APPENDIX A

NUMERICAL METHODS

A.1 INTRODUCTION

The goal of Appendix A is to provide enough information so the reader can effectively use some subroutines (functions) that implement commonly used numerical methods. For detail about the methods readers may refer to any of a number of books on numerical analysis; for example, one "oldy but goody" is *Applied Numerical Methods*, by Carnahan et al. (1969). *Numerical Recipes: The Art of Scientific Computing*, by Press et al. (1992) with versions that emphasize either Fortran, Pascal, C or Basic, provides detail on effectively implementing these methods in computer codes. The order in which numerical methods will be described in this appendix is (1) linear algebra, (2) numerical integration, and (3) the solution of ordinary differential equations (ODEs).

If the derivative of the dependent variable y with respect to the independent variable x is only a function of the independent variable, then the solution $y = f(x)$ can be obtained by direct integration. If dy/dx depends upon both y and x, then the methods for solving ODEs must be used. Sometimes it is possible to rearrange the form of the original equation so only x appears on one side of the equal sign, and y on the other, i.e. separate variables, and then integration will provide the solution. The same principles apply for second derivatives etc. Since the methods for solving ODEs normally let $dy/dx = f(x, y)$, and this implies dy/dx may only be a function of x or y, the methods for solving ODEs can be used to solve a problem for which numerical integration could be used. However, the reverse is not true.

A.2 LINEAR ALGEBRA

A.2.1. GAUSSIAN ELIMINATION

The simplest method for solving a linear system of equations is Gaussian elimination; in this method we multiply an equation, or row in the coefficient matrix, by a value so that the first term in a resulting equation becomes zero, or is eliminated, when we subtract that equation from a given equation . This process is continued until all elements before the diagonal are zero. Then the solution vector is obtained by back substitution. (If you are unfamiliar with Gaussian elimination, you should read about it in a book on linear algebra, because the following discussion will assume you have some understanding of this method.) While it is simple and straightforward in its implementation in computer codes, Gaussian elimination can produce inaccurate solutions due to truncation error. For example, in the elimination process the product of two values, when it is subtracted from another value, can produce a difference that is several digits less accurate than can be carried in the word length being used by the computer. Therefore, especially when using single precision in a computer program, it is well to improve the accuracy by applying iterative corrections to the solution vector. The subroutine GAUSEL on the accompanying CD is a relatively simple program that uses one iterative correction to the Gaussian elimination method in solving a linear system of equations $[A]\{x\} = \{b\}$. You should obtain a listing of this code and study it as you continue reading this section. Comments in the code indicate what is done by the statements in the section which follows.

Gaussian elimination first solves the linear system $[A]\{x\} = \{b\}$ for the approximate solution $\{x_a\}$. This solution can be denoted as $[A]^{-1}\{b\}$. Next the residual vector $\{r\}$ is

computed, which is defined by $[A]\{x_a\} = \{b\} + \{r\}$. By subtracting the original matrix equation from this matrix equation, we obtain

$$[A](\{x_a\} - \{x\}) \equiv [A]\{e\} = \{r\} \tag{A.1}$$

or

$$(\{x_a\} - \{x\}) \equiv \{e\} \approx [A]^{-1}\{r\} \tag{A.2}$$

The first of these equations indicates, if $\{r\}$ is used in place of $\{b\}$, that the same solution process that has obtained $\{x_a\}$ can be used to find the error vector as $\{e\} = [A]^{-1}\{r\}$. In this case $\{e\}$ can be considered an approximate value for the error vector $\{e\} = \{x_a\} - \{x\}$. By rearrangement of the terms, with the subscript i to indicate the iteration number, we might write $\{x\}_i = \{x_a\}_i - \{e\}_i$. This iterative equation indicates for any solution component $\{x\}_i$ that an improved approximation is the original calculated value minus the calculated error value. Each subsequent error should be smaller than the current estimated error, and therefore $|\{e\}_i|$ will be a conservative estimate of the error in the new approximation. The relative error is defined as $|\{e\}_i|/|\{x_a\}_i - \{e\}_i|$. Generally one iterative improvement is all that is necessary, and that is what is done in GAUSEL.FOR.

The call to this subroutine should contain a statement of the form

```
CALL GAUSEL(N, M, A, B, DET, ERRNOR)
```
(A.3)

The arguments are the following:

N = number of equations to be solved; A must then contain N x N values, and B must contain N values.

M = the dimensions of arrays A and B in the main calling program, A(M, M) and B(M).

A = the coefficient matrix [A]. This matrix will always be a two-dimensional array that is dimensioned A(M, M).

B = the known vector {B}, which will be a one-dimensional array in the calling program that is dimensioned B(M).

DET = the determinant for the matrix [A]. Its value is returned from GAUSEL.

ERRNOR = the estimate of the relative error; it must be dimensioned in the calling program as a one-dimensional array ERRNOR(M). The values in this array are returned by GAUSEL and provide a way to decide whether the solution has sufficient accuracy.

The subroutine always returns the determinant in DET and the relative error for each component of the unknown vector in ERRNOR. If these are not wanted, then they can be eliminated from the statements in the subroutine. When GAUSEL is written to use double precision, then A, B, DET, and ERRNOR must also be double precision in the calling program.

A.2.2. USE OF THE LINEAR ALGEBRA SOLVER SOLVEQ

This section describes subroutine SOLVEQ, a more sophisticated subroutine than GAUSEL. It will (1) solve a linear system of equations, given the coefficient matrix and the known vector, (2) provide the inverse matrix of a square matrix, (3) evaluate the determinant, (4) evaluate the determinant and produce the inverse matrix, and/or (5) evaluate the determinant, produce the inverse matrix and solve the system of equations. SOLVEQ is used by a number of programs described in this text. You should extract it (the object element if you are using MS-Fortran, or the source code if you are using another compiler so you can create an object element, or the C source if you are a C user) from the CD, so it will be available to link with programs that use it.

Subroutine SOLVEQ must be called by a program that defines the problem it is to solve; the program does this by supplying the coefficient matrix in a two-dimensional

array and, if it is required, the known vector in a one-dimensional array. In Fortran the matrix and vector indexes begin with default subscript 1 and end with subscript N. Thus these arrays are dimensioned as REAL A(100, 100), B(100). A call to subroutine SOLVEQ should consist of the following (the names of the arguments can be different, but the types must be as described below, and the dimensions of arrays must be as indicated.):

```
CALL SOLVEQ(N, NPROB, NDIM, A, B, ITYPE, DET, INDX)
```
(A.4)

The arguments in the call are as follows:

N = the integer number of equations to be solved, or the size of the matrix if only the inverse is requested. The program that calls SOLVEQ must supply values for a square coefficient matrix with N rows and N columns.

NPROB = the integer number of problems to be solved by providing solution vectors, i.e., we seek NPROB separate solutions from NPROB known vectors. (A modified version of SOLVEQ may omit this argument.)

NDIM = the integer number of dimensions of matrix A(NDIM, NDIM) and vector B(NDIM). NDIM can be larger than, or equal to, N. Its value allows SOLVEQ to locate the proper positions of the elements within the two-dimensional coefficient array A.

A = the real two-dimensional array in the calling program which contains the coefficient matrix; it must be square with N rows and N columns. The correct coefficient values for the problem to be solved must all be contained within this array upon entry into subroutine SOLVEQ. Upon returning from SOLVEQ, this two-dimensional array will contain the inverse matrix, if it is requested. The values of the coefficient matrix are altered during the execution of SOLVEQ.

