UIUCDCS-R-80-1013

ONE-LEG FORMULAS FOR STIFF DIFFERENTIAL EQUATIONS

by

Q. M. Sheikh

August 1980

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

DISCLAIMER

This book was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or 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 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. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Supported in part by the U. S. Department of Energy under grant DE-AC02-76ER02383.A003, and submitted in partial fulfillment of the requirements of the Graduate College for the degree of Master of Science.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or 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 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. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

ACKNOWLEDGEMENT

The author wishes to express his gratitude to Professor Daniel Watanabe for his invaluable ideas, suggestions, criticisms and supervision. Without them, this thesis could not have been possible. The author would like to thank Rahul Pandit and every single member of his research group for their unfailing help, suggestion and support. The author is also indebted to Mrs. Barbara Armstrong for her help in typing the thesis. Special thanks are due, of course, to his parents, Sheikh Mohammad Shafi and Safia Begum, for every thing they have done, and for this the author would like to acknowledge his great debt to them.

TABLE OF CONTENTS

	<u>Page</u>
1. INTRODUCTION.	1
2. ONE-LEG FORMULAS.	3
2.1. Derivation.	3
2.2. Regions of Absolute Stability	5
2.3. Effective Growth Per Step	8
3. IMPLEMENTATION.	11
3.1. Nordsieck Representation.	11
3.2. Coefficients.	14
3.3. Updating the Jacobian	14
4. NUMERICAL RESULTS	16
4.1. Van der Pol Oscillator.	16
4.2. Krogh's Problem	19
4.3. Test Problems Suggested by Lindberg	20
LIST OF REFERENCES.	27
APPENDIX.	28

1. INTRODUCTION

There are several effective codes available for solving stiff ordinary differential equations. However, for certain problems these codes will produce reasonable looking solutions which are not close to the desired solution. This difficulty can arise in several ways including a step control that selects a stepsize so large that an active region of the solution is missed entirely, or a formula that is stable for unstable problems and completely ignores increasing components of the solutions. Unfortunately, the backward differentiation formulas which are the most widely used stiff formulas suffer from this disadvantage. For example, the backward Euler formula

$$y_{n+1} - y_n = hf(x_{n+1}, y_{n+1}), \quad (1.1)$$

when applied to the scalar test problem

$$y' = \lambda y, \quad y(0) = 1, \quad (1.2)$$

produces a decreasing numerical solution $\{y_n\}$ if $|h\lambda - 1| > 1$. However, the exact solution of the test problem,

$$y(x) = \exp(\lambda x),$$

is decreasing only if $\text{Re}(\lambda) < 0$.

Lindberg [2] gives test problems for which the eigenvalues change from large negative values to large positive values. Linear multistep formulas which are stable in large portions of the right half plane fail

to detect this change in the eigenvalues unless the step size h is extremely small. This thesis presents new one-leg multistep formulas which are generalizations of the backward differentiation formulas and whose regions of instability are potentially infinitely large.

2. ONE-LEG FORMULAS

2.1. Derivation

Given the initial value problem

$$y' = f(x, y) \quad y(0) = y_0, \quad (2.1)$$

we wish to generate a sequence $\{y_n\}$ which approximates the sequence of exact values $\{y(x_n)\}$ on the equispaced mesh $\{x_n\}$ with fixed stepsize h . Given the k past values $y_{n-1}, y_{n-2}, \dots, y_{n-k}$, we seek a polynomial $p_k(x)$ of degree k which interpolates these values and the future value y_n and choose y_n so that $p_k(x)$ satisfies the differential equation exactly at the collocation point $x_n + \theta_k h$. Thus

$$p_k(x_{n-i}) = y_{n-i}, \quad i = 0, 1, \dots, k, \quad (2.2)$$

$$p_k'(x_n + \theta_k h) = f(x_n + \theta_k h, p_k(x_n + \theta_k h)). \quad (2.3)$$

If we write

$$p_k(x) = \sum_{i=0}^k w_i(x) y_{n-i}, \quad (2.4)$$

where $w_i(x)$ is the Lagrange weight function

$$w_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^k (x - x_{n-j}) / (x_{n-i} - x_{n-j}), \quad (2.5)$$

we can write the collocation condition 2.3 as the k -step one-leg formula

$$\sum_{j=0}^k \alpha_j y_{n-j} = hf(x_n + \theta_k h, \sum_{j=0}^k \beta_j y_{n-j}), \quad (2.6)$$

where the coefficients α_j and β_j are given by

$$\alpha_j = w_j'(x_n + \theta_k h), \quad (2.7)$$

$$\beta_j = w_j(x_n + \theta_k h). \quad (2.8)$$

The generating polynomials for the one leg method are

$$\rho(\xi) = \sum_{j=0}^k \alpha_j \xi^{k-j}, \quad (2.9)$$

$$\sigma(\xi) = \sum_{j=0}^k \beta_j \xi^{k-j}. \quad (2.10)$$

Note that the choice $\theta_k = 0$ yields the k -step backward differentiation formula.

The local truncation error for the one-leg formula 2.6 is the sum of the errors due to differentiating and interpolating p_k at the collocation point. The differentiation error is the dominant term and has the form

$$w'(x) h y^{(k+1)}(\xi(x))/(k+1)! + w(x) h \frac{d}{dx} y^{(k+1)}(\xi(x))/(k+1)!,$$

where $w(x) = (x-x_n)(x-x_{n-1})\dots(x-x_{n-k})$. The first term is of $O(h^{k+1})$

and the second of order $O(h^{k+2})$. Hence the dominant term in the local

truncation error has the form $C h^{k+1} y^{(k+1)}(x)$ as in the case of the backward differentiation formula, and the k -step formula is of order k .

2.2. Regions of Absolute Stability

If we apply the one-leg formula 2.6 to the model equation $y' = \lambda y$, we obtain a difference equation whose characteristic equation is

$$\rho(\xi) - \mu \sigma(\xi) = 0,$$

where $\mu = h\lambda$. The locus in the μ -plane for which a root of this equation has magnitude one is obtained by plotting

$$\mu = \rho(e^{i\phi}) / \sigma(e^{i\phi}), \quad 0 \leq \phi < 2\pi.$$

The region of absolute stability is the exterior of this curve. Clearly it is desirable to maximize the size of the locus in the right half plane. By varying the collocation point, we can vary the size of the locus.

The rightmost point of the curve corresponds to $\phi = \pi$. If we choose the collocation point so that $\sigma(e^{i\pi}) = 0$, then this point is at infinity. There are k such choices of the collocation point, and we chose the point satisfying $-1 < \theta < 0$. For $k = 1$, this choice yields a formula of order 2. In order to fix the order at 1 to prevent our code from "chattering" between the 1- and 2-step formulas, we chose $\theta_1 = -0.45$ so that the rightmost point is finite. Figure 2.1 presents the region of absolute stability for $k = 5$. The rightmost point is finite

Figure 2.1. Region of Absolute Stability for One-Leg Formula

Figure 2.2. Region of Instability Extending in the Right Half Plane

because of the finite precision of the value of the collocation point. The portion of the curve lying in the left half plane is shown in Figure 2.2. The regions of stability for $k = 1, 2, 3, 4$, and 6 are similar, but the regions for $k = 1$ and 2 lie completely in the right half plane.

2.3. Effective Growth Per Step

Although the region of absolute stability of a one-leg formula is large, the amplification factor is essentially 1 in most of the region. For example, the factor for $\mu = 7800$ and $k = 5$ is 1.001, and the sequence $\{y_n\}$ would increase by a factor of only 1.001 per step. A fairer comparison of the one-leg and backward differentiation formulas is obtained by comparing the loci given by

$$\mu = \rho((1+\epsilon)e^{i\phi})/\sigma((1+\epsilon)e^{i\phi}), \quad 0 < \phi < \pi.$$

Figures 2.3 and 2.4 present the loci for $\epsilon = 1.05$ for the one-leg and backward differentiation formulas. The loci for the former are approximately one order of magnitude larger than those of the latter.

