EFFICIENT SOLUTION OF THE VARIATIONAL EQUATION FOR PIECEWISE-LINEAR DIFFERENTIAL EQUATIONS

by

T. S. Parker and L. O. Chua

Memorandum No. UCB/ERL M85/28

26 April 1985

(cover)

EFFICIENT SOLUTION OF THE VARIATIONAL EQUATION FOR PIECEWISE-LINEAR DIFFERENTIAL EQUATIONS

by

T. S. Parker and L. O. Chua

Memorandum No. UCB/ERL M85/28

26 April 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Efficient Solution of the Variational Equation for Piecewise-Linear Differential Equations†

Thomas S. Parker and Leon O. Chua

Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley

ABSTRACT

An efficient method for computing the solution of the variational equation associated with a piecewise-linear differential equation is presented. The method relies on the matrix exponential and requires no integration beyond the calculation of the trajectory along which the variational equation is being solved. Initial results show a ten-fold increase in efficiency over traditional methods. Some source code is included for the three-dimensional case.

1. Introduction

Due to the complexity of nonlinear phenomena, much recent work on nonlinear dynamical systems has relied on numerical simulation. In the hope of obtaining analytical results, several researchers have focused their attention on piecewise-linear systems [1, 2, 3]. However, even though analytical results are the goal, numerical simulations still play a crucial role in these piecewise-linear studies.

One widely used tool in the numerical study of differential equations is the variational equation. Most numerical techniques for finding periodic solutions, such as the shooting method [4], rely on the variational equation in the correction phase of the Newton-Raphson algorithm. Once a periodic solution has been found, its stability can be determined by calculating the eigenvalues of the solution to the variational equation. Current methods for calculating Lyapunov exponents also utilize the variational equation [5].

The solution to the variational equation is typically obtained by numerical integration. The variational equation is a matrix equation and thus computationally costly to integrate. To make matters worse, every evaluation of the right-hand side of the variational equation requires some additional integration of the original differential equation.

We will show that once the trajectory of a piecewise-linear differential equation has been calculated, the associated variational equation can be quickly and cheaply calculated with *no additional integration* (provided the solution exists).

In [1], Chua et al. study a second-order piecewise-linear differential equation with sinusoidal forcing. They derive an expression for the solution of the variational equation along a periodic solution with transverse boundary crossings. Our treatment removes these restrictions and extends the result to forced

†Research supported in part by the Hertz Foundation, by the Office of Naval Research under Contract N00014-76-C-0572 and by National Science Foundation Grant ECS8542885.

and unforced equations of any order.

2. Notation and Statement of the Theorem

Consider the piecewise-linear differential equation

$$\dot{x} = f(x) + u(t) \tag{1}$$

where $x \in \mathbb{R}^n$, $u : \mathbb{R} \to \mathbb{R}^n$, and where $f : \mathbb{R}^n \to \mathbb{R}^n$ is in piecewise-linear canonical form [6]

$$f(x) = a + Bx + \sum_{i=1}^{p} c_i \left| \langle \alpha_i, x \rangle - \beta_i \right|. \tag{2}$$

Here $B \in \mathbb{R}^{n \times n}$, a, c_i, $\alpha_i \in \mathbb{R}^n$, and $\beta_i \in \mathbb{R}$ are constants. Note that f is a sum of continuous functions and is itself continuous. Most piecewise-linear functions of interest may be represented in this canonical form [7]; however, our result relies only on the continuity of $f(\cdot)$.

Let K denote the number of linear regions defined by $f(\cdot)$. We say x *lies on a boundary* if one or more of the absolute value functions in $f(x)$ evaluates to zero. If x does not lie on a boundary, we say x *lies in the interior* of one of the K linear regions.

Suppose $\varphi(t; x_0, t_0)$ is the solution to (1) at time t with initial condition x_0 and initial time t_0. As t increases from t_0, $\varphi(t; x_0, t_0)$ repeatedly leaves one region and enters another. Thus for any time $t > t_0$, $\varphi(\cdot; x_0, t_0)$ defines a finite sequence of linear regions $R = \{r_0, r_1, \ldots r_m\}$ where $r_i \in \{1, 2, \ldots K\}$ denotes a particular region. For simplicity we assume neither x_0 nor $\varphi(t; x_0, t_0)$ lies on a boundary. Thus x_0 and $\varphi(t; x_0, t_0)$ are in r_0 and r_m, respectively. Note that it is possible for r_i to equal r_{i-1}. This situation may occur when the trajectory is tangent to a boundary.

For any time $t > t_0$, $\varphi(\cdot; x_0, t_0)$ also defines a unique, ordered set of boundary crossing times $T = \{t' : t_0 \le t' \le t, \varphi(t'; x_0; t_0)$ *lies on a boundary* $\}$. If each boundary crossing occurs at a single instant of time, then T is a finite, ordered sequence $\{t_1, t_2, \ldots t_m\}$ where t_i is the time at which the trajectory passes from region r_{i-1} to region r_i. We say such a trajectory has *finite boundary crossings*. A trajectory without finite boundary crossings must lie on a boundary for a non-zero interval of time.

The sample trajectory in Figure 1 has $R = \{3, 2, 1, 2, 1, 2, 3\}$ and $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$. It is important to remember that the sequences R and T both depend on t as well as on x_0 and t_0.

