

# Programming Examples

## Contents

<b>Chapter 6 – Programming Examples</b>	<b>115</b>
§1 Infinite series	116
§§1 Examples of infinite series	116
§§2 Numerical examples of convergent series	118
§§3 The infinite sum program	122
§2 Transcendental equations	127
§§1 Binary search	127
§§2 Regula falsi	128
§3 Ordinary differential equations	131
§§1 Runge-Kutta method	132
§§2 An implicit Runge-Kutta formula	136

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

$$f(x) = \sum_{n=0}^{\infty} c_n x^n \quad (1)$$

where  $x$  is a real number and  $c_n$  is an infinite sequence of coefficients. The extensive mathematical theory<sup>1</sup> of such functions can be summarized as follows: the series of terms only has meaning if – for fixed  $x$  – the sequence of partial sums

$$f_N(x) = \sum_{n=0}^{N-1} c_n x^n \quad (2)$$

has a definite limit.

### §§1 Examples of infinite series

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

#### §§1-1 Divergent examples

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

$$1 + 1 + 1 + 1 + \dots \quad (3)$$

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

---

1. *The Handbook of Mathematical Functions* ed. Milton Abramowitz and Irene Stegun (Dover Publications, 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.

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

$$1 - 1 + 1 - 1 + 1 - \dots \quad (4)$$

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

$$f_N = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{N}. \quad (5)$$

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

### §§1-2 A convergent example

Are there ever cases of infinite series that *do* mean something? Obviously, or there could hardly be a theory of them! An example<sup>2</sup> is  $c_n = 1$ ,  $x = 1/2$ . Here the partial sums

$$\begin{aligned} f_0 &= 1 \\ f_1 &= 1 + \frac{1}{2} \\ f_2 &= 1 + \frac{1}{2} + \frac{1}{4} \\ &\dots\dots\dots \\ \lim_{N \rightarrow \infty} f_N &\equiv \lim_{N \rightarrow \infty} \frac{1 - (1/2)^N}{1 - 1/2} = 2 \end{aligned} \quad (6)$$

*do* have a definite limit, so the series *converges*.

This is the example 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 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!

### §§2 Numerical examples of convergent series

**H**ow do we tell whether a series has converged? As a practical 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 slowly converging series<sup>3</sup>. Consulting the literature in difficult cases is 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.

