Computer Code Documentation

Larry C. Young

January 2020 edition

Chapter 2 describes fundamental calculations needed to solve problems with collocation and other MWR. To facilitate solution of problems with the described methods, computer source code in several languages is provided to perform these fundamental calculations. The problems discussed in other chapters are implemented in several examples codes. Referring to the symbols defined in Chapter 2, the following fundamental quantities can be calculated:

1. Base points or roots, $\boldsymbol{x}$
2. Quadrature and barycentric weights, $\boldsymbol{W}$ and $\boldsymbol{W^b}$
3. Lagrange interpolating polynomials, $\boldsymbol{\ell}(x)$
4. First derivative operators, $\boldsymbol{A}$ and $\widehat{\boldsymbol{A}}$
5. Laplacian operator, $\boldsymbol{B}$
6. Stiffness matrix, $\boldsymbol{C}$
7. Mass matrix for Galerkin or moments methods, $\boldsymbol{M}$
8. Jacobi and Legendre transforms, $\boldsymbol{Q}$
9. Monomial transform, $\breve{\boldsymbol{Q}}$
10. Orthogonal polynomials and there derivative $P_n^{(\alpha,\beta)}$
11. Polynomial derivative relationships, Eqs. (2.32), (2.35) and (2.38)

The available codes are for use with Python, Matlab/Octave, Fortran 90+, C++ and Excel. These codes have been run under Windows, with MinGW and Cygwin. The GNU compilers gfortran and g++ have been used. The calculations are performed with native code, except Excel uses a dynamic link library. Also, in some languages, large problems can optionally be solved faster and more accurately using a library. Since this project is a work in progress, not all calculations are available in all languages. The following table lists the functionality available for each language, where the symbols refer those used in Chapter 2.

Software Functionality

	Symbol	Fortran	Matlab	Python[2]	C++[2]	Excel[1,2]
Nodes, Weights	$\boldsymbol{x, W, W^b}$	✓	✓	✓	✓	✓
Derivatives	$\boldsymbol{A, \widehat{A}, B}$	✓	✓	✓	✓	✓
Stiffness Matrix	$\boldsymbol{C}$	✓	✓	✓	✓	✓
Mass Matrix	$\boldsymbol{M}$	✓	✓	✓	✓[1]	✓[1]
Polynomial Evaluation	$\boldsymbol{P_n^{(\alpha,\beta)}}$	✓	✓	✓		
Polynomial Derivatives	$\boldsymbol{P_n^{(\alpha,\beta)\prime}}$	✓	✓	✓		
Lagrange Interpolation	$\boldsymbol{\ell}$	✓	✓	✓	✓	✓
Monomial Transform	$\boldsymbol{\breve{Q}}$	✓	✓			
Legendre Transform	$\boldsymbol{Q}$	✓				
Jacobi Transform	$\boldsymbol{a}$	✓				
Derivative relationships		✓	✓[1]	✓[1]		

Notes: [1] limited functionality
[2] Chebyshev not supported

For each programming system, a simple driver is provided to demonstrate the call syntax, and how to compute derivatives, integrals and interpolants. You are strongly encouraged to take some time to examine these results before tackling the example problems. You may wish to delve into the fundamental calculations discussed in Chapter 2. For example, the fundamental calculations can be studied by working with the underlying functions or by modifying the demonstration code.

Tabular results from all the codes are given in tab delimited form, usually implemented with a utility program. I frequently pull the tables into Excel for plotting, testing and debugging or into Tecplot for plotting. I have not attempted to create graphs using Python or Matlab/Octave. The examples are about how to solve differential equations, not how to create graphs. Tutorials on how to plot results can be found from many sources.

The fundamental calculations can also be done directly in Excel using the supplied dynamic link library. However, the library is valid only for 32 bit Windows. Online documentation suggests 64 bit libraries can be created with the GNU compilers, but my attempts to do so have failed and my requests for help remain unanswered (see https://stackoverflow.com).

The descriptions below use the following common terminology:

n - number interior points, total points are n + 1 (symmetric) or n + 2 (nonsymmetric)
$symm$ – symmetry 0/1 no/yes or true/false
$TType$ – type of points 0 Gauss, 1 Lobatto, 2 Chebyshev (2nd kind), 3 Radau-right, Radau-left
$Geometry$ or $geom$ – geometry 0,1,2,3 nonsymmetric, symmetric cartesian, cylindrical, spherical (note: some older codes use different enumeration)
$Shift$ – true for shifted interval [0,1], false for [-1,1]
Xc – collocation points
Wc – quadrature weights
Wb – barycentric weights
$Wbfac$ – normalization factor for barycentric weights
Ac – first derivative matrix
An – first derivative of odd quantity (symmetric problem)
Bc – 2nd derivative or Laplacian matrix
Cc – stiffness matrix representation, Laplacian
Mc – mass matrix
Lc – Lagrange interpolating polynomial
$Func$ or $func$ – represents a function argument

