Ps_Quasi, an ODE BVP Solver

November 12, 2000
22:10

Contents

Bart Childs
Computer Science – Texas AM University

Abstract. *This code is for the solution of multipoint boundary value problems in ordinary differential equations. The solution process is often called a "shooting method" but we prefer to call it a "superposition method." Nonlinear problems are solved through a Newton-like linearization. The method of integration is based upon the Fröbenius method for determining the Taylor series coefficients.*

Printed September 1995

1 Some fweb preliminaries

This program is written to run under the Cray `cft77` and the usual compilers for *SUN*s. This is controlled by the macros *CRAY* and *SUN*. It should be straightforward to accomodate other environments such as *VAX* VMS and ...

The code creates one large file in its current form. The source can be modified to defeat the two `WEB` includes. The includes are placed just before the last few modules that are the ones most likely to be changed with each application. This is done to place that code next to the index. There are companion files, `ps_subs.web` and `gjrwls.web` which also input the included files if the user wishes to keep their listings separately.

The `ftangle` command line to produce *CRAY* code needs two parts that are not defaults: `-m"CRAY" -d`. The `-d` converts **do**/**end do** constructions into labeled **do** loops. Cray, like everybody else, has extended their compiler beyond the standard, but they omitted this logical extension.

We start each FORTRAN module with a command that defeats default typing of variable names. FORTRAN compilers need default variable types (preferably a command line option) for compatibility with old code.

Another macro, **floating**, enables 64 bit precision in all the environments. The Cray uses 64 bit precision for **real** variables by default. The other machines will use this when **real**∗8 is specified. The actual storage may vary among machines. The macro, *const*, is used to convert floating point constants by appending the "`d0`" exponent on 32-bit machines.

The canonical form of this is kept on an IBM RISC SYSTEM/6000 and is compiled using the `xlf` compiler. This is considered stable in its current form and future changes will be to rewrite it using Fortran 90. The current version of `xlf` will be the basis of port.

```
"psq.f" 1 ≡
@#if defined (CRAY)
  @m CRAY 1
  @m SUN 0
@#else
  @m CRAY 0
  @m SUN 1
@#endif

  @f floating real
@#if CRAY
  @m floating real
  @m const(x) x
@#else
  @m floating real∗8
  @m const(x) x##d0
@#endif
```

2 Nonlinear Multipoint Boundary Value Problems

The nonlinear multipoint boundary value problem and the method of solution used in this code are described in reasonable detail in the companion users guide. A short statement of the problem is repeated for the reader's convenience.

The following problem is offered as a sample problem of the type `Ps_Quasi` is intended to solve. It is a model of a *spring mass dashpot, electrical circuit, or other physical systems.* This is a linear problem if the parameters are known.

$$\ddot{x} + \mu\dot{x} + \xi x = \lambda sin(t)$$

The *constants* μ, ξ, and λ are unknown and their estimation is a primary goal. We realize that these parameters will not be a linear function of data like:

t_i	1.0	2.0	3.0	4.0	5.0	6.0	$\cdots$	15.
$\ddot{x}(t_i)$	-0.220	0.035	-0.474	-0.589	0.393	1.597	$\cdots$	-2.816

The values of $x(0)$ and $\dot{x}(0)$) are also not linearly dependent upon the data.

One might consider this to be *Regression Analysis* where the model is a differential equation. A word of caution is that the results cannot be viewed in the same manner as the usual *Regression Analysis.* In these problems we generally have significant detailed knowledge of the model. The model (the ODEs) usually is based upon a sound theoretical or scientific analysis. The usual regression analysis is based on knowing nothing about the model and considering means of (hyper)planar models.

The linear multipoint boundary value problem is to find the solution of the differential equation:

$$\dot{y} = Ly + f \qquad \text{or} \qquad \dot{y} = g(y,t)$$

subject to the m boundary conditions:

$$q_i(y(t_i)) = b_i, \quad \text{for} \quad i = 1, 2, ..., m$$

where:

y is the state vector of n elements,

L is a linear operator that is a variable or constant coefficient matrix,

t is the independent variable, often *time.*

f is an n element vector function of the independent variable t, and

$(\dot{\cdot})$ denotes differentiation with respect to t.

q_i is an operator that defines a linear combination of the elements of the state vector, y, that is equal to the boundary value b_i at $t = t_i$.

2.1 Superposition methods (also known as shooting methods)

This section is a brief exposition on the superposition of particular solutions which is mathematically equivalent to the usual superposition methods. In the homogeneous superposition method the solution of the linear differential equation, $\dot{y} = Ly + f$, was expressed as the weighted sum of n linearly independent homogeneous solutions and a particular solution of this equation. Some recommended sources for discussions of boundary value problems are Osborne, Fox, and Keller. Many different aspects of a wide range of this class of problems are discussed in detail.

The solution can also be written as the sum of $n+1$ particular solutions of the differential equation, given that a certain condition (or constraint) is met. We write:

$$y = \sum_{j=0}^{n} P_{(\cdot,j)}\beta_j = P\beta$$

where:

β_j are superposition coefficients and
P is a matrix whose j^{th} column is denoted by $P_{(\cdot,j)}$.

The j^{th} column of P is a solution of the ODE:

$$\dot{P}_{(\cdot,j)} = LP_{(\cdot,j)} + f \qquad j = 0, 1, 2, \ldots, n$$

We multiply each of these equations by the appropriate superposition coefficient, β_j, and sum over the indicated range which yields:

$$\sum_{j=0}^{n} \dot{P}_{(\cdot,j)}\,\beta_j = L\Big(\sum_{j=0}^{n} P_{(\cdot,j)}\,\beta_j\Big) + f\sum_{j=0}^{n}(\beta_j)$$

Comparing this with equation (3) and its derivative with respect to t, we see that:

$$\sum_{j=0}^{n} \beta_j \equiv 1$$

2.2 Linear independence of the initial value solutions

The equivalent of the non*zero* Wronskian for the superposition of particular solutions is:

$$\operatorname{rank}(P) = n$$

This is ensured if the initial values of P are appropriately chosen. This is easily done by the following strategy. Let the initial conditions for $P_{(\cdot,0)}$ be the current best estimate of the initial conditions. Then, $P_{(\cdot,i)}$ will be the same except for the i^{th} element will be

$$P_{(i,i)} = P_{(i,0)} + \rho_i$$

The only requirement is that each element of $\rho \neq 0$. It is obvious that the $\operatorname{rank}(P) = n$ because the subtraction of the 0^{th} column from all other columns of $P(t = 0)$ yields a matrix whose rightmost n columns form a diagonal matrix whose diagonal elements are ρ_i. The determinant of this $n \times n$ submatrix is $\prod_{i=1}^{n} \rho_i$ and with the assumption these "perturbations" be non-zero, the rank requirement is satisfied.

2.3 Boundary condition specification

The superposition coefficients for this definition are found by substituting into the boundary conditions. The system of equations which must be solved to determine the superposition coefficients are:

$$C\beta = d$$

where the elements of these arrays are:

$$C_{(0,j)} = 1 \quad \text{for } j = 0, 1, 2, \ldots, n$$

$$C_{(i,j)} = q_i\big(P_{(\cdot,j)}(t_i)\big) \quad \text{for} \begin{cases} i = 1, 2, \ldots, m \\ j = 0, 1, \ldots, n \end{cases}$$

$$d_0 = 1$$

$$d_i = b_i \quad \text{for } i = 1, 2, \ldots, m$$

3 Power Series

The power series is one of the most common and important tools in applied mathematics. It often comes from the truncated Taylor series. The solution of ordinary differential equations can be expressed in this form. The usual integration formulae, Runge-Kutta and multistep methods, are based upon the approximation of a Taylor series.

One disadvantage of the Taylor series as an integrator is that the formal manipulation of these power series can often be quite complex and require detailed manual preparation. The procedures furnished in this paper offset this in some cases by giving a process that is more accurate and faster.

3.1 Recurrence equations

The method of Frobenius is an economical means of calculating these coefficients through the use of recurrence equations rather than using the repeated differentiation. We will describe various routines which will expedite some of the formal manipulations necessary to solve a fairly broad subset of the common differential equations. For the differential equation (1):

$$\dot{y} = g(y, t)$$

we assume that the solution y and the right-hand-side can be expanded in power series. Since y is a vector, the coefficient of t^k is a vector and is denoted by $y_{(\cdot,j)}$. Thus:

$$\begin{aligned} y(t) &= \textstyle\sum_{k=0}^{\infty} y_{(k,\cdot)} t^k && (1)\\ g(t) &= \textstyle\sum_{k=0}^{\infty} g_{(k,\cdot)} t^k && (2)\\ \dot{y}(t) &= \textstyle\sum_{k=0}^{\infty} (k+1) y_{(k+1,\cdot)} t^k && (3)\\ & && (4) \end{aligned}$$

The upper limit of these summations is theoretically ∞, but the practical use requires these be finite, say m. Tests of some years ago indicated a value of $m = 12$ was the best to give economical precision. These tests should be run again on the most common computers now available. This should use the algorithms included in this document, be carefully documented, and published.

Substituting the second and third of equations (15) into (14) yields:

$$\sum_{k=0}^{m} (k+1) y_{(k+1,\cdot)} t^k = \sum_{k=0}^{m} g_{(k,\cdot)} t^k$$

This equality requires that coefficients of like powers of t must be equal. Selecting the $(k-1)^{\text{th}}$ term, we get a Frobenius recurrence:

$$y_{(k,\cdot)} = \frac{1}{k} g_{(k-1,\cdot)}$$

3.2 The differential equations, first-order, and linearization

The second order ODE stated in equation (0) can be rewritten as two first order equations:

$$\dot{y}_1 = y_2 \tag{5}$$
$$\dot{y}_2 = -\xi y_1 - \mu y_2 + \lambda \sin(t) \tag{6}$$
$$\tag{7}$$

For these equations, the recurrence (17) can be written:

$$y_{(k,1)} = (y_{(k-1,2)})/k \tag{8}$$
$$y_{(k,2)} = (-\xi y_{k-1,1)} - \mu y_{(k-1,2)} + \lambda sn_{(k-1)})/k \tag{9}$$
$$\tag{10}$$

where sn is the power series representing $\sin(t)$ about the current center of expansion.

The recurrence equations for this identification problem are obviously longer. The notation is confusing at first glance in that some functions are not multiplied by the use of the power series operators. When the constants μ, ξ, and λ are added to the state vector, they remain constants which means their power series have only the zero term that is non-trivial. When an element of y is a constant, only its zero term of the power series will be referenced.

$$y_{(k,1)} = (y_{(k-1,2)})/k \tag{11}$$
$$y_{(k,2)} = (-y_4 y_{k-1,1)} - y_3 y_{(k-1,2)} + y_5 sn_{(k-1)})/k \tag{12}$$
$$\tag{13}$$

Actually, the identification process will require linearized equations which will be based upon the base or approximate solution, w. The linearized form of the equations are:

$$\dot{y}_1 = y_2 \tag{14}$$
$$\dot{y}_2 = w_4 w_1 + w_3 w_2 \tag{15}$$
$$-w_4 y_1 - w_3 y_2 - w_2 y_3 - w_1 y_4 + y_5 \sin(t) \tag{16}$$
$$\tag{17}$$

and the relevant recurrence equations for these linearized equations are:

$$y_{(k,1)} = (y_{(k-1,2)})/k \tag{18}$$
$$y_{(k,2)} = (w_4 w_{k-1,1)} + w_3 w_{(k-1,2)} \tag{19}$$
$$-w_4 y_{k-1,1)} - w_3 y_{(k-1,2)} - w_{k-1,2)} y_3 - w_{(k-1,1)} y_4 + y_5 sn_{(k-1)})/k \tag{20}$$
$$\tag{21}$$

3.3 References to boundary value problem literature

Fox, L., "The Numerical Solution of Two-Point Boundary Problems in Ordinary Differential Equations." O.U.P., 1957.

Kalaba, R., "On nonlinear differential equations, the maximum operation, and monotone convergence," *J. Math. Mech.* **8**, 519, 1959.

Kaplan, W., *Ordinary Differential Equations,* Addison-Wesley, 1958.

Keller, Herbert B., *Numerical Solution of Two Point Boundary Value Problems*, SIAM, 1976.

Osborne, M.R., On Shooting Methods for Boundary Value Problems, *Journal of Mathematical Analysis and Applications*, **27** (1969), 417-433.

Roberts, S.M. and Shipman, J.S., The Kantorovich theorem and two-point boundary value problems. *IBM J. Res. Develop.*. **10** (1966) 402-406.

4 The program, *Ps_Quasi*

The name comes from 'Power Series' and *quasilinearization* which was used by Bellman and Kalaba to denote an approach to solving nonlinear boundary value problems. They developed the method from a dynamic programming approach. It is often referred to as a Newton method. It is a Newton-Raphson-Kantorovich method since most of the Newton expansions are in terms of vectors of functions.

The psuedo-code like view of the program is an eloquent statement of the outline of the program.

"psq.f" 4 ≡

```
program Ps_Quasi
  implicit none

  ⟨ Quasi's parameters 4.1.3 ⟩
  ⟨ Quasi's input variables 4.1.4 ⟩
  ⟨ Quasi's local variables 4.1.7 ⟩
  ⟨ Initialization of variables 4.1.10 ⟩
  ⟨ Input parameters, estimates, and boundary conditions 5.0.1 ⟩
  ⟨ Iterate on the boundary value problem 4.1 ⟩
  stop
end
```

4.1 Iterations, iterations, iterations

The original form of this code did some parts of a "Fisher's scoring" but not a complete one. The complete scoring algorithm requires that a line search be performed at the end of each iteration. Without this line search, there is about 300 lines of code in this iteration part.

I made the decision to reuse most of this code by imbedding a loop with the index *scoring* and checking on its value to implement the complete scoring algorithm. When $scoring \equiv 0$ we do a boundary value iteration and when $scoring \equiv 1$ we do a line search iteration. The variable *iteration* is not updated until the completition of the line search.

⟨ Iterate on the boundary value problem 4.1 ⟩ ≡
```
iteration = 0
do while(iteration ≤ iteration_max)
  do scoring = 0, 1
    ⟨ Set up for this iteration 5.1 ⟩
    ⟨ Perform the forward integration 4.1.1 ⟩
    ⟨ End of forward integration 5.6 ⟩
  end do
end do
```

This code is used in section 4.

Integration of the $n\_ps + 1$ different solutions has several scalar parts and some which can be parallelized. This could pay significant dividends because in an older version of this code it was determined that 93 percent of the CPU utilization was in the forward integration.

⟨ Perform the forward integration 4.1.1 ⟩ ≡
```
do while(¬at_end)
  ⟨ Set up for current center of expansion 5.2 ⟩
  ⟨ Calculate the power series coefficients, recurrence 5.3 ⟩
  ⟨ Determine the limit of numeric convergence 5.2.2 ⟩
  do while((¬at_limit) ∧ (¬at_end))
    ⟨ Determine τ and advance t 5.4 ⟩
    ⟨ The parallel forward integration parts 4.1.2 ⟩
  end do
end do
```

This code is used in section 4.1.

These two parts can each proceed in $n\_ps + 1$ parallel streams in many problems. Of course, there will probably be many problems where this is so quick that the communication costs will offset the gains in parallelization.

⟨ The parallel forward integration parts 4.1.2 ⟩ ≡
```
⟨ Evaluate the power series 5.4.6 ⟩
⟨ Use the result of the evaluation 5.5 ⟩
```

This code is used in section 4.1.1.

In this module, **parameter** is assumed to have its FORTRAN meaning, that is a named constant. The **integer** names $n\_$ and $m\_$ will be central throughout this code. The **parameter** $m\_$ will be a constant and will be the maximum order of the power series that are used. The variable $n\_$ will be the order of the differential equation being solved. **Notice** that the variables n and m have underscores added to their names. This causes them to be long enough that all references to them will appear in the index. It is a feature of WEB **(and probably a good one)** that the use of single character variable names will not appear in the index except at the definition. It was Don Knuth's decision that every use of a dummy index would be too much!

Previous experiments have generally indicated that the value of $m\_$ should be 12. In the current form we are assuming that the value of $n\_$ will be input and will not exceed $max\_n = 8$.

The number of boundary conditions, n_bc, will have an upper limit of $max\_n\_bc = 200$. There is a related variable n_bc_in. This is the total number of boundary conditions input. It normally will be the same as n_bc, but in some cases the user may want to actually use only n_bc of the input values. Thus $n\_bc \leq n\_bc\_in$.

We also need max_n_p as a parameter. This is one more than max_n and is used for dimension information in arrays. The speed of today's compilers makes this a trivial decision.

After this module, the word parameter will have its mathematical meaning. This module is also used in the included file: the_gjrwls.

⟨ Quasi's **parameter**s 4.1.3 ⟩ ≡
integer max_n, $m\_$, max_n_bc, max_n_p, $random\_count$, max_shoot
parameter $(m\_ = 12,\ max\_n = 15,\ max\_n\_p = 16,\ max\_n\_bc = 400)$
parameter $(random\_count = 1000,\ max\_shoot = 2)$

This code is used in section 4.

Now for the declarations of the variables already described.

⟨ Quasi's input variables 4.1.4 ⟩ ≡
integer $n\_$, n_bc_in, n_bc

See also sections 4.1.5, 4.1.6, 4.1.8, 4.1.13, 4.1.15, 5.0.4, 5.0.5, 5.2.4, and 5.4.7.

This code is used in section 4.

The variable *i_debug* is important in testing new applications and in the development of the code itself. This variable will be zero or positive. Positive values will indicate that additional output is to be performed in some kind of debug or checkout mode.

Since there are many levels of this, it is just easier to use an **integer** rather than **logical** and let its level indicate its use.

The variable *iteration_max* is input. However, its value may be changed if the problem is solved in fewer iterations.

The variable *random_shift* enables the user to select a different set of random variables. This is discussed later.

The variable *final* is set to $\mathcal{T}$ whenever *gjrwls* is called in the final iteration and statistics are to be calculated.

A function *number_of_fields* is included to relax the format of input.

⟨ Quasi's input variables 4.1.4 ⟩ +≡
integer *i_debug*, *iteration_max*, *random_shift*, *number_of_fields*
logical *final*

Each boundary condition consists of several parts. We denote the individual boundary condition as $q_i(y(t_i)) = b_i$.

The operator q_i defines the function of y at the time t_i which should equal the boundary value b_i. The boundary value and time are stored in the arrays *bv* and *t_bc*.