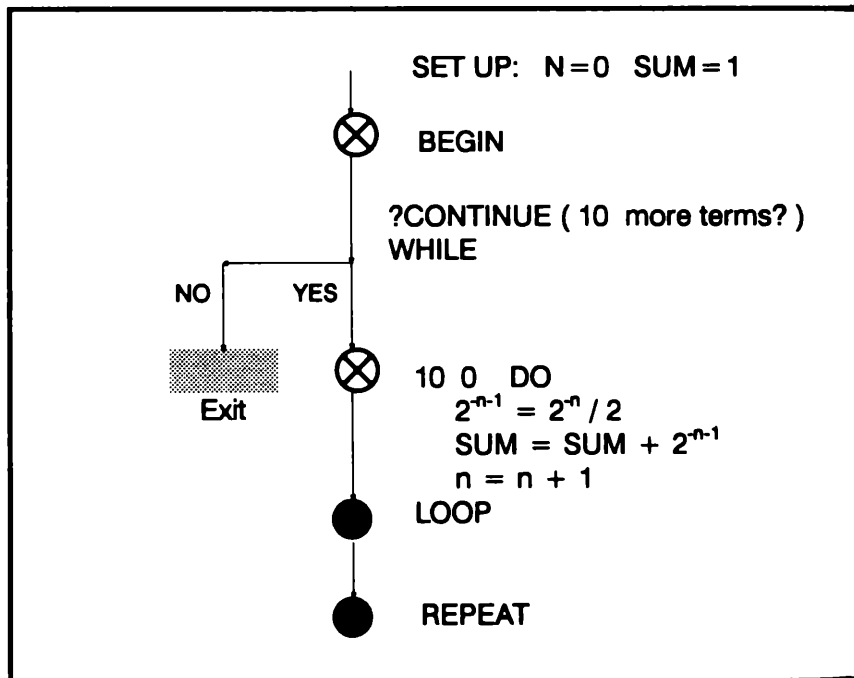


Fig. 6-1 Computing 2 the hard way

3. HMF, Ch. 3 §6 ff.

The corresponding program is

```

15 #PLACES 1 \ 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 SET.UP
  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 = 1.0000000000000000
Another 10 terms? Y
n = 10 sum = 1.99902343750000
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 §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<sup>4</sup>

---

4. HMF, Ch. 4 §4.42.

$$\tan^{-1}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \dots \quad (7)$$

Since a 45° right triangle has equal height and base, the tangent of 45° is 1 (side opposite over side adjacent). That is<sup>5</sup>,

$$\frac{\pi}{4} \equiv \tan^{-1}(1) = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \dots \quad (8)$$

Actually the series Eq. 8 is slowly converging and therefore a poor way to compute  $\pi$ <sup>6</sup>. But anyway, let us proceed. The flow diagram is now that of Fig. 6-2 below.

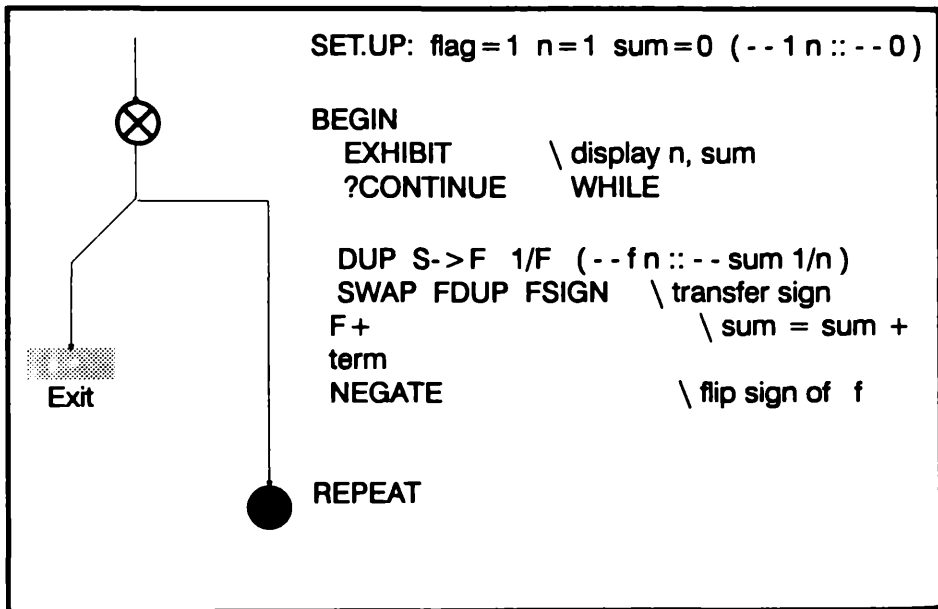


Fig. 6-2 Computing  $\pi/4$  by the infinite series for  $\tan^{-1}(1)$

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  $\pi/8$ .

The corresponding program is

```

15 #PLACES 1 \set F. to 15 digits
: NEXT.TERM (f 2n+1 -- f' 2n+3 :: sum -- sum')
  DUP S->F 1/F SWAP DUP FSIGN F+
  NEGATE SWAP 2+ ;
: SET.UP FINIT F=0 1 1 ;
: EXHIBIT CR DUP
  ." n = " . 2 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<sup>7</sup>:

FLOAD PIBY4 Loading PIBY4 ok  
PI/4

n = 1 4*sum = 0.00000000000000 Another term? Y	n = 25 4*sum = 3.05840276582733 Another term? Y
n = 3 4*sum = 4.00000000000000 Another term? Y	n = 27 4*sum = 3.21840276582733 Another term? Y
n = 5 4*sum = 2.66666666666666 Another term? Y	n = 29 4*sum = 3.07025461777918 Another term? Y
n = 7 4*sum = 3.46666666666666 Another term? Y	n = 31 4*sum = 3.20818565226194 Another term? Y
n = 9 4*sum = 2.89523809523809 Another term? Y	n = 33 4*sum = 3.07915339419742 Another term? Y
n = 11 4*sum = 3.33666253966253 Another term? Y	n = 35 4*sum = 3.20036551540954 Another term? Y
n = 13 4*sum = 2.97604617604617 Another term? Y	n = 37 4*sum = 3.08607980112383 Another term? Y
n = 15 4*sum = 3.28373848373848 Another term? Y	n = 39 4*sum = 3.19418790923194 Another term? Y
n = 17 4*sum = 3.01707181707181 Another term? Y	n = 41 4*sum = 3.09162380666783 Another term? Y
n = 19 4*sum = 3.25236593471887 Another term? Y	n = 43 4*sum = 3.18918478227759 Another term? Y
n = 21 4*sum = 3.04183861892940 Another term? Y	n = 45 4*sum = 3.08616152646364 Another term? Y
n = 23 4*sum = 3.23231580940556 Another term? Y	n = 47 4*sum = 3.18505041535253 Another term? N ok

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<sup>8</sup>. The error (that is, the difference between a partial sum and the limit) is of the order of

---

Note we have set up to display  $\pi$  rather than  $\pi/4$ .

