Copyright © 1986, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

NONLINEAR ELECTRONICS (NOEL) PACKAGE 9RKF: A SINGLE-STEP, VARIABLE STEP-SIZE INTEGRATION ROUTINE FOR NON-STIFF ODES

by

Thomas S. Parker, Greg M. Bernstein, and L. O. Chua

Memorandum No. UCB/ERL M86/28

28 March 1986

NONLINEAR ELECTRONICS (NOEL) PACKAGE 9 RKF: A SINGLE-STEP, VARIABLE STEP-SIZE INTEGRATION ROUTINE FOR NON-STIFF ODES

by

Thomas S. Parker, Greg M. Bernstein, and L. O. Chua

Memorandum No. UCB/ERL M86/28

28 March 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

NONLINEAR ELECTRONICS (NOEL) PACKAGE 9 RKF: A SINGLE-STEP, VARIABLE STEP-SIZE INTEGRATION ROUTINE FOR NON-STIFF ODES

by

Thomas S. Parker, Greg M. Bernstein, and L. O. Chua

Memorandum No. UCB/ERL M86/28

28 March 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

RKF: A Single-Step, Variable Step-Size Integration Routine for Non-Stiff ODEs

Thomas S. Parker, Greg M. Bernstein and L. O. Chua

ABSTRACT

RKF is a set of C functions which implements the Runge-Kutta-Fehlberg (4,5) formulas for integration of ordinary differential equations. Features include variable step-size, reliable error control, stiffness detection, and a control structure that is designed for interactive programs.

The Algorithm

The basic method used in RKF is the Runge-Kutta Fehlberg (4, 5) formula. However, much more is involved in turning a basic integration formula into a computer program than just coding the formula. In [1] and [2] Shampine and Watts go into great detail describing what constitutes a good Runge-Kutta algorithm and they explain in detail the design decisions and workings of RKF45, the original FORTRAN code on which RKF is based.

Why use RKF? The fourth order Runge-Kutta integration formulas documented in every basic differential equation book leave out one essential ingredient for practical implementation: efficient error estimation. The typical method for error estimation using 4th-order Runge-Kutta is a process of halving step-sizes which leads to inefficient code. The Fehlberg (4, 5) method estimates the error by using a combination of 4th and 5th order formulas. This error estimation can be achieved using just six function evaluations and it gives a result accurate to the 5th order. The halving technique can use as many as eight function evaluations to get a result accurate to only the 4th order. This feature along with stiffness detection (see below) makes RKF a useful integration package.

Single-step Versus Multi-step Solvers

The differential equation to be integrated has the following form:

$$y' = f(y, t), \quad y(t_0) = y_0$$

When the above equation is solved numerically, one obtains the approximation $\bar{y}(t_0)$, $\bar{y}(t_1)$, ... $\bar{y}(t_n)$ to the true trajectory at the discrete time points t_0, t_1, ... t_n. To obtain the solution at time t_n a single-step solver makes use of the solution at the previous time point t_{n-1}, along with evaluations of f and possibly its derivative. A multi-step solver makes use of the solution at a number of previous time points, the function f evaluated at various time points, and possibly the derivative of f.

One of the advantages of a multi-step method is that it can be designed to use relatively few function evaluations per step. Thus, if function evaluations are costly, a multi-step method can be more economical. However, the increase in computational overhead due to keeping track of the solution at a number of time points is the main disadvantage of multi-step methods. So, if function evaluations are cheap, a single step method can be quicker.

Research sponsored by the Office of Naval REsearch under Contract N00014-76-C-0572, and by the National Science Foundation under Grant ECS-8313278.

Stiffness in Single-Step Solvers

Stiffness can make a differential equation unsolvable numerically; however, programs like RKF with good error control will not give inaccurate answers but will become inefficient, sometimes to the point of being worthless for solving a problem. For a readable account of what stiffness means in differential equations and some all around good advice concerning solving differential equations numerically, see [3] and [4].

How does one know whether the differential equation is stiff or just difficult to integrate accurately? RKF contains an algorithm that detects whether the differential equation is looking stiff with respect to the Fehlberg (4, 5) formulas. Hence, if the solver is becoming inefficient, check what the stiffness detection algorithm says. Note that if the integration seems to be proceeding at an acceptable pace and stiffness is detected, don't panic! There will not be a loss of accuracy and the stiffness warning can be ignored. However, if stiffness is detected and is causing the integration to become intractable, it is necessary to switch to another integration package that is designed for stiff equations.

The theory behind the stiffness detection algorithm in RKF is well documented in [5] and [6].

Interactive Programs

With the abundance of personal computers and work-stations, ease of use is becoming a priority in software design. Most integration packages are written in FORTRAN by scientists. The combination of these two factors does not always lead to good software design. RKF was specifically created to allow interactive programs to be written easily and quickly.

The control structure of RKF is clean. Many integration packages have one function call that does everything. It sets the initial condition, the error tolerances, the integration mode, and returns the integration status. To make a simple call to move one step in the integration, the programmer must supply ten or fifteen parameters. The approach in RKF is to group parameters together and have many function calls. For example, there is a function that sets the initial condition, one that sets the error tolerances, one that sets the integration mode, one that returns the integration status, and one that performs the integration. This modularity makes program writing easier and less error prone.

RKF is designed to be used in interactive programs. There is a facility for suspending the integration, usually upon receipt of input from the console. The integration can be resumed later with no loss in computation.

Another unusual feature of RKF is that it can integrate two or more equations at the same time. The ODE interface of RKF is based on the standard file interface of C. ODEs are opened and closed and just as more than one file can be accessed from a single C program, more than one ODE can be integrated by a single RKF program.

References

1. L. F. Shampine and H. A. Watts, "The Art of Writing a Runge-Kutta Code, Part I," in *Mathematical Software III*, ed. J. R. Rice, pp. 257-275, Academic, New York, 1977.
2. Lawrence F. Shampine and Herman A. Watts, "The Art of Writing a Runge-Kutta Code. II.," *Applied Mathematics and Computation*, pp. 93-121, 1979.
3. L. F. Shampine, "What Everyone Solving Differential Equations Numerically Should Know," in *Computational Techniques for Ordinary Differential Equations*, ed. D. K. Sayers, pp. 1-17, Academic Press, New York, 1980.
4. L. F. Shampine and C. W. Gear, "A User's View of Solving Stiff Ordinary Differential Equations," *SIAM Review*, vol. 21, no. 1, January 1979.
5. L. F. Shampine, "Stiffness and Nonstiff Differential Equation Solvers, II: Detecting Stiffness With Runge-Kutta Methods," *ACM Transactions on Mathematical Software*, vol. 3, no. 1, March 1977.
6. L. F. Shampine and K. L. Hiebert, "Detecting Stiffness with the Fehlberg (4,5) Formulas," *Comp. & Maths. with Appls.*, vol. 3, pp. 41-46, 1977.