Python Code

Collocation, pseudospectral and other MWR have been implemented in Python. These problems have been tested with Python 3.7.4 running under Windows 10. The fundamental calculations are implemented in the files **jacobi.py** and **occ.py**. In addition to these routines there is a utility program for printing arrays and vectors – **arrayprint.py** and several example codes. These programs are briefly described below

Fundamental Calculations

Most of the functions needed to implement MWR are contained int the **occ.py** file. How to use these functions are briefly described here, while the **testp.py** code demonstrates most of the functionality. The module contains not only functions, but also some useful enums and strings. The strings are useful for labeling the type of points, geometry and symmetry for output. To use the module, you must first import it using $import\ occ$. Next, an occ object must be instantiated, this documentation uses the name oc for the object. After instantiation most quantities are obtained using array valued functions.

$oc = occ.OrthColloc(n, TType=Lobatto, Geometry=nonsymmetric, Shift=True)$ – instantiates an occ object, where n is the number of interior points, $TType$ is the type of points - 1, 2, 4 or 5 for $Gauss, Lobatto, Radau\text{-}right\ and\ Radau\text{-}left$, respectively. Valid geometries are 0 – 3 for nonsymmetric, symmetric planar, cylindrical and spherical cases. For nonsymmetric problems the interval [-1,1] can be specified by setting $Shift=False$.

$Xc = oc.Xpoints()$ –points for the *oc* object

Wc = oc.WeightQ() –quadrature weights

Wb = oc.WeightB() –barycentric weights

Wbfac = oc.WBfac() – scaling factor for barycentric weights

Ac = oc.Deriv1() – differentiation matrix, 1st derivative

An = oc.Deriv1odd() – first derivative of an odd function (symmetric problems)

Bc = oc.Deriv2() – differentiation matrix, 2nd derivative or Laplacian

Cc = oc.Stiffness() – stiffness matrix

Mc = oc.MassMatrix([func,nextra=1]) – mass matrix, for f(x) = 1 if no arguments provided or for the function *func*. When a function is provided, the number of extra quadrature points used in the calculations is specified by *nextra.*

Lc = oc.Lcoeff(x) – Lagrange interpolating polynomials through the points, evaluated at x.

Lc = _Lcoefx(x,xi,wb=None) – Lagrange interpolating polynomials. This is a more general program which interpolates through points *xi* rather than the nodes associated with a collocation type.

The **occ.py** code relies on **jacobi.py** for the most fundamental calculations, such as quadrature nodes and weights. Some of the functions in **jacobi.py** are of interest in their own right. The code can use a shared or dynamic link library created from the Fortran 90 code. The library and code to create it are in the **lib** subdirectory. Use of the library is purely optional, but should be faster and more accurate for large n. Use of the library is controlled by variable $NumLib$. The library is not used for $n < NumLib$. The library is provided for use with Windows 10, but for other systems it can easily be created using the scripts and code in the **lib** directory.

The **polytest.py** code demonstrates some of the functionality of **jacobi.py**. If it is first imported using *import jacobi as jp*, some of the calculations available are:

ar = jp.jac_reccoef(n,abeta=[0,0],Monic=False,Shift=False) – returns the 4 x n array containing the recursion coefficients. If *Monic=True*, rows 0 and 1 contain the recursion coefficients, Eq. (2.12) and row 2 is the leading coefficient $\rho_k^{(\alpha,\beta)}$ in the conventional form, Eq. (2.15). If *Monic=False*, the first 3 rows are the recursion coefficient, Eq. (2.16). Row 3 always contains $\zeta_k^{(\alpha,\beta)}$ of Eq. (2.7). If *Shift=True*, the values are for the shifted polynomials.

p = jp.jac_poly_recurrent(x,n,abeta=[0,0],nd=0,n0=0,Monic=False,Shift=False) – returns an *(n0 + nd + 1) x (x.size)* array containing values for the polynomials and derivatives at x. Polynomials $P_{\text{n-n0}}$ thru P_{n} are contained in the first *n0* + 1 rows, while derivatives are contained in the last *nd* rows.

w,wbfac = jp.jac_quad(n,abeta=[0,0],geom=0) – returns quadrature nodes and weights. w is 4 x (n+2) for nonsymmetric problems (*geom = 0*) or 4 x (n+1) for symmetric problems. The respective rows are Xc, $arccos(Xc)$, Wb and Wc. The barycentric weights are normalized, where *wbfac* is the normalization factor.

