Appendix H

Codes

H.1 Codes for Chapter 1

precision_1.f

```
program precision1

    real*4   s1(10000000), s2(10000000)
    integer i, N
    character*2 eps

    write(6,*) "Enter precision"
    read(5,*) eps

    if (eps .eq. "sp")  then
    open(11,file="single_precision_sum.data",status="unknown")
    else
    open(11,file="double_precision_sum.data",status="unknown")
    endif

    do N = 1e+6,1e+7,1e+6

    Forward sum

    s1(1) =1.
    do i=2,N,1
      s1(i) = s1(i-1)  + 1./i
    enddo
```

```
      Backward sum

      s2(N+1)= 0
      do i=N,1,-1
        s2(i) = s2(i+1)  + 1./i
      enddo

       write(11,*) N, s1(N), s2(1)

       enddo

       close(11)
       end
```

single_precision_sum.data

```
  1000000  14.357358  14.3926516
  2000000  15.3110323  15.0865793
  3000000  15.4036827  15.4919109
  4000000  15.4036827  15.7773352
  5000000  15.4036827  16.0078545
  6000000  15.4036827  16.1824932
  7000000  15.4036827  16.3368549
  8000000  15.4036827  16.4742279
  9000000  15.4036827  16.6022205
  10000000  15.4036827  16.6860313
```

H.2 Codes for Chapter 2

lup_decomposition.f90

```
  !
  ! lup_decomposition.f90
  !
  ! L U Decomposition of a matrix A into L U P
  !     P A = L U
  !
module lup_decomposition
```

```
implicit none
contains
subroutine lup_decomp(A, L, U, IDX)
real, dimension(:,:), intent(in) :: A
real, dimension(:,:), intent(out) :: L
real, dimension(:,:), intent(out) :: U
integer, dimension(:), intent(out) :: IDX
integer :: n
real :: P
real :: T
integer :: kp, kt, i, j, k

n = size(A,dim=1)  ! should be square
L = A
do i = 1,n
  IDX(i)=i
end do
do k=1,n-1
  P=0.0

  do i=k,n
    if( abs(L(i,k)) > P) then
      P=abs(L(i,k))
      kp=i
    end if
  end do

  if(P == 0.0) then
    print *, 'singular matrix'
    return
  end if

  kt=IDX(k)
  IDX(k)=IDX(kp)
  IDX(kp)=kt
  do i=1,n
    T=L(k,i)
    L(k,i)=L(kp,i)
    L(kp,i)=T
  end do

  do i=k+1,n
    L(i,k)=L(i,k)/L(k,k)
    do j=k+1,n
      L(i,j)=L(i,j)-L(i,k)*L(k,j)
```

```
      end do
    end do
  end do

  do i=1,n
   do j=1,n
     if(i > j)then
       U(i,j)=0.0
     else
       U(i,j)=L(i,j)
       L(i,j)=0.0
     end if
    end do
    L(i,i)=1.0
  end do
  return
end subroutine lup_decomp

! given A decomposed into L, U, P  and given Y, solve for X
!  A x = y

subroutine lup_solve(L, U, IDX, X, Y)
   real, dimension(:,:), intent(in)  :: L
   real, dimension(:,:), intent(in)  :: U
   real, dimension(:), intent(out)   :: X
   real, dimension(:), intent(in)    :: Y
   integer, dimension(:), intent(in) :: IDX
   integer :: n
   real, dimension(size(Y)) :: B
   integer :: i, j
   real :: sum

  n = size(L,dim=1)  ! square matrix, all dimensions start with 1
  do i=1,n
    sum=0.0
    do j=1,i-1
      sum=sum+L(i,j)*B(j)
    end do
    B(i) = Y(IDX(i)) - sum
  end do

  do i=n,1,-1
    sum=0.0
    do j=i+1,n
      sum=sum+U(i,j)*X(j)
```

```
    end do
    X(i) = (B(i)-sum)/U(i,i)
  end do

  return
end subroutine lup_solve

subroutine make_perm(P, IDX)
   real, dimension(:,:), intent(out) :: P
   integer, dimension(:), intent(in) :: IDX
   integer :: n
   integer :: i, j

  n = size(IDX,dim=1)
  do i=1,n
    do j=1,n
      P(i,j)=0.0
    end do
  end do
  do i=1,n
    P(i,IDX(i))=1.0
  end do
  return
end subroutine make_perm
end module lup_decomposition
```