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

the first neglected term and has the same sign. We see that for this case, the error in computing  $\pi$  is of order  $2/n$ , where  $n$  is the number of terms in the sum. To get  $\pi$  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  $\pi$ . A better way uses *e.g.*,

$$\pi = 16\tan^{-1}\left(\frac{1}{5}\right) - 4\tan^{-1}\left(\frac{1}{239}\right),$$

which converges much faster<sup>9</sup>.

### §§3 The infinite sum program

**E**valuating 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 coefficients  $c_n$  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).

---

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



ifstack). For example, to evaluate the exponential *via* the power series

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \quad (9)$$

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 ;
```

### §§3-1 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

```
REAL FUNCTION XINFSUM(X, E, C)
C
C   EVALUATE INFINITE POWER SERIES
C
  EXTERNAL 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
```

where the program to evaluate the exponential would be

```
REAL FUNCTION EXP(X)
EXTERNAL XINFSUM, COEFEXP
REAL C0, XINFSUM
COMMON /CBLK/C0
C0 = 1
EXP = XINFSUM(X, 1.E-7, COEFEXP)
RETURN
END
```

given the coefficient function

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

### §§3–2 A function protocol for FORTH

We would like to extend to FORTH FORTRAN's 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-down 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 it is 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:

```

: SUM.POWERS          ( 87: err -- :: x -- sum)
  E GI                \ 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-1 sum )
    FS>F GOVER G*
      ( -- n 87: -- sum -- :: x x**n )
    DUP adr.c EXECUTE
      ( -- n -- :: x x**n c[n] )
    3 GPICK G* F>FS
      ( -- n -- :: x x**n term sum )
  ENUF? NOT WHILE
    G+
  REPEAT G+ CLEAN.UP ;

```

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

Manifestly, **adr.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<sub>n</sub>** 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**n )
    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:

<pre> \ evaluate exponential by summing series REAL*4 SCALAR E : ENUF? ( :: term -- ) ( -- f )   GABS FS&gt;F EG@ F&gt;; : CLEAN.UP ( n -- :: x x**n sum -- sum )   DROP FS&gt;F   GDROP GDROP F&gt;FS ; \ definition of SUM.POWERS from above BEHEAD" E CLEAN.UP \ hide these def'ns REAL*8 SCALAR c F=1 c G! : COEFF.EXP ( n -- ) ( :: -- c[n] )   DUP 1 &lt;=   IF F=1 c G! F=1 DROP   ELSE c G@ S-&gt;F   F* FDUP c G! 1/F </pre>	<pre> THEN REAL*8 F&gt;FS ; BEHEAD' c \ hide this variable : USE( [COMPILE] ' CFA LITERAL ;   IMMEDIATE \ this means "use address of" \ -- crucial def'n of function lexcon : E**X ( :: x -- e**x ) % 1.E-8   \ error on 87stack   USE( COEFF.EXP   \ adr.c on stack   SUM.POWERS ; </pre>
