Implementation of the parallel code for solving the ODE system - a prospective roadmap

Muthu Ramalingam

1 Introduction

The crux of the problem was to solve a system of non-linear, second order ordinary differential equations [8] with multi-point boundary conditions dependent on a set of parameters containing a subset of known parameters and unknown parameters. The boundary conditions are chosen in such a way that they minimize a cost objective dictated by the parameters. This is quite a complicated problem but can be solved with MATLAB as the main tool, given the range of functionalities that MATLAB provides including function pointers and parallel constructs. However, MATLAB has its own limitations in terms of speed and problem size. So the clients wanted the code to be transcribed to another mathematically-oriented modern and free language aka Julia.

Julia so far has not seen a development[6] in the domain of Multipoint Boundary Value Problems(MPBVP) [5], except for an attempt or two by enthusiasts. After involved research, it was found out the only way was to put to use the COLNEW library based on F77. However, due to the timeline being extended indefinitely, since the signing of the project, the client wishes to terminate the project and expects a documentation clearly laying out the course of the process.

This write-up describes how the problem could be solved after explaining in detail what has been done so far towards finding ways to solve it. It is presented in a more informal and non-mathematical manner since there was not much time to be precise on the notational details and generic logical styling. So it is advisable to keep the original document provided by the clients as a side-by while going through this one.

2 The present algorithm

The notation used corresponds to the MATLAB program variables [4]

1. Define (known) parameters vector `k` common to all units.
2. Initialize (unknown) parameters vector `ksi` common to all units.
3. Define (known) parameters vector `k_g` specific to each unit.

4. Initialize (unknown) parameters vector `Kappa_g` specific to each unit.

5. Initialize mesh. `x0`

6. Solve for optimal boundary solutions `b` following steps 7 and 8.

7. Define the parameter range `[params1, params2]` which is a function of the parameters collective vector
   `k, ksi, k_g, Kappa_g`

8. Solve the ODE in the parameter range defined by the step 7 to optimize the cost function which depends on `paramsH.gamma` and `paramsL.gamma`

9. Define BC's and solve ODE system with updated parameter values

MATLAB has unique solvers such `bvp4c` and `bvp5c` to solve such a complicated set of ODE's albeit much slower than other compiled languages like C and FORTRAN.

3 Julia resources available

Julia has no known libraries catering to the solutions of MPBVP so far. Problems involving until three-point boundary values and Strome-Liouville type problems can be solved with the Shooting methods.Beyond that there is no prospect and the community has been actively involved in developing a library for this. A few instances whereby the developers advise to use the Chebyshev Polynomial[1] have been found. I used the Chebyshev methods for ODEs involving a single variable but when it comes to multiple variables, the method does not help on Julia. I discovered a promising library called ODEInterface[7], which accesses many NETLIB FORTRAN subroutines with a Julia API and tested it for several problems. However I found that it could not be applied to our problem, unless modifications are made to the COLNEW/COLSYS[2] library itself.So, I decided to explore the COLNEW library

4 COLNEW

COLNEW developed by U. Ascher [3] follows one of the collocation methods with Gaussian quadrature. Details of the method can be found widely in literature but instances and examples of how to use COLNEW are quite rare due to its oldness and prevalent usage of commercial solvers such as MATLAB and Mathematica for MPBVP problems by mathematicians. COLNEW is essentially a 4000+ lines code written in FORTRAN77 and available as a single file from the website[2].

The subroutine Fig. 1 takes as its main arguments

- the RHS of the ODE system (Fig. 7)

```
      SUBROUTINE COLSYS (NCOMP, M, ALEFT, ARIGHT, ZETA, IPAR, LTOL,
     1                   TOL, FIXPNT, ISPACE, FSPACE, IFLAG,
     2                   FSUB, DFSUB, GSUB, DGSUB, GUESS)

*       variables

*       NCOMP = d  - number of differential equations   (≤ 20)

*       M(j) - order of the j^th differential equation

*       ALEFT = a ,   ARIGHT = b  - interval ends.

*       ZETA(j) = ζ_j, 1 ≤ j ≤ sum_{l=1}^{NCOMP} M(l). Must be a mesh point in all meshes used,
                    see description of IPAR(11) and FIXPNT below.

*       IPAR - an integer array dimensioned at least 11. A list of the parameters in
                    IPAR and their meaning follows:
```

