Lawrence Berkeley National Laboratory
Recent Work

Title
SOLVING ONE-DIMENSIONAL BOUNDARY-VALUE PROBLEMS WITH BANDAID: A FUNCTIONAL
PROGRAMMING STYLE AND A COMPLEMENTARY SOFTWARE TOOL

Permalink
https://escholarship.org/uc/item/0443038j

Authors
Matlosz, M.
Newman, J.

Publication Date
1986-06-01
Submitted to Journal of the Electrochemical Society

SOLVING ONE-DIMENSIONAL BOUNDARY-VALUE PROBLEMS WITH BANDAID: A FUNCTIONAL PROGRAMMING STYLE AND A COMPLEMENTARY SOFTWARE TOOL

M. Matlosz and J. Newman

June 1986
DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.
Solving One-Dimensional Boundary-Value Problems with BandAid:  
A Functional Programming Style and a Complementary Software Tool

by

Michael Matlosz and John Newman

Materials and Molecular Research Division,  
Lawrence Berkeley Laboratory, and  
Department of Chemical Engineering,  
University of California, Berkeley, CA 94720

June 25, 1986

Abstract

This study examines the use of a functional programming style for writing computer programs to solve two-point boundary-value problems. The results of the study consist of a suggestion for an effective writing style, and of the implementation of a software tool that supports programming in that style.
Scope

The programming style presented in this article is a functional (or applicative) style. The computer programs are composed exclusively of constants and function subprograms. Complicated functional descriptions of boundary-value problems are created by defining composite function subprograms (which accept other function subprograms as parameters). Since the traditional mathematical language used to describe differential equations is also based largely on the concept of functions, it is hoped that such a style will result in a more direct correspondence between the differential equations to be solved and the computer code needed to solve them. In this way, the maintenance and extension of computer programs for complicated mathematical models should be easier than with more conventional codes.

To illustrate a specific application, this article explores the possibility of a functional style for computer programs to solve two-point boundary value problems. The functional style is associated with a high-level description of the problem in a calling routine. Details of the numerical algorithm must be contained in a supporting software tool that accepts problem descriptions written in this high-level functional style. A software tool for this purpose, BandAid, has been written. BandAid accepts the problem description from the calling routine, and it employs a conventional finite-difference technique to solve the equations. The functional style, then, is used in the construction of the calling programs, and this article (through the use of three examples) shows how to create these programs.

Conclusions and Significance

The programs presented here are models of a functional style that has been developed for describing two-point boundary-value problems. The use of this style facilitates the creation of efficient, reliable, and elegant computer programs for the mathematical modeling of physical processes by ordinary differential equations. The computer programs produced with this technique provide a general description of the problem statement that is independent of the numerical
method used to solve the equations. The complete separation of the method of solution from the
description of the equations to be solved adds to an increased lifetime for developed software and
permits the development of new numerical algorithms (solution of the equations) independently of
the construction of mathematical models (description of the equations).
Solving One-Dimensional Boundary-Value Problems with BandAid: A Functional Programming Style and a Complementary Software Tool

Summary

This manuscript describes a functional style for writing computer programs to solve one-dimensional boundary-value problems, and it presents a software tool, BandAid, that supports programming in this style. The programming methodology described here permits a static, self-documenting description of the differential equations to be placed directly in the program listing, thereby increasing the reliability of new programs and simplifying the maintenance and extension of existing ones. BandAid creates a finite-difference representation of the problem described in the program listing, and it uses the banded-matrix algorithm of Newman [1,2,3] to solve the resulting set of algebraic equations.

The writing style is introduced by discussing the use of BandAid to solve a set of problems representative of typical applications.

Introduction and Overview

This project was begun, in 1981, in an attempt to automate the determination of concentration and potential profiles in one-dimensional models of electrochemical systems. In general, determining these profiles involves solving boundary-value problems composed of sets of coupled, nonlinear, ordinary differential equations. Although a powerful finite-difference technique, Newman’s BAND algorithm [1,2,3], had been developed to solve these equations, little general-purpose software was available to aid preparation of the computer programs necessary to use it. In addition, maintenance of existing programs was difficult, since the differential equations being solved were not easily deduced from the code produced to solve them.

The goals of the project were, first, to identify a systematic approach to solving ordinary differential equations by Newman’s method, and, second, to create a software tool that would
facilitate reliable implementation of that approach for any arbitrary equation specification.

The results of the project consist of a recommendation for a specific writing style and of a software tool that supports the style. The writing style is a functional, or applicative, style [4]; the major portion of the calling program is written using function subprograms and constants only, and variables and procedure subprograms are avoided. Although limited, this approach is advantageous, since it permits the mathematical nature of the problem to be reflected directly in the computer program, and, therefore, the code generated for a particular application can be determined directly from the differential equations to be solved.

The software tool, BandAid, accepts code written in the functional style, interprets it, and uses the BAND algorithm to calculate a numerical solution to the problem described. The procedure is written in ANSI/IEEE-standard Pascal, and it is small and portable. Furthermore, the class of ordinary differential equations supported is general enough that BandAid can be used for many one-dimensional modeling problems in electrochemistry as well as in other physical sciences.

The purpose of this document is to explain both the writing style and its use in preparing computer programs that employ the BandAid routine. Examples are used to illustrate important points, and possible pitfalls and limitations of the program are indicated where they can be anticipated. As with any software package, the ultimate authority on what the program does or does not do is the source code itself, and the reader is encouraged to examine the listing of the BandAid procedure (provided in [5]) whenever questions or ambiguities arise. In order to guarantee the accuracy of the listings provided in this document, the text of all of the programs has been typeset directly from the machine-readable code used to check the examples. Nevertheless, a few bugs may have slipped through, and the reader should beware.

The BandAid procedure and all programs presented in this document are written in the Pascal programming language. Pascal was chosen, since it encourages modular, top-down programming design, compilers supporting numerical computation with Pascal are widely available, and
programs written carefully in Pascal are generally easy to read and maintain. Naturally, those readers interested in using BandAid for non-trivial extensions to the examples provided should be familiar with Pascal programming, but readers familiar with any high-level programming language should be able to follow the general discussion.

Cooper and Clancy’s *Oh! Pascal* [6] contains an excellent introduction to the language, and Cooper’s book *Standard Pascal* [7] provides an explanation of the International Organization for Standardization (ISO) version of Pascal approved by the American National Standards Institute (ANSI) and the Institute of Electrical and Electronics Engineers (IEEE). Readers interested in the development and philosophy of the language should consult the original work by Wirth [8,9,10] and by Jensen and Wirth [11]. Discussion of the advantages and disadvantages of Pascal may be found in references [12] and [13]. Formatting conventions and general rules for preparation of the source code found in this document follow the recommendations in [14] and [15].