---	--

## §2 Transcendental equations

A transcendental equation has the form

$$f(x) = 0, \quad (10)$$

where  $f(x)$  is a transcendental function rather than, say, a polynomial or ratio of polynomials<sup>10</sup> (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

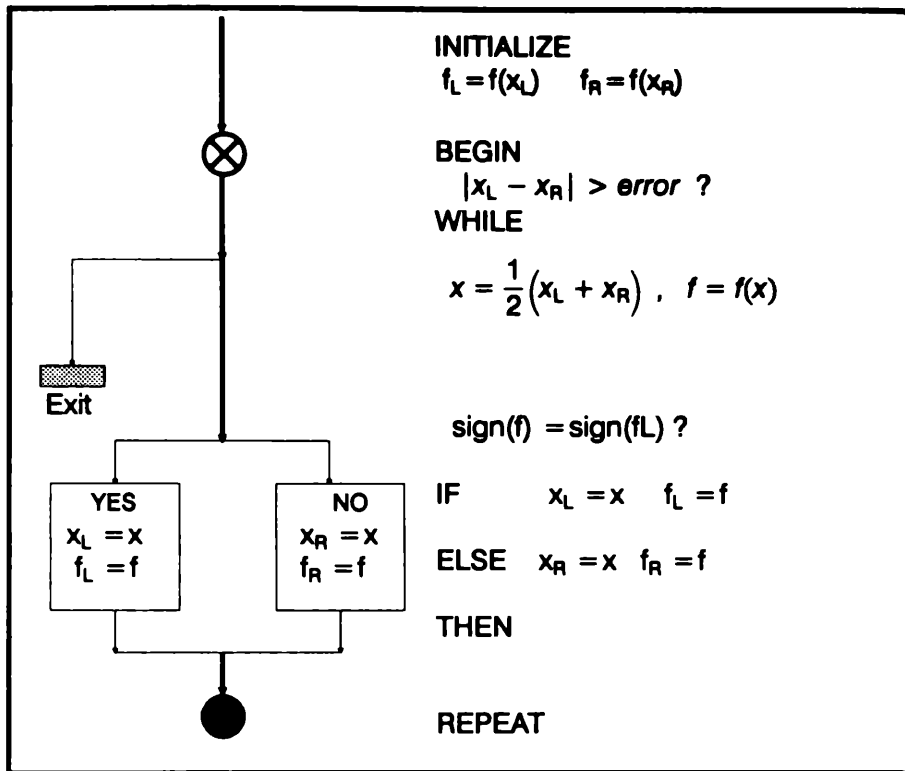
Let us look first at binomial search, since its algorithm is easy to understand. We know some interval,  $x_L \leq x \leq x_R$ , contains a root because  $f(x)$  changes sign when  $x$  goes from  $x_L \rightarrow x_R$ . In pseudocode (and FORTH flow chart) the binomial search algorithm is shown in Fig.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

The method begins with upper and lower bounds on  $x$  that capture the root. Next we look at  $f(x_{AV})$  halfway between  $x_L$  and  $x_R$ . If  $f = f(x_{AV})$  has the same sign as  $f_L = f(x_L)$ , the new left end of the interval becomes  $x_{AV}$ . If the signs are opposite,  $x_{AV}$  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 accuracy 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.

---

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.

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

## §§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  $(x_L, x_R)$ , 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, and the same if  $x'$  is to the right of the root.

As Fig. 6-4 on page 129 shows, the straight line is supposed to 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).

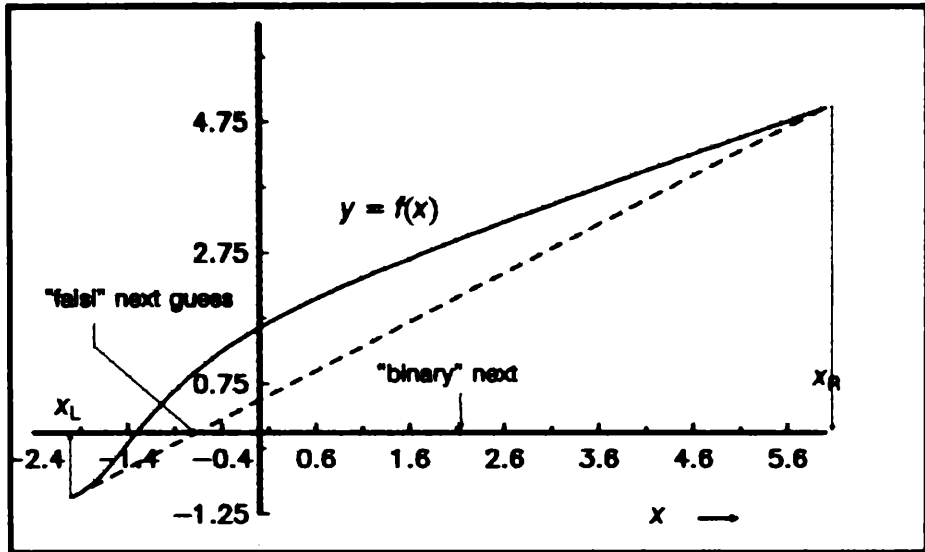


Fig. 6-4 Graphical Illustration of regula falsi

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

$$y = ax + b \quad (11)$$

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  $x$ :

$$x' = \frac{-b}{a} \quad (12)$$

To determine  $a$  and  $b$  we use the two equations

$$f_L = ax_L + b \quad (13)$$

$$f_R = ax_R + b$$

giving

$$a = \frac{1}{2} \left[ f_L + f_R - \frac{f_R - f_L}{x_R - x_L} (x_L + x_R) \right] \quad (14)$$

and thus

$$x' = \frac{f_R x_L - f_L x_R}{f_R - f_L} \quad (15)$$

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

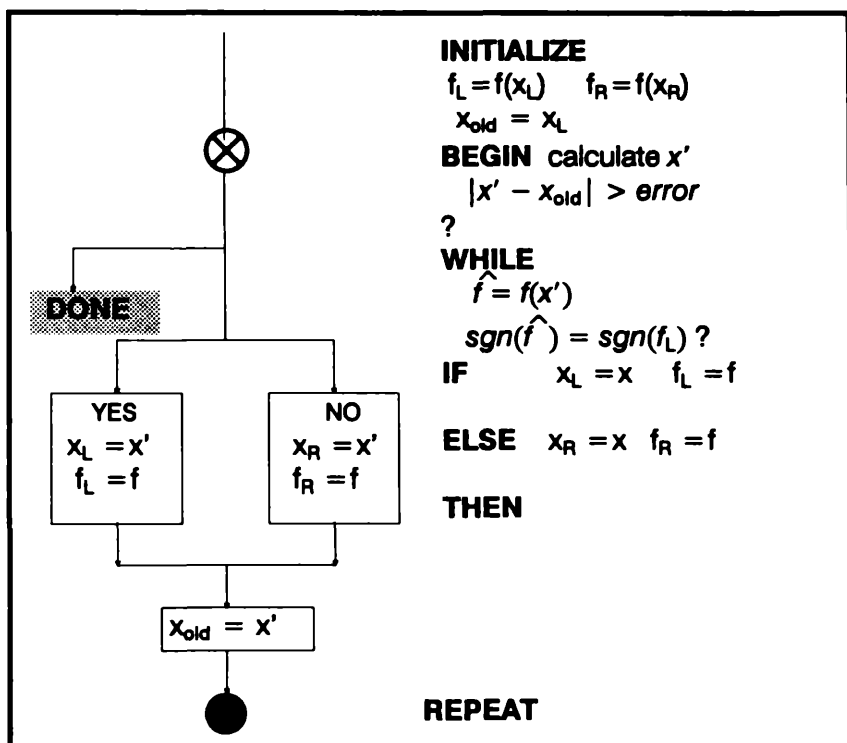


Fig. 6-5 *Regula falsi* algorithm for roots of  $f(x)$

The corresponding program is shown on page 132.

Here is an example of the program in action:

