2017-08-17

2D Surface Creation Using Intel MKL

R. Steven Turley
Brigham Young University, turley@byu.edu

2D Surface Creation Using Intel MKL

Steve Turley

July 18, 2017

Contents

1. Introduction

This document illustrates how to use the Intel Math Kernel Library (which is installed on nystrom and marylou) to create surfaces with a given cut-off spatial frequency and rms height. I've included example Fortran-95 code which is included in its entirety in the Appendix.

In this introduction, I'll go through the theory of what the program accomplishes. Implementation details are found in later sections. Section 2 discusses how to set up the frequency filter Section 3 discusses how to force the desired RMS surface height. Section 4 has implementation details about the inverse Fourier Transform. The plot routines are described in Section 5. Section 6 discusses how to interolate to arbitrary points on the FFT surface ans find the partial derivatives of the surface there. Finally, Section A discusses how to use cmake to compile and link the example programs.

Constructing the surface involves the following steps:

1. Create random surface height.
2. Transform the surface from the to k-space using a Fourier Transform (FFT).
3. Pass the amplitudes through a low pass filter.
4. Use an inverse FFT to reconstruct the filtered surface from the frequency amplitudes.
5. Adjust the height of the surface to have the desired RMS height.
6. Interpolate from the Fourier points to the desired surface points.

1.1. Creating Random Frequency Amplitudes

In order to compute the effects of roughness from a variety of arbitrary surfaces, it is important to be able to geneate different surfaces each time. Doing so requires generating random values for the frequency amplitudes. Random number generators in FORTRAN aren't truely random. Rather, they produce a sequence of uncorrelated numbers based on an initial starting point or seed. In the examples below, I'll use the same default see each time. To get a random seed, you could set the seed using the system clock. Here is an example code snippet to do that.

```
integer :: seed_size, clock
integer, allocatable :: seed(:)

call random_seed(size=seed_size)
allocate(seed(seed_size))
call system_clock(clock)
seed = clock/1000 + 37 * (/ (i,i=0,n-1) /)
call random_seed(put=seed)
deallocate(seed)
```

the system random number generator in FORTRAN generates a number which is equally likely to have any value between 0 and 1. It's physically more realistic to expect the surface heights to have a gaussian probability distribution.

$$\mathcal{P}_x \propto \exp\left(-\frac{x^2}{2\sigma^2}\right) \tag{1}$$

where σ is the rms surface height. The following module uses the Box-Muller transform to create pseudorandom numbers centered about 0 with unit variance. These number are generated from the standard Fortran random number generator, so they can be seeded the same way outlined above.

```
module normal
  !"use Box-Muller transform to create pseudorandom numbers
  ! centered about zero with a unit invariance"

  use iso_fortran_env,only:real64
  implicit none
  private

  real(real64)::second
  logical::first=.TRUE.

  public nrand

contains
  function nrand()
    real(real64)::n(2),sr,nrand
    real(real64),parameter::pi=4.0d0*atan(1.0d0)
    if(first) then
       call random_number(n)
       sr = sqrt(-2.d0*log(n(1)))
       nrand=sr*cos(2*pi*n(2))
       second=sr*sin(2*pi*n(2))
       first=.FALSE.
    else
       nrand=second
       first=.TRUE.
    end if
  end function nrand
end module normal
```

This module is utilized in the random_surface subroutine to generate the initial surface heights.

```
  subroutine filter(z,sigma)
    use iso_fortran_env
    implicit none
    real(real64), intent(inout) :: z(:,:)
    real(real64), intent(in) :: sigma
    integer :: i, j
    real(real64)::x
```