**The Module Hierarchy**

Figure 1 illustrates the relationships among the three major program segments. The module AidMod, shown in the upper-left corner of the figure, contains the BandAid package that is used for all of the applications shown in this paper. The package includes the BandAid procedure as well as a few global definitions needed by the user program. The BAND algorithm of Newman, which performs most of the computational work, is defined as a subprogram within BandAid. Discussion of the BAND algorithm or of the construction of the BandAid code is beyond the scope of this paper. Interested readers, however, can find a complete description of BandAid and the AidMod module (as well as implementation details and a presentation of the numerical method) in [5].

The module CatIO contains user-defined routines (in this case, input/output routines for the catalytic reactor example that follows). These routines are stored in a separate module since they are uninteresting and tend to obscure the main text of the user program. If needed, complete listings of all of the source code can be found in [5].
Figure 1. The Module Hierarchy.
The module CatReac contains the user program. For the solution of differential equations with BandAid, this program must contain a functional description of the equations to be solved. This description, written as function subprogram Equation, is then passed to BandAid, where it is used during the determination of a converged solution. Although other information must be sent to BandAid as well, the key to the method is the structure of this function subprogram. This document, consequently, is directed toward the construction of these Equation functions.

A First Example: Concentration Distributions in a Catalytic Reactor

Problem Statement

Consider the simultaneous diffusion, convection, and reaction of two chemical species, A and B, at steady state in a packed-bed, catalytic reactor of length $L$. The catalyzed chemical reaction is reversible and can be represented by the following equation:

\[ 2A \rightleftharpoons B . \]

If the reaction is an elementary step, then the rate of disappearance of species A, $R_A$, can be determined directly from the stoichiometry as

\[ R_A = k_f c_A^2 - k_b c_B , \quad (1) \]

where $c_A$ and $c_B$ are the concentrations of the species and $k_f$ and $k_b$ are rate constants.

A material balance on each species yields the following set of coupled, nonlinear, ordinary differential equations:

\[ D_A \frac{d^2 c_A}{dz^2} - v \frac{dc_A}{dz} - R_A = 0 , \quad (2) \]

and
\[ D_B \frac{d^2 c_B}{dx^2} - v \frac{dc_B}{dx} + \frac{1}{2} R_A = 0, \]  

where \( v \) is the average fluid velocity and \( D_A \) and \( D_B \) are diffusion coefficients. Furthermore, if the inlet to and the outlet from the reactor are packed with inert material, the boundary conditions may be represented by the classical Wehner-Wilhelm-Danckwerts conditions [16,17]:

at \( x = 0, \)

\[ -D_A \frac{dc_A}{dx} + v(c_A - c_{A,0}) = 0, \]
\[ -D_B \frac{dc_B}{dx} + v(c_B - c_{B,0}) = 0, \]

and, at \( x = L, \)

\[ \frac{dc_A}{dx} = \frac{dc_B}{dx} = 0, \]

where \( c_{A,0} \) and \( c_{B,0} \) are the concentrations of species A and B in the feed to the reactor.

**Sample Program**

Figure 2 shows program CatReac, the calling program for this first example. It contains a function subprogram representing the differential equations to be solved along with the corresponding boundary conditions. In addition to the function subprogram, it lists parameter declarations (in the `const` and `var` statements) and a program body (near the end) containing the BandAid call. The next few sections describe the program, beginning with the declarations at the top\(^1\) and proceeding down to the program body.

---

\(^1\) The first line contains the non-standard `INHERIT` directive of VAX-11 Pascal. This directive alerts the compiler to the existence of other, separately-compiled modules (specifically AidMod and CatIO) that will be linked to the user program. Separate compilation of modules is permitted in VAX-11 Pascal, although its use is non-standard; other systems will be different. The directive is included in the listing here so that the program can be compiled exactly as written without modification. A more complete discussion of the use of modules in VAX-11 Pascal may be found in [5].
program CatReac( input, output );

const n = 2;
  Tolerance = 1e-6;

var DA, DB, cAo, cBo, v, L, kf, kb : RealNumber;
  jMax, j, ItMax, It, CPUTime : integer;
  Guess, Result, Residual : ValueArray;

function Equation ( i, j : integer;
  x, h : RealNumber;
  var NewResult : ValueArray;
  function cInterp ( k : integer;
    x : RealNumber;
    var Result : ValueArray ) : RealNumber;
  function cA( x : RealNumber ) : RealNumber;
begin cA := cInterp(1,x,NewResult) end;
  function cB( x : RealNumber ) : RealNumber;
begin cB := cInterp(2,x,NewResult) end;

function ddx( function G( x : RealNumber ) : RealNumber;
  x : RealNumber ) : RealNumber;
begin ddx := dnFdxn(1,G,x) end;

function d2dx2( function G(x : RealNumber ) : RealNumber;
  x : RealNumber ) : RealNumber;
begin d2dx2 := dnFdxn(2,G,x) end;

function Ra( x : RealNumber ) : RealNumber;
begin Ra := kf*cA(x)*cA(x) - kb*cB(x) end;

function DiffEQ( i integer; x : RealNumber ) : RealNumber;
begin case i of
  1: DiffEQ := DA*d2dx2(cA,x) - v*ddx(cA,x) - Ra(x); { = 0 }
  2: DiffEQ := DB*d2dx2(cB,x) - v*ddx(cB,x) + Ra(x)/2 { = 0 }
end { i cases }
function BC1( i : integer ) : RealNumber;
begin
  case i of
    1: BC1 := -DA*ddx(cA,0) + v*( cA(0) - cAo ) ; { = 0 } 
    2: BC1 := -DB*ddx(cB,0) + v*( cB(0) - cBo ) { = 0 }
  end { i cases }
end;

function BC2( i : integer ) : RealNumber;
begin
  case i of
    1: BC2 := ddx(cA,L); { = 0 }
    2: BC2 := ddx(cB,L) { = 0 }
  end { i cases }
end;

begin { body of Equation }
  if ( i = 1 ) then 
    Equation := BC1(i)
  else if ( i > 1 ) and ( i < jMax ) then 
    Equation := DiffEQ(i,x)
  else if ( i = jMax ) then 
    Equation := BC2(i)
end; { Equation }

begin { body of CatReac }
  ReadAndPrintParameters( DA, DB, cAo, cBo, v, L, kf, kb, jMax, ItMax );
  for j := 1 to jMax do
  begin
    Guess[1,j] := cAo;
    Guess[2,j] := cBo
  end;
  BandAid( n, jMax, ItMax, It, CPUMTime, 
          L, Tolerance, Equation, Guess, Result, Residual );
  PrintOut( Result, jMax, It, CPUMTime, L )
end { CatReac }
Special Data Types and Parameter Declarations

Two special data types, RealNumber and ValueArray, are employed throughout. A RealNumber is a floating-point number (usually double-precision), and a ValueArray is a matrix for storage of the values of the dependent variables. These types are defined externally, in a scope global to the program, and they are used in the same way as the standard data types integer, real, boolean, and char.