Example Calculations

The first two codes use **jacobi.py** and **occ.py** to perform some of the fundamental calculations:

polytest.py – demonstrates some fundamental polynomial calculation in **jacobi.py**.

testp.py – test code showing how to obtain and use the basic data needed to solve problems

ex01.py – first example, catalyst pellet problem with linear source and variable coefficient. The problem is solved twice: (1) by collocation and (2) by the moments or Galerkin method.

ex01s.py – first example problem, catalyst pellet problem, nonlinear symmetric problems.

ex04ff.py – falling film problem from chapter 4, analytical continuous solution in z.

ex04ffs.py – falling film problem from chapter 4, solved using a variety of stepping methods in z.

Utility Functions

A routine for easily creating tabular output is included with the **arrayprint.py** code. The output is tab delimited so the values can easily be imported by a spreadsheet for further analysis or plotting. For this documentation, we assume the module is imported using $import\ arrayprint\ as\ ap$. The following functions are provided.

ap.fopen(fname) – opens a file with the given name.

ap.fclose() – closes the file

file = ap.file() – gives file handle for use with the print function

ap.vectorprint(title, value, nl=20, fmtf=None) – print a one-dimensional array. *title* is a string printed. *nl* specifies a maximum number of values per line. *fmtf* specifies a format string.

ap.arrayprint(title, a0, a1=None, a2=None, a3=None, fmtf=None) - *title* is a string printed. *a0*, *a1, a2* and *a3* are one or two dimensional arrays which are tabulated. *fmtf* specifies a format string.

Matlab/Octave

The Matlab/Octave code is organized with the main codes in one directory and the functions used in subdirectory **mn**. Originally, Matlab/Octave was supported by creation of Matlab executable (*.mex*) files from the C++ code for fundamental calculations. This approach appears not to be portable, since it failed after installing Cygwin and Octave on a new computer. For this reason, all the calculations are now done with native Matlab code. Some improvements to the node and weight calculations have not yet been implemented. However, I believe this native code should be reasonably efficient and much more portable.

Although these codes have not been tested under Matlab, I have tried to make them compatible by avoiding Octave specific extensions. Unfortunately, Octave does not have an option to assist with compatibility. I would appreciate hearing from anyone that tries them on Matlab.

Fundamental Calculations

Several functions are provided to perform the fundamental calculations. A simple driver is provided to demonstrate the call syntax and the results in tab delimited tables. I frequently pull these tables in Excel for

OCCdefs.m – contains named constants for ptyp, geom and sym and string designations useful for program output

[xc,wc,Ac] = OCnonsym(n,typ) – nonsymmetric points, weights and derivative operator given n and typ 1 - 5 (Gauss, Lobatto, Chebyshev, Radau right/left)

[xc,wc,Ac,An] = OCsym(n,typ,geom) – symmetric base points, weights and derivative operators (even and odd) given n, typ (1 Gauss, 2 Lobatto, 3 Chebyshev) and $geom$ = 0,1,2 for planar, cylindrical and spherical geometry

[Bc,Cc] = OCBCcoef(wc,Ac,An,symm) – 2nd derivative Laplacian and stiffness matrix given weights, first derivative matrices and symmetry (0/1 no/yes)

Dc = MassMatrix(xc,[@func,nextra]) – mass matrix, function handle may be supplied as an option, defaults to f(x) = 1.

Lc = Lcoef(x,xc) – returns Lagrange interpolating polynomials evaluated at x (scalar or array), xc are the interpolation points, result is size(x) x size(xc)

Q = Lpoly(xc,symm) – Lagrange interpolating polynomial as monomials $\sum Q_{ij} x^{j-1}$

Example Calculations

testm.m – driver program to demonstrate how to perform fundamental calculations

ex1_non.m – simple code uses Orthogonal Collocation to solve boundary value problem, diffusion with nonlinear source and variable coefficients, Dirichlet boundary conditions. Creates output file *ex1n.dat*.

ex1_nonx.m – extended solution of diffusion problem with linear source and variable coefficients with Dirichlet boundary conditions. The problem is solved 4 times: (1) conventional formulation (nonsymmetric matrix). (2) weak formulation, (3) Galerkin/Moments with interpolated source, (4) full Galerkin/Moments. Galerkin and Moments calculations occur when Lobatto and Gauss points are selected, respectively.

ex2_sym.m – symmetric diffusion with nonlinear source, third kind boundary conditions, solved with Gauss, Lobatto and Chebyshev points. Planar, cylindrical or spherical geometry can be selected.

ex3_ax.m – solves problem for chemical reactor with axial dispersion, Ch. 3.2. The three methods of solution described in the text are coded.

ex4_ff.m – solves falling liquid film problem, continuous solution in z via eigenvalues and eigenvectors

ex4_ffs.m – solves falling film problem with numerical stepping methods in z. A choice of 12 different solution procedures are coded.

exheatflux.m – solves the first example conduction problem described in section 1.3.