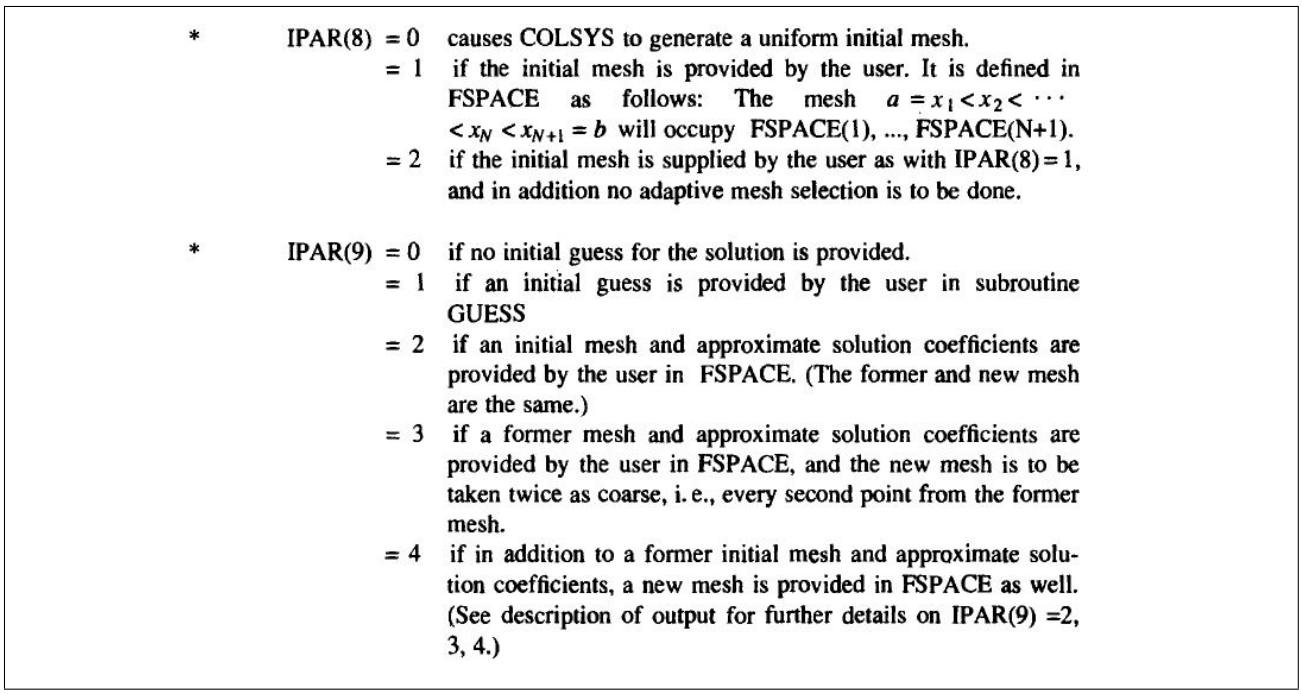
Figure 1: COLNEW subroutine definition

```
*       IPAR(1) = 0 if the problem is linear
                = 1 if the problem is nonlinear

*       IPAR(2) = number of collocation points per subinterval  (=k).

*       IPAR(3) = number of subintervals in the initial mesh (=N).

*       IPAR(4) = number of solution and derivative tolerances, 0 < IPAR(4) ≤ m* .

*       IPAR(5) = dimension of FSPACE.

*       IPAR(6) = dimension of ISPACE.

*       IPAR(7) - output control
                = -1 for full diagnostic printout
                =  0 for selected printout
                =  1 for no printout
```

Figure 2: Properties vector defined

- linearity/non-linearity (Fig. 2)
- various user-supplied iterative control measures (Fig. 4)
- the degree and order of the system (Fig. 3)
- mesh intervals for collocation Fig. (6)

- number of divisions of the collocation grid.
- the RHS of the ODE system Fig.(7)
- the Jacobian matrix of the system (Fig. 7)
- the boundary condtions vector (Fig. 8)
- Jacobian matrix of the boundary conditions vector
- a guess vector (Fig. 9)