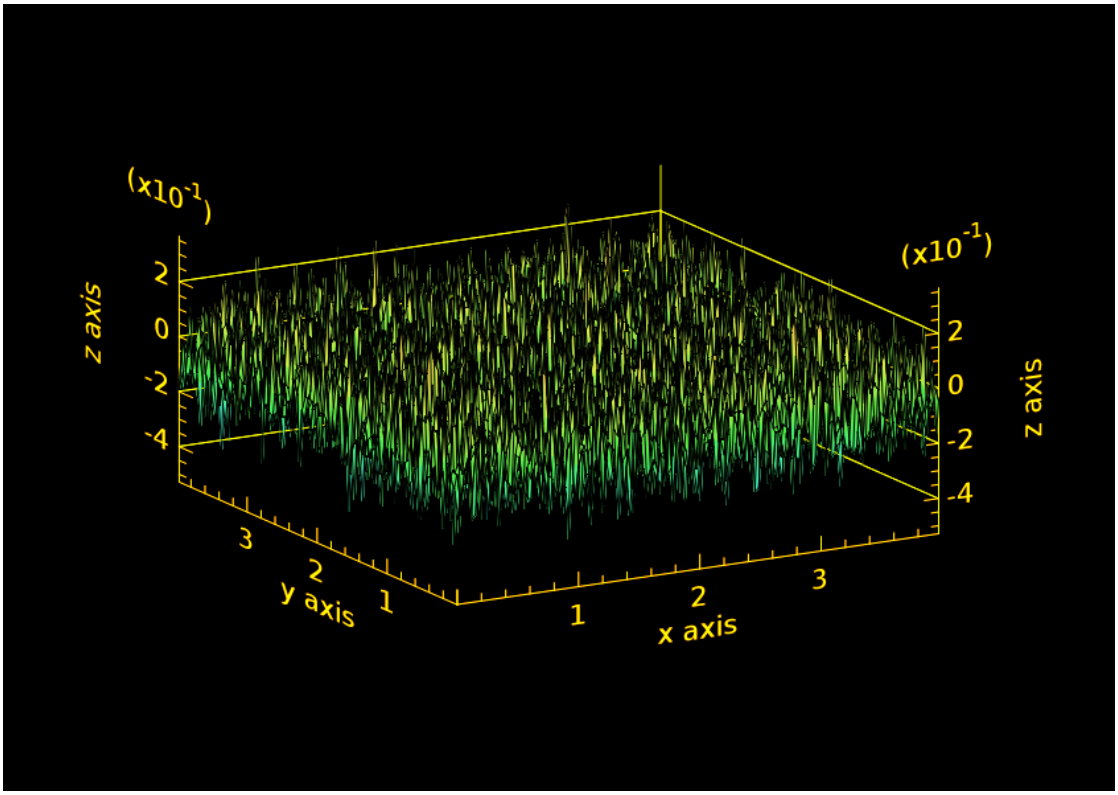

Figure 1: Original random surface before filtering.

```
    do i=1,size(y,1)
        x = i-1
        y(i)=y(i)*exp(-x**2/(2*sigma**2))
    end do
end subroutine filter
```

Figure 1 is a graph of the random surface generated by this routine before filtering.

1.2. Fourier Transform

In order to get a surface dominated by low spatial frequencies, the surface first has to be transformed to the spatial frequency domain. We do this using the Intel Math Kernel Library (MKL).

The symbols needed to control the FFT are included with the statement

```
use MKL_DFTI
```

The initialization then requires the following steps:

1. Setting the x and y dimensions of the FFT

2. Creating a descriptor

3. Committing the descriptor

Here is a code fragment the accomplishes these steps.

```
integer, intent(out) :: status
integer :: l(2)
type(DFTI_DESCRIPTOR), POINTER :: handle

! Initialize data descriptors
l(1)=nx
l(2)=ny
status = DftiCreateDescriptor(handle, DFTI_DOUBLE,&
      DFTI_COMPLEX, 2, l)
status = DftiCommitDescriptor(handle)
```

The array l of dimension 2 hold the length of the x and y data for the transform. The call to DftiCreateDescriptor specifies that we are doing a double precision transform of an array of complex numbers. The third argument is the number of dimensions in the transform. If changes to the default parameters were needed, it would be done between the call to DftiCreateDescriptor and DftiCommitDescriptor. After the required space is computer and defaults are set for various parameters, the call to DftiCommitDescriptor allocates this space and precomputes some needed constants.

After initialization, the forward transform is accomplished in a single call. All of the DFTI routines return a status integer which is 0 on success and has a non-zero error code otherwise.

```
status = DftiComputeForward(handle,z)
```

1.3. Low Pass Filter

Once in the spatial frequency domain, random frequency amplitudes need to be passed through a low pass filter. At first, a simple cut-off frequency might seem appropriate, but that will add ringing to the surface after the Fourier transform (like the side bands that appear from passing through a slit with a sharp cut-off). There are a number of filters that are commonly used in signal processing to provide smooth frequency cut-offs. The Wikipedia page at `https://en.wikipedia.org/wiki/Window\_function` discusses some of the possibilities. I chose a half-gaussian filter for our case.

$$f(\nu) = e^{-\nu^2/2\sigma^2} \tag{2}$$

where ν is the frequency in Fourier space. Another possibility to try would be an error function ($\mathrm{erf}(x)$) joined to a constant function.