Fortran 90+

The Fortran code is organized to have the source code in a directory and makefiles, object files, module files and executables in the **o** subdirectory. A **makefile** is included to build the executables and is run using the **mk** script. For example to build the test program from the prompt in the upper directory, type:

$o/mk\ testf$

Then to execute the test program type:

$o/testf$

Most of the examples require that some data be supplied. If you get tired of supplying the data manually, you can modify the program or put the data in a file and execute the program using redirection by typing, for example:

$o/testf < testin.dat$

where the input data is stored in a file named *testin.dat*. This is an archaic way to run a program, but since we want to solve differential equations not build user interfaces, it is the best we can do.

All of the codes use an include file **defs.fi** which contains some global data. For example, the precision of the calculations is set the correct way, using a single parameter $float$. Also, many of the files have a $Debug$ parameter which causes some of the intermediate calculations to be printed when $Debug > 0$, larger values give more information.

The Fortran code has been completely reorganized from previous releases. It is now more accurate and efficient and is in a much easier and the correct object-oriented formulation. Unfortunately, it is not backward compatible, but it is easy to convert to the new format. The fundamental calculations of Chapter 2 are implemented with three files, each containing one module. They are **OrthPoly.f90**, **OrthCheb.f90** and **OCC.f90** and they contain modules: $Orth\_Poly$, $Orth\_Cheb$ and $OrthogonalColloc$, respectively. $Orth\_Poly$ and $Orth\_Cheb$ implement the most fundamental calculations for Jacobi and Chebyshev polynomials. These modules are used by $OrthogonalColloc$. Only a few of these most fundamental procedures are documented, but several example codes illustrate the functionality.

Fundamental Calculations

To solve problems, one normally interfaces only with $OrthogonalColloc$. One begins by first defining and initializing a $ColDataType$ object, called OC in this documentation. Once initialized the various quantities needed can be obtained using array valued functions. The code *testf.f90* demonstrates how to obtain and use most quantities of interest. The following data and functions are contained in the $OrthogonalColloc$ module:

General, Gauss, Lobatto, Chebyshev, RadauR, RadauL,Chebyshev1 – values (0 - 6) for point types types

Nonsymmetric, Planar, Cylindrical, Spherical – values (0 - 3) for geometry and symmetry.

TxtType(0:6) – text strings for point types

TxtGeom(0:3) – text strings for geometry

TxtSym(0:1) – text strings for symmetry

Initialize(OC,n,[TType],[Geometry],[Shift]) – initializes *OC* object for *n* interior points. *TType*, *Geometry* and *Shift* are optional. *TType* = 1 to 5, defaults to 2 (Lobatto), *Geometry* = 0 to 3, defaults to 0 (nonsymmetric), *Shift* = true (default) for interval [0,1], false for [-1,1], nonsymmetric problems.

Initialize(OC,Xc,[Geometry],[Shift]) – alternative initialization for general problems, where the points, *Xc*, are supplied. *Geometry* and *Shift* are the same as above.

CollocCoef(Xc,Wc,Ac,[Bc],[Cc]) – routine to get all the values at once, *Bc* and *Cc* are optional, All arrays have extent n + 2 (nonsymmetric) or n + 1 (symmetric).

Xc = Xpoints(OC) - returns quadrature points, given number interior points

Wc = WeightQ(OC) - returns quadrature weights

Wb = WeightQ(OC,Wbfac) - returns barycentric weights and normalization factor

Ac = Deriv1(OC) - returns first derivative matrix

An = DerivOdd(OC) - returns first derivative matrix for an odd quantity in a symmetric problem

Bc = Deriv2(OC) - returns second derivative or Laplacian matrix

Cc = Stiffness(OC) – returns stiffness matrix

Mc = MassMat(OC,[fx]) - mass matrix, where *fx* is optional. If *fx* is absent, f(x) = 1 is assumed. If *fx* is an array of function values at *Xc*, interpolation of f(x) is used.

Mc = MassMat(OC,Func,[nqx]) – alternate function for Galerkin or Moments mass matrix. *Func* is the name of a function to use in the calculation. *nqx* specifies *n* + *nqx* interior quadrature points will be used in the calculation. *nqx* = 1 is the default

Lc = Lcoeff(x,OC) – Lagrange polynomials through the points *Xc* of *OC* evaluated at *x*, result is shape(size(*x*),size(*Xc*)) if *x* is a vector or shape(*Xc*) if *x* is scalar

Lc = Lcoeff(x,xc) – alternate function for Lagrange polynomials evaluated at *x*. The interpolation points are, *xc*, result is shape(size(*x*),size(*xc*)) if *x* is a vector or shape(*Xc*) if *x* is scalar

y = Interp(x,OC,yc) – given the values *yc* at the points of *OC*, interpolates the values *y* at *x*. *x* may be a scalar or an array. *y* has the same size as *x*.