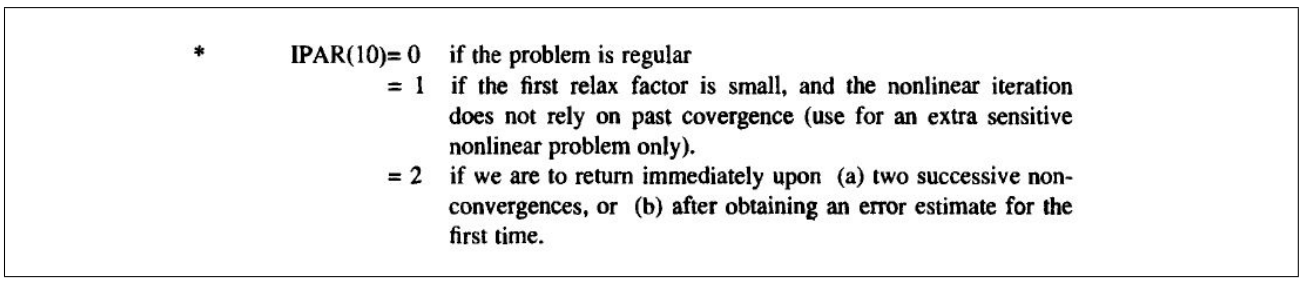


* IPAR(8) = 0 causes COLSYS to generate a uniform initial mesh.
= 1 if the initial mesh is provided by the user. It is defined in FSPACE as follows: The mesh $a = x_1 < x_2 < \cdots < x_N < x_{N+1} = b$ will occupy FSPACE(1), ..., FSPACE(N+1).
= 2 if the initial mesh is supplied by the user as with IPAR(8) = 1, and in addition no adaptive mesh selection is to be done.

* IPAR(9) = 0 if no initial guess for the solution is provided.
= 1 if an initial guess is provided by the user in subroutine GUESS
= 2 if an initial mesh and approximate solution coefficients are provided by the user in FSPACE. (The former and new mesh are the same.)
= 3 if a former mesh and approximate solution coefficients are provided by the user in FSPACE, and the new mesh is to be taken twice as coarse, i. e., every second point from the former mesh.
= 4 if in addition to a former initial mesh and approximate solution coefficients, a new mesh is provided in FSPACE as well. (See description of output for further details on IPAR(9) =2, 3, 4.)

Figure 3: Properties vector defined

The layman point of view is that in the interval of each mesh element, the function is a polynomial which coincides with the neighbouring polynomial governed by its boundary conditions that maintaining continuity across the domain.



* IPAR(10)= 0 if the problem is regular
= 1 if the first relax factor is small, and the nonlinear iteration does not rely on past covergence (use for an extra sensitive nonlinear problem only).
= 2 if we are to return immediately upon (a) two successive non-convergences, or (b) after obtaining an error estimate for the first time.

Figure 4: Iterant control

```
*       IPAR(11)= number of fixed points in the mesh other than ALEFT and
                  ARIGHT.

*       LTOL    - an INTEGER array of dimension IPAR(4). LTOL(j) = l specifies
                  that the j^th tolerance in TOL controls the error in the l^th
                  component of z(u).

*       TOL     - a REAL array of dimension IPAR(4). TOL(j) is the error tolerance
                  on the LTOL(j)^th component of z(u). Thus, the code attempts
                  to satisfy for j = 1 , ..., IPAR(4) on each subinterval
                  |(z(v)-z(u))_LTOL(j)| ≤ TOL(j) (|z(u)_LTOL(j)| + 1) if v(x) is the
                  approximate solution vector.

*       FIXPNT  - an array of dimension IPAR(11). It contains the points, other than
                  ALEFT and ARIGHT, which are to be included in every
                  mesh.
```

Figure 5: Tolerance control

The iterative process in the computation is an attempt to make sure that the continuity falls within the defined tolerance limits.

```
*       FSPACE - a REAL work array of dimension IPAR(5). Its size provides a
                 constraint on the maximum mesh size.

*       IFLAG  - the mode of return from COLSYS.
                 = 1   for normal return
                 = 0   if the collocation matrix is singular.
                 =-1   if the expected number of subintervals exceeds storage
                       specifications.
                 =-2   if the nonlinear iteration has not converged.
                 =-3   if there is an input data error.
```

Figure 6: Collocation grid-properties

Apart from the numerical techniques pertaining to any collocation method, the author has followed several nuances explained in detail in his textbook [], to generalize the code and a problem framed for the code, as much as possible. It is quite demanding to refactor the code and debug it.