B = a real array containing the known vector {b} in the linear system of equations [A]{x} = {b}, and the correct values for this known vector must be in the elements of B when SOLVEQ is called. Upon returning from the call to SOLVEQ, this array will contain the solution vector {x} for the linear algebra problem. Generally B will be dimensioned as a one-dimensional array. However, if NPROB is greater than one so that more than one linear algebra problem is to be solved with the same coefficient matrix [A], then B can be a two-dimensional array with the second dimension being NPROB.

ITYPE = the integer that tells SOLVEQ which tasks are to be done, described by the value selected from the following menu of choices:

= 1, solves the linear system of equations;
= 2, produces the inverse matrix (in A);
= 3, evaluates the determinant and places the result in DET;
= 4, solves the equation set and produces the inverse matrix;
= 5, evaluates the determinant and produces the inverse matrix;
= 6, finds the determinant and the inverse matrix and solves the equations.

DET = a real variable that returns the value of determinant if it is requested.

INDX = an integer *2 one-dimensional array with the size NDIM which is used for work space. Upon entry to SOLVEQ, it can be empty or it can be another integer array used subsequently in the calling program. The values in this array will be destroyed upon returning from SOLVEQ, so if it is an INTEGER*2 array used for some other purpose, this purpose must be located after all calls to SOLVEQ have been completed.

Example programs in the body of the text can be used as examples of how to implement a call to SOLVEQ properly. If you wish to work in C (C++), then you should print the file SOLVEQC.DOC from the CD to obtain additional help in using the function solveq.

A.3 NUMERICAL INTEGRATION

A.3.1. TRAPEZOIDAL RULE

The trapezoidal rule (TR) states that an approximation to the integral of a function $f(x)$ from x_b (beginning value of the independent variable) to x_e (ending value of the independent variable) equals the average of the function, evaluated at these two end points, times the interval $\Delta x = x_e - x_b$, or

$$\Delta F = \int f(x)dx = (x_e - x_b)\{f(x_b) + f(x_e)\}/2 \qquad \text{(A.5)}$$

The accuracy of the numerical integration depends upon the size of the interval Δx. In other words, to satisfy an error requirement the Δx in the process must be chosen to be small enough. How can it be determined what is small enough? Normally Δx is small enough when the result that is obtained by using an increment $\Delta x/2$ produces the same final answer as when Δx is used. In other words, the numerical integration can be repeated after reducing Δx (usually by a factor of two); then these results are compared with those previously obtained, until the difference is less than some chosen error criterion. If this process is implemented without a consideration of how to minimize the amount of computation, much more arithmetic will be done than is required. We now describe an algorithm that allows a reduction in interval size without losing the benefit of the previous arithmetic. To facilitate this discussion a first order approximation, the trapezoidal rule (TR) will be used.

Applying the TR repeatedly over consecutive intervals Δx, in which the integration interval has been divided into N intervals $\Delta x = (x_e - x_b)/N$, produces the following result:

$$F(x_e) - F(x_b) = \Delta x\left\{f_0/2 + f_1 + f_2 + \ldots + f_{N-1} + f_N/2\right\} \qquad \text{(A.6)}$$

Thus the function values at all intermediate points are added together, except for the first and the last which are halved before being added. Here $f_0 = f_b$ (the function at the beginning of the interval) and $f_N = f_e$ (the function at the end of the interval). Now how can this extended TR be implemented repeatedly with new Δx's that are each equal to one-half the previous value without losing the previous evaluations of the function, i.e. the previous arithmetic? To visualize how such an algorithm can be developed, consider first the coarsest implementation of the TR as the average of the function at the two end points x_b and x_e, multiplied by $(x_e - x_b)$, as shown in Fig. A.1 with $N = 1$. When the range of integration is divided into two intervals with $\Delta x = (x_e - x_b)/2$, the function must be evaluated at one additional point, the midpoint shown for $N = 2$ in Fig. A.1. The application of the extended TR will multiply the previous end values by 1/2 because Δx is now half as large as before, and this result is then added to the value of the function at the midpoint, multiplied by Δx. Upon dividing the range of integration into 4 intervals so $\Delta x = (x_e - x_b)/4$, the function must be evaluated at two additional points, i.e., at $x = x_b + (\Delta x)_{i-1}/2$ and at $x = x_b + (\Delta x)_{i-1}/2 + (\Delta x)_{i-1}$, in which $(\Delta x)_{i-1}$ is the previous increment. These two additional points are shown on the line associated with $N = 3$. For $N = 4$ the function must be evaluated at four additional points, and this process continues. As shown in Fig. A.1, the sum of all of the evaluations provides all of the values that are needed to implement the extended TR. This process could continue until the evaluations of the integral between consecutive increases in N produce the same value within the error limit that has been selected.

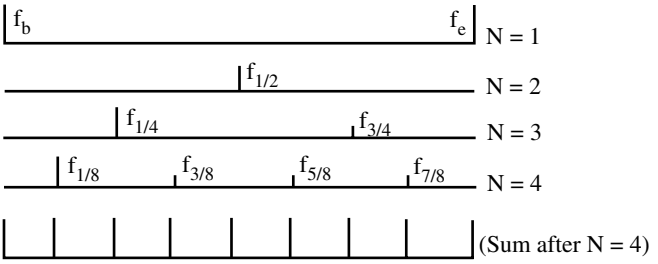


Figure A.1 Implementation of trapezoidal rule with ever decreasing increments Δx that are half of the previous increment, so that only functions at the new points are evaluated as Δx is decreased.

The Fortran listing (a C function is on the CD) in Fig. A.2 implements this algorithm; thus it is a subroutine that numerically evaluates an integral using the TR.

```
      SUBROUTINE TRAPR(EQUAT, XB, XE, VALUE, ERR, MAX)
      EXTERNAL EQUAT
      EV=-1.E30
      VALUE=0.5*(XE-XB)*(EQUAT(XB)+EQUAT(XE))
      M=1
      I=1
   10 I=I+1
      DELX=(XE-XB)/FLOAT(M)
      X=XB+0.5*DELX
      SUM=0.
      DO 20 J=1,M
      SUM=SUM+EQUAT(X)
   20 X=X+DELX
      VALUE=0.5*(VALUE+(XE-XB)*SUM/FLOAT(M))
      M=2*M
      IF(ABS(VALUE-EV).LT.ERR*ABS(EV)) RETURN
      EV=VALUE
      IF(I.LT.MAX) GO TO 10
      WRITE(*,*)' FAILED TO SATISFY ERROR REQ.', VALUE-EV
      RETURN
      END
```

Figure A.2 Program TRAPR.FOR. It integrates a function by repeatedly reducing Δx until the selected error criterion is satisfied.

To use this subroutine, two other programs are needed. The first program is a FUNCTION subroutine (with the name EQUAT as the first argument in the call to TRAPR) that evaluates the integrand (the function being integrated) at the argument X; so this program begins FUNCTION EQUAT(X). The second program is a main program that, among other tasks, properly calls TRAPR.

The arguments for TRAPR are as follows:

EQUAT = the name of the external FUNCTION SUBPROGRAM that evaluates the integrand at the given X value.

XB = the real value of the independent variable at the beginning of the interval.

XE = the real value of the independent variable at the end of the interval, i.e., the integral is from XB to XE.

VALUE = the real value of the integral that is returned to the main program.

