Contents

Chapter 1

Programming Examples

This chapter illustrates how we may apply the floating point extensions of FORTH, developed in preceding chapters, to some standard problems in numerical analysis.

§1 Infinite series

Frequently we must evaluate a function defined by an infinite sum

$$f(x) = sum_{n=0}^{\infty} c_n x^n \tag{1.1}$$

where x is a real number and c_n is an infinite sequence of coefficients. The extensive mathematical theory[1] of such functions tions can be summarized as follows: the series of terms only have meaning if -for fixed x- the sequence of partial sums

$$f_N(x) = sum_{n=0}^{N-1} c_n x^x \tag{1.2}$$

has a definite limit.

[1] *The Handbook of Mathematical Functions* ed. Milton Abramowitz and Irene Stegun (Dover Hlications, Inc, New York, 1965) –henceforth abbreviated **HMF**– is a mine of useful information and references, on this as well as many other aspects of numerical analysis.

§§1 Examples of Infinite series

Perhaps the formal definitions will seem clearer after some concrete examples. We now examine some divergent an convergent series.

Divergent examples

What does it mean to say a series diverges? Consider the serices whose terms are all 1's (that is, $c_n = 1, x = 1$):

$$1 + 1 + 1 + 1 + ... \tag{1.3}$$

The partial sum f_N is just N, and therefore increases without limit as more terms are added. The series *diverges*.

A harder case is the series whose terms are alternately 1's and -1's:

$$1 - 1 + 1 - 1 + 1 - ... \tag{1.4}$$

Depending on how the terms are grouped together, the partial sums can have any value. Certainly the partial sums do not settle down to any definite value. The series *diverges*.

Yet a third case is the harmonic series

$$f_N = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{N} \tag{1.5}$$

It can be shown that for large N, f_N is approximately $log_e(N)$, so the harmonic series *diverges*.

A convergent example

Are there ever mes of infinite series that *do* mean something? Obviously, or there could hardly be a theory of them! An example[2] is $c_n = 1, x = \frac{1}{2}$. Here the partial sums

[2]This is the examle of **Zeno's paradox** where you are at one end of a sofa and an attractive person of the opposite gender is at the other end. You move half the distance, then half the remainder. Clearly an an infinite number of moves is necessary to achieve the desired proximity. Thus, according to Zeno, you never get there. According to Eq. 6, however, you *do* get there and in a **finite** time, to boot!

$$f_0 = 1 \tag{1.6}$$

$$f_1 = 1 + \frac{1}{2} \tag{1.7}$$

$$f_2 = 1 + \frac{1}{2} + \frac{1}{4} \tag{1.8}$$

$$\ldots\ldots\ldots\ldots\ldots \tag{1.9}$$

$$\lim_{N\to\infty} f_N \equiv \lim_{N\to\infty} \frac{1-(\frac{1}{2})^N}{1-\frac{1}{2}} = 2 \tag{1.10}$$

do have a definite limit, so the series converges.

§§2 Numerical examples of convergent series

How do we tell whether a series has converged? As a practice matter, we keep adding terms to the partial sum until it no longer changes, within the desired precision. Sometimes this can involve a great many terms, even when the series converges. There exists an extensive mathematical literature on testing for the convergence of an infinite series. Mathematicians have also developed many tricks for accelerating the convergence of slow converging series[3]. Consulting the literature in difficult cases I strongly recommended - it can save time galore!

As an heuristic exercise, let us write some simple programs to evaluate partial sums and see how convergence works.

first we sum the terms 2^{-n} from Eq. 6. The flow diagram is shown in Fig. 6-1 below.

```
SETUP: N=0 SUM=1
BEGIN
?CONTINUE ( 10 more terms?)
WHILE
NO YES
10 0 DO
Exit 2^{-n-1} = 2^{-n} / 2
SUM = SUM + 2^{-n-1}
n = n + 1
LOOP
REPEAT
```

Fig. 6-1 *Computing 2 the hard way*

[3] **HMF**, Ch. 3 §6 ff.

