Computer Code Documentation

Larry C. Young, tildentechnologies.com

Table of Contents

To facilitate solution of problems with the described methods, some computer source code is provided. These codes have been run under Windows with MinGW/G95, Excel 2010, and g++, gfortran and Octave using Cygwin. Older versions were run with Lahey Fortran 95 and MS Visual C++. Native code has been written in Matlab/Octave, Fortran 90 and C++ for calculating the quadrature base points (collocation points), quadrature weights and various operators for derivatives, etc. The calculations and methods are described in Chapter 2 of the text. The Fortran 90 code was used to create a DLL which interfaced with Visual Basic to support Excel (only 32 bit currently).

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 Matlab functions or by modifying the demonstration code *testm.m*. The Fortran calculations can be examined by working with codes *PolyTest.f90* and *PolyBVP.f90* or by revising *testf.f90*. For example, the PolyTest code calculates the roots by four different methods. This code is an excellent starting point for further improvements, e.g. better initial estimates particularly for Gauss points.

Tabular results from all the codes are given in tab delimited form. I frequently pull these tables into Excel for testing and debugging. The fundamental calculations can also be done directly in Excel using the supplied dynamic link library. I discovered the

current dynamic link library does not work with newer 64 bit systems. A 64 bit library is being developed. I have not attempted to create Matlab/Octave graphs of the results, since Tecplot is better for this purpose. This tutorial is about solving equations. How to plot results can be found from many sources.

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
typ – type of points 0 Gauss, 1 Lobatto, 2 Chebyshev (2nd kind), etc.
geom – geometry 0/1/2 cartesian, cylindrical, spherical grid (symmetric problems)
xc – collocation points
wc – quadrature weights
Ac – first derivative matrix
An – first derivative of odd quantity (symmetric problem)
Bc – 2nd derivative Laplacian matrix
Cc – stiffness matrix representation, 2nd derivative Laplacian
Dc – mass matrix
Lc – Lagrange interpolating polynomial

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. With the improvements made for calculating roots, 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

testm.m – driver program to demonstrate how to perform fundamental calculations
OCCdefs.m – contains named constants for typ, 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

[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 unity.

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)

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

Example Codes

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 – bonus problem not described in text. Solves simple conduction problem with nonlinear thermal conductivity. The problem is described in Finlayson (1972), pp 16-19. Solved here using Orthogonal Collocation with Lobatto points. Treatment of the nonlinearity gives an exact result for the flux. Solution by several other methods can be found at:
http://faculty.washington.edu/finlayso/ebook

Fortran 90/95/20xx

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 a 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*. Some example files are included in the *o* directory. 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*. For example, the precision of the calculations is set the correct way, using a single parameter *float*. Many of the intermediate calculations are output when the parameter *Debug* > 0, larger values give more information. Many will claim that include files are wrong and a module should be used. However, modules have the downside that they must be compiled, which can sometimes make the build order complicated or impossible. Why make it complicated when a simple include file works just fine?

While working on Chapter 2, Fundamental Calculations, I developed some code to help me understand orthogonal polynomials. I tend to learn by doing, so I have included some of the experimental code I developed in a module Orth_Poly. These calculations are demonstrated by the drivers PolyTest and PolyBVP.

Fundamental Calculations

testf.f90 – a simple program which demonstrates calls to most of the functions below and use of the results for integration and differentiation.

OCC.f90 – module OrthogonalColloc contains public entities:

Gauss, Lobatto, Chebyshev, RadauR, RadauL – values (1-5) for types

Planar, Cylindrical, Spherical – values (0-2) for geometry.

TxtType(1:5) – text strings for point types

TxtGeom(0:2) – text strings for geometry

TxtSym(0:1) – text strings for symmetry

CollocSet(symm,typ,geom) - set symmetry (true/false), type (1-5) and geometry (0-2), all arguments are optional (can call using keywords)

CollocCoef(xc,wc,A,[B],[C]) – calls the various functions to get 5 arrays with one call, B and C are optional. All arrays are size(xc)

xc = Xpoints(n) - returns quadrature points, given number interior points

wc = Weights(xc) - returns quadrature weights

Ac = Acoeff(xc) - returns first derivative matrix

Bc = Bcoeff(xc,[Ac]) - returns second derivative matrix, Ac optional

Cc = Ccoeff(xc,[wc],[Ac]) – returns stiffness matrix wc and Ac optional

Dc = MassMat(xc,[Fx],[nx]) - Galerkin or Moments mass matrix, Fx and nx are optional: Fx is the name of a function in Eq.(2.64) and $n + nx$ interior quadrature points will be used. If omitted, the function will be unity and $nx = 1$.

An = AnonSym(xc,[Ac]) - first derivative of odd function, Ac is optional

Lc = Lcoeff(x,xc) – Lagrange polynomials through xc evaluated at x (may be scalar or array), result is shape(size(x),size(xc)) or shape(xc) if x is scalar

y = Interp(x,xc,yc) – given values yc at xc, estimate y at a vector of x values, xc is a vector, yc may be a vector of size(xc) or an array with first dimension the same as xc. y is either size(x) or shape(size(x),size(yc,2))

p = Lpoly(xc) – Lagrange interpolating polynomial as monomials $\sum p_{ij} x^{j-1}$

symm = CollocSym() – returns 0 or 1, the current setting for the symmetry

u = Weights_pp(xc) – barycentric weights

xc = Jacobi_Roots(n, [alpha],[beta],[Shift]) Jacobi polynomial roots for n interior points. Optional arguments: a = α and b = β, and logical Shift for shifted roots. See OrthPoly for defaults values Utilizes shortcut procedure when alpha = beta (see Section 2.2)

xc = Jacobi_Xroots(n,[alpha],[beta],[Shift]) Jacobi polynomial roots with a = α and b = β, logical Shift. Does not use shortcut procedure.

xc = JacobiRootsEstimate(n,alpha,beta) estimated value of Jacobi polynomial roots (interior points) with alpha = α and beta = β.