```

FINIT 6 #PLACES I 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.

### §3 Ordinary differential equations

**W**e wish to solve the first-order general differential equation

$$\dot{x} \equiv \frac{dx}{dt} = f(x, t) \quad (16)$$

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 Taylor's series<sup>11</sup>

$$x(t+h) = x(t) + h\dot{x}(t) + \frac{h^2}{2}\ddot{x}(t) + \dots \quad (17)$$

---

11. See, e.g., HMF §3.6.1.

```
\ USE( Fname % a % b % err )FALSEI
( 87: -- root)
\ 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: x y -- -- f)
    F* F0> ;
```

```
: INITIALIZE ( cfa -- 87: a b e --)
    IS f1 \ store 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 yL 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')
    FDUP YL G@
    SAME.SIGN? FOVER
    ( -- f 87: -- x' y' x')
    IF XL G! YL G!
    ELSE XR G! YR G! THEN
    OLDX G! ; ( -- 87: --)
```

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

and keep only the lowest order terms:

$$x(t+h) \approx x(t) + hf(x(t), t). \quad (18)$$

### §§1 Runge-Kutta method

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

12. HMF, §25.5.6.

$$k = hf(x, t)$$

$$x' = x + \frac{1}{2} (k + hf(x+k, t+h)) + O(h^3).$$
(19)

How does this work? Clearly,

$$k + hf(x+k, t+h) = hf(x, t) + hf(x, t) + h^2 \frac{\partial f}{\partial t} + hk \frac{\partial f}{\partial x}$$

$$\equiv 2h\dot{x}(t) + h^2\ddot{x}(t) + O(h^3)$$
(20)

Substituting 19 in 20 we now find

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

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.