The variational equation of (1) along $\varphi(\cdot; x_0, t_0)$ is

$$\dot{\Phi} = D_x f(\varphi(t; x_0, t_0))\Phi. \tag{3}$$

where $\Phi(t) \in \mathbb{R}^{n \times n}$ and $\Phi(t_0) = I$, the identity matrix. If $\varphi(\cdot; x_0, t_0)$ has finite boundary crossings over $[t_0, t]$, $\Phi(t)$ exists and is the derivative of $\varphi(t; x_0, t_0)$ with respect to x_0 [8].

Theorem: If the solution to (1), $\varphi(\cdot; x_0, t_0)$, has finite boundary crossings over the time interval $[t_0, t]$ with sequences R and T defined above, then the solution of the variational equation (3) at time t is

$$\Phi(t) = \exp(J_{r_m}(t - t_m)) \exp(J_{r_{m-1}}(t_m - t_{m-1})) \ldots \exp(J_{r_0}(t_1 - t_0)) \tag{4}$$

where the Jacobian $J_i \in \mathbb{R}^{n \times n}$ is equal to $D_x f(x)$ in region i and $\exp(A)$ denotes the matrix exponential of A.

The proof is presented in Appendix 1.

3. Discussion

To evaluate Φ, we need the matrices J_i, $1 \le i \le K$ and the two sequences R and T. The Jacobians J_i are generally known *a priori* since the coefficients of (2) depend on them. If not, we can find them analytically by differentiating (2). R and T can be found by integrating (1) numerically and checking at each timestep whether the trajectory has entered a new region. When it does, a simple timestep-halving scheme can be used to calculate the crossing time.

The problem of calculating the matrix exponential is a tricky one. There is currently no satisfactory algorithm available for reliably calculating the exponential of an arbitrary $n \times n$ real matrix [9].§

We have included C functions in Appendix 2 for calculating the matrix exponential for the three-dimensional case. Though limited in dimension, these functions are still useful since the majority of recent research in nonlinear differential equations has concentrated on the three-dimensional case. We use La Grange interpolation (method 9 in [9]):

$$\exp(At) = \sum_{j=0}^{n-1} e^{\lambda_i t} \prod_{\substack{k=1 \\ k \neq i}}^{n} \frac{A - \lambda_k I}{\lambda_j - \lambda_k} \tag{5}$$

where the λ_k are the eigenvalues of A.

This method has several advantages for our application:

1) There is a large one-time expense to initialize the algorithm for each matrix A; however, evaluations of $\exp(At)$ for different t are very efficient.

2) Complex eigenvalues are easily handled with real arithmetic. See Appendix 2 for details.

3) The algorithm is easy to program.

The major disadvantage is that the algorithm is not reliable when the eigenvalues lie near each other. Fortunately, in most real-world applications this constraint is not a problem.†

Comparisons of the La Grange interpolation method versus the exact solution of $\exp(At)$ using 64 bit floating point arithmetic indicate accuracy to about twelve digits. No integration routine can hope to achieve this accuracy.

We also calculated the solution to the variational equation of the piecewise-linear circuit presented in [2] using the matrix exponential and by direct integration of (2)‡ using an RKF45 variable stepsize integration routine. Direct integration was roughly ten times slower than the matrix exponential method and the result was less accurate. The matrix exponential method required slightly less than twice the time it took to integrate the original differential equation. Most of the additional time was spent locating the boundary crossings—the evaluation of the matrix exponential was very quick.

---

§Evaluating matrix exponentials is currently a hot research topic for numerical analysts. It is likely that robust and efficient computation methods will become available in the near future.

†To make the routine robust, it could be altered to solve $\dot{x} = Ax$, $x(0) = I$ whenever the eigenvalues are close.

‡This is not as easy as it sounds. Each timestep of the variational equation integration requires integration of the original differential equation. This nested integration becomes even more complicated when a variable timestep integration routine is used.

Acknowledgements

The authors would like to thank Greg Bernstein for his comments and suggestions on the manuscript. T. Parker gratefully acknowledges the John and Fannie Hertz Foundation for its support.

Appendix 1: Proof of the Theorem

In this appendix we assume x_0 and t_0 are fixed and we abbreviate $\varphi(\cdot; x_0, t_0)$ by $\varphi(\cdot)$. By $O(\varepsilon)$ we mean a function such that $\|O(\varepsilon)\|/\varepsilon$ is bounded as $\varepsilon \to 0$, $\varepsilon > 0$. By $o(\varepsilon)$ we mean a function such that $\|o(\varepsilon)\|/\varepsilon \to 0$ as $\varepsilon \to 0$, $\varepsilon > 0$.

First, we present a lemma.

Lemma: Let $A \in \mathbb{R}^{n\times n}$ and $t \in \mathbb{R}$. Then $\exp(A(t + O(\varepsilon))) = \exp(At) + O(\varepsilon)$.†

Proof of Lemma: We use the Taylor series expansion for $\exp(\cdot)$:

$$\begin{aligned} \exp(A(t + O(\varepsilon))) &= \exp(At)\exp(AO(\varepsilon)) \\ &= \exp(At)(I + AO(\varepsilon) + (AO(\varepsilon))^2/2 + \ldots \\ &= \exp(At) + O(\varepsilon) \end{aligned} \tag{6}$$

which completes the proof.

Proof of the Theorem: What we want to find is $\Phi(t) = D_{x_0}\varphi(t)$. If (1) were linear with Jacobian J, $\varphi(t)$ would have the form

$$\varphi_L(t) = \exp(J(t-t_0))x_0 + \int_{t_0}^{t} \exp(J(t-\tau))u(\tau)d\tau \tag{7}$$

and $\Phi(t)$ would simply be $\Phi_L(t) = \exp(J(t-t_0))$. The subscript L denotes the linear case.