The const and var statements declare a number of important identifiers for the program, as follows: n is the number of equations and unknowns (two, in this case); Tolerance is a convergence tolerance; jMax is the number of mesh points; ItMax is a limit on the number of iterations; It is the actual number of iterations required; and L is the size of the domain. Guess stores the values of an initial guess of the solution, Result stores the answer returned by BandAid, and Residual stores the values of the equations at the final conditions. The meshpoint index j is used in the program body, and the remaining identifiers are parameters for the model.

Those identifiers declared as const are defined at compilation time, and those declared as var are read from the input or are assigned values during execution. With the exception of It, Result, and Residual, all remain unchanged during the execution of BandAid.
The Parameter List of the Equation Function

The name of the function subprogram describing the equations to be solved is Equation, and the structure of this subprogram illustrates the basic principles underlying the functional style: the description of the differential equations to be solved is made solely in terms of constants, functions, and composite functions (i.e., functions of functions). Variables are not employed as in conventional programming, but, rather, only as parameters to the functions. This is what is meant by a functional style.

Equation's parameter list contains two primitive functions (clnterp and dnFdxn) that form the basis of the problem description. All of the local functions declared in Equation will be defined directly in terms of these primitives (as, for example, cA or cB), or they will be defined in terms of composite functions of those functions (as, for example, RA or DiffEQ). These primitives, clnterp and dnFdxn, form the link between BandAid and the user's description of the problem to be solved. When the BandAid procedure requires information about the differential equations, it calls the Equation function, replaces the formal parameter list (i, j, x, h, NewResult, clnterp, dnFdxn) with its own actual parameters, and then interprets the function described therein. This process is performed repeatedly as the BandAid procedure searches for the solution to the equations.

The user program defines the Equation function over a continuous domain, x, and, for coupled problems, it distinguishes the individual equations by case statements in the function body. The first four arguments of Equation are simple value parameters: the particular equation under consideration (the ith equation) is indicated by the parameter i (an integer), the mesh point (the jth point) is indicated by the parameter j (an integer), a location in the domain by x (a floating-point number), and the mesh size by h (a floating-point number). The fifth parameter, NewResult (a ValueArray), contains the values of the dependent variables in BandAid.†

† Notice that the declaration is preceded by the keyword var. This indicates that the parameter is passed by name, rather than by value. Since the array is typically quite large, this provides a considerable improvement in efficiency.
The cInterp function, Equation's sixth parameter, takes three parameters of its own. The first, k (an integer), represents a particular dependent variable (the \( k \)-th dependent variable), the second, x (a floating-point number), represents a location in the domain, and the third, Result (a ValueArray), represents an array of values of the dependent variables at discrete mesh points along the domain.\(^{11}\) The Result matrix contains values of the variables along the domain, and the cInterp function interpolates (linearly) between two of these to determine the value of a variable at a point x. Thus, the cInterp function provides a conversion from the discrete representation of the equations required by the numerical method to the continuous representation of the equations supplied by the user.

Equation's final parameter is the dnFdxn function. It takes three parameters, one of which is itself a function. The first parameter, n (an integer), represents the order of the differentiation (0, 1, or 2). The second parameter, F (a function subprogram), represents any function defined continuously over the x domain. Finally, the third parameter, x (a floating-point number), represents the point in the domain where the derivative is to be taken.

**Local Declarations and the Body of the Equation Function**

Development of a typical Equation function proceeds as follows:

First, all of the dependent variables are defined and given names by suitable redefinition of the cInterp functions. In the example shown, these functions are called cA(x) and cB(x). These new functions are now functions of a single variable x, and they may be used in the dnFdxn function, or they may be employed in the definition of composite functions. For example, the reaction rate at a point, RA(x), is a composite defined in terms of cA(x) and cB(x). Of course, functions may also be composed from the dnFdxn primitive. In the program fragment, for example, a first-derivative operator, ddx(G,x), and a second-derivative operator, d2dx2(G,x), are defined from

\(^{11}\) Result is also a var parameter. (See above.)
$dnFdxn$, where $G(x)$ is any function or composite defined continuously over the domain. Once all of these intermediate functions and composite functions are declared, Equation itself is defined in terms of them in the function body. As shown in figure 2, a function (like DiffEQ) normally provides a case statement to indicate exactly which differential equation represents a given number in the set. The equation should be written as a residual, which will be equal to zero for a converged solution, since this is the convention assumed by BandAid. It is recommended, therefore, to emphasize this fact by the comment

\{ = 0 \}

following the equation definition.

In general, each mesh point in the domain may be represented by a different set of equations, but, normally, only three are needed: differential equations (DiffEQ), boundary conditions at \( x = 0 \) (BC1), and boundary conditions at \( x = L \) (BC2).

Although the content will vary from application to application, every Equation subprogram declared for use with BandAid has the same fundamental character. All of the intermediate functions declared simply define identifiers within the subprogram. There is no action; no assignments are made to storage locations within the computer. In fact, the Equation function does not do anything, nor does any function declared within it affect any aspect of the program outside of its scope. The only non-local identifiers used are those representing model constants (like \( D_A \) or \( k_f \)) that remain unchanged throughout execution. In short, the function is purely descriptive, the description is local, and there are no side effects. This static, insulated property is key to the method.

---

1 The term constant here does not necessarily mean that the identifiers are declared as const. They may, in fact, be declared as var, particularly if the BandAid call is only a part of a larger program. All that is required is that they remain unchanged during the execution of the BandAid procedure.
Naturally, the representation of the equations shown here is not the only possibility. A rather large number of functions has been defined to enhance readability and to illustrate the programming style, and, ultimately, each programmer will develop his own personal taste. For successful use of BandAid, the user program need not look exactly like the example. All that is required is that the Equation subprogram conform to the basic structure of the functional style.

The Program Body and Sample Output

The program body prints the parameters, sets a guess of the problem solution, calls the BandAid routine, and prints the results. Just as the types RealNumber and ValueArray are defined outside of the scope of the user program, the procedures BandAid, ReadAndPrintParameters, and PrintOut are assumed to be defined externally.¹ (Their texts are shown in [5].) Figure 3 shows sample output for a typical set of parameters.