Figure 2.3. Regions of Instability for One-Leg Formulas for $\epsilon = 1.05$

Figure 2.4. Regions of Instability of Backward Differentiation Formulas for $\epsilon = 1.05$

3. IMPLEMENTATION

Let $\underline{y}_n = [y_n, y_{n-1}, \dots, y_{n-k}]^T$ be the vector of saved information. An initial approximation $\underline{y}_{n,(0)}$ to $\underline{y}_n$ is predicted by extrapolating the polynomial defined by $\underline{y}_{n-1}$ to x_n . This approximation is corrected using

$$\underline{y}_n = \underline{y}_{n,(0)} + \omega \underline{1}_y, \quad (3.1)$$

where $\underline{1}_y = [1, 0, \dots, 0]^T$ and ω is chosen to satisfy Equation 2.6 written in the form

$$F(\underline{y}_{n,(0)} + \omega \underline{1}_y) = 0, \quad (3.2)$$

where

$$\begin{aligned} F(\underline{y}) &= hf(p_k(x_n + \theta_k h)) - hp'_k(x_n + \theta_k h) \\ &= hf\left(\sum_{j=0}^k \beta_j y_{n-j}\right) - \sum_{j=0}^k \alpha_j y_{n-j}. \end{aligned} \quad (3.3)$$

The dependence of f on x has been suppressed for convenience. Newton's method is used to solve this nonlinear equation for ω .

3.1. Nordsieck Representation

The Nordsieck representation of the saved information,

$$\underline{s}_n = [y_n, hy_n^{(1)}, \dots, h^{k-1}y_n^{(k-1)} / (k-1)!]^T, \quad (3.4)$$

is convenient to use in practice. We can transform the preceding

formulation into the Nordsieck form using the matrix T such that

$\underline{a}_n = T \underline{y}_n$. The predictor assumes the form

$$\underline{a}_{n,(0)} = P \underline{a}_{n-1}, \quad (3.5)$$

where P is the pascal triangle with $p_{ij} = \binom{j}{i}$. The corrector assumes the form

$$\underline{a}_n = \underline{a}_{n,(0)} + \omega \underline{l}_N, \quad (3.6)$$

where $\underline{l}_N = T \underline{l}_y$ is the first column of T and ω is chosen to satisfy

$$F(\underline{a}_{n,(0)} + \omega \underline{l}_N) = 0, \quad (3.7)$$

where

$$F(\underline{a}) = \text{hf} \left(\sum_{j=0}^k (\theta_k)^j (\underline{a})_j \right) - \sum_{j=0}^k j(\theta_k)^{j-1} (\underline{a})_j, \quad (3.8)$$

and $(\underline{a})_i$ is the i th component of $\underline{a}$. This nonlinear equation is solved using the Newton iteration

$$\omega_{(m+1)} = \omega_{(m)} - [(\partial F / \partial \underline{a}) \cdot \underline{l}_N]^{-1} F(\underline{a}_{n,(0)} + \omega_{(m)} \underline{l}_N). \quad (3.9)$$

By setting $\underline{a}_{n,(m)} = \underline{a}_{n,(0)} + \underline{l}_N \omega_{(m)}$, we can rewrite this equation as

$$\underline{a}_{n,(m+1)} = \underline{a}_{n,(m)} - [(\partial F / \partial \underline{a}) \cdot \underline{l}_N]^{-1} F(\underline{a}_{n,(m)}) \underline{l}_N. \quad (3.10)$$

The Jacobian for this iteration is

$$(\partial F / \partial \underline{a}) \cdot \underline{1}_N = h \sum_{j=0}^k (\theta_k)^j (\underline{1}_N)_j \partial f / \partial y - \sum_{j=1}^k j (\theta_k)^{j-1} (\underline{1}_N)_j. \quad (3.11)$$

The second term on the right reduces to 1 for the backward differentiation formula ($\theta_k = 0$). Hence it is convenient to define the scaled coefficients

$$\underline{1}_a = \underline{1}_N / c(\theta_k), \quad (3.12)$$

where

$$c(\theta_k) = \sum_{i=1}^k 1(\theta_k)^{i-1} (\underline{1}_N)_i, \quad (3.13)$$

so that this term becomes 1. The scaled formulation

$$\underline{a}_n = \underline{a}_{n,(0)} + \omega \underline{1}_a, \quad (3.14)$$

$$\underline{a}_{n,(m+1)} = \underline{a}_{n,(m)} - [(\partial F / \partial \underline{a}) \cdot \underline{1}_a]^{-1} F(\underline{a}_{n,(m)}) \underline{1}_a, \quad (3.15)$$

$$(\partial F / \partial \underline{a}) \cdot \underline{1}_a = h \sum_{j=0}^k (\theta_k)^j (\underline{1}_a)_j / c(\theta_k) \partial f / \partial y - 1, \quad (3.16)$$

is almost identical to the Nordsieck formulation of the backward differentiation formulas. Therefore it is relatively easy to adapt a code like DIFSUB to the one-leg formulas.

3.2. Coefficients

Table 3.1 lists the coefficients for the k -step one-leg formulas for $k = 1, 2, \dots, 6$. In the appendix these coefficients are given in IBM double precision as a part of the code DIFSOL.

3.3. Updating the Jacobian

The Newton iteration requires the solution of the linear system of equations $Wx = b$, where $W = (hq) \partial f / \partial y - I$. The Jacobian W must be updated whenever the stepsize h is changed. This updating can be performed very efficiently using a scheme given by Enright [1]. We write Equation 3.15 as

$$\underline{a}_{n,(m+1)} = \underline{a}_{n,(m)} - (hq) [\partial f / \partial y - (1/hq)I]^{-1} F(\underline{a}_{n,(m)}) \underline{I}_n,$$

and set $W = \partial f / \partial y - (1/hq)I$. Instead of using the typical LU factorization, we use the factorization $W = LHL^{-1}$, where L is a lower triangular matrix and H is an upper Hessenberg matrix. If the stepsize is changed from h to h' , then

$$\partial f / \partial y - (1/h'q)I = L(H + (1/hq - 1/h'q)I)L^{-1}.$$

Hence only the diagonal elements of H must be changed, and this requires only $2n$ additions for an $n \times n$ system.

Table 3.1. Coefficients for k-step One-Leg Formulas

k =	1	2	3	4	5	6
θ	- 0.45	- 0.2928932188	- 0.1771243445	- 0.1076102858	- 0.0649539076	- 0.0387480654
l_0	1.0	0.8284271247	0.6689987243	0.5606206764	0.4881417441	0.4386455474
l_1	1.0	1.2426406871	1.2264976613	1.1679597425	1.1145903158	1.0746815913
l_2		0.4142135623	0.6689987243	0.8175718197	0.9152657702	0.9893894015
l_3			0.1114997873	0.2335919485	0.3457670687	0.4477839963
l_4				0.0233591948	0.0610177180	0.1066152372
l_5					0.0040678478	0.0127938284
l_6						0.0006092299

4. NUMERICAL RESULTS

Different test problems were solved using DIFSUB, DIFSOL, and OL. DIFSOL implements the one-leg formulas described above and uses the technique for updating the Jacobian discussed in Section 3.3. OL implements the one-leg formulas but uses the same subroutines as DIFSUB for solving linear systems. This enables us to compare the efficiency of the technique for updating the Jacobian discussed in Section 3.3 with the one used in DIFSUB. The first two problems are such that any existing package for solving stiff systems should give reasonable results. These two problems were solved to compare the efficiency of DIFSOL with DIFSUB. The last two problems are the ones described in Lindberg's paper. These results show that DIFSOL can detect large positive eigenvalues for reasonable error tolerance per step.

4.1. Van der Pol Oscillator

The van der Pol oscillator is given by the following system of ordinary differential equations

$$\begin{aligned} y_1' &= y_2, & y_1(0) &= 2, \\ y_2' &= 10^2(1 - y_1^2)y_2 - 10^3y_1, & y_2(0) &= 0, \\ x &\in [0, 0.163]. \end{aligned}$$

Table 4.1 gives the results obtained by solving the above system of ordinary differential equations using DIFSUB, DIFSOL, and OL. The statistics reported in Table 4.1 are:

EPS	requested error tolerance;
FCN CALLS	total number of function evaluations;
JACOB CALLS	total number of Jacobian evaluations;
TOTAL TIME	time in seconds, on IBM 360/75;
MAX REL ERR	maximum relative error per step as estimated by the respective codes.