In applying the filter, I have assumed the δk is the same in the x and y directions.

Figure 2 is the filtered surface corresponding the the data in Figure 1. Note how the characteristic length scale between bumps on the surface is much large after the filtering.

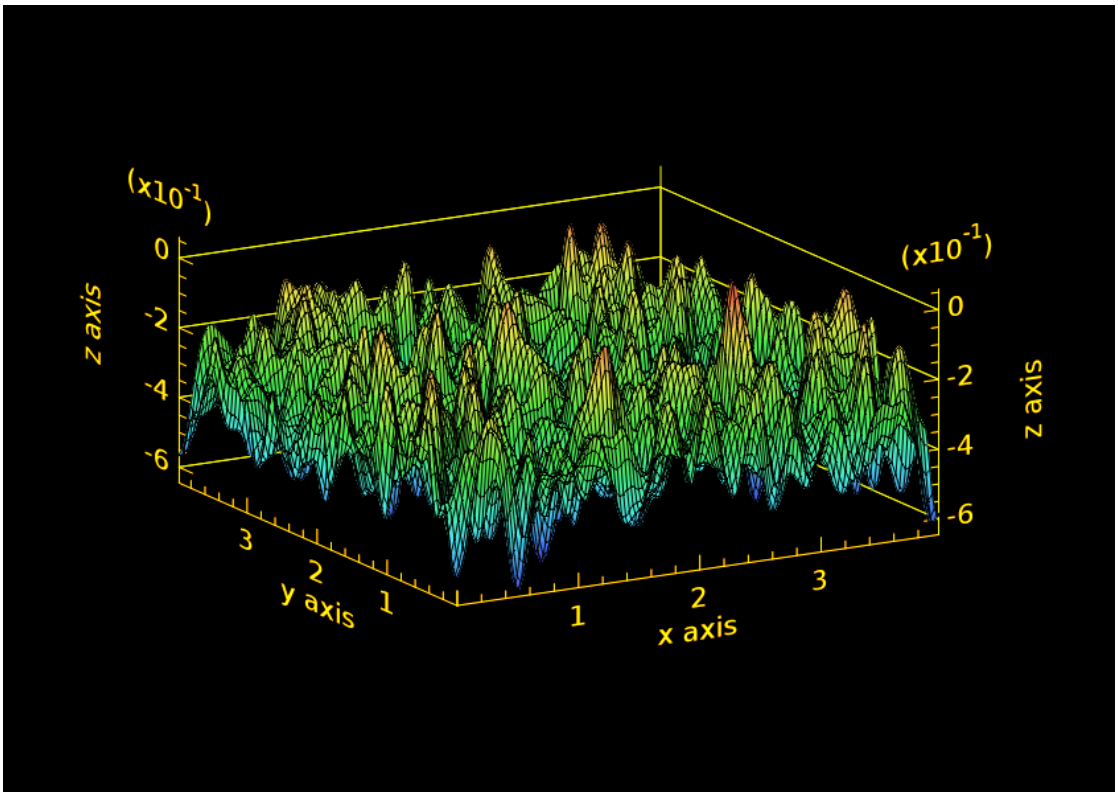

Figure 2: Random surface after filtering.

1.4. Setting RMS Height

Depending on the width of the filter, the surface in real space will have a varying RMS height. To select a particular RMS height, one needs to calculate the RMS height of the surface and then scale it as desired. Let $\langle h \rangle$ be the mean height and $\langle h^2 \rangle$ be the mean of the square of the height. The RMS height is

$$\Delta h = \sqrt{\langle h^2 \rangle - \langle h \rangle^2}. \tag{3}$$

If the desired height is δ, that can be achieve by multplying each height y_i by $\delta / \Delta h$.