Appendix I: Manual Pages for RKF

NAME

rkf_intro – introduction to RKF integration package

SYNOPSIS

# include < local/ rkf.h>

DESCRIPTION

rkf integrates a system of differential equations in state equation form. It is a C implementation of the RKF45 subroutine originally written in FORTRAN by H. A. Watts and L. F. Shampine of Sandia Laboratories Albuquerque, New Mexico. *rkf* is specially designed for use in interactive programs.

rkf uses the Runge-Kutta-Fehlberg (4,5) method of solving differential equations and for estimating the local truncation error. The Fehlberg (4,5) method is a 5th-order Runge-Kutta method that also calculates a 4th-order solution without any additional function evaluations. From these two different solutions, an estimate of the local truncation error can be obtained. This estimate of the local truncation error (i.e. the difference between the approximate solution and the true solution, both starting from the same initial point, over one step) is the basis of the variable stepsize algorithm.

Much as file functions in C operate on FILE pointers, the functions in *rkf* operate on RKF pointers. Just as each file must be opened to associate it with a FILE pointer, each differential equation must be opened to associate it with an RKF pointer.

Like files, more than one differential equation may be open at the same time. Furthermore, the same differential equation may be simultaneously associated with more than one RKF pointer (i.e. be open more than once at the same time). This allows the user great flexibility in writing programs using this integration package.

A differential equation is opened by a call to *rkf_open(3ML)*. Once the differential equation is opened, *rkf_error(3ML)* and *rkf_init(3ML)* should be called to set up the error tolerances and initial condition, respectively. The actual integration is performed by *rkf(3ML)*. When the integration is finished, the differential equation should be closed by a call to *rkf_close(3ML)*.

DIAGNOSTICS

rkf utilizes an error code denoting the status of the integration. The error codes are defined as macros in < local/ rkf.h> as follows:

0 OK
: Everything is okay.

1 NEG_ERROR
: *rel_err* is not positive or *abs_err* is negative.

2 REL_LIM
: *rel_err* is below its predefined minimum, that is *rel_err* < 2.0**m_eps* + *re_min* where *m_eps* is a machine-dependent constant and *re_min* = 1.0E-12 (default). If this error occurs, *rkf_error(3ML)* will set *rel_err* to its smallest allowed value so you don't need to recall *rkf_error(3ML)*.

3 FUNC_LIM
: *rkf* is doing too much work, that is the number of function evaluations exceeded *max_nfe* = 3000 (default) (approximately 500 steps). If you wish to continue the integration just recall *rkf(3ML)*.

4 IO_LIM
: *rkf* is becoming inefficient because of too much output, that is, the output is restricting the natural stepsize. If you wish to continue the integration just recall *rkf(3ML)*.

5 ZERO_ABS
: Vanishing solution needs a non-zero *abs_err* to continue the integration. A

pure relative error test is impossible (and infeasible) due to a zero solution. Before you resume integration, call *rkf_error(3ML)* with a non-zero value for *abs_err*.

6 STEP_LIM
: Requested accuracy cannot be achieved with the smallest allowed stepsize. This flag probably indicates that the integration cannot go on, that is, the solution is singular or has finite escape time. If you want to continue, reset either *abs_err* or *rel_err* or both to a larger value by a call to *rkf_error(3ML)*.

7 STIFF_EQN
: The differential equation is stiff causing *rkf* to do too much work, that is, the number of function evaluations has exceeded *max_nfe* = 3000 (default). If you wish to continue the integration just recall *rkf(3ML)*.

8 FUNC_ERR
: The state equation could not be evaluated at a point and time where *rkf* needed it, that is, the function *f()* returned FALSE.

9 STDIN_RDY
: Characters are ready to read at stdin. See *stdin_chk(3ML)*.

10 ERR_SET
: Error tolerances have not been properly set. Call *rkf_error(3ML)* to set the error tolerances.

11 INIT_SET
: The initial condition has not been properly set. Call *rkf_init(3ML)* to set the initial condition.

EXAMPLE

Integrate y' = -y. Only minimal error checking is done in this example.

```
/*C program to integrate y'= -y, y(0) = 1.0 and get
solution at t = 1.0.*/

# include < stdio.h>
# include < local/ rkf.h>

# define TRUE 1
# define SYS_DIM       1
# define N_INIT        1

main()

{
        int neqn = 1;
        double y[SYS_DIM];
        double t = 0.0;
        double tout = 1.0;
        double rel_err = 1.0E-7;
        double abs_err = 0.0;
        int err_flag;
        char *rkf_mess();
        int test_f();
        RKF *dp, *rkf_open();

        y[0] = 1.0;
        if ((dp = rkf_open(test_f, neqn)) == NULL)
        {
                fprintf(stderr, "Cannot open RKF\n");
                exit(1);
        }
        rkf_error(rel_err, abs_err);
        rkf_init(t, y);
        err_flag = rkf(tout, &t, y, dp);
        printf(" t = %e ", t);
        printf(" y[0] = %e ", y[0]);
        printf(" status: %s\n", rkf_mess(err_flag));
        rkf_close(dp);
}

test_f(z, x, t)
double z[], x[], t;

{
        z[0] = -x[0];
        return (TRUE);
}
```

LIBRARY

The functions reside in /usr/local/lib/librkf.a and may be loaded by specifying the -lrkf flag to *cc(1)* or *ld(1)*.

NOTE

The file < local/rkf.h> must be # included in any source file referencing functions in *rkf*.

FILES

< local/rkf.h>
/usr/local/lib/librkf.a

SEE ALSO

rkf(3ML), rkf_open(3ML), rkf_error(3ML), rkf_init(3ML), rkf_mess(3ML), rkf_cntrl(3ML), rkf_read(3ML)

BUGS

rkf almost invariably returns STIFF_EQN after 3000 (*max_nfe*) function calls.

We had problems in the PC-DOS version because it kept returning ZERO_ABS even when *abs_err* was positive.

AUTHORS

Greg Bernstein
Tom Parker

NAME

rkf – perform RKF integration

SYNOPSIS

```
#include < local/rkf.h>

int rkf(t_out, t, y, p)
double t_out, *t, y[];
RKF *p;
```

DESCRIPTION

rkf performs the integration of the equation associated with **p*. *t_out* is the desired output time. Upon return, **t* is the actual output time and *y* contains the output vector evaluated at **t*.

DIAGNOSTICS

rkf returns the *rkf* error code (see *rkf_intro(3ML)*).

LIBRARY

The function resides in **/usr/local/lib/librkf.a** and may be loaded by specifying the **-lrkf** flag to *cc(1)* or *ld(1)*.

NOTES

rkf_open(3ML), *rkf_error(3ML)* and *rkf_init(3ML)* must be called before *rkf.*

**t* will not be equal to *t_out* only when an error occurs (*rkf* does not return OK).

y should point to a buffer large enough to hold the result.