ERR = the relative error criterion, a real number. The increment Δx will be repeatedly reduced by one half until the absolute difference between two successive values of

the integral is less than the product of ERR and the absolute value of the integral from the previous evaluation. A value for ERR of 1.0×10^{-6} is near the limit that can be used with 32 bit arithmetic and not have truncation error cause the algorithm to fail to meet the criterion.

MAX = the maximum integer number of reductions in Δx that are allowed.

If the function can be defined as a single statement in the declaration portion of the main Fortran program, then a function statement can be used in place of the FUNCTION EQUAT. This approach is used in Example Problem A.1 below.

A.3.2. SIMPSON'S RULE

Simpson's rule is a double interval integration formula; that is, it evaluates the integral over $2\Delta x$ and produces a second-order approximation by passing a second degree polynomial through three consecutive, evenly spaced points. Simpson's Rule is

$$\Delta F_{i-1}^{i+1} = \Delta x\left\{f_b + 4f_m + f_e\right\}/3 \tag{A.7}$$

in which f_m is the integrand at the midpoint of the interval $2\Delta x$, that is, at $x = \Delta x$.

As with the trapezoidal rule, Simpson's rule can be implemented in an algorithmic form that is arithmetically efficient by comparing the result with a new interval size Δx with that which was previously obtained by using $2\Delta x$. The algorithm works in the following way. We start with the entire range of the independent variable as the increment $\Delta x_0 = x_e - x_b$. An approximate (TR) value for the integral is $VALU1_0 = \Delta x_0(f_b + f_e)/2$. If we then divide this interval by 2 so $\Delta x_1 = \Delta x_0/2$ and evaluate the function at the original midpoint to obtain f_m, we can apply this rule twice and add to obtain an approximate value for the integral from x_b to x_e as $VALU1_1 = \Delta x_1(f_b + 2f_m + f_e)/2$. Simpson's rule will be obtained if we multiply this new value by 4, subtract the first value and divide the ensuing result by 3, or

$$\begin{aligned} VALUE &= (4VALU1_1 - VALU1_0)/3 \\ &= \left\{4\Delta x_1\left(f_b + 2f_m + f_e\right)/2 - \Delta x_1\left(f_b + f_e\right)\right\}/3 \\ &= \Delta x_1\left(f_b + 4f_m + f_e\right)/3 \end{aligned} \tag{A.8}$$

This algorithm can be applied with a successive halving of Δx_i for i = 2, 3, ..., and the new approximate value of the integral for Simpson's rule, *VALUE*, that is associated with each new, halved interval is

$$VALUE = (4VALU1_i - VALU1_{i-1})/3 \tag{A.9}$$

in which the *VALU*1's are obtained by the trapezoidal rule algorithm. Each $VALU1_i$ is evaluated with the new Δx, and each $VALU1_{i-1}$ is evaluated with the previous Δx.

A Fortran listing of SIMPR, which implements Simpson's rule to evaluate integrals numerically, appears in Fig. A.3. (A similar C function is on the CD.) Its arguments are identical to those for subroutine TRAPR. In fact, to use it in a program that previously called TRAPR, just change the name itself to SIMPR.

```
      SUBROUTINE SIMPR(EQUAT, XB, XE, VALUE, ERR, MAX)
      EXTERNAL EQUAT
      EV1=-1.E30
      EV=-1.E30
      VALU1=0.5*(XE-XB)*(EQUAT(XB)+EQUAT(XE))
      M=1
      I=0
   10 I=I+1
      DELX=(XE-XB)/FLOAT(M)
      X=XB+0.5*DELX
      SUM=0.
      DO 20 J=1,M
      SUM=SUM+EQUAT(X)
   20 X=X+DELX
      VALU1=0.5*(VALU1+(XE-XB)*SUM/FLOAT(M))
      M=2*M
      VALUE=(4.*VALU1-EV1)/3.
      IF(ABS(VALUE-EV).LT.ERR*ABS(EV)) RETURN
      EV=VALUE
      EV1=VALU1
      IF(I.LT.MAX) GO TO 10
      WRITE(*,*)' FAILED TO SATISFY ERROR REQ.', VALUE-EV
      RETURN
      END
```

Figure A.3 Program SIMPR.FOR. It integrates a function by repeatedly reducing Δx until the selected error criterion is satisfied.

Example Problem A.1

Integrate the function $f(x) = x^2(x^2 - 2)sin(x)$ between the limits of 0 and $\pi/2$ using first the trapezoidal rule and then Simpson's rule, and compare the results with the exact integral.

The exact indefinite integral is $F(x) = 4x(x^2 - 7)sin(x) - (x^4 - 14x^2 + 28)cos(x)$. The following programs have been written to complete the solution to this problem:

```
FORTRAN MAIN program and SUBROUTINE EQUAT to solve problem:
      PARAMETER (NMAX=21, A=0, B=1.5707963, ERR=1.E-5)
      EXTERNAL EQUAT
      CLOINT(X)=4.*X*(X**2-7.)*SIN(X)-(X**4-14.*X**2+28.)*COS(X)
      CALL TRAPR(EQUAT,A,B,VALUE,ERR,NMAX)   ! The name is changed to
SIMPR  to use Simpson's rule.
      WRITE(*,*) VALUE,CLOINT(B)-CLOINT(A)
      END
      FUNCTION EQUAT(X)
      EQUAT=X**2*(X**2-2.)*SIN(X)
      RETURN
      END
```

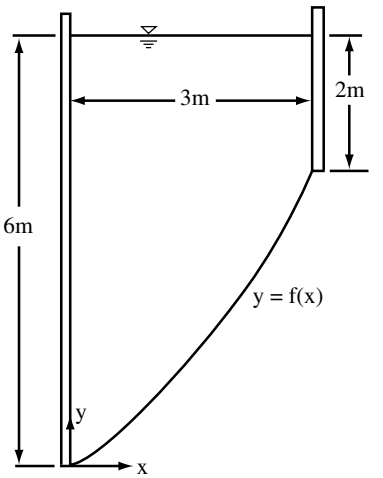
Upon executing the program, the following results can be compared:

Method	Result
Exact integral	- 4.791598E-1
Trapezoidal rule	- 4.791531E-1
Simpson's rule	- 4.791583E-1

Example Problem A.2

Find the equivalent concentrated vertical component of force, and its location, on the bottom surface that is 3 m long with water standing to a height of 6 m. The distance between vertical walls is 3 m. The surface is defined by the equation

$$y = f(x) = 2\left[x - \cos\left\{\frac{\pi}{2}\left(1 - \frac{x}{3}\right)\right\}\right]$$



The vertical component of the hydrostatic force on the bottom of the tank is simply the weight of fluid above it, that is, the product of the specific weight of the water and the fluid volume, which in turn is the product of the area and the 3 m length. Since the bottom of the tank is given as a function of x, the area can be determined by numerically integrating the differential area $dA = (6 - y)\,dx$ or

$$A = \int_0^3 \left(6 - 2\left[x - \cos\left\{\frac{\pi}{2}\left(1 - \frac{x}{3}\right)\right\}\right]\right) dx$$