An **integer** array q is used to indicate the operator. Most boundary conditions are that an element of y should equal the boundary value. It is not uncommon that a boundary condition be a measurement on an element of $\dot{y}$. When $1 \le q_i \le n$, then the i^{th} boundary condition is on the element of q_i^{th} element of y. If q_i is negative and its absolute value is in the same interval, the boundary condition is on that element of $\dot{y}$.

Boundary conditions may be required to be met exactly or in a best fit sense. We will refer to boundary conditions which are to be met exactly as *constraints* and those to be met in a best fit sense to be *observations*. This specification is indicated by the value of the elements of the array *exact_bc*. A value of one will indicate a best fit desire. This is stored as an **integer** array. Larger positive values are used as an indication of weight.

⟨ Quasi's input variables 4.1.4 ⟩ +≡
floating *t_bc*(1 : *max_n_bc*), *bv*(1 : *max_n_bc*)
integer *q*(1 : *max_n_bc*), *exact_bc*(1 : *max_n_bc*)

Again, n is the order of the differential equations. The variable *n_ps* is the maximum index of the solutions of the differential equations being generated. It has an upper limit of n, but if some initial values are known, then $n\_ps < n$.

We will strive for consistency in the use of variable names, including indices:

i is an index for the different particular solutions which are being superimposed. $0 \le i \le n\_ps$

i_bc is an index for the boundary conditions. $i\_bc \le n\_bc + 1$. The '+1' happens when the range of the independent variable exceeds its range that covers the boundary conditions. Thus, it happens only when 'extrapolating.' The value of $n\_bc + 1$ is not used, only assigned. (We have assumed that $n\_bc \equiv n\_bc\_in$ in the above discussion.)

j is an index into the dependent variable vector, y, the solutions of the differential equations. $1 \le j \le n$. The variable *jj* substitutes for j in some cases.

k is an index into the power series coefficients. $0 \le k \le m$. It is also used as a dummy variable.

iteration is an index of the iterations. $iteration \le iteration\_max$

n_out is a temporary variable that is used when abbreviated output may be advantageous.

i_return is used to check on the status of return flags from subroutines.

scoring is a loop index used to implement a second (sub)iteration for implementation of the line search required by the method of scoring.

scoring_max is the loop range for *scoring*. It is set to unity and reduced to zero when convergence starts slowing down. This requires *convergence_count* to be at least two. The thought behind this is that we may be asking for too much convergence.

convergence_count is the number of iterations for which there has been a strong indication of convergence occurring.

max_ivp is a loop limit that will take on values of *n_ps* when doing the BVP or *unity* when doing the line search for scoring.

⟨ Quasi's local variables 4.1.7 ⟩ ≡
integer *n_ps*, *i*, *iteration*, *j*, *k*, *i_bc*, *jj*, *n_out*, *i_return*
integer *scoring*, *scoring_max*, *max_ivp*

See also sections 4.1.9, 4.1.11, 4.1.14, 4.1.17, 5.0.9, 5.1.1, and 15.2.

This code is used in section 4.

These variables are related to the independent variable.

t is the current value of the independent variable,

t_center is the value of t at the current center of expansion,

tau is defined by $tau = t - t\_center$,

t_limit specifies the maximum value of t that the power series can be expected to give accurate results,

t_output should contain the next value of t where regular output is expected, say for creating plot values,

t_output_delta is the increment of the independent variable for regular output,

t_output_start is the initial value that output is expected for. It may be set to a large value to supress output. The program will reset it to *t_start* at the final iteration if it is larger than *t_stop*,

t_external is similar but is for external purposes, and finally

t_start and *t_stop* specify the range of t that is of interest.

error_norm is used to add errors to the boundary conditions in checking things out.

While determining the value of *tau*, we will set a flag to aid later handling of updating. Each time we output at one of these 'regular points,' we will increment the value of *t_output* by the value *t_output_delta*.

accuracy is set to a value like 10^{-7} to indicate seven (7) significant decimal digits are desired in the integrations,

convergence is similar except that it applies to the precision with which the initial values are estimated, and

max_beta is the value that will be compared to *convergence*.

weight is used in weighting the boundary conditions.

K_n_0, *K_n_1*, and *K_n_2* are variables that are used in the calculation of the value of *line_step* in the scoring algorithm.

line_step_limit is the limiting fraction of the Newton step to take in each iteration. Thus, it is a bound of *line_step*.

⟨ Quasi's input variables 4.1.4 ⟩ +≡
floating *t_start*, *t_stop*, *t_output_start*, *t_output_delta*
floating *y_change_max*, *accuracy*, *convergence*, *max_beta*
floating *error_norm*, *weight*, *K_n_0*, *K_n_1*, *K_n_2*
floating *line_step*, *line_step_limit*

These are local scalar variables. Of course, the input variables just declared are also local and scalar, but those were input while these are assigned values within the code.

There are also several **logical** variables that we use as flags when checking our whereabouts.

The variable *random_numbers* contains 100 random numbers in the interval of zero to unity. These come in handy in checking things out. The scalar *wasted* is appropriately named and used to avoid natting warning messages.

⟨ Quasi's local variables 4.1.7 ⟩ +≡
floating *t*, *t_center*, *tau*
floating *t_limit*, *t_output*, *t_external*, *t_bc_i*
logical *at_limit*, *at_output*, *at_external*, *at_end*, *at_bc*
floating *random_numbers*(*random_count*), *normal_numbers*(*random_count*)
floating *wasted*

The random numbers are furnished in a separate file and input rather than using the local random number generator. This should enable the same results on different machines, nearly.

⟨ Initialization of variables 4.1.10 ⟩ ≡

```
open (9, file = 'random.numbers')
read (9, *) random_numbers
close (9)
open (9, file = 'normal.numbers')
read (9, *) normal_numbers
close (9)
do i = 1, random_count
  if (random_numbers(i) ≥ 0.50000) then
    normal_numbers(i) = −normal_numbers(i)
  end if
end do
```

See also sections 4.1.12, 4.1.16, and 15.3.

This code is used in section 4.

The array *t_ps* is a power series that is a representation of the independent variable t. Thus, most elements will be zero. The 0^{th} element will be set to the value of the independent variable at the current center of expansion. The 1^{st} element is set to unity. This array is often used with the functions supplied to produce the power series that are needed, say the power series that represents $\sin(t)$ about the current center of expansion.

We solve these problems by superposition. We always use only particular solutions. The solution of the differential equation is

$$y = \sum_{i=0}^{n} P_{(\cdot,i)}\beta_i$$

We will need arrays for P and its power series.

The variable P is a two dimensional array. It can be considered a vector of vectors. Recall that FORTRAN arrrays are column major. Each column of the array, P, is a particular solution of the ODE. The 0^{th} column is always the solution of the ODE subject to the current best estimate of the initial values. This is referred to as the 'base' or 'reference' solution in nonlinear problems.

Many of the problems that we solve with codes like this have need of evaluation of the derivative of the power series. The two dimensional array *DP* is used for storing these results.

The three dimensional array, *PS*, is an expansion of each element of P in a power series. The array β is the superposition coefficients.

This structure is needed because when we call the routines for manipulating power series, we want them to see a contiguous set of coefficients.

The vector *beta* will be the superposition coefficients and the matrix *Coef* will be the coefficient matrix in the equation $C\beta = d$ to determine these superposition coefficients. The vector d is not allocated separately, it is an augmented column of the coefficient matrix *Coef* when the routine is called that calculates the superposition coefficients. Upon return from that routine, this column contains the values of β, so it also is not allocated.

The vectors *predicted* and *observed* are used in determining the statistics of the fit.

⟨ Quasi's local variables 4.1.7 ⟩ +≡
floating *t_ps*(0 : *m_*)
floating *P*(1 : *max_n*, 0 : *max_n*), *DP*(1 : *max_n*, 0 : *max_n*)
floating *PS*(0 : *m_*, 1 : *max_n*, 0 : *max_n*)
floating *C*(0 : *max_n_bc*, 0 : *max_n_p*)

We must initialize the power series *t_ps*. Again, it represents t and when the center of expansion moves, the 0^{th} coefficient will change to the center.

There are several lines of this module are *wasted*. They are present to supress a few compiler warnings. Because these lines may change the array *t_ps*, it is re-initialized after we have avoided the compiler warnings by those *wasted* statements.

⟨ Initialization of variables 4.1.10 ⟩ +≡
```
do k = 0, m_
  t_ps(k) = const(0.0)
end do
t_ps(1) = const(1.0)
t_ps(0) = const(1.0)
wasted = ps_sqr(t_ps, 0)
wasted = ps_mult(t_ps, t_ps, 0)
wasted = ps_sqrt(t_ps, 0, t_ps)
wasted = ps_div(t_ps, t_ps, 0, t_ps)
wasted = ps_exp(t_ps, 0, t_ps)
wasted = ps_pwr(t_ps, const(1.0), 0, t_ps)
do k = 0, m_
  t_ps(k) = const(0.0)
end do
t_ps(1) = const(1.0)
```

The best estimate of the initial values are input into *y_input*. This is saved to be used in comparing the final results and initial estimates. Each initial value is also accompanied by data fields *exact_iv*, *min_iv*, and *max_iv*.

exact_iv is an integer array. Negative values indicate the initial value is exact and not allowed to change. Zero values indicate the initial value is to be estimated and there are no bounds to its value. Positive values indicate the value is to be constrained between the values of *min_iv* and *max_iv*.

min_iv and *max_iv* are limits to the calculated estimates of initial values. It is expected that these will normally be a very small positive and very large positive value for parameters such as mass, damping coefficients, ... where negative values would be physically impossible.

⟨ Quasi's input variables 4.1.4 ⟩ +≡
```
floating y_input(1 : max_n), min_iv(1 : max_n), max_iv(1 : max_n)
integer exact_iv(1 : max_n)
```

In linear problems, the results should be available in one iteration. In those cases we really don't need to save the initial values. In nonlinear problems, we save the initial values for the current iteration. This is used to calculate the initial values for the next iteration.

It is also convenient to save the perturbations of the initial conditions. We could have repeated the algorithm for calculating the *perturbation*s, it is just easier to store them when we calculate them. These are used when we perform the superposition to calculate the initial values for the next iteration. The array *perturbation* was denoted by ρ in the module 4.

The variable *i_r* is used in adding noise to the boundary values.

The variables *i_0*, *i_1*, and *i_2* are convenient little scratch variables that are used at least once, maybe more.

⟨ Quasi's local variables 4.1.7 ⟩ +≡
floating *y_initial*(1 : *max_n*, 0 : *max_shoot*)
floating *perturbation*(1 : *max_n*, 0 : *max_shoot*)
floating *y_change*(1 : *max_n*), *y_change_norm*
integer *i_r*, *i_0*, *i_1*, *i_2*

The **format**s are used to make the output more readable.

⟨ Quasi's input variables 4.1.4 ⟩ +≡
character *output_flag*∗6
character *format_t_y*∗32, *format_t_0*∗32, *format_C*∗32
character *format_bc*∗32, *format_iv*∗32, *format_a*∗32

Give values to the **format**s. The output of the 'solutions' is done in a 'hanging indent' manner through the use of tab, `T`, positional editing and imbedded parentheses.

⟨ Initialization of variables 4.1.10 ⟩ +≡
output_flag = '␣␣␣␣␣>␣'
format_t_y = '(6X,F6.2,(T14,6G12.5))'
format_t_0 = '(␣A,F6.2,(T14,6G12.5))'
format_a = '(A)'
format_C = '(1X,6G14.7/(3X,6G14.7))'
format_bc = '(1X,2I5,1X,2G12.5,I5)'
format_iv = '(1X,2I5,1X,3G12.5)'

There are a number of variables that are needed. The types of the functions that are defined in the file `the_subs.hweb`, a separate `WEB`, must be declared. We also declare the built-in functions that are likely to be used in this program.

⟨ Quasi's local variables 4.1.7 ⟩ +≡
floating *ps_sqr*, *ps_sqrt*, *ps_mult*, *ps_div*, *ps_exp*, *ps_pwr*
floating ***log***, ***abs***, ***exp***, ***min***

5 The input routines

This is still in a rather primitive form. This code should have a companion code that also inputs this data and does extensive analysis of it to ensure that it is consistent.

This module serves to delimit the different types of input that the code needs. The last module referenced is used to reduce the order of the problem, if possible, based upon the nature of the data.

⟨ Input parameters, estimates, and boundary conditions 5.0.1 ⟩ ≡
⟨ Input alphanumeric information 5.0.2 ⟩
⟨ Input **integer** parameters 5.0.3 ⟩
⟨ Input **real** parameters 5.0.6 ⟩
⟨ Input boundary conditions 5.0.7 ⟩
⟨ Input initial value estimates 5.0.10 ⟩
⟨ Input formats 5.0.11 ⟩
⟨ Reduce the order of the problem, if possible 5.0.12 ⟩

This code is used in section 4.

This is simply a set of records that will be output at each reasonable opportunity. Obviously, we would output it at the beginning of each iteration to identify the output.

⟨ Input alphanumeric information 5.0.2 ⟩ ≡

This code is used in section 5.0.1.

The first value that should be input is obviously n. Actually we will calculate it. This code is primarily designed for the parameter estimation problem. We might have a second order ODE with two unknown parameters. This makes the order of the problem four. We will input values for *n_ode* and *n_parameters*. The sum of these quantities is the value of n for this problem. The other variables input in this module have been declared and described.

```
⟨ Input integer parameters 5.0.3 ⟩ ≡
    open (9, file = 'data_quasi')
    read (9, *) n_ode, n_parameters
    n_ = n_ode + n_parameters
    n_ps = n_
    write (*, *) ' n_ , n_ode, n_parameters = ', n_, n_ode, n_parameters
    read (9, '(a)') scratch
    i = number_of_fields(scratch)
    if (i ≡ 1) then
      read (scratch, *) n_bc_in
      n_bc = n_bc_in
    else if (i ≡ 2) then
      read (scratch, *) n_bc_in, n_bc
    else
      write (*, *) 'Error inputting n_bc .. ' ‖ scratch(1 : 40)
    end if
    read (9, *) iteration_max
    read (9, *) i_debug
    read (9, *) random_shift
    read (9, format_a) scratch
    write (*, *) n_bc_in, n_bc, iteration_max, ' n_bc_in, n_bc, max_iter'
    write (*, *) i_debug, ' debug level'
```

This code is used in section 5.0.1.

We need to declare those parameters we just input.

```
⟨ Quasi's input variables 4.1.4 ⟩ +≡
    integer n_ode, n_parameters
```

We also declare two scratch variables. These receive values during input and are then discarded. Future revisions may make extensive use of the temporary input of alphanumeric strings.

```
⟨ Quasi's input variables 4.1.4 ⟩ +≡
    integer i_s
    character scratch∗80
```

The **real** parameters need input as well. These have already been declared and described.

⟨ Input **real** parameters 5.0.6 ⟩ ≡

```
read (9, *) t_start
read (9, *) t_stop
read (9, *) t_output_start
read (9, *) t_output_delta
read (9, *) y_change_max
read (9, '(a)') scratch
i = number_of_fields(scratch)
if (i ≡ 1) then
  read (scratch, *) error_norm
  line_step_limit = 0.0
else if (i ≡ 2) then
  read (scratch, *) error_norm, line_step_limit
else
  write (*, *) 'Error␣inputting␣error_norm,␣line_step_limit␣..␣' ‖ scratch(1 : 40)
end if
read (9, *) convergence
read (9, *) accuracy
read (9, format_a) scratch
write (*, *) '␣t␣start␣and␣stop␣', t_start, t_stop
write (*, *) '␣output␣start,␣incr␣', t_output_start, t_output_delta
write (*, *) '␣max␣IV␣change␣norm␣', y_change_max
write (*, *) '␣Norm␣of␣the␣errors␣added␣to␣BVs␣', error_norm
if (line_step_limit > 0.0) then
  write (*, *) '␣Limit␣on␣line␣step', line_step_limit
end if
write (*, *) '␣accuracy␣of␣integration,␣IV␣convergence␣', accuracy, convergence
weight = const(2.0)
```

This code is used in section 5.0.1.

This is a fairly simple loop with an echo of the data. We also count the number of boundary conditions which are to be met exactly. Finally, we make sure that *t_stop* is at least as big as the largest *t_bc*.

```
⟨ Input boundary conditions 5.0.7 ⟩ ≡
    count_exact_bv_s = 0
    write (*, *) ' The boundary conditions are:'
    write (*, *) '      i      q    t             b           exact?'
    do i_bc = 1, n_bc_in
      read (9, *) i_s, q(i_bc), t_bc(i_bc), bv(i_bc), exact_bc(i_bc)
      if (i_debug > 2) then
        write (*, format_bc) i_bc, q(i_bc), t_bc(i_bc), bv(i_bc), exact_bc(i_bc)
      end if
      if (exact_bc(i_bc) ≡ 0) then
        count_exact_bv_s = count_exact_bv_s + 1
      end if
    end do
    read (9, format_a) scratch
```

See also section 5.0.8.

This code is used in section 5.0.1.

If *error_norm* has a value greater than zero, we add noise to the boundary values. These changed boundary values are also output again. This gives a simple way to test the code. We add noise by adding a random number shifted to the interval of [-0.5–0.5] times *error_norm* to the boundary value.

Then, we make sure that *t_stop* is at least as big as the largest *t_bc*.

```
⟨ Input boundary conditions 5.0.7 ⟩ +≡
    if (error_norm > const(0.0)) then
      write (*, *) ' The corrupted boundary conditions are:'
      write (*, *) '      i      q    t             b           exact?'
      i_r = random_shift
      do i_bc = 1, n_bc
        i_r = i_r + 1
        if (i_r > random_count)
          i_r = 1
        bv(i_bc) = bv(i_bc) + error_norm * normal_numbers(i_r)
        write (*, format_bc) i_bc, q(i_bc), t_bc(i_bc), bv(i_bc), exact_bc(i_bc)
      end do
    end if
    if (n_bc > 0) then
      if (t_bc(n_bc) > t_stop) then
        t_stop = t_bc(n_bc)
        write (*, *) ' t start and stop ', t_start, t_stop
      end if
    end if
```

The processing of boundary conditions that are to be met exactly and in a least squares sense requires a little more overhead. The routine which solves a constrained least squares system will require that the exact equations are stored in the first few rows of the coefficient matrix. The *count_exact_bv_s* is used to indicate that count.

⟨ Quasi's local variables 4.1.7 ⟩ +≡
```
integer count_exact_bv_s
```

We similarly read and echo the estimates of the initial values. After the values are input, we copy them into *y_initial* which is used in the calculations.

