CALIFORNIA SOFTWARE CO.
CALIFORNIA SOFTWARE CO.
CALIFORNIA SOFTWARE CO.

SCRUNCH

Numerical Computations
on Very
Small Machines

by KRIS STEWART

CALIFORNIA SOFTWARE CO.
CALIFORNIA SOFTWARE CO.
CALIFORNIA SOFTWARE CO.
CALIFORNIA SOFTWARE CO.

SCRUNCH

NUMERICAL COMPUTATIONS

ON VERY SMALL MACHINES

Kris Stewart
Dept. Mathematical Sciences
San Diego State University
San Diego, CA 92182

Abstract:

SCRUNCH is a suite of 9 numerical analysis routines coded in BASIC for microcomputers providing solutions to many typical computations. Part 1 of this paper discusses general aspects of numerical computations on memory-limited computers with specific data for the routines in SCRUNCH. Part 2 provides support for each individual routine. A separately available appendix (in the the form of listings or 5" diskette) contains the source code of the numerical routines along with test drivers for each.

This paper is presented to the faculty of
San Diego State University
in partial support of the degree
Master of Science in Computer Science

Approved by

Vernor Vinge ..... Vernor Vinge, Chairman

David Lesley ..... F. David Lesley

Arnold L. Villone ..... Arnold L. Villone

EL CERRITO
CALIFORNIA SOFTWARE
1979

Disclaimer of Warranties and Limitation of Liabilities.

California Software makes no expressed or implied waranties on any kind with regard to the programs and/or documentation supplied in this package. In no event shall California Software be liable for incidental or consequential damages in connection with or arising out of the furnishing, performance or use of any of these programs.

PUBLISHER'S NOTES

Kris Stewart contacted me in October of 1977 to discuss her Masters degree project and wondered about its publication. She wanted to know if I was interested. I immediately recognized the fact that published academic work which resulted in the production of real and usable software would be a major breakthrough for the users of the small and powerful machines we call microcomputers. Besides being immediately useful to students and teachers of numerical analysis, these programs represent quality programming at its best and can only be an encouragement to other academicians to seriously consider incorporating their work into this growing environment.

John C. Dvorak
Berkeley, April 30, 1979

AVAILABILITY OF SOURCE CODE

SCRUNCH is provided with a 5" Northstar compatible diskette on which the documented source code is provided.

For those who must have listings they are available as a separate publication for $20.

Also CBASIC and MBASIC diskettes will be available for $20.

For ordering information, write:

CALIFORNIA SOFTWARE
BOX 275
EL CERRITO, CA 94530

Part 1. Numerical Computing on Very Small Machines

This paper is primary documentation for SCRUNCH, a package of numerical analysis routines providing solutions to the basic problems

Ordinary differential equations using RKF45 [1]
Adaptive quadrature using SIMP [2]
Optimization (function of one real variable) using FNM [1]
Root-finding (function of one real variable) using ZEROIN [2]
Spline interpolation using SPLINE and FNS [1]
Linear equations using DECOMP and SOLVE [1]
Least squares solution of an over-determined system of linear equations using HECOMP and HOLVE [3]
Singular value decomposition of an m by n matrix using SVD [1]
Symmetric eigensystem using SYMEIG [3]

The routines in SCRUNCH are all coded in BASIC and were developed and tested on a CROMEMCO Z-80 microcomputer in an IMSAI box using North Star Disk and BASIC with 32K of 250 ns memory. This system is also equipped with a North Star Floating Point Board (FPB-A) and a Canada Systems Real Time Clock for accuracy and timing comparisons.

Each routine begins with a long initial segment of remarks, defining the routine, input and output parameters, dimension requirements and internal variables. All the routines will fit and execute properly in 24K memory systems. On 24K systems, a few of the longer routines (RKF45, SVD) may require the initial segment of remarks be deleted before medium to large sized drivers can be added in front.

References:

The routines in SCRUNCH are translations of state-of-the-art FORTRAN routines taken from two excellent introductory numerical analysis texts and an unpublished set of course notes.

[1] Computer Methods for Mathematical Computations
by George E. Forsythe, Michael A. Malcolm and Cleve B. Moler
Prentice-Hall, Inc. 1977

[2] Numerical Computing: an Introduction
by Lawrence F. Shampine and Richard C. Allen, Jr.
W. B. Saunders Company 1973

[3] Matrix Eigenvalue and Least Squares Computations
by Cleve B. Moler
Computer Science Department, Stanford University
March, 1974 (unpublished course notes used with permission of the author)

General form of SCRUNCH routines:

The nine routines in SCRUNCH were all coded in BASIC by the author and a serious attempt was made to keep the parameter lists, and input and output information consistent. Therefore the user should find it easy to go from one routine to the next.

The 9 routines each have a long initial segment of comments designed to allow them to stand-alone. This paper defines the methods used, but the general user should be able to obtain all the information he needs to use the routine from the initial segment of remarks.

The following information is specified for each routine:

Variables which the user must initialize before calling the routine.

Variables which the routine sets after being called.

All arrays and vectors which the routine uses to perform the numerical solution. These must all be dimensioned by the user in his driver.

All variable names used internally in the routine.
Note: the routine will change these values and usually they shouldn't be changed by the user in between calls to the routine.

Any necessary user-written defined function to be used by the routine (SIMP, FNM, ZEROIN).

Any necessary user-written subroutine to be used by the the numerical routine (RKF45).

This paper is intended to support the routines in SCRUNCH which run in BASIC on a memory-limited, small computer. It is not intended to be a text on numerical analysis in general, but should be of great value to anyone with a good background in numerical analysis who wants to use a microcomputer to perform the computations. If a microprocessor is available, it can also be used as a dramatic and responsive class room tool for an introductory undergraduate numerical analysis course. The theory behind the original FORTRAN routines is available in the above references (although some details for the routines taken from [3] will be provided since these are unpublished notes).