Thereafter the centroid of this area will be determined from $Ax_c = \int_0^3 x(6 - y)dx$ The programs to obtain the area and the first moment of the area are listed next:

```
PROGRAM  SIMP1.FOR  TO INTEGRATE THE AREA
      PARAMETER (NMAX=21,A=0,B=3.,ERR=1.E-5)
      EXTERNAL EQUAT
      CALL SIMPR(EQUAT,A,B,VALUE,ERR,NMAX)
      WRITE(*,*) VALUE
      END
      FUNCTION EQUAT(X)
      EQUAT=6.-2.*(X-COS(1.5707963*(1.-X/3.)))
      RETURN
      END
PROGRAM  SIMP2.FOR  TO FIND THE FIRST MOMENT OF THE AREA
      PARAMETER (NMAX=21,A=0,B=3.,ERR=1.E-5)
      EXTERNAL EQUAT
```

```
CALL SIMPR(EQUAT,A,B,VALUE,ERR,NMAX)
      WRITE(*,*) VALUE
      END
      FUNCTION EQUAT(X)
      EQUAT=X*(6.-2.*(X-COS(1.5707963*(1.-X/3.))))
      RETURN
      END
```

The area is computed to be 12.820 m^2, and the solution for the first moment of the area is 16.295 m^3. Therefore the force on the bottom of the tank is $F = \gamma V = \gamma A b$ = 9.806(12.820)(3) = 377.1 kN. It acts downward at the position x_c = 16.295/12.820 = 1.271 m from the origin.

A.4 SOLUTIONS TO ORDINARY DIFFERENTIAL EQUATIONS

A.4.1. INTRODUCTION

The need to solve ordinary differential equations (ODE's) occurs frequently in many fields. Often closed-form solutions to these equations do not exist, and they must be solved numerically. General purpose mathematics application software, such as MathCAD and TK-Solver, facilitates the solution of ODE's and allows the user to select the method to be used. Pocket calculators, such as the HP48G(X), also have the ability to solve ODE's (in addition to numerical integration and algebraic integrations). In the following paragraphs a brief description of the Runge-Kutta method (one of many methods, but a widely used method) for solving ODE's is presented, to be followed by descriptions of how more sophisticated ODE solvers can be used.

A.4.2. RUNGE-KUTTA METHOD

The Runge-Kutta method of numerical integration is well known as a very dependable method, although it is neither very fast or efficient. This section will describe how to implement the Runge-Kutta method. The solver DVERK in IMSL (International Mathematical Statistical Libraries), which has been widely used for years and is included in Microsoft Fortran Powerstation and its descendants, uses a Runge-Kutta method. The logic, methods etc. in DVERK are more comprehensive than that in this description, but since several programs in this text call on DVERK, the reader should extract it from the CD along with ODESOL and RUKUST, which is the program that will be described herein. A description of how to use DVERK is in the file DVERK.DOC on the CD.

The Runge-Kutta method evaluates the dependent variable y after the next increment with $y_{i+1} = y_i + \Delta y$. The Euler predictor obtains Δy by multiplying the increment Δx in the independent variable by the derivative $dy/dx = y'$, evaluated at x_i, so that $\Delta y = \Delta x y'(x_i, y_i)$. Consider a trial step to the midpoint of the increment; now use x and y here to compute Δy, or $\Delta y = \Delta x y'(x_i + \Delta x/2,\ y_i + \Delta y_m)$, in which Δy_m is the Δy obtained from the Euler predictor for the midpoint. This way of obtaining Δy is a second-order approximation since the first-order terms cancel. This method of evaluating Δy is called the second-order Runge-Kutta, or midpoint, method. The derivative y' can be evaluated by using different combinations of the independent and dependent variables, and from these combinations different values of Δy can be obtained by multiplying by the appropriate Δx's. We define the following increments:

$$
\begin{aligned}
\Delta y_1 &= \Delta x \cdot y'(x_i, y_i) \\
\Delta y_2 &= \Delta y_m = \Delta x \cdot y'(x_i + \Delta x/2, y_i + \Delta y_m) = \Delta x \cdot y'(x_i + \Delta x/2, y_i + \Delta y_1/2) \\
\Delta y_3 &= \Delta x \cdot y'(x_i + \Delta x/2, y_i + \Delta y_2/2) \\
\Delta y_4 &= \Delta x \cdot y'(x_i + \Delta x, y_i + \Delta y_3)
\end{aligned}
\qquad \text{(A.10)}
$$

The first of these four equations is the Euler predictor, and the second of these equations is the midpoint method.

The fourth-order Runge-Kutta formula can be developed from these relations as

$$y_{i+1} = y_i + (\Delta y_1 + 2\Delta y_2 + 2\Delta y_3 + \Delta y_4)/6 \qquad \text{(A.11)}$$

This formula requires the derivative to be evaluated four times in order to advance one increment Δx, and an analysis of the terms that have been truncated from the final result would show that terms involving Δx^5 are dropped; therefore the result provides a fourth-order approximation. Computer code to implement this fourth-order Runge-Kutta formula can consist of Fortran statements (two versions are presented) in a subroutine, as presented in Fig. A.4 (C and Pascal statements are on the CD). In these subroutines the current values for the independent variable x, the increment Δx and the dependent variable y are passed as arguments X, DX, and Y, respectively. (In the C and Pascal programs these variables must be global, and consequently they are defined in the function or procedure.) The Fortran routine(s) could be modified so that these variables appear in a common statement rather than in arguments.

```
SUBROUTINE RUKU4(X, DX, Y)
XH=X+0.5*DX
DY1=DX*SLOPE(X,Y)
DY2=DX*SLOPE(XH,Y+0.5*DY1)
DY3=DX*SLOPE(XH,Y+0.5*DY2)
Y=Y+(DY1+DX*SLOPE(X+DX,Y+DY3))/6.+(DY2+DY3)/3.
RETURN
END

SUBROUTINE RUKU4A(X, DX, Y)
DX5=0.5*DX
XH=X+DX5
DY1=SLOPE(X,Y)           ! 1st sub-step
DY2=SLOPE(XH,Y+DX5*DY1) ! 2nd sub-step
DY3=SLOPE(XH,Y+DX5*DY2) ! 3rd sub-step
Y=Y+DX*((DY1+SLOPE(X+DX,Y+DX*DY3))/6.+(DY2+DY3)/3.)
RETURN
END
```

Figure A.4 Two alternative Fortran subroutines for the Runge-Kutta fourth-order formula.

The use of either Runge-Kutta subroutine requires a main program that calls it appropriately, and a FUNCTION Subprogram SLOPE to evaluate the derivatives. The listings in Fig. A.4 are designed to solve a single ODE.

If a system of ODE's is to be solved, as accommodated by solvers like DVERK and ODESOL, whose use is described below, then arrays for Y and its derivatives are needed. Let SLOPE be a subroutine that returns N derivatives for N ODE's in its last array argument, evaluated at X and Y, its first two arguments. (Y must also be an array.) Then the solver could consist of the subroutine that is listed in Fig. A.5.