PolyTest.f90 – demonstrates many of the functions in the Orth_Poly module, e.g. recurrence coefficients, orthogonal polynomial values and derivatives, polynomial roots by 4 different calculations, Jacobi and Legendre transform demonstration. (to see calculation details, see description of parameter *Debug* above)

PolyBVP.f90 – solves linear boundary value problem using modal formulation with Legendre polynomial trial functions. It solves the problems described in section 3.1.6 with Moments, Galerkin and collocation methods. (to see calculation details, see description of parameter *Debug* above)

OrthPoly.f90 – Orth_Poly module contains experimental code for various calculations with orthogonal polynomials. Not needed to solve problems with nodal formulation, but provides some functionality needed for modal formulations. Optional arguments common to several functions are:

alpha = α defaults to 0, Jacobi polynomial weight parameter

beta = β defaults to alpha, Jacobi polynomial weight parameter

Monic - specifies monic polynomial, defaults to .true.

Shift - specifies shifted polynomial, defaults to .false.

Public entities are:

ab = x_Jacobi (n,[alpha],[beta],[Monic],[Shift]) – gives an array ab(0:n-1,1:4) of values for polynomials, where if Monic, ab(:,1:2) are the recurrence

coefficients, Eq. (2.11) and ab(:,3) are coefficients of the leading terms, Eq. (2.14); if not Monic, ab(:,1:3) are the recurrence coefficients. ab(:,4) are the integrals of the squared polynomials, Eq. (2.6). Shifted values are given in Eq. (2.16).

d = d_Jacobi(n,[alpha],[beta],[Monic],[Shift]) – d(0:n) are the integrals of the squared polynomials. These are the same ab(:,4) above.

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

p = Jacobi (x,n,[alpha],[beta],[ap],[Monic],[Shift]) – p(:,0:n) are 0 to nth Jacobi polynomials at vector of x values. ap(0:n-1,4) is optional array of recurrence coefficients from x_Jacobi

u = Jacobi_Calc (a,x, [alpha],[beta],[ap],[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.

d = Legendre_Deriv(n,der) – integer coefficients 0 to n for derivatives of nth Legendre polynomials, der = 1 and 2 for 1st and 2nd derivative, Eq. (2.42), and Eq. (2.43), der = 3 for difference $P''_{k+2} - P''_k$. This expresses the derivatives in terms of the undifferentiated polynomials, e.g. $P'_n = \sum_{k=0}^{n-1} d_k P_k(x)$

d = Jacobi_Deriv(n,alpha,beta) – d(0:n-1,0:n-1) first derivatives for 0 thru (n-1)th Jacobi polynomials, type is *float*, Eq. (2.45). i.e. $P_n^{(\alpha,\beta)\prime} = \sum_{k=0}^{n-1} d_{nk} P_k^{(\alpha,\beta)}$.

Q = LegendreTransform(xc,wc,typ) – Legendre transform, Eq. (2.47). xc and wc are the quadrature points and weights, ityp = 2 for Lobatto. May be used for any quadrature, but correct transform only for Lobatto quadrature.

Q = JacobiTransform(xc,wc,typ) – Legendre transform calculated via Jacobi transform, see Eq. (2.56). xc and wc are the quadrature points and weights, and ityp = 1 for Gauss. Transform correct for Gauss, Lobatto or Radau quadrature.

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. May use call ArrayPrint (preferred) or ArrayPrintI.

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 9 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 must be deleted from LUSolve.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. $\boldsymbol{A}$ is symmetric and $\boldsymbol{B}$ 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.

Example Codes

The example codes create one or two output files. For example, if you designate a root name of “ex”, file *ex.dat* and possibly *ex2.dat* will be created. They rely on parameter *Debug* (see discussion of *defs.fi* above) to control output of intermediate calculations, which will create a third file *ex.log*. There are also some commented output statements that can be activated.

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. It does exact solution and error calculations for constant coefficient problem and the variable coefficient problem discussed in the text. Can be run interactively or with supplied data, e.g. *o/ex01 < o/x1in.dat*.

Ex02s.f90 – solves diffusion with nonlinear source (Eq. 3.44) for symmetric problems in planar, cylindrical or spherical geometry. Set up to loop over a range of n and φ. φ can increment backwards or forward to follow upper and lower solutions. Parameters provided to select weak or conventional formulation and boundary condition treatment. With parameter change program can work with supplied values of either φ or generalized parameter φ^*. A data file is included, so it can be run interactively or by typing *o/ex02s < o/x2in.dat*

Ex03ax.f90 – solves the boundary value problem of section 3.2 for a reactor with axial dispersion. The three methods discussed in the text are coded. File *o/x3in.dat* contains data which can be used with redirection as described above, i.e. *o/ex03ax < o/x3in.dat*. This code creates two or three output files. If you specify the file name “root” you will get files *root.dat* and *root2.dat*. If *Debug* > 0 (in *defs.fi*) a file *root.log* is also created. This code reads “exact” solution for n = 99 Lobatto points from file *o/ex3Lexact.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. The Galerkin

method is only briefly mentioned in the text. 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 code. The DIRK methods are implemented using a general framework, so others can be implemented by supplying the Butcher tableau. It creates two output files.

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 for use with Excel was created on a 32 bit machine, and has not yet been updated to 64 bits. This work is in progress.

Test.xls – simple spreadsheet which calls the various functions
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) – 2^{nd} derivative operator
OCC_Ccoef(xc) – 2^{nd} 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.

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