5 How to frame a problem for COLNEW - some cases

5.1 Problem-1

Let us consider a linear MPBVP (Fig. 10). It is a fourth-order system of two variables u_1, u_2 and a defined parameter ϵ. x lies in $[0, 1]$ and the various boundary conditions are shown. We choose the inhomogeneities on the RHS vector viz., $q_1(x)$ and $q_2(x)$ such that we arrive at the solutions:

$$u_1(x) = e^{-\pi x} + cos\pi x$$

* DFSUB – name of subroutine for evaluating the Jacobian of F at a point X. it should have the heading

SUBROUTINE DFSUB (X , Z , DF)

where Z = z(u(x)) is defined as for FSUB and the $d \times m^*$ array DF should be filled by the partial derivatives of F, namely, for a particular call one calculates

DF(i,j) = $\frac{\partial f_i}{\partial z_j}$, $i=1, \ldots, d, j = 1, \ldots, m^*$

Figure 7: RHS Vector

* GSUB – name of subroutine for evaluating the j^{th} side condition g_j at a point x = ZETA(j) ,$1 \le j \le m^*$. It should have the heading

SUBROUTINE GSUB (J , Z , G)

where Z is as for FSUB, and j and the scalar G are as above.

* DGSUB – name of subroutine for evaluating the j^{th} row of the Jacobian of g_j. It should have the heading

SUBROUTINE DGSUB (J , Z , DG)

where Z is as for FSUB, J as for GSUB and the m^*-vector DG gives DG(l) = $\frac{\partial g_j}{\partial z_l}$, $j = 1, \ldots, m^*$.

Figure 8: Boundary conditions vector and its Jacobian matrix

* GUESS – name of subroutine to evaluate the initial approximation for Z = z(u(x)) and for DMVAL = vector of m_j^{th} derivatives of u(x). It should have the heading

SUBROUTINE GUESS (X , Z , DMVAL)

Note that this subroutine is needed only for nonlinear problems if using IPAR(9) = 1.

Figure 9: Guess solution vector

$$u_2(x) = sin\pi x$$

We substitute these above values of $q_1(x)$ and $q_2(x)$ in the original equations. We know

$$u_1^{'} = -\frac{1}{\epsilon}e^{-x/\epsilon} - \pi sin\pi x$$

Consider the linear BVP

$$\varepsilon u_1' = -u_1 + u_2 + q_1(x)$$

$$0 < x < 1$$

$$u_2^{(iv)} = u_1 + u_2 + q_2(x)$$

$$u_1(0) = 2, \qquad u_2(0) = u_2(1) = u_2''(0) = u_2''(1) = 0$$

where we choose the inhomogeneities $q_1(x)$ and $q_2(x)$ such that the solution is

$$u_1(x) = e^{-x/\varepsilon} + \cos \pi x, \qquad u_2(x) = \sin \pi x$$

The parameter ε is small and positive; we take $\varepsilon = 10^{-4}$.

Figure 10: Problem-1

$$u_2^{iv} = \pi^4 sin\pi x$$

which we can write in expressions for the RHS vectors viz., `f(1)` and `f(2)` (Fig. 11)

In (Ref. fig) we have the code for the Jacobian of the RHS vectors. `df(1,1)` and `df(1,2)` are derivatives of `f(1)` with respect to u_1. Similarly `df(2,1)` and `df(2,2)` are derivatives of `f(1)` with respect to u_1. (Fig. 12)

We follow the same principles to the boundary conditions vector subroutine `gsub` (Fig. 13). Since we don't have any intra-domain or dependent boundary conditions, the framing of the BC's is quite easy.The `z` vector passed contains u_1, u_2, u_2', u_2'', u_2''' and u_2^{iv} respectively.Shown in Fig. 13, is the first boundary condition $u_1(0) = 2$, by simplifying to a vector $u_1(0) - 2 = 0$, as `g = z(1)-2.0D+0`, and the BC's $u_2(0) =$