Rows one and three of Table 4.1 show that DIFSOL took 18 percent fewer function evaluations and 50 percent fewer Jacobian evaluations than DIFSUB. In spite of this the total time taken by DIFSOL is slightly more than the total time taken by DIFSUB. But this would not be the case for large n . A k -step one-leg formula takes $2kn$ more operations than a k -step backward differentiation formula for each corrector iteration performed. This occurs because our choice of the collocation point requires two summations as shown in Equation 3.8. However, $O(n^2)$ operations are required to solve the system of linear equations given by Equation 3.9. For large n , $n^2 \gg 2kn$ ($k \leq 6$), and the extra $2kn$ operations would take an insignificant fraction of the total time. Thus, the time saved in the function and Jacobian evaluations would make DIFSOL faster than DIFSUB for higher order systems.

Since the exact solution for the van der Pol Oscillator is not available, it is impossible to assess the accuracy of the error estimates produced by the codes. They do indicate, however, that the errors generated in the case of DIFSOL are nearly the same as those

generated by DIFSUB.

Table 4.1. Statistics for Van der Pol Oscillator

METHOD	EPS	FCN CALLS	JACOB CALLS	TOTAL TIME	MAX REL ERR
DIFSUB	10^{-5}	1432	84	615	1.22×10^{-4}
OL	10^{-5}	1442	56	670	1.9×10^{-4}
DIFSOL	10^{-5}	1216	40	635	1.4×10^{-4}

Table 4.2. Statistics for Krogh's Problem

METHOD	EPS	FCN CALLS	JACOB CALLS	TOTAL TIME	MAX ABS ERR
DIFSUB	10^{-6}	719	33	567	5.1×10^{-6}
OL	10^{-6}	1120	58	808	6.1×10^{-6}
DIFSOL	10^{-6}	601	24	651	6.4×10^{-6}

4.2. Krogh's Problem

Gear [3] has suggested that Krogh's problem is a good test for methods for solving stiff systems of ordinary differential equations.

Krogh's problem can be defined as follows. The equations

$$(z_i)' = -\beta_i z_i + (z_i)^2, \quad i = 1, 2, 3, 4,$$

have the solutions

$$z_i = \beta_i / (1 + c_i e^{\beta_i x}). \quad (4.1)$$

If the initial value $z_i(0) = -1$, then $c_i = -(1 + \beta_i)$. If we set $\underline{y} = U\underline{z}$, where $\underline{z} = (z_i)^T$ and U is the unitary matrix $(1/2 - \delta_{ij})$, then the differential equation for $\underline{y}$ is

$$\underline{y}' = -B\underline{y} + U\underline{w}, \quad (4.2)$$

where $B = U \text{diag} (\beta_i^2) U^*$ and $\underline{w} = (z_i^2)^T$. The eigenvalues of the Jacobian are $2z_i - \beta_i$. If $\underline{y}(0) = \underline{z}(0)$, then $\lambda_i \rightarrow -|\beta_i|$ as $x \rightarrow \infty$. The following values for β_i are suggested by Gear [3]: $\beta_1 = 1000$, $\beta_2 = 800$, $\beta_3 = -10$ and $\beta_4 = 0.001$.

Table 4.2 shows the results of solving the above system of ordinary differential equations for $x \in [0, 1000]$ using DIFSUB, DIFSOL, and OL. The columns in this table have the same meaning as those in Table 4.1 except the last column which gives the maximum absolute error in the

components of y obtained using the exact solution to this problem.

The first and the third rows of Table 4.2 show that DIFSOL took 20 percent fewer function evaluations and 30 percent fewer Jacobian evaluations than DIFSUB. The total time taken by DIFSOL is more than the total time taken by DIFSUB. This is again due to the reasons given in Example 4.1. The errors produced by DIFSUB and DIFSOL are comparable.

The difference between the total time taken by DIFSOL and the total time taken by OL shows the effectiveness of the scheme used in DIFSOL to reflect the changes in the Jacobian caused by changes in the stepsize.

4.3. Test Problems Suggested by Lindberg

4.3.1. Problem 1

$$\begin{aligned} y'_1 &= 10^4 y_1 y_3 + 10^4 y_2 y_4, & y_1(0) &= 1, \\ y'_2 &= -10^4 y_1 y_4 + 10^4 y_2 y_3, & y_2(0) &= 1, \\ y'_3 &= 1 - y_3, & y_3(0) &= -1, \\ y'_4 &= -y_4 - 0.5 y_3 + 0.5, & y_4(0) &= 0, \\ x &\in [0, 10]. \end{aligned}$$

The exact solution of this problem can be characterized as follows.

Clearly

$$y_3(x) = 1 - e^{-2x}, \quad y_4(x) = x e^{-2x}.$$

If we set $Y(x) = [y_1(x), y_2(x)]^T$, then

$$Y' = A(x)Y, \quad Y(0) = [1, 1]^T$$

where

$$A(x) = 10^4 \begin{bmatrix} 1-2e^{-x} & xe^{-x} \\ -xe^{-x} & 1-2e^{-x} \end{bmatrix}.$$

The eigenvalues of $A(x)$ are

$$\lambda_{1,2}(x) = 10^4 [1-2e^{-x} \pm ixe^{-x}].$$

$\lambda_1(x)$ is plotted in Figure 4.1. Table 4.3 gives the exact solution $Y(x)$ and the numerical solution $\tilde{Y}(x)$ using DIFSUB and DIFSOL. These results were obtained using an error tolerance of 10^{-6} .

The Euclidean norm of $\tilde{Y}(x)$ obtained using DIFSUB shows that DIFSUB failed to detect the presence of large positive eigenvalues and produced a decaying solution. The last column of Table 4.3 shows that DIFSOL was able to detect the presence of large positive eigenvalues and indicated the unstable nature of the problem. The norm of $\tilde{Y}(x)$ produced by DIFSOL is not very close to the norm of $Y(x)$ shown in the second column of Table 4.3. The norm of $Y(x)$ starts growing at $x = 0.7$, whereas, the norm of $\tilde{Y}(x)$ starts increasing sooner. This is due to the fact that the region of instability for a k -step one-leg formula for $k > 2$ contains a small portion of the left half plane. Thus, if the eigenvalues of a system have large imaginary components, the solution could grow even when the eigenvalues have small negative real components. This "bulge" in the region of instability is shown in Figure 2.2. Also, our error tolerance per step is 10^{-6} so that DIFSOL does not trace the solution

Table 4.3. Numerical Results for Problem 1

x	Exact	Using DIFSUB		Using DIFSOL	
	$\ y(x)\ _2$	h	$\ \tilde{y}(x)\ _2$	h	$\ \tilde{y}(x)\ _2$
0.00	1.4	1.0×10^{-6}	1.4	1.0×10^{-6}	1.4
0.10	7.5×10^{-393}	1.5×10^{-2}	5×10^{-10}	1.6×10^{-2}	4.7×10^{-9}
0.30	6.5×10^{-949}	3.2×10^{-2}	3×10^{-9}	3.8×10^{-2}	4.1×10^{-8}
0.70	4.1×10^{-1333}	3.2×10^{-2}	1×10^{-9}	1.2×10^{-2}	8.3×10^{-7}
1.00	3.7×10^{-1148}	3.2×10^{-2}	2×10^{-9}	1.2×10^{-2}	9.8×10^{-7}
1.10	7.0×10^{-1018}	3.2×10^{-2}	6.8×10^{-11}	4.2×10^{-3}	1.1×10^{-6}
1.11	2.7×10^{-1003}	3.2×10^{-2}	6.8×10^{-11}	7×10^{-5}	1.0×10^{-4}
1.12	2.4×10^{-988}	3.2×10^{-2}	6.8×10^{-11}	3×10^{-5}	1.1×10^{11}
1.13	4.1×10^{-973}	3.2×10^{-2}	6.8×10^{-11}	3×10^{-5}	1.0×10^{35}
1.14	1.3×10^{-958}	3.2×10^{-2}	6.8×10^{-11}	3×10^{-5}	6.6×10^{41}
1.15	8.2×10^{-942}	3.2×10^{-2}	6.8×10^{-11}	3×10^{-5}	4.0×10^{58}
1.40	2.0×10^{-464}	8.0×10^{-2}	1.0×10^{-9}	overflow	
1.60	4.2×10^{-16}	8.0×10^{-2}	1.0×10^{-10}		
4.0	1.3×10^{8845}	1.0×10^{-1}	1.4×10^{-10}		
10.00	1.3×10^{34744}	2.0×10^{-1}	2×10^{-13}		

much below this value, and when the eigenvalues start increasing the norm of $\tilde{Y}(x)$ becomes much bigger than the norm of $Y(x)$. In spite of this difficulty, DIFSOL can detect the presence of large positive eigenvalues even when DIFSUB fails to do so.