play_with_lup_decomp.f90

```
! play_with_lup_decomp.f90
!
! test program for solving simultaneous equations and matrix inverse by the
! L U Decomposition method, using the general solution that requires the
! permutations P
!
! We will test on three matrices: A1, A2 and A3
!
!
! hint:
!
! play_with_lup_decomp.f90 DEPENDS on lup_decomposition.f90 (a module
!  called by the "use" command below). So, if I wanted to compile
!  it interactively I would use the following:
```

```
!
!  gfortran -o play_lup.x lup_decomposition.f90 play_with_lup_decomp.f90
!
!  DO NOT compile interactively ---> use a Makefile
!

program play_with_lup_decomp
  use lup_decomposition          ! bring in the LUP module
  implicit none
  integer, parameter   :: n = 4  ! keep it small, must fit data
  real, dimension(n,n) :: A1 = &
      reshape( (/4.0, 4.0, 3.0, 3.0,   5.0, 1.0, 1.0, 2.0, &
                 1.0, 4.0, 3.0, 2.0,   1.0, 2.0, 1.0, 4.0/), shape(A1))
  real, dimension(n,n) :: A2 = &
      reshape( (/1.0, 2.0, 3.0, 4.0,   2.0, 2.0, 3.0, 4.0, &
                 3.0, 3.0, 3.0, 4.0,   4.0, 4.0, 4.0, 4.0/), shape(A2))
  real, dimension(n,n) :: A3 = &
      reshape( (/4.0, 4.0, 4.0, 4.0,   4.0, 3.0, 3.0, 3.0, &
                 4.0, 3.0, 2.0, 2.0,   4.0, 3.0, 2.0, 1.0/), shape(A3))

  real, dimension(n,n) :: AA  ! temporary
  real, dimension(n)   :: X   ! computed by solve
  real, dimension(n), parameter :: Y1 = (/ 3.5, 2.4, -1.2, 6.1 /)
  real, dimension(n), parameter :: Y2 = (/30.0, 31.0, 34.0, 40.0/)
  real, dimension(n), parameter :: Y3 = (/16.0, 13.0, 11.0, 10.0/)
  real, dimension(n,n) :: L   ! lower triangular matrix computed
  real, dimension(n,n) :: U   ! upper triangular matrix computed
  real, dimension(n)   :: Z   ! scratch vector for checking solve
  real, dimension(n,n) :: P   ! permutation matrix, used for checking
  integer, dimension(n) :: IDX  ! permutation indices, computed

  print *, "##################################################"
  print *, "Playing with LUP Decomposition for PHYS499, case 1"

  print *, 'A1=  ', A1
  call lup_decomp(A1, L, U, IDX)
  print *, 'U=  ', U
  print *, 'L=  ', L
  print *, 'IDX= ', IDX
  print *, ''

  call lup_solve(L, U, IDX, X, Y1)
  print *, 'X=  ', X
  print *, 'Y1=  ', Y1
  Z = matmul(A1, X)
```

```
print *, "Z=  ", Z
print *, "If Z equal Y1 ---> no refinement required"
call make_perm(P, IDX)  ! IDX vector becomes P matrix
print *, 'P= ', P
AA = matmul(P, A1) - matmul(L, U)
print *, 'P*A1-L*U=0 ', AA
print *, ' '


print *, "##################################################"
print *, "Playing with LUP Decomposition, case 2"

print *, 'A2= ', A2
call lup_decomp(A2, L, U, IDX)
print *, 'U=  ', U
print *, 'L=  ', L
print *, 'IDX= ', IDX
print *, ''

call lup_solve(L, U, IDX, X, Y2)
print *, 'X=  ', X

print *, 'Y2= ', Y2
Z = matmul(A2, X)
print *, 'Z= ', Z
print *, "If Z equal Y2 ---> no refinement required"
call make_perm(P, IDX)
print *, 'P= ', P
AA = matmul(P, A2) - matmul(L, U)
print *, 'P*A2-L*U=0 ', AA
print *, ' '


print *, "##################################################"
print *, "Playing with LUP Decomposition, case 3"

print *, 'A3= ', A3
call lup_decomp(A3, L, U, IDX)
print *, 'U=  ', U
print *, 'L=  ', L
print *, 'IDX= ', IDX
print *, ''

call lup_solve(L, U, IDX, X, Y3)
print *, 'X=  ', X
```

```
  Z = matmul(A3, X)
  print *, 'Y3= ', Y3
  print *, 'Z= ', Z
  print *, "If Z equal Y3 ---> no refinement required"
  call make_perm(P, IDX)
  print *, 'P= ', P
  AA = matmul(P,A3) - matmul(L,U)
  print *, 'P*A3-L*U=0 ', AA
  print *, ' '

  print *, 'finished LUP Decomposition'
end program play_with_lup_decomp
```