Q = Lpoly(xc) – Lagrange interpolating polynomial through *xc* as monomials, where $L_i(x) = \sum Q_{ij} x^{j-1}$

The *Orth_Poly* module contains code for the most fundamental calculations with Jacobi polynomials. Many are not needed to solve problems with a nodal formulation, but provides some functionality needed for modal formulations. Optional arguments common to several functions are:

abeta = array of two values containing α and β, the Jacobi polynomial weight parameters, defaults to (/0,0/), i.e. Legendre polynomials

arec – (n-1) x 4 array of recurrence coefficients, if not supplied values are calculated using *abeta*.

Monic - specifies monic polynomial, defaults to .false.

Shift - specifies shifted polynomial, defaults to .false.

Some functions in the *Orth_Poly* module are:

ar = x_Jacobi (n,[abeta],[Monic],[Shift]) – calculates recurrence relations for Jacobi polynomials. *ar(0:n-1,1:4)* values for polynomials, where if *Monic* is true *ar(:,1:2)* are the recurrence coefficients, Eq. (2.11) and *ar(:,3)* are coefficients of the leading terms, Eq. (2.14); if not *Monic* is false *ar(:,1:3)* are the recurrence coefficients, Eq. (2.16). *ar(:,4)* are the integrals of the squared polynomials, i.e. $\zeta_k^{(\alpha,\beta)}$, Eq. (2.7). If *Shift* is true the values are for the shifted polynomials.

d = d_Jacobi(n,[abeta],[Monic],[Shift]) – d(0:n) are the integrals of the squared polynomials, $\zeta_k^{(\alpha,\beta)}$. These are the same *ar(:,4)* above.

p = Legendre (x,n,nder) – p(:,0:n,0:nder) are 0 to n[th] Legendre polynomials and their derivatives through *nder* at vector of *x* values.

p = Jacobi_All(x,n,[abeta],[arec],[Monic],[Shift]) – p(:,0:n) are Jacobi polynomials 0 to *n* at vector of *x* values. *abeta* optionally contains the two weight parameters. *arec(0:n-1,4)* is optional array of recurrence coefficients from *x_Jacobi*. *abeta* is not used if *arec* is supplied.

u = Jacobi_Series(a,x,[abeta],[arec],[Monic],[Shift]) – values of discrete Jacobi series at vector of *x* values, i.e. $u(x) = \sum_{k=0}^{n} a_k p_k(x)$, *n* = size(*a*) - 1.

c = Jacobi_Deriv1(n,[abeta],[Monic],[Shift]) – returns 3 coefficients of Eq. (2.35)

c = Jacobi_Deriv2(n,[abeta],[nder],[Monic],[Shift]) – returns 3 coefficients of Eq. (2.33) for derivative *nder*

d = Jacobi_Deriv(n,[abeta],[ar],[Monic],[Shift]) – d(0:n-1,0:n-1) first derivatives for 0 thru (n-1)[th] Jacobi polynomials, type is a real(*float*). Eq. (2.38). i.e. $P_n^{(\alpha,\beta)\prime} = \sum_{k=0}^{n-1} d_{nk} P_k^{(\alpha,\beta)}$.

d = Legendre_Deriv(n,nder) – integer coefficients 0 to *n* for derivatives of *nth* Legendre polynomials, *nder* = 1 and 2 for 1[st] and 2[nd] derivative, Eqs. (2.44) and (2.45), *nder*

= 3 for difference $P''_{k+2} - P''_k$. This expresses the derivatives in terms of the undifferentiated polynomials, e.g. for a first derivative $P'_n = \sum_{k=0}^{n-1} d_k P_k(x)$

Q = LegendreTransform(Xc,Wc,ityp,[Shift]) – calculate transform from nodal values to Legendre coefficients. *Xc* and *Wc* are the nodes and weights for either Gaussian (*ityp* = 1) or Lobatto (*ityp* = 2) quadrature. Uses Eq. (2.140) for Lobatto case, while for the Gauss case the procedure ending with Eq. (2.148) is used.

a =Jacobi_Transform(n,t,g,Func,{Shift}) – calculates n^{th} degree Jacobi transform of function *Func* for points of type *t* and geometry *g* . Uses Eq. (2.138).