The Jacobian matrix of the BC's defined by $u_1(0)$, $u_2(0)$, $u_2(1)$, $u_2''(0)$ and $u_2''(1)$ are given by

$$\begin{pmatrix} \frac{\partial u_1(0)}{\partial u_1(0)} & \frac{\partial u_1(0)}{\partial u_2(0)} & \frac{\partial u_1(0)}{\partial u_2(1)} & \frac{\partial u_1(0)}{\partial u_2''(0)} & \frac{\partial u_1(0)}{\partial u_2''(1)} \\ \frac{\partial u_2(0)}{\partial u_1(0)} & \frac{\partial u_2(0)}{\partial u_2(0)} & \frac{\partial u_2(0)}{\partial u_2(1)} & \frac{\partial u_2(0)}{\partial u_2''(0)} & \frac{\partial u_1(0)}{\partial u_2''(1)} \\ \frac{\partial u_2(1)}{\partial u_1(0)} & \frac{\partial u_2(1)}{\partial u_2(0)} & \frac{\partial u_2(1)}{\partial u_2(1)} & \frac{\partial u_2(1)}{\partial u_2''(0)} & \frac{\partial u_1(1)}{\partial u_2''(1)} \\ \frac{\partial u_2''(0)}{\partial u_1(0)} & \frac{\partial u_2''(0)}{\partial u_2(0)} & \frac{\partial u_2''(0)}{\partial u_2(1)} & \frac{\partial u_2''(0)}{\partial u_2''(0)} & \frac{\partial u_2''(0)}{\partial u_2''(1)} \\ \frac{\partial u_2''(1)}{\partial u_1(0)} & \frac{\partial u_2''(1)}{\partial u_2(0)} & \frac{\partial u_2''(1)}{\partial u_2(1)} & \frac{\partial u_2''(1)}{\partial u_2''(0)} & \frac{\partial u_2''(1)}{\partial u_2''(1)} \end{pmatrix} \tag{1}$$

All BC's being constants we reckon that only the diagonal of the matrix is non-zero which is represented as `dg(5)` in the subroroutine (Fig. 14)

5.2 Problem-2

Consider the following pair of second order differential equations, which is a model of the deformation of a shallow sphere cap):

In this problem, the `fsub` is defined by `Z(1)`, `Z(2)`, `Z(3)` and `Z(4)` are respectively assigned ϕ, ϕ', ψ and ψ' respectively. These can be checked from Fig. 15

```
      implicit none

      double precision eps
      double precision f(2)
      double precision pi
      parameter ( pi = 3.141592653589793D+00 )
      double precision pix
      double precision x
      double precision z(5)

      type params
        real::r, delta, v, sigmaP, sigmaA, theta, lambda,l, gamma,phi
        real::ksi, etaA, etaP,chiA, chiP, rho, alpha, mu
      end type
      type (params) :: paL,paH

      eps = 1.0D-04
      pix = pi * x

      f(1) = ( - z(1) + z(2) + cos ( pix )
     &         - ( 1.0D+00 + eps * pi ) * sin ( pix )
     &       ) / eps

      f(2) = z(1) + z(2) + ( pi ** 4 - 1.0D+00 ) * sin ( pix )
     &  - cos ( pix ) - exp ( - x / eps )

      return
      end
```

Figure 11: Problem-1: RHS

```
      implicit none

      double precision df(2,5)
      double precision eps
      integer i
      integer j
      double precision x
      double precision z(5)

      eps = 1.0D-04

      do j = 1, 5
        do i = 1, 2
          df(i,j) = 0.0D+00
        end do
      end do

      df(1,1) = - 1.0D+00 / eps
      df(1,2) = 1.0D+00 / eps
      df(2,1) = 1.0D+00
      df(2,2) = 1.0D+00

      return
      end
```

Figure 12: Problem-1: Jacobian of RHS

```
c
      implicit none

      double precision g
      integer i
      double precision z(5)

      if ( i == 1 ) then
        g = z(1) - 2.0D+00
      else if ( i == 2 ) then
        g = z(2)
      else if ( i == 3 ) then
        g = z(4)
      else if ( i == 4 ) then
        g = z(2)
      else if ( i == 5 ) then
        g = z(4)
      end if

      return
      end
```

Figure 13: Problem-1: Boundary conditions

```
      implicit none

      double precision dg(5)
      integer i
      integer j
      double precision z(5)

      do j = 1, 5
        dg(j) = 0.0D+00
      end do

      if ( i == 1 ) then
        dg(1) = 1.0D+00
      else if ( i == 2 ) then
        dg(2) = 1.0D+00
      else if ( i == 3 ) then
        dg(4) = 1.0D+00
      else if ( i == 4 ) then
        dg(2) = 1.0D+00
      else if ( i == 5 ) then
        dg(4) = 1.0D+00
      end if

      return
      end
```