Mathematical Software:

The choice of these three sources was made because the original authors employed consistent attitudes towards high-quality mathematical software when writing the FORTRAN routines, making the translation into BASIC straightforward. High-quality mathematical software is taken to mean software that is well-documented (both commented in the code and in auxilliary documentation) and user-oriented. The routines should always execute, and even when a correct answer is not obtained, the numerical routine itself should provide the user with diagnostics indicating that the result provided is not to be trusted, hopefully along with an indication of what caused the problem to go sour.

The user should be required to provide only a minimal amount of information to specify the problem to be solved. For example, the ordinary differential equation solver, RKF45, written by H. A. Watts and L. F. Shampine [1], only asks the user to specify the mathematical definition of the initial value problem to be solved and error tolerances which should be met by the code in performing the numerical solution. The code takes care of selecting an initial step-size, monitoring its estimated error as the solution progresses and adjusting the step-size accordingly, and checking for trouble spots in the numerical solution. The code will set a flag to indicate to the user that either the numerical solution was successfully advanced, or trouble was encountered (in which case an indication of the cause is given).

Test drivers:

A complete set of test drivers for each of the numerical routines in SCRUNCH has been provided. All the test drivers begin with an initial segment of remarks which define the problem being solved. These routines are admittedly North Star dependent since the I/O statements make use of the formatting available with North Star BASIC. They should still be of practical value to a casual user as a model to follow when setting up and using the numerical routines. The test drivers are each in a separate file on the diskette provided with SCRUNCH. Before using them, the proper numerical routine must be APPENDed to the end of the driver. For 24K systems, it is also necessary to delete the leading set of remarks from RKF45 and SVD before they can be APPENDed.

SCRUNCH routine	Test driver	Problem solved - additional remarks
RKF45	ORBIT	3-body orbit problem of APOLLO capsule about Earth and Moon.
SIMP FNM	DAWSON	Find where Dawson's integral attains its maximum. NOTE: FNM should as usual begin at stt. 1000, but SIMP should be renumbered to begin at stt. 3000 before APPENDing.
ZEROIN	CATEN	Find maximum tension in a wire suspended between 2 towers.
SPLINE FNS	THERMO	Given voltage and temperature data at specific points, use SPLINE to interpolate at other points.
DECOMP SOLVE	ELECNET	Find the potentials at the junctions of a given electrical network.
HECOMP HOLVE	BEVHEC	Find the 3rd degree least squares fit for the voltage of a thermocouple junction as a function of temperature.
SVD	BEVSVD	Solves same problem as BEVHEC, but gives more information on the design matrix.
SYMEIG	VIBRAT	Mechanical vibration frequencies of a spring-mass system.
RKF45 ZEROIN	SHOOT	Non-linear two point boundary value problem using shooting method. NOTE: RKF45 should begin at stt. 1000, but ZEROIN should be renumbered to begin at stt. 5000 before being APPENDed.

Considerations due to the language BASIC:

1. There are variable name limitations and all variables are global (except in defined-functions). Since BASIC only allows a letter or a letter and a number for variable names, it was decided to sacrifice some readability in the BASIC code so as to minimize the letters used internally by the routines. For example, in RKF45 many variables were needed to keep track of errors and to select step-size. The code uses E0,E1,E2,E3,E4 for errors and T0,T1,T2,T3,T4,T5,T6,T7,T8,T9 for step-size computations and, since they are globals, the user is asked not to use these in between calls to the routine.

2. All dimensioning must be done by the user. Since space is considered one of the big limitations on small computers, this isn't too big a nuisance. The initial remarks precisely describe the requirements of each routine.

3. The numerical routines must specify statement numbers for auxilliary subroutines and use specific function names for routines to be written by the user. Following is a list of those routines in SCRUNCH that require user-written routines and the form they should have:

SCRUNCH routine	User-supplied auxilliary routine
RKF45	subroutine yprime input (X1,Y1) output (Y2) This subroutine must begin at statement #500 and should evaluate the system of first order differential equations.
SIMP	DEF FNY (x) Defined-function to evaluate the integrand.
FNM	DEF FNF (x) Defined-function to evaluate the function whose minimum is sought.
ZEROIN	DEF FNF (x) Defined-function to evaluate the function whose root is sought.

Space considerations:

All of the routines will fit in 24K of memory along with the North Star DOS and BASIC (which occupy about 15K). The following table gives the size of each routine on the diskette provided to the user, the size when the initial remarks are deleted, and the additional space needed for the routines to run (i.e. for vectors and temporary storage). It was found that in putting together the routines with some test drivers [ORBIT,BEVSVD,SHOOT], these remarks needed to be deleted in order to have the test driver and the numerical routine fit in 24K. It is suggested that the 24K user keep the routines and their remarks intact (since these remarks are very helpful when trying to get a problem running), but make a duplicate copy of the routines without the remarks for use in memory-restricted environments.

Additional tests were run to find that when using the full 32K memory, the largest system of linear equations that could be handled was a 45 by 45 system and the largest system of ordinary differential equations was about 150 (very time consuming, though).

Routine	Length (bytes)	Length w/o rems	Delete stt. 1010 thru -	Additional space needed to run the driver
RKF45	8990	5900	1760	
ORBIT	2810			670
SHOOT	2200			1900 for RKF45 200 for ZEROIN
SIMP	4050	2065	1500	
FNM	2450	1450	1330	
DAWSON	2200			1700 for SIMP 300 for FNM
ZEROIN	3700	1700	1490	
CATEN	1350			225
SPLINE & FNS	3800	2350	1500	
THERMO	2100			675
DECOMP &SOLVE	4500	2900	1350	
ELECNET	2500			450
HECOMP &HOLVE	3700	2400	1440	
BEVHEC	3800			1600
SVD	7850	5750	1570	
BEVSVD	3500	2450	10-145;505-620	1700
SYMEIG	4550	3225	1390	
VIBRAT	2250			1100

Accuracy considerations:

The first consideration when discussing accuracy is that any numerical routine will be limited by the specific machine and language it has been implemented in. In the SCRUNCH package, the driver PRECIS has been provided. It is a very short routine that will compute an estimate of the unit roundoff of the system. The unit roundoff is the smallest number, U0, that can be represented in the machine such that

$$1.0 + U0 > 1.0 \quad .$$

The value actually computed by PRECIS will be within a few powers of 2 of this number, but this is all that is necessary. The routines use U0 in a conservative manner to forestall the user asking for more accuracy than his system can deliver and to prevent overflows and divide checks in numerical computations. The routines provided on the diskette all have U0 set for 8-digit BASIC on the developmental machine. The user should run the routine PRECIS to verify that the approximate unit roundoff of his system is comparable to the value listed below. If it is way off (i.e., by 10, 100), the user is advised to edit the routines and insert his estimate U0. For this purpose, the routines which use U0 are listed below along with the statement number where U0 is defined.