FILES

< local.rkf.h>
/usr/local/lib/librkf.a

SEE ALSO

rkf_intro(3ML), rkf_open(3ML), rkf_init(3ML), rkf_error(3ML), rkf_mess(3ML), rkf_cntrl(3ML), rkf_read(3ML)

AUTHORS

Greg Bernstein
Tom Parker

NAME

rkf_cntrl – miscellaneous functions which set RKF parameters

SYNOPSIS

```
#include <stdio.h>
#include <local/rkf.h>

rkf_mode(mode, p)
int mode;
RKF *p;

rkf_stdin(fp, p)
int (*fp)();
RKF *p;

rkf_nfe(max_nfe, p)
int max_nfe;
RKF *p;

rkf_kop(max_kop, p)
int max_kop;
RKF *p;

rkf_seq_len(seq_len, p)
int seq_len;
RKF *p;

rkf_re_min(re_min, p)
double re_min;
RKF *p;

rkf_copy(dp, sp)
RKF *dp, *sp;
```

DESCRIPTION

rkf_mode sets the current integration mode of the integration associated with **p* to *mode*. *mode* is either ENDPT or SING_STEP. In END_PT mode, *rkf(3ML)* returns at the final time *t_out*. In SING_STEP mode, *rkf(3ML)* returns after every step of the integration until it hits *t_out*. In neither case is *t_out* ever passed. The default mode is END_PT.

rkf_stdin is used to set whether the STDIN_RDY error can occur. If *fp* is NULL, no checking is done; otherwise, *fp* is a pointer to a function which returns TRUE (= = 1) if input is ready and FALSE (= = 0) otherwise. The default is no checking done.

rkf_nfe sets the maximum allowed number of function evaluations before the FUNC_LIM error is returned by *rkf(3ML)*. The default is 3000.

rkf_kop sets the maximum allowed number of output points which can impair the efficiency of *rkf(3ML)* before the IO_LIM error is returned. The default is 100.

rkf_seq_len is used to set the maximum allowed sequence length (used by the stiffness test). The default is 50.

rkf_re_min is used to set the minimum allowed error tolerance. This effects REL_LIM error detection. The default is 1.0e-12.

rkf_copy copies the RKF structure **sp* to **dp*. Both structures need to have been created by calls to *rkf_open(3ML)*.

LIBRARY

These functions reside in **/usr/local/lib/librkf.a** and may be loaded by specifying the **-lrkf** flag to *cc(1)* or *ld(1)*.

NOTE

< stdio.h> should be # included to satisfy references to NULL.

BUGS

Too many functions on one manual page.

rkf_copy does not really fit in here, but it is such a seldomly used function, it does not really deserve a manual page of its own.

FILES

< local/rkf.h>
/usr/local/lib/librkf.a

SEE ALSO

rkf_intro(3ML), rkf(3ML), rkf_open(3ML), rkf_error(3ML), rkf_init(3ML), rkf_mess(3ML), rkf_read(3ML)

AUTHORS

Greg Bernstein
Tom Parker

NAME

rkf_error – set error tolerances for RKF integration

SYNOPSIS

```
#include <local/rkf.h>

int rkf_error(rel_err, abs_err, p)
double rel_err, abs_err;
RKF *p;
```

DESCRIPTION

rkf_error sets the relative and absolute error tolerances associated with **p* to *rel_err* and *abs_err*, respectively.

Accuracy

The local truncation error is controlled on an error-per-step basis, that is

|loc. trunc. err. |< = *abs_err* + *rel_err**|x|

where is the previous solution point.

Intuitively, it might seem that an error-per-unit-step criterion would be better for controlling the global error. However, the literature does not seem to indicate this and, for differential equations with discontinuities in their derivatives (e.g. piecewise linear models), an error-per-unit-step criterion can cause an algorithm to blow up.

DIAGNOSTICS

rkf_error returns the *rkf* error code (see *rkf_intro(3ML)*). The possible return values are OK, NEG_ERROR, REL_LIM, ZERO_ABS and STEP_LIM.

If REL_LIM is returned, *rkf_error* will increase the relative error tolerance to the smallest allowed amount. Use *rkf_read(3ML)* to get the new value for the relative error tolerance.

ZERO_ABS (STEP_LIM) will only be returned if ZERO_ABS (STEP_LIM) is the error condition when *rkf_error* is called and then only if the new values of *rel_err* and *abs_err* do not clear the error condition.

LIBRARY

The function resides in /usr/local/lib/librkf.a and may be loaded by specifying the **-lrkf** flag to *cc(1)* or *ld(1)*.

FILES

<local/rkf.h>
/usr/local/lib/librkf.a

SEE ALSO

rkf_intro(3ML), rkf(3ML), rkf_open(3ML), rkf_init(3ML), rkf_mess(3ML), rkf_cntrl(3ML), rkf_read(3ML)

AUTHORS

Greg Bernstein
Tom Parker

NAME

rkf_init – set initial condition for RKF integration

SYNOPSIS

```
#include <local/rkf.h>

rkf_init(t, x, p)
double t, x[];
RKF *p;
```

DESCRIPTION

rkf_init sets the initial time *t* and the initial state *x* for the state equation associated with **p*.

LIBRARY

The function resides in /usr/local/lib/librkf.a and may be loaded by specifying the **-lrkf** flag to *cc(1)* or *ld(1)*.

FILES

< local/rkf.h>
/usr/local/lib/librkf.a

SEE ALSO

rkf_intro(3ML), rkf(3ML), rkf_open(3ML), rkf_error(3ML), rkf_mess(3ML), rkf_cntrl(3ML), rkf_read(3ML)

AUTHORS

Greg Bernstein
Tom Parker

NAME

rkf_mess – get text version of RKF error status

SYNOPSIS

# include < local/rkf.h>

char *rkf_mess(err_flag)
int err_flag;

DESCRIPTION

rkf_mess returns a pointer to a string containing a text version of the *rkf* error code represented by *err_flag*.

EXAMPLE

```
        char *rkf_mess();

        if (err_flag != OK)
                printf("RKF error: %s\n", rkf_mess(err_flag));
```

LIBRARY

The function resides in /usr/local/lib/librkf.a and may be loaded by specifying the **-lrkf** flag to *cc(1)* or *ld(1)*.

NOTES

Do not alter the contents of the returned string.

rkf_mess must be declared as returning a character pointer for it to work properly.

FILES

< local/rkf.h>
/usr/local/lib/librkf.a

SEE ALSO

rkf_intro(3ML), rkf(3ML), rkf_open(3ML), rkf_init(3ML), rkf_error(3ML), rkf_cntrl(3ML), rkf_read(3ML)

AUTHORS

Greg Bernstein
Tom Parker

NAME

rkf_open, rkf_close – open and close equation for integration

SYNOPSIS

```
# include < stdio.h>
# include < local/rkf.h>

RKF *rkf_open(f, n)
int (*f)(), n;

rkf_close(p)
RKF *p;
```

DESCRIPTION

rkf_open returns a pointer to an RKF structure associated with the *n*-dimensional state equation specified by *f*.

rkf_close disposes of the RKF structure **p* obtained by a previous call to *rkf_open*.