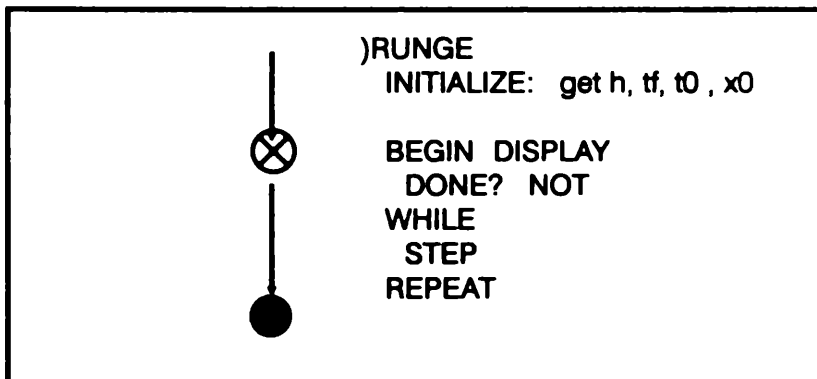


Fig. 6-6 2nd-order Runge-Kutta for  $dx/dt = f(x, t)$

<pre> \ STRAIGHT 2ND ORDER RUNGE-KUTTA \ SOLUTION OF FIRST-ORDER DIFEQ \ dx/dt = f(x,t)  \ See Abramowitz &amp; Stegun, HMF §25.5.6  \ Usage: \ 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! ; </pre>	<pre> \ These words increment x &amp; 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+hf(x+k,t+h)]/2 )   X G@ F+ X G! ;  : DONE? T G@ TMAX G@ F&gt; ; (: -- 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   DONE? NOT </pre>
---	---

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

$$\dot{x} = t^2 e^{-x} \quad (21)$$

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

$$x(t) = \log_e \left( 1 + \frac{1}{3} t^3 \right). \quad (22)$$

t	x	x <sub>ex</sub>	t	x	x <sub>ex</sub>
.0000	.0000	.0000	2.5000	1.9254	1.9256
.1000	.00060	.00033	2.6000	2.0228	2.0230
.2000	.00300	.00286	2.7000	2.1181	2.1183
.3000	.00946	.00886	2.8000	2.2113	2.2115
.4000	.02177	.02111	2.9000	2.3023	2.3025
.5000	.04164	.04082	3.0000	2.3912	2.3915
.6000	.07049	.06953	3.1000	2.4781	2.4784
.7000	.10934	.10825	3.2000	2.5630	2.5633
.8000	.15875	.15757	3.3000	2.6459	2.6462
.9000	.21877	.21752	3.4000	2.7270	2.7273
1.0000	.28896	.28768	3.5000	2.8061	2.8065
1.1000	.36846	.36718	3.6000	2.8836	2.8839
1.2000	.45612	.45489	3.7000	2.9592	2.9596
1.3000	.55063	.54946	3.8000	3.0333	3.0336
1.4000	.65061	.64954	3.9000	3.1057	3.1060
1.5000	.75473	.75377	4.0000	3.1768	3.1769
1.6000	.86175	.86091	4.1000	3.2460	3.2463
1.7000	.97061	.96989	4.2000	3.3139	3.3142
1.8000	1.0803	1.0797	4.3000	3.3804	3.3808
1.9000	1.1902	1.1897	4.4000	3.4456	3.4467
2.0000	1.2996	1.2992	4.5000	3.5095	3.5099
2.1000	1.4080	1.4078	4.6000	3.5722	3.5725
2.2000	1.5151	1.5149	4.7000	3.6336	3.6339
2.2999	1.6206	1.6205	4.8000	3.6939	3.6942
2.3999	1.7242	1.7241	4.9000	3.7531	3.7534
2.4999	1.8258	1.8258	5.0000	3.8111	3.8114

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.

## §§2 An Implicit Runge-Kutta formula

A variation on straight Runge-Kutta is a so-called *implicit* algorithm<sup>13</sup>. For example, in the second-order formulae given above, suppose  $x + k$  were replaced by  $x'$ :

$$k = hf(x, t) \quad (23)$$

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

and the resulting (transcendental) equation solved for  $x'$  by *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<sup>14</sup> as shown in Fig. 6-10 on page 137 below.