The corresponding program is

```
15 #PLACES !                \ set F. to 15 digits
: NEXT.TERM
    ( n -- n+1 :: sum 2^-n -- sum'2^-n-1 )
    F2/ FUNDER F+ FSWAP 1+ ;
: SET.UP FINIT F=1 F=1 0 ;
: EXHIBIT CR DUP ." n = " .
    2 SPACES
    FOVER ." sum = " F. ;
: ?CONTINUE  CR ." Another 10 terms?" ?YN ;
\ ?YN expects a "y" or "n" from the keyboard and
\ leaves -1 ( "true" ) if "y" is pressed, 0 if "n"
: SUM SETUP
    BEGIN EXHIBIT ?CONTINUE
    WHILE 10 0 DO NEXT.TERM LOOP
    REPEAT ;
```

This is what a run looks like:

```
FLOAD TEST.SUM Loading TEST.SUM ok
SUM
n = 0  sum = 1.00000000000000
Another 10 terms? Y
n = 10 sum = 1.919902343750000
Another 10 terms? Y
n = 20 sum = 1.99999904632568
Another 10 terms? Y
n = 30 sum = 1.99999999906867
Another 10 terms? Y
n = 40 sum = 1.99999999999909
Another 10 terms? N ok
```

The partial sums converge rapidly to 2 (which, as we saw in Ch. 1.1.2 above, is the exact sum).

For a second example, let us sum a standard infinite series representation for $\pi/4$: We note that the function $\tan^{-1}(x)$(that is, arctan(x)) has the infinite series representation[4]

$$\tan x^{-1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \ldots \tag{1.11}$$

Since a 45° right triangle has equal height and base, the tangent of 45° is 1 (side

[4] **HMF**, Ch. 4 §4.42

opposite over side adjacent). That is[5],

$$\frac{\pi}{4} \equiv \tan 1^{-1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \ldots \tag{1.12}$$

Actually the series Eq. 8 is slowly converging and therefore a poor way to compute π[6]. But anyway, let us proceed. The flow diagram is now that of Fig. 6-2 below.

```
SETUP:  flag=1  n=1  sum=0  (--1n::--O)

BEGIN
     EXHIBIT \ display n, sum
     ?CONTINUE WHILE

     DUP S->F 1/F      ( -- fn :: -- sum 1/n )
     SWAP FDUP FSIGN   \ transfer sign
     F+                \ sum = sum +
     term
     NEGATE            \ flip sign of f
REPEAT
```

Fig. 6-2 *Computing* $\pi/4$ *by the infinite series for* $\tan 1^{-1}$

The corresponding program is

```
15 #PIACES ! \ set F. to 15 digits
: NEXT.TERM ( f 2n+1 -- f1 2n+3 :: sum -- sum1 )
     DUP S->F 1/F SWAP DUP FSIGN F+
     NEGATE SWAP 2+ ;
: SETUP FINIT F=0 1 1 ;
: EXHIBIT CR DUP
     ." n = " . Z SPACES
     FDUP F2* F2* ." 4*sum = " F. ;
: ?CONTINUE ." Another term?' ?YN ;

: PI/4 SET.UP
     BEGIN EXHIBIT
     ?CONTINUE
     WHILE NEXT.TERM REPEAT ;
```

Now we run the program[7]:

---

[5]Since $\pi/4$ radians is 45°.

[6]A better method would be to evaluate the series for $x = 1\sqrt{3}$ which is the tangent of $\pi/6$, or $x = \sqrt{2} - 1$, the tangent of $\pi/8$

[7]Note we have set up to display π rather than $\pi/4$.

```
FLOAD PIBY4 Loading PIBY4 ok
PI/4

n = 1   4*num = Another term? Y
n = 3   4*num = Another term? Y
n = 5   4*num = Another term? Y
n = 7   4*num = Another term? Y
n = 9   4*num = Another term? Y
n = 11  4*num = Another term? Y
n = 13  4*num = Another term? Y
n = 15  4*num = Another term? Y
n = 17  4*num = Another term? Y
n = 19  4*num = Another term? Y
n = 21  4*num = Another term? Y
n = 23  4*num = Another term? Y

n = 25  4*num = Another term? Y
n = 27  4*num = Another term? Y
n = 29  4*num = Another term? Y
n = 31  4*num = Another term? Y
n = 33  4*num = Another term? Y
n = 35  4*num = Another term? Y
n = 37  4*num = Another term? Y
n = 39  4*num = Another term? Y
n = 41  4*num = Another term? Y
n = 43  4*num = Another term? Y
n = 45  4*num = Another term? Y
n = 47  4*num = Another term? Y
```

The first thing we notice is that the numbers seem to be converging to *something*; however, unlike the previous series, the differences between successive partial sums are fairly large.