4.3.2. Problem 2

$$\begin{aligned}
 y'_1 &= -10^4 y_1 y_3 + 10^4 y_2 y_6, & y_1(0) &= 1, \\
 y'_2 &= -10^4 y_1 y_6 - 10^4 y_2 y_3, & y_2(0) &= 1, \\
 y'_3 &= -y_3 - y_4 + 1, & y_3(0) &= 0, \\
 y'_4 &= -2y_4, & y_4(0) &= 8, \\
 y'_5 &= 1 - y_5, & y_5(0) &= -1, \\
 y'_6 &= -y_6 - 0.5y_5 + 0.5, & y_6(0) &= 0, \\
 x &\in [0, 10].
 \end{aligned}$$

The exact solution of this problem can be characterized as follows. Clearly

$$\begin{aligned}
 y_4(x) &= 8e^{-2x}, \quad y_3(x) = 1 + 8e^{-2x} - 8e^{-x}, \\
 y_5(x) &= 1 - 2e^{-x}, \quad y_6(x) = xe^{-x}.
 \end{aligned}$$

If we set $Y(x) = [y_1(x), y_2(x)]^T$, then

$$Y' = A(x)Y, \quad Y(0) = [1, 1]^T$$

where

$$A(x) = -10^4 \begin{vmatrix} 1+8e^{-2x}-8e^{-x} & -8e^{-x} \\ xe^{-x} & 1+8e^{-2x}-8e^{-x} \end{vmatrix}.$$

The eigenvalues of $A(x)$ are

$$\lambda_{1,2}(x) = -10^4 [1 + Be^{-2x} - Be^{-x} \pm ixe^{-x}].$$

In Figure 4.2 $\lambda_2(x)$ is plotted for $B=5$.

B has been chosen such that the maximum of the Euclidean norm of $Y(x)$ is very large. For $B = 4.912$ the system does not have any large components, and $\max ||Y(x)||$ is about 20. For even smaller values of B , $A(x)$ does not have any positive eigenvalues.

DIFSOL failed to detect the presence of large positive eigenvalues for $\text{eps} = 10^{-6}$. This is due to the fact that the amplification factor is essentially 1 in most of the region of instability for a one-leg formula. Thus, if the eigenvalues of the system lie in this portion of the region of instability, the growth of the solution is nearly imperceptible. However, for $\text{eps} = 10^{-8}$, DIFSOL managed to detect the change in eigenvalues from large negative values to large positive values. Table 4.4 presents the results produced by DIFSUB and DIFSOL for $\text{eps} = 10^{-8}$. The norm of $\tilde{Y}(x)$ produced by DIFSOL is not very close to the norm of $Y(x)$. This is again due to the reasons given in Example 4.3. DIFSUB could not detect the presence of large positive eigenvalues even for this stricter error tolerance. This can be explained by the fact that backward differentiation formulas are stable in most of the right half plane.

Figure 4.1. $\lambda_1(x)$ for Problem 1 Figure 4.2. $\lambda_2(x)$ for Problem 2

Table 4.4. Numerical Results for Problem 2

x	Exact	Using DIFSUB		Using DIFSOL	
	$\ y(x)\ _2$	h	$\ y(x)\ _2$	h	$\ y(x)\ _2$
0	1.4	1.0×10^{-8}	1.4	1.0×10^{-8}	1.4
0.1	1.5×10^{-336}	1.9×10^{-2}	1.2×10^{-15}	1.0×10^{-2}	3.0×10^{-10}
0.3	4.1×10^{-574}	1.8×10^{-2}	7.6×10^{-11}	1.0×10^{-2}	4.2×10^{-10}
0.4	1.4×10^{-558}	1.8×10^{-2}	1.8×10^{-10}	1.0×10^{-2}	6.0×10^{-10}
0.5	3.9×10^{-491}	1.8×10^{-2}	2.6×10^{-10}	1.0×10^{-2}	7.3×10^{-10}
0.6	3.9×10^{-396}	1.8×10^{-2}	2.14×10^{-10}	8.3×10^{-3}	3.16×10^{-9}
0.65	3.1×10^{-343}	2.2×10^{-2}	2.1×10^{-11}	4.2×10^{-5}	1.94×10^{-4}
0.7	4.3×10^{-289}	2.7×10^{-2}	1.0×10^{-12}	2.0×10^{-5}	2.5×10^{50}
0.72	2.1×10^{-267}	2.7×10^{-2}	1.1×10^{-13}	2.0×10^{-5}	1.2×10^{72}
1.00	3.6×10^{-5}	3.2×10^{-2}	5.2×10^{-11}	overflow	
1.4	6.9×10^{82}	3.6×10^{-2}	3.3×10^{-11}		
1.6	1.8×10^{-33}	3.6×10^{-2}	2.6×10^{-11}		
4.0	4.5×10^{-6909}	9.2×10^{-2}	1.8×10^{-12}		
10.00	1.2×10^{-32572}	2.1×10^{-1}	1.0×10^{-10}		

LIST OF REFERENCES

- [1] Enright, W. H., "Improving the efficiency of matrix operations in the numerical solution of stiff ordinary differential equations," ACM Trans. Math. Software 4, 2, June 1978, 127-136.
- [2] Lindberg, B., "On a dangerous property of methods for stiff differential equations," BIT 14, 1974, 430-436.
- [3] Gear, C. W., Numerical Initial Value Problems in Ordinary Differential Equations, Prentice-Hall, Englewood Cliffs, NJ 1971.

DIFSOL

c this subroutine integrates a set of n ordinary differential first
 c order equations over one step of length h at each call. h can be
 c specified by the user for each step, but it may be increased or
 c decreased by difsub within the range hmin to hmax in order to
 c achieve as large a step as possible while not committing a single
 c step error which is larger than eps in the 1-2 norm, where each
 c component of the error is divided by the components of ymax.

c
 c the program requires six subroutines named
 c diffun(t,y,dy)
 c decomp(n,pw,jperm)
 c dech(n,pw,lperm)
 c comph(n,pw,lperm)
 c solve(n,pw,jperm,lperm)
 c pederv(t,y,pw,m)

c the first, diffun, evaluates the derivatives of the dependent
 c variables stored in y(1,i) for i = 1 to n, and stores the
 c derivatives in the array dy. the next four are called only if the
 c method flag mf is set to 1 or 2 for stiff methods.
 c the matrix in both should be single precision,
 c while all floating point vectors should be double precision.
 c pederv is used only if mf is 1, and computes the partial
 c derivatives of the differential equations as described under the
 c mf parameter.

c
 c the program uses double precision arithmetic for all floating
 c point variables except those starting with p. the former are
 c single precision to save time and space.

c
 c the temporary storage space is provided by the caller in the
 c integer array ip, the single precision array pw, and the double
 c precision arrays save and csave. the array pw is used only to hold
 c the matrix of the same name, and save is used to save the values
 c of y in case a step has to be repeated, but csave is used to hold
 c several arrays. the regions used are

c csave(1,1) is used mainly to hold the correction terms in the
 c corrector loop, and holds the derivatives during
 c jacobian evaluations.