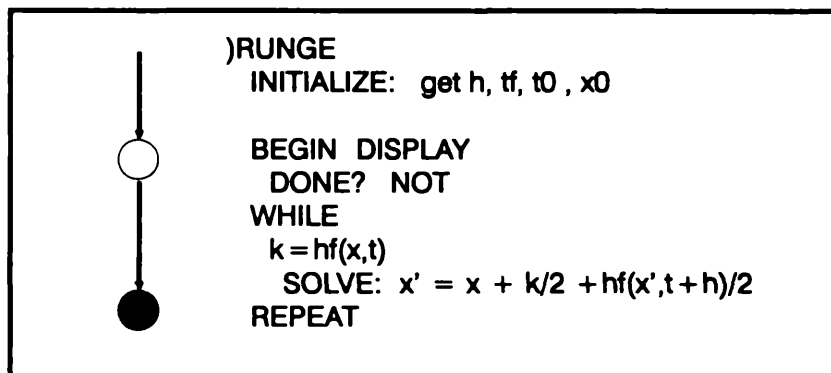


Fig. 6-9 2nd-order *implicit* Runge-Kutta for  $dx/dt = f(x,t)$

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 coefficient may be smaller.
14. **Note:** “ ? ( ” means “conditionally execute to next right parenthesis”.

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
```

<pre>FIND )FALSI 0= ?( FLOAD FALSI.FTH ) 7 REAL*4 SCALARS T T' H X X' TMAX K 0 VAR f1 \to hold cfa of f(x,t) : INITIALIZE IS f1   H G! TMAX G! T G! X G! ;  \ These words increment x &amp; t : inc.T T G@ HG@ 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)</pre>	<pre>X G@ F+ ; %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&gt; ;   ( : -- 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 t h --)   BEGIN DISPLAY</pre>
--	--

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.

t	x	x <sub>ex</sub>	t	x	x <sub>ex</sub>
0.0000	.00000	.00000	2.5999	1.9253	1.9255
.10000	.00050	.00033	2.6999	2.0228	2.0230
.20000	.00299	.00266	2.7999	2.1181	2.1183
.30000	.00945	.00896	2.8999	2.2113	2.2115
.40000	.02173	.02111	2.9999	2.3024	2.3025
.50000	.04155	.04082	3.0999	2.3914	2.3915
.60000	.07032	.06953	3.1999	2.4783	2.4784
.70000	.10906	.10825	3.2999	2.5632	2.5633
.80000	.15834	.15757	3.3999	2.6461	2.6462
.90000	.21822	.21752	3.4999	2.7272	2.7273
1.0000	.28825	.28768	3.5999	2.8064	2.8065
1.1000	.36762	.36718	3.6999	2.8838	2.8839
1.2000	.45518	.45489	3.7999	2.9595	2.9596
1.3000	.54962	.54946	3.8999	3.0336	3.0336
1.4000	.64958	.64954	3.9999	3.1060	3.1060
1.5000	.75370	.75377	4.0999	3.1769	3.1769
1.6000	.86077	.86091	4.1999	3.2463	3.2463
1.7000	.96969	.96989	4.2999	3.3142	3.3142
1.8000	1.0795	1.0797	4.3999	3.3808	3.3808
1.9000	1.1895	1.1897	4.4999	3.4460	3.4460
2.0000	1.2990	1.2992	4.5999	3.5099	3.5099
2.1000	1.4075	1.4078	4.6999	3.5725	3.5725
2.2000	1.5147	1.5149	4.7999	3.6340	3.6339
2.2999	1.6202	1.6205	4.8999	3.6942	3.6942
2.3999	1.7239	1.7241	4.9999	3.7534	3.7534
2.4999	1.8256	1.8258	5.0999	3.8114	3.8114

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

$$\dot{x} = t^2 e^x \quad (24)$$

whose exact solution is

$$x(t) = -\log_e \left( 1 - \frac{1}{3} t^3 \right). \quad (25)$$

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



```

: FNB FEXP G@ F**2 F* ;
  ( [t] -- 87: x -- f[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 method's 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.