However (1) is not linear. To account for the boundary crossings, we isolate them by choosing any sequence $\{\tau_0, \tau_1, \ldots \tau_m\}$ such that $\tau_0 = t_0$, $\tau_m = t$ and $t_{i-1} < \tau_i < t_i$ for $1 \le i \le m-1$. By the chain rule,

$$D_{x_0}\varphi(\tau_m) = D_{\varphi(\tau_0)}\varphi(\tau_m) = D_{\varphi(\tau_{m-1})}\varphi(\tau_m)\, D_{\varphi(\tau_{m-2})}\varphi(\tau_{m-1}) \ldots D_{\varphi(\tau_0)}\varphi(\tau_1). \tag{8}$$

Hence the problem is reduced to calculating $\Phi_i := D_{\varphi(\tau_{i-1})}\varphi(\tau_i)$ for $1 \le i \le m$. Then $\Phi = \Phi_m \Phi_{m-1} \ldots \Phi_1$.

Each Φ_i accounts for exactly one boundary crossing. We will show that

$$\Phi_i = \exp(J_{\tau_i}(\tau_i - t_i))\exp(J_{\tau_{i-1}}(t_i - \tau_{i-1})) \quad 1 \le i \le m \tag{9}$$

which will conclude the proof. Intuitively, (9) means that the boundary crossing has no ill effect; Φ_i is simply the product of the Φ_L's in regions τ_{i-1} and τ_i.

Some sample trajectories $\varphi(\cdot)$ are shown in Figure 2 over the interval $[\tau_{i-1}, \tau_i]$. Figure 2 also shows trajectories with perturbed initial conditions $\hat{\varphi}(\cdot) := \varphi(\cdot; x_0 + \Delta x, t_0)$ over the same time period. The perturbation is slight, that is $\|\hat{\varphi}(t') - \varphi(t')\| < \varepsilon$ for $t_0 \le t' \le t$. $\varepsilon > 0$ is chosen small enough that $\varphi(\tau_i)$ and $\hat{\varphi}(\tau_i)$ lie in the same region for $0 \le i \le m$. Continuity of $\varphi(\cdot)$ with respect to

†We have abused notation here. On the left-hand side, $O(\varepsilon) \in \mathbb{R}$ while on the right-hand side $O(\varepsilon) \in \mathbb{R}^{n\times n}$. Of course the two are not equal.

the initial condition allows us to find such an ε.

Define $\Delta\varphi(\cdot) := \hat{\varphi}(\cdot) - \varphi(\cdot)$. From the definition of the derivative, Φ_i is the matrix such that

$$\Delta\varphi(\tau_i) = \Phi_i \Delta\varphi(\tau_{i-1}) + o(\varepsilon). \tag{10}$$

Let t' be the first time (after τ_{i-1}) such that either $\varphi(t')$ or $\hat{\varphi}(t')$ lies on a boundary. Similarly, let t'' be the last time (before τ_i) such that either $\varphi(t'')$ or $\hat{\varphi}(t'')$ lies on a boundary†. Finite boundary crossings and the continuity of φ with respect to initial conditions guarantee that $t_i - t' = O(\varepsilon)$ and $t'' - t_i = O(\varepsilon)$. Hence $t'' - t' = O(\varepsilon)$.

$\varphi(\cdot)$ and $\hat{\varphi}(\cdot)$ stay in region τ_{i-1} over $[\tau_{i-1}, t')$. Thus from (7)

$$\Delta\varphi(t') = \exp(J_{\tau_{i-1}}(t' - \tau_{i-1}))\Delta\varphi(\tau_{i-1}). \tag{11}$$

Since $t_i - t' = O(\varepsilon)$ we use the lemma and the fact that $O(\varepsilon)O(\varepsilon) = o(\varepsilon)$ to get

$$\Delta\varphi(t') = \exp(J_{\tau_{i-1}}(t_i - \tau_{i-1}))\Delta\varphi(\tau_{i-1}) + o(\varepsilon). \tag{12}$$

Similarly

$$\Delta\varphi(\tau_i) = \exp(J_{\tau_i}(\tau_i - t_i))\Delta\varphi(t'') + o(\varepsilon). \tag{13}$$

To get Φ_i all that remains is to take care of the boundary crossing. We need to find a relationship between $\Delta\varphi(t'')$ and $\Delta\varphi(t')$. Recall that

$$\varphi(t'') = \varphi(t') + \int_{t'}^{t''} [f(\varphi(\tau)) + u(\tau)]d\tau \tag{14}$$

is the solution to (1) with initial condition $\varphi(t')$. Using a similar expression for $\hat{\varphi}(t'')$ with initial condition $\hat{\varphi}(t')$ we get

$$\Delta\varphi(t'') = \Delta\varphi(t') + \int_{t'}^{t''} [f(\hat{\varphi}(\tau)) - f(\varphi(\tau))]d\tau. \tag{15}$$

Since $t'' - t' = O(\varepsilon)$ and f is continuous, the integral is $o(\varepsilon)$ and

$$\Delta\varphi(t'') = I\Delta\varphi(t') + o(\varepsilon). \tag{16}$$

Finally we combine (12), (13) and (16) to get

$$\Delta\varphi(\tau_i) = \exp(J_{\tau_i}(\tau_i - t_i))\, I\, \exp(J_{\tau_{i-1}}(t_i - \tau_{i-1}))\Delta\varphi(\tau_{i-1}) + o(\varepsilon) \tag{17}$$

which, when compared with (10), yields the desired result (9) and finishes the proof.