Figure 14: Problem-1: Jacobian of BC's

$$\frac{\epsilon}{\mu}\left[\phi'' + \frac{1}{x}\phi' - \frac{1}{x^2}\phi\right] + \psi\left[1 + \frac{1}{x}\phi\right] - \phi = -\gamma x\left[1 - \frac{1}{2}x^2\right] \tag{2}$$

$$\mu\left(\psi'' + \frac{1}{x}\psi' - \frac{1}{x^2}\psi\right) - \phi\left[1 - \frac{1}{2x}\phi\right] = 0 \tag{3}$$

subject to the BC's in $x = [0, 1]$

$$\phi = x\psi' - 0.3\psi + 0.7x = 0 \tag{4}$$

at $x = 0, 1$

Re-arranging Eq(2) and (3) we get Eqs.(5) and (6) respectively.

$$\frac{\epsilon}{\mu}\phi'' = \frac{\epsilon}{\mu}\left[-\frac{1}{x}\phi' + \frac{1}{x^2}\phi\right] - \psi\left[1 + \frac{1}{x}\phi\right] + \phi - \gamma x\left[1 - \frac{1}{2}x^2\right] \tag{5}$$

The terms `df(1,1), df(1,2), df(1,3)` and `df(1,4)` are the partial derivatives of the RHS of Eq.5 with respect to ϕ, ϕ', ψ and ψ' respectively.

$$\mu\psi'' = \mu\left[-\frac{1}{x}\psi' + \frac{1}{x^2}\psi\right] - \phi\left[1 - \frac{1}{2x}\phi\right] = 0 \tag{6}$$

Similarly `df(2,1), df(2,2), df(2,3)` and `df(2,4)` are the partial derivatives of the RHS of Eq.5 with respect to ϕ, ϕ', ψ and ψ' respectively (Fig. **??**).

```
C.....................................................................
      SUBROUTINE FSUB (X, Z, F)
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION Z(4), F(2)
      COMMON EPS, DMU, EPS4MU, GAMMA, XT
      F(1) = Z(1)/X/X - Z(2)/X + (Z(1) - Z(3)*(1.D0-Z(1)/X) -
     .       GAMMA*X*(1.D0-X*X/2.)) / EPS4MU
      F(2) = Z(3)/X/X - Z(4)/X + Z(1)*(1.D0-Z(1)/2.D0/X) / DMU
      RETURN
      END
```

Figure 15: Problem-2: fsub

We apply the same rationale to the `gsub` and `dgsub` subroutines as explained for the previous case as seen from Figs. 17 and 18 respectively.

6 How to frame the given problem in COLNEW

The MATLAB code provides the RHS vector as `dfdc = [f(2) a2 f(4) a4]` where
`a2 = Func(f(1), f(2), f(3), Parameters)`

```
C.....................................................................
      SUBROUTINE DFSUB (X, Z, DF)
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION Z(4), DF(2,4)
      COMMON EPS, DMU, EPS4MU, GAMMA, XT
      DF(1,1) = 1.D0/X/X +(1.D0 + Z(3)/X) / EPS4MU
      DF(1,2) = -1.D0/X
      DF(1,3) = -(1.D0-Z(1)/X) / EPS4MU
      DF(1,4) = 0.D0
      DF(2,1) = (1.D0 - Z(1)/X) / DMU
      DF(2,2) = 0.D0
      DF(2,3) = 1.D0/X/X
      DF(2,4) = -1.D0/X
      RETURN
      END
```

Figure 16: Problem-2: dfsub

```
      END
C.....................................................................
      SUBROUTINE GSUB (I, Z, G)
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION Z(4)
      GO TO (1, 2, 1, 3), I
    1 G = Z(1)
      RETURN
    2 G = Z(3)
      RETURN
    3 G = Z(4) - .3D0*Z(3) + .7D0
      RETURN
      END
```