t	x	Xex	t	x	Xex
.71999	.13287	.13286	1.0899	.56508	.56506
.72999	.13889	.13888	1.0999	.58640	.58638
.73999	.14512	.14511	1.1099	.60860	.60857
.74999	.15156	.15154	1.1199	.63171	.63169
.75999	.15821	.15820	1.1299	.65580	.65578
.76999	.16509	.16508	1.1399	.68093	.68091
.77999	.17220	.17219	1.1499	.70718	.70715
.78999	.17955	.17954	1.1599	.73461	.73458
.79999	.18714	.18713	1.1699	.76331	.76329
.80999	.19499	.19497	1.1799	.79337	.79335
.81999	.20309	.20308	1.1899	.82491	.82489
.82999	.21147	.21145	1.1999	.85803	.85801
.83999	.22012	.22010	1.2099	.89287	.89286
.84999	.22906	.22904	1.2199	.92959	.92958
.85999	.23829	.23828	1.2299	.96834	.96834
.86999	.24783	.24782	1.2399	1.0093	1.0093
.87999	.25769	.25767	1.2499	1.0527	1.0527
.88999	.26788	.26786	1.2599	1.0989	1.0989
.89999	.27841	.27839	1.2699	1.1481	1.1482
.90999	.28928	.28926	1.2799	1.2007	1.2008
.91999	.30053	.30051	1.2899	1.2571	1.2572
.92999	.31215	.31213	1.2999	1.3179	1.3180
.93999	.32417	.32415	1.3099	1.3836	1.3837
.94999	.33660	.33657	1.3199	1.4550	1.4552
.95999	.34945	.34943	1.3299	1.5332	1.5334
.96999	.36274	.36272	1.3399	1.6193	1.6196
.97999	.37650	.37648	1.3499	1.7151	1.7154
.98999	.39074	.39072	1.3599	1.8226	1.8231
.99999	.40548	.40546	1.3699	1.9449	1.9457
1.0099	.42075	.42073	1.3799	2.0865	2.0876
1.0199	.43656	.43654	1.3899	2.2539	2.2557
1.0299	.45295	.45293	1.3999	2.4581	2.4611
1.0399	.46995	.46993	1.4099	2.7187	2.7242
1.0499	.48757	.48755	1.4199	3.0760	3.0884
1.0599	.50586	.50584	1.4299	3.6376	3.6782
1.0699	.52485	.52483	1.4399	4.8831	5.3657
1.0799	.54458	.54456			

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

## Chapter 6 – Programming Examples

t	x	xex	t	x	xex
.71999	.13288	.13288	1.0799	.54464	.54466
.72999	.13890	.13888	1.0899	.56515	.56508
.73999	.14513	.14511	1.0999	.58648	.58638
.74999	.15156	.15154	1.1099	.60868	.60857
.75999	.15822	.15820	1.1199	.63180	.63169
.76999	.16510	.16508	1.1299	.65590	.65578
.77999	.17221	.17219	1.1399	.68104	.68091
.78999	.17956	.17954	1.1499	.70729	.70715
.79999	.18715	.18713	1.1599	.73473	.73458
.80999	.19500	.19497	1.1699	.76345	.76329
.81999	.20310	.20308	1.1799	.79353	.79336
.82999	.21148	.21145	1.1899	.82508	.82489
.83999	.22013	.22010	1.1999	.85822	.85801
.84999	.22907	.22904	1.2099	.89309	.89286
.85999	.23831	.23828	1.2199	.92983	.92958
.86999	.24785	.24782	1.2299	.96861	.96834
.87999	.25771	.25767	1.2399	1.0096	1.0093
.88999	.26789	.26786	1.2499	1.0531	1.0527
.89999	.27842	.27839	1.2599	1.0993	1.0989
.90999	.28930	.28926	1.2699	1.1486	1.1482
.91999	.30055	.30051	1.2799	1.2013	1.2008
.92999	.31217	.31213	1.2899	1.2578	1.2572
.93999	.32419	.32415	1.2999	1.3186	1.3180
.94999	.33662	.33657	1.3099	1.3845	1.3837
.95999	.34947	.34943	1.3199	1.4561	1.4552
.96999	.36277	.36272	1.3299	1.5345	1.5334
.97999	.37653	.37648	1.3399	1.6209	1.6196
.98999	.39077	.39072	1.3499	1.7171	1.7154
.99999	.40551	.40546	1.3599	1.8252	1.8231
1.0099	.42078	.42073	1.3699	1.9485	1.9457
1.0199	.43660	.43654	1.3799	2.0914	2.0876
1.0299	.45300	.45293	1.3899	2.2612	2.2557
1.0399	.46999	.46993	1.3999	2.4698	2.4611
1.0499	.48762	.48755	1.4099	2.7395	2.7242
1.0599	.50582	.50584	1.4199	3.1222	3.0884
1.0699	.52491	.52483	1.4299	3.6149	3.6782

Fig. 6-13 *Implicit Runge-Kutta for singular case*