For trajectories without finite boundary crossings, $t'' - t' \neq O(\varepsilon)$ and (16) no longer holds. For a particular perturbation of x_0 (Figure 3(a)), (16) becomes

$$\Delta\varphi(t'') = \exp(J_{\tau_i}(t'' - t'))\Delta\varphi(t') + o(\varepsilon) \tag{18}$$

while for another perturbation (Figure 3(b)) it becomes

$$\Delta\varphi(t'') = \exp(J_{\tau_{i-1}}(t'' - t'))\Delta\varphi(t') + o(\varepsilon). \tag{19}$$

Thus Φ_i depends on the perturbation and Φ is not well-defined.

---

†If $\hat{\varphi}(\cdot)$ does not hit a boundary during the interval $[\tau_{i-1}, \tau_i]$ (Figure 2(c)), then $\tau_{i-1} = \tau_i$ and we have $\Phi_i = \Phi_L = \exp(J_{\tau_i}(\tau_i - \tau_{i-1}))$ which satisfies equation (9).

Appendix 2: Computer Algorithm

A computer routine for solving the variational equation using matrix exponentials has three parts:

1) Integrate the trajectory of the original differential equation.

2) Locate the boundary crossings.

3) Calculate the matrix exponential.

Typically the differential equation would be integrated until two successive points lie in different regions. Then a timestep-halving scheme would be used to accurately locate the time of the boundary crossing. Next the matrix exponential would be calculated and multiplied into the result. Since steps 1) and 2) depend upon which integration package you choose to use, we will concentrate only on the matrix exponential.

The La Grange interpolation formula (5) requires the eigenvalues of the matrix A. Since we are restricting our attention to the three-dimensional case, we use the closed-form solution to a cubic equation to find the eigenvalues.

When a complex conjugate pair of eigenvalues exists, the La Grange formula can be rewritten

$$\exp(At) = e^{\lambda_1 t}\frac{(A-\alpha I)^2+\beta^2 I}{(\lambda_1-\alpha)^2+\beta^2} + \frac{e^{\alpha t}(A-\lambda_1 I)}{(\lambda_1-\alpha)^2+\beta^2}\Big\{((2\alpha-\lambda_1)I-A)\cos(\beta t)+((\alpha-\lambda_1)/\beta)(A-\alpha I)+\beta I)\sin(\beta t)\Big\} \tag{20}$$

where λ_1 is the real eigenvalue and the complex conjugate eigenvalues are $\alpha \pm j\beta$.

The C functions to implement the La Grange Interpolation formula for the three-dimensional case are given in Figure 4. There are two main functions. The function *init_mat_exp()* performs the one-time initialization for a particular matrix. If the eigenvalues are too close† *init_mat_exp()* returns NULL; otherwise, it returns a pointer to a structure which contains all the necessary information for *mat_exp()* to compute $\exp(At)$ at different values of t.

We have also included a test routine *main()* which can be activated by #defining TEST on the compiler command line.

†There is no strict definition of *close*. We believe the tolerances used in the source code are very conservative.

References

1. L. O. Chua, M. Hasler, J. Neirynck, and P. Verburgh, "Dynamics of a Piece-Wise Linear Resonant Circuit," *IEEE Trans. Circuit Syst.*, vol. CAS-29, pp. 535-547, August 1982.
2. T. Matsumoto, L. O. Chua, and M. Komuro, "The Double Scroll," *IEEE Trans. Circuits Syst.*, vol. CAS-32, August 1985.
3. T. Matsumoto, L. O. Chua, and S. Tanaka, "Simplest Chaotic Non-Autonomous Circuit," *Physical Review A*, vol. 30, no. 2, pp. 1155-1157, August 1984.
4. L. O. Chua and P.-M. Lin, *Computer-Aided Analysis of Electronic Circuits*, pp. 687-702, Prentice-Hall Inc., Englewood Cliffs, N. J., 1975.
5. I. Shimada and T. Nagashima, "A Numerical Approach to Ergodic Problem of Dissipative Dynamical Systems," *Prog. of Theo. Phys.*, vol. 61, no. 6, pp. 1605-1616, June 1979.
6. L. O. Chua and R. L. P. Ying, "Canonical Piecewise-Linear Analysis," *IEEE Trans. Circuit Syst.*, vol. CAS-30, pp. 125-140, March 1983.
7. S. M. Kang and L. O. Chua, "A Global Representation of Multidimensional Piecewise-Linear Functions with Linear Partitions," *IEEE Trans. Circuit Syst.*, vol. CAS-25, pp. 938-940, November 1978.
8. I. N. Hajj and S. Skelboe, "Steady-State Analysis of Piecewise-Linear Dynamic Systems," *IEEE Trans. Circuit Syst.*, vol. CAS-28, pp. 234-242, March 1981.
9. C. Moler and C. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix," *SIAM Review*, vol. 20, no. 4, pp. 801-836, October 1978.

Figure Captions

Figure 1. A typical trajectory for a piecewise-linear differential equation with three linear regions.

Figure 2. Some typical trajectories φ, each making a finite boundary crossing. Also shown is a typical perturbation $\hat{\varphi}$ of each trajectory.