c csave(1,2) is used to save the values of the sums of all of the
 c correction terms in the previous step after they
 c have been accumulated in the array error in the
 c current step. this enables the backwards difference
 c of error to be formed. it is used to estimate the
 c step size for one order higher than current.


```

c         the differential equations with respect
c         to the y's.  this is done by call
c         pederv(t,y,pw,n). pw is an n by n array
c         which must be set to the partial of
c         the i-th equation with respect
c         to the j dependent variable in pw(i,j).
c         2 the same as case 1, except that this
c           subroutine computes the partial
c           derivatives by numerical differencing
c           of the derivatives. hence pederv is
c           not called.
c
c   ymax  an array of n locations which contains the maximum
c         of each y seen so far. it should normally be set to
c         1 in each component before the first entry. (see the
c         description of eps.)
c
c   error  an array of n elements which contains the estimated
c         one step error in each component.
c
c   kflag  a completion code with the following meanings..
c         +1 the step was successful.
c         -1 the step was taken with h = hmin, but the
c           requested error was not achieved.
c         -3 corrector convergence could not be
c           achieved for h .gt. hmin.
c         -4 the requested error is smaller than can
c           be handled for this problem.
c
c   jstart an input indicator with the following meanings..
c         -1 repeat the last step with a new h
c         0  perform the first step. the first step
c           must be done with this value of jstart
c           so that the subroutine can initialize
c           itself.
c         +1 take a new step continuing from the last.
c           jstart is set to nq, the current order of the method
c           at exit. nq is also the order of the maximum
c           derivative available.
c
c   maxder the maximum derivative that should be used in the
c         method. since the order is equal to the highest
c         derivative used, this restricts the order. it must
c         be less than 13 for adams and 7 for stiff methods.
c
c   pw     a block of at least n*2 floating point locations.
c
c
c   subroutine difsub(n,t,y,save,csave,h,hmin,hmax,eps,mf,ymax,
c   1 error,kflag,jstart,maxder,pw,ip1,ip2)
c   integer failur,i,iret,idoub,iweval,j,j1,j2,jstart
c   integer k,kflag,l,mf,maxder,ntyp,n,nq,nqold,nqnew
c   double precision a(13),bnd,br,csave(n,3),d,del,dell,cr
c   double precision e,edwn,enq1,enq2,enq3,eps,error(n),eup,f
c   double precision g,h,hmax,hmin,hnew,hold,r,rl,save(13,n),t,told

```

```

double precision y(13,n),ymax(n),qls,theta,tcu,qsave,savqls(6)
real paps,pr1,pr2,pr3
dimension ipl(n),ip2(n),pertst(12,2,3),pw(n,n)
common /stat/ ifun,ijac
c
c the coefficients in pertst are used in selecting the step and
c order, therefore only about one percent accuracy is needed.
c
  data pertst / 20.0,11.66,11.06,12.72,15.18,18.11,
1              +1.0,+1.0,+1.0,+1.0,+1.0,+1.0,
2              2.0,12.0,24.0,37.89,53.33,70.08,
3              87.97,100.9,126.7,147.3,168.8,191.4,
4              9.66,8.93,10.66,13.22,16.27,+1.0,
5              +1.0,+1.0,+1.0,+1.0,+1.0,+1.0,
6              12.0,24.0,37.89,53.33,70.08,87.97,
7              100.9,126.7,147.3,168.9,191.4,+211.0,
8              +1.0,10.0,1.61,.3082,.05943,.01029,
9              +1.0,+1.0,+1.0,+1.0,+1.0,+1.0,
a              +1.0,1.0,2.0,1.0,.3157,.07407,
b              .0139,.0021818,.00029,.000035,.0000037,.00000035/
c
  data savqls /0.550000,0.5,0.472125053137566139,
1              0.444115702508635415,
2              0.4195125902341039625,0.39846338506548709108 /
c
  kflag = 0
  failur = 0
c
  case
    if (jstart.eq.-1) go to 120
c
  1: jstart = +1 or 0
    if (jstart.eq.1) go to 110
c
c on the first call, the order is set to 1 and the initial
c derivatives are calculated.
c
    cr = 1.0
    nq = 1
    call diffun(t,y,csave(1,3))
    do 101 i = 1,n
101      y(2,i) = csave(i,3) * h
    hnew = h
    k = 2
c
c begin by saving information for possible restarts and changing
c h by the factor r if the caller has changed h. all variables
c dependent on h must also be changed.
c e is a comparison for errors of the current order nq. eup is
c to test for increasing the order, edwn. for decreasing the order.

```

```

c  hnew is the step size that was used on the last call.
c
110  do 111 i = 1,n
      do 111 j = 1,k
111      save(j,i) = y(j,i)
      hold = hnew
      if (h.eq.hold) go to 130
c      "change stepsize"
      iret = 1
      go to 500
c
c  repeat last step by restoring saved information.
c
c      2: jstart = -1
120  if (nq.eq.nqold) jstart = 1
      t = told
      nq = nqold
      k = nq + 1
c      "change stepsize"
      iret = 1
      go to 500
130  nqold = nq
      told = t
      if (jstart.gt.0) go to 200
c      "change order"
      iret = 1
      go to 600
c
c  this section computes the predicted values by effectively
c  multiplying the saved information by the pascal triangle
c  matrix.
c
c      repeat until step successful
c      repeat until corrector converges
c
200      t = t + h
      do 201 j = 2,k
        do 201 j1 = j,k
          j2 = k - j1 + j - 1
          do 201 i = 1,n
201          y(j2,i) = y(j2,i) + y(j2+1,i)
c
c  up to 3 corrector iterations are taken.  convergence is tested by
c  requiring the 12 norm of changes to be less than bnd which is
c  dependent on the error test constant.
c  the sum of the corrections is accumulated in the array
c  error(i).  it is equal to the k-th derivative of y multiplied
c  by  $h^k / (factorial(k-1) \cdot (-a(k)))$ , and is therefore proportional

```

```

c   to the actual errors to the lowest power of h present. (h k)
c
c       k1=k-1
c       k2=k-2
c
c       do 202 i = 1,n
202         error(i) = 0.0
c
c       tcur=t-theta*h
c
c       do 242 l = 1,3
c
c   next two loops calculate y and y' at the collocation point
c
c       do 205 iqj=1,n
c         y(8,iqj) = y(k,iqj)
c         y(9,iqj) = y(k,iqj)*dfloat(k1)
c
c       do 203 iq2 = 1 , k1
c         y(8,iqj)=y(8,iqj)*(-theta)+y(k-iq2,iqj)
203       continue
c
c       if (k.lt.3) go to 205
c       do 204 iq2 = 1 , k2
c         y(9,iqj)=y(9,iqj)*(-theta)+
1         dfloat(k1-iq2)*y(k-iq2,iqj)
204       continue
c
c       205       continue
c
c       call diffun(tcur,y(8,1),csave(1,3))
c
c   if there has been a change of order or there has been trouble
c   with convergence, pw is re-evaluated prior to starting the
c   corrector iteration in the case of stiff methods. iweval is
c   then set to -1 as an indicator that it has been done.
c
c       if (iweval.lt.1) go to 230
c       case
c         if (mf.eq.2) go to 210
c       1: mf = 1
c         call pederv(t,y,pw,n)
c         go to 220
c
c   evaluate the jacobian into pw by numerical differencing. r is the
c   change made to the element of y. it is eps relative to y with
c   a minimum of eps**2. f stores the unchanged value of y.
c

```

```

c          2: mf = 2
210      do 212 j = 1,n
          f = y(8,j)
          r = eps**1.0 * dmax1(eps,dabs(f))
          y(8,j) = y(8,j) + r
          d = -qls * h / r
          call diffun(tcur,y(8,1),csave(1,1))
          do 211 i = 1,n
211              pw(1,j) = (csave(1,1) - csave(1,3)) / r
212          y(8,j) = f
          ifac=ifac+1

c
c  decompose the jacobian pw = L * H * L** (-1)
c  and add 1/(qls*h) * I to H
c
220      continue
          call decomp(n,pw,ipl)
          qsave = - qls * h
          do 221 i = 1,n
221          pw(1,i) = 1.0 / qsave + pw(1,i)
          ifeval = -1
c          .....exit from do loop
          call dech (n,pw,ip2)
230      do 231 i = 1,n
231          csave(1,1) = (csave(1,3) - y(9,1)/h)/(-1.0*qls)
          if (mf.eq.0) go to 235
          call solve(n,pw,csave(1,1),ipl,ip2)

c
c  correct and compare del, the l2 norm of change/ymax, with bnd.
c  estimate the value of the l2 norm of the next correction by
c  cr 2 del and compare with bnd.  if either is less, the corrector
c  is said to have converged.
c
235      del = 0.d0
          do 241 i = 1,n
              do 240 iq2=1,k
                  y(iq2,i) = y(iq2,i) + a(iq2) * csave(1,1)
240          continue
              error(1) = error(1) + csave(1,1)
              del = del + (csave(1,1) / ymax(1)) ** 2
241          continue
              if (1.ge.2) cr = dmax1(.9 * cr, del / dell)
              dell = del
c          .....exit from inner repeat block
          if (dmin1(del,cr*del).le.bnd) go to 300
242      continue

c
c  the corrector iteration failed to converge in 3 tries.  various