DIAGNOSTICS

rkf_open returns NULL if there is not enough memory to create an RKF structure or if *n* is not positive.

LIBRARY

Both functions reside in **/usr/local/lib/librkf.a** and may be loaded by specifying the **-lrkf** flag to *cc(1)* or *ld(1)*.

NOTES

**f* must be in the form

```
double f(z, x, t)
double z[], x[], t;

{
}
```

and should set *z* to the value of the state equation evaulated at *x* and *t*.

rkf_open must be declared as returning an RKF pointer for it to work properly.

< stdio.h> must be # included to satisfy references to NULL.

FILES

< local/rkf.h>
/usr/local/lib/librkf.a

SEE ALSO

makef(1L)
rkf_intro(3ML), rkf(3ML), rkf_error(3ML), rkf_init(3ML), rkf_mess(3ML), rkf_cntrl(3ML), rkf_read(3ML)

AUTHORS

Greg Bernstein
Tom Parker

NAME

rkf_read – fetches various RKF parameters

SYNOPSIS

```
# include < local/rkf.h>

char *rkf_read(code, result, p)
int code;
char *result;
RKF *p;
```

DESCRIPTION

rkf_read is used to read a variety of parameters associated with the RKF structure **p*. *code* is a macro indicating what information is wanted. The result is returned in **result*. The result may be an integer, a double or a pointer to a double.

The codes are # defined in < local/rkf.h> as follows:

**result* is an integer:

ERROR_STATUS
: **result* set to current error status.

KOP
: **result* set to the current value of *kop* (indicates the amount of output which has restricted the natural stepsize selection).

NUM_FE
: **result* set to the current value of the number of function evaluations.

EQN_STIFF
: **result* set to the current value of the stiff equation indicator.

ACCEPT
: **result* set TRUE (= = 1) if the last call to the state equation *f* resulted in an integration step that was accepted. This is useful if *f* must do some updating or initializing for each integration step.

MODE
: **result* set to current integration mode.

STDIN_CHK
: **result* set to current *stdin* check mode.

SEQ_LEN
: **result* set to current *eq_len* value.

MAX_NFE
: **result* set to the current maximum for number of function evaluations.

MAX_KOP
: **result* set to the current maximum for number of outputs which will cause an IO_LIM error.

STEP_SIZE
: **result* set to current stepsize.

**result* is a double:

MIN_REL
: **result* set to current minumum value of the (machine independent portion of the) relative error tolerance.

REL_ERR
: **result* set to current value of relative error tolerance.

ABS_ERR
result set to current value of absolute error tolerance.

*result is a pointer to a double:

Y_PRIME
result set to a pointer to an array containing the state equation evaluated at the last output point.

EXAMPLE

```
int i;
double x, *y;
RKF *p;

rkf_read(ERROR, (char *)&i, p);
rkf_read(STEP_SIZE, (char *)&x, p);
rkf_read(Y_PRIME, (char *)&y, p);
```

LIBRARY

The function resides in **/usr/local/lib/librkf.a** and may be loaded by specifying the **-lrkf** flag to *cc(1)* or *ld(1)*.

NOTES

result should be cast into a character pointer as in the example.

Do not change any of the values in the buffer returned by the Y_PRIME code.

BUGS

To be more portable, *result* should be a pointer to a union, but that would make everything a little more complicated.

Some of the parameters only make sense if you know the internal workings of *rkf*.

FILES

< local/rkf.h>
/usr/local/lib/librkf.a

SEE ALSO

rkf_intro(3ML), rkf(3ML), rkf_open(3ML), rkf_init(3ML), rkf_error(3ML), rkf_mess(3ML), rkf_cntrl(3ML)

AUTHORS

Greg Bernstein
Tom Parker

Appendix II: Source Code for RKF

Here we present the source code for RKF. There are two files:

rkf.h
rkf.c.

```
typedef char RKF;

#define SING_STEP        1
#define ENDPT            2

#define OK               0
#define NEG_ERROR        1
#define REL_LIM          2
#define FUNC_LIM         3
#define IO_LIM           4
#define ZERO_ABS         5
#define STEP_LIM         6
#define STIFF_EQN        7
#define FUNC_ERR         8
#define STDIN_RDY        9
#define ERR_SET          10
#define INIT_SET         11

#define STEP_SIZE        1
#define KOP              2
#define NUM_FE           3
#define EQN_STIFF        4
#define Y_PRIME          5
#define ACCEPT           6
#define MODE             7
#define STDIN_CHK        8
#define SEQ_LEN          9
#define MAX_NFE          10
#define MAX_KOP          11
#define MIN_REL          12
#define REL_ERR          13
#define ABS_ERR          14
#define ERROR_STATUS     15
```

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

/*This file contains rkf()--a C86 version of the RKF45 differential
equation solving subroutine which was originally written by H. A. Watts
and L. F. Shampine of Sandia Laboratories, Albuquerque New Mexico.

               Written by: Greg Bernstein
               Date started: 10/2/84
               Update: 9/26/85 TSP


Updated on 10/9/84 to add stiffness detection as described in the
references:

    1) "Stiffness and Nonstiff Differential Equation Solvers, II:
       Detecting Stiffness With Runge-Kutta Methods" L. F. Shampine,
       ACM Transactions on Mathematical Software, Vol. 3, No. 1,
       March 1977, Pages 44-53.

    2) "Detecting stiffness with the Fehlberg (4,5) formulas" L. F.
       Shampine and K. L. Hiebert, Comp. and Maths. with Appls.
       Vol. 3, pp. 41-46. Pergamon Press 1977.

    Updated on 10/11/84 to:

    1)  Make internal procedures static

    2)  Allocate working storage for each problem according to system
        dimension this will also allow for simultaneous solutions of
        differential equations as in the variational equation problem.
        This was a major overhaul and involved adding procedures and
        changing slightly how the user interfaces with the routines.

    Updated on 10/12/84 to:

    1)  Get rid of redundant evaluations of yp.  In the case of a stepsize
        failure f1 thru f5 must be reevaluated but yp can be used again.

    Updated on 10/23/84 to:

    1)  take into account the possibility that the derivative function
        might not exist at a point where it is to be evaluated.  Note:
        that this required a slight addition to the "standard function"
        interface.

    Updated on 11/19/84 to:

    1)  Add supplementary functions to give user: the stepsize h,
        the number of function evaluations num_fe, the number of output
        points that impact the stepsize kop, and the stiff equation flag
        eqn_stiff.