module refine_mod

```
 !**********************************************************************
 !
 module refine_mod
         implicit none
         contains
         subroutine refine(A,L,U,N,NP,INDX,B,X)
         use lubksb_mod
*********************************************************************72
        GIVEN THE NxN MATRIX  A  WITH PHYSICAL DIMENSIONS NPxNP,
        AND LU DECOMPOSITION ALU, X THE APPROX SOLUTION
        OF   AX = b
        USE ITERATIVE REFINEMENT TO IMPROVE THE SOLN
        indx is output vector which records row permutations

         implicit double precision(a-h,o-z)
         INTEGER :: N , Np, i, j
         REAL :: spd
         integer, parameter   :: nmax=100
         real, parameter   :: tiny=100
         parameter (nmax=100,tiny=1.0e-20)
         REAL , DIMENSION(Np,Np) :: A, L, U
         REAL , DIMENSION(N) :: B, X
         REAL , DIMENSION(nmax) :: r, dX
         INTEGER , DIMENSION(N) :: indx


        do i=1,n
          spd=-B(i)
```

```
            do j=1,n
             spd=spd+A(i,j)*X(j)
            enddo
        r(i)=spd
        enddo

      call lup_solve_2(L, U, indx, dX, r)

      do i=1,n
          X(i)=X(i)-dX(i)
      enddo

!      return
end subroutine refine

 subroutine lup_solve_2(L, U, IDX, X, Y)
 real, dimension(:,:), intent(in)  :: L
 real, dimension(:,:), intent(in)  :: U
 real, dimension(:), intent(out)   :: X
 real, dimension(:), intent(in)    :: Y
 integer, dimension(:), intent(in) :: IDX
 integer :: n
 real, dimension(size(Y)) :: B
 integer :: i, j
 real :: sum

 n = size(L,dim=1)  ! square matrix, all dimensions start with 1
 do i=1,n
   sum=0.0
   do j=1,i-1
     sum=sum+L(i,j)*B(j)
   end do
   B(i) = Y(IDX(i)) - sum
 end do

 do i=n,1,-1
   sum=0.0
   do j=i+1,n
     sum=sum+U(i,j)*X(j)
   end do
   X(i) = (B(i)-sum)/U(i,i)
 end do

  return
end subroutine lup_solve_2
```

```
end module refine_mod

 !*******************************************************************
```