2. Setting Frequency Filter

Once in the spatial frequency domain, we can filter out the highest frequencies in the surface noise. This is complicated by the fact that the FFT doesn't store the transformed data completely in order of increasing frequency. For a transform with an even number of points, the transform first stores the positive frequencies

$$k_i = \frac{(i-1)2\pi}{L} \, , i = 1, \frac{n}{2} + 1 \tag{4}$$

followed by the negative frequencies in increaseing order.

$$k_i = \frac{(i-1)2\pi}{L} - \frac{2n\pi}{L} \, , i = \frac{n}{2} + 2, n \tag{5}$$

For ease of computation, I scale the k_i values by the frequency increment

$$dk = \frac{2\pi}{L} \tag{6}$$

so that the actual values of k_i are integers.

$$k_i = \begin{cases} i-1 & \text{for } i \leq \frac{n}{2} + 1 \\ i-n-1 & \text{for } i > \frac{n}{2} + 1 \end{cases} \tag{7}$$

Here is the code snippet to filter the random frequency surface with a cut-off frequency of σ. To allow for different surface resolutions, I found it easiest to

```
subroutine filter(z,sigma)
  use iso_fortran_env
  implicit none
  complex(real64), intent(inout) :: z(nx,ny)
  real(real64), intent(in) :: sigma
  integer :: i,j
  real(real64)::kx,ky,ksq,zf

  ! Assume even number of points
```

```
    do i=1,nx
        if(i <= nx/2+1)then
            kx=i-1
        else
            kx=i-nx-1
        end if
        ! if j = 1, do nothing, we are multiplying by 1
        do j=1,ny
            if(j <= ny/2+1) then
                ky = j-1
            else
                ky = j-ny-1
            end if
            ksq=kx**2+ky**2
            zf = exp(-ksq/(2*sigma**2))
            z(i,j)=z(i,j)*zf
        end do
    end do
end subroutine filter
```

The filtering is followed by an inverse FFT to transform the data back to the spatial domain and convert the transform from complex values (with a zero imaginary part) to a real array.

```
    status = DftiComputeBackward(handle,z)
    s = real(z)/(nx*ny)
```

I have verified that the imaginary parts are close to zero after that transform, which they wouldn't be if different spatial frequencies had different attenuations.

3. Setting RMS Height

After an inverse fourier transform, we set the RMS of the surface to the desired value σ using the following routine.

```
subroutine setrms(y, sigma)
    use iso_fortran_env
    implicit none
    real(real64), intent(inout) :: y(:)
    real(real64), intent(in) :: sigma
    real(real64) :: sm, smsq, rms
    integer :: n

    n = size(y)
    sm = sum(y)/n
    smsq = sum(y*y)/n
```

```
    rms = sqrt(smsq-sm**2)
    y = y*sigma/rms
  end subroutine setrms
```

4. FFT Routines

The MKL FFT routines require setup, the FFT call, and then cleanup of the auxilliary storage. The routines return a status in the stat variable which will be zero on success. I haven't been checking the status to make sure it is zero in the code, but we probably should. There is a POINTER variable 'handle' of type DFTI_DESCRIPTOR declared at the top level of the module to provide storage for internal data needed by the FFT.

```
    type(DFTI_DESCRIPTOR), POINTER :: handle
```

Here is the initialization routine which needs to be called first.

```
  subroutine fft_init(status)
    use MKL_DFTI
    use iso_fortran_env
    integer, intent(out) :: status
    integer :: l(2)

    ! Initialize data descriptors
    l(1)=nx
    l(2)=ny
    status = DftiCreateDescriptor(handle, DFTI_DOUBLE,&
         DFTI_COMPLEX, 2, l)
    status = DftiCommitDescriptor(handle)
  end subroutine fft_init
```

Here is the routine for the forward FFT.

```
    subroutine fft_forward(z, status)
    use iso_fortran_env
    implicit none
    integer, intent(out) :: status
    complex(real64), intent(inout) :: z(:)

    status = DftiComputeForward(handle,z)
  end subroutine fft_forward
```

This is the call to a routine for the inverse FFT.

```
  function fft_backward(z, status) result(s)
    use iso_fortran_env
    implicit none
    integer, intent(out) :: status
```

```
    complex(real64), intent(inout) :: z(:)
    real(real64), dimension(size(z)) ::  s

    status = DftiComputeBackward(handle,z