three points. It is now assumed that \( f(t) \) can be approximated by a quadratic. If this quadratic has a minimum, then this is taken as a better estimate of the minimum of \( f(t) \). If, however, the quadratic is concave, a step is taken in the direction of the best point so far. If the four points obtained form an increasing or decreasing sequence with respect to \( t \) then the largest is rejected. If they do not, then they must bracket a local linear minimum and the three points retained are those that most closely enclose this minimum. This process is repeated until it is felt that a good estimate of \( t \) is available (see parameter toliter), or until some limit on the number of function evaluations is violated (see parameter maxiters). The parameters are described as follows. \( n \) is the number of variables. \( maxiters \) is the maximum number of function evaluations allowed in the linear search. \( toliter \) is the tolerance for minimization, exit if \( \text{abs}((t - \text{last} t)/t) < \text{toliter} \). \( x[i : n] \) is the array of independent variables. \( f \) contains the function value \( f(x) \). computef\((x, f)\) is the user provided routine to evaluate the function values at any point. \( t \) contains the best value of the scalar used for the step length. \( p[1 : n] \) is the vector which gives the direction of the step. If \( tf \) is the final value of \( t \) then the actual step taken is \( p \times tf \). This routine is based on the procedure quadmin by Broyden [2].


begin
  integer i, left, center, right, count;
  real alpha, beta, gamma, lastt, ptp;
  array vt, phi[1 : 3];
  procedure reject(j);
    value j;
    integer j;
    comment This procedure replaces one of the old values of \( t \) and then sorts the remaining three in ascending order of \( t \) in the array \( vt \);
    begin
      procedure interchange(i, j);
        integer i, j;
        comment if \( vt[i] > vt[j] \) interchange \( i \) and \( j \);
        begin
          integer k;
          if \( vt[i] > vt[j] \) then
            begin
              k := i; i := j; j := k
            end
        end
      end of interchange;
      start of reject;
      vt[j] := t;
      phi[j] := f;
      interchange (center, right);
      interchange (left, center);
      interchange (center, right);
    end of reject;
end
This procedure evaluates a new value for $x$ and the corresponding value of $f$:

```
procedure basic
begin
  for $i := 1$ step 1 until $n$ do
    $x[i] := x[i] + (t - lastt) \times p[i];$
    lastt := $t$; compute $f(x, f);$ 
end of basic;
```

Start of Linmin2 itself:

```
comment Initialize $phi$, $vt$, $left$, $center$ and $right$;
$phi[1] := f;$ 
$left := 1$; $center := 2$; $right := 3$; 
$lastt := vt[1] := ptp := 0.0;$ 
for $i := 1$ step 1 until $n$ do 
  $ptp := ptp + p[i] ↑ 2;$ 
  $ptp := 1.0/sqrt(ptp);$ 
comment $ptp$ is now used to limit the initial step; 
$vt[2] := t := if ptp < 1.0$ then $ptp$ else $1.0;$ 
basic; 
$phi[3] := f;$ 
comment Sets up first three values before entering main loop; 
for $count := 3$ step 1 until $maxiters$ do 
begin 
  $alpha := -(phi[1] \times alpha + phi[2] \times beta + phi[3] \times gamma)/(alpha \times beta \times gamma);$ 
comment If the quadratic through the three points is convex, 
  $t$ is chosen as the minimum of it. If it is concave, however, 
  $t$ is chosen as a step in the direction of steepest descent; 
  $t := if alpha > 0.0$ then $- beta/(2.0 \times alpha)$ 
    else if $phi[right] > phi[left]$ 
      then $2.0 \times vt[left] - 2.0 \times vt[center]$ 
      else $3.0 \times vt[right] - 2.0 \times vt[center];$
  if $abs((t - lastt)/t) < toliter$ then 
begin 
  $t := lastt; go to exit;$ 
end of exit where minimum has been found; 
basic; 
if $t > vt[right]$ 
  $\lor (t > vt[center] \land f < phi[center])$ 
  $\lor (t > vt[left] \land f < vt[center] \land f > phi[center])$ 
then reject(left) 
else reject(right); 
comment Choose which point to reject; 
end of main loop which used count as an index; 
exit ;
end of Linmin2
```
The DPL version of BROMIN is:

BEGIN BROMIN (X, FUN, GRAD, INITIAL.DATA, CONVERGENCE.DATA, LIMINDATA)
  DCL X REAL, (FUN, GRAD) FUNCTION
  DCL INITIAL.DATA : (F,G,H), LIMINDATA : (MAXITER, TOL)
  DCL CONVERGENCE.DATA : (ITERATIONS, TOLERANCE)
  (F,G,H) := (FUN(X), GRAD(X), ID(DIM(X))) IF F=NIL

  BEGIN FOR NUMBER IN 1...ITERATIONS
    P := -H.G
    (X,F,T) := LINMIN2 (X, FUN, P, MAXITER, TOL)
    (Y, G) := (GRAD(X)-G, GRAD(X))
    TEMP := T + Y.(H.Y) / (P.Y)
    H := H + (TEMP * OPROD(P,P) - OPROD(P,H.Y) - OPROD(H.Y,P)) / (P.Y)
    EXIT IF LENGTH(G) < TOLERANCE
  END

  EXIT (TRUE, NUMBER, X, F, G, H) IF NUMBER NE NIL
  EXIT (FALSE, ITERATIONS, X, F, G, H)
END

BEGIN LINMIN (X, FUN, P, MAXITER, TOL)
  DCL (POINTS(3), BEST.YET) : (T,F)
  POINTS := FIRST.POINTS (X, FUN, P)
  BEST.YET := POINTS (3)

  BEGIN FOR COUNT IN 3...MAXITERS
    NEW.T := ESTIMATED.MIN (POINTS)
    EXIT IF ABS (NEW.T - BEST.YET.T) < TOL
    BEST.YET := (NEW.T, FUN (X + P * NEW.T))
    POINTS := BEST3.OF (POINTS, BEST.YET)
  END

  EXIT (BEST.YET)
END

BEGIN FIRST.POINTS (X, FUN, P)
  TO := MIN (1. | .1/LENGTH(P))
  EXIT ((0, FUN(X)) | (TO, FUN(X+TO*P)) | (2*TO, FUN(X+@*TO*P)))
END

BEGIN ESTIMATED.MIN (POINTS)
  TMIN := MIN.IF.CONVEX (POINTS)
  EXIT (TMIN) IF TMIN NE NIL
  I := 1
  I := 3 IF POINTS(1).F > POINTS(3).F
  EXIT (POINTS(2).T + 2 * (POINTS(1).T - POINTS(2).T))
END

BEGIN MIN.IF.CONVEX (POINTS)
  DCL SLOPE(2), SUM(2)
  SLOPE(I) := (POINTS(I+1).F - POINTS(I).F) / (POINTS(I+1).T - POINTS.T)
  FOR I IN 1...2
    EXIT (NIL) IF SLOPE (1) < SLOPE (2)
  SUM (I) := POINTS(I).T + POINTS(I+1).T FOR I IN 1...2
  EXIT ((SLOPE(1) * SUM(2) - SLOPE(2) * SUM(1)) / (SLOPE(2) - SLOPE(1)))
END
BEGIN BEST.OF (POINTS, BEST.YET)
    EXIT (BEST.YET | POINTS(1...2)) IF BEST.YET.T < POINTS(1).T
    EXIT (POINTS(2|3) | BEST.YET) IF BEST.YET.T < POINTS(3).T
    EXIT (LOCAL.MIN (POINTS BEST.YET))
END

BEGIN LOCAL.MIN (G)
    DCL G(4) : (T,F)
    G := G(ORDER (G.T))
    EXIT (G(I-1 | I | I+1))
END

References