Note: The 8-digit value for U0 = 2.98e-8, guarantees that asking for relative and absolute error tolerances in all the test drivers of 1.e-6 should be deliverable on this equipment.

For the developmental system, the following values were found:

Software	Unit roundoff
	**** value set in SCRUNCH ****
North Star 8-digit BASIC	2.98 e -08
North Star 14-digit BASIC	2.84 e -14

SCRUNCH routine	Statement number where approximate unit roundoff U0 is defined
RKF45	1780
SIMP	1520
FNM	1350
ZEROIN	1520

Timing Considerations

A variety of timing results are provided here. The CL2400 Real Time Clock from Canada Systems, Inc. was used to get integral numbers of seconds for executing the test drivers provided for each numerical routine. The North Star Floating Point Board (FPB-A) used by the North Star 8-digit and 14-digit FP BASIC to perform BCD arithmetic in hardware substantially improves execution times for computationally bound numerical problems. A good example is the performance of RKF45 in the test driver ORBIT, with and without the FP Board (Fig. 1, 2, 3 and 4). The time to take an interval step is cut in half when the FP BASIC is used with 8-digit precision and the 14-digit FP BASIC timings are about the same as the 8-digit FP BASIC timings even though a full extra 6 digits of accuracy are being delivered. Notice that it wasn't necessary for the code to reset the error tolerances for the 14-digit FP BASIC run. All the test drivers were run using the 4 sets of North Star software and very similar results were obtained for the improvement of computation times, i.e. the 14-digit FP BASIC always beat the 8-digit BASIC and was comparable to the 8-digit FP BASIC. So if you need the accuracy provided by 14 digit arithmetic, the Floating Point Board is invaluable.

Following is a comparative table of execution times for the entire test driver, i.e. from the first statement to the END. This shows that solving a numerical problem is not all computation, since for most of the drivers the decrease in time using the FP BASIC is not as dramatic as in the computationally-bound ORBIT. All test drivers asked for error tolerances of 1.e-6 (except the linear algebra routines which don't have error tolerances).

Time to compute in seconds using 8-digit BASIC

Driver	without FPB	with FPB
ORBIT	578	325
DAWSON	218	147
(time for FNM to find missing end point of integration)	207	139
(time for SIMP to integrate to this end point)	6	4
CATEN	6	4
VIBRAT	60	45
ELECNET	15	14
BEVHEC	30	30
BEVSVD	43	36
THERMO	36	33
SHOOT	101	70
(time for ZEROIN to find missing slope)	84	57
(time for RKF45 to integrate the entire interval)	12	8

FIGURE 1. This test uses North Star 8-digit BASIC

```
 THREE BODY ORBIT PROBLEM

 TEST DRIVER FOR RKF45 - THE ODE SOLVER

    X          Y1                Y2          FLAG    TIME TO COMPUTE

  .0000    1.2000000000     .0000000000       3            0
 ERROR RESET   REL.=   .0000030396    ABS.=   .0000010000
  .2000    1.1643714000    -.2042720000       2           11
  .4000    1.0661770000    -.3801237600       2            7
  .6000     .9196409000    -.5082030100       2            4
  .8000     .7401470200    -.5717005400       2            9
 1.0000     .5453131900    -.5537130100       2            6
 1.2000     .3528287600    -.4314459600       2           12
 1.4000     .1551087100    -.1370142500       2           36
 1.6000    -.2219833100    -.3120721600       2          145
 1.8000    -.3802075700    -.5466494700       2           16
 2.0000    -.5634176900    -.6511868500       2           10
 2.2000    -.7557344200    -.6627526000       2            7
 2.4000    -.9370602400    -.5981720500       2            7
 2.6000   -1.0895784000    -.4720756300       2            4
 2.8000   -1.1990978000    -.3003142600       2            3
 3.0000   -1.2557876000    -.1004546200       2            3
 3.2000   -1.2546596000     .1086739000       2            3
 3.4000   -1.1958127000     .3077998700       2            4
 3.6000   -1.0844279000     .4781295500       2            4
 3.8000    -.9305148000     .6021594000       2            4
 4.0000    -.7484203200     .6641077800       2            4
 4.2000    -.5560831700     .6493629200       2            9
 4.4000    -.3736415800     .5408655200       2            6
 4.6000    -.2163311400     .2996989500       2           18
 4.8000     .1644907100     .1536650600       2          158
 5.0000     .3599660700     .4380612100       2           22
 5.2000     .5528541200     .5557733200       2           10
 5.4000     .7474862800     .5703359600       2            7
 5.6000     .9260668600     .5040750700       2            7
 5.8000    1.0710534000     .3738418600       2            4
 6.0000    1.1671386000     .1964849400       2            4

 FINAL OUTPUT POINT SET FOR PERIOD, Y1,Y2 SHOULD BE THE INITIAL VALUES AGAIN
6.1922    1.2001725000    -.0002010430       2           10

 ERROR TOLERANCES USED   REL.=        .00000304 ABS=        .00000100

 TIME TO COMPUTE ENTIRE ORBIT IS     578 SECONDS

 APPROXIMATE MACHINE UNIT ROUNDOFF IS  2.98E-08
```