LU_on_circuits.f90

```
 ! LU on electric circuits
 !


program LU_on_circuits
  use lup_decomposition
  implicit none
  integer :: n
  real, dimension(:,:), allocatable  :: A
  real, dimension(:), allocatable    :: X
  real, dimension(5), parameter     :: B =  (/1.0,0.0,0.0,0.0,0.0/)
  real, dimension(:,:), allocatable  :: L   ! lower triangular matrix computed
  real, dimension(:,:), allocatable  :: U   ! upper triangular matrix computed
  real, dimension(:), allocatable    :: Z   ! scratch vector for checking solve
  integer, dimension(:), allocatable :: IDX ! permutation indices, computed
  integer :: iostat, stat
  integer :: i, j
  integer :: bool=0


  print *, "!!!!!!!!!!!!!!!!BEGIN!!!!!!!!!!!!!!!!"
! open(unit=11, file="circuits.data", action="read", iostat=iostat)
  open(unit=11, file="midterm_in.data", action="read", iostat=iostat)
  if(iostat /= 0) then
    print *, "can not open file:  circuits.data"
    stop
  end if

  do
    read(unit=11, fmt="(i2)", iostat=iostat) n
    if(iostat < 0) exit  ! end of file
    if(iostat > 0) cycle ! bad data
    print *, "allocating working arrays of size ", n
    allocate(A(n,n), stat=stat)
    if(stat > 0) print *, "out of memory"
    allocate(L(n,n))
```

```
    allocate(U(n,n))
    allocate(X(n))
    allocate(Z(n))
    allocate(IDX(n))

    do i=1,n
!     read(unit=11, fmt="(11F6.1)", iostat=iostat) (A(i,j),j=1,n)
      read(unit=11, fmt="(5F10.6)", iostat=iostat) (A(i,j),j=1,n)
      if(iostat /= 0) print *, "may have bad data"
    end do
    print *, 'A=  ', A

!
    call lup_decomp(A, L, U, IDX)
    print *, 'U=  ', U
    print *, 'L=  ', L
    print *, 'IDX= ', IDX
    print *, ''
!
    call lup_solve(L, U, IDX, X, B)
    print *, 'Estimated   X=  ', X
!
    Z = matmul(A, X)
    do i=1,n
    if (Z(i) /= B(i)) bool=1
    enddo

    deallocate(A)
    deallocate(L)
    deallocate(U)
    deallocate(X)
    deallocate(Z)
    deallocate(IDX)
  end do
  print *, "!!!!!!!!!!!!!!!!!DONE!!!!!!!!!!!!!!!!"

contains
RECURSIVE FUNCTION factorial(n) RESULT(nfact)
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: n
  INTEGER :: nfact
  IF(n > 0) THEN
     nfact = n * factorial(n-1)
  ELSE
     nfact = 1
```

```
  END IF
END FUNCTION factorial

end program LU_on_circuits
```

circuits.data

```
 11 11
 1.0    -1.0   -1.0    0.0   0.0    0.0    0.0    0.0    0.0    0.0    0.0
 0.0    1.0    0.0    -1.0   -1.0   0.0    0.0    0.0    0.0    0.0    0.0
 0.0    0.0    1.0    1.0    0.0    -1.0   0.0    0.0    0.0    0.0    0.0
 0.0    0.0    0.0    0.0    1.0    1.0    -1.0   0.0    0.0    0.0    0.0
14.0    0.0    0.0    0.0    0.0    0.0    0.0    1.0    0.0    0.0    0.0
 0.0    2.0    0.0    0.0    0.0    0.0    0.0    -1.0   1.0    0.0    0.0
 0.0    0.0    6.0    0.0    0.0    0.0    0.0    -1.0   0.0    1.0    0.0
 0.0    0.0    0.0    0.1    0.0    0.0    0.0    0.0    -1.0   1.0    0.0
 0.0    0.0    0.0    0.0    7.0    0.0    0.0    0.0    -1.0   0.0    1.0
 0.0    0.0    0.0    0.0    0.0   15.0    0.0    0.0    0.0    -1.0   1.0
 0.0    0.0    0.0    0.0    0.0    0.0    5.0    0.0    0.0    0.0    -1.0
```

H.3 Codes for Appendix C

gnuplot-fractal.gp

```
# plot and splot are the primary commands in Gnuplot.
# plot is for 2D functions and data, while splot plots 3D surfaces
# and data
#
# When using splot it would need three sets of data:
# say you have a data file with 3 columns:  x, y and z
# splot 'myfile.data' will plot 3D surfaces of the z
# variable at coordinates (x,y)
#
# whereas plot would only need two sets of data say x and y
# in order to plot for example y versus x
#

## First, start with the following simple example
```

```
set xrange [-1.5:0.5]
set yrange [-1:1]
set logscale z
set isosample 50
set hidden3d
set contour
splot cos(x)*sin(x) notitle

## Secondly, simply read the data from the file
## you created from your Fortran code for fractals.
##

#splot 'myfractal.data'

##  Here are usefull gnuplot tricks that
##  will help control the duration of the display

# pause 5  (wait for 5 secondes)

## the one I prefer is

pause -1 "Press Return to quit"
```