An infinite series of terms that alternate in sign and decrease in magnitude is guaranteed to converge[8]. The error (that is, the difference between a partial sum and the limit) is of the order of the first neglected term and has the same sign. We see that for this case, the error in computing π is of order $2/n$, where n is the number of terms in the sum. To get π to 3 significant figures, therefore, we need about 1000 terms! This is why the series representation for $4\tan 1^{-1}$ is not a very good way to calculate π A better way uses *e.g.*,

$$\pi = 16\tan\frac{1}{5}^{-1} - 4\tan\frac{1}{239} \tag{1.13}$$

[8]This theorem, due to Weierstrass, is found in all standard intermediate-level calculus texts.

, which converges much faster[9].

The infinite sum program

Evaluating a function from its infinite series representation Eq. 1 provides an illustration both of indefinite loops and of tests of floating point numbers. We anticipate a program structure something like this:

```
BEGIN
   Calculate next term
   Not converged?
WHILE
   Update:
      Add next term to sum
      increment n
REPEAT
```

To actually write the program, we begin at the end, by specifying how we want to invoke the function.

Functions in the standard FORTH library (Ch. 3.3) typically expect a single real number on the fstack replacing it with the function: $x \rightarrow f(x)$. Since the sum is infinite, we supply the coeifcients c_n as a function of n rather than as an array. For any given $c_n = f(n)$, it is easy enough to write a FORTH word that evaluates it and leaves the result on the appropriate stack (87stack, fstack, ifstack). For example, to evaluate the exponential *via* the power series

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + ... \tag{1.14}$$

we would define the word NEXT.TERM as

```
: NEXT.TERM ( 87: term x n -- term*x/[n+1] x n+1 )
   F=1 F+ \n-> n+1
   FROT FOVER F/ ( 87: -- x n+1 term/[n+1])
   FROT FUNDER F* F-ROT ;
```

Function calls In FORTRAN

However, FORTRAN (as well as languages that emulate it) achieves readability by passing arguments to functions in a list. In fact, function names can also be passed in the argument list. Thus, *e.g.*, a general FORTRAN program to evaluate a function by summing an infinite series could be written

[9] See, *e.g.*, J, Mathews and R.L. Walker, *Mathematical Methods of Physics*, 2nd ed. (WA. Benjamin, Inc, New Jersey, 1970).

```
      REAL FUNCTION XINFSUM(X. E. C)
C
C     EVALUATE INFINITE POWER SERIES
C
      EXIERNAL C
      REAL X, E, C
      SUM  = 0
      TERM = 1
      N = 1
1     SUM  = SUM + TERM
      TERM = TERM*X*C(N)
      N = N + 1
      IF (TERM .LE. E) RETURN
      GOTO 1
      END
\end{align}
```

where the program to evaluate the exponential would be

```
\begin{lstlisting}
      REAL FUNCTION EXP(x)
      EXTERNAL XINFSUM, COEFEXP
      REAL C0, XINFSUM
      COMMON /C8LK/C0
      C0 = 1
      EXP = XINFSUM(X, 1.E-7, C0EFEXP)
      RETURN
      END
```

given the coefficient function

```
REAL FUNCTION OOEFEXP(N)
COMMON /C8LK/C0
C0 = C0/N
COEFEXP = C0
RETURN
END
```

A function protocol for FORTH

We would like to extend to FORTH FORTRANs ability to write a generic series summation function, passing the variable and the name of the coefficient function as arguments. In other words, we now face the task of devising the function protocol we plan to use throughout the rest of the book and in our future programming - a heavy responsibility since we do not know what

form these future programs will take. **For once we must engage in top-dawn programming!**

We want our protocol to have several features:

- It must be **telegraphic**, *i.e.*it must immediately suggest what it is doing - a matter of choosing good names.
- It must be simple to implement and easy to remember - the advantages of using it must not be outweighed by complexity.
- It must be fast - a major drawback to FORTRAN's way of doing things is the overhead in function calls.
- It must be portable – it cannot depend on specific details of the FORTH implementation or the machine 1t 1s running on.