⟨ Input initial value estimates 5.0.10 ⟩ ≡
```
do j = 1, n_
  read (9, *) i_s, exact_iv(j), y_input(j), min_iv(j), max_iv(j)
end do
write (*, *) ' The initial values are:'
write (*, *) '     i exact?    y              min          max'
do j = 1, n_
  write (*, format_iv) j, exact_iv(j), y_input(j), min_iv(j), max_iv(j)
  y_initial(j, 0) = y_input(j)
end do
read (9, format_a) scratch
```

This code is used in section 5.0.1.

The usefulness of a computer program can be affected in a most significant way by the formatting of the output. The user must be able to change this by input.

⟨ Input formats 5.0.11 ⟩ ≡

This code is used in section 5.0.1.

It is obvious that the number of independent particular solutions that are needed is reduced if some of the initial values are known. The limiting case is that of the initial value problem where all the initial values are known. In that case, *n_ps* will be zero. Remember that this is the maximum index of particular solutions or the number of *additional* particular solutions needed to meet the boundary conditions.

The calculation of *n_ps* is a straightforward count of the elements of *exact_iv* with non-negative values.

⟨ Reduce the order of the problem, if possible 5.0.12 ⟩ ≡
```
n_ps = 0
do j = 1, n_
  if (exact_iv(j) ≥ 0) then
    n_ps = n_ps + 1
  end if
end do
```

This code is used in section 5.0.1.

5.1 Set up each iteration

The independent variable obviously must be set for each iteration. This is not the only overhead of an iteration. The pointer to boundary conditions, *i_bc*, the variable indicating the value of t where output is desired, and the matrix where the superposition equations are stored must also be initialized. Two additional variables are needed because each boundary condition may be stored in different sections of the coefficient matrix. The value of *i_bc_constraint* points to the next row for storing an exact equation or constraint. The value of *i_bc_observation* points to the next row for storing an observation or boundary condition that is to be met in a least squares sense. The value of *i_bc_row* will be assigned one of those values when the functions of the solution are stored.

⟨ Set up for this iteration 5.1 ⟩ ≡
```
t = t_start
t_output = t_output_start
i_bc = 1
i_bc_constraint = 1
i_bc_observation = count_exact_bv_s + 1
⟨ Check to see if this is last boundary condition 5.1.2 ⟩
if (scoring ≡ 0) then
  max_ivp = n_ps
  ⟨ Perturb the initial value to ensure independence? 5.1.4 ⟩
else
  max_ivp = 1
  ⟨ Set initial values for scoring integrations 5.1.5 ⟩
end if
⟨ Store the superposition identity in C 5.1.3 ⟩
⟨ Debug output? Initial values 5.8 ⟩
```

This code is used in section 4.1.

We need to delcare the **integer** variables described above that are used in storing constraints and/or observations.

⟨ Quasi's local variables 4.1.7 ⟩ +≡
```
integer i_bc_constraint, i_bc_observation, i_bc_row
```

When we have processed the last boundary condition, we set *t_bc_i* to a very large value.

⟨ Check to see if this is last boundary condition 5.1.2 ⟩ ≡
```
if (i_bc ≤ n_bc) then
  t_bc_i = t_bc(i_bc)
else
  t_bc_i = 1.0 · 10^33D
end if
```

This code is used in sections 5.1 and 5.5.5.

The first row of C (and d which is in C) must be initialized to reflect the identity that the sum of the superposition coefficients must be unity. These are equations (6) in module 2.

⟨ Store the superposition identity in C 5.1.3 ⟩ ≡
```
do i = 0, max_ivp
  C(0, i) = const(1.0)
end do
C(0, max_ivp + 1) = const(1.0)
```

This code is used in section 5.1.

After we input the initial values, we copied them into *y_initial*. We did not destroy *y_input* because we may want to compare the final initial values with the original estimates. When we are iterating on a nonlinear problem, the new estimates of the initial values will be in *y_initial*.

The 0^{th} column of P is the current best estimate of the solution. The 1^{st} column is defined by the first element of y that is not known at *t_start* being perturbed. This is repeated until each of the unknown elements of y have been "perturbed" to ensure linear independence of the solutions.

⟨ Perturb the initial value to ensure independence? 5.1.4 ⟩ ≡
```
jj = 0
do i = 0, max_ivp
  do j = 1, n_
    P(j, i) = y_initial(j, 0)
  end do
  if (i > 0) then
    jj = jj + 1
    do while((exact_iv(jj) < 0) ∧ (jj < n_))
      jj = jj + 1
    end do
    ⟨ Perturb the jjth element of this column 5.1.6 ⟩
    perturbation(jj, 0) = P(jj, i) − P(jj, 0)
  end if
end do
```

This code is used in section 5.1.

After we have calculated a new estimate of the initial values, we perform a line search as required by the scoring method. If we denote these new initial values by $y(0)$ and the change by δ, then we will integrate the nonlinear equations with initial values of $y(0)$ and $y(0) + \delta$. The third point necessary for a line search comes from integration with initial values $y(0) - \delta$ which was the initial values from the previous iteration. Those values were saved.

⟨ Set initial values for scoring integrations 5.1.5 ⟩ ≡

```
do j = 1, n_
  P(j, 0) = y_initial(j, 0)
  if (exact_iv(j) ≥ 0) then
    P(j, 1) = y_initial(j, 0) + y_change(j)
  else
    P(j, 1) = y_initial(j, 0)
  end if
end do
```

This code is used in section 5.1.

We may have skipped over some elements of the initial values that are known. This element is unknown, so we perturb it to ensure that we have linear independence of the columns of P. If the current estimate of this element is not small, then we just multiply it by $const(1.2)$ which seems like a good number. After all, Don Knuth used it as the magnification step in TEX. If the initial value is less than some small value, then we arbitrarily replace it with another value. In some cases, this heuristic may be a bit too simple.

⟨ Perturb the jjth element of this column 5.1.6 ⟩ ≡

```
if (abs(y_initial(jj, 0)) ≥ const(0.1)) then
  P(jj, i) = 1.2 * y_initial(jj, 0)
else
  P(jj, i) = const(0.12)
end if
```

This code is used in section 5.1.4.

5.2 Set up for the current center

The center of expansion is assigned the value of the independent variable and the current solution is copied into the 0^{th} elements of the power series. We also reset a few variables.

⟨ Set up for current center of expansion 5.2 ⟩ ≡
```
t_center = t
do i = 0, max_ivp
  do j = 1, n_
    PS(0, j, i) = P(j, i)
  end do
end do
at_output = F
at_limit = F
at_external = F
at_end = F
at_bc = F
```

See also sections 5.2.1 and 15.4.

This code is used in section 4.1.1.

The 0^{th} element of the array *t_ps* is always set to the current center of expansion. The value of *t_center* that was set in the previous module.

⟨ Set up for current center of expansion 5.2 ⟩ +≡
```
t_ps(0) = t_center
```

For now, let's ignore the externals. If we have some real externals, this is where special handling would be programmed. Two examples of these will be discussed briefly. It is assumed that these may require the knowledge of the power series.

The occassional user may have a forcing function that is modeled as a step function. The power series will be piecewise continuous and must be re-evaluated at the discontinuities in the forcing function. The user would set the value of *t_external* to be at the next discontinuity. It a later module, the value of *t_limit* would have to be set to *t_external* if $t\_external < t\_limit$.

Another example comes from problems involving quadratic damping. This should be coded as $\mathbf{abs}(\dot{x})\dot{x}$ in oscillatory problems. Thus, after the series is known, the value of *t_external* is determined to be the next zero crossing of the function. Again, this may require comparison with and adjusting of *t_limit*.

⟨ Determine the limit of numeric convergence 5.2.2 ⟩ ≡
```
t_external = 1.0 · 10^33D
```

See also section 5.2.3.

This code is used in section 4.1.1.

To determine the max range of numeric confidence of the power series, we will look only at the first one. Before we do that, we make a little effort to avoid zero coefficients. We only check the base solution, $P_{(\cdot,0)}$, because all other solutions are purturbations and are used to change the initial values of the base. Since they are corrections, usually only the first few digits are really needed precisely.

The algorithm is to determine the maximum value of τ that satisfies the condition of $\mathbf{abs}(a_m\tau^m) \leq \mathbf{abs}(a_0)$. We include two loops to ensure that the coefficients used in the calculation are non-zero.

```
⟨ Determine the limit of numeric convergence 5.2.2 ⟩ +≡
    k_0 = 0
    k_m = m_
    do while((PS(k_0, 1, 0) ≡ const(0.0)) ∧ (k_0 < m_))
      k_0 = k_0 + 1
    end do
    do while(PS(k_m, 1, 0) ≡ const(0.0) ∧ (k_m > k_0))
      k_m = k_m − 1
    end do
    if (k_m > k_0 + (m_ / 2)) then
      tau_ok = abs(accuracy ∗ PS(k_0, 1, 0) / PS(k_m, 1, 0))
      tau_ok = exp(log(tau_ok) / float(k_m − k_0))
      if (tau_ok < const(0.1)) then
        tau_ok = const(0.1)
      end if
    else
      tau_ok = const(0.05)
    end if
    t_limit = t_center + tau_ok
```

We used several temporary variables there.

```
⟨ Quasi's input variables 4.1.4 ⟩ +≡
    floating tau_ok
    integer k_0, k_m
```

5.3 The initial elements have been established using the initial conditions

Now, we use the Frobenius recurrence equations to calculate the rest of the terms. In some nonlinear problems, the first part is necessarily the base solution and then *max_ivp* solutions can be calculated in parallel.

```
⟨ Calculate the power series coefficients, recurrence 5.3 ⟩ ≡
    do i = 0, max_ivp
      do k = 1, m_
        ⟨ Calculate the k-th term of the power series 15 ⟩
      end do
    end do
    ⟨ Debug output? Power series coefficients 5.8.1 ⟩
```

This code is used in section 4.1.1.

5.4 Move forward using this power series

There are at least five reasons to set the point of evaluation of the power series. These include:

1. The numeric convergence of the power series specifies *t_limit*. At this point, the power series will need to be re-expanded.
2. Many problems require a regular output. This may be done to give results for graphical output. In some cases the user may wish to generalize or change this. For example, this output may be directed to a file for graphical use.
3. External requirements may specify a point of output. One example is that there may be discontinuities in the model.
4. Whenever a boundary condition is encountered, some work must be done with the solution.
5. Output should be performed at the final value of the independent variable, *t_stop*.

⟨ Determine τ and advance t 5.4 ⟩ ≡
$t = \textbf{\textit{min}}(t\_output,\ t\_limit,\ t\_external,\ t\_bc\_i,\ t\_stop)$
⟨ Are we at the limit of accuracy of the power series 5.4.1 ⟩
⟨ Should we output something 5.4.2 ⟩
⟨ Is this where an external event is 5.4.3 ⟩
⟨ Is this the point of a boundary condition 5.4.4 ⟩
⟨ Check to see if we have integrated far enough 5.4.5 ⟩

This code is used in section 4.1.1.

In these next several modules we check to see what the next value of t we will encounter is. Notice that these are not nested conditionals because we can have several of these true at the same time.

⟨ Are we at the limit of accuracy of the power series 5.4.1 ⟩ ≡
if $(t\_limit \leq t)$ **then**
$tau = t\_limit - t\_center$
$t = t\_limit$
$at\_limit = \mathcal{T}$
$evaluate\_function = \mathcal{T}$
end if

This code is used in section 5.4.

Just like the previous one except for some permutations of names. Some applications may need the derivatives evaluated too.

⟨ Should we output something 5.4.2 ⟩ ≡
```
if (t_output ≤ t) then
  tau = t_output − t_center
  t = t_output
  at_output = T
  evaluate_function = T
end if
```

This code is used in section 5.4.

Just like the previous two except for some permutations of names. Some applications may need the derivatives evaluated at these external points. The 'external' problems mentioned previously would not require rewriting this code, the previous section would probably be a better place.

⟨ Is this where an external event is 5.4.3 ⟩ ≡
```
if (t_external ≤ t) then
  tau = t_external − t_center
  t = t_external
  at_external = T
end if
```

This code is used in section 5.4.

This module is different from the previous ones because there is a possibility of several boundary conditions occurring at the same time. Each could be different.

⟨ Is this the point of a boundary condition 5.4.4 ⟩ ≡
```
if (t_bc_i ≤ t) then
  tau = t_bc_i − t_center
  t = t_bc_i
  at_bc = T
  j = i_bc
  do while(t_bc_i ≡ t_bc(j))
    if (q(j) > 0) then
      evaluate_function = T
    else if (q(j) < 0) then
      evaluate_derivative = T
    else
      evaluate_function = T
      evaluate_derivative = T
    end if
    j = j + 1
  end do
end if
```

This code is used in section 5.4.

If one of the above modules has set t to a value greater than *t_stop*, then we need to reduce it.

```
⟨ Check to see if we have integrated far enough 5.4.5 ⟩ ≡
    if (t_stop ≤ t) then
      tau = t_stop − t_center
      t = t_stop
      at_end = T
      evaluate_function = T
    end if
```

This code is used in section 5.4.

The evaluation of the power series is done by calling the *ps_eval* routine. This is quite simple in the initial value problem. It would be in a loop on the boundary value case.

The **if**'s are separate because both may need evaluation.

```
⟨ Evaluate the power series 5.4.6 ⟩ ≡
    do i = 0, max_ivp
      do j = 1, n_
        if (evaluate_derivative) then
          call ps_eval_d(PS(0, j, i), m_, tau, DP(j, i))
        end if
        if (evaluate_function) then
          call ps_eval(PS(0, j, i), m_, tau, P(j, i))
        end if
      end do
    end do
```

This code is used in section 4.1.2.

We introduced two **logical** variables. They are *evaluate_function* and *evaluate_derivative*. The first is set to $\mathcal{T}$ if we need the function the power series represents and the latter is used in a similar nature if we need its derivative.

```
⟨ Quasi's input variables 4.1.4 ⟩ +≡
    logical evaluate_function, evaluate_derivative
```

5.5 Use the solution

The *at_output*, *at_external*, and *at_bc* flags must be reset to $\mathcal{F}$ when we finish their operations. There may be several boundary conditions at the current point. The other flags, *at_limit* and *at_end* are not reset because they indicate it is appropriate to exit the loop.

```
⟨ Use the result of the evaluation 5.5 ⟩ ≡
    if (at_limit) then
      ⟨ Reached the limit of this expansion 5.5.1 ⟩
    end if
    if (at_output) then
      ⟨ Output the results, regularly 5.5.2 ⟩
    end if
    if (at_external) then
      ⟨ Perform any needed work for external reasons 5.5.3 ⟩
    end if
    if (at_bc) then
      ⟨ Save the row of the coefficient matrix 5.5.5 ⟩
    end if
    if (at_end) then
      ⟨ End game processing 5.7 ⟩
    end if
```

This code is used in section 4.1.2.

This is a short one. If it were not using *i_debug*, it would probably be removed in a working version. It allows us to track where the solution is proceeding.

```
⟨ Reached the limit of this expansion 5.5.1 ⟩ ≡
    if (i_debug > 3) then
      write (*, *) '␣At␣limit␣', tau, t
    end if
```

This code is used in section 5.5.

When we reach a regular output point, we simply output the results. It may be that these would also be written to a file for later graphics use. When one of these is used, the value of *t_output* must be updated.

The initial values are output elsewhere and cannot be supressed. The parameters are also output at the point where we 'start the output.' After the start of the output, we output only the elements of the ODE's that may change, in other words we don't output the constant coefficients or parameters.

⟨ Output the results, regularly 5.5.2 ⟩ ≡
```
if (t ≡ t_output_start) then
  n_out = n_
else
  n_out = n_ode
end if
if ((t ≥ t_output_start) ∧ (t > t_start)) then
  write (∗, format_t_y) t, (P(j, 0), j = 1, n_out)
  ⟨ Debug output? All particular solutions 5.8.3 ⟩
end if
t_output = t_output + t_output_delta
at_output = F
```

This code is used in section 5.5.

This will be important when extended for boundary value problems. When one of these is used, the value of *t_external* must be updated.

If we reach a new "shooting point," we then must reestablish the initial values.

⟨ Perform any needed work for external reasons 5.5.3 ⟩ ≡
```
⟨ Change initial values because we are at a shooting point 5.5.4 ⟩
if (i_debug ≥ 4) then
  write (∗, ∗) 'At external ', t
end if
at_external = F
```

This code is used in section 5.5.

⟨ Change initial values because we are at a shooting point 5.5.4 ⟩ ≡

This code is used in section 5.5.3.

When we are at a boundary condition and storing some function of the solution in the coefficient matrix. This is nested in a loop because we may have multiple boundary conditions at the same time, t.

```
⟨ Save the row of the coefficient matrix 5.5.5 ⟩ ≡
    do while(t ≡ t_bc_i)
      ⟨ Assign i_bc_row based on constraint vs. observation 5.5.6 ⟩
      do i = 0, max_ivp
        if (q(i_bc) > 0) then
          ⟨ Store the element based upon the solution vector 5.5.7 ⟩
        else if (q(i_bc) < 0) then
          ⟨ Store the element based upon the derivative 5.5.8 ⟩
        else
          ⟨ Special boundary condition operators 5.5.9 ⟩
        end if
      end do
      C(i_bc_row, max_ivp + 1) = bv(i_bc)
      ⟨ Debug output? Saving 5.8.4 ⟩
      i_bc = i_bc + 1
      ⟨ Check to see if this is last boundary condition 5.1.2 ⟩
    end do
    at_bc = F
```

This code is used in section 5.5.

The first *count_exact_bv_s*+1 rows of the coefficient matrix will be constraints or equations. The rest of the rows represent observations and are to be modeled in the least squares sense.

```
⟨ Assign i_bc_row based on constraint vs. observation 5.5.6 ⟩ ≡
    if (exact_bc(i_bc) ≡ 0) then
      i_bc_row = i_bc_constraint
      i_bc_constraint = i_bc_constraint + 1
    else
      i_bc_row = i_bc_observation
      i_bc_observation = i_bc_observation + 1
    end if
```

This code is used in section 5.5.5.

If the boundary condition is on an element of y, then we store it directly (or multiply it by an appropriate weight.)

```
⟨ Store the element based upon the solution vector 5.5.7 ⟩ ≡
    if (exact_bc(i_bc) ≤ 1) then
      C(i_bc_row, i) = P(q(i_bc), i)
    else if (exact_bc(i_bc) > 1) then
      C(i_bc_row, i) = P(q(i_bc), i) * (weight^(exact_bc(i_bc)−1))
    end if
```

This code is used in section 5.5.5.