    Updated on 3/29/85:
```

```
    1)  rkf_accept(), rkf_copy(), rkf_mess() added.

    2)  Cosmetic modifications and name changes to lower case.

    3)  stdin check added; also added h_failed to RKF_WS.

    Updated on 4/10/85:

    1) rkf_error(), rkf_init(), rkf_mode() added and corresponding
        arguments to rkf() deleted.

    2) Also error checking was distributed to these functions as much
        as possible.

    3) Initialization modes were deleted (if initialization is
        required, it is detected internally.

    4) nfe_max, kop_max, re_min and seq_len added to RKF_WS structure.

    Updated on 6/12/85:

    1) Error fixed in rkf() where *t was used instead of ws->t.

    2) rkf_copy() fixed to copy init_set, err_set & stdin_chk.

    3) calling rkf_init() clears STEP_LIM error.

    Updated on 9/22/85:

    1) Added user-supplied console status function constat().
        Deleted the stdin_chk flag. This alters rkf_stdin().

    Updated on 9/26/85:

    1) Added check in rkf() for ee == 0.0 to avoid divide by
    zero.

*/


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

#include <stdio.h>
#include "rkf.h"

/*  Program Constants  */

#define  NFE_MAX 3000             /* The expense is controlled by restricting
                                   the number of function evaluations to be
                                   approximately NFE_MAX. As set this corre-
                                   sponds to about 500 steps.  */
#define  KOP_MAX 100              /* Maximum number of outputs, that can
                                   impare the efficiency of the program, until
                                   the program returns with a warning flag. */
```

```
#define  RE_MIN 1.0E-12         /* Relative error minimum, machine indepen-
                                 dent part.  Attemps to obtain higher accur-
                                 acy with this algorithm are usually very
                                 expensive and often unsuccessful. */

#define  TRUE   1               /* The standard boolean values.  */
#define  FALSE  0

#define SUBSEQ_LEN      50      /* Subsequence length in stiffness test */

typedef struct
{
    unsigned n;                 /* Dimension of system to be solved */
    int (*f)();                 /* Pointer to the function to be
                                 integrated.  */
    int  step_init;             /* Indicates whether the step size has been
                                 initialized.  */
    double t;                   /* current time*/
    double h;                   /* The integration stepsize.  */
    double rel_err, abs_err;    /* error tolerances*/
    double re_min;              /* Some sort of min for error tolerances*/
    int mode;                   /* integration mode*/
    int err_flag;               /* Error indicator*/
    unsigned kop;               /* Efficiency impaired by output counter */
    unsigned kop_max;           /* Maximum for output counter*/
    unsigned num_fe;            /* Number of function evaluations counter */
    unsigned nfe_max;           /* Maximum allowed function evaluations*/
    int eqn_stiff;              /* Stiffness flag.  */
    unsigned seq_count;         /* Sequence counter for stiffness test. */
    unsigned seq_len;           /* Maximum stiffness sequence length*/
    unsigned sucss_12;          /* Number of successes of the (1,2) step
                                   in a sequence of length seq_len.  used
                                   in stiffness test.  */
    int step_accept;            /*TRUE if last evaluation of f()
                                   completed a successful integration step*/
    int h_failed;               /*TRUE if stepsize has been reduced in
                                previous iteration*/
    int init_set;               /* TRUE if init conds have been set*/
    int err_set;                /* TRUE if errors have been set*/
    int (*constat)();           /* function to call for console status */
    double *y;                  /* current trajectory point */
    double *yp, *f1, *f2, *f3,
                *f4, *f5, *s;   /* Pointers to arrays for holding
                                   intermediate calculations.  */
} RKF_WS;

/* Variables for machine epsilon calculation */
static double m_eps, u26;       /* Machine epsilon and 26 times the unit
                                 roundoff. */

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

RKF_WS *
rkf_open(f, n)
```

```
int (*f)();
unsigned n;

{
    char *calloc();
    RKF_WS *ws;

    if (n < 1)
        return(NULL);
    if ((ws = (RKF_WS *)calloc(1, sizeof(RKF_WS))) == NULL)
        return(NULL);

    /* allocate working storage arrays based on n */
    ws->y = (double *)calloc(n, sizeof(double));
    ws->yp = (double *)calloc(n, sizeof(double));
    ws->f1 = (double *)calloc(n, sizeof(double));
    ws->f2 = (double *)calloc(n, sizeof(double));
    ws->f3 = (double *)calloc(n, sizeof(double));
    ws->f4 = (double *)calloc(n, sizeof(double));
    ws->f5 = (double *)calloc(n, sizeof(double));
    ws->s  = (double *)calloc(n, sizeof(double));

    if (ws->y == NULL || ws->yp == NULL || ws->f1 == NULL ||
            ws->f2 == NULL || ws->f3 == NULL || ws->f4 == NULL ||
            ws->f5 == NULL || ws->s == NULL)
    {
        rkf_close(ws);
        return (NULL);
    }

    /* Initialize variables */
    ws->n = n;
    ws->f = f;
    ws->mode = ENDPT;
    ws->step_init = FALSE;
    ws->err_flag = OK;
    ws->kop = 0;
    ws->kop_max = KOP_MAX;
    ws->num_fe = 0;
    ws->nfe_max = NFE_MAX;
    ws->eqn_stiff = FALSE;
    ws->seq_count = 0;
    ws->seq_len = SUBSEQ_LEN;
    ws->sucss_12 = 0;
    ws->kop = 0;
    ws->rel_err = 0.0;
    ws->abs_err = 0.0;
    ws->step_accept = TRUE;
    ws->h_failed = FALSE;
    ws->init_set = FALSE;
    ws->err_set = FALSE;
    ws->constat = NULL;
    ws->re_min = RE_MIN;