It is simpler to thread this particular maze in reverse: begin with where we want to end up, and determine what steps got us there. We suppose we have defined a generic power series summation function, using the ifstack defined in Ch. 5.2.5:

```
    : SUMPOWERS          ( 87: err -- :: x -- sum )
    E G!                 \ store error
    FS>F DUP F>FS        \ get type of x
        DUP G=1          \ x**0
        G=0 0 DUP        ( -- n=0 :: -- x 1 0 )
    adr.c EXECUTE G+     ( -- 0   :: -- x 1 c[0] )
BEGIN 1+ --n::~ - x x  n -13um)
    FS>F GOVER G*
        ( -- n 87: -- sum :: -- x x**n)
    DUP adr.c EXECUTE
        ( -- n :: -- x x**n )
    3 GPICK G* F>FS
        ( -- n :: -- x x**n term sum )
    ENUF? NOT WHILE
        G+
    REPEAT G+ CLEANUP ;
```

The words E, ENUF? and CLEANUP have straightforward definitions with obvious meanings; adr.c has not been defined because we have yet to figure out what it will do.

Manifestly, edr.c must place the execution address ("code-field address" – cfa) on the stack for EXECUTE to find. There are several ways to accomplish this. Clearly, we want to keep the variable that holds the cfa local, so it will not get confused with another function's cfa. While it is straightforward to define a variable and then make it headerless –hence local– (via BEHEAD", e.g.) we would prefer to avoid defining a variable at all. Here is a perfect opportunity to use the return stack (rstack).

We imagine that SUM.POWERS expects the cfa of the function c_n on the stack. Then the first thing SUM.POWERS must do is stash the cfa somewhere convenient but local. One such place is the stack itself. Even easier, since SUM.POWERS does not use the rstack explicitly (e.g. in a DO ... LOOP), is to let the first step in SUM.POWERS be ¿R. Then the code for adr.c would be merely R@. The final word, after invoking CLEAN.UP (that drops unwanted items from the various stacks), then must be RDROP (for systems without it, : RDROP R¿ DROP ;).

The revised version of SUM.POWERS is then

```
: SUM.POWERS     ( adr.c -- 87: err -- :: x -- sum )
    > R              \ adr.c -> rstack
    E G!             \ store error
    FS>F  DUP F>FS   \ get x's type
          DUP G=1    \ x**0
    G=0  0 DUP       ( -- n=0 :: -- x 1 0 )
    R@ EXECUTE G+    ( -- 0   :: -- x 1 c[0] )
    BEGIN 1+         ( -- n   :: -- x x**n-1 sum )
        FS>F GOVER G*  ( -- n 87: -- sum :: -- x x**r )
        DUP R@ EXECUTE ( -- n:: -- x x**n c[n] )
        3 GPICK G* F>FS( -- n:: -- x x**n term sum )
        GOVER
    ENUF? NOT    WHILE G+ REPEAT
    G+ CLEAN.UP RDROP ;
```

The full program to evaluate the exponential by summing power series would then have the form shown below:

evaluate exponential by summing series REAL*4 SCALAR E : ENUF? (::term–) (–F) GABS FS¿F EG@ F¿ ; : CLEAN.UP (n– ::x x**n sum – sum) DROP FS ¿F GDROP GDROP F¿FS ; definition of SUM.POWERS from above BEHEAD" E CLEAN.UP hide these def'ns REAL*8 SCALAR c F=1 c G! : COEEEXP (n–) (:: – c[n]) DUP 1 ¡ = IF F=1 c G! F=1 DROP ELSE c G@ S-¿F F* FDUP c G! 1/F

THEN REAL*4 F¿FS ; BEHEAD' c hide this variahie : USE([COMPILE] ' CFA LITERAL ; IMMEDIATE this means "use address of" –crucial def'n of function lexicon :E**X (::x–e**x) error on 87stack USE(COEFF.EXP adr.c on stack SUMPOWERS ;

§2 Transcendental equations

A transcendental equation has the form

f(x) = 0,

where f(x) is a transcendental function rather than, say, a polynomial or ratio of polynomials[10] (**rational** function).