af = Jacobi_Scale_Sym(n,[abeta]) – calculates scaling factors, Eqs. (2.19) and (2.21), for shortcut polynomials or symmetric problems where *abs(abeta(2))* = 0.5. from shortcut polynomials of degree *n* to full polynomials of degree *2n + abeta(2) + ½*. *abeta* is optional, but since it defaults to (/0,0/) the result is *af* = 1

Example Codes

The example codes create one or up to three output files. For example, if you designate a root name of *ex*, file *ex.dat* and possibly *ex2.dat* and *ex.log* may be created. Several of the examples contain a parameter *Debug* to control output of intermediate calculations, which may create the third file *ex.log*.

testf.f90 –demonstrates calls to most of the functions in *OrthoganolColloc* and use of the results for integration and differentiation.

PolyTest.f90 – demonstrates some functions of the *Orth_Poly* module and demonstrates some methods for calculating polynomial roots.

PolyTran.f90 – demonstrates *Orth_Poly* functions for Jacobi transforms of several functions and how to calculate modal coefficients from nodal values

PolyBVP.f90 – demonstrates use of *Orth_Poly* functions to solve boundary value problems, section 3.1.7. Also demonstrates nodal/modal transforms and functions *Legendre_Deriv* and *Jacobi_Deriv*.

Ex01.f90 – solves constant or variable coefficient diffusion problem of section 3.1 with first order source by (1) conventional formulation with boundary collocation, (2) weak formulation with natural boundary condition, (3) Galerkin/Moments with interpolated source, (4) full Galerkin/Moments. Galerkin and Moments calculations occur when Lobatto and Gauss points are selected, respectively. Can be run interactively or with supplied data, e.g. *o/ex01 < o/x01in.dat.*

Ex01ex.f90 – extended version of **Ex01.f90**. This version calculates errors and transforms of residual functions (see section 3.1.6)

Ex01s.f90 – solves diffusion with nonlinear source Eq. (3.35) with rate Eq. (3.37) for symmetric problems in planar, cylindrical or spherical geometry. Set up to loop over a range of *n* and/or φ. φ can increment backwards or forward to follow

upper and lower solutions. Parameters $Conventional, Bcolloc\ and\ Generalized$ set within the code respectively govern the use conventional or weak formulation, boundary collocation, and whether supplied Thiele parameters are φ or generalized parameter φ^*. A data file is included, so it can be run interactively or by typing $o/ex01s < o/x01sin.dat.$

Ex01sg.f90 – a more complex version of **Ex01s.f90**. This version offers solution by collocation or Galerkin methods based on parameter $method$. It also calculates transforms of the residual function to produce the τ coefficients of Eq. (3.56)

Ex02ax.f90 – solves the boundary value problem of section 3.2 for a chemical reactor with axial dispersion. The three methods discussed in the text are coded. File *o/x02in.dat* contains data which can be used in place of interactive execution, i.e. *o/ex02ax* < *o/x02in.dat*. This code creates two or three output files. If you specify the file name ex you will get files $ex.dat$ and $ex2.dat$. If $Debug$ > 0 a file $ex.log$ is also created. This code is setup to treat output from a previous run as an "exact" solution for n = 99 Lobatto points from file $o/ex02axL99.dat$. If it is accidentally deleted, just run the program again to recreate the data.

Ex04ff.f90 – solution of falling liquid film problem by Orthogonal Collocation, continuous in z by eigenvalues and eigenvectors, section 4.1.1. The two output files contain average y, flux and Sh, and their errors vs z, and the solutions $y_i(z)$. The values and errors for the first 10 eigenvalues and coefficients are also listed.

Ex04ffg.f90 – solution of falling liquid film problem by the Galerkin Method, continuous in z by eigenvalues and eigenvectors, section 4.1.1. This code is like the collocation method code, but there are some interesting experimental calculations for the initial conditions.

Ex04ffs.f90 – solution of the falling liquid film problem with stepping methods of section 4.1.2. The code implements all the methods discussed in the monograph. The DIRK methods are implemented using a general framework, so others can be implemented by supplying the Butcher tableau. It creates two output files.