    return(ws);
}
```

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

/*Disposes of the working storage structure pointed to by ws.  It
first disposes of the working storage arrays and then the entire
structure */

rkf_close(ws)

RKF_WS *ws;

{
    if (ws->y != NULL)
        free((char *)ws->y);
    if (ws->yp != NULL)
        free((char *)ws->yp);
    if (ws->f1 != NULL)
        free((char *)ws->f1);
    if (ws->f2 != NULL)
        free((char *)ws->f2);
    if (ws->f3 != NULL)
        free((char *)ws->f3);
    if (ws->f4 != NULL)
        free((char *)ws->f4);
    if (ws->f5 != NULL)
        free((char *)ws->f5);
    if (ws->s != NULL)
        free((char *)ws->s);
    if (ws != NULL)
        free((char *)ws);
    return;
}

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

/*Copies working storage structure s to d. d must be previously
allocated by a call to rkf_open(). The working vectors s->f1 through
s->f5 and s->s are not copied.*/

rkf_copy(d, s)

RKF_WS *d, *s;

{
    d->n = s->n;
    d->f = s->f;
    d->step_init = s->step_init;
    d->t = s->t;
    d->h = s->h;
    d->rel_err = s->rel_err;
    d->abs_err = s->abs_err;
    d->re_min = s->re_min;
    d->mode = s->mode;
    d->err_flag = s->err_flag;
    d->kop = s->kop;
    d->kop_max = s->kop_max;
    d->num_fe = s->num_fe;
    d->nfe_max = s->nfe_max;
    d->eqn_stiff = s->eqn_stiff;
    d->seq_count = s->seq_count;
    d->seq_len = s->seq_len;
    d->sucss_12 = s->sucss_12;
    d->step_accept = s->step_accept;
    d->h_failed = s->h_failed;
    d->init_set = s->init_set;
    d->err_set = s->err_set;
    d->constat = s->constat;
    vec_copy(s->n, d->y, s->y);
    vec_copy(s->n, d->yp, s->yp);
}

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

/*Returns a string describing err.*/

char *
rkf_mess(err)

int err;

{
    switch (err)
    {
    case OK:
        return ("integration successful");
    case NEG_ERROR:
        return ("negative error tolerance");
    case REL_LIM:
        return ("relative error tolerance too small");
    case FUNC_LIM:
        return ("too many function evaluations");
    case IO_LIM:
        return ("too much output");
    case ZERO_ABS:
        return ("nonzero absolute error required");
    case STEP_LIM:
        return ("stepsize too small");
    case STIFF_EQN:
        return ("stiff equation");
    case FUNC_ERR:
        return ("function evaluation error");
    case STDIN_RDY:
        return ("character ready at stdin");
    case ERR_SET:
        return ("abs_err and rel_err not properly initialized");
    case INIT_SET:
        return ("initial conditions not properly initialized");
    default:
        return ("Bad error code");
    }
}

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

rkf_error(rel_err, abs_err, ws)

double rel_err, abs_err;
RKF_WS *ws;

{
#ifdef DEBUG
    fprintf(stderr, "Entering rkf_error()...\n");
#endif
    ws->err_set = TRUE;
    if (ws->err_flag == ZERO_ABS)
    {
        if (abs_err > 0.0)
        {
            ws->abs_err = abs_err;
            ws->err_flag = OK;
        }
    }
    else if (ws->err_flag == STEP_LIM)
    {
        if (abs_err > ws->abs_err)
        {
            ws->abs_err = abs_err;
            ws->err_flag = OK;
        }
        if (rel_err > ws->rel_err)
        {
            ws->rel_err = rel_err;
            ws->err_flag = OK;
        }
    }
    else
    {
        if (rel_err < 0.0 || abs_err < 0.0)
        {
            ws->err_flag = NEG_ERROR;
            ws->err_set = FALSE;
        }
        else if (rel_err < 2.0*m_eps + ws->re_min)
        {
            ws->rel_err = 2.0*m_eps + ws->re_min;
            ws->abs_err = abs_err;
            ws->err_flag = REL_LIM;
        }
        else
        {
            ws->rel_err = rel_err;
            ws->abs_err = abs_err;
            ws->err_flag = OK;
        }
    }
#ifdef DEBUG
    fprintf(stderr, "...leaving rkf_error()\n");
#endif
    return (ws->err_flag);
}

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

rkf_init(t, x, ws)

double t, x[];
RKF_WS *ws;

{
#ifdef DEBUG
    fprintf(stderr, "Entering rkf_init()...\n");
#endif
    ws->t = t;
    vec_copy(ws->n, ws->y, x);
    ws->step_init = FALSE;      /*need to re-initialize*/
    ws->init_set = TRUE;
    if (ws->err_flag == INIT_SET || ws->err_flag == STEP_LIM)
        ws->err_flag = OK;
#ifdef DEBUG
    fprintf(stderr, "...leaving rkf_init()\n");
#endif
}

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

/*the main routine, returns err_flag*/

rkf(tout, t, y, ws)

double tout, *t, y[];
RKF_WS *ws;

{
    int k;
    int sol_out;
    double dt, scale, ae, h_min;
    double eeoet, et, temp_et, ee, esttol, h_scale;
    double eeoet_12, ee_12, esttol_12;
    double temp1, temp2, temp3;
    double max(), min(), sign(), pow(), fabs();

#ifdef DEBUG
    fprintf(stderr, "Entering rkf()...\n");
#endif
    if (m_eps == 0.0)
        calc_eps();
    if (!par_check(ws))
        return (rkf_exit(t, y, ws->err_flag, ws));
    if (tout == ws->t)  /*no computation necessary*/
        return (rkf_exit(t, y, ws->err_flag, ws));
    if (!ws->step_init)
    {
        /* Evaluate yp if possible. */
        if (!(*ws->f)(ws->yp, ws->y, ws->t))
            return (rkf_exit(t, y, FUNC_ERR, ws));
        ws->num_fe++;
        init_step(ws, tout);
        ws->step_init = TRUE;
    }
    dt = tout - ws->t;

    /* "Efficiency impaired by too much output" check */
    if (fabs(ws->h) >= 2.0*fabs(dt))
        ws->kop++;
    if (ws->kop >= ws->kop_max)
        return (rkf_exit(t, y, IO_LIM, ws));

    /* If "too close" to output extrapolate and return */
    if (fabs(dt) <= u26*fabs(ws->t))
    {
        if (!(*ws->f)(ws->yp, ws->y, ws->t))
            return (rkf_exit(t, y, FUNC_ERR, ws));
        ws->num_fe++;
        for (k = 0; k < ws->n; k++)
            ws->y[k] += dt*ws->yp[k];
        return (rkf_exit(t, y, OK, ws));
    }

    sol_out = FALSE;  /* initialize solution output indicator */

    /* to avoid premature under flow in the error tolerance function
       scale the error tolerances */
    scale = 2.0/ws->rel_err;
    ae = scale*ws->abs_err;

    do          /* Step by step integration loop */
    {
        h_min = u26 * fabs(ws->t);  /* init smallest allowed step */

        /* Adjust stepsize if necessary to hit the output point.  Look ahead