The deficiency in using RUKU4 (or RUNK4S) is that the accuracy of the solution will be dependent upon the step size Δx that is used. One way to proceed would be to solve the ODE twice, once with some Δx and then with $\Delta x/2$, and if the solution agrees within an allowable error, accept the solution; otherwise reduce Δx again by one-half, etc. Rather than putting this burden on the user, it is much better to adjust the step size to satisfy some error criterion. The step sizes may then be decreased or increased, as suggested by the accuracy of the solution being obtained. To automate this step, an estimate of the

```
SUBROUTINE RUKU4S(N, X, DX, Y)
      PARAMETER (NM=5)
      REAL Y(N),YT(NM),DY1(NM),DYT(NM),DYM(NM)
      DX5=0.5*DX
      XH=X+DX5
      CALL SLOPE(X,Y,DY1)       ! 1st sub-step
      DO 10 I=1,N
   10 YT(I)=Y(I)+DX5*DY1(I)
      CALL SLOPE(XH,YT,DYT)     ! 2nd sub-step
      DO 20 I=1,N
   20 YT(I)=Y(I)+DX5*DYT(I)
      CALL SLOPE(XH,YT,DYM)     ! 3rd sub-step
      DO 30 I=1,N
      YT(I)=Y(I)+DX*DYM(I)
   30 DYM(I)=DYM(I)+DYT(I)
      CALL SLOPE(X+DX,YT,DYT)   ! 4th sub-step
      DO 40 I=1,N
   40 Y(I)=Y(I)+DX*((DY1(I)+DYT(I))/6.+DYM(I)/3.)
      RETURN
      END
```

Figure A.5 Two alternative Fortran subroutines for the Runge-Kutta fourth-order formula.

error is needed. This estimate can be obtained with a "step doubling;" i.e. each step is repeated, once using the full Δx and then independently as two half steps $\Delta x/2$. Each of the three separate Runge-Kutta steps that are needed in using this procedure require four evaluations of y', but the single and double computations initially share common arguments of x and y, so the required total number of evaluations of y' is 11. Let the exact solution be denoted by y (without a subscript), the solution based on Δx by y_1, and the solution based on $\Delta x/2$ by y_2. Using the fourth-order Runge-Kutta method, the exact and two numerical solutions are related by

$$\begin{aligned} y(x+\Delta x) &= y_1 + C(\Delta x)^5 \\ y(x+\Delta x) &= y_2 + 2C(\Delta x/2)^5 \end{aligned} \tag{A.12}$$

in which C should remain constant over the step since the Taylor series representation of $C = (d^5y/dx^5)/5!$. Since y_1 contains $C\Delta x5$ and y_2 contains $C\Delta x^5/16$, the difference between the two solutions provides a convenient estimate of the error as

$$ERR = y_2 - y_1 \tag{A.13}$$

Hence the exact solution can be expressed as

$$y(x+\Delta x) = y_2 + ERR/15 + O(\Delta x^6) \tag{A.14}$$

To develop a criterion to decide whether Dx should be changed to satisfy an accuracy requirement, let ERR_1 be the error from using Δx_1. Then the step size Δx_o to produce an error of ERR_o can be estimated as

$$\Delta x_o = \Delta x_1 \,|\, ERR_o / ERR_1 \,|^{1/5} \tag{A.15}$$

Let ERR_o be the error associated with the desired accuracy. If $|\mathrm{ERR}_1|$ is larger than ERR_o, then the above equation gives the $\Delta x = \Delta x_o$ to use to recompute the solution

over the failed increment to satisfy the error condition ERR_o. In other words, if $|ERR_1|$ is larger than ERR_o, then the computations over Δx_1 did not satisfy the error requirement and must be repeated with a smaller increment given by Eq. A.15. If $|ERR_1|$ is less than ERR_o, then Eq. A.15 provides the Δx to use for the solution over the next step. Thus the most recent step in the solution exceeds the desired accuracy and will be accepted, but the next increment will be enlarged to avoid doing more arithmetic than is necessary to satisfy the error ERR_o, as given by Eq. A.15. For a system of ODE's the errors ERR_1 are an array of values, and the largest in magnitude should be used in Eq. A.15. The listing in Fig. A.6 includes logic to redetermine the step size to satisfy the error condition associated with the magnitude of ERROR.

```
      SUBROUTINE RUKUST(N, DXS, XBEG, XEND, ERROR, Y, YTT)
      PARAMETER (NM=5)
      REAL Y(N),YTT(N),YORI(NM)
      X1=XBEG
      DX=DXS
    1 DO 10 I=1,N
      YTT(I)=Y(I)
   10 YORI(I)=Y(I)
      X=X1
      IF(ABS(X+DX).GT.ABS(XEND)) DX=XEND-X
   20 DX5=0.5*DX
      CALL RUKU4S(N,X,DX5,Y)    ! Solve with half increment
      CALL RUKU4S(N,X+DX5,DX5,Y)
      X1=X+DX
      IF(ABS(X1).GT.ABS(XEND)-1.E-8) RETURN
      CALL RUKU4S(N,X,DX,YTT)    ! Solve with full increment
      ERRM=0.0
      X1=X+DX
      DO 30 I=1,N
      YTT(I)=Y(I)-YTT(I)
   30 ERRM=MAX(ERRM,ABS(YTT(I)/Y(I)))
      IF(ERRM.EQ.0.0) THEN
      DX=5.*DX
      DXS=DX
      GO TO 1
      ELSE
      ERRM=ERRM/ERROR
      DX=DX/ERRM**0.2
      DXS=DX
      IF(ERRM.GT. 1.0) THEN
      DO 40 I=1,N
      YTT(I)=YORI(I)
40    Y(I)=YORI(I)
      GO TO 20
      ENDIF
      ENDIF
      DO 50 I=1,N
50    Y(I)=Y(I)+YTT(I)/15.   ! Accounts for truncation error
      GO TO 1
      END
```

Figure A.6 A fourth-order Runge-Kutta code with automatic adjustment of step size.

The arguments for RUKUST now have the following meanings:

N = number of ODE's to be solved, and for which derivatives will be given.

DXS = an initial value for Δx. This value will be decreased or increased, depending upon what is needed to satisfy the error criterion. In previous Runge-Kutta subroutines DX was the interval over which the problem was solved. Now DXS normally will be smaller than this value. Upon returning from this subroutine, DXS is the Δx that was found to be satisfactory at the end of the solution, and it can be used as the initial increment in a subsequent call to RUKUST.

XBEG = the initial value of the independent variable.

XEND = the end value of the independent variable. The difference between XEND and XBEG was called DX in the previous subroutines.

ERROR = the error criterion to meet in obtaining the numerical solution.

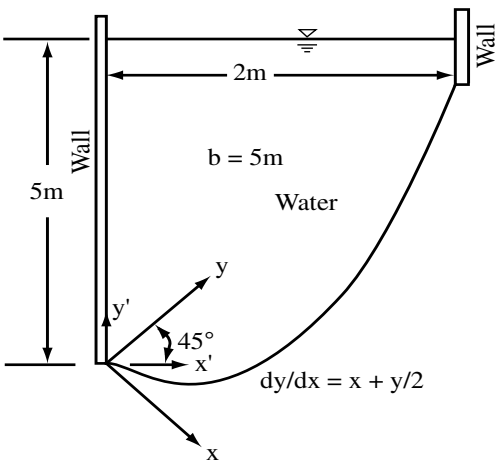
Y = an array of N values that, upon entry to the subroutine, provides the initial conditions for the dependent variable. Upon return from the subroutine it is the solution for the dependent variable(s) at x = XEND.

YTT = a work array of N values. It is used only to store the solution for the last interval, which is then compared with the solution Y in making decisions about the next increment in the independent variable to use in satisfying the error criterion.