Discussion

The program in figure 2 illustrates the basic principle of the programming style, namely, that the specification of the equations to be solved will be composed, as much as possible, of constants (such as DA or kf), functions (such as cA or d2dx2), and functions of functions (such as RA) (i. e., it contains no variables). This approach renders the description of the equations static, and, as a result, the statement of the problem is not affected by aspects of the program that are changing during execution. Furthermore, since (with the exception of constants) only local identifiers are employed, the description is insulated from side effects in other parts of the user program as well as from side effects in the BandAid package. By contrast, a traditional programming style does not distinguish between the description of the problem and the machinery involved in the problem solution: variables are employed directly in the user program. This has the unfortunate

¹ As noted earlier, these external definitions are permitted by the independent-compilation mechanism of VAX-11 Pascal. Specific features of that mechanism are described in [5].
**Figure 3. Sample Output from CatReac**

### CatReac Program Parameters

- **DA** = 1.0000E-06 cm²/sec
- **DB** = 1.0000E-06 cm²/sec
- **cAo** = 0.20000 mol/cm³
- **cBo** = 0.050000 mol/cm³
- **v** = 0.0050000 cm/sec
- **L** = 10.000 cm
- **kf** = 0.100000 cm³/mol-sec
- **kb** = 1.0000E-03 1/sec
- **jMax** = 26
- **ItMax** = 12

### Profile Listing

- Number of iterations = 7
- Execution time = 10410 milli-seconds

<table>
<thead>
<tr>
<th>j</th>
<th>x</th>
<th>cA</th>
<th>cB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0000</td>
<td>0.1999315</td>
<td>0.05003427</td>
</tr>
<tr>
<td>2</td>
<td>0.4000</td>
<td>0.09745880</td>
<td>0.1012706</td>
</tr>
<tr>
<td>3</td>
<td>0.8000</td>
<td>0.06423252</td>
<td>0.1178837</td>
</tr>
<tr>
<td>4</td>
<td>1.2000</td>
<td>0.05032645</td>
<td>0.1248368</td>
</tr>
<tr>
<td>5</td>
<td>1.6000</td>
<td>0.04368964</td>
<td>0.1281552</td>
</tr>
<tr>
<td>6</td>
<td>2.0000</td>
<td>0.04029397</td>
<td>0.1298530</td>
</tr>
<tr>
<td>7</td>
<td>2.4000</td>
<td>0.03849005</td>
<td>0.1307550</td>
</tr>
<tr>
<td>8</td>
<td>2.8000</td>
<td>0.03751184</td>
<td>0.1312441</td>
</tr>
<tr>
<td>9</td>
<td>3.2000</td>
<td>0.03697534</td>
<td>0.1315123</td>
</tr>
<tr>
<td>10</td>
<td>3.6000</td>
<td>0.03667924</td>
<td>0.1316604</td>
</tr>
<tr>
<td>11</td>
<td>4.0000</td>
<td>0.03651526</td>
<td>0.1317424</td>
</tr>
<tr>
<td>12</td>
<td>4.4000</td>
<td>0.03642427</td>
<td>0.1317879</td>
</tr>
<tr>
<td>13</td>
<td>4.8000</td>
<td>0.03637372</td>
<td>0.1318131</td>
</tr>
<tr>
<td>14</td>
<td>5.2000</td>
<td>0.03634563</td>
<td>0.1318272</td>
</tr>
<tr>
<td>15</td>
<td>5.6000</td>
<td>0.03633001</td>
<td>0.1318350</td>
</tr>
<tr>
<td>16</td>
<td>6.0000</td>
<td>0.03632132</td>
<td>0.1318393</td>
</tr>
<tr>
<td>17</td>
<td>6.4000</td>
<td>0.03631649</td>
<td>0.1318418</td>
</tr>
<tr>
<td>18</td>
<td>6.8000</td>
<td>0.03631380</td>
<td>0.1318431</td>
</tr>
<tr>
<td>19</td>
<td>7.2000</td>
<td>0.03631231</td>
<td>0.1318438</td>
</tr>
<tr>
<td>20</td>
<td>7.6000</td>
<td>0.03631148</td>
<td>0.1318443</td>
</tr>
<tr>
<td>21</td>
<td>8.0000</td>
<td>0.03631102</td>
<td>0.1318445</td>
</tr>
<tr>
<td>22</td>
<td>8.4000</td>
<td>0.03631076</td>
<td>0.1318446</td>
</tr>
<tr>
<td>23</td>
<td>8.8000</td>
<td>0.03631062</td>
<td>0.1318447</td>
</tr>
<tr>
<td>24</td>
<td>9.2000</td>
<td>0.03631054</td>
<td>0.1318447</td>
</tr>
<tr>
<td>25</td>
<td>9.6000</td>
<td>0.03631049</td>
<td>0.1318448</td>
</tr>
<tr>
<td>26</td>
<td>10.000</td>
<td>0.03631047</td>
<td>0.1318448</td>
</tr>
</tbody>
</table>
consequence that the description of the equations is dynamic (i.e., it depends upon the state of machine storage) and, therefore, it varies during program execution.

The approach taken here avoids this complication. All of the variables required for program execution are hidden in the supporting software tool, insulated from the user program, so that (providing that the software tool is bug-free) the user of the system need only examine a single, static description of the equations when evaluating the correctness of a program. The BandAid procedure is designed to use the static representation of the equations, exactly as written by the programmer, in the numerical solution of the problem. The procedure in no way modifies the meaning of the description during program execution. In fact, the user programs remain sufficiently independent of the details of the underlying BAND algorithm, that, even if that algorithm were to be changed (to make use of parallel processing, for example), it would not be necessary to rewrite them.

In addition, it is desirable that the functional description of the system of equations should resemble closely the notation used in their original development. Thus, for example,

$$D_A \frac{d^2 c_A}{dx^2} - v \frac{dc_A}{dx} - R_A = 0$$

is written in the DiffEQ function as

```diff
DiffEQ := DA*d2dx2(cA,x) - v*ddx(cA,x) - RA(x);  { 0 }
```

This notational similarity is important, because it can be used to make the functional specification self-documenting. That is, the program listing of the differential equations can be made sufficiently similar to the equations themselves that a reader, without the aid of comments, can immediately recognize them. This does not mean that comments will never be necessary or
helpful, but it does mean that much of the typical annotation of the program listing can be avoided. The benefit is that, since there are no comments, discrepancies between comments and code are eliminated\(^1\). Furthermore, this combining of documentation and code can help not only in the development of new programs, but in the maintenance of existing ones as well. With this approach, when the code is modified, the documentation is simultaneously revised to correspond to the new problem statement.

As demonstrated in the two examples that follow, the program listings for a wide variety of problems can be written to resemble very closely the code for CatReac. Therefore, although the differential equations in the following examples are more complicated than those for the catalytic reactor problem, the basic structure of each computer program is the same, and the complexity of the programs does not increase any faster than the complexity of the problems themselves.

\(^{1}\text{Debugging can be very difficult if comments do not accurately describe the code they claim to represent.}\)
A Second Example: Flow-Through Porous Electrodes

The second illustrative example also involves a packed-bed reactor. This time, however, the problem is electrochemical, and the kinetic expressions are more complicated. Although the problem is more difficult, the approach to writing the user program is practically identical to that taken in the first example, and, in fact, the program listing for this example was obtained by simple editing of the code for CatReac.