          two steps to avoid drastic changes in the stepsize and thus lessen
          the impact of output points on the code. */
        dt = tout - ws->t;
        if (fabs(dt) < 2.0 * fabs(ws->h))
        {
            if (fabs(dt) <= fabs(ws->h))
            {
                sol_out = TRUE;
                ws->h = dt;
            }
            else
                ws->h = 0.5*dt;
        }
        do      /*  Stepsize adjustment and error checking loop */
        {
            if (ws->constat != NULL  && (*ws->constat)())
                return (rkf_exit(t, y, STDIN_RDY, ws));
            if (ws->num_fe > ws->nfe_max)
            {
                if (ws->eqn_stiff)
                    return (rkf_exit(t, y, STIFF_EQN, ws));
                else
                    return (rkf_exit(t, y, FUNC_LIM, ws));
            }

            /* Advance an approximate solution over one step of length h */
            ws->step_accept = FALSE;
            if (!fehl45(ws->f, ws->n, ws->y, ws->t,
                    ws->h, ws->yp, ws->f1, ws->f2,
                    ws->f3, ws->f4, ws->f5, ws->s))
                return (rkf_exit(t, y, FUNC_ERR, ws));
            ws->num_fe += 5;

            /* Compute and test allowable tolerances versus local error
            estimates and remove scaling of tolerances.  Note that
            relative error is measured with respect to the average of
            the magnitudes of the solution at the beginning and end
            of the step.  The error estimate formula has been
            grouped to control loss of significance.*/

            /* Do this also for the imbedded (1,2) formula as part of
            the stiffness test */

            eeoet = eeoet_12 = 0.0; /*Used in stiffness test*/
            for (k = 0; k < ws->n; k++)
            {
                temp_et = fabs(ws->y[k]) + fabs(ws->s[k]);
                et = ae + temp_et;
                if (et <= 0.0)  /*inappropriate error tolerance*/
                    return (rkf_exit(t, y, ZERO_ABS, ws));

                /* Group terms to avoid loss of signifigance */
                temp1 = (21970.0*ws->f3[k] - 15048.0*ws->f4[k]);
                temp2 = (-2090.0*ws->yp[k] + temp1);
                temp3 = (22528.0*ws->f2[k] - 27360.0*ws->f5[k]);
                ee = fabs(temp2 + temp3);

                /* Stiffness testing */
                if (!ws->eqn_stiff &&
                    (ws->seq_count - ws->sucss_12 <= ws->seq_len/2))
                {
                    ee_12 = fabs(0.055455*ws->yp[k] -
                        0.035493*ws->f1[k] - 0.036571*ws->f2[k] +
                        0.023107*ws->f3[k] - 0.009515*ws->f4[k] +
                        0.003017*ws->f5[k]);
                    /*Here we need to avoid a divide-by-zero. This can
                    happen if all the f?[k] are zero which can happen
                    if the state equation returns 0.0 for the kth
                    entry.  I assume that if ee == 0.0 then ee_12 ==
                0.0 and eeoet_12 does not need to be updated.*/
                if (ee != 0.0)
                    eeoet_12 = max(eeoet_12, ee_12/ee);
            }
            eeoet = max(eeoet, ee/et);
        }
        esttol = fabs(ws->h)*eeoet*scale/752400.0;

        /* Stiffness testing */
        if (!ws->eqn_stiff &&
                (ws->seq_count - ws->sucss_12 <= ws->seq_len/2))
            esttol_12 = fabs(ws->h)*eeoet_12*scale;

        if (esttol > 1.0)
        {
            /*Unsuccessful step: reduce the stepsize and try
            again.  The stepsize decrease is limited to a factor of
            ten.  Practical limits on the change in the stepsize
            are enforced to smooth the stepsize selection process
            and to avoid excessive chattering on problems having
            discontinuities. */

            ws->h_failed = TRUE;
            sol_out = FALSE;
            h_scale = 0.1;
            if (esttol < 59049.0)
                h_scale = 0.9/pow(esttol, 0.20);
            /* To prevent unnecessary failures the code uses 9/10
             the stepsize it estimates will succeed.  */
            ws->h *= h_scale;
            if (fabs(ws->h) <= h_min)
                return (rkf_exit(t, y, STEP_LIM, ws));
        }
    }
    while (esttol > 1.0);

    /* Successful step: store solution at t+h */
    ws->step_accept = TRUE;
    if (!(*ws->f)(ws->f1, ws->s, ws->t + ws->h))
        return (rkf_exit(t, y, FUNC_ERR, ws));
    ws->num_fe++;
    ws->t += ws->h;
    vec_copy(ws->n, ws->y, ws->s);
    vec_copy(ws->n, ws->yp, ws->f1);

    /* choose next stepsize: the increase is limited to a factor of
       5.  If a step failure has just occurred, the next stepsize is not
       allowed to increase. This makes the code more efficient on
       problems with discontinuities. */
    h_scale = 5.0;
    if (esttol > 1.889568E-4)
        h_scale = 0.9/pow(esttol, 0.2);
    /*don't increase stepsize if step failure in last iteration*/
    if (ws->h_failed)
        h_scale = min(h_scale, 1.0);
    ws->h = sign(ws->h)*max(h_scale*fabs(ws->h), h_min);
```

```
        /* Stiffness testing */
        if (!ws->eqn_stiff)
        {
            if (ws->seq_count++ - ws->sucss_12 <= ws->seq_len/2)
            {
                if (esttol_12 >= 1.0)
                        ws->sucss_12++;
                if (ws->sucss_12 >= ws->seq_len/2)
                    ws->eqn_stiff = TRUE;
            }
            if (ws->seq_count > ws->seq_len || ws->eqn_stiff)
                ws->seq_count = ws->sucss_12 = 0;
        }
        ws->h_failed = FALSE;   /*reinit step failure*/
    } while (!sol_out && ws->mode != SING_STEP);

    if (sol_out)
        ws->t = tout;
    return (rkf_exit(t, y, OK, ws));
}

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

/*checks parameter values in ws*/

static
par_check(ws)

RKF_WS *ws;

{
#ifdef DEBUG
    fprintf(stderr, "Entering par_check()...\n");
#endif
    switch (ws->err_flag)
    {
    case OK:
    case REL_LIM:
    case FUNC_ERR:
    case STDIN_RDY:
        ws->err_flag = OK;
        break;
    case FUNC_LIM:
        ws->num_fe = 0;
        ws->err_flag = OK;
        break;
    case IO_LIM:
        ws->kop = 0;
        ws->err_flag = OK;
        break;
    case STIFF_EQN:
        ws->num_fe = ws->seq_count = ws->sucss_12 = 0;