Observe that this subroutine calls RUKU4S three times; the first two times complete the solution over the increment Δx in two steps of length $\Delta x/2$ (this solution is stored in array Y), and the third time uses the increment Δx, i.e. uses the four sub-steps in the Runge-Kutta method (and this solution is stored in the work array YTT). The difference between these two solutions is used to determine the error ERR (or ERR_1); then, based on Eq. A.15, the Δx that should supply the desired accuracy is computed. If the accuracy is insufficient, then the solution is repeated, using the computed Δx (the statement GO TO 20 does this). Another test checks whether the solution has proceeded to XEND. If not, then the solution proceeds over the next increment with the newly computed Δx by going back to statement 1. The program is required to end the solution at XEND, and this is accomplished by adjusting the last Δx so it equals the difference between the current value of x and XEND.

Example Problem A.3

The bottom of the tank is defined by the ODE $dy/dx = x + y/2$, measured in a coordinate system that is rotated downward 45° from the horizontal. The bottom is located between two vertical walls that are 2 m apart, and the tank is 5 m long. It contains water that is 5 m deep at the left wall. Find the vertical component of force on the bottom, and the location of its line of action.



To solve this problem, we must first solve the ODE to obtain the shape of the bottom and then numerically integrate from this bottom position to the water surface. The program below obtains the solution. We will need to establish the relation between the x-direction and the horizontal direction, denoted by x' in the sketch, to account for the rotated coordinate system. The differential area to integrate can be written $dA = hdx'$, in which h is the distance from the bottom of the tank to the water surface; from the sketch this distance is $h = H + x \sin\theta - y \cos\theta$, with $H = 5$ being the water depth at the origin. Also from the sketch $x' = x\cos\theta + y\sin\theta$. Since $\sin 45^o = \cos 45^o$, only $\sin 45^o$ will be used in the program. While there are alternate approaches, in this solution the ODE will be solved first to obtain the shape of the bottom, with the results stored in an array, YY; when needed, values from this array will be interpolated. We choose this approach because, although the numerical integration is in terms of x' and the corresponding value of x along the rotated axis will be needed, the only way to determine this x is to use $x'/\sin 45^o = x + y$. For any x', therefore, the table is searched for the entry where $x + y$ is just larger than $x'/\sin 45^o$, and then a linear interpolation is used to find x by writing $y = y_o + (\Delta y/\Delta x)(x - x_o)$ between the two entries in the table, with subscript o denoting the first entry. Thus $x'/\sin 45^o = y_o + (\Delta y/\Delta x)(x - x_o) + x$, and the solution is found to be $x = (x'/\sin 45^o - y_o + (\Delta y/\Delta x)x_o)/(\Delta y/\Delta x + 1)$. Once x is determined, y is interpolated as $y = y_o + (x - x_o)/[\Delta x(y_1 - y_o)]$. The ODE is solved first in the main program, and the function EQUAT performs the interpolations to obtain h so that Simpson's rule can properly evaluate the area. The solution produces a cross-sectional area of 9.44 m^2, leading to a force per unit length of 92.5 kN/m and a total vertical force F_V = 5x92.5 = 462.5 kN. The first moment of the area is determined by changing the EQUAT statement slightly, as the comment statement in the program shows, and the result of the integration gives $x_cA = 8.82$ m^3 so the line of action of this vertical force is at $x_c = 8.82/9.44 = 0.934$ m.

```
TANKODEK.FOR
      EXTERNAL EQUAT
      COMMON /TRAS/XX(30),YY(30),H,SIN45,DELX,II
      REAL Y(1),YTT(1)
      II=2
      WRITE(*,*)' GIVE XB,XE,H,Y0,GAMMA,ERR,MAX'
      READ(*,*) XB,XE,H,YO,GAMMA,ERR,MAX
      SIN45=SIN(0.7853982)
      SXY=(XE-XB)/SIN45
      TOL=0.000001
      DXS=0.01
      I=1
      DELX=(XE-XB)/50.
      XX(1)=XB
      YY(1)=YO
      Y(1)=YO
   10 X=XX(I)+DELX
      CALL RUKUST(1,DXS,XB,X,TOL,Y,YTT)
      I=I+1
      XX(I)=X
      YY(I)=Y(1)
      IF(X+Y(1).LT.SXY) GO TO 10
      CALL SIMPR(EQUAT,XB,XE,VALUE,ERR,MAX)
      WRITE(*,"(' AREA = ',F10.3,/,' FORCE = ',F10.3)")
     &VALUE,VALUE*GAMMA
      END
      SUBROUTINE SLOPE(X,Y,DY)
      REAL Y(1),DY(1)
      DY(1)=X+0.5*Y(1)
```

```
      RETURN
      END
      FUNCTION EQUAT(X)
      COMMON /TRAS/XX(30),YY(30),H,SIN45,DELX,II
      SX=X/SIN45
   10 IF(XX(II)+YY(II).GT.SX) GO TO 20
      II=II+1
      GO TO 10
   20 DYDX=(YY(II)-YY(II-1))/DELX
      XP=(SX-YY(II-1)+DYDX*XX(II-1))/(DYDX+1.)
      FAC=(XP-XX(II-1))/DELX
      YP=YY(II-1)+FAC*(YY(II)-YY(II-1))
      EQUAT=H+(XP-YP)*SIN45
C     EQUAT=(H+(XPYP)*SIN45)*X
      RETURN
      END
```

A.4.3. USE OF ODE SOLVER ODESOL

The subroutine ODESOL solves either a system of first-order ordinary differential equations (equations with first derivatives) or a higher-order ordinary differential equation. It utilizes an extrapolation with a modified midpoint that is called the Bulirsh-Stoer method. If a higher-order equation, of order N, is to be solved, it must first be reduced to a system (or coupled set) of N first-order differential equations. For example, if the second-order equation

$$\frac{d^2y}{dx^2} + f(x)\frac{dy}{dx} = g(x) \tag{A.16}$$

is to be solved, it is first rewritten as the following two first-order coupled equations:

$$\frac{dz}{dx} = g(x) - f(x)\cdot z(x) \tag{A.17a}$$

and

$$\frac{dy}{dx} = z(x) \tag{A.17b}$$

Subroutine ODESOL provides users considerable flexibility. A common use will call ODESOL repeatedly, with each new call over a new increment of the independent variable x, until the solution has been found over the desired range. Another use will call ODESOL once, with the end values of the desired range given for the independent variable. In this second use, intermediate values of the dependent variables (and the corresponding independent variable) can be stored and printed. In fact, these intermediate values can also be stored and examined when the first application, with several calls between the end values of the independent variable, is employed . The sizes of the arrays in ODESOL are established by integer values passed through arguments of the call; thus the amount of memory required by ODESOL is related to the size of the problem being solved. For a single first-order equation and a very limited storage of intermediate values, a very small amount of memory is required by ODESOL, e.g., that of its code and variables and the very small arrays that are passed as arguments. On the other hand, if a system of eight ordinary differential equations is being solved, the memory requirements for arrays will be larger. A smaller version that does not return intermediate values and does not use a blank COMMON statement is called ODESOLS.

The call in the driver program for ODESOL must contain a statement such as

```
CALL ODESOL(YBEG,DYDX,NV,X1,X2,ERR,H1,HMIN,NSTOR,XP,YP,WK1,SLOPE)
```
(A.18a)

or

```
CALL ODESOLS(YBEG,DYDX,NV,X1,X2,ERR,H1,HMIN,WK1,SLOPE)    (A.18b)
```

in which

YBEG = real array of dimension NV. Its elements are the dependent variables at X1 for which a solution is being sought. Before the call this array contains the starting values of the dependent variables, i.e. the values of the y's at X1. Upon return from the call, this array contains the dependent variables at X2.