Figure 17: Problem-2: gsub

```
C.....................................................................
      SUBROUTINE DGSUB (I, Z, DG)
      IMPLICIT REAL*8 (A-H,O-Z)
      DIMENSION Z(4), DG(4)
      DO 10 J=1,4
   10    DG(J) = 0. D0
      GO TO (1, 2, 1, 3), I
    1 DG(1) = 1.D0
      RETURN
    2 DG(3) = 1.D0
      RETURN
    3 DG(4) = 1.D0
      DG(3) = -.3D0
      RETURN
      END
```

Figure 18: Problem-2: dgsub

and `a2 = Func(f(1), f(3), f(4), Parameters)`
in the four variables viz., `f(1), f(2), f(3), f(4)`

The subroutines `fsub,dfsub, gsub, dgsub` should be re-written in such a way that they take `parameters` as an argument. This means a lot of refactoring in every instace where theses subroutines are used. It should be noticed that for the five BC's involved in the problem and four dependent

variables, a Jacobian matrix of 4 x 4 x 5 = 80 elements has to be supplied by the user, of course only a few of them being non-zero.

7 Status

By the words of the author[3],the library is quite robust and can handle several kinds of MPBVP problems with ease. But unfortunately, the number of examples and test cases solved is too few to come to such conclusions. Except for a few problems solved by the author in his book the usage of this library is not very prevalent. Probably academic interest in using this library was not widespread during the mid-90's which is when digitizing and mining all scientific resources peaked. I tested the code with a few MPBVPs of varying characteristics. The code yielded meaningful results and was found to be obviously faster than MATLAB.

But the trouble arises, when we need to pass the parameter sets and subsets of our problem into the nearly hard-code written for the user-defined subroutines. I did sustained attempts to modify the code but met with several hardships on the way.

The author-developer has put in years of research into the code, and it is quite difficult to keep track of the operations of the various subroutines. Debugging of F77 is not a simple task as it does not follow modern software paradigms. It needs an exponent of F77 with years of experience to debug such a large code.THe best way to avoid this aadvised by F77 practitioners of early times is to moduling the code so that the variable passing could happen naturally.

Moduling is a provision available in later versions of FORTRAN like FORTRAN 90 and FORTRAN 2008, for which reason I converted the code to FORTRAN 90 with the use of automated parsers and was quite successful. But again I met with an impending issue while linking the library to driver codes. These are quite open-ended to be solved if I spend a few days on the fundamentals of Advanced FORTRAN programming constructs.

7.1 Code

The git repo named *gpu-ode* has two branches viz., *master* and *dev* the former containing the initial framework and the latter containing recent developments. It has a library named *f77split* and *f90split* to split the 4000+ line code of *COLNEW* to the constituent subroutines. It contains a library to convert F77 to F90. It has several other additions to the code tree developed by me to make the application user-friendly. A FORTRAN exponent can easily track through the logic if at all he is familiar with the method of collocation. The various demo problems are in folders such as *prb-1*.

8 Roadmap and parallelisation

With the understanding that Julia has no other option but to call the COLNEW subroutine, there is no way that the GPU parallelization in Julia(which is meant for atomic operations on vectors, arrays

and matrices) is going to effect any acceleration. It is rather advisable to develop the whole problem in FORTRAN and then parallelise with the SIMD methods such as BROADCAST, GATHER and so on.

1. Refactor COLNEW in F77 version to include parameters in the subroutines
2. Test the library for various cases
3. Commit the NLOpt library for Non-linear optimization of solution search involving minimization of cost due to boundary neighboourhood
4. Chain calls of NLOpt to COLNEW
5. Test the callback routines
6. Commit MPI to parallelize the callbacks
7. Test the parallelism for NLOpt callbacks
8. Parallelize the main callback routine on MPI
9. Test the parallelized constructs

References

[1] M. Amano, *http://juliaapproximation.github.io/approxfun.jl/v0.7/usage/equations.html*, 2019.

[2] U. M. Ascher, *Colnew code*, 1977.

[3] U. M. Ascher, R. M. Mathheij, and R. D. Russell, *Numerical Solutions of Boundary value Problems*, SIAM in Applied Mathematics, 1995.

[4] M. M. et. al., *The matlab code.*

[5] M. Inc., *bvp4c*, 2021.

[6] Julia, *Adaptivity in mirk4 for bvps*, 2019.

[7] Luchr, *https://github.com/luchr/odeinterface.jl*, 2017.

[8] M. Medhat, E. Pazaj, and E. Schroth, *Computing the numerical solution to an ordinary differential equation.*