Figure 3. A typical trajectory φ, making a boundary crossing which is not finite. (a) and (b) show different perturbations of this trajectory.

Figure 4. C source code for the evaluation of three-dimensional matrix exponentials.

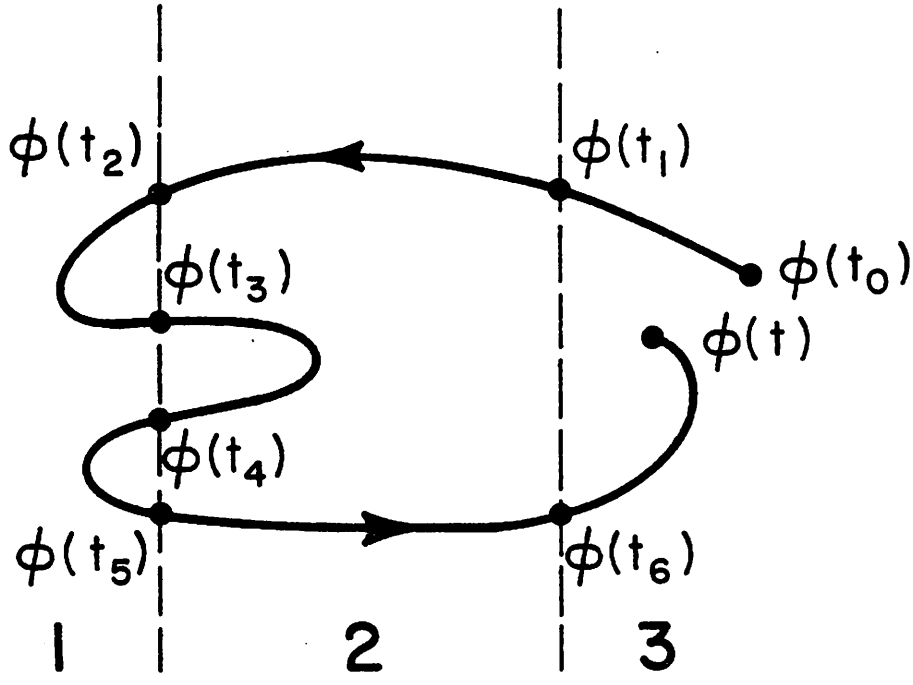


Fig. 1

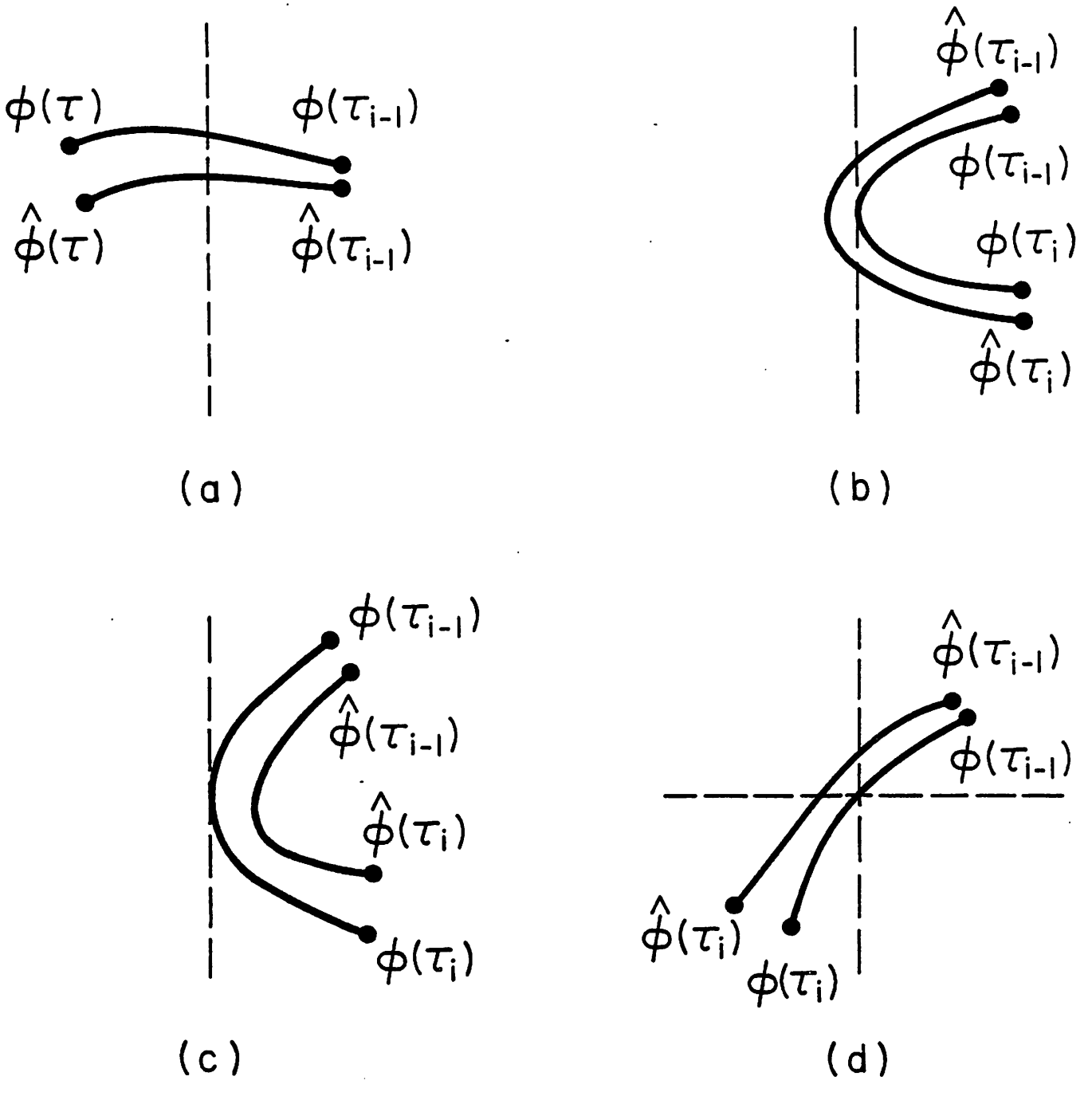
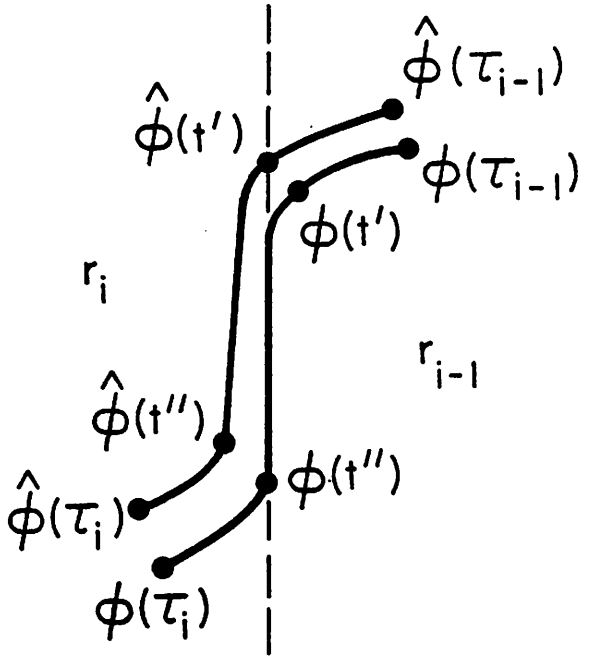
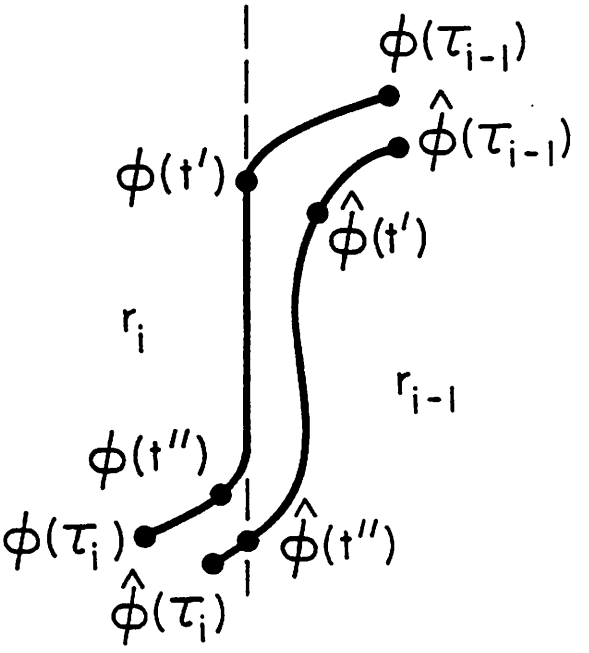

$\phi(\tau)$
$\phi(\tau_{i-1})$
$\hat{\phi}(\tau)$
$\hat{\phi}(\tau_{i-1})$
(a)
$\hat{\phi}(\tau_{i-1})$
$\phi(\tau_{i-1})$
$\phi(\tau_i)$
$\hat{\phi}(\tau_i)$
(b)
$\phi(\tau_{i-1})$
$\hat{\phi}(\tau_{i-1})$
$\hat{\phi}(\tau_i)$
$\phi(\tau_i)$
(c)
$\hat{\phi}(\tau_{i-1})$
$\phi(\tau_{i-1})$
$\hat{\phi}(\tau_i)$
$\phi(\tau_i)$
(d)

Fig. 2


$\hat{\phi}(\tau_{i-1})$
$\hat{\phi}(t')$
$\phi(\tau_{i-1})$
$\phi(t')$
r_i
r_{i-1}
$\hat{\phi}(t'')$
$\hat{\phi}(\tau_i)$
$\phi(t'')$
$\phi(\tau_i)$

(a)


$\phi(\tau_{i-1})$
$\hat{\phi}(\tau_{i-1})$
$\phi(t')$
$\hat{\phi}(t')$
r_i
r_{i-1}
$\phi(t'')$
$\phi(\tau_i)$
$\hat{\phi}(t'')$
$\hat{\phi}(\tau_i)$

(b)