FIGURE 2. This test uses North Star 8-disit Floatins Point BASIC

```
 THREE BODY ORBIT PROBLEM

 TEST DRIVER FOR RKF45 - THE ODE SOLVER

    X          Y1                 Y2          FLAG    TIME TO COMPUTE

 .0000    1.2000000000     .0000000000       3            1
 ERROR RESET    REL.=   .0000030396    ABS.=   .0000010000
 .2000    1.1643714000    -.2042720000       2            6
 .4000    1.0661772000    -.3801237700       2            4
 .6000     .9196411200    -.5082030300       2            3
 .8000     .7401472600    -.5717005800       2            5
1.0000     .5453134600    -.5537131100       2            4
1.2000     .3528291200    -.4314462400       2            7
1.4000     .1551093000     -.1370150900      2           19
1.6000    -.2219831400     -.3120707100      2           77
1.8000    -.3802072500     -.5466484200      2            9
2.0000    -.5634170300     -.6511860200      2            6
2.2000    -.7557334400     -.6627520000      2            4
2.4000    -.9370589300     -.5981717200      2            3
2.6000   -1.0895767000     -.4720756300      2            2
2.8000   -1.1990958000     -.3003146600      2            2
3.0000   -1.2557853000     -.1004555200      2            3
3.2000   -1.2546571000      .1086723800      2            2
3.4000   -1.1958100000      .3077976100      2            2
3.6000   -1.0844251000      .4781263900      2            3
3.8000    -.9305119500      .6021551600      2            2
4.0000    -.7484175600      .6641021800      2            2
4.2000    -.5560806200      .6493555400      2            5
4.4000    -.3736391600      .5408554500      2            4
4.6000    -.2163277100      .2996831000      2           10
4.8000     .1644995300       .1536849000     2           85
5.0000     .3599697000      .4380718400      2           13
5.2000     .5528566400      .5557806300      2            6
5.4000     .7474879100      .5703413600      2            4
5.6000     .9260675100      .5040793200      2            4
5.8000    1.0710530000      .3738455600      2            2
6.0000    1.1671371000      .1964886700      2            2

 FINAL OUTPUT POINT SET FOR PERIOD, Y1,Y2 SHOULD BE THE INITIAL VALUES AGAIN
6.1922    1.2001701000    -.0001968430       2            5

 ERROR TOLERANCES USED   REL.=        .00000304 ABS=        .00000100

 TIME TO COMPUTE ENTIRE ORBIT IS     325 SECONDS

 APPROXIMATE MACHINE UNIT ROUNDOFF IS  2.98E-08
```

FIGURE 3. This test uses North Star 14-digit BASIC

THREE BODY ORBIT PROBLEM

TEST DRIVER FOR RKF45 - THE ODE SOLVER

X	Y1	Y2	FLAG	TIME TO COMPUTE
.0000	1.2000000000	.0000000000	2	1
.2000	1.1643714287	-.2042720051	2	17
.4000	1.0661771200	-.3801237610	2	11
.6000	.9196414390	-.5082029843	2	15
.8000	.7401479199	-.5717006072	2	11
1.0000	.5453139066	-.5537132612	2	10
1.2000	.3528295073	-.4314465623	2	20
1.4000	.1551094518	-.1370155770	2	58
1.6000	-.2219799345	-.3120709124	2	275
1.8000	-.3801990041	-.5466463189	2	26
2.0000	-.5634023239	-.6511819430	2	16
2.2000	-.7557103665	-.6627471274	2	11
2.4000	-.9370257153	-.5981681199	2	11
2.6000	-1.0895323322	-.4720760847	2	10
2.8000	-1.1990388421	-.3003228808	2	5
3.0000	-1.2557153833	-.1004759783	2	5
3.2000	-1.2545745335	.1086344425	2	6
3.4000	-1.1957161188	.3077361303	2	6
3.6000	-1.0843223023	.4780343456	2	6
3.8000	-.9304038358	.6020241661	2	15
4.0000	-.7483089221	.6639214405	2	10
4.2000	-.5559748694	.6491084358	2	11
4.4000	-.3735365353	.5405083418	2	20
4.6000	-.2161894464	.2991191218	2	35
4.8000	.1648104437	.1543971447	2	293
5.0000	.3600996071	.4384539399	2	42
5.2000	.5529513550	.5560404133	2	16
5.4000	.7475520769	.5705269731	2	10
5.6000	.9260951585	.5042178217	2	10
5.8000	1.0710383381	.3739592603	2	11
6.0000	1.1670770425	.1965986982	2	15

FINAL OUTPUT POINT SET FOR PERIOD, Y1,Y2 SHOULD BE THE INITIAL VALUES AGAIN

6.1922	1.2000665638	-.0000733951	2	10

ERROR TOLERANCES USED REL.= .00000100 ABS= .00000100

TIME TO COMPUTE ENTIRE ORBIT IS 439 SECONDS

APPROXIMATE MACHINE UNIT ROUNDOFF IS 2.84E-14

FIGURE 4. This test uses North Star 14-digit Floating Point BASIC

```
 THREE BODY ORBIT PROBLEM

 TEST DRIVER FOR RKF45 - THE ODE SOLVER

    X          Y1                 Y2          FLAG    TIME TO COMPUTE

  .0000    1.2000000000     .0000000000       2          1
  .2000    1.1643714287    -.2042720051       2          6
  .4000    1.0661771200    -.3801237610       2          4
  .6000     .9196414390    -.5082029843       2          6
  .8000     .7401479199    -.5717006072       2          4
 1.0000     .5453139066    -.5537132612       2          4
 1.2000     .3528295073    -.4314465623       2          7
 1.4000     .1551094518    -.1370155770       2         22
 1.6000    -.2219799345    -.3120709124       2        101
 1.8000    -.3801990041    -.5466463189       2         10
 2.0000    -.5634023239    -.6511819430       2          7
 2.2000    -.7557103665    -.6627471274       2          4
 2.4000    -.9370257153    -.5981681199       2          4
 2.6000   -1.0895323322    -.4720760847       2          4
 2.8000   -1.1990388421    -.3003228808       2          3
 3.0000   -1.2557153833    -.1004759783       2          3
 3.2000   -1.2545745335     .1086344425       2          3
 3.4000   -1.1957161188     .3077361303       2          3
 3.6000   -1.0843223023     .4780343456       2          3
 3.8000    -.9304038358     .6020241661       2          6
 4.0000    -.7483089221     .6639214405       2          5
 4.2000    -.5559748694     .6491084358       2          5
 4.4000    -.3735365353     .5405083418       2          8
 4.6000    -.2161894463     .2991191218       2         13
 4.8000     .1648104437     .1543971447       2        108
 5.0000     .3600996071     .4384539399       2         16
 5.2000     .5529513550     .5560404133       2          6
 5.4000     .7475520769     .5705269731       2          4
 5.6000     .9260951585     .5042178217       2          4
 5.8000    1.0710383381     .3739592603       2          5
 6.0000    1.1670770425     .1965986982       2          6

 FINAL OUTPUT POINT SET FOR PERIOD, Y1,Y2 SHOULD BE THE INITIAL VALUES AGAIN
6.1922    1.2000665638    -.0000733951       2          5

 ERROR TOLERANCES USED   REL.=        .00000100 ABS=        .00000100

 TIME TO COMPUTE ENTIRE ORBIT IS     405 SECONDS

 APPROXIMATE MACHINE UNIT ROUNDOFF IS  2.84E-14
```