There are several standard methods for finding a (or possibly, the) value of x that satisfies this equation, i.e. a root of Eq. 10 (which might have many or no roots). To guarantee that we find a root we must know an interval of the x-axis that it certainly will be found in. Two methods that can be applied under these circumstances are binomial search and regula falsi.

§§1 Binary search

Let us look first at binomial search, since its algorithm is euy to understand. We know some interval, $x_L \leq x \leq x_r$, contains a root because $f(x)$ changes sign when x goes from $x_L \rightarrow x_R$. In pseudocode (and FORTH flow chart) the binomial search algorithm is shown in Fig. 6-3 on page 128.

The method begins with upper and lower bounds on x that capture the root. Next we look at $f(x_A V)$ halfway between x_L and x_R. If $f = f(x_A V)$ has the same sign as $f_L = f(x_L)$, the new left end of the interval becomes $x_A V$. If the signs are opposite, $x_A V$ becomes the new right end of the interval. The algorithm is done when left and right ends agree within some predetermined accuracy.

Binary search has the following virtues: the time it takes to achieve a given accuraqr is predictable, and it is guaranteed to find a captured root. Creating a FORTH program from the pseudocode skeleton of Fig. 6-3 is left as an exercise.

```
INITIALIZE
f_L = f(x_L) f_R= f(x_r)

BEGIN

|x_L − x_R| > error ?
WHILE
x=1/2(x_L+x_R), f=f(x)
sign(f) = sign(f_L) ?
IF x_L =X f_L =f
ELSE x_R =x f_R=f
THEN

REPEAT
```

Fig. 6-3 Binary search algorinn for roots of f(x)

[10] We specialize to transcendental equations because our root-finding methods will work also for polynomials, whereas the methods developed for polynomials will not work in the more general case.

§§2 Regula falsi

Now we look at *regula falsi*, Latin for "rule of false approach" Here the basic premise is:

- Assume the root lies in the interval (x1, x3, and plot a straight line between the points (x_L, f_L) and (x_R, f_R).
- This line must intersect the x-axis somewhere in the interval, and we take that point, call It x, as our next guess.
- If x is to the left of the root, adjust the interval accordingly, an the same if x is to the right of the root.

As Fig. 6-4 on page 129 shows, the straight line is supposed approximate the curve f(x). The new guess may be much closer to the root than is the midpoint of the interval (which was the next guess in binomial search).

Fig. 6-4 Graphical Illustration of ragula falsi

A straight line in the x-y plane has the analytic form

y = ax + b

where a and b are constants. The intercept of the straight line with the x-axis is gotten by setting y = 0 and solving for x:

$$x' = \frac{-b}{a} \tag{1.15}$$

To determine a and b we use the two equations

$$f_L = ax_L + bf_R = ax_R + b \tag{1.16}$$

giving

$$a = \frac{1}{2}[f_L + f_R - \frac{f_R - f_L}{x_L + x_R}(x_L + x_R) \tag{1.17}$$

and thus

$$x' = \frac{f_R x_L - f_L x_R}{f_R - f_L} \tag{1.18}$$

A FORTH flow diagram for this algorithm appears in Fig. 6-5.

```
INITIALIZE
f_L=f(x_L)  f_R=f(x_R)
x_old = X_L
```

```
BEGIN calculate x
|x' - x_old| > error
?
WHILE
f\hat = f(x')
sgn(f\hat) = sgn(f_L) ?
IF x_L = x f_L = f

ELSE x_R = x f_R = f

THEN

REPEAT
```

Fig. 6-5 Regula falsi algorithm for roots of f(x)

The corresponding program is shown on page 132.

Here is an example of the program in action:

```
FINIT 6 #PLACE ! ok \ set display to 6 digits
\ Example: f(x) = exp(-x) - x
: FNA FDUP FNEGATE FEXP FR- ;
\ Clearly, the root lies between 0 and 3. (Why?)

USE( FNA % 0. % .3 % 1.E-6 )FALSI
\ st(4) st(3) st(2)  x'       x'-xold
?????? ?????? ?????? .759452  -.291530
?????? ?????? ?????? .588025  -.0326025
?????? ?????? ?????? .569459  -.00362847
?????? ?????? ?????? .567400  -.000403560
?????? ?????? ?????? .567171  -.0000448740
?????? ?????? ?????? .567146  -.0000050002
?????? ?????? ?????? .567143  -.0000005544 ok
```

The display was generated by the word .FS –placed in the definition of APART? for debugging – we show the top 5 of the eight 80x87 registers. The ?????? means the contents of that register are not a properly defined fp number, either because of a mistake or because nothing was stored in them after FINIT. Here, since the program is obviously working, the latter explanation is the correct one.