Fig. 3

```
/*************************************************************************/
/*************************************************************************/
/*************************************************************************/
/*************************************************************************/

#include <stdio.h>

#define FALSE    0
#define TRUE     1

typedef struct
{
    int m_cplx;                     /*TRUE if complex eigenvalues*/
    double m_ev[3];                 /*the eigenvalues*/
    double m_a[3][3][3];            /*the three interpolation matrices*/
} MAT_EXP;

/*************************************************************************/
/*************************************************************************/

/*Evaluates the matrix exponential of the matrix associated with m (as
the result of a call to init_mat_exp()) at a given time t. The result
is returned in exp_at as a 3x3 matrix.*/

mat_exp(exp_at, t, m)                                          mat_exp

double exp_at[][3], t;
MAT_EXP *m;

{
    int i, j;
    double exp_t[3], exp(), cos(), sin();

    if (m->m_cplx)
    {
        exp_t[0] = exp(m->m_ev[0]*t);
        exp_t[1] = exp(m->m_ev[1]*t);
        exp_t[2] = exp_t[1]*sin(m->m_ev[2]*t);
        exp_t[1] *= cos(m->m_ev[2]*t);
    }
    else
        for (i = 0; i < 3; i++)
            exp_t[i] = exp(m->m_ev[i]*t);
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            exp_at[i][j] = exp_t[0]*m->m_a[0][i][j] +
                           exp_t[1]*m->m_a[1][i][j] +
                           exp_t[2]*m->m_a[2][i][j];
}

/*************************************************************************/
/*************************************************************************/

/*Initializes the 3x3 matrix a for subsequent matrix exponential
evaluation. Returns a pointer to an initialized MAT_EXP structure.
If an error occurs (no more memory, eigenvalues too close), NULL is
returned.*/

MAT_EXP *
init_mat_exp(a)                                          init_mat_exp

{
    char *malloc();
    MAT_EXP *m;
```

```
    m = (MAT_EXP *)malloc(sizeof(MAT_EXP));
    if (m == NULL)
        return (NULL);
    e_values(a, &m->m_cplx, m->m_ev);
    if (ev_close(m->m_cplx, m->m_ev))
    {
        free(m);
        return (NULL);
    }
    if (m->m_cplx)
        cplx_init(a, m);
    else
        real_init(a, m);
    return (m);
}

/*************************************************************************/
/*************************************************************************/

/*Performs initialization for the complex case*/

static
cplx_init(a, m)                                                    cplx_init

double a[];
MAT_EXP *m;

{
    double scale, b[3][3], c[3][3];

    scale = (m->m_ev[0] - m->m_ev[1])*(m->m_ev[0] - m->m_ev[1]) +
            m->m_ev[2]*m->m_ev[2];

    mat_copy(b, a);
    add_diag(b, -m->m_ev[1]);
    mat_mult(m->m_a[0], b, b);
    add_diag(m->m_a[0], m->m_ev[2]*m->m_ev[2]);
    mat_scale(m->m_a[0], 1.0/scale);

    mat_copy(b, a);
    add_diag(b, -m->m_ev[0]);
    mat_scale(b, 1.0/scale);
    mat_copy(c, a);
    add_diag(c, -2.0*m->m_ev[1] + m->m_ev[0]);
    mat_mult(m->m_a[1], b, c);
    mat_scale(m->m_a[1], -1.0);

    mat_copy(c, a);
    add_diag(c, -m->m_ev[1]);
    mat_scale(c, (m->m_ev[1] - m->m_ev[0])/m->m_ev[2]);
    add_diag(c, m->m_ev[2]);
    mat_mult(m->m_a[2], b, c);

}

/*************************************************************************/
/*************************************************************************/

/*Performs initialization for the real case.*/

static
real_init(a, m)                                                    real_init

double a[];
MAT_EXP *m;
```

```
{
    int i, j;
    double b[3][3][3];      /*b[i] is (a - ev[i])*/

    for (i = 0; i < 3; i++)
    {
        mat_copy(b[i], a);
        add_diag(b[i], -m->m_ev[i]);
    }
    mat_mult(m->m_a[0], b[1], b[2]);
    mat_mult(m->m_a[1], b[0], b[2]);
    mat_mult(m->m_a[2], b[0], b[1]);
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            if (i != j)
                mat_scale(m->m_a[i], 1.0/(m->m_ev[i] - m->m_ev[j]));
}

/*************************************************************************/
/*************************************************************************/

/*Copies 3x3 matrix a to a_copy.*/

mat_copy(a_copy, a)                                          mat_copy

double a_copy[], a[];

{
    int i;

    for (i = 0; i < 9; i++)
        a_copy[i] = a[i];
}

/*************************************************************************/
/*************************************************************************/

/*Scales each element of 3x3 matrix a by alpha.*/

mat_scale(a, alpha)                                          mat_scale

double a[], alpha;

{
    int i;

    for (i = 0; i < 9; i++)
        a[i] *= alpha;
}

/*************************************************************************/
/*************************************************************************/

/*Adds alpha to diagonal terms of a, i.e a = a + alpha*I.*/

add_diag(a, alpha)                                           add_diag

double alpha, a[][3];

{
    int i;

    for (i = 0; i < 3; i++)
        a[i][i] += alpha;
}
```

```
/*****************************************************************/
/*****************************************************************/

/*3x3 matrix multiplication: c = ab.*/

mat_mult(c, a, b)                                          mat_mult

double a[][3], b[][3], c[][3];

{
    int i, j;

    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            c[i][j] = a[i][0]*b[0][j] + a[i][1]*b[1][j] + a[i][2]*b[2][j];