Part 2. Specific support for the routines in SCRUNCH

Part 2 supports the individual numerical routines in SCRUNCH. Since each routine has an initial set of remarks detailing specific implementation considerations, Part 2 avoids duplicating this information and the user is asked to read these remarks for more details, i.e. exact definitions of variables used and all dimensioning requirements which the user must take care of in his driver. For each routine in the package, the following information is specified in Part 2:

1. Input and output parameters. There are many variables used internally by each routine, and the user is cautioned to avoid using any of the internal variables in writing a driver.

2. The numerical method employed.

3. Any user-written auxilliary routines required.

4. A brief outline of how to use the routine. As in (1) above, the user may need to consult the initial remarks of the routine for a description of alternate flag returns from the routine and their meaning.

Numerical solution of a system of first order differential equations.

subroutine RKF45 input (X,Y,N,X0,E0,E1,G1)
output (X,Y,E0,E1,G1)

The original FORTRAN version of RKF45 was written by

H. A. Watts and L. F. Shampine
Sandia Laboratories
Albuquerque, New Mexico 87185

and is presented in Chapter 6 of [1].

RKF45 uses two Runge-Kutta formulas developed by E. Fehlberg. One is a fourth order formula, the other a fifth order formula. Using them together allows an error estimate to be computed which the code can use to adjust its step-size. This method is intended to be used on non-stiff, or mildly stiff differential equations. Should the differential equations become too stiff for RKF45 to handle, a built in test will detect this, set a flag and return control to the user. Several other tests are built into the code. Estimated error at each step is used to adjust the step-size for the next step.

The user must write a subroutine of the form:

subroutine yprime input (X1,Y1) output (Y2)

to evaluate Y2 = F (X1,Y1) which specifies the right hand side of the system of first order differential equations.

To use RKF45, the user need only:

0. dimension the N-vectors Y,Y0,Y1,Y2,K1,K2,K3,K4,K5
1. initialize the number of differential equations, N
2. specify the initial point, X, and initial function values in the N-vector, Y
3. specify relative and absolute error tolerances, E0 and E1, respectively.
4. set the initialization flag, G1=1.
5. tell the code how far to advance the solution by setting X0.
6. GOSUB 1000

If G1=2 on return, indicating that everything went well, then X and Y have been overwritten with the updated solution values and the user need only set the end point, X0, to some point further along in the numerical solution and GOSUB 1000 again. (All other input parameters have been set on return by RKF45 in order to continue integration).

Numerical integration of a function of one real variable.

subroutine SIMP input (T1,T2,A0) output (Y0,Y1,A1,A2)

The original FORTRAN version of SIMP was written by

Lawrence F. Shampine and Richard C. Allen, Jr.
Numerical Computing: an introduction
W. B. Saunders Company 1973

The underlying integration method is Simpson's rule which integrates the quadratic polynomial through three points to give:

$$\int_{x=a}^{x=b} f(x)\,dx = \frac{b-a}{6} * \left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right) .$$

Simpson's rule has an error expression on the order of the length of the interval, b-a, to the fifth power and therefore isn't accurate enough to apply directly to a standard quadrature problem. The additive property of integration can be used to split the interval up into many smaller intervals, on which Simpson's rule can give good results. Notice the error made on each of the intervals can be made small by simply making the interval small.

The code SIMP is an iterative, adaptive implementation of the composite Simpson's rule. The iterative part gives the code an estimate of the error made by first integrating over a small subinterval, then dividing this subinterval into 2 sub-sub-intervals and integrating over each separately. Adding the results from integrating the two sub-sub intervals gives a more accurate answer than the original subinterval, and their difference gives an estimate of the numerical error. Since the iteration step can be used to estimate the error being made on any of the subintervals, the code knows how well it is doing and if, at some particular point in the integration process, smaller subintervals need to be used to get the desired user-specified accuracy, the code detects this and further refines the grid used only in the vicinity of the trouble spot. Where the integration can proceed successfully using the larger subinterval, this larger one will be used. Therefore, the code is adaptive, locally changing the grid size to accomodate the error requested by the user.

The user must write a defined-function

DEF FNY (x) to evaluate the integrand

To use SIMP:

0. Dimension the 30-vectors L,H1,H2,H3,H4,H5,H6,H7,H8 and the 5-vector L1
1. Define the interval of integration (T1,T2)
2. Specify the error you are willing to tolerate A0
3. GOSUB 1000
4. If the code sets flag A2=1, then the integral is in Y0

Optimization of a function of one real variable.

function FNM input (G0,G1,E0)

The original FORTRAN version of the function FNM is from

Computer Methods for Mathematical Computations
by George E. Forsythe, Michael A. Malcolm
and Cleve B. Moler Prentice-Hall, 1977

The original routine was written in ALGOL by R. P. Brent and was presented in his book

Algorithms for minimization without derivatives
by R. P. Brent
Prentice-Hall, 1973

The function FNM attempts to find a global minimum of a unimodal function of a single real variable, given an initial interval in which the minimum must lie. Let XBAR be where f attains its minimum. A unimodal function is one which is strictly decreasing for X <= XBAR and strictly increasing for X >= XBAR. It need not be smooth. The method of search employed uses a combination of a golden-section search and, once the interval of uncertainty has been reduced, successive parabolic interpolation. FNM is an interval oriented code that continually shrinks the interval of uncertainty (G0,G1) until the following holds at x:

abs (x-gmid) <= 2.*U0*abs(x) - (G0-G1)/2. gmid = (G0+G1)/2.