Ordinary differential equations We wish to solve the first-order general differential equation

$$x \doteq \frac{dx}{dt} = f(x,t) \tag{1.19}$$

In general we can only solve Eq. 16 approximately, starting from the value of x

– call it x_0 – at some initial time t_0, then advancing the time by small increments dt = h, using the differential equation itself to give us x(t+h) given x(t).

For example, we could expand in Taylors series[11]

$$x(t+h) = x(t) + h\dot{x}(t) + \frac{h^2}{2}\ddot{x}(t) + ... \tag{1.20}$$

```
\ USE( Fname % a % b % err )FALSI
( 87:--rooot )
\ Fname is the name of a FORTH function

\ function notation
: USE( [COMPILE] ' CFA LITERAL ;
       IMMEDIATE

6 REAL*4 SCALARS ERR XL XR YL YR OLDX

0 VAR f1 \ a place to store cfa
: SAME.SIGN? (87:xy -- -- f)
    F* F0> ;

:INITIALIZE ( cfa -- 87: a  be -- )
    IS f1    \ stare cfa
    XDUP     \ interval has root?
    SAME.SIGN?
    ABORT" Even number of roots!!!"
    XR G! XL G!
    XL G@ f1 EXECUTE YL G!
    XR G@ f1 EXECUTE YR G!
    F=0 OLDX G! ;

: X' XL G@ FR G@ ( 87: -- x')
    FUNDER F*     ( 87: -- yR xL *yR)
    XR G@ YL G@
    FUNDER F* ( 87: -- yR xL*yR xL xR*yL )
    FROT   F- ( 87: -- yR yL xR*yL -xL*yR)
    F-ROT  F- F/ ;

: APART? ( 87: x' -- x' -- f )
    FDUP OLDX G@ F-
    .FS FABS ERR G@ F> ;

: REVISE ( 87: x' -- )
    FDUP f1 EXECUTE ( 87: -- x' y')
```

[11] See, e.g., **HMF** §3.6.1.

```
        FDUP YL G@
        SAMESIGN? FOVER
        ( --f 87: -- x'y' x')
        IF      XL G! YL G!
        ELSE    XR G! YR G! THEN
        OLDX G! ; ( -- 87: -- )

: )FALSI ( cfa-- 87: a b e -- root ) INITIALIZE ;
```

and keep only the lowest order terms:

$$x(t+h) \approx x(t) + hf(x(t), t). \tag{1.21}$$

Runge-Kutta method

One standard class of methods that had fallen into disfavor by now are popular again, are the Runge-Kutta algorithms[12]. The algorithms can be classified according to order n (that is, if h the step size, the error at each step will be $O(h^n)$. The second order Runge-Kutta algorithm is ($x' \equiv x(t+h), x \equiv x(t)$)

$$k = hf(x,t)x' = x + \frac{1}{2}(k + hf(x+k, t+h)) + O(h^3). \tag{1.22}$$

How does this work? Clearly,

$$k+hf(x+k,t+h) \approx hf(x,t)+hf(x,t)+h^2\frac{\partial f}{\partial t}+hk\frac{\partial f}{\partial x} \equiv 2h\dot{x}(t)+h^2\ddot{x}(t)+O(h^3) \tag{1.23}$$

Substituting 19 in 20 we now find

$$x' = x(t+h) = x(t) + h\dot{(t)} + \frac{h^2}{2}\ddot{x}(t); \tag{1.24}$$

that is, the Runge-Kutta x' agrees with the Taylor's series expansion 17, to $O(h^3)$.