This boundary condition is on the derivative of the state vector. Thus, we use the *DP* matrix as the source of values.

⟨ Store the element based upon the derivative 5.5.8 ⟩ ≡

```
if (exact_bc(i_bc) ≤ 1) then
  C(i_bc_row, i) = DP(−q(i_bc), i)
else if (exact_bc(i_bc) > 1) then
  C(i_bc_row, i) = DP(−q(i_bc), i) ∗ (weight^(exact_bc(i_bc)−1))
end if
```

This code is used in section 5.5.5.

Special operators specify use more than a single element of the state vector in specifying a boundary conditions.

⟨ Special boundary condition operators 5.5.9 ⟩ ≡

This code is used in section 5.5.5.

5.6 The end of an iteration

If *scoring* ≡ 0 then we determine the superposition coefficients and the new estimates for the initial values. We may output some statistics. If *scoring* ≡ 1 we complete the line search that is used to really get the new estimates of the initial values.

⟨ End of forward integration 5.6 ⟩ ≡

```
if (i_debug ≥ 2) then
  write (∗, ∗) ' Augmented matrix'
  do i = 0, n_bc
    write (∗, format_C) (C(i, j), j = 0, max_ivp + 1)
    if (i ≡ count_exact_bv_s)
      write (∗, ∗)
  end do
end if
final = iteration ≥ iteration_max
if (scoring ≡ 0) then
  ⟨ Calculate the new initial value estimates 5.6.1 ⟩
else
  ⟨ Perform the line search part of scoring 5.6.2 ⟩
  iteration = iteration + 1
end if
```

This code is used in section 4.1.

Vanilla calculation of new estimates of initial values.

```
⟨ Calculate the new initial value estimates 5.6.1 ⟩ ≡
    if (¬final) then
      ⟨ Solve for the superposition coefficients 5.6.4 ⟩
      ⟨ Estimate the new initial values, to be revised 5.6.5 ⟩
      ⟨ Check for convergence of the initial values 5.6.7 ⟩
    else
      if (n_bc > max_ivp) then
        ⟨ Solve for the superposition coefficients 5.6.4 ⟩
        ⟨ Perform the statistical analysis 7 ⟩
      end if
    end if
    at_end = F
```

This code is used in section 5.6.

The minimization of the line search part of scoring is quite straightforward. We find two values that correspond to the values of the function we are minimizing. (A third one is already available.) Then, using a three point formula to determine a quadratic and its minimum point is trivial. The independent variable of the formula used is scaled such that -1 is the initial values of the previous iteration, 0 is the vanilla estimate from the Newton-like step, and 1 would be an accelerated step twice as much as the Newton-like step gave. Thus, we would expect values fairly close to 0, certainly in a limiting sense.

```
⟨ Perform the line search part of scoring 5.6.2 ⟩ ≡
    K_n_1 = const(0.0)
    K_n_2 = const(0.0)
    do i = 0, n_bc
      K_n_1 = K_n_1 + (C(i, 0) − C(i, max_ivp + 1))^2
      K_n_2 = K_n_2 + (C(i, 1) − C(i, max_ivp + 1))^2
    end do
    ⟨ Calculate the line step 5.6.3 ⟩
    do j = 1, n_
      y_initial(j, 0) = y_initial(j, 0) + y_change(j) ∗ line_step
      if (exact_iv(j) > 0) then
        y_initial(j, 0) = min(y_initial(j, 0), max_iv(j))
        y_initial(j, 0) = max(y_initial(j, 0), min_iv(j))
      end if
    end do
    at_end = F
```

This code is used in section 5.6.

The line step calculation is implemented as a heuristic. If the exponents of the sums of squares are all the same, then we are so close to convergence that it is of no use. In that case, we use zero because the Newton step is already incorporated.

We limit this to an interval of $(-0.04, 0.04)$? Maybe even tighter? Should we create two more input variables for these limits? Well, it is now an input variable *line_step_limit*.

```
⟨ Calculate the line step 5.6.3 ⟩ ≡
    i_0 = nint(log(K_n_0))
    i_1 = nint(log(K_n_1))
    i_2 = nint(log(K_n_2))
    if ((i_0 ≠ i_1) | (i_0 ≠ i_2) | (i_1 ≠ i_2)) then
      line_step = const(0.5) * (K_n_2 − K_n_0) / (K_n_2 − const(2.0) * K_n_1 + K_n_0)
      if (line_step < −line_step_limit)
        line_step = −line_step_limit
      if (line_step > line_step_limit)
        line_step = line_step_limit
    else
      line_step = 0.0
    end if
    write(∗, "(a,4g11.3)") 'Step', line_step, K_n_0, K_n_1, K_n_2
```

This code is used in section 5.6.2.

The solution for the superposition coefficients is accomplished through the use of a rather specialized Gaussian routine.

The argument *final* has been set to $\mathcal{T}$ if this is the final iteration and it is an overdetermined problem. Thus, the name can be misleading because we might not actually solve for the superposition coefficients.

A more complete response rather than just **stop**ping should be based upon the value of *i_return*.

The calculation of *K_n_0* is for use in the method of scoring part of the iteration.

This routine may be effectively parallelized in some problems.

```
⟨ Solve for the superposition coefficients 5.6.4 ⟩ ≡
    K_n_0 = const(0.0)
    do i = 0, n_bc
      K_n_0 = K_n_0 + (C(i, 0) − C(i, max_ivp + 1))^2
    end do

    call gj_svd(C, n_bc, n_ps, count_exact_bv_s, i_debug, i_return)
    if (i_return > 1)
      stop
```

This code is used in section 5.6.1.

The calculation of the initial values is quite straightforward, if you really understand this. Equation (3), with the identity (6), and the strategy used to perturb the initial values in 2 make this a simple calculation. The change in each initial value will be the product of the superposition constant and the perturbation. Most of the code in this module is needed to make the code efficient in the presence of known initial values.

The variable *jj* is used in acknowledgement that some efficiency is realized if some of the initial values are specified to be met exactly.

The change may be scaled and then we also check to see if the new initial value is within the range that may have been specified.

```
⟨ Estimate the new initial values, to be revised 5.6.5 ⟩ ≡
    ⟨ Calculate the norm of the change in the initial values 5.6.6 ⟩
    do j = 1, n_
      if (exact_iv(j) ≥ 0 ∧ y_change_norm > y_change_max) then
        y_change(j) = y_change(j) * (y_change_max / y_change_norm)
      end if
      if (exact_iv(j) ≥ 0) then
        y_initial(j, 0) = y_initial(j, 0) + y_change(j)
      end if
      if (exact_iv(j) > 0) then
        y_initial(j, 0) = min(y_initial(j, 0), max_iv(j))
        y_initial(j, 0) = max(y_initial(j, 0), min_iv(j))
      end if
    end do
```

This code is used in section 5.6.1.

There is always a possibility of overshoot and limited intervals where the Newton methods work quadratically. We have found that constraining the change to a finite limit often improves the range of workable initial estimates and at little cost.

```
⟨ Calculate the norm of the change in the initial values 5.6.6 ⟩ ≡
    y_change_norm = const(0.0)
    jj = 0
    do j = 1, n_
      y_change(j) = const(0.0)
      if (exact_iv(j) ≥ 0) then
        jj = jj + 1
        y_change(j) = perturbation(j, 0) * C(jj, max_ivp + 1)
        y_change_norm = y_change_norm + y_change(j)^2
      end if
    end do
    y_change_norm = sqrt(y_change_norm)
```

This code is used in section 5.6.5.

We check to see if the initial values have converged. To do this, we compare this to the values that β should have if the initial values were correct, namely $(1, 0, 0, \ldots)$. If we have converged, then we may reduce *iteration_max* because further iterations are useless.

On the last iteration, we may set the value of *t_output_start* to ensure some output. This allows the user to set it to an arbitrarily large number to avoid output until it is really desired.

⟨ Check for convergence of the initial values 5.6.7 ⟩ ≡
```
max_beta = abs(const(1.0) − C(0, max_ivp + 1))
do j = 1, max_ivp
  if (abs(C(j, max_ivp + 1)) > max_beta) then
    max_beta = abs(C(j, max_ivp + 1))
  end if
end do
if ((max_beta ≤ convergence) ∧ (iteration < iteration_max)) then
  iteration_max = iteration + 1
end if
if ((iteration + 1) ≡ (iteration_max) ∧ (t_output_start > t_stop)) then
  t_output_start = t_start
end if
```

This code is used in section 5.6.1.

5.7 The very end

We can envision several instances when a specific application might wish to make changes at this point.

⟨ End game processing 5.7 ⟩ ≡
```
⟨ Debug output? At the end 5.8.2 ⟩
```

This code is used in section 5.5.

5.8 Debug output

These are clustered at the end of the major section because they should not interrupt the natural flow of reading the program. This first module does some output in all cases.

```
⟨ Debug output? Initial values 5.8 ⟩ ≡
    if (scoring ≡ 0 | i_debug ≥ 1) then
      write (output_flag, '(i3,a)') iteration, '␣>␣'
      if (iteration ≥ iteration_max)
        output_flag(4 : 4) = '*'
      write (∗, ∗) '␣Initial␣values␣for␣iteration␣', iteration, '␣of␣', iteration_max
      write (∗, format_t_0) output_flag, t, (P(j, 0), j = 1, n_)
    end if
    if (i_debug ≥ 2) then
      do i = 1, max_ivp
        write (∗, format_t_y) t, (P(j, i), j = 1, n_)
      end do
    end if
```

This code is used in section 5.1.

The user will get lots of output if *i_debug* is set to a large value and this output is triggered.

```
⟨ Debug output? Power series coefficients 5.8.1 ⟩ ≡
    if (i_debug ≥ 5) then
      write (∗, ∗) '␣Power␣series␣coefficients,␣t␣=␣', t_center
      do i = 0, max_ivp
        do j = 1, n_
          if (j ≤ n_ode) then
            write (∗, format_C) (PS(k, j, i), k = 0, m_)
          else
            write (∗, format_C) PS(0, j, i)
          end if
        end do
      end do
    end if
```

This code is used in section 5.3.

This gives the user the means of determining the state of the world when the end of the integration interval has been reached.

```
⟨ Debug output? At the end 5.8.2 ⟩ ≡
    if (i_debug ≥ 3) then
      do i = 0, max_ivp
        write (∗, format_t_y) t, (P(j, i), j = 1, n_ode)
      end do
      write (∗, ∗) '␣At␣the␣end␣', t
    end if
```

This code is used in section 5.7.

Sometimes we wish to output all the particular solutions, not just the base one.

⟨ Debug output? All particular solutions 5.8.3 ⟩ ≡
```
if (i_debug ≥ 3) then
  do i = 1, max_ivp
    write (∗, format_t_y) t, (P(j, i), j = 1, n_out)
  end do
end if
```

This code is used in section 5.5.2.

We can output the individual row of the augmented coefficient matrix as it is saved.

⟨ Debug output? Saving 5.8.4 ⟩ ≡
```
if (i_debug > 3) then
  write (∗, ∗) '␣Saving␣', i_bc, t_bc_i, exact_bc(i_bc), q(i_bc)
  write (∗, format_C) (C(i_bc_row, i), i = 0, max_ivp + 1)
end if
```

This code is used in section 5.5.5.

6 Extraneous functions

The function *number_of_fields* returns the **integer** that is the apparent number of numeric fields in the current input line.

```
"psq.f" 6.1 ≡
    integer function number_of_fields(line)
      implicit none
      character line*(*)

      integer i, length
      logical in_a_number
      character c*1

      number_of_fields = 0
      length = len(line)
      i = 1
      c = line(i : i)
      do while(i ≤ length)
        in_a_number = index("+-0123456789.", c) > 0
        if (in_a_number) then
          number_of_fields = number_of_fields + 1
          do while(in_a_number)
            i = i + 1
            c = line(i : i)
            in_a_number = index("+-0123456789.dDeE", c) > 0
          end do
        end if
        i = i + 1
        c = line(i : i)
      end do
    end
```

7 The statistical analysis

The entire procedure we are doing here can be called *regression analysis with differential equation models.* Well, the code solves a number of other problems but the central one in the design is to solve those with observations (boundary conditions with error to be met in a best fit sense).

You might have noticed that when convergence has been achieved, we do one more iteration. This extra iteration is used to calculate these statistics.

This routine has not been rewritten in a WEB *form. At this writing, the routines for calculation of the basic statistical functions have been completed and tested. The inclusion of all parts of this code in a* WEB *form is done for pedagogical reasons, primarily transportability. Production codes will have these replaced by calls to supported libraries.*

⟨ Perform the statistical analysis 7 ⟩ ≡

This code is used in section 5.6.1.

8 Power Series

[the_subs.hweb] The power series is one of the more common tools in applied mathematics. It often comes from the truncated Taylor or Maclaurin series. The solution of ordinary differential equations can be expressed in this form but it often makes for a lot of work and long expressions.

Power series are an important theoretical tool in mathematics. They can also be used to great advantage in numerical computations. In many cases the solution of a differential equation can be expressed in terms of their power series expansions. One disadvantage of this representation is that the formal manipulation of these power series can often be quite complex.

The Frobenius form is an economical means of calculating these coefficients rather than using the repeated differentiation. We will describe various routines which will expedite some of the formal manipulations necessary to solve a fairly broad subset of the common differential equations.

8.1 Evaluation of a Power Series

[the_subs.hweb] This will be a straightforward use of Horner's method. Although we will often have many related power series, each one will be treated separately.

Horner's method is to evaluate the power series in a factored form. We use the right hand side of:

$$\sum_{k=0}^{m} a_k t^k = a_0 + (a_1 + (\cdots (a_{m-1} + (a_m)t)t \cdots)t)t$$

The evaluation of the left side of this equation can be reduced to $2m$ multiplications and m additions at best. The right hand side halves the number of multiplications.

The variable v is declared as an array because it is normal to call it with an element of an array, just in case some compiler is checking these things.

This is so short, we really brute force it!

"psq.f" 8.1 ≡

```
subroutine ps_eval(a, m_, t, v)
  implicit none
  integer m_, k
  floating a(0 : m_), t, v(1)

  v(1) = a(m_)
  do k = (m_ − 1), 0, −1
    v(1) = a(k) + v(1) ∗ t
  end do
  return
end
```

[the_subs.hweb] Evaluation of the derivative of a power series. The evaluation of the derivative of a power series will also be a use of Horner's method. This is frequently needed in the inverse problems. It is often easiest to measure derivatives rather than the phenomena itself.

The derivative is:

$$\sum_{k=0}^{m-1}(k+1)a_{k+1}t^k = \sum_{k=1}^{m} k a_k t^{(k-1)} \tag{22}$$

$$= a_1 + (2a_2 + (\cdots((m-1)a_{m-1} + m(a_m)t)t\cdots)t)t \tag{23}$$

$$\tag{24}$$

We normally don't write the summations except from zero, in this case it makes it a bit more efficient. Notice that there is one less term in the derivative of the series than there is in the series itself. Also notice that the evaluation of a derivative requires twice as many multiplications as the function does.

"psq.f" 8.1.1 ≡

```
subroutine ps_eval_d(a, m_, t, v)
  implicit none
  integer m_, k
  floating a(0 : m_), t, v(1)

  v(1) = a(m_) * m_
  do k = (m_ - 1), 1, -1
    v(1) = a(k) * k + v(1) * t
  end do
  return
end
```

9 Power Series Operations, arithmetic

[the_subs.hweb] The basic arithmetic operations on power series (addition, subtraction, multiplication, and division) are relatively straightforward. Addition and subtraction are trivial, as they involve only adding and subtracting the appropriate coefficients. Multiplication is likewise trivial, but involves enough computation to make a function call worthwhile. Division is quite a bit more complicated, but still can be computed by a means which is equivalent to that of the longhand division technique for polynomials taught in elementary algebra.

Other operations to be performed are the exponential, sine and cosine, square root, and arbitrary power of a power series. We will explain in detail how these functions are calculated.

It should be stressed here that in all the operations described, the function or subroutine computes only one term of the series each time it is called. Although this may seem counter-intuitive at first, it is this feature which allows us to be able to use these functions efficiently in the complicated application in which we will see them.

Again, we will consistently approximate:

$$x(t) \approx \sum_{i=0}^{n} x_i t^i$$

This power series, and many others, will be developed by the use of Frobenius recurrence relations. Of course, we may use $a(t)$, $b(t)$, and a wide array of symbols for these truncated power series.

[the_subs.hweb] We first consider computing the product of two power series. The inputs to this function are two vectors a and b, with indices $0, \ldots, n$. The product of these is

$$\sum_{k=0}^{n} c_k t^k = \left(\sum_{k=0}^{n} a_k t^k \right) \left(\sum_{k=0}^{n} b_k t^k \right)$$

where c_k expands to

$$c_k = a_0 b_k + a_1 b_{k-1} + \ldots + a_k b_0.$$

The values of elements $a_0, \ldots, a_i$ and $b_0, \ldots, b_i$ are known at the time of the call. The function result, *ps_mult*, is the term c_i defined above.

"psq.f" 9.0.1 ≡

```
floating function ps_mult(a, b, i)
  implicit none
  ⟨ Declare ps_mult variables 9.0.2 ⟩
  ⟨ Compute ps_mult result 9.0.3 ⟩
  return
end
```

[the_subs.hweb] We will declare the input parameters to the function. The arrays a and b probably have an actual range of $0, \ldots, n$. We declare a subset range of $0, \ldots, i$. We must also declare a single local loop index.

i is the index of the product term to be computed.

a and b are as described above.

k is a local variable used as a loop index.

⟨ Declare *ps_mult* variables 9.0.2 ⟩ ≡
```
integer i
floating a(0 : i), b(0 : i)
integer k
```

This code is used in section 9.0.1.

[the_subs.hweb] The computation is a rather straightforward implementation of the above formula for c_k, using a loop to sum the result. We note that the saving of one addition will not work in FORTRAN 66. That does not count because it did not have zero indices anyway.

⟨ Compute *ps_mult* result 9.0.3 ⟩ ≡
```
ps_mult = a(0) * b(i)
do k = 1, i
  ps_mult = ps_mult + a(k) * b(i − k)
end do
```

This code is used in section 9.0.1.