The user must write a defined-function

DEF FNF (x)

to evaluate the function whose minimum is sought.

To use FNM:

1. Specify search interval (G0,G1)
2. Specify allowable error E0
3. The value returned by the function FNM is the location of the minimum.

Note: There are no flags to indicate how well the function did because it is guaranteed to find a global minimum for unimodal functions. If the user-provided function is not unimodal, then FNM will give at least the value of a local minimum of the function. The user should check the value returned by FNM, and if it is equal to one of the original end points, G0 or G1, then it is likely that a different initial interval should be provided by the user on which the function is more nearly unimodal, and call in FNM again.

Root finder for non-linear equations

subroutine ZEROIN input (V1,V2,C1,C2) output (V1,G0)

The original FORTRAN version of ZEROIN was presented in

Numerical Computing: an introduction
by L. F. Shampine and R. C. Allen, Jr.
W. B. Saunders Company, 1973

The method employed by ZEROIN is a combination of bisection and the secant rule. The initial user-supplied interval (V1,V2) is repeatedly shortened to a small interval (V1,V2) by using the secant rule whenever possible, but resorting to bisection anytime secant generates a spurious point. At all times, the interval (V1,V2) will contain the root. This shortening of the interval of uncertainty is continued until the following stopping criterion is met:

$$\frac{\text{abs}(V1-V2)}{2.} \leq C2 * \text{abs}(V1) + C1$$

The final interval of uncertainty (V1,V2) in which the root must lie has collapsed to a size specified by the user in the relative error tolerance, C2, and the absolute error tolerance, C1. The best estimate of the root is placed in V1 by the code.

The user must write a defined-function

```
DEF FNF (x)
```

to evaluate the function whose root is sought.

To use ZEROIN:

1. Specify initial interval of search (V1,V2)
2. Specify absolute error tolerance C1
   and relative error tolerance C2
3. GOSUB 1000
4. If G0=1 (or possibly if G0=2) then the root is in V1

Numerical interpolation using cubic splines

subroutine SPLINE input (XO,FO,N) output (BO,CO,DO)

function FNS input (x)

The original FORTRAN version of SPLINE and FNS (SEVAL) are from:

Computer Methods for Mathematical Computations
by George E. Forsythe, Michael A. Malcolm and Cleve B. Moler
Prentice-Hall, 1977

The subroutine SPLINE uses the N data points given in the vectors, XO and FO, to construct the cubic spline interpolating through these N points. This cubic spline is defined by the three N-vectors of coefficients, BO, CO, DO in the following manner:

$$S(x) = FO_i + BO_i\,(x-XO_i) + CO_i\,(x-XO_i)^2 + DO_i\,(x-XO_i)^3$$

where the index i defines the interval in which x lies by:

$$XO_i <= x <= XO_{i+1}$$

Once the subroutine SPLINE has constructed the coefficients and placed them into the vectors BO, CO and DO, the defined-function FNS can be repeatedly called upon to evaluate the cubic spline S(x) for any value of x.

To use SPLINE and FNS:

0. Dimension the N-vectors XO,FO,BO,CO,DO
1. Initialize N and the N data points (XO,FO)
2. GOSUB 1000 to get spline coefficients
3. Invoke FNS (x) for any value of X desired.

Note: Normally the user will go through steps 1 and 2 only once and repeatedly use the defined-function FNS for different arguments.

Numerical solution of systems of linear equations

subroutine DECOMP input (A,N) output (A,K0,I0)

subroutine SOLVE input (A,I0,W0,N) output (W0)

The original FORTRAN versions of DECOMP and SOLVE are from:

Computer Methods for Mathematical Computations
by G. E. Forsythe, M. A. Malcolm and C. B. Moler
Prentice-Hall, 1977

The subroutines DECOMP and SOLVE are used to solve the system of linear equations

$$A X = B \quad .$$

The subroutine DECOMP forms the LU decomposition of the N by N matrix A using Gaussian elimination with partial pivoting. The subroutine SOLVE can then be used to find the N-vector of unknowns, X, corresponding to a given right hand side, B.

To use DECOMP and SOLVE:

0. Dimension the N-vectors W0,I0 and the N by N array A
1. Initialize N and the N by N array A
2. GOSUB 1000
3. Check condition number K0.
   (A rough test is don't call SOLVE if K0 > 1./U0)
4. Put the right hand side in W0
5. GOSUB 2250 (SOLVE puts the answer in W0)

In general, one goes through steps 1,2,3 once and repeatedly through steps 4 and 5.

Linear equations, continued.

The subroutine DECOMP decomposes the N by N input matrix A into its LU factorization using Gaussian elimination with partial pivoting. The upper triangular matrix U is stored in the upper triangular part of the original matrix A. Information about a permuted version of a lower triangular matrix I - L is is stored in the lower triangular portion of the original A, and the pivot information is stored in the N-vector IO. This computation is done in such a manner that

(permutation matrix based on IO) * A = L * U

Gaussian elimination with partial pivoting is a very stable first-step for solving linear systems.

A by-product of the decomposition is KO, which is an estimate of the condition of the matrix A. Condition of a matrix corresponds to how nearly singular the matrix is, and relates how errors in the original data will be propagated in the numerical solution. The user should consult section 3.2 of Forsythe, Malcolm and Moler for a full discussion of conditioning of a matrix. A rough rule to use is be wary of using SOLVE if KO > 1./UO, where UO is the machine's approximate unit roundoff.

Once the matrix A has been decomposed by DECOMP, the subroutine SOLVE can be used to find the solution vector, X, given a right hand side, B.

The manner in which SOLVE goes about this employs the LU decomposition of A to make finding the unknowns a relatively simple computation.

Given a right hand side, B, and the decomposition, LU:

solve the triangular system: L C = B for the n-vector C

then solve the triangular system: U X = C for the n-vector X

Solving triangular systems is simple, since it only involves back substitution.