The Mathematical Model

Trainham and Newman [18] have developed the following model, in dimensionless form, for the concentration and potential distribution in a flow-through porous electrode for dilute metal-ion removal of a single species. The model contains two coupled equations: a mass balance,

$$- \frac{dN_R}{dy} = J_R ,$$

and a charge balance,

$$\frac{d^2\eta'}{dy^2} = P_2(J_R + J_S) .$$

\( \theta \) represents metal-ion concentration, \( \eta' \) electric driving force, and \( y \) distance through the packed bed. \( N_R \), the reactant-ion flux, is defined as

$$N_R = -D_R \frac{d\theta}{dy} + \theta ,$$

\( J_R \), a reaction-rate term for the main reaction (metal deposition), as

$$J_R = \frac{\theta - P_1 \exp\left(\frac{\alpha e_R (\alpha e_R + 1)\eta'}{\alpha e_R + 1}\right)}{1 + \exp(\eta')} ,$$

and, \( J_S \), a reaction-rate term for the side reaction (hydrogen evolution), as
The parameter $D'$ is a dimensionless axial dispersion coefficient. $P_1$ characterizes the backward term of the main-reaction rate, $P_2$ the relative importance of ohmic resistance to mass-transfer resistance, $P_3$ the rate of the side reaction, and $P_4$ the backward term in the side-reaction rate.

Boundary conditions at the inlet to and exit from the reactor are represented as follows:

at $y = 0$,

$$\theta - D' \frac{d\theta}{dy} = 1 \quad \text{and} \quad \frac{d\eta'}{dy} = -P_2 I^* ,$$  \hspace{1cm} (10)

and, at $y = \alpha L$,

$$\frac{d\theta}{dy} = \frac{d\eta'}{dy} = 0 .$$  \hspace{1cm} (11)

The parameter $\alpha$ is the reciprocal of a penetration depth into the reactor, and $I^*$ is the ratio of the current density to the limiting-current density that would exist if all of the metal-ion feed were completely reacted in the absence of a side reaction.

Figure 4 shows the user program for the solution of this boundary-value problem with BandAid, and figure 5 shows sample output. (The parameters chosen are those used by Trainham and Newman to model the behavior of a flow-through porous electrode for the removal of copper contamination from sulfuric acid solutions.)
program FlowThru( input, output );

const n = 2;
Tolerance = 1e-6;

var P1, P2, P3, P4, DP, AlphaAR, AlphaCR,
    AlphaAS, AlphaCS, AlphaL, IStar : RealNumber;
    jMax, i, ItMax, It, CPUPTime : integer;
    Guess, Result, Residual : ValueArray;

function Equation ( i, j : integer;
    y, h : RealNumber;
var NewResult : ValueArray;

function cInterp( k : integer;
    y : RealNumber;
var Result : ValueArray ) : RealNumber;

function dnFdyn( n : integer;
    function F ( y : RealNumber ) : RealNumber;
    y : RealNumber ) : RealNumber;

function T( y : RealNumber ) : RealNumber;
begin T := cInterp(1,y,NewResult) end;

function E( y : RealNumber ) : RealNumber;
begin E := cInterp(2,y,NewResult) end;

function ddy( function G( y : RealNumber ) : RealNumber;
    y : RealNumber ) : RealNumber;
begin ddy := dnFdyn(1,G,y) end;

function d2dy2( function G( y : RealNumber ) : RealNumber;
    y : RealNumber ) : RealNumber;
begin d2dy2 := dnFdyn(2,G,y) end;

function NR( y : RealNumber ) : RealNumber;
begin NR := -DP*ddy(T,y) + T(y) end;

function JR( y : RealNumber ) : RealNumber;
begin JR := d2dy2(T,y) end;

var Top, Bottom : RealNumber;
begin
  Top := T(y) - P1*exp( (AlphaAR/AlphaCR) + 1 )*E(y);
  Bottom := 1 + exp( E(y) );
  JR := Top/Bottom
end;

function JS( y : RealNumber ) : RealNumber; { ... Side-reaction rate }
var
  expl, exp2 : RealNumber;
begin
  expl := exp( -(AlphaCS/AlphaCR)*E(y) );
  exp2 := exp( (AlphaAS + AlphaCS)/AlphaCR )*E(y);
  JS := P3*expl*( 1 - P4*exp2 )
end;

function MassBalance( y : RealNumber ) : RealNumber;
begin
  MassBalance := -ddy(NR,y)' - JR(y) { = 0 }
end;

function ChargeBalance( y : RealNumber ) : RealNumber;
begin
  ChargeBalance := d2dy2(E,y) - P2*( JR(y) + JS(y) ) { = 0 }
end;

function UpStrmBC( i : integer ) : RealNumber;
begin
  case i of
    1: UpStrmBC := T(O) - DP*ddy(T,O) - 1; { = 0 }
    2: UpStrmBC := ddy(E,O) + P2*IStar { = 0 }
  end; { i cases }
end;

function DnStrmBC( i : integer ) : RealNumber;
begin
  case i of
    1: DnStrmBC := ddy(T,AlphaL); { = 0 }
    2: DnStrmBC := ddy(E,AlphaL) { = 0 }
  end; { i cases }
end;

begin { body of Equation }
  if ( j = 1 ) then
    Equation := UpStrmBC(i)
  else if ( j > 1 ) and ( j < jMax ) then
    case i of

1. Equation := MassBalance(y);
2. Equation := ChargeBalance(y)
end { i cases }

else if ( j = jMax ) then
  Equation := DnStrmBC(i)
end; { Equation }

begin { body of FlowThru }
  ReadAndPrintParameters( P1, P2, P3, P4, DP, AlphaAR, AlphaCR,
  AlphaAS, AlphaCS, AlphaL, IStar, jMax, ItMax );
  for j := 1 to jMax do
    begin
      Guess[1,j] := 1;
      Guess[2,j] := 0
    end;
  BandAid( n, jMax, ItMax, It, CPUTime, AlphaL,
            Tolerance, Equation, Guess, Result, Residual );
  PrintOut( Result, jMax, It, CPUTime, AlphaL )
end. { FlowThru }

Figure 5. Sample Output from FlowThru

FlowThru Program

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1.0490E-07</td>
</tr>
<tr>
<td>P2</td>
<td>-3.2540</td>
</tr>
<tr>
<td>P3</td>
<td>1.2470E-05</td>
</tr>
<tr>
<td>P4</td>
<td>5.8630E-09</td>
</tr>
<tr>
<td>DP</td>
<td>0.12170</td>
</tr>
<tr>
<td>AlphaAR</td>
<td>1.50</td>
</tr>
<tr>
<td>AlphaCR</td>
<td>0.50</td>
</tr>
<tr>
<td>AlphaAS</td>
<td>0.50</td>
</tr>
<tr>
<td>AlphaCS</td>
<td>0.50</td>
</tr>
<tr>
<td>AlphaL</td>
<td>8.6630</td>
</tr>
<tr>
<td>I-Star</td>
<td>0.95000</td>
</tr>
</tbody>
</table>
jMax = 26  
ItMax = 12