[the_subs.hweb] To compute the quotient of two power series, we use an adaptation of longhand division of polynomials (with some "neat tricks" to help simplify the computation.) We compute the quotient q as follows:

$$q(t) = \frac{u_0 + u_1 t + \ldots + u_i t^i}{d_0 + d_1 t + \ldots + d_i t^i}$$

If we attempt to perform the long division symbolically at this point, we might quickly despair at the successively increasing complexity of each coefficient. However, we observe that each successive coefficient of the quotient series depends on the previous coefficients. Simplifying, we obtain the following formula:

$$q_i = \frac{u_i - \sum_{k=0}^{i-1} q_k d_{i-k}}{d_0}$$

The following function will return the coefficient of the i^{th} term of of the quotient q, where u is the numerator and d is the denominator. The quotient q is also an argument because the $q_0, \ldots, q_{i-1}$ are needed to compute q_i.

"psq.f" 9.0.4 ≡
```
floating function ps_div(u, d, i, q)
  implicit none
  ⟨ Declare ps_div variables 9.0.5 ⟩
  ⟨ Compute ps_div result 9.0.6 ⟩
  return
end
```

[the_subs.hweb] We will declare the parameters to the function. u and d are vectors with indices of $0, \ldots, n$. The vector of quotient coefficients, q, must be also be sent, as described above. We must declare a single local loop index.

i is the index of the quotient term to be computed.

u is the numerator, d is the denominator, and q is the quotient. All of these have an upper limit of i.

k is a local variable used as a loop index.

⟨ Declare ps_div variables 9.0.5 ⟩ ≡
```
integer i
floating u(0 : i), d(0 : i), q(0 : i)
integer k
```

This code is used in section 9.0.4.

[the_subs.hweb] The computation is a straightforward application of the above formula for q_i, using a loop to perform the required sum. The function result is summed in *ps_div* and copied into q_i .

⟨ Compute *ps_div* result 9.0.6 ⟩ ≡
```
ps_div = u(i)
do k = 0, i − 1
  ps_div = ps_div − q(k) ∗ d(i − k)
end do
ps_div = ps_div / d(0)
q(i) = ps_div
```

This code is used in section 9.0.4.

[the_subs.hweb] Transcendental functions. To compute the exponential function of a power series it is necessary to use our knowledge of elementary calculus. We derive the series as follows: If

$$e(t) = \exp(a(t))$$

then by differentiating (and omitting the (t) of each term) we obtain

$$\dot{e} = \exp(a)\dot{a} = e\dot{a}$$

or

$$e_1 + 2e_2t + \ldots + ie_it^{i-1} = (e_0 + e_1t + \ldots + e_it^i)(a_1 + 2a_2t + \ldots + ia_it^{i-1})$$

If we equate the coefficients of t^{i-1} on either side we obtain the relation:

$$ie_i = ia_ie_0 + (i-1)a_{i-1}e_1 + \ldots + a_1e_{i-1}$$

or

$$e_i = \frac{\sum_{k=1}^{i} ka_ke_{i-k}}{i}$$

It is clear that $e_0 = \exp(a_0)$ from the definition of $e(t)$. The ka_k term is a manifestation of $\dot{a}(t)$.

"psq.f" 9.0.7 ≡
```
floating function ps_exp(a, i, e)
  implicit none
  ⟨ Declare ps_exp variables 9.0.8 ⟩
  if (i ≡ 0) then
    ⟨ Perform the trivial case for ps_exp 9.0.9 ⟩
  else
    ⟨ Perform the general case for ps_exp 9.0.10 ⟩
  end if
  e(i) = ps_exp
  return
end
```

[the_subs.hweb] We declare the parameters for the *ps_exp*. The argument of the the exponential function is a, i is the current term that we are calculating, and e is the vector of the previous results (e needs to be passed because the previous coefficients are used in computing the next one.) We also declare a single local loop index.

i is the index of the exponential term to be computed.

a and e are the zero-indexed vectors described above. Their upper limit is i.

k is a local variable used as a loop index.

⟨ Declare *ps_exp* variables 9.0.8 ⟩ ≡
```
integer i
floating a(0 : i), e(0 : i)
integer k
```
This code is used in section 9.0.7.

[the_subs.hweb] The 0^{th} term is the trivial case, where $e_0 = \exp(a_0)$.

⟨ Perform the trivial case for *ps_exp* 9.0.9 ⟩ ≡
```
ps_exp = exp(a(0))
```
This code is used in section 9.0.7.

[the_subs.hweb] The general case is a straightforward implementation of the formula for e_i described above.

⟨ Perform the general case for *ps_exp* 9.0.10 ⟩ ≡
```
ps_exp = a(1) * e(i − 1)
do k = 2, i
  ps_exp = ps_exp + k * a(k) * e(i − k)
end do
ps_exp = ps_exp / i
```
This code is used in section 9.0.7.

[the_subs.hweb] The natural logarithm. To compute the natural logarithm of a power series it is necessary to use our knowledge of elementary calculus. We derive the series as follows: If

$$u(t) = \ln(a(t))$$

then by differentiating (and omitting the (t) of each term) we obtain

$$\dot{u} = \frac{\dot{a}}{a}$$

or

$$u_1 + 2u_2 t + \ldots + iu_i t^{i-1} = \frac{(a_1 + 2a_2 t + \ldots + ia_i t^{i-1})}{(a_0 + a_1 t + \ldots + a_i t^i)}$$

If we equate the coefficients of t^{i-1} on either side we obtain the relation:

$$iu_i = q_i$$

where q_i is the appropriate term in the quotient.

It is clear that $u_0 = \ln(a_0)$ from the definition of $u(t)$. The efficient form of the division routine requires knowing the quotient at each stage. This is calculated in this routine based upon the realization that this quotient is the derivative of the result. This additional cost is considered worthwhile when compared with the long range confusion of having an additional argument to the call.

"psq.f" 9.0.11 ≡

```
floating function ps_ln(a, i, u)
  implicit none
  ⟨ Declare ps_ln variables 9.0.12 ⟩
  if (i ≡ 0) then
    ⟨ Perform the trivial case for ps_ln 9.0.13 ⟩
  else
    ⟨ Perform the general case for ps_ln 9.0.14 ⟩
  end if
  u(i) = ps_ln
  return
end
```

[the_subs.hweb] We declare the parameters for the *ps_ln*. The argument of the the logarithm function is a, i is index of the term we are calculating, and u is the vector of the previous results. This needs to be passed because the previous coefficients are used in computing the next one. We also declare a local loop index, k.

⟨ Declare *ps_ln* variables 9.0.12 ⟩ ≡

```
integer i
floating a(0 : i), u(0 : i)
integer k
```

This code is used in section 9.0.11.

[the_subs.hweb] The 0^{th} term is the trivial case, where $u_0 = \ln(a_0)$.

⟨ Perform the trivial case for *ps_ln* 9.0.13 ⟩ ≡
```
ps_ln = log(a(0))
```

This code is used in section 9.0.11.

[the_subs.hweb] The general case is not a straightforward implementation of the above formula. As indicated, this needs a quotient of power series. This quotient is $u_1 + 2u_2t + \ldots + iu_it^{i-1}$ and is "re-calculated" as needed. The rest of this can be understood when carefully compared with the routine *ps_div*. Notice the last line of code is the division required by the method of Frobenius while the division routine is algebra, not calculus.

⟨ Perform the general case for *ps_ln* 9.0.14 ⟩ ≡
```
ps_ln = i * a(i)
do k = 0, i - 1
  ps_ln = ps_ln - (k + 1) * u(k + 1) * a(i - k - 1)
end do
ps_ln = ps_ln / a(0)
ps_ln = ps_ln / (i)
```

This code is used in section 9.0.11.

[the_subs.hweb] The square. The next function we will need for the power series operations is the function to square the vector. It should be noted that the square function is simply a special case of multiplication. We have optimized it here for reasons of efficiency.

We pair the identical terms in the sum so that they need not be computed for a second time. The modification of the recurrence formula can be expressed as follows:

$$c_i = \begin{cases} (a_0)^2 & \text{if } i = 0 \\ 2\sum_{k=0}^{\frac{i-1}{2}} a_k a_{i-k} & \text{if } i > 0 \text{ and odd} \\ (a_{\frac{i}{2}})^2 + 2\sum_{k=0}^{\lfloor \frac{i-1}{2} \rfloor} a_k a_{i-k} & \text{if } i > 0 \text{ and even} \end{cases}$$

"psq.f" 9.0.15 ≡
```
floating function ps_sqr(a, i)
  implicit none
  ⟨ Declare ps_sqr variables 9.0.16 ⟩
  if (i ≡ 0) then
    ps_sqr = a(0) * a(0)
  else
    ⟨ Compute ps_sqr sum 9.0.17 ⟩
    ⟨ Check for an odd number of terms and update ps_sqr 9.0.18 ⟩
  end if
  return
end
```

[the_subs.hweb] We will declare the input parameters to the function. The vector a has indices $0, \ldots, n$ and is input to the function. We must also declare a single local loop index.

i is the index of the square term to be computed.

a is the input vector of power series coefficients.

k is a local variable used as a loop index.

⟨ Declare *ps_sqr* variables 9.0.16 ⟩ ≡
```
integer i
floating a(0 : i)
integer k
```

This code is used in section 9.0.15.

[the_subs.hweb] The computation is a rather straightforward adaptation of the *ps_mult* computation. It is made more efficient by pairing the identical terms in the sum so that they are computed only once. The **if** statement is necessary around the **do** to prevent the execution of the **do** loop when computing the coefficient of the 0^{th} term.

⟨ Compute *ps_sqr* sum 9.0.17 ⟩ ≡
```
ps_sqr = a(0) * a(i)
do k = 1, (i − 1) / 2
  ps_sqr = ps_sqr + a(k) * a(i − k)
end do
ps_sqr = const(2.0) * ps_sqr
```

This code is used in section 9.0.15.

[the_subs.hweb] We now need to check for an odd number of terms. If so the middle term has not been added to *ps_sqr* and we need to add it to our sum.

⟨ Check for an odd number of terms and update *ps_sqr* 9.0.18 ⟩ ≡
```
if (mod(i, 2) ≡ 0) then
  ps_sqr = ps_sqr + a(i / 2)^2
end if
```

This code is used in section 9.0.15.

[the_subs.hweb] The power function. To compute the coefficients r_i of the power series resulting from raising a power series a to an arbitrary real power p, we write

$$r = a^p$$

Differentiation yields

$$\dot{r} = pa^{p-1}\dot{a}$$

After multiplying both sides by a we obtain

$$\dot{r}a = pr\dot{a}$$

Expanded out, this is

$$(r_1 + 2r_2t + \ldots + ir_it^{i-1})(a_0 + a_1t + \ldots + a_it^i) = p(r_0 + r_1t + \ldots + r_it^i)(a_1 + 2a_2t + \ldots + ia_it^{i-1})$$

Thus, by equating coefficients for t^{i-1} we have

$$\sum_{k=1}^{i} kr_ka_{i-k} = p\sum_{k=1}^{i} ka_kr_{i-k}$$

Solving this we can obtain the coefficients r_i by the recurrence

$$r_i = \frac{ipa_ir_0 + \sum_{k=1}^{i-1} k(pa_kr_{i-k} - r_ka_{i-k})}{ia_0}$$

"psq.f" 9.0.19 ≡

```
floating function ps_pwr(a, p, i, r)
  ⟨ Declare variables for ps_pwr 9.0.20 ⟩
  if (i ≡ 0) then
    ⟨ Perform the trivial case for ps_pwr 9.0.21 ⟩
  else
    ⟨ Perform the general case for ps_pwr 9.0.22 ⟩
  end if
  r(i) = ps_pwr
  return
end
```

[the_subs.hweb] We will declare the parameters to the function. a is our input power series, p is the power to which we wish to raise a, i is the current term that we are calculating, and r is the vector of previously computed coefficients (r needs to be passed because the previous coefficients are used in computing the next one.) We also declare a single local loop index k.

⟨ Declare variables for ps_pwr 9.0.20 ⟩ ≡

```
implicit none
integer i
floating a(0 : i), p, r(0 : i)
integer k
```

This code is used in section 9.0.19.

[the_subs.hweb] the 0^{th} term is the trivial case, where $r_0 = (a_0)^p$.

⟨ Perform the trivial case for *ps_pwr* 9.0.21 ⟩ ≡
```
ps_pwr = a(0)^p
```

This code is used in section 9.0.19.

[the_subs.hweb] Now, we can perform the calculation for all other cases by the recurrence formula derived above.

⟨ Perform the general case for *ps_pwr* 9.0.22 ⟩ ≡
```
ps_pwr = p * i * a(i) * r(0)
do k = 1, i - 1
  ps_pwr = ps_pwr + (k) * (p * a(k) * r(i - k) - r(k) * a(i - k))
end do
ps_pwr = ps_pwr / (i * a(0))
```

This code is used in section 9.0.19.

[the_subs.hweb] The square root. As a final example, we will compute the square root r of a power series a. (We could use the *ps_pwr* function with $p = 0.5$, but this will be more efficient.) The derivation is reasonably straightforward:

$$r = \sqrt{a}$$

is equivalent to

$$r^2 = a$$

or

$$(r_0 + r_1 t + \ldots + r_i t^i)^2 = a_0 + a_1 t + \ldots + a_i t^i$$

By equating coefficients on t^i we obtain

$$r_0 r_i + r_1 r_{i-1} + \ldots + r_i r_0 = a_i$$

Solving for r_i yields the recurrence

$$r_i = \frac{1}{2r_0}\left[a_i - \sum_{k=1}^{i-1} r_k r_{i-k}\right]$$

with $r_0 = \sqrt{a_0}$. The implementation takes advantage of the symmetry of the sum, like it was done in the *ps_sqr* routine.

"psq.f" 9.0.23 ≡
```
floating function ps_sqrt(a, i, r)
  ⟨ Declare ps_sqrt variables 9.0.24 ⟩
  if (i ≡ 0) then
    ⟨ Perform the zero case for ps_sqrt 9.0.25 ⟩
  else
    ⟨ Perform the general case for ps_sqrt 9.0.26 ⟩
  end if
  r(i) = ps_sqrt
  return
end
```

[the_subs.hweb] We will declare the parameters to the function. a is the zero-indexed vector of coefficients for the power series we are taking the square root of, i is the current term that we are calculating, and r is the vector of previous results (which needs to be passed because the previous coefficients are used in computing the next one.) We also declare a single local loop index.

i is the index of the square root term to be computed.

a and r are the zero-indexed vectors described above and their upper limit is i.

k is a local variable used as a loop index.

⟨ Declare *ps_sqrt* variables 9.0.24 ⟩ ≡
```
implicit none
integer i
floating a(0 : i), r(0 : i)
integer k
```

This code is used in section 9.0.23.

[the_subs.hweb] The 0^{th} term is a trivial case, where $r_0 = \sqrt{a_0}$.

⟨ Perform the *zero* case for *ps_sqrt* 9.0.25 ⟩ ≡
```
ps_sqrt = sqrt(a(0))
```

This code is used in section 9.0.23.

[the_subs.hweb] The general case is not quite straightforward. We have a special form when $i = 1$ and have to check for the odd central term because we avoid summing both ends of the sum to take advantage of the symmetry.

⟨ Perform the general case for *ps_sqrt* 9.0.26 ⟩ ≡
```
if (i ≡ 1) then
  ps_sqrt = const(0.5) * a(1) / r(0)
else
  ps_sqrt = const(0.0)
  do k = 1, (i − 1) / 2
    ps_sqrt = ps_sqrt + r(k) * r(i − k)
  end do
  if (mod(i, 2) ≡ 0) then
    ps_sqrt = ps_sqrt + const(0.5) * r(i / 2)^2
  end if
  ps_sqrt = (const(0.5) * a(i) − ps_sqrt) / r(0)
end if
```

This code is used in section 9.0.23.

9.1 Trigonometric Functions

[the_subs.hweb] To compute the trigonometric functions of a power series we will use a "trick" similar to that of the exponential function. We derive the sine and cosine of a power series as follows: If

$$s = \sin(a)$$

and

$$c = \cos(a)$$

then by differentiating, we obtain

$$\dot{s} = \cos(a)\dot{a} = c\dot{a}$$

and

$$\dot{c} = -\sin(a)\dot{a} = -s\dot{a}$$

Expanded, these become

$$s_1 + 2s_2t + \ldots + is_it^{i-1} = (c_0 + c_1t + \ldots + c_it^i)(a_1 + 2a_2t + \ldots + ia_it^{i-1})$$

and

$$c_1 + 2c_2t + \ldots + ic_it^{i-1} = -(s_0 + s_1t + \ldots + s_it^i)(a_1 + 2a_2t + \ldots + ia_it^{i-1})$$

If we equate the coefficients of t^{i-1} on either side we obtain the recurrences:

$$is_i = ia_ic_0 + (i-1)a_{i-1}c_1 + \ldots + a_1c_{i-1}$$

and

$$-ic_i = ia_is_0 + (i-1)a_{i-1}s_1 + \ldots + a_1s_{i-1}$$

with

$$s_0 = \sin(a_0)$$

and

$$c_0 = \cos(a_0)$$

It is thus convenient to calculate the sine and cosine series together, although only one of them might be present in the equation.

[the_subs.hweb] Now for the FORTRAN code. This will be a subroutine because we actually calculate (and in a sense return) two values with each call, namely $s(i)$ and $c(i)$.

"psq.f" 9.1.1 ≡

```
subroutine ps_trig(a, i, s, c)
  implicit none
  ⟨ Declare ps_trig variables 9.1.2 ⟩
  if (i ≡ 0) then
    ⟨ Do the trivial case for ps_trig 9.1.3 ⟩
  else
    ⟨ Do the general case for ps_trig 9.1.4 ⟩
  end if
  return
end
```

[the_subs.hweb] The arguments must be declared. They are: a, the argument of the sine and cosine functions; s and c, the trig functions; and k, the index of the coefficients being calculated. The first three of these are power series and therefore have a lowest subscript of **zero**. k is used as a loop control variable.

⟨ Declare *ps_trig* variables 9.1.2 ⟩ ≡
```
integer i
floating a(0 : i), s(0 : i), c(0 : i)
integer k
```

This code is used in section 9.1.1.

[the_subs.hweb] When calculating the zeroth coefficient, we use the library functions.

⟨ Do the trivial case for *ps_trig* 9.1.3 ⟩ ≡
```
s(0) = sin(a(0))
c(0) = cos(a(0))
```

This code is used in section 9.1.1.

[the_subs.hweb] The general coefficient, *i.e.* when $i \neq 0$, requires a multiplication of power series operators. Remember that we are calculating sin and cos both because they are dependent upon each other.