}

/*****************************************************************/
/*****************************************************************/

/*Finds eigenvalues of the real 3x3 matrix a. cplx is returned TRUE if
the eigenvalues are complex; FALSE otherwise. If the eigenvalues are
real they are returned in ev[0], ev[1] and ev[2]. In the complex case,
ev[0] is the real eigenvalue, ev[1] and ev[2] are the real part and the
imaginary part of the complex eigenvalue, respectively.*/

e_values(a, cplx, ev)                                      e_values

int *cplx;
double a[][3], ev[];

{
    double p, q, r;        /*coeffs of the characteristic polynomial*/

    p = -(a[0][0] + a[1][1] + a[2][2]);
    q = a[0][0]*a[1][1] + a[0][0]*a[2][2] + a[1][1]*a[2][2] -
        a[0][1]*a[1][0] - a[1][2]*a[2][1] - a[0][2]*a[2][0];
    r = a[0][2]*a[2][0]*a[1][1] + a[2][1]*a[1][2]*a[0][0] +
        a[1][0]*a[0][1]*a[2][2] - a[0][0]*a[1][1]*a[2][2] -
        a[0][1]*a[1][2]*a[2][0] - a[0][2]*a[1][0]*a[2][1];

    cubic(p, q, r, cplx, ev);
}

/*****************************************************************/
/*****************************************************************/

/*Computes roots to cubic eqn: s^3 + p*s^2 + q*s + r = 0. The roots
are returned in x. If complex roots exist, cplx is set TRUE, x[0] is
the real root, x[1] is the real part of the complex root and x[2] > 0
is the imaginary part of the complex root; otherwise cplx is set FALSE
and x[0], x[1] and x[2] hold the three real roots.*/

static
cubic(p, q, r, cplx, x)                                    cubic

int *cplx;
double p, q, r, x[];

{
    double a, b, c, A, B, dummy, fabs(), pow(), sqrt();

    a = q/3.0 - p*p/9.0;
    b = p*p*p/27.0 - p*q/6.0 + r/2.0;
```

```
    c = a*a*a + b*b;
    if (c > 0.0)          /*complex roots exist*/
    {
        *cplx = TRUE;
        cube_root(-b + sqrt(c), 0.0, &A, &dummy);
        cube_root(-b - sqrt(c), 0.0, &B, &dummy);
        x[0] = A + B - p/3.0;
        x[1] = -(A + B)/2.0 - p/3.0;          /*real part*/
        x[2] = fabs(A - B)*sqrt(3.0)/2.0;     /*imag part*/
    }
    else          /*all real roots*/
    {
        *cplx = FALSE;
        cube_root(-b, sqrt(-c), &A, &B);
        x[0] = 2.0*A - p/3.0;
        x[1] = -A - sqrt(3.0)*B - p/3.0;
        x[2] = -A + sqrt(3.0)*B - p/3.0;
    }
}

/*************************************************************************/
/*************************************************************************/

/*Computes the cube root of xr + j*xi.  Returns the real and imaginary
parts in yr and yi, respectively.*/

static
cube_root(xr, xi, yr, yi)                                    cube_root

double xr, xi, *yr, *yi;

{
    double mag, phs, cos(), sin(), atan2(), pow(), fabs();

    if (xi == 0.0)
    {
        *yi = 0.0;
        *yr = pow(fabs(xr), 1.0/3.0);      /*pow() needs positive arg*/
        if (xr < 0.0)
            *yr *= -1.0;
    }
    else
    {
        mag = pow(xr*xr + xi*xi, 1/6.0);
        phs = atan2(xi, xr)/3.0;
        *yr = mag*cos(phs);
        *yi = mag*sin(phs);
    }
}

/*************************************************************************/
/*************************************************************************/

/*Returns TRUE if eigenvalues are too close for the algorithm to be
accurate; otherwise FALSE.  The relative and absolute tolerances are
chosen to be conservative.*/

#define REL_TOL 0.01
#define ABS_TOL 0.01

static
ev_close(cplx, ev)                                           ev_close

int cplx;
double ev[];
```

```
{
    double fabs();

    if (cplx)
    {
        if (2.0*fabs(ev[2]) < REL_TOL*fabs(ev[1]) + ABS_TOL)
            return (TRUE);
    }
    else
    {
        if (fabs(ev[0] - ev[1]) < (REL_TOL*fabs(ev[0]) + ABS_TOL) ||
                fabs(ev[0] - ev[2]) < (REL_TOL*fabs(ev[0]) + ABS_TOL) ||
                fabs(ev[1] - ev[2]) < (REL_TOL*fabs(ev[1]) + ABS_TOL))
            return (TRUE);
    }
    return (FALSE);
}

/*************************************************************************/
/*************************************************************************/

/*Writes the 3x3 matrix a to stdout.*/

mat_write(a)                                                    mat_write

double a[][3];

{
    int i, j;

    printf("\nmat_write():\n");
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 3; j++)
            printf("%15g", a[i][j]);
        printf("\n");
    }
}


/*************************************************************************/
/*************************************************************************/

/*Testing routine*/

#ifdef TEST

main()                                                              main

{
    int cplx, i, j;
    double t, a[3][3];
    MAT_EXP *m, *init_mat_exp();

    fprintf(stderr, "\nEnter matrix elements:\n");
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
        {
            fprintf(stderr, "  a[%d][%d]: ", i, j);
            scanf("%lf", &a[i][j]);
        }
    if ((m = init_mat_exp(a)) == NULL)
    {
        fprintf(stderr, "no more memory or eigenvalues too close\n");
        exit(1);
```

```
    }
    do
    {
        fprintf(stderr, "Enter t (0.0 to quit): ");
        scanf("%lf", &t);
        mat_exp(a, t, m);
        mat_write(a);
    } while (t != 0.0);
}

#endif

/*****************************************************************/
/*****************************************************************/
/*****************************************************************/
/*****************************************************************/
```