Profile Listing

Number of iterations = 6  
Execution time = 9310 milli-seconds

<table>
<thead>
<tr>
<th>j</th>
<th>x</th>
<th>T</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00000</td>
<td>0.9087793</td>
<td>-3.326501</td>
</tr>
<tr>
<td>2</td>
<td>0.34652</td>
<td>0.8824420</td>
<td>-2.377333</td>
</tr>
<tr>
<td>3</td>
<td>0.69304</td>
<td>0.5229007</td>
<td>-1.672223</td>
</tr>
<tr>
<td>4</td>
<td>1.0396</td>
<td>0.4103896</td>
<td>-1.139143</td>
</tr>
<tr>
<td>5</td>
<td>1.3861</td>
<td>0.3300385</td>
<td>-0.7275479</td>
</tr>
<tr>
<td>6</td>
<td>1.7326</td>
<td>0.2714530</td>
<td>-0.4029130</td>
</tr>
<tr>
<td>7</td>
<td>2.0791</td>
<td>0.2276855</td>
<td>-0.1418589</td>
</tr>
<tr>
<td>8</td>
<td>2.4256</td>
<td>0.1941693</td>
<td>0.07155833</td>
</tr>
<tr>
<td>9</td>
<td>2.7722</td>
<td>0.1678955</td>
<td>0.2483940</td>
</tr>
<tr>
<td>10</td>
<td>3.1187</td>
<td>0.1468550</td>
<td>0.3964782</td>
</tr>
<tr>
<td>11</td>
<td>3.4652</td>
<td>0.1295815</td>
<td>0.5214832</td>
</tr>
<tr>
<td>12</td>
<td>3.8117</td>
<td>0.1154263</td>
<td>0.6276105</td>
</tr>
<tr>
<td>13</td>
<td>4.1582</td>
<td>0.1034168</td>
<td>0.7180381</td>
</tr>
<tr>
<td>14</td>
<td>4.5048</td>
<td>0.09315665</td>
<td>0.7952169</td>
</tr>
<tr>
<td>15</td>
<td>4.8513</td>
<td>0.08431748</td>
<td>0.8610707</td>
</tr>
<tr>
<td>16</td>
<td>5.1978</td>
<td>0.07660056</td>
<td>0.9171343</td>
</tr>
<tr>
<td>17</td>
<td>5.5443</td>
<td>0.06981113</td>
<td>0.9646502</td>
</tr>
<tr>
<td>18</td>
<td>5.8908</td>
<td>0.06379091</td>
<td>1.004638</td>
</tr>
<tr>
<td>19</td>
<td>6.2374</td>
<td>0.05841585</td>
<td>1.037943</td>
</tr>
<tr>
<td>20</td>
<td>6.5839</td>
<td>0.05358760</td>
<td>1.065278</td>
</tr>
<tr>
<td>21</td>
<td>6.9304</td>
<td>0.04922702</td>
<td>1.087246</td>
</tr>
<tr>
<td>22</td>
<td>7.2769</td>
<td>0.04527167</td>
<td>1.104363</td>
</tr>
<tr>
<td>23</td>
<td>7.6234</td>
<td>0.04165978</td>
<td>1.117076</td>
</tr>
<tr>
<td>24</td>
<td>7.9700</td>
<td>0.03840698</td>
<td>1.125774</td>
</tr>
<tr>
<td>25</td>
<td>8.3165</td>
<td>0.03511203</td>
<td>1.130797</td>
</tr>
<tr>
<td>26</td>
<td>8.6630</td>
<td>0.03401371</td>
<td>1.132471</td>
</tr>
</tbody>
</table>
A Third Example: Flow Generated by a Rotating Disk

As a final example, the solution to the problem of fluid motion to a rotating disk is developed. This particular application is discussed quite thoroughly by White et al. [19], where Newman's method is compared to a number of other possible techniques, and interested readers may wish to consult that article for more detail.

Governing Equations

The von Kármán transformation [20] can be used to reduce the equations of motion and continuity in three dimensions \((r, \theta, \text{and} z)\) to the set of four coupled, nonlinear equations shown below, where \(\zeta = z \sqrt{\Omega/\nu}\):

\[
2F + \frac{dH}{d\zeta} = 0, \tag{12}
\]

\[
F^2 - G^2 + H \frac{dF}{d\zeta} - \frac{d^2 F}{d\zeta^2} = 0, \tag{13}
\]

\[
2FG + H \frac{dG}{d\zeta} - \frac{d^2 G}{d\zeta^2} = 0, \tag{14}
\]

and

\[
P = -\frac{1}{2}H^2 - 2F. \tag{15}
\]

The unknowns are related to the three components of the velocity vector and to the pressure according to the following relations:

\[
F = \frac{v_r}{r \Omega}, \tag{16}
\]

\[
G = \frac{v_\theta}{r \Omega}, \tag{17}
\]
\begin{equation}
H = \frac{v_z}{\sqrt{\nu \Omega}},
\end{equation}

and

\begin{equation}
P = \frac{P}{\mu \Omega},
\end{equation}

and boundary conditions on the velocity variables complete the specification of the problem:

\[ F(0) = H(0) = 0, \]
\[ G(0) = 1, \]
\[ F(\infty) = G(\infty) = 0. \]

The system contains a set of three ordinary differential equations (one first-order equation and two second-order equations) and one algebraic equation. Figure 6 illustrates how BandAid can be used to construct a solution to this problem,\(^1\) and figure 7 shows sample output from a typical run (more accurate values of the results can be found in reference 19).

Unlike the first two examples, the system of equations here is defined over a semi-infinite domain (0 < \( \zeta < \infty \)). Although the numerical method can only be used on a finite domain, it is expected that the effect of the rotating disk will not be felt far from the surface. Consequently, the program can be run with a number of different values of \( \zeta_{\text{max}} \) until changing \( \zeta_{\text{max}} \) no longer affects the solution to the problem. (In the run shown here, \( \zeta_{\text{max}} \) is 10.) In short, for sufficiently large domains, this solution to the bounded problem should be identical to that for the semi-infinite system. (This point is discussed in detail in [19].) With this approach, the BAND algorithm can be used not only for this problem but for many other boundary-value problems defined on semi-infinite domains as well.

---

\(^1\) Since \( H \) is one of the components of the velocity vector, the identifier \( dz \) (rather than \( h \)) represents the mesh size in this program.
Figure 6. Text of Program vonKarman

[INHERIT('AidMod.pen', 'KarmanIO.pen')]

program vonKarman(input, output);

const  
n = 4;
Tolerance = 1e-6;
L = 10;

var  
jMax, j, k, ItMax, It, CPUTime : integer;
Guess, Result, Residual : ValueArray;

function Equation (i, j : integer;
  z, dz : RealNumber;
  var NewResult : ValueArray);

function cInterp(k : integer;
  z : RealNumber;
  var Result : ValueArray) : RealNumber;

function dFdz( n : integer;
  function F ( z : RealNumber ) : RealNumber;
  z : RealNumber ) : RealNumber;

function d2Fdz2( function G( z : RealNumber ) : RealNumber;
  z : RealNumber ) : RealNumber;

begin  
  F := cInterp(1,z,NewResult)  
  G := cInterp(2,z,NewResult)  
  H := cInterp(3,z,NewResult)  
  P := cInterp(4,z,NewResult)  
  ddz := dFdz(1,G,z)  
  d2dz2 := dFdz(2,G,z)  
end;
begin

function EQ1( z : RealNumber ) : RealNumber;
begin
  EQ1 := 2*F(z) + ddz(H,z) \{ = 0 \}
end;

function EQ2( z : RealNumber ) : RealNumber;
begin
  EQ2 := sqr(F(z)) - sqr(G(z)) + H(z)*ddz(F,z) - d2dz2(F,z) \{ = 0 \}
end;

function EQ3( z : RealNumber ) : RealNumber;
begin
  EQ3 := 2*F(z)*G(z) + H(z)*ddz(G,z) - d2dz2(G,z) \{ = 0 \}
end;

function EQ4( z : RealNumber ) : RealNumber;
begin
  EQ4 := P(z) + sqr(H(z))/2 + 2*F(z) \{ = 0 \}
end;

begin \{ body of Equation \}
  if \( j = 1 \) then
    case \( i \) of
      1: Equation := H(0); \{ = 0 \}
      2: Equation := F(0); \{ = 0 \}
      3: Equation := G(0) - 1; \{ = 0 \}
      4: Equation := EQ4(0)
    end \{ i cases \}
  else if \( j > 1 \) and \( j < jMax \) then
    case \( i \) of
      1: Equation := EQ1(z);
      2: Equation := EQ2(z);
      3: Equation := EQ3(z);
      4: Equation := EQ4(z)
    end \{ i cases \}
  else if \( j = jMax \) then
    case \( iMax \) then
      1: Equation := EQ1(L);
      2: Equation := F(L); \{ = 0 \}
      3: Equation := G(L); \{ = 0 \}
      4: Equation := EQ4(L)
    end \{ i cases \}
end; \{ Equation \}
begin { body of vonKarman }

ReadAndPrintParameters( jMax, ItMax );

for j := 1 to jMax do
  for k := 1 to n do Guess[k,j] := 0;
  BandAid( n, jMax, ItMax, It, CPUMtime, L,
             Tolerance, Equation, Guess, Result, Residual );
  PrintOut( Result, jMax, It, CPUMtime, L );

end. { vonKarman }

Figure 7. Sample Output from vonKarman

vonKarman Program

Parameters

jMax = 101
ItMax = 12

Profile Listing

Number of iterations = 10
Execution time = 95510 milli-seconds

<table>
<thead>
<tr>
<th>j</th>
<th>z</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00000</td>
<td>0.000000</td>
<td>1.00000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>2</td>
<td>0.10000</td>
<td>0.04615171</td>
<td>0.9384873</td>
<td>-0.005084297</td>
<td>-0.09231635</td>
</tr>
<tr>
<td>3</td>
<td>0.20000</td>
<td>0.08349592</td>
<td>0.8778719</td>
<td>-0.01846069</td>
<td>-0.1671622</td>
</tr>
<tr>
<td>4</td>
<td>0.30000</td>
<td>0.1131414</td>
<td>0.8188329</td>
<td>-0.03848266</td>
<td>-0.2270233</td>
</tr>
<tr>
<td>5</td>
<td>0.40000</td>
<td>0.1361088</td>
<td>0.7618700</td>
<td>-0.05371725</td>
<td>-0.2742476</td>
</tr>
<tr>
<td>6</td>
<td>0.50000</td>
<td>0.1533290</td>
<td>0.7073362</td>
<td>-0.09292619</td>
<td>-0.3109756</td>
</tr>
<tr>
<td>7</td>
<td>0.60000</td>
<td>0.1656438</td>
<td>0.6554659</td>
<td>-0.1250488</td>
<td>-0.3391061</td>
</tr>
<tr>
<td>8</td>
<td>0.70000</td>
<td>0.1738085</td>
<td>0.6063983</td>
<td>-0.1591837</td>
<td>-0.3602868</td>
</tr>
</tbody>
</table>
The vonKarman program completes a basic introduction to the programming technique. Most problems solved by the BAND method can be constructed to fit into the form of the programs shown here, and many other problems can be solved by simple modification of the examples. Additional information necessary for non-trivial extensions to examples may be found in [5].

**Conclusions and Perspectives on Future Work**

The programs in this document are models of a functional programming style that has been developed for solving one-dimensional boundary-value problems by finite-difference techniques. This style produces efficient, reliable, and elegant computer programs, and, combined with the supporting BandAid procedure, it facilitates greatly the development of sophisticated mathematical models of physical processes. In the next few years, as the technology of very large scale integration (VLSI) advances, supercomputers containing many thousands of processors operating in parallel will become available. Concurrent aspects of Newman's algorithm can be used to take advantage of this increased computing power, and, with this extension, the non-sequential format advocated here will be particularly useful. In addition, although this manuscript concentrates on finite-difference procedures, the overall approach is general, and it is likely that the methods shown here could be used effectively with other numerical techniques, such as collocation.

<table>
<thead>
<tr>
<th>n</th>
<th>x</th>
<th>h_1</th>
<th>h_2</th>
<th>y_1</th>
<th>y_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>93</td>
<td>9.2000</td>
<td>1.373720E-04</td>
<td>1.788158E-04</td>
<td>-0.8817505</td>
<td>-0.3890167</td>
</tr>
<tr>
<td>94</td>
<td>9.3000</td>
<td>1.144586E-04</td>
<td>1.498220E-04</td>
<td>-0.8817756</td>
<td>-0.3889930</td>
</tr>
<tr>
<td>95</td>
<td>9.4000</td>
<td>9.348022E-05</td>
<td>1.23615E-04</td>
<td>-0.8817963</td>
<td>-0.3889693</td>
</tr>
<tr>
<td>96</td>
<td>9.5000</td>
<td>7.427352E-05</td>
<td>9.722051E-05</td>
<td>-0.8818130</td>
<td>-0.3889456</td>
</tr>
<tr>
<td>97</td>
<td>9.6000</td>
<td>5.668893E-05</td>
<td>7.420298E-05</td>
<td>-0.8818260</td>
<td>-0.3889219</td>
</tr>
<tr>
<td>98</td>
<td>9.7000</td>
<td>4.058949E-05</td>
<td>5.312956E-05</td>
<td>-0.8818356</td>
<td>-0.3888982</td>
</tr>
<tr>
<td>99</td>
<td>9.8000</td>
<td>2.584979E-05</td>
<td>3.383604E-05</td>
<td>-0.8818423</td>
<td>-0.3888746</td>
</tr>
<tr>
<td>100</td>
<td>9.9000</td>
<td>1.235501E-05</td>
<td>1.617266E-05</td>
<td>-0.8818460</td>
<td>-0.3888509</td>
</tr>
<tr>
<td>101</td>
<td>10.0000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>-0.8818472</td>
<td>-0.3888272</td>
</tr>
</tbody>
</table>
Acknowledgements

Special thanks are due to Mr. Bill Hogan for many valuable discussions regarding this work and to Mr. John Cain and Mr. Paul Shain for their willingness to work with earlier (incomplete) versions of BandAid.

This work was supported by the United States Department of Energy under Contract No. DE-AC03-76SF00098 through the Director, Office of Energy Research, Office of Basic Energy Sciences, Chemical Sciences Division, and through the Assistant Secretary of Conservation and Renewable Energy, Office of Energy Systems Research, Energy Storage Division.
List of Symbols

A  Reactant species in catalytic reactor problem
B  Product species in the catalytic reactor problem
c  Concentration, mol/cm$^3$
$c_A$  Concentration of species A, mol/cm$^3$
$c_{A,s}$  Feed concentration of A, mol/cm$^3$
$c_B$  Concentration of species B, mol/cm$^3$
$c_{B,s}$  Feed concentration of B, mol/cm$^3$
$D$  Diffusion coefficient, cm$^2$/s
$D_A$  Diffusion coefficient of species A, cm$^2$/s
$D_B$  Diffusion coefficient of species B, cm$^2$/s
$D'$  Dimensionless dispersion coefficient in flow-through porous electrode model
$F$  Radial velocity in the von Kármán transformation for flow to a rotating disk
$G$  Angular velocity in the von Kármán transformation for flow to a rotating disk
$h$  Mesh size
$H$  Axial velocity in the von Kármán transformation for flow to a rotating disk
$I^*$  Dimensionless current density to porous electrode
$J_R$  Dimensionless rate term for the main reaction in the porous electrode
$J_S$  Dimensionless rate term for the side reaction in the porous electrode
$k_b$  Rate constant for reverse reaction in catalytic reactor, s$^{-1}$
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_f$</td>
<td>Rate constant for forward reaction in catalytic reactor, cm$^3$/mol–s</td>
</tr>
<tr>
<td>$L$</td>
<td>Length of catalytic reactor, cm</td>
</tr>
<tr>
<td>$L$</td>
<td>Porous electrode reactor length, cm</td>
</tr>
<tr>
<td>$N_R$</td>
<td>Dimensionless reactant-ion flux</td>
</tr>
<tr>
<td>$P$</td>
<td>Pressure variable in the von Kármán transformation for flow to a rotating disk</td>
</tr>
<tr>
<td>$P$</td>
<td>Pressure, dyne/cm$^2$</td>
</tr>
<tr>
<td>$P_1$</td>
<td>Parameter characterizing the backward term of the main reaction in the flow-through porous electrode</td>
</tr>
<tr>
<td>$P_2$</td>
<td>Parameter characterizing the relative importance of ohmic resistance to mass-transfer resistance in the flow-through porous electrode</td>
</tr>
<tr>
<td>$P_3$</td>
<td>Parameter characterizing the rate of side reaction in the flow-through porous electrode</td>
</tr>
<tr>
<td>$P_4$</td>
<td>Parameter characterizing the backward term in the side reaction in the flow-through porous electrode</td>
</tr>
<tr>
<td>$r$</td>
<td>Radial distance, cm</td>
</tr>
<tr>
<td>$R_A$</td>
<td>Rate of disappearance of species A, mol/cm$^3$–s</td>
</tr>
<tr>
<td>$v$</td>
<td>Velocity of fluid through reactor, cm/s</td>
</tr>
<tr>
<td>$v_r$</td>
<td>Radial velocity, cm/s</td>
</tr>
<tr>
<td>$v_\theta$</td>
<td>Angular velocity at rotating disk, cm/s</td>
</tr>
<tr>
<td>$v_z$</td>
<td>Axial velocity, cm/s</td>
</tr>
<tr>
<td>$x$</td>
<td>Distance through reactor, cm</td>
</tr>
<tr>
<td>$y$</td>
<td>Dimensionless distance through porous electrode</td>
</tr>
<tr>
<td>$y$</td>
<td>Normal distance from surface of rotating disk, cm</td>
</tr>
<tr>
<td>$z$</td>
<td>Axial distance</td>
</tr>
<tr>
<td>$z$</td>
<td>Distance variable, cm</td>
</tr>
<tr>
<td>Symbol</td>
<td>Definition</td>
</tr>
<tr>
<td>--------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Reciprocal of penetration depth into the porous electrode, cm$^{-1}$</td>
</tr>
<tr>
<td>$\alpha L$</td>
<td>Dimensionless length of flow-through porous electrode</td>
</tr>
<tr>
<td>$\alpha_{aR}$</td>
<td>Anodic transfer coefficient for the main reaction in the porous electrode</td>
</tr>
<tr>
<td>$\alpha_{aS}$</td>
<td>Anodic transfer coefficient for the side reaction in the porous electrode</td>
</tr>
<tr>
<td>$\alpha_{cR}$</td>
<td>Cathodic transfer coefficient for the main reaction in the porous electrode</td>
</tr>
<tr>
<td>$\alpha_{cS}$</td>
<td>Cathodic transfer coefficient for the side reaction in the porous electrode</td>
</tr>
<tr>
<td>$\zeta$</td>
<td>Dimensionless axial distance</td>
</tr>
<tr>
<td>$\zeta_{\text{max}}$</td>
<td>Maximum axial distance</td>
</tr>
<tr>
<td>$\eta'$</td>
<td>Dimensionless potential driving force in flow-through porous electrode model</td>
</tr>
<tr>
<td>$\theta$</td>
<td>Dimensionless reactant ion concentration</td>
</tr>
<tr>
<td>$\mu$</td>
<td>Viscosity, g/cm-s</td>
</tr>
<tr>
<td>$\nu$</td>
<td>Kinematic viscosity, cm$^2$/s</td>
</tr>
<tr>
<td>$\xi_{\text{max}}$</td>
<td>Maximum dimensionless distance from surface of rotating disk</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>Rotation speed, rad/s</td>
</tr>
</tbody>
</table>
References


This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.