⟨ Do the general case for *ps_trig* 9.1.4 ⟩ ≡
```
s(i) = c(i − 1) ∗ a(1)
c(i) = s(i − 1) ∗ a(1)
do k = 2, i
  s(i) = s(i) + c(i − k) ∗ (k ∗ a(k))
  c(i) = c(i) + s(i − k) ∗ (k ∗ a(k))
end do
s(i) = s(i) / i
c(i) = −c(i) / i
```

This code is used in section 9.1.1.

10 Additional work on the power series operators

[the_subs.hweb] There are a few more operations that are needed for some applications. They may not be added in the next month or so, but they are planned. These include *ps_shift* which would allow the multiplication or division of a power series by the independent variable to an arbitrary integral power, like in Bessel's equations. Of course, this can be done by defining that operand in terms of the *ps_mult* or *ps_div* operations and using a series (like x in the tests above). That would not give maximum efficiency and can cause some more problems when considering singularities.

Another routine more in line with those already shown is integration, like in integro-differential equations. This has been done but we want to let it settle in out minds and use it in several samples before releasing its format and implied need of maintenance for a reasonable future.

Gibbons also included a *reciprocal* routine. We have chosen not to implement it because our target is differential equations and such terms are rare in the ones we use. Gibbons included a ***log*** routine which we probably need but have not implemented.

June 1, 1990.

11 References

[the_subs.hweb] The following references are available to some extent and show a number of uses and background work. They are annotated to a slight extent and additional references are welcome.

Corliss, G. F., and Chang, Y. F. Solving Ordinary Differential Equations using Taylor Series, *ACM Transactions on Mathematical Software*, vol. 8, no. 2, pp. 114-144 (1982). *Chang and Corliss have written several articles over the past few years applying power series methods to systems of ODEs. Their work has focused mostly on the series analysis of long power series. This has important theoretical use, but the empirical evidence we have obtained indicates that long power series are not needed in the types of problems commonly encountered.*

Doiron, H. H. Numerical Integration via Power Series Expansions, M.S. Thesis, University of Houston, Houston, Texas, August 1967. *The primary results in this thesis show that these integration methods are fast. These and other tests usually show that the use of these methods saves approximately 80 percent of the CPU time. It uses some archaic data structures rather to do some of these functions (like sine and cosine). The algorithms here are superior.*

Fehlberg, E. Numerical Integration of Differential Equations by Power Series Expansions, Illustrated by Physical Examples. NASA Technical Note No. TN D-2356, October 1964. *The two examples are a restricted three-body problem and the motion of an electron in the field of a magnetic dipole. The results indicate a required CPU time of 15 to 20 percent of the Runge-Kutta-Nyström method.*

Gibbons, A. A Program for the Automatic Integration of Differential Equations using the Method of Taylor Series, *Computer Journal*, vol. 3, pp. 108-111 (1960). *A good source of algorithms and discussion of the procedures. The term "practical radius of convergence" is introduced which is like the "numeric continuation" term used in this paper.*

Henrici, P. Automatic Computations with Power Series, *Journal of the ACM*, Vol. 3, no. 1, (1956). *An early reference of doing similar work in a symbolic fashion. His operations are not as efficient as those described in this paper. Applications included are combinatorial analysis, asymptotic expansions, computing Legendre polynomials, and solving differential equations.*

Moore, R. E. *Interval Analysis*, Englewood Cliffs, New Jersey: Prentice-Hall, 1966. *Moore used the control of the local truncation error in solving differential equations by power series as an application of interval analysis. He gave heruistics for choosing step-sizes and series length for this method.*

There are many other references to the use of power series as integrators but they don't really affect the spirit of this work. We will include them in the next revision.

12 A constrained least squares

[gj_svd.hweb] This routine will solve a system of simultaneous equations. If the system is overdetermined, it will reduce the equations which are to be met exactly (within machine precision) and then form the normal equations. It will do full pivoting and a call to the **subroutine** *svd*.

This routine will also perform some initializations for the call to the **subroutine** *stanal*. (This has not yet been converted to the WEB form.)

When the entire code is rewritten using current compilers such as C++ and/or Fortran 90, we will be considering using Bai's version from LaPack as the base.

12.1 Gauss Jordan reduction with least squares

[gj_svd.hweb] This subroutine has a large number of arguments. Some of these are obvious and some require a bit more explanation. We also merge into this list some **parameter**s.

A is the FORTRAN name of the augmented coefficient matrix, A.

max_n_bc is the number of rows in the type statement declaring A. **parameter**

max_n_p is the number of columns in the dimension of the coefficient matrix A while *max_n* is the maximum order of the system $max\_n\_p = max\_n + 1$. **parameter**s

n is the order of the system. It is one less than the number of actual columns used in A, since the right hand side is an augmented column.

number_exact is the number of constraints. This is the number of equations to be met exactly while the rest of the equations are met in a least squares fit.

i_debug is normally zero. Positive values will cause different levels of output which are handy in debugging a new application of the routines.

i_return should return with a zero value if the solution is successful. A value of one is considered a warning. Larger positive values should be considered as fatal errors. The routines need extensive work to put some form of a condition number into the implementation.

[gj_svd.hweb] One of the great advantages of the WEB style of literate programming is that the global view of the code can appear in one module:

"psq.f" 12.1.1 ≡
```
subroutine gj_svd(A, num_rows_A, n, number_exact, i_debug, i_return)
  implicit none
  ⟨ Declare variables for gj_svd 12.1.2 ⟩
  ⟨ Check parameters and initialize variables for gj_svd 12.1.5 ⟩
  k = 0
  do while(k ≤ n)
    ⟨ Debug output? Reducing k 13.1.1 ⟩
    if (k ≡ (number_exact + 1) ∧ n < num_rows_A) then
      ⟨ This is the first observation, call svd for the rest 13 ⟩
      k = n + 1
    else
      ⟨ Find max pivot element 12.1.9 ⟩
      ⟨ Swap column and row pointers 12.1.10 ⟩
      ⟨ Divide Row by Max element 12.1.11 ⟩
      ⟨ Perform in place reduction 12.1.12 ⟩
      k = k + 1
    end if
  end do
  ⟨ Permute the solution matrix into original form 12.1.13 ⟩
  ⟨ Output the whole mess 13.0.6 ⟩
  i_return = 0
  return
end
```

[gj_svd.hweb] First we will declare the parameters for this **subroutine**. The equations will be partitioned as:

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \begin{Bmatrix} \beta_e \\ \beta_l \end{Bmatrix} = \begin{Bmatrix} d_e \\ d_l \end{Bmatrix}$$

where A_1 will be a square and order n. This will be transformed into the equivalent form:

$$\begin{bmatrix} I & A_2^{'} \\ 0 & A_4^{'} \end{bmatrix} \begin{Bmatrix} \beta_e \\ \beta_l \end{Bmatrix} = \begin{Bmatrix} d_e^{'} \\ d_l^{'} \end{Bmatrix}$$

The prime superscripts indicate those elements have changed in the reduction process. The value of *final* is set to true on the last iteration if statistics are to be displayed.

⟨ Declare variables for *gj_svd* 12.1.2 ⟩ ≡
```
integer num_rows_A, n, number_exact, i_debug, i_return
logical final
```

See also sections 12.1.3 and 12.1.4.

This code is used in section 12.1.1.

[gj_svd.hweb] Now we will need some local variables. i,j,and k are counters. kk is a temporary variable used on the innermost loops when multiplying arrays. The variables *row_ptr* and *col_ptr* are pointers used in swapping rows and columns. The variable, *value*, is the element we are currently comparing. The maximum value is stored in *pivot_value*.

⟨ Declare variables for *gj_svd* 12.1.2 ⟩ +≡
```
integer max_n, m, max_n_bc, max_n_p
parameter (m = 12, max_n = 15, max_n_p = 16, max_n_bc = 400)
integer row_ptr(0 : max_n_bc), col_ptr(0 : max_n_p), i, j, k, kk
floating A(0 : max_n_bc, 0 : max_n_p)
integer max_row_pivot_select, max_row_elimination
integer row_k, col_k, row_kk, col_kk, i_ls, j_ls, rows_ls, cols_ls, ierr
floating value, pivot_value
logical over
character format_pivot*64, format_matrix*64, format_solution*64
```

[gj_svd.hweb] These arrays are used to pass to *svd*.

⟨ Declare variables for *gj_svd* 12.1.2 ⟩ +≡
```
floating A_4(1 : max_n_bc, 1 : max_n_p), U(1 : max_n_bc, 1 : max_n_p)
floating V(1 : max_n_bc, 1 : max_n_p), sigma(max_n), work(max_n)
floating A_4_t_A_4_i(max_n, max_n)
```

[gj_svd.hweb] There are a few consistency checks that we should make. These take so little time and can point out errors that are otherwise hard to catch.

⟨ Check parameters and initialize variables for *gj_svd* 12.1.5 ⟩ ≡
```
⟨ Check for non-fatal and fatal sizes of A 12.1.6 ⟩
if (i_return > 1) then
  return
end if
```

See also sections 12.1.7 and 12.1.8.

This code is used in section 12.1.1.

[gj_svd.hweb] These are fairly self-explanatory. These would make a little more sense when this code is rewritten to use strictly calling arguments and avoid the use of **parameter** statements. All these checks result in fatal messages.

⟨ Check for non-fatal and fatal sizes of A 12.1.6 ⟩ ≡
```
if (n > max_n_bc) then
  write (*, *) ' Order higher than row dimension ', n, max_n_bc
  i_return = 2
end if
if (n ≥ max_n_p) then
  write (*, *) ' Order higher than column dimension ', n, max_n_p
  i_return = 2
end if
if (num_rows_A > max_n_bc) then
  write (*, *) ' More rows than row dimension ', num_rows_A, max_n_bc
  i_return = 2
end if
if (n > num_rows_A) then
  write (*, *) ' Order more than number of rows ', n, num_rows_A
  i_return = 2
end if
```

This code is used in section 12.1.5.

[gj_svd.hweb] We need to define the formats we will use.

⟨ Check parameters and initialize variables for *gj_svd* 12.1.5 ⟩ +≡
```
format_pivot = '(20x,i5,''th pivot '',g15.7,2i5)'
format_matrix = '(1x,i3,(t5,7g10.4))'
format_solution = '(1x,g11.4,(t12,6g11.4))'
```

[gj_svd.hweb] Now, we need to make some initializations of the row and column index vectors. The nature of some of the problems we intend to solve may have constraints in the parameters of the ODEs. We will always place these parameters in the higher order of the state vector. This may create a few exact equations that necessitate maximum pivot selection in the first part of the process. Further, after we have put the normal equations in place, we no longer have to do elimination on an overdetermined set.

⟨ Check parameters and initialize variables for *gj_svd* 12.1.5 ⟩ +≡
```
max_row_pivot_select = number_exact
max_row_elimination = num_rows_A
do i = 0, max_n_bc
  row_ptr(i) = i
end do
do j = 0, max_n_p
  col_ptr(j) = j
end do
```

[gj_svd.hweb] We first need to loop throught the rows finding the column with the maximum pivot element.

⟨ Find max pivot element 12.1.9 ⟩ ≡
```
pivot_value = 0.0
do i = k, max_row_pivot_select
  do j = k, n
    if (abs(A(row_ptr(i), col_ptr(j))) > abs(pivot_value)) then
      row_k = i
      col_k = j
      pivot_value = A(row_ptr(i), col_ptr(j))
    end if
  end do
end do
```

This code is used in section 12.1.1.

[gj_svd.hweb] Now, we need to swap our pointers for moving the row and column with the maximum element.

⟨ Swap column and row pointers 12.1.10 ⟩ ≡
```
row_kk = row_ptr(row_k)
row_ptr(row_k) = row_ptr(k)
row_ptr(k) = row_kk
col_kk = col_ptr(col_k)
col_ptr(col_k) = col_ptr(k)
col_ptr(k) = col_kk
```

This code is used in section 12.1.1.

[gj_svd.hweb] Next, we can divide the row by the maximum element in order to get unity on the diagonal of the matrix.

⟨ Divide Row by Max element 12.1.11 ⟩ ≡
```
if (i_debug > 1) then
  write(*, format_pivot) k, pivot_value, row_kk, col_kk
end if
A(row_kk, col_ptr(k)) = const(1.0)
do j = k + 1, n + 1
  A(row_kk, col_ptr(j)) = A(row_kk, col_ptr(j)) / pivot_value
end do
```

This code is used in section 12.1.1.

[gj_svd.hweb] Now we can reduce the matrix, and produce the Identity matrix needed for the least squares method. It may seem like a wasted action to set the below diagonal values to zero, but this is done for aid in debugging.

⟨ Perform in place reduction 12.1.12 ⟩ ≡
```
do i = 0, max_row_elimination
  if (i ≠ k) then
    value = A(row_ptr(i), col_kk)
    A(row_ptr(i), col_kk) = const(0.)
    do kk = k + 1, n + 1
      A(row_ptr(i), col_ptr(kk)) = A(row_ptr(i), col_ptr(kk)) − value ∗ A(row_kk, col_ptr(kk))
    end do
  end if
end do
```

This code is used in section 12.1.1.

[gj_svd.hweb] Now, we are ready to permute the solution matrix such that the original rows are in the correct position. This is necessary because we had to do maximum pivoting and it is possible that some columns may have been interchanged. In the call to these routines with the last iteration, we don't actually solve the equations. We return with the inverse of the cTc matrix in the lower right square part. Also, there must be an identity in the upper left part. The upper right and lower right parts are used in establishing the covarience matrices and so this permuting is necessary.

⟨ Permute the solution matrix into original form 12.1.13 ⟩ ≡
```
do k = 0, n
  if (row_ptr(k) ≠ col_ptr(k)) then
    ⟨ Swap rows in the augmented matrix 12.1.14 ⟩
  end if
  ⟨ Update the row pointers 12.1.15 ⟩
end do
```

This code is used in section 12.1.1.

[gj_svd.hweb] This **do** loop is **not wasted** as we explained in the previous module. It also happens that swapping the rows to end up with an identity was an aid during the debugging stages.

⟨ Swap rows in the augmented matrix 12.1.14 ⟩ ≡
```
do j = 0, n + 1
  value = A(row_ptr(k), j)
  A(row_ptr(k), j) = A(col_ptr(k), j)
  A(col_ptr(k), j) = value
end do
```

This code is used in section 12.1.13.

[gj_svd.hweb] This is simply necessary bookkeeping. The loop may look strange at first, but the incrementing of the index takes us one too far, so we subtract one after the loop.

⟨ Update the row pointers 12.1.15 ⟩ ≡
```
i = k
over = F
do while((i ≤ n) ∧ (¬over))
  if (row_ptr(i) ≡ col_ptr(k)) then
    over = T
  end if
  i = i + 1
end do
i = i − 1
row_ptr(i) = row_ptr(k)
row_ptr(k) = i
```

This code is used in section 12.1.13.

13 Do the *svd* stuff

[gj_svd.hweb] When the index of the reduction process, k, points at the first observation (a boundary condition to be met in a least squares sense) then we **call** *svd* for the rest. We don't do this unless there are more observations than needed so a best fit will be meaningful.

⟨ This is the first observation, **call** *svd* for the rest 13 ⟩ ≡
```
rows_ls = num_rows_A − k + 1
cols_ls = n − k + 1
⟨ Copy the least squares part of A to A_4 13.0.1 ⟩
call svd(max_n_bc, rows_ls, cols_ls, ierr, A_4, sigma, U, V, work, T, T)
⟨ Perform appropriate multiplications by U and V 13.0.2 ⟩
⟨ Copy the inverse of the normal matrix into A 13.0.4 ⟩
⟨ Finish it up by back substitution 13.0.5 ⟩
```

This code is used in section 12.1.1.

[gj_svd.hweb] We need only the least squares part copied into A_4. We must be careful and use the pointers into the original matrix.

⟨ Copy the least squares part of A to A_4 13.0.1 ⟩ ≡
```
i_ls = 0
do i = k, num_rows_A
  i_ls = i_ls + 1
  j_ls = 0
  do j = k, n + 1
    j_ls = j_ls + 1
    A_4(i_ls, j_ls) = A(row_ptr(i), col_ptr(j))
  end do
end do
```

This code is used in section 13.

[gj_svd.hweb] First, we get the solution of the overdetermined equations by creating *work* by postmultiplying U^T by the rhs. Then we postmultiply V by *work* and include the Σ^{-1} (remember that Σ is diagonal.)

⟨ Perform appropriate multiplications by U and V 13.0.2 ⟩ ≡

```
do i = 1, cols_ls
  work(i) = 0.0
  do j = 1, rows_ls
    work(i) = work(i) + U(j, i) * A_4(j, cols_ls + 1)
  end do
end do

do i = 1, cols_ls
  A_4(i, cols_ls + 1) = 0.0
  do j = 1, cols_ls
    A_4(i, cols_ls + 1) = A_4(i, cols_ls + 1) + V(i, j) * work(j) / sigma(j)
  end do
end do
```

See also section 13.0.3.

This code is used in section 13.

[gj_svd.hweb] Calculate the inverse of the coefficient matrix of the normal equations. A little algebra from the definition of U and V

$$A = U\Sigma V^T$$

gives

$$(A^T A)^{-1} = V\Sigma^{-2}V^T$$

Since Σ is diagonal and U and V are orthogonal, it is rather straightforward.

⟨ Perform appropriate multiplications by U and V 13.0.2 ⟩ +≡

```
do i = 1, cols_ls
  do j = 1, cols_ls
    A_4_t_A_4_i(i, j) = 0.0
    do kk = 1, cols_ls
      A_4_t_A_4_i(i, j) = A_4_t_A_4_i(i, j) + (V(i, kk) * V(j, kk)) / sigma(kk)^2
    end do
  end do
end do
```

[gj_svd.hweb] Now that we have finished calculating, we need to copy this back to where it will be used.

⟨ Copy the inverse of the normal matrix into A 13.0.4 ⟩ ≡

```
i_ls = 0
do i = k, num_rows_A
  i_ls = i_ls + 1
  j_ls = 0
  do j = k, n + 1
    j_ls = j_ls + 1
    if (j > n) then
      A(row_ptr(i), col_ptr(j)) = A_4(i_ls, j_ls)
    else
      A(row_ptr(i), col_ptr(j)) = A_4_t_A_4_i(i_ls, j_ls)
    end if
  end do
end do
```

This code is used in section 13.

[gj_svd.hweb] Now we have a little back substitution that is to be finished.