        ws->eqn_stiff = 0;
        ws->err_flag = OK;
```

```
            break;
        default:
            return (FALSE); /*uncorrected error*/
        }
        if (!ws->err_set)
        {
            ws->err_flag = ERR_SET;
            return (FALSE);
        }
        if (!ws->init_set)
        {
            ws->err_flag = INIT_SET;
            return (FALSE);
        }
#ifdef DEBUG
        fprintf(stderr, "...leaving par_check()\n");
#endif
        return (TRUE);
}

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

/*Initializes the stepsize h.*/

static
init_step(ws, tout)

RKF_WS *ws;
double tout;

{
        int k;
        double  ypk, toln, tol, dt;
        double pow(), max(), sign(), fabs();

#ifdef DEBUG
        fprintf(stderr, "Entering init_step()...\n");
#endif
        dt = tout - ws->t;
        ws->h = fabs(dt);
        toln = 0.0;

        for (k= 0; k < ws->n; k++)
        {
            tol = ws->rel_err * fabs(ws->y[k]) + ws->abs_err;
            if (tol > 0.0)
            {
                toln = tol;
                ypk = fabs(ws->yp[k]);
                if (ypk * pow(ws->h, 5.0) > tol)
                    ws->h = pow(tol/ypk, 0.20);
            }
        }

        if (toln <= 0.0)
        ws->h = 0.0;
    ws->h = max(ws->h, u26*max(fabs(ws->t), fabs(dt)));
    ws->h *= sign(dt);
#ifdef DEBUG
    fprintf(stderr, "...leaving init_step()\n");
#endif
}

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

/*exit routine for rkf()*/

static
rkf_exit(t, y, err_code, ws)

int err_code;
double *t, y[];
RKF_WS *ws;

{
    *t = ws->t;
    vec_copy(ws->n, y, ws->y);
#ifdef DEBUG
    fprintf(stderr, "...leaving rkf()\n");
#endif
    return (ws->err_flag = err_code);
}

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

/* The Fehlberg 4,5 formulas as implemented by Watts and Shampine.  The
terms have been grouped to avoid loss of signifigance. fehl45() assumes
that the derivative at time t, yp, is given. If derivative function
cannot be evaluated fehl45() returns FALSE, otherwise it returns TRUE. */

static
fehl45(f, n, y, t, h, yp, f1, f2, f3, f4, f5, s)

int (*f)();
unsigned n;
double y[], t, h, yp[], f1[], f2[], f3[], f4[], f5[], s[];

{
    unsigned k;
    double ch, temp1, temp2, temp3, temp4;

#ifdef DEBUG
    fprintf(stderr, "Entering fehl45()...\n");
#endif
    ch = h/4.0;
    for (k = 0; k < n; k++)
        s[k] = y[k] + ch * yp[k];
    if (!(*f)(f1, s, t + ch))
        return(FALSE);
    ch = 3.0*h/32.0;
    for (k = 0; k < n; k++)
    {
        temp1 = yp[k] + 3.0 * f1[k];
        s[k] = y[k] + ch * temp1;
    }
    if (!(*f)(f2, s, t + 3.0*h/8.0))
        return(FALSE);

    ch = h/2197.0;
    for (k = 0; k < n; k++)
    {
        temp1 = (7296.0*f2[k] - 7200.0*f1[k]);
        temp2 = (1932.0*yp[k] + temp1);
        s[k] = y[k] + ch * temp2;
    }
    if (!(*f)(f3, s, t + 12.0*h/13.0))
        return(FALSE);

    ch = h/4104.0;
    for (k = 0; k < n; k++)
    {
        temp1 = (8341.0*yp[k] - 845.0*f3[k]);
        temp2 = (29440.0*f2[k] - 32832.0*f1[k]);
        temp3 = (temp1 + temp2);
        s[k] = y[k] + ch * temp3;
    }
    if (!(*f)(f4, s, t + h))
        return(FALSE);

    ch = h/20520.0;
    for (k = 0; k < n; k++)
    {
        temp1 = (9295.0*f3[k] -5643.0*f4[k]);
        temp2 = (41040.0*f1[k] - 28352.0*f2[k]);
        temp3 = (-6080.0*yp[k] + temp1);
        temp4 = (temp3 + temp2);
        s[k] = y[k] + ch * temp4;
    }
    if (!(*f)(f5, s, t + h/2.0))
        return(FALSE);

    /* compute approximate solution at t+h. */
    ch = h/7618050.0;
    for (k = 0; k < n; k++)
    {
        temp1 = (3855735.0*f3[k] - 1371249.0*f4[k]);
        temp2 = (902880.0*yp[k] + temp1);
        temp3 = (3953664.0*f2[k] + 277020.0*f5[k]);
        temp4 = (temp2 + temp3);
        s[k] = y[k] + ch * temp4;
    }
#ifdef DEBUG
    fprintf(stderr, "...leaving fehl45()\n");
#endif
```

```
    return (TRUE);
}

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

/* These functions allow the user to alter the default values for
rkf() "constants".*/

rkf_mode(mode, ws)

int mode;
RKF_WS *ws;

{
    if (mode == SING_STEP)
        ws->mode = SING_STEP;
    else
        ws->mode = ENDPT;
}

rkf_stdin(constat, ws)

int (*constat)();
RKF_WS *ws;

{
    ws->constat = constat;
}

rkf_nfe(n, ws)

unsigned n;
RKF_WS *ws;

{
    ws->nfe_max = n;
}

rkf_kop(n, ws)

unsigned n;
RKF_WS *ws;

{
    ws->kop_max = n;
}

rkf_seq_len(n, ws)

unsigned n;
RKF_WS *ws;

{
    ws->seq_len = n;
}
```

```
rkf_re_min(x, ws)

double x;
RKF_WS *ws;

{
    if (x > 0.0)
        ws->re_min = x;
}

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

typedef union read_ptr
{
    char *cp;
    int *ip;
    double *dp;
    double **ddpp;
};

rkf_read(code, p, ws)

char *p;
int code;
RKF_WS *ws;

{
    union read_ptr rp;

    rp.cp = p;
    switch (code)
    {
    case STEP_SIZE:
        *rp.dp = ws->h;
        break;
    case KOP:
        *rp.ip = ws->kop;
        break;
    case NUM_FE:
        *rp.ip = ws->num_fe;
        break;
    case EQN_STIFF:
        *rp.ip = ws->eqn_stiff;
        break;
    case Y_PRIME:
        *rp.ddpp = ws->yp;
        break;
    case ACCEPT:
        *rp.ip = ws->step_accept;
        break;
    case MODE:
        *rp.ip = ws->mode;
        break;
    case STDIN_CHK:
```

```
		*rp.ip = ws->constat != NULL;
		break;
	case SEQ_LEN:
		*rp.ip = ws->seq_len;
		break;
	case MAX_NFE:
		*rp.ip = ws->nfe_max;
		break;
	case MAX_KOP:
		*rp.ip = ws->kop_max;
		break;
	case MIN_REL:
		*rp.dp = ws->re_min;
		break;
	case REL_ERR:
		*rp.dp = ws->rel_err;
		break;
	case ABS_ERR:
		*rp.dp = ws->abs_err;
		break;
	case ERROR_STATUS:
		*rp.ip = ws->err_flag;
		break;
	}
}

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

/*Calculates the machine epsilon and 26 times the unit roundoff.*/

static
calc_eps()

{
	double sum = 2.0;	/*used because of 8087. Without it, m_eps would
					represent the internal (80 bit) accuracy
					of the 8087, not the external (64 bit)
					accuracy of the double representation.*/

	for (m_eps = 1.0; sum > 1.0 ; sum = 1.0 + m_eps)
		m_eps /= 2.0;
	u26 = 26.0*m_eps;
}

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

/*The standard sign function: +1 if argument greater than or equal to
zero, -1 otherwise.*/

static double
sign(x)

double	x;
```