DYDX = real array of dimension NV. Its elements are the derivatives of the dependent variables with respect to the independent variable x. The subroutine SLOPE defines these derivatives. The main program must dimension this array.

NV = integer variable, the number of first-order equations to be solved. In Eq. A.17 NV = 2.

X1 = the independent variable at the beginning of the solution interval.

X2 = the independent variable at the end of the solution interval. A solution will be obtained for x in the range between X1 and X2. Either X2 or X1 can be the smaller number.

ERR = real variable that defines the desired accuracy that ODESOL is to achieve. The step size will be changed as needed to achieve this accuracy.

H1 = the initial increment in x that ODESOL will use in obtaining the solution. It will be modified as needed to satisfy ERR. Normally ODESOL is called repeatedly to solve a problem over an extended range of the independent variable. When this is done, H1 will be used only in the first call to ODESOL. Thereafter the increment that was found to be appropriate in the previous call will be used. Therefore, it is best to provide H1 only in the first call to ODESOL.

HMIN = the minimum step size that will be allowed in seeking the solution. It may be set to zero (but this act may cause an infinite loop), and it is positive even if X2 is less than X1.

NSTOR = an integer that agrees with the dimensions of XP and the second dimension of YP. If KMAX in the common statement is 0, then NSTOR can be 1. NSTOR should never be less than 1.

XP = a one-dimensional real array of size NSTOR that contains the values of x upon return from ODESOL if KMAX is nonzero. These values will not be equally spaced but will range from X1 to X2.

YP = a two-dimensional real array of size NV (first subscript) and NSTOR (second subscript) that contains the values of the dependent variable upon return from ODESOL if NBETW is nonzero. The values in YP(I, J), with I between 1 and NV and J constant but between 1 and IBETW, will be the values of y corresponding to XP(J).

WK1 = a two-dimensional real array with NV as the first dimension and 13 as the second dimension. It is used for work space by ODESOL in obtaining the solution. Thus WK1 is dimensioned in the main program as WK1(NV, 13).

SLOPE must be declared as EXTERNAL in the main or driver program and is the name of the subroutine described below that defines the derivatives.

The main program only requires a COMMON statement when ODESOL is used; it must contain the five variables defined below. The statement should be similar to

```
COMMON NGOOD,NBAD,NBETW,IBETW,DXBETW                (A.19)
```

in which

NGOOD = an integer variable, the number of steps in the solution that equal or exceed the accuracy established by the error criterion ERR, i.e. good steps.

NBAD = an integer variable, the number of steps in the solution that did not meet the accuracy established by the error criterion ERR, i.e. bad steps.

NBETW = an integer variable, the maximum number of intermediate values of x and y in arrays XP and YP. If NBETW is zero, then no intermediate values will be returned. NBETW should not exceed the dimension of XP or YP, as defined by NSTOR, but the number of values in XP and YP will usually be less than NBETW. However, if the required step sizes are small, values will not be stored in XP and YP after NBETW values have been placed in these arrays.

IBETW = an integer variable, the number of intermediate values of x and y that are stored in XP and YP after returning from ODESOL. Hence the value of IBETW will never exceed NBETW.

DXBETW = a real variable, the smallest increment in the independent variable for which intermediate values will be stored in XP and YP. Should the increment that is needed to meet the error criterion ERR become less than DXBETW, then some values obtained in the solution process will not be stored.

The user of ODESOL must supply subroutine SLOPE to define the derivative(s) in the differential equation(s). The first statement should be

```
SUBROUTINE SLOPE(X, Y, DYDX)                    (A.20)
```

in which the name SLOPE is the EXTERNAL variable that is the last argument of the call to ODESOL. The arguments are

X = a real variable, the independent variable x. Its value will be passed from ODESOL to SLOPE to define the derivatives.

Y = a real array with the dimension NV. Values of this array pass from ODESOL to SLOPE to define the derivatives at X.

DYDX = a real array with dimension NV. Subroutine SLOPE must contain statements to define the elements of this array using X and the elements of Y to define each derivative of the individual dependent variables with respect to the independent variable x. The name must match the second argument of the call to ODESOL from the main program. Somewhere in subroutine SLOPE there must be a statement DYDX(J) = ... with J ranging from 1 through NV. SLOPE will be called repeatedly by ODESOL and must be written to provide the correct derivatives in the array DYDX which defines the system of ordinary differential equations that are being solved.

It would be a worthwhile exercise to use ODESOL to solve Example Problem A.2. To do so, first add SLOPE to the EXTERNAL declaration, and replace the REAL declaration at the beginning of the program with two statements:

```
COMMON NGOOD,NBAD,KMAX,ICOUNT,DXSAVE
REAL W(1,13),XP(1),YP(1,1),DY(1),Y(1)
```

Then replace DXS=0.01 with two statements H1=0.01 and HMIN=1.E-8, and replace the call to RUKUST with

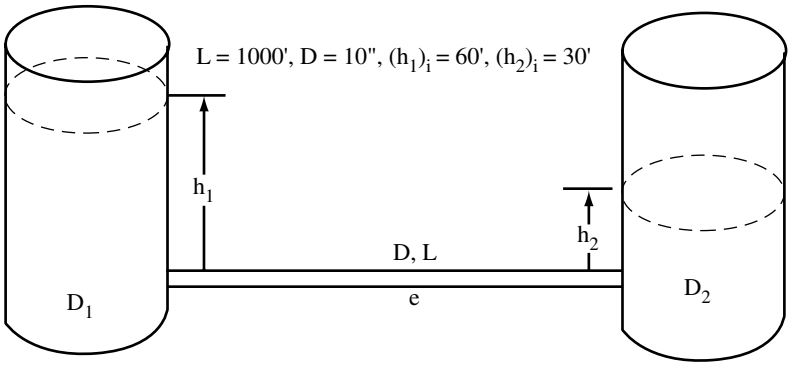
```
CALL ODESOL(Y,DY,1,XB,X,TOL,H1,HMIN,1,XP,YP,W,SLOPE)
```

Example Problem A.4

A pipe connects two tanks, as shown below, that have different water surface elevations; at time $t = 0$ a valve in the pipe is instantaneously and completely opened. Analyze the ensuing unsteady flow for a pipe with a length $L = 1000$ ft and a diameter $D = 10$ in, with tank diameters $D_1 = D_2 = 4$ ft. Initially the water surface elevations in the tanks are $h_1 = 60$ ft and $h_2 = 30$ ft. Assume all inertial effects are negligible in the tanks.

Obtain a solution for several situations:

(a) the fluid is idealized as inviscid, creating no entrance or exit losses;
(b) the fluid is idealized as inviscid, but assume that the velocity head into the second tank is lost;
(c) the fluid is real, and the pipe has a Darcy friction factor $f = 0.02$;
(d) the fluid is real, and the Darcy friction factor is to be determined for a pipe with an equivalent sand grain roughness $e = 0.005$ in.