The flow chart of second-order Runge-Kutta is shown in Fig. 6.6 below. We express the algorithm in FORTH as shown in Fig. 6-7 on page 134 below.

```
)RUNGE
INITIALIZE: get h, tf, t0, x0

BEGIN DISPLAY
      DONE? NOT
WHILE
```

[12] **HMF**, §25.5.6.

```
    STEP
REPEAT
\begin{lstlisting}
```

Fig. 6-6 2nd-order Runge-Kutta for dx/dt = f(x,t)

```
\begin{lstlisting}
    \ STRAIGHT 2ND ORDER RUNGE-KUTTA
    \ SOLUTION OF FIRST-ORDER DIFEQ
    \ dx/dt = f(x,t)

    \ See Abramowitz & Stegun, HMF 25.5.6

    \ Usege:
    \ USE( FNB % x0 % t0 % tf % h )RUNGE
    \
    \ FNB (:[t]-- 87:x--f[x,t]) evaluates f(x.t)
    \ x0 = starting value of dep. variable
    \ t0 = initial time
    \ tf = end time
    \ h = step.size
    \
    \ k = hf(x,t), x' = x + (k +hf(x+k,t+h))/2

    6 REAL*4 SCALARS T T' H X TMAX K
    0 VAR f1 \ to hold cfa
    : USE( [COMPILE]' CFA LITERAL;
          IMMEDIATE
    : INITIALIZE (:cfa-- 87: x0 t0 tf h -- )
       IS f1 H G! TMAX G! T G! X G! ;

    \ These words increment x & t.
    : inc.T     T G@ H G@ F+ T' G! ;
    : inc.X     X G@
            T  f1 EXECUTE (87:--f[x,t])
            H  G@ F*      (87:--k=hf[x,t])
            FDUP K G!     \ save k
            X  G@ F+      (87:-- x+k)
            T' G@ T G!
            T  f1 EXECUTE (87:--f[x+k,t+h])
            H  G@ F*
            K  G@ F+ F2/
                          (87:--[k+i[x+k,t+h] ]/2)
            X  G@ F+ X G! ;

    : DONE? TG@ TMAXG@ F>) ; (:--f)
```

```
0 VAR exact                \ cfa
: DISPLAY    exact EXECUTE
    XG@  T  G@ CR F. F. F. ;
\ emit "t x exact

: )RUNGE                   (:cfa-- 87: x0 t0 tf h -- )
        BEGIN DISPLAY
                DONE? NOT
```

Fig. 6-7 Explicit 2nd-order Runge-Kutta solver

As an example of second-order Runge-Kutta in action, let us solve numerically the equation ,

$$\dot{x} = t^2 e^{-} x \tag{1.25}$$

with the initial condition x(t = 0) = 0 , whose exact solution is

$$x(t) = log_e(1 + \frac{1}{3}t^3). \tag{1.26}$$

t x h. t x 1. Big List

Fig. 6-8 Second order Runge-Kutta - results

Thus define

```
: FNB (:[T]-- 87: x--f[x,t])
    FNEGATE FEXP G@ F**2 F*
: EXACT T G@ FDUP FDUP F* F* (87:--t^3)
    3 S->F F/ F=1 F+ FLN ;
```

and say

```
USE( EXACT IS exact ok
USE( FNB % 0. % 0. % 5. % 0.1 )RUNGE
```

The resulting output is shown in fig. 6-8 above.

§§3 An implicit Runge-Kutta formula

A variation on straight Runge-Kutta is a so-called implicit algorithm[13].For example, in the second-order formulae given above, suppose x + k were

[13]See, e.g., A. Ralston, *A First Course in Numerical Analysis* (McGraw-Hill Book Company, New York, 1965) Ch. 5. Implicit methods increase the stability of numerical solution, compared with explicit methods. The formula below is exact for second-order polynomials. The error is of the same order as the explicit formula, but the coeffiient may be smaller.

replaced by x:

$$k = hf(x,t)x = x + \frac{1}{2}(k + hf(x', t+h)) + O(h^3) \tag{1.27}$$

and the resulting (transcendental) equation solved for x by –say– regula falsi. Since we have already written a regula falsi program, we can apply it here to get the algorithm shown diagrammatically in Fig. 6-9 below. We program it[14] as shown in Fig. 6-10 on page 137 below.