The n-vector X now is the solution sought, since

A X = L (U X) = L C = B

Least squares fit to a linear model equation

subroutine HECOMP input (A,M,N) output (A,U,G1)
subroutine HOLVE input (A,U,M,N,B) output (B)

The original FORTRAN version of HECOMP and HOLVE is from

Matrix Eigenvalue and Least Squares Computations
by Cleve B. Moler
Computer Science Department Course Notes
Stanford University, March, 1974

An alternate reference for the numerical linear algebra is:

Introduction to Matrix Computations
by G. W. Stewart
Academic Press, 1973

The linear model is specified by

$$y(t) = x_1 f_1(t) + x_2 f_2(t) + \ldots + x_N f_N(t)$$

where the $f_i(t)$ are continuous functions. Given the data:

$$(t_i, y_i),\ i=1,2,\ldots,M$$

the least squares solution for the unknowns x_i, i=1,..,N of the linear model is gotten from solving

$$A\dot{X} = B \qquad \text{where } A = (a_{ij}) = (f_j(t_i)) \qquad \text{and } B = (b_i) = (y_i),\ i=1,\ldots,M;\ j=1,\ldots,N.$$

Chapter 9 of [1] gives many more details on how to set up a linear model, i.e. the construction of the design matrix A and the definition of the right hand side data vector, B. The routine HECOMP will decompose the M by N (with M > N) design matrix, A, using Householder reflections. The routine HOLVE can then be used to find the coefficient vector which for a given right hand side data vector, B, best fits the model, i.e.

minimize the sum of squares of the components of $A\dot{X} - B$.

To use HECOMP and HOLVE:

0. Dimension the M-vectors B,U and the M by N array A
1. Initialize M,N and the M by N array A
2. GOSUB 1000 (to decompose A)
3. If G1<>1, then set up right hand side, B
4. GOSUB 1760 (HOLVE puts unknowns in first N components of B)
5. Compute residual of fit, which is the sum of squares of the remaining M-N components of B.

Least squares fits using Householder reduction, continued.

The following reference gives many details on solving least squares problems. Chapter 19 presents operations counts for Householder reduction vs. forming and solving the normal equations $(A^T A)\, X = A^T B$ and comes to the conclusion that the numerically stable orthogonal Householder reduction takes twice as many operations, but the normal equations must be solved using far more precision (about twice as many significant digits) to even compare with HECOMP. Therefore, the safe general method to use is HECOMP.

Solving Least Squares Problems
by Charles L. Lawson and Richard J. Hanson
Prentice-Hall 1974

Following is a brief justification (taken directly from Cleve Moler's notes) of the algorithm used in his routines. Theorem 4.1 is stated with proof since it actually constructs the decomposition of the design matrix A. Corollary 3.1 is without proof, and can be taken as a definition of a Householder reflection. (Normally one defines the Householder reflection in this way, and then proves that it has the desired properties).

Moler's Corollary 3.1: Let $a = (a_1, a_2, \ldots, a_M)^T$ be any vector.

For any k=1,...,M, define

$$alfa = sign(a_k) * sqrt(a_k^2 + \ldots + a_M^2)$$

$$u = (0, \ldots, 0, a_k + alfa, a_{k+1}, \ldots, a_M)^T$$

$$beta = alfa * u_k = alfa * (alfa + a_k)$$

then $P = I - (1/beta)\, u\, u^T$ is a Householder reflection (and therefore orthogonal and symmetric) which has the following effects on the original vector a:

$$P\, a = (a_1, \ldots, a_{k-1}, -alfa, 0, \ldots, 0) \quad .$$

Furthermore, if b is any other vector, define

$$gamma = (u^T b)/beta \quad .$$

Then $P\, b = b - gamma * u$.

In particular, if $b_k = \ldots = b_M = 0$, then $P\, b = b$.

Least squares fits using Householder reduction, continued.

Moler's Theorem 4.1: Let A be any M by N matrix with M>N. Then there is an orthogonal matrix Q, which is a product of Householder reflections

$$Q = P_N \ldots P_2 P_1$$

so that

$$Q A = R$$

is upper triangular.

Proof. The proof is the outline of an algorithm for actually computing $P1, P2, \ldots, PN$ and R. Let $A1 = A$ and let a_{*1} be the first column of A1. Use Cor. 3.1 to produce a Householder reflection P1 so that $P1\, a_{*1}$ is a multiple of e1, the first unit vector. Let $A2 = P1\, A1$. (Because of the special structure of P1, this does not involve a complete matrix multiplication. Only 2*N*N multiplications are needed, not N*N*N.) Now let a_{*2} be the second column of A2. By Cor. 3.1, there is a Householder reflection P2 which zeroes all but the first two components of a_{*2} and which does not alter the first column of A2. Let $A3 = P2\, A2$. Continuing in this way, after N-1 steps we have a matrix

$$AN = PN\text{-}1 \ldots P1\, A$$

which is upper triangular except for its last column. Let a_{*N} be this last column. Let PN be the Householder reflection which zeros that last M-N components of a_{*N} and which does not alter any of the previous columns. Finally, we let

$$R = AN{+}1 = PN\, PN\text{-}1 \ldots P1\, A$$

Singular value decomposition of an M by N matrix A

subroutine SVD input (A,M,N,G2,G3) output (W,U if G2=1,V if G3=1,G1)

The original FORTRAN version of SVD is presented in:

Computer Methods for Mathematical Computations
by G. E. Forsythe, M. A. Malcolm and C. B. Moler
Prentice-Hall, 1977

The subroutine SVD is an extremely powerful numerical tool that has a wide variety of applications in the field of over-determined linear equations and least squares data fitting. It decomposes the M by N matrix A into

$$A = U\,W\,V^{T}$$

where U,V are orthogonal matrices of order M and N, respectively, and W is an M by N upper diagonal matrix with all zeros in the lower block. The theory behind the SVD is presented in the above reference, but more details on the power of this routine can be obtained from:

Solving Least Squares Problems
by Charles L. Lawson and Richard J. Hanson
Prentice-Hall, 1974

and

Applied Linear Algebra - second edition
by Ben Noble and James W. Daniel
Prentice-Hall, 1977

It is recommended that the user look into these two references since no attempt is made here to develop the theory of the singular value decomposition.