```

```

c possibilities are checked for.  if h is already hmin and
c this is either adams method or the stiff method in which the
c matrix pw has already been re-evaluated, a no convergence exit
c is taken. otherwise the matrix pw is re-evaluated and/or the
c step is reduced to try and get convergence.
c
250      t = told
        if (dabs(h).gt.hmin * 1.00001d0.or.(iweval - mtyp).ge.-1)
1          go to 260
            kflag = -3
            nq = nqold
c .....exit
c      "restore and return"
            go to 800
260      if ((mf.eq.0).or.(iweval.ne.0)) h = h * 0.25d0
        cr=1.0
        iweval = mf
c      "change stepsize"
        iret = 2
        go to 500
c
c if the step is o.k. it is accepted.  if idoub has been reduced
c to one, a test is made to see if the step can be increased
c at the current order or by going to one higher or one lower.
c such a change is only made if the step can be increased by at
c least 1.1.  if no change is possible idoub is set to 8 to
c prevent further testing for 8 steps.
c if a change is possible, it is made and idoub is set to
c nq + 1 to prevent further testing for that number of steps.
c if the error was too large, the optimum step size for this or
c lower order is computed, and the step retried.  if it should
c fail twice more it is an indication that the derivatives that
c have accumulated in the y array have errors of the wrong order
c so the first derivatives are recomputed and the order is set
c to 1.
c
300      d = 0.0
        do 301 i = 1,n
301      d = d + (error(i) / ymax(i)) ** 2
        iweval = 0
c      ...exit from outer repeat block
        if (d.le.e) go to 400
c
c reduce the failure flag count to check for multiple failures.
c restore t to its original value and try again unless there have
c three failures.  in that case the derivatives are assumed to have
c accumulated errors so a restart from the current values of y is
c tried.  this is continued until success or h = hmin.

```



```

c      failur = failur + 1
      if (dabs(h).gt.(hmin * 1.00001d0)) go to 310
      kflag = -1
      hnew = h
      jstart = nq
c      .....exit
      return
310    t = told
c      case
      if (failur.lt.3) go to 320
      if (nq.gt.1) go to 350
      if (dabs(h).gt.2.d0 * hmin) go to 320
c      1: failur = 3 and nq = 1 and !h! <= 2 * hmin
      kflag = -4
c      .....exit
c      "restore and return"
      go to 800
c      2: failur < 3 or (nq = 1 and !h! > 2 * hmin)
c      "determine order and stepsize"
320      go to 700
330      k = nqnew + 1
      h = h * r
c      "change stepsize"
      iret = 3
      go to 500
340      if (nqnew.eq.nq) go to 200
      nq = nqnew
c      "change order"
      iret = 1
      go to 600
c      3: failur = 3 and nq > 1
350      r = h / hold
      do 351 i = 1, n
          y(1,i) = save(1,i)
351      y(2,i) = save(2,i) * r
      nq = 1
      failur = 0
c      "change order"
      iret = 1
      go to 600
400 continue
410 kflag = +1
      hnew = h
c      case
      if (idoub.gt.1) go to 450
c      1: idoub <= 1
c      "determine order and stepsize"

```

```

      go to 700
420  if (r.lt.(1.1)) go to 460
      if (nqnew.le.nq) go to 430
c
c  compute one additional scaled derivative if order is increased.
c
      do 421 i = 1,n
421  y(nqnew+1,i) = error(i) * a(k) / dfloat(k)
430  k = nqnew + 1
      r = dmin1(r,hmax / dabs(h))
      h = h * r
      hnew = h
      if (nq.eq.nqnew) go to 440
      nq = nqnew
      "change order"
      irat = 2
      go to 600
440  r1 = 1.0
      do 441 j = 2,k
        r1 = r1 * r
      do 441 i = 1,n
441  y(j,i) = y(j,i) * r1
c
      call comph (n,pw,ip2)
      do 445 iq2 = 1 , n
        pw(iq2,iq2) = pw(iq2,iq2)-1.0/qsave-1.0/(qls * h)
445  continue
      qsave = - qls * h
      call dech (n,pw,ip2)
c
      idoub = k
      go to 460
c  2: idoub > 1
450  idoub = idoub - 1
      if (idoub.gt.1) go to 460
      do 451 i = 1,n
451  csave(i,2) = error(i)
460  do 461 i = 1,n
461  ymax(i) = dmax1(ymax(i), dabs(y(1,i)))
      jstart = nq
c  ...exit
      return
c
c  this section scales all variables connected with h and returns
c  to the entering section.
c
c  where "change stepsize"
500 h = dmax1(hmin, dmin1(h,hmax))

```

```

      if (nq.ne.nqnew) go to 510
c
c      dont update jacobian since it will be recomputed soon
c
c
      call compb (n,pw,ip2)
      do 505 iq2 = 1, n
        pw(iq2,iq2) = pw(iq2,iq2)-1.0/qsave-1.0/(savqls(nq) * h)
505 continue
      qsave = - savqls(nq) * h
      call dech (n,pw,ip2)
c
510 continue
      rl = 1.0
      do 501 j = 2,k
        r = h / hold
        rl = rl * r
        do 501 i = 1,n
501          y(j,i) = save(j,i) * rl
        do 502 i = 1,n
502          y(1,i) = save(1,i)
        idoub = k
        go to ( 130, 200, 340 ), ired
c
c      set the coefficients that determine the order and the method
c      type. check for excessive order. the last two statements of
c      this section set lweval .gt.0 if pw is to be re-evaluated
c      because of the order change, and then repeat the integration
c      step if it has not yet been done (ired = 1) or skip to a final
c      scaling before exit if it has been completed (ired = 2).
c
c      where "change order"
600 if (mf.eq.0) go to 602
      nq=min0(nq,6)
      go to ( 621,622,623,624,625,626), nq
602 nq = min0(nq,12)
      go to (605,606,607,608,609,610,611,612,613,614,615,616), nq
c
c      the following coefficients should be defined to the maximum
c      accuracy permitted by the machine. they are, in the order used..
c
c      1
c      1/2, 1/2
c      5/12, 3/4, 1/6
c      3/8, 11/12, 1/3, 1/24
c      251/720, 25/24, 35/72, 5/48, 1/120
c      95/288, 137/120, 5/8, 17/96, 1/40, 1/720
c      19087/60480, 49/40, 203/270, 49/192, 7/144, 7/1440, 1/5040

```

```

c 5257/17280, 363/280, 469/540, 967/2880, 7/90, 23/2160, 1/1260,
c 1/40320
c 1070017/3628800, 761/560, 29531/30240, 267/640, 1069/9600, 3/160,
c 13/6720, 1/8960, 1/362880
c 4165506/14515200, 7129/5040, 6515/6048, 4523/9072, 19/128,
c 3013/1036800, 5/1344, 29/96768, 1/72576, 1/3628800
c 134211265/479001600, 7381/5040, 177133/151200, 84095/145152,
c 341693/1814400, 8591/207360, 7513/1209600, 121/193536,
c 11/272160, 11/7257600, 1/39916800
c 262747265/958003200, 83711/55440, 190553/151200, 341747/518400,
c 139381/604800, 2667907/47900160, 1903/201600, 10831/9676800,
c 11/120960, 1/207369, 1/6652800, 1/479001600
c