)RUNGE INITIALIZE: get h, tf, t0, x0

BEGIN DISPLAY DONE? NOT WHILE k=hf(x,t) SOLVE: x' = x + k/2 +hf(x,t+h)/2 REPEAT

Fig. 6-9 2nd-order implicit Runge-Kutta for dx/dt = f(x,t)

Now we consider the same example as previously:

```
: FNB  (:[T]--  87:x--f[x,t])
     FNEGATE FEXP G@ F"2 F' ;
: EXACT T G@ FDUP FDUP F* F* (87: -- t^3)
     F=3 F/ F=1 F+ FLN ;
```

and say

```
USE( EXACT IS exact ok
USE( FNB % 0. % 0. % 5. % 0.1  )RUNGE ok

FIND )FALSI 0= ?( FLOAD FALSI.FTH)
7REAL*4 SCALARS T T' H X X' TMAX K
0 VAR f1      \ to hold cfa of f(xt)
: INITIALIZE IS f1
         H G! TMAX G! T G! X G! ;

\ These words Increment x & t.
: inc.T T G@ H G@ F+ T' G! ;
: k      X G@
     T f1 EXECUTE    ( 87:--f[x,t])
     H G@ F* K G! ; ( 87:--k=hf(x,t)

: X" ( 87:x'--g[ x   ])
     T f1 EXECUTE    ( 87:--f[x',t+h])
     H G@ F* K G! F+ F2/
       ( 87:--[k+f[x+k,t+h] ]/2)
     X G@ F+ ;
```

[14] **Note**: "**?(**" means "conditionally execute to next right parenthesis".

```
% 3. FCONSTANT F=3
:INTERVAL X G@ FDUP
        K G@ F=3 F* F+ ;

: X' USE( X" INTERVAL % 1.E-6
       )FALSI ;
: inc.X k X    X G! T' G@ T G! ;
: DONE? T G@ TMAX G@ F> ;
      (:--f)
0 VAR exact          \ cfa
: DISPLAY exact EXECUTE
    X G@ T G@ CR F. F. F. ;
\ emit "t x exact"
: )RUNGE (:cfa--87:x0 t0 tf h -- )
    BEGIN DISPLAY
```

Fig. 6-10 Implicit 2nd-order Runge-Kutta program

The resulting output is shown in Fig. 6-11 on page 138.

Clearly the implicit form is more accurate; whether the putative gain in stability justifies solving a transcendental equation is unclear, however.

tXX_exttXX_ex Big List

Fig. 6-11 Second order implicit Runge-Kutta - results

Let us now compare the two algorithms for functions f(x,t) that lead to singular solutions. This time we consider the equation

$$\dot{x} = t^2 e^x \tag{1.28}$$

whose exact solution is

$$x(t) = -log_e(1 - \frac{1}{3}t^3). \tag{1.29}$$

Manifestly, 25 blows up at $t = 3^{\frac{1}{3}} = 1.442...$ We expect the behavior to become apparent in the numerical solution. So we say

```
: FNB FEXP G@ F**2 F*;
    ([t] -- 07: x --i[x,t])

: EXACT F=1 T G@ (87:--t^3)
    FDUP FDUP F* F* F=3 F/ F+ FLN ;
```

and say again

```
USE( EXACT IS exact ok
USE( FNB % 0. %0 % 5. % 0.1 )RUNGE ok
```

The results of doing this with straight-, and then implicit Runge-Kutta are displayed in Fig. 6-12 (p. 140) and 6-13 (p. 141), respectively. We only show the second half of the interval (near the singular point) in either case.

The straight Runge-Kutta algorithm, without the fancy implicit solution for $x(t+h)$, appears more accurate near the singularity, although both methods are acceptably accurate. Does this mean implicit Runge-Kutta is no good? No!

The implicit scheme lost accuracy through roundoff: the arithmetic was insufficiently precise. To take advantage of the methods power, we must increase the precision beyond one part in 10^6. This requires changing all scalars to 64-bit precision (REAL$*$8) rather than 32-bit as we have done here. The generic fetch/-store techniques developed in Chapter 5 and used here, permit this change with a minimum of fuss. We leave this as an exercise.

tXX_EXtXX_EX Big list

Fig. 6-12 *Straight Runge-Kutta for singular case*

Fig. 6-13 *Implicit Range-Kym for singular case*