⟨ Finish it up by back substitution 13.0.5 ⟩ ≡

```
do i = 0, k − 1
  do j = k, n
    A(row_ptr(i), n + 1) = A(row_ptr(i), n + 1) − A(row_ptr(i), col_ptr(j)) ∗ A(row_ptr(j), n + 1)
  end do
end do
if (i_debug > 0) then
  write (∗, ∗) 'Solution:', (A(i, n + 1), i = 0, n)
end if
```

This code is used in section 13.

[gj_svd.hweb] This needs to be modified to include *debug* options rather than always outputting.

⟨ Output the whole mess 13.0.6 ⟩ ≡

```
if (i_debug ≥ 2) then
  if (ierr ≠ 0)
    write (∗, '('' trouble. ierr='', i4)') ierr
  write (∗, '("Sigma",7f10.6)') (sigma(j), j = 1, cols_ls)
  write (∗, ∗) "U"
  do i = 1, rows_ls
    write (∗, '(7f10.6)') (U(i, j), j = 1, cols_ls)
  end do
  write (∗, ∗) "V"
  do i = 1, cols_ls
    write (∗, '(7f10.6)') (V(i, j), j = 1, cols_ls)
  end do
end if
```

This code is used in section 12.1.1.

13.1 More debug output

[gj_svd.hweb] This module is used by several of the following ones.

The coefficient matrix is output with the row number. Notice that the number of rows output may change.

⟨ Output the coefficient matrix 13.1 ⟩ ≡
```
do i = 0, max_row_elimination
  write(*, format_matrix) i, (A(i, j), j = 0, n + 1)
  if (i ≡ number_exact)
    write(*, *) ' ^ Constraints ^ / v Observations v'
end do
```

This code is used in section 13.1.1.

[gj_svd.hweb] This is one of the most verbose requests that can be made. The augmented coefficient matrix will be output and then again ...

⟨ Debug output? Reducing k 13.1.1 ⟩ ≡
```
if (i_debug > 3 | (i_debug ≡ 3 ∧ i ≡ 1)) then
  write(*, *) ' Reducing k =', k
  ⟨ Output the coefficient matrix 13.1 ⟩
end if
```

This code is used in section 12.1.1.

14 The subroutine SVD

[svd.hweb] This subroutine is a translation of the ALGOL procedure svd, *Num. Math.* 14, 403-420(1970) by Golub and Reinsch. Also see *Handbook for Automatic Computation*, vol. ii-linear algebra, 134-151(1971). It was modified and adapted to the WEB programming style. Its modification is rather imcomplete at this stage in that it still suffers from the FORTRAN tradition of short names, no underscores, ... The actual source used came from the text by Forsythe, Malcolm, and Moler.

```
"psq.f" 14 ≡
    subroutine svd(nm, m, n, ierr, A, w, U, V, rv1, matv, matu)
      implicit none
      integer mn, m, n, ierr
      real*8 A(nm, n), w(n), U(nm, n), V(nm, n), rv1(n)
      logical matv, matu, test_split, test_conv

      ⟨ Subroutine svd local variables 14.2 ⟩
      ⟨ Set Initial Values 14.3 ⟩
      ⟨ Copy the Array 14.1 ⟩
      ⟨ Do Householder Reduction 14.4 ⟩
      if (matu) then
        ⟨ Create Right Singular Vector 14.19 ⟩
      end if
      if (matv) then
        ⟨ Create Left Singular Vectors 14.24 ⟩
      end if
      ⟨ Make Bidiagonal Matrix 14.32 ⟩

      return
    end
```

[svd.hweb] This module is used to copy the array A into array U for all subsequent manipulations.

```
⟨ Copy the Array 14.1 ⟩ ≡
    do i = 1, m
      do j = 1, n
        U(i, j) = A(i, j)
      end do
    end do
```

This code is used in section 14.

[svd.hweb] These are the variables used only in this subroutine. I wanted to use the lower case 'L' but it is just too much like a one. Thus, is used *el*.

```
⟨ Subroutine svd local variables 14.2 ⟩ ≡
    integer i, j, k, el, nm, its
    real*8 c, f, g, h, s, x, y, z, scale, anorm
    real*8 zero, one, two
    parameter (zero = 0.0 · 10^0D, one = 1.0 · 10^0D, two = 2.0 · 10^0D)
```

This code is used in section 14.

[svd.hweb] This module sets the initial values of *ierr*, *g*, *scale* and *anorm* to zero. The variable *ierr* is used to indicate that the matrix is non-singular, *scale* is the value used to scale the original array and *anorm* is used to keep track of the largest element in the diagonal or super diagonal. At first glance it appears redundant to initialize *g* and *scale*, but that is not the case.

⟨ Set Initial Values 14.3 ⟩ ≡
```
ierr = 0
g = 0.0
scale = 0.0
anorm = 0.0
```

This code is used in section 14.

[svd.hweb] Householder reduction to bidiagonal form. Householder reduction must first do a column transformation and then a row transformation repeating these steps until the matrix is bidiagonal.

⟨ Do Householder Reduction 14.4 ⟩ ≡
```
do i = 1, n
  ⟨ Transform by Column 14.5 ⟩
  ⟨ Transform by Row 14.11 ⟩
end do
```

This code is used in section 14.

[svd.hweb] The transformations of the column values is done in this module. The transforming matrix U, form the text, is not really created and the transformation is done so to say *in place.* First the array *u* values are scaled by a factor equal to the sun of the absolute values of all the values in the column being transformed. This value is stored in *scale* and will be used to restore the values once the transformation is completed. The variable *l* is used as a holding variable to start loops at the *i*th plus 1 value in the array.

⟨ Transform by Column 14.5 ⟩ ≡
```
el = i + 1
rv1(i) = scale * g
⟨ Reset Zeros 14.18 ⟩
if (i ≤ m) then
  ⟨ Calculate column scale factor 14.6 ⟩
  if (scale ≠ zero) then
    ⟨ Scale column 14.7 ⟩
    ⟨ Prepare column transformation values 14.8 ⟩
    if (i ≠ n) then
      ⟨ Apply column transform 14.9 ⟩
    end if
    ⟨ Restore column values 14.10 ⟩
  end if
end if
```

This code is used in section 14.4.

[svd.hweb] The scaling factor *scale* is just the positive sum of the absolute values of the column entries from i, the count of the transformation pair that is being done at the moment, to m the total number of columns.

⟨ Calculate column scale factor 14.6 ⟩ ≡
```
do k = i, m
  scale = scale + abs(U(k, i))
end do
```

This code is used in section 14.5.

[svd.hweb] This module scales the column by *scale* and then totals the square of the scaled values in the column from i to m to be used in the calculation of the length of the vector.

⟨ Scale column 14.7 ⟩ ≡
```
do k = i, m
  U(k, i) = U(k, i) / scale
  s = s + U(k, i)^2
end do
```

This code is used in section 14.5.

[svd.hweb] This module calculates the value that the current column components will be multiplied by to get the value of the component in the column being transformed. The variable f is just the value of the diagonal entry in the column being transformed, g is the calculated length of the vector. To insure that the sign of g is proper this is done within a call to the intrinsic function ***sign***. First the squared values of the entries is accumulated in s. The call to ***sqrt***, using s as its parameter, is used as the first parameter in the call to ***sign***. The sign of the value that is returned from ***sign*** is dictated by the sign of the entry $U(i,\ i)$. Recall that the transforming matrix U is only theoretically calculated by the formula

$$U_t = I - \beta u_t u_t^T, \qquad t = 1, \ldots n,$$

Where t is the number of the column transformation being done and n is the number of columns in the matrix.

⟨ Prepare column transformation values 14.8 ⟩ ≡
```
f = U(i, i)
g = -sign(sqrt(s), f)
h = f * g - s
U(i, i) = f - g
```

This code is used in section 14.5.

[svd.hweb] The transformation is applied to the existing values of the matrix in this module. remember that these transformations are performed *in place*, none of the matrices talked about in the text are actually formed.

```
⟨ Apply column transform 14.9 ⟩ ≡
    do j = el, n
      s = zero
      do k = i, m
        s = s + U(k, i) * U(k, j)
      end do
      f = s / h
      do k = i, m
        U(k, j) = U(k, j) + f * U(k, i)
      end do
    end do
```

This code is used in section 14.5.

[svd.hweb] This module uses the scaling factor *scale* to restore the values in the matrix to their prescaled values.

```
⟨ Restore column values 14.10 ⟩ ≡
    do k = i, m
      U(k, i) = scale * U(k, i)
    end do
```

This code is used in section 14.5.

[svd.hweb] This is the same process as above except that transformations are applied to the rows to introduce zeros into the matrix above the superdiagonal.

```
⟨ Transform by Row 14.11 ⟩ ≡
    w(i) = scale * g
    ⟨ Reset Zeros 14.18 ⟩
    if (i ≤ m ∧ i ≠ n) then
      ⟨ Calculate row scale factor 14.12 ⟩
      if (scale ≠ zero) then
        ⟨ Scale row 14.13 ⟩
        ⟨ Prepare row transformation values 14.14 ⟩
        if (i ≠ m) then
          ⟨ Apply row transformation 14.15 ⟩
        end if
        ⟨ Restore row values 14.16 ⟩
      end if
    end if
    ⟨ Set anorm 14.17 ⟩
```

This code is used in section 14.4.

[svd.hweb] This module calculates the scaling factor for the row vector.

⟨ Calculate row scale factor 14.12 ⟩ ≡

```
do k = el, n
  scale = scale + abs(U(i, k))
end do
```

This code is used in section 14.11.

[svd.hweb] This module scales the row.

⟨ Scale row 14.13 ⟩ ≡

```
do k = el, n
  U(i, k) = U(i, k) / scale
  s = s + U(i, k)^2
end do
```

This code is used in section 14.11.

[svd.hweb] This module calculates the transformation values.

⟨ Prepare row transformation values 14.14 ⟩ ≡

```
f = U(i, el)
g = -sign(sqrt(s), f)
h = f * g - s
U(i, el) = f - g
do k = el, n
  rv1(k) = U(i, k) / h
end do
```

This code is used in section 14.11.

[svd.hweb] This module applies the transform to the row.

⟨ Apply row transformation 14.15 ⟩ ≡

```
do j = el, m
  s = zero
  do k = el, n
    s = s + U(j, k) * U(i, k)
  end do
  do k = el, n
    U(j, k) = U(j, k) + s * rv1(k)
  end do
end do
```

This code is used in section 14.11.

[svd.hweb] This module restores the row components to their unscaled values.

⟨ Restore row values 14.16 ⟩ ≡
```
do k = el, n
  U(i, k) = scale * U(i, k)
end do
```

This code is used in section 14.11.

[svd.hweb] This module places the value of the largest component in *anorm*

⟨ Set anorm 14.17 ⟩ ≡
```
anorm = max1(anorm, abs(w(i)) + abs(rv1(i)))
```

This code is used in section 14.11.

[svd.hweb] The variables s, g end and *scale* are as a group set to zero several times during the process of *householder reduction* so I have put that process in a separate module.

⟨ Reset Zeros 14.18 ⟩ ≡
```
g = zero
s = zero
scale = zero
```

This code is used in sections 14.5 and 14.11.

[svd.hweb] If *matv* is set to true the right singular vector array is wanted and this module is used to initialize its components so that it can be correctly calculated when doing the QR reduction on the bidiagonal matrix. Accumulation of values is sort of reverse engineered from the computed values of the singular values. This is why the inner loop goes from n down to 1.

⟨ Create Right Singular Vector 14.19 ⟩ ≡
```
do i = n, 1, −1
  if (i ≠ n) then
    if (g ≠ zero) then
      ⟨ right double division 14.20 ⟩
      ⟨ set matrix V values 14.21 ⟩
      ⟨ zero V off-diagonal elements 14.22 ⟩
    end if
    ⟨ zero V off-diagonal elements 14.22 ⟩
  end if
  ⟨ set matrix V diagonal values 14.23 ⟩
end do
```

This code is used in section 14.

[svd.hweb] Double division is used to avoid underflow.

⟨ right double division 14.20 ⟩ ≡
```
do j = el, n
  V(j, i) = (U(i, j) / U(i, el)) / g
end do
```

This code is used in section 14.19.

[svd.hweb] This module establishes the initial values of the right singular vectors. These values are then used by the QR transformation process to compute the final values of the right singular vector.

⟨ set matrix V values 14.21 ⟩ ≡
```
do j = el, n
  s = zero
  do k = el, n
    s = s + U(i, k) * V(k, j)
  end do
  do k = el, n
    V(k, j) = V(k, j) + s * V(k, i)
  end do
end do
```

This code is used in section 14.19.

[svd.hweb] The off-diagonal values are set to zero where the value of the calculated superdiagonal is equal to zero and in all columns and rows $> m$.

⟨ zero V off-diagonal elements 14.22 ⟩ ≡
```
do j = el, n
  V(i, j) = zero
  V(j, i) = zero
end do
```

This code is used in section 14.19.

[svd.hweb] The value of the diagonal are set to 1.0.

⟨ set matrix V diagonal values 14.23 ⟩ ≡
```
V(i, i) = one
g = rv1(i)
el = i
```

This code is used in section 14.19.

[svd.hweb] If *matu* is set to true the left singular vector array is wanted and this module is used to initialize its components so that it can be correctly calculated when doing the QR reduction on the bidiagonal matrix. Accumulation of values is sort of reverse engineered from the computed values of the singular values. This is why the inner loop goes from mn down to 1. The value of mn is the minimum of the extents of A.

```
⟨ Create Left Singular Vectors 14.24 ⟩ ≡
    mn = min(m, n)
    do i = mn, 1, −1
      ⟨ set left loop initial values 14.25 ⟩
      if (i ≠ n) then
        ⟨ Set a row of U to zero 14.26 ⟩
      end if
      if (g ≠ zero) then
        if (i ≠ mn) then
          ⟨ set matrix U values 14.28 ⟩
        end if
        ⟨ scale matrix U columns 14.30 ⟩
      else
        ⟨ zero matrix U column values 14.27 ⟩
      end if
      ⟨ final diagonal value 14.31 ⟩
    end do
```

This code is used in section 14.

[svd.hweb] This module establish the initial values used in calculating the values to be

```
⟨ set left loop initial values 14.25 ⟩ ≡
    el = i + 1
    g = w(i)
```

This code is used in section 14.24.

[svd.hweb] Initially the values of the row components for $j > m$.

```
⟨ Set a row of U to zero 14.26 ⟩ ≡
    do j = el, n
      U(i, j) = zero
    end do
```

This code is used in section 14.24.

[svd.hweb] Initially the values of the column components for $j \leq m$ for all singular values equal to zero.

```
⟨ zero matrix U column values 14.27 ⟩ ≡
    do j = i, m
      U(j, i) = zero
    end do
```

This code is used in section 14.24.

[svd.hweb] This module establishes the initial values of the left singular vectors. These values are then used by the QR transformation process to compute the final values of the left singular vector.

```
⟨ set matrix U values 14.28 ⟩ ≡
    do j = el, n
      s = zero
      do k = el, m
        s = s + U(k, i) * U(k, j)
      end do
      ⟨ left double division 14.29 ⟩
      do k = i, m
        U(k, j) = U(k, j) + f * U(k, i)
      end do
    end do
```

This code is used in section 14.24.

[svd.hweb] This double division should prevent underflow in some cases.

```
⟨ left double division 14.29 ⟩ ≡
    f = (s / U(i, i)) / g
```

This code is used in section 14.28.

[svd.hweb] The values of U are scaled by g which is set equal to the calculated singular values.

```
⟨ scale matrix U columns 14.30 ⟩ ≡
    do j = i, m
      U(j, i) = U(j, i) / g
    end do
```

This code is used in section 14.24.

[svd.hweb] One is added to all calculated left singular vector values.

```
⟨ final diagonal value 14.31 ⟩ ≡
    U(i, i) = U(i, i) + one
```

This code is used in section 14.24.

[svd.hweb] Diagonalization of the bidiagonal form for k=n step -1 until 1 do

```
⟨ Make Bidiagonal Matrix 14.32 ⟩ ≡
    do k = n, 1, -1
      its = 0
      ⟨ test for splitting 14.33 ⟩
      ⟨ test for convergence 14.36 ⟩
    end do
```

This code is used in section 14.

[svd.hweb] Test for splitting. for $el = k$ step -1 until 1 do $rv1\,(1)$ is always zero, so there is no exit through the bottom of the loop

```
⟨ test for splitting 14.33 ⟩ ≡
100: el = k
     test_split = T
     do while(test_split)
        test_split = abs(rv1(el)) + anorm ≠ anorm
        test_split = test_split ∧ abs(w(el − 1)) + anorm ≠ anorm
        el = el − 1
        test_split = test_split ∧ el ≥ 1
     end do
     el = el + 1
     if (abs(rv1(el)) + anorm ≠ anorm) then
        ⟨ cancel rv1 14.34 ⟩
     end if
```

This code is used in section 14.32.

[svd.hweb] Cancellation of $rv1\,(el)$ **if** $el > 1$

```
⟨ cancel rv1 14.34 ⟩ ≡
     c = zero
     s = one
     i = el
     test_conv = i ≤ k
     do while(test_conv)
        f = s ∗ rv1(i)
        rv1(i) = c ∗ rv1(i)
        if (abs(f) + anorm ≠ anorm) then
           g = w(i)
           h = sqrt(f ∗ f + g ∗ g)
           w(i) = h
           c = g / h
           s = −f / h
           if (matu) then
              ⟨ set matrix U bidiagonal columns 14.35 ⟩
           end if
        end if
        i = i + 1
        test_conv = i ≤ k ∧ (abs(f) + anorm ≠ anorm)
     end do
```

This code is used in section 14.33.

[svd.hweb]

⟨ set matrix U bidiagonal columns 14.35 ⟩ ≡
```
do j = 1, m
  y = U(j, el − 1)
  z = U(j, i)
  U(j, el − 1) = y ∗ c + z ∗ s
  U(j, i) = −y ∗ s + z ∗ c
end do
⟨ test for convergence 14.36 ⟩
```

This code is used in section 14.34.

[svd.hweb] test for convergence

⟨ test for convergence 14.36 ⟩ ≡
```
z = w(k)
if (el ≡ k) then
  ⟨ make eigenvalue positive 14.37 ⟩
else
  if (its ≡ 30) then
    ierr = k
    return
  end if
  ⟨ calculate shift for 2 by 2 minor 14.39 ⟩
  ⟨ do QR transform 14.40 ⟩
end if
```

This code is used in sections 14.32 and 14.35.