To use SVD:

0. Dimension the N-vectors W,WO and the M by N matrix A. If the transformation matrices are desired, then dimension V to be N by N and U to be M by N.
1. Initialize M,N and the M by N matrix A
2. Set G2=1 if the matrix U is desired.
   Set G3=1 if the matrix V is desired.
3. GOSUB 1000
4. If G1=0 then the singular values are in W and the orthogonal transformation matrices U,V are set if requested.

Singular value decomposition, continued.

The following theorem defines the significance of W, U and V, and reveals the power of the SVD. (A detailed discussion of applications of SVD may be found in the book by Lawson and Hanson).

Theorem 9.7 (Singular Value Decomposition) (Noble & Daniel, p.327)

Let the M by N matrix A have rank K. Then there exist numbers

$$W_1 >= W_2 >= \ldots >= W_K > 0$$

the singular values, an M by M unitary matrix $U = (U_1,\ldots,U_M)$, and an N by N unitary matrix $V = (V_1,\ldots,V_N)$, such that

$$A = U\,W\,V^T$$

where W is the M by N matrix defined by

$$W = \begin{pmatrix} WI & 0 \\ 0 & 0 \end{pmatrix}$$

Note: The 0 block matrices may be null depending on the relative sizes of M, N and K.

and WI indicates the K by K diagonal matrix with i-th entry, W_i.

Moreover, for $1 <= i <= k$, $U_i = W_i^{-1} A V_i$, and $V_i = W_i^{-1} A^T U_i$, are eigenvectors of $A\,A^T$ and $A^T A$, respectively, both associated with the eigenvalue $W_i^2 > 0$.

Singular value decomposition, continued.

Three uses of the singular value decomposition:

1. One sees from the above theorem, that SVD produces the rank of an arbitrary M by N matrix which is of great value when trying to develop a linear model of some particular system to use for least squares data fitting.

2. The SVD can be used to solve the least squares problem

$$A X = B$$ where A is M by N, B is an M-vector and X is an N-vector.

a. Let $A = U W V^T$.
b. Apply U-transpose to original data vector, B
c. Divide the i-th component $U^T B$ by the i-th singular value, W_i.
d. Apply V to the resulting vector to obtain

$$X = V W^+ U^T B$$

where W^+ is the Moore-Penrose inverse defined below.

3. The Moore-Penrose inverse, A^+, of an M by N matrix can be obtained from the SVD using

$$A^+ = V W^+ U^T$$

where

$$W^+ = \begin{bmatrix} WI & 0 \\ 0 & 0 \end{bmatrix}$$

and the K by K diagonal matrix WI, is made up of the W_i^{-1}'s (the reciprocals of the singular values of A).

Numerical computation of eigenvalues and eigenvectors for a real symmetric matrix

subroutine SYMEIG input (A,N,F1) output (L,X0 if F1=1,A)

The original FORTRAN version of SYMEIG is presented in

Matrix Eigenvalue and Least Squares Computations
by Cleve B. Moler
Course Notes from Computer Science Department of
Stanford University, March 1974

An alternate reference for the numerical linear algebra is

Introduction to Matrix Computations
by G. W. Stewart
Academic Press, 1973

Since the user will probably not have access to the original documentation contained in the Course Notes, an attempt will be made here to define the algorithm employed by this subroutine. The user should read the documentation for HECOMP and HOLVE first, since some of the terms used here are developed there.

SYMEIG will find the eigenvalues, and optionally the eigenvectors, of a real symmetric N by N matrix A.

To use SYMEIG:

0. Dimension the N-vectors L,W0 and the N by N matrix A. If the eigenvectors are desired, then dimension X0 to be N by N.
1. Initialize N and at least the lower triangular part of the N by N matrix A
2. Set F1=1 if you want the eigenvectors
3. GOSUB 1000
4. The eigenvalues are in L
5. If F1=1 then the eigenvectors are in X0. The k-th column of X0, i.e. X0(*,k) corresponds to L(k)

Symmetric eigensystem, continued.

The first step in the symmetric eigensystem problem is to transform the symmetric N by N input matrix, A, to a tridiagonal form using Householder reflections. Moler's Algorithm 6.1 replaces the matrix A with the tridiagonal matrix, A1, where

$$A1 = P_{N-1} P_{N-2} \cdots P_2 \, A \, P_2 P_3 \cdots P_{N-1}$$

where the Householder reflection matrices, P_i, i=2,...,N-1 are not explicitly computed as matrices, but the effect of their action on the original A is produced and the scalar beta and N-vector u which define P are stored in the upper triangular part of A. Note these P_i, i=2,...,N-1 are the same Householder reflections that would be generated by HECOMP in reducing A to right triangular form.

If the user has specified F1=1 on input, indicating that the eigenvectors are desired, the next step is to form the matrix X0 which will contain the eigenvectors. This matrix is simply

$$X0 = P_2 \cdots P_{N-1}$$

which is recovered from information that was stored in the upper triangular portion of the matrix A as the tridiagonal transformation was being performed.

The third step in SYMEIG is the tridiagonal QR algorithm to find the eigenvalues of the tridiagonal matrix A1, above. The tridiagonal QR algorithm produces a sequence of orthogonal matrices $Q_1,\ldots,Q_{N-1}$ so that the matrix

$$Q_{N-1} \cdots Q_1 \, A1 \, Q_1 \cdots Q_{N-1}$$

is a tridiagonal matrix of the form:

$$\begin{matrix} l_1 & e_2 & 0 & 0 & \cdots & 0 \\ e_2 & l_2 & e_3 & 0 & \cdots & 0 \\ 0 & e_3 & l_3 & e_4 & \cdots & 0 \\ & \ddots & \ddots & \ddots & & \ddots \\ & & \ddots & \ddots & \ddots & \ddots \\ 0 & \cdots & & & e_N & l_N \end{matrix}$$

where the e_i's are neglible, indicating that the l_i's are eigenvalues.