The Fortran program TANKPI (a C version is on the CD) has been written to solve all four cases. It calls the solver RUKUST to obtain solutions to the ODE for the problem. The general ordinary differential equation for this problem is

$$\frac{dV}{dt} = \frac{g}{L}\left\{h_1 - h_2 - \left(K_e + 1 + f\frac{L}{D}\right)\frac{V|V|}{2g}\right\}$$

in which K_e is the entrance loss coefficient and V is the velocity in the pipe. The velocity in each tank is found by multiplying the pipe velocity by the ratio of the pipe and tank cross-sectional areas; for example, for the first tank $dh_1/dt = -(D/D_1)^2 V$. Thus the changes in tank water surface elevation over any time increment Δt, based on the trapezoidal rule for numerical integration, are given by

$$\Delta h_1 = -\left(D/D_1\right)^2\left(V_p + V_{pi}\right)\Delta t/2$$

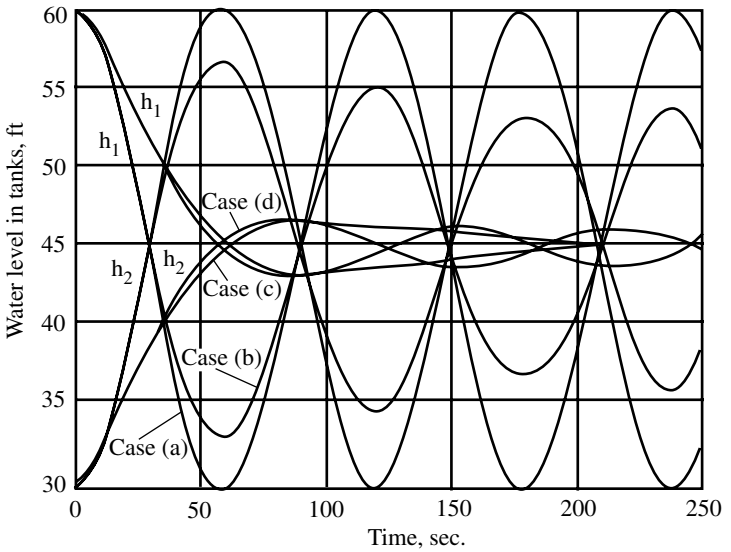
and

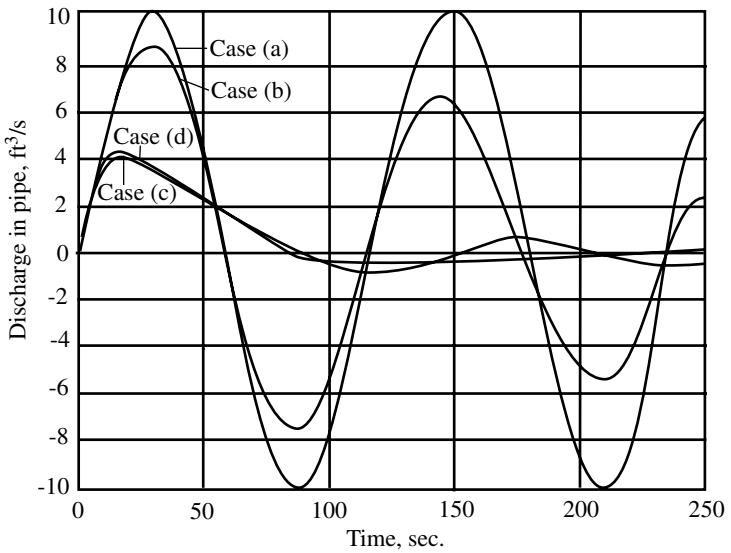
$$\Delta h_2 = +\left(D/D_2\right)^2\left(V_p + V_{pi}\right)\Delta t/2$$

This numerical integration is needed in the main program, where Δt is the time increment specified in the input, and within subroutine SLOPE since it will be called for times that will be determined by the ODE solver RUKUST.

For case (a) all terms after h_2 in the ODE are ignored, so that $dV/dt = g(h_1 - h_2)/L$ is to be solved. Case (b) is obtained by setting K_e and f to zero. For case (c) the program uses $EK1 = K_e + 1 + fL/D$ as the multiplier of the velocity head. In case (d) the Colebrook-White equation is solved for f, unless the Reynolds number is below 2100, in which case $f = 64/Re$. If $Re < 100$, then $f = 0.64$.

The results for the four cases are plotted on the two graphs which follow. In case (a) the water surface elevations in the tanks oscillate repeatedly between elevations 60 ft and 30 ft. For all of the other cases the magnitude of the oscillations is damped, or reduced, with time, but the rates of change differ from case to case.





The variables that are read by the program have the following meanings: IOUT = number of the logic unit for written output, L = pipe length (ft or m), E = equivalent sand grain roughness (ft or m), D = pipe diameter (ft or m), D1 = diameter of tank one (ft or m), D2 = diameter of tank two (ft or m), VI = initial velocity in pipe (ft/s or m/s), H1 = head in tank one (ft or m), H2 = head in tank two (ft or m), G = acceleration of gravity, KE = entrance loss coefficient, NT = number of time steps for the solution, DELT = time increment (s), VISC = fluid kinematic viscosity (ft^2/s or m^2/s).

```
TANKPI.FOR
      REAL V(1),VTT(1)
      COMMON D,FL,GL,AR1,AR2,H1,H2,COE,VISC,VI,SF,G2,EK1,EK,E,TIM1
C IF E=-10 THEN THE DOWNSTREAM VELOCITY HEAD IS LOST; IF E = - ABS(F),
C THEN F = CONSTANT; OTHERWISE KE IS THE ENTRANCE LOSS AND E IS THE
C EQUIVALENT SAND GRAIN ROUGHNESS
      WRITE(*,*) 'GIVE IOUT,L,E,D,D1,D2,VI,H1,H2,G,EK,NT,DELT,VISC'
      DTS=0.05
      V(1)=VI
      G2=2.*G
      GL=G/FL
      AR1=(D/D1)**2/2.
      AR2=(D/D2)**2/2.
      AREA=0.78539816*D**2
      IF(ABS(EK).GT.1.E-7) EK=(1.+EK)/G2
      IF(E.LT.0.) EK=EK+ABS(E)*FL/(D*G2)
      EK1=EK
      IF(E.LT.-9.) EK1=1./G2
      TIM1=0.
      WRITE(IOUT,100) TIM1,VI,AREA*VI,H1,H2
  100 FORMAT(F5.1,4F10.3)
      DO 10 I=1,NT
      TIME=DELT*FLOAT(I)
      CALL RUKUST(1,DTS,TIM1,TIME,1.E-6,V,VTT)
      H1=H1-AR1*V(1)+VI)*DELT
      H2=H2+AR2*(V(1)+VI)*DELT
      WRITE(IOUT,100) TIME,V(1),AREA*V(1),H1,H2
      VI=V(1)
   10 TIM1=TIME
      END
      SUBROUTINE SLOPE(T,V,DVT)
      REAL V(1),DVT(1)
      COMMON D,FL,GL,AR1,AR2,H1,H2,COE,VISC,VI,SF,G2,EK1,EK,E,TIM1
      IF(E.GT.1.E-7) THEN
      RE=V(1)*D/VISC
      IF(RE.LT.100.) THEN
      F=0.64
      ELSE IF(RE.LT.2100.) THEN
      F=64./RE
      ELSE
    1 SF1=SF
      SF=1.14-2.*ALOG10(E/D+9.35*SF/RE)
      IF(ABS(SF-SF1).GT.1.E-6) GO TO 1
      F=1./SF/SF
      ENDIF
      EK1=EK+F*FL/D/G2
      ENDIF
      DVT(1)=GL*(H1-H2-(V(1)+VI)*(AR2+AR1)*(T-TIM1)-EK1*V(1)*ABS(V(1)))
      RETURN
      END
```

Modifying this program to use ODESOL and/or DVERK (part of IMSL) in place of RUKUST is an instructive exercise.