[svd.hweb] $w(k)$ is made non-negative

⟨ make eigenvalue positive 14.37 ⟩ ≡
```
if (z < zero) then
  w(k) = −z
  if (matv) then
    ⟨ change sign of matrix V columns 14.38 ⟩
  end if
end if
```

This code is used in section 14.36.

[svd.hweb]

⟨ change sign of matrix V columns 14.38 ⟩ ≡
```
do j = 1, n
  V(j, k) = −V(j, k)
end do
```

This code is used in section 14.37.

[svd.hweb] Shift from bottom 2 by 2 minor.

⟨ calculate shift for 2 by 2 minor 14.39 ⟩ ≡

```
its = its + 1
x = w(el)
y = w(k − 1)
g = rv1(k − 1)
h = rv1(k)
f = ((y − z) ∗ (y + z) + (g − h) ∗ (g + h)) / (two ∗ h ∗ y)
g = sqrt(f ∗ f + one)
f = ((x − z) ∗ (x + z) + h ∗ (y / (f + sign(g, f)) − h)) / x
```

This code is used in section 14.36.

[svd.hweb] Next QR transformation

⟨ do QR transform 14.40 ⟩ ≡

```
c = one
s = one
do i = el, k − 1
  ⟨ calculate QR transform coefficients 14.41 ⟩
  if (matv) then
    ⟨ set eigenvector matrix V values 14.42 ⟩
  end if
  ⟨ calculate rotation values 14.43 ⟩
  if (matu) then
    ⟨ set eigenvector matrix U values 14.44 ⟩
  end if
end do
rv1(el) = zero
rv1(k) = f
w(k) = x
goto 100
```

This code is used in section 14.36.

[svd.hweb]

⟨ calculate QR transform coefficients 14.41 ⟩ ≡

```
g = rv1(i + 1)
y = w(i + 1)
h = s ∗ g
g = c ∗ g
z = sqrt(f ∗ f + h ∗ h)
rv1(i) = z
c = f / z
s = h / z
f = x ∗ c + g ∗ s
g = −x ∗ s + g ∗ c
h = y ∗ s
y = y ∗ c
```

This code is used in section 14.40.

[svd.hweb]

⟨ set eigenvector matrix V values 14.42 ⟩ ≡

```
do j = 1, n
  x = V(j, i)
  z = V(j, i + 1)
  V(j, i) = x ∗ c + z ∗ s
  V(j, i + 1) = −x ∗ s + z ∗ c
end do
```

This code is used in section 14.40.

[svd.hweb] Rotations can be arbitrary **if** $z \equiv zero$.

⟨ calculate rotation values 14.43 ⟩ ≡

```
z = sqrt(f ∗ f + h ∗ h)
w(i) = z
if (z ≠ zero) then
  c = f / z
  s = h / z
end if
f = c ∗ g + s ∗ y
x = −s ∗ g + c ∗ y
```

This code is used in section 14.40.

[svd.hweb]

⟨ set eigenvector matrix U values 14.44 ⟩ ≡

```
do j = 1, m
  y = U(j, i)
  z = U(j, i + 1)
  U(j, i) = y * c + z * s
  U(j, i + 1) = −y * s + z * c
end do
```

This code is used in section 14.40.

15 Modules that change: the ODEs

The modules that may need changing for different sets of differential equations or because of the number of of boundary conditions. The **parameter**s are set in module 14. This module is included in the source for the linear equation solver. The differential equations are codes from here to the end.

If someone is using this in production environments, it is recommended that this major section be an include file. The FWEB allows this, but we have included this code directly to ease the distribution of the sources.

The differential equation that is being solved was presented in modules 2 and 8.

For linear problems there is one set of equations and the **else** clause of the **if** −**then** − **else** can be eliminated. Common calculations for forcing functions or variable coefficients are placed in the first part of the conditional. For nonlinear ODEs, the **if** −**then** clause also contains the base equations and the **else** clause contains the linearized equations. These linearized equations can be put onto separate processors. Each processor would process the recurrence equations in a loop on k. Further, these are nested in the loop through the particular solutions, a loop on i, and each $i > 0$ is what can be assigned to separate processors.

⟨ Calculate the k-th term of the power series 15 ⟩ ≡

```
if (i ≡ 0 | scoring > 0) then
  ⟨ Do common calculations 15.1 ⟩
end if
PS(k, 1, i) = PS(k − 1, 2, i) / (k)
PS(k, 2, i) = (−xi * PS(k − 1, 1, i) − mu * PS(k − 1, 2, i) + sn(k − 1)) / (k)
```

This code is used in section 5.3.

These calculations are independent of which solution of the ODE it is. These are usually the forcing functions or variable coefficients that need not be calculated more than once.

⟨ Do common calculations 15.1 ⟩ ≡

```
call ps_trig(t_ps, k − 1, sn, cs)
```

This code is used in section 15.

We need to declare the variable for μ and ξ. The series sn and cs represent the independent variable t and the sine and cosine of t, respectively.

⟨ Quasi's local variables 4.1.7 ⟩ +≡
floating mu, xi, $sn(0 : m\_)$, $cs(0 : m\_)$
parameter $(mu = 0.05,\ xi = 1.0)$

Sometimes we declare additional variables in coding the differential equations. These often need to be initialized.

⟨ Initialization of variables 4.1.10 ⟩ +≡

These variables may be dependent upon the center of expansion.

⟨ Set up for current center of expansion 5.2 ⟩ +≡

16 INDEX

⟨ Apply column transform 14.9 ⟩ Used in section 14.5.
⟨ Apply row transformation 14.15 ⟩ Used in section 14.11.
⟨ Are we at the limit of accuracy of the power series 5.4.1 ⟩ Used in section 5.4.
⟨ Assign *i_bc_row* based on constraint *vs.* observation 5.5.6 ⟩ Used in section 5.5.5.
⟨ Calculate column scale factor 14.6 ⟩ Used in section 14.5.
⟨ Calculate row scale factor 14.12 ⟩ Used in section 14.11.
⟨ Calculate the line step 5.6.3 ⟩ Used in section 5.6.2.
⟨ Calculate the new initial value estimates 5.6.1 ⟩ Used in section 5.6.
⟨ Calculate the norm of the change in the initial values 5.6.6 ⟩ Used in section 5.6.5.
⟨ Calculate the power series coefficients, recurrence 5.3 ⟩ Used in section 4.1.1.
⟨ Calculate the k-th term of the power series 15 ⟩ Used in section 5.3.
⟨ Change initial values because we are at a shooting point 5.5.4 ⟩ Used in section 5.5.3.
⟨ Check for an odd number of terms and update *ps_sqr* 9.0.18 ⟩ Used in section 9.0.15.
⟨ Check for convergence of the initial values 5.6.7 ⟩ Used in section 5.6.1.
⟨ Check for non-fatal and fatal sizes of A 12.1.6 ⟩ Used in section 12.1.5.
⟨ Check parameters and initialize variables for *gj_svd* 12.1.5, 12.1.7, 12.1.8 ⟩ Used in section 12.1.1.
⟨ Check to see if this is last boundary condition 5.1.2 ⟩ Used in sections 5.1 and 5.5.5.
⟨ Check to see if we have integrated far enough 5.4.5 ⟩ Used in section 5.4.
⟨ Compute *ps_div* result 9.0.6 ⟩ Used in section 9.0.4.
⟨ Compute *ps_mult* result 9.0.3 ⟩ Used in section 9.0.1.
⟨ Compute *ps_sqr* sum 9.0.17 ⟩ Used in section 9.0.15.
⟨ Copy the Array 14.1 ⟩ Used in section 14.
⟨ Copy the inverse of the normal matrix into A 13.0.4 ⟩ Used in section 13.
⟨ Copy the least squares part of A to *A_4* 13.0.1 ⟩ Used in section 13.
⟨ Create Left Singular Vectors 14.24 ⟩ Used in section 14.
⟨ Create Right Singular Vector 14.19 ⟩ Used in section 14.
⟨ Debug output? All particular solutions 5.8.3 ⟩ Used in section 5.5.2.
⟨ Debug output? At the end 5.8.2 ⟩ Used in section 5.7.
⟨ Debug output? Initial values 5.8 ⟩ Used in section 5.1.
⟨ Debug output? Power series coefficients 5.8.1 ⟩ Used in section 5.3.
⟨ Debug output? Reducing k 13.1.1 ⟩ Used in section 12.1.1.
⟨ Debug output? Saving 5.8.4 ⟩ Used in section 5.5.5.
⟨ Declare variables for *gj_svd* 12.1.2, 12.1.3, 12.1.4 ⟩ Used in section 12.1.1.
⟨ Declare variables for *ps_pwr* 9.0.20 ⟩ Used in section 9.0.19.
⟨ Declare *ps_div* variables 9.0.5 ⟩ Used in section 9.0.4.
⟨ Declare *ps_exp* variables 9.0.8 ⟩ Used in section 9.0.7.
⟨ Declare *ps_ln* variables 9.0.12 ⟩ Used in section 9.0.11.
⟨ Declare *ps_mult* variables 9.0.2 ⟩ Used in section 9.0.1.
⟨ Declare *ps_sqrt* variables 9.0.24 ⟩ Used in section 9.0.23.
⟨ Declare *ps_sqr* variables 9.0.16 ⟩ Used in section 9.0.15.
⟨ Declare *ps_trig* variables 9.1.2 ⟩ Used in section 9.1.1.
⟨ Determine τ and advance t 5.4 ⟩ Used in section 4.1.1.
⟨ Determine the limit of numeric convergence 5.2.2, 5.2.3 ⟩ Used in section 4.1.1.
⟨ Divide Row by Max element 12.1.11 ⟩ Used in section 12.1.1.
⟨ Do Householder Reduction 14.4 ⟩ Used in section 14.
⟨ Do common calculations 15.1 ⟩ Used in section 15.
⟨ Do the general case for *ps_trig* 9.1.4 ⟩ Used in section 9.1.1.
⟨ Do the trivial case for *ps_trig* 9.1.3 ⟩ Used in section 9.1.1.
⟨ End game processing 5.7 ⟩ Used in section 5.5.
⟨ End of forward integration 5.6 ⟩ Used in section 4.1.
⟨ Estimate the new initial values, to be revised 5.6.5 ⟩ Used in section 5.6.1.
⟨ Evaluate the power series 5.4.6 ⟩ Used in section 4.1.2.

⟨ Find max pivot element 12.1.9 ⟩ Used in section 12.1.1.
⟨ Finish it up by back substitution 13.0.5 ⟩ Used in section 13.
⟨ Initialization of variables 4.1.10, 4.1.12, 4.1.16, 15.3 ⟩ Used in section 4.
⟨ Input alphanumeric information 5.0.2 ⟩ Used in section 5.0.1.
⟨ Input boundary conditions 5.0.7, 5.0.8 ⟩ Used in section 5.0.1.
⟨ Input formats 5.0.11 ⟩ Used in section 5.0.1.
⟨ Input initial value estimates 5.0.10 ⟩ Used in section 5.0.1.
⟨ Input parameters, estimates, and boundary conditions 5.0.1 ⟩ Used in section 4.
⟨ Input **integer** parameters 5.0.3 ⟩ Used in section 5.0.1.
⟨ Input **real** parameters 5.0.6 ⟩ Used in section 5.0.1.
⟨ Is this the point of a boundary condition 5.4.4 ⟩ Used in section 5.4.
⟨ Is this where an external event is 5.4.3 ⟩ Used in section 5.4.
⟨ Iterate on the boundary value problem 4.1 ⟩ Used in section 4.
⟨ Make Bidiagonal Matrix 14.32 ⟩ Used in section 14.
⟨ Output the coefficient matrix 13.1 ⟩ Used in section 13.1.1.
⟨ Output the results, regularly 5.5.2 ⟩ Used in section 5.5.
⟨ Output the whole mess 13.0.6 ⟩ Used in section 12.1.1.
⟨ Perform any needed work for external reasons 5.5.3 ⟩ Used in section 5.5.
⟨ Perform appropriate multiplications by U and V 13.0.2, 13.0.3 ⟩ Used in section 13.
⟨ Perform in place reduction 12.1.12 ⟩ Used in section 12.1.1.
⟨ Perform the forward integration 4.1.1 ⟩ Used in section 4.1.
⟨ Perform the general case for *ps_exp* 9.0.10 ⟩ Used in section 9.0.7.
⟨ Perform the general case for *ps_ln* 9.0.14 ⟩ Used in section 9.0.11.
⟨ Perform the general case for *ps_pwr* 9.0.22 ⟩ Used in section 9.0.19.
⟨ Perform the general case for *ps_sqrt* 9.0.26 ⟩ Used in section 9.0.23.
⟨ Perform the line search part of scoring 5.6.2 ⟩ Used in section 5.6.
⟨ Perform the statistical analysis 7 ⟩ Used in section 5.6.1.
⟨ Perform the trivial case for *ps_exp* 9.0.9 ⟩ Used in section 9.0.7.
⟨ Perform the trivial case for *ps_ln* 9.0.13 ⟩ Used in section 9.0.11.
⟨ Perform the trivial case for *ps_pwr* 9.0.21 ⟩ Used in section 9.0.19.
⟨ Perform the *zero* case for *ps_sqrt* 9.0.25 ⟩ Used in section 9.0.23.
⟨ Permute the solution matrix into original form 12.1.13 ⟩ Used in section 12.1.1.
⟨ Perturb the initial value to ensure independence? 5.1.4 ⟩ Used in section 5.1.
⟨ Perturb the $jj_{\rm th}$ element of this column 5.1.6 ⟩ Used in section 5.1.4.
⟨ Prepare column transformation values 14.8 ⟩ Used in section 14.5.
⟨ Prepare row transformation values 14.14 ⟩ Used in section 14.11.
⟨ Quasi's input variables 4.1.4, 4.1.5, 4.1.6, 4.1.8, 4.1.13, 4.1.15, 5.0.4, 5.0.5, 5.2.4, 5.4.7 ⟩ Used in section 4.
⟨ Quasi's local variables 4.1.7, 4.1.9, 4.1.11, 4.1.14, 4.1.17, 5.0.9, 5.1.1, 15.2 ⟩ Used in section 4.
⟨ Quasi's **parameter**s 4.1.3 ⟩ Used in section 4.
⟨ Reached the limit of this expansion 5.5.1 ⟩ Used in section 5.5.
⟨ Reduce the order of the problem, if possible 5.0.12 ⟩ Used in section 5.0.1.
⟨ Reset Zeros 14.18 ⟩ Used in sections 14.5 and 14.11.
⟨ Restore column values 14.10 ⟩ Used in section 14.5.
⟨ Restore row values 14.16 ⟩ Used in section 14.11.
⟨ Save the row of the coefficient matrix 5.5.5 ⟩ Used in section 5.5.
⟨ Scale column 14.7 ⟩ Used in section 14.5.
⟨ Scale row 14.13 ⟩ Used in section 14.11.
⟨ Set Initial Values 14.3 ⟩ Used in section 14.
⟨ Set a row of U to *zero* 14.26 ⟩ Used in section 14.24.
⟨ Set anorm 14.17 ⟩ Used in section 14.11.
⟨ Set initial values for scoring integrations 5.1.5 ⟩ Used in section 5.1.
⟨ Set up for current center of expansion 5.2, 5.2.1, 15.4 ⟩ Used in section 4.1.1.

⟨ Set up for this iteration 5.1 ⟩ Used in section 4.1.
⟨ Should we output something 5.4.2 ⟩ Used in section 5.4.
⟨ Solve for the superposition coefficients 5.6.4 ⟩ Used in section 5.6.1.
⟨ Special boundary condition operators 5.5.9 ⟩ Used in section 5.5.5.
⟨ Store the element based upon the derivative 5.5.8 ⟩ Used in section 5.5.5.
⟨ Store the element based upon the solution vector 5.5.7 ⟩ Used in section 5.5.5.
⟨ Store the superposition identity in C 5.1.3 ⟩ Used in section 5.1.
⟨ Subroutine *svd* local variables 14.2 ⟩ Used in section 14.
⟨ Swap column and row pointers 12.1.10 ⟩ Used in section 12.1.1.
⟨ Swap rows in the augmented matrix 12.1.14 ⟩ Used in section 12.1.13.
⟨ The parallel forward integration parts 4.1.2 ⟩ Used in section 4.1.1.
⟨ This is the first observation, **call** *svd* for the rest 13 ⟩ Used in section 12.1.1.
⟨ Transform by Column 14.5 ⟩ Used in section 14.4.
⟨ Transform by Row 14.11 ⟩ Used in section 14.4.
⟨ Update the row pointers 12.1.15 ⟩ Used in section 12.1.13.
⟨ Use the result of the evaluation 5.5 ⟩ Used in section 4.1.2.
⟨ calculate QR transform coefficients 14.41 ⟩ Used in section 14.40.
⟨ calculate rotation values 14.43 ⟩ Used in section 14.40.
⟨ calculate shift for 2 by 2 minor 14.39 ⟩ Used in section 14.36.
⟨ cancel rv1 14.34 ⟩ Used in section 14.33.
⟨ change sign of matrix V columns 14.38 ⟩ Used in section 14.37.
⟨ do QR transform 14.40 ⟩ Used in section 14.36.
⟨ final diagonal value 14.31 ⟩ Used in section 14.24.
⟨ left double division 14.29 ⟩ Used in section 14.28.
⟨ make eigenvalue positive 14.37 ⟩ Used in section 14.36.
⟨ right double division 14.20 ⟩ Used in section 14.19.
⟨ scale matrix U columns 14.30 ⟩ Used in section 14.24.
⟨ set eigenvector matrix U values 14.44 ⟩ Used in section 14.40.
⟨ set eigenvector matrix V values 14.42 ⟩ Used in section 14.40.
⟨ set left loop initial values 14.25 ⟩ Used in section 14.24.
⟨ set matrix U bidiagonal columns 14.35 ⟩ Used in section 14.34.
⟨ set matrix U values 14.28 ⟩ Used in section 14.24.
⟨ set matrix V diagonal values 14.23 ⟩ Used in section 14.19.
⟨ set matrix V values 14.21 ⟩ Used in section 14.19.
⟨ test for convergence 14.36 ⟩ Used in sections 14.32 and 14.35.
⟨ test for splitting 14.33 ⟩ Used in section 14.32.
⟨ zero matrix U column values 14.27 ⟩ Used in section 14.24.
⟨ zero V off-diagonal elements 14.22 ⟩ Used in section 14.19.

COMMAND LINE: `"fweave psq"`.
WEB FILE: `"psq.web"`.
CHANGE FILE: `(none)`.
GLOBAL LANGUAGE: FORTRAN.