Utility Programs

ArrayPrint.f90 – the $Array\_Print$ module contains code to simplify output of tabular results. There is a short learning curve to use this module, but it can save a huge amount of time (which is the reason it was coded). If you look through the example codes, you will find many different uses of these routines. All routines are organized as follows:

$title$ – optional string, which is printed above the data.

data – a minimum of 1, maximum 4 array or vector quantities

fmtx – an optional format string

flog – is an optional logical file designator output unit used is: *iout* if not supplied, *ilog* if true and *iout2* if false (units defined in *defs.fi*)

call OpenFile(name,[ext],[append]) opens a file *name.ext* where name is the *root* and *ext* is an optional 3 character extension. If not supplied, an extension of *"dat"* is used. If supplied, *append* is appended to the root name. The unit opened is: *iout* if the extension is *"dat"* (including if not supplied), *ilog* if *ext* = *"log"*, *iout2* is opened if another extension is supplied or an *append* is supplied.

call VectorPrint(title,v1,v2,v3,v4,fmtx,flog) – to print one or more vectors (of type integer or *float*). Only *v1* is required. *v2, v3* and *v4* are optional additional vectors to be printed after *v1*.

call ArrayPrint(title,i1,i2,i3,i4,fmtx,flog) – to print one or more arrays of type integer. *i1* is required, all others are optional.

call ArrayPrint(title,a1,a2,a3,a4,fmtx,flog) – an overloaded routine to print a tab delimited table of one or more arrays or vectors of type *float* (or integers if all are arrays). Supply any combination of arrays and vectors. The number of rows printed is the smallest of the first dimensions. The number of columns printed is the total of the second dimensions (1 for a vector). For example if you have declared:

Real(float) :: x(20), a(5,5),r(5)

call ArrayPrint('Output:',x,a,r) - 5 rows by 7 columns printed. *x(6:20)* not printed.

The gyrations required to accomplish this shows some of the stupidity of Fortran 90. Why can't it treat a vector A(5) like an array A(5,1). Instead, I have set up numerous different routines and not covered all the possible combinations. Please enlighten me if there is an easier way to accomplish this task.

Linear Algebra Software:

The code relies on a combination of LAPack and non-LAPack routines for performing linear algebra. The routines are packaged into modules which provides a modern interface which takes care of work storage allocation and other bookkeeping. This approach makes them much easier to use.

If you do not have LAPack installed on your computer, you can still work all the examples, but you may have to make minor modifications to the code. The routine that rely on LAPack are deleted from **LUSolveX.f90** any calls to those routines should be changed, e.g. substitute function *LUSolveC* for *LUSolve*. For the eigenvalue calculations, the relevant LAPack routines are included in file **EigenLA.f90**, so the affected examples can be run by compiling and linking this file instead of the LAPack library.

LUSolve.f90 – contains module *LUSolvers* with the following public routines for solving systems of linear equations as follows:

LAPack routines:

X = LUSolve(A,b,[ipivot]) – solve Ax = b, *ipivot* is optional integer pivot array of size(*x*) uses LAPack DGETRF and DGETRS

call LUFactr(A,ipivot) – calculate LU factors of A, *ipivot* is pivot array of size(*x*), uses LAPack DGETRF

X = LUSubst(A,b,ipivot) – uses LU factors and *ipivot* from LUFactr to solve Ax = b, uses LAPack DGETRS

call LUInvert(A) – invert A, uses LApack DGETRF and DGETRI

LUSolveSym, LUFactrSym, LUSubstSym, and *LUInvertSym* – are just like the four routines above, but work on symmetric matrix problems. Works with the full matrix, i.e. no compact storage scheme is used. No *ipivot* is needed or used with these routines. Uses LAPack DPOTRF, DPOTRS, and DPOTRI.

LUSolveSB, LUFactrSB, LUSubstSB, and *LUInvertSB* – are just like the four routines above, but work on symmetric banded matrix problems. Only the diagonal and lower bands are supplied. No ipivot is needed or used with these routines. Uses LAPack DPBTRF, DPBTRS and DPBTRI.

Non-LAPack routines:

X = LUSolveC(A,b) – solve $Ax = b$, where b can be a single right-hand-side or an array of right-hand-sides. Uses non-LAPack routine implementing the Crout algorithm. No pivoting is used.

call LUFactrC(A) – calculate LU factors of A using Crout algorithm

X = LUSubstC(A,b) – uses A containing the LU factors from *LUFactrC* to solve $Ax = b$

LUSolveSq, LUFactrSq and *LUSubstSq* – are identical to the Crout routines above, but are for symmetric problems using full matrix storage and can use only one right-hand-side. Uses the square root method to produce LL^T factors of the matrix.

LUSolveLDL, LUFactrLDL and *LUSubstLDL* – are identical to the Crout routines above, but are for symmetric problems using full matrix storage and can use only one right-hand-side. Factors the matrix into LDL^T form. This should be the most efficient code for symmetric problems.