```

```

605 a(1) = 1.0
    go to 630
606 a(1) = .500000000000d0
    a(3) = 0.5000000000d0
    go to 630
607 a(1) = 0.4166666666666667d0
    a(3) = 0.7500000000d0
    a(4) = 0.1666666666666667d0
    go to 630
608 a(1) = 0.37500000000000d0
    a(3) = 0.9166666666666667d0
    a(4) = 0.3333333333333333d0
    a(5) = 0.0416666666666667d0
    go to 630
609 a(1) = 0.3486111111111111d0
    a(3) = 1.0416666666666667d0
    a(4) = 0.4861111111111111d0
    a(5) = 0.1041666666666667d0
    a(6) = 0.008333333333333333d0
    go to 630
610 a(1) = 0.3298611111111111d0
    a(3) = 1.1416666666666667d0
    a(4) = 0.6250000000d0
    a(5) = 0.1770833333333333d0
    a(6) = 0.0250000000d0
    a(7) = 0.001388888888888889d0
    go to 630
611 a(1) = 0.3155919312169312d0
    a(3) = 1.2250000000d0
    a(4) = 0.7518518518518519d0
    a(5) = 0.2552083333333333d0
    a(6) = 0.0486111111111111d0
    a(7) = 0.004861111111111111d0
    a(8) = 0.0001984126984126984d0
    go to 630

```

```

612 a(1) = 0.3042245370370370d0
    a(3) = 1.296428571428485d0
    a(4) = 0.8685185185185185d0
    a(5) = 0.3357638888888888d0
    a(6) = 0.0777777777777777d0
    a(7) = 0.01064814814814815d0
    a(8) = 0.0007936507936507937d0
    a(9) = 0.0000248015873015873d0
    go to 630
613 a(1) = 0.2948680004409171d0
    a(3) = 1.35892857142857d0
    a(4) = 0.976554232804233d0
    a(5) = 0.41718750000d0
    a(6) = 0.1113541666666667d0
    a(7) = 0.018750000d0
    a(8) = 0.001934523809523809d0
    a(9) = 0.000111607142857143d0
    a(10) = 0.00000275573192239859d0
    go to 630
614 a(1) = 0.2869754464285714d0
    a(3) = 1.41448412698413d0
    a(4) = 1.07721560846561d0
    a(5) = 0.498567019400353d0
    a(6) = 0.14843750000d0
    a(7) = 0.0290605709876543d0
    a(8) = 0.0037202380952381d0
    a(9) = 0.000299685846560847d0
    a(10) = 0.0000137786596119929d0
    a(11) = 0.00000027557319223986d0
    go to 630
615 a(1) = 0.280189564439367d0
    a(3) = 1.46448412698413d0
    a(4) = 1.17151455026455d0
    a(5) = 0.579358190035273d0
    a(6) = 0.188322861552028d0
    a(7) = 0.0414303626543210d0
    a(8) = 0.0062111447989418d0
    a(9) = 0.000625206679894180d0
    a(10) = 0.0000404174015285126d0
    a(11) = 0.00000151565255731922d0
    a(12) = 0.0000000250521083854417d0
    go to 630
616 a(1) = 0.274265540031599d0
    a(3) = 1.50993867243867d0
    a(4) = 1.26027116402116d0
    a(5) = 0.659234182098765d0
    a(6) = 0.230458002645503d0
    a(7) = 0.0556972461052322d0

```

```

a(8) = 0.00943948412698413d0
a(9) = 0.00111927496693122d0
a(10) = 0.0000909391534391534d0
a(11) = 0.00000482253086419753d0
a(12) = 0.000000150312650312650d0
a(13) = 0.00000000208767569878681d0
go to 630
621 a(1) = 1.0000000000d0
a(2) = 1.0000000000d0
qls = 0.55000000000d0
theta = 0.45000000000d0
go to 630
622 a(1) = 0.8284271247369577d0
a(2) = 1.2426406871054366d0
a(3) = 0.4142135623684788d0
qls = 0.50000000000d0
theta = 0.29289321880d0
go to 630
623 a(1) = 0.6689987243959355d0
a(2) = 1.22649766139254839d0
a(3) = a(1)
a(4) = 0.11149978739932258d0
qls = 0.472125053137566139d0
theta = 0.17712434450000000d0
go to 630
624 a(1) = 0.560620676422353847d0
a(2) = 1.167959742546570515d0
a(3) = 0.817571819782599361d0
a(4) = 0.233591948509314103d0
a(5) = 0.023359194850931410d0
qls = 0.444115702508635415d0
theta = 0.107610285886125000d0
go to 630
625 a(1) = 0.488141744154493636d0
a(2) = 1.114590315819427136d0
a(3) = 0.915265770289675567d0
a(4) = 0.345767068776099658d0
a(5) = 0.061017718019311704d0
a(6) = 0.0040678478679541136d0
qls = 0.4195125902341039625d0
theta = 0.0649539076702870000d0
go to 630
626 a(1) = 0.43864554748227397456d0
a(2) = 1.07468159133157123770d0
a(3) = 0.98938940154335129820d0
a(4) = 0.44778399638815468236d0
a(5) = 0.10661523723527492436d0
a(6) = 0.01279382846823299092d0

```

```

a(7) = 0.00060922992705871384d0
qls = 0.39846338506548709108d0
theta= 0.03874806548429888d0
630 k = nq + 1
    idoub = k
    mtyp = (4 - mf)/2
    enq2 = .5/float(nq + 1)
    enq3 = .5/float(nq + 2)
    enq1 = 0.5/float(nq)
    peps=eps
    eup = (pertst(nq,mtyp,2)*peps**1.0)**2
    e = (pertst(nq,mtyp,1)*peps**1.0)**2
    edwn = (pertst(nq,mtyp,3)*peps**1.0)**2
    if (edwn.ne.0) go to 640
        kflag = -4
c \.....exit
c     "restore and return"
        go to 800
640 bnd = (eps**1.0 * enq3) ** 2.0
    iweval = mf
    go to ( 200, 440 ), 1ret
c
c pr1, pr2, and pr3 will contain the amounts by which the step size
c should be divided at order one lower, at this order, and at order
c one higher respectively.
c
c where "determine order and stepsize"
700 pr2 = (d / e) ** enq2 * 1.2
    pr3 = 1.e+20
    if ((nq.ge.maxder).or.(kflag.eq.0)) go to 710
702 continue
    d = 0.0
    do 701 i = 1,n
701     d = d + ((error(i) - csave(i,2)) / ymax(i)) ** 2
    pr3 = (d / eup) ** enq3 * 1.4
710 pr1 = 1.e+20
    if (nq.le.2) go to 720
    d = 0.0
    do 711 i = 1,n
711     d = d + (y(k,i) / ymax(i)) ** 2
    pr1 = (d / edwn) ** enq1 * 1.2
720 continue
c     case
        if (pr2.gt.pr3) go to 730
        if (pr2.gt.pr1) go to 740
c     1: pr2 <= pr1 and <= pr3
721     continue
        nqnew = nq

```

```

      r = 1.0 / amax1(pr2,1.e-4)
c      if ((nqnew.eq.1).and.(jstart.eq.1).and.(kflag.ne.0)) go to 731
      go to 750
730    if (pr3.ge.pr1) go to 740
731    continue
c      2: pr3 < pr2 and < pr1
      r = 1.0 / amax1(pr3, 1.e-4)
      nqnew = nq + 1
      go to 750
c      3: pr1 < pr2 and <= pr3
740    r = 1.0 / amax1(pr1, 1.e-4)
      nqnew = nq - 1
      if (nqnew.eq.1) go to 721
750    idoub = 8
      if (kflag.eq.+1) go to 420
      go to 330
c  where "restore and return"
800  do 801 i = 1,n
      do 801 j = 1,k
801    y(j,i) = save(j,i)
      h = hold
      jstart = nq
      return
      end
      subroutine decomp (neqn,a,jperm)
c
c      neqn    input    no of rows or columns in a
c      a       input    matrix to be decomposed into l**(-i)*h*1
c               output  upper trapezoid giving h
c               lower triangle giving l one column shifted
c      jperm   output  array giving row and col interchanges in h , l
c      templ   input    work array
c
      implicit real*8 (b-h,q-z)
      integer neqn,jperm(neqn)
      dimension a(neqn,neqn)
c
c      initialize jperm
c
      do 25 i = 1 , neqn
        jperm(i)=1
25    continue
c
      if (neqn.le.2) go to 250
      n1=neqn-1
c
      do 200 i = 2 , n1
c

```



```

c      find maximum element in i-1 coln starting ith row
c
      iml=i-1
      ipl=i+1
      iperm=i
      yl=a(i,i-1)
      do 40 j = ipl , neqn
        if (dabs(yl).ge.abs(a(j,i-1))) go to 40
        yl=a(j,i-1)
        iperm=j
40    continue
c
      if (yl.eq.0.0) go to 150
      if (iperm.eq.i) go to 60
c
      swap pointers in jperm
c
      jperm(i)=iperm
c
c      interchange rows and columns
c
      do 50 j = iml , neqn
        temp2=a(i,j)
        a(i,j)=a(iperm,j)
        a(iperm,j)=temp2
50    continue
c
      do 55 j = 1 , neqn
        temp2=a(j,i)
        a(j,i)=a(j,iperm)
        a(j,iperm)=temp2
55    continue
c
60    continue
c
c      perform row and col operations on a and fill l
c
      do 100 j = ipl , neqn
        temp2=a(j,i-1)/yl
        a(j,i-1)=temp2
c
        do 70 k = i , neqn
          a(j,k)=a(j,k)-temp2*a(i,k)
70      continue
c
        do 80 k = 1 , neqn
          a(k,i)=a(k,i)+temp2*a(k,j)
80      continue

```

```

c
c 100    continue
c
c        go to 200
c 150    continue
c
c        do 180 j = ipl , neqn
c            a(j,i-1)=0.0
c 180    continue
c
c 200    continue
c
c 250    continue
c        return
c        end
c        subroutine dech (neqn,a,lperm)
c
c        a        input  upper trapezoid giving h
c                   output upper triangle giving ul , lower diagonal
c                   giving li so that h=li*ul
c        lperm     output permutation vector for li , hi
c        temp1     input  temporary storage
c
c        implicit real*8 (b-h,q-z)
c        integer neqn,lperm(neqn)
c        dimension a(neqn,neqn)
c
c        initialize lperm and lp
c
c        do 50 i = 1 , neqn
c            lperm(i)=i
c 50    continue
c
c        if (neqn.eq.1) go to 250
c        reduce h to ul
c
c        do 200 i = 2 , neqn
c            iml=i-1
c            yl=a(i-1,i-1)
c        find row for pivoting
c            if (dabs(yl).ge.abs(a(i,i-1))) go to 60
c            yl=a(i,i-1)
c
c        interchange pointers in lperm
c        interchange corresponding rows
c
c            lperm(i-1)=i
c

```

```

      do 55 j = 1ml , neqn
        y2=a(1-1,j)
        a(1-1,j)=a(1,j)
        a(1,j)=y2
55      continue
c
c 60      continue
c
      y2=a(1,1ml)/y1
c      fill 11
      a(1,1ml)=y2
      do 80 k = 1 , neqn
        a(1,k)=a(1,k)-y2*a(1ml,k)
80      continue
c
c 200 continue
c
c 250 continue
      return
      end
      subroutine comph (neqn,a,lperm)
c
c      this subroutine takes a bidiagonal unit lower triangular matrix
c      and an upper triangular matrix gives an upper hassenburg matrix
c      as their product
c
c      neqn      input      dimension of a
c      a          input      upper triangle giving u
c                        lower subdiagonal giving l
c      lperm      output     upper trapezoid giving h
c      lperm      input      permutation vector
c
      implicit real*8 (b-h,q-z)
      integer neqn,lperm(neqn)
      dimension a(neqn,neqn)
c
      if (neqn.eq.1) go to 100
      nl=neqn-1
c
      do 50 i = 1 , nl
        j=nl-i+1
        jpl=j+1
        y1=a(j+1,j)
        a(j+1,j)=y1*a(j,j)
c
c      add y1*row(j) to row(j+1)
c
        do 10 k = jpl , neqn

```

```

      a(j+1,k)=a(j+1,k)+yl*a(j,k)
10    continue
c
c    apply lperm on h
c
c      if (lperm(j).eq.j) go to 50
c
c    interchan row(j) with row(j+1)
c
c      do 20 k = j , neqn
c        yl=a(j,k)
c        a(j,k)=a(j+1,k)
c        a(j+1,k)=yl
20    continue
c
50    continue
c
100  continue
c
c    return
c    end
c    subroutine solve (neqn,a,b,jperm,lperm)
c
c    implicit real*8 (b-h,q-z)
c    integer neqn,jperm(neqn),lperm(neqn)
c    dimension a(neqn,neqn),b(neqn)
c
c    if (neqn.eq.1) go to 200
c
c    solve l*x=b using forward substitution
c
c    nl=neqn-1
c    if (neqn.eq.2) go to 50
c    do 40 i = 2 , nl
c      ipl=i+1
c
c    apply jperm(i) on b
c
c      if (jperm(i).eq.1) go to 5
c      k=jperm(i)
c      yl=b(1)
c      b(i)=b(k)
c      b(k)=yl
c
c    5    continue
c
c    solve li*x=b
c

```

```

        do 10 k = ipl , neqn
            b(k)=b(k)-a(k,i-1)*b(i)
10      continue
c
c      40 continue
c
c      50 continue
c
c      solve l1**(-1)*x=b by l1*b=x
c
c      do 70 i = 1 , n1
          ipl=i+1
c
c      apply lperm(i) on b
c
          if (lperm(i).eq.i) go to 60
          k=lperm(i)
          y1=b(i)
          b(i)=b(k)
          b(k)=y1
c
c      60 continue
c
c      solve n1*x=b
c
          b(ipl)=b(ipl)-b(i)*a(ipl,i)
70      continue
c
c      solve u1*x=b by backward substitution
c
          b(neqn)=b(neqn)/a(neqn,neqn)
c
c      do 90 i = 2 , neqn
          j=neqn-i+1
          y1=0.0
          iml=i-1
c
          do 80 k = 1 , iml
              l=neqn-k+1
              y1=y1+a(j,l)*b(l)
80      continue
c
          b(j)=(b(j)-y1)/a(j,j)
90      continue
c
c      if (neqn.eq.2) go to 250
c
c      solve l**(-1)*x=b by l*b=x

```

```

c      do 120 i = 2 , n1
c          j=n1-i+2
c          jpl=j+1
c      multiply lj with b
c          do 100 l = jpl , neqn
c              b(l)=b(l)+a(l,j-1)*b(j)
100      continue
c      apply jperm(j) on b
c          if (jperm(j).eq.j) go to 120
c              k=jperm(j)
c              yl=b(j)
c              b(j)=b(k)
c              b(k)=yl
120      continue
c      go to 250
c      200 continue
c          b(1)=b(1)/a(1,1)
c      250 continue
c          return
c          end

```

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-80-1013	2.	3. Recipient's Accession No.
4. Title and Subtitle ONE-LEG FORMULAS FOR STIFF DIFFERENTIAL EQUATIONS		5. Report Date August 1980	
7. Author(s) Q. M. Sheikh		8. Performing Organization Rept. No. UIUCDCS-R-80-1013	
9. Performing Organization Name and Address Department of Computer Science University of Illinois U-C Urbana, IL 61801		10. Project/Task/Work Unit No.	
		11. Contract/Grant No. DE-AC02-76ER02383.A003	
12. Sponsoring Organization Name and Address U. S. Department of Energy Chicago Operations Office Argonne, IL		13. Type of Report & Period Covered Thesis	
		14.	
15. Supplementary Notes			
16. Abstracts There are several effective codes available for solving stiff ordinary differential equations. However, for certain problems these codes will produce reasonable looking solutions which are not close to the desired solution. This problem can arise if the formula used is stable for unstable problems and completely ignores increasing components of the solution. Unfortunately, the backward differentiation formulas which are the most widely used stiff formulas suffer from this disadvantage. Linear multistep formulas which are stable in large portions of the right half plane fail to detect the change in the eigenvalues from large negative values to large positive values unless the step size h is extremely small. This thesis presents new one-leg multistep formulas which are generalizations of the backward differentiation formulas and whose regions of instability are potentially infinitely large.			
17. Key Words and Document Analysis. 17a. Descriptors multistep one-leg stiff ordinary differential equations unstable			
17b. Identifiers/Open-Ended Terms			
17c. COSATI Field/Group			
18. Availability Statement unlimited		19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 56
		20. Security Class (This Page) UNCLASSIFIED	22. Price