Eigen.f90 – contains module *EigenValue* with the following routines for eigenvalue calculations as follows:

w = EigenVal(A,[VR],[VL]) – for a nonsymmetric matrix A, solve for the eigenvalues and optionally the left, VL, and/or right, VR, eigenvectors. The real and imaginary parts of the eigenvalues are in an array *w(:,2)*. Uses LAPack routine DGEEV. Sorts the eigenvalues and eigenvectors from small to large.

w = EigenValGen(A,B, [VR],[VL]) – calculates the eigenvalues and optionally the left and right eigenvectors for a generalized nonsymmetric matrix problem $\boldsymbol{Av_r} = \lambda \boldsymbol{Bv_r}$. Uses LAPack routine DGGEV.

w = EigenValSym(A,B,[v],[typ]) – calculates the eigenvalues and optionally the eigenvectors for a general symmetric matrix problem. A is symmetric and B is symmetric positive definite. The configuration is designated by the value of optional *typ* (default = 1): (1) $\boldsymbol{Av} = \lambda \boldsymbol{Bv}$, (2) $\boldsymbol{ABv} = \lambda \boldsymbol{v}$, or (3) $\boldsymbol{BAv} = \lambda \boldsymbol{v}$. Uses LAPack routine DSYGV.

w = EigenValTriSym(AD,AS) – calculates the eigenvalues of a real symmetric tridiagonal matrix, where *AD* and *AS* are the diagonal and subdiagonal of the matrix. Uses LAPack routine DSTEQR.

C++ Code

occtest.cpp – code which sets up an *Orthcc* class and calls most of the class functions.

Fundamental Calculations

occ.h, occ.cpp – class *Orthcc* contains public functions. Definitions: *Gauss, Lobatto, Planar, Cylindrical, Spherical, Symmetric, Nonsymmetric* - const values useful for *occ_set*

occ_set(symm,typ,geom) – set symmetry (0/1 no/yes), type (1 Gauss, 2 Lobatto) and geometry(0/1/2 planar/cylindrical/spherical), can be done in constructor also quadrature(x,w,n) – calculate quadrature points and weights for n interior points

Acoeff(A,x,n) – calculate first derivative operator given x and n

Anonsym(An,A,x) – calculate first derivative operator (odd) given A and x

Bcoeff(B,A,x) – calculate second derivative operator given A and x

Ccoeff(C,A,x,w) – symmetric second derivative, given A, x and w

MassMat(D,x,w,n) – mass matrix given x, w and n

Lcoeff(Li,x,xi,nt) – Li are the Lagrange interpolating polynomials through x evaluated at xi, nt size(x), xi is scalar.

LcoeffN(L,x,xi,nt,ni) – L is a 2D array of the Lagrange interpolating polynomials through x evaluated at array of points xi, nt is size(x) and ni is size(xi)

Array.h, Array.cpp – All arrays in bold are type *Array2D*. A simple 2D array class. It also contains linear solver routines

Array() – default constructor, zero sized array

Array(n,m) – constructs n x m array

~Array() – destructor

void ArraySet(n,m) – resets to n x m array

*void ArrayChk(const char *Aname)* – for checking allocation status

Linear Algebra Software:

*int LUSolveA(double *b)* - solves linear equations with rhs b

int LUFactr() – factors matrix into LU form

*int LUSubst(double *b)* – solves linear equations with factored matrix and rhs b

Note: see note above (Fortran) stating the shape of the various arrays.

Excel Dynamic Link Library (OCCdll.dll)

The dynamic link library, OCCdll.dll, for use with Excel was created on a 32 bit machine, and has not yet been updated to 64 bits. You must put OCCdll.dll where it can be found by Excel (either in the "current" directory or better yet somewhere on the path).

Test.xls – simple spreadsheet which calls the various functions to get points, weights, derivatives and Lagrange polynomials.

OCC_Setup(s,t,g) – set symmetry, type and geometry

OCC_Points(nt) – returns points given total number

OCC_Weights(xc) – returns quadrature weights given points

OCC_Acoef(xc) – first derivative operator

OCC_AcoefN(xc) – first derivative operator for odd function

OCC_Bcoef(xc) – 2nd derivative operator

OCC_Ccoef(xc) – 2nd derivative operator symmetric form

OCC_Dcoef(xc) – mass matrix

OCC_Linterp(x,xc) – Lagrange interpolating polynomials evaluated at x

OCCdll.bas – visual basic code to interface with **OCCdll.dll**. This Visual basic code is needed to interface with the dynamic link library. This code is embedded in the Excel spreadsheets. It is already installed in **test.xls**. For a new spreadsheet, started from scratch, you will have to add this visual basic code.

LUFactor.bas – VBA code to interface with OCCdll.dll for solving systems of linear equations.

Ex01G.xls - solves the nonsymmetric diffusion/reaction example with Gauss points

Ex01L.xls - solves the nonsymmetric diffusion/reaction example with Lobatto points

Note: Excel has trouble keeping the spreadsheet up to date. Press ctrl-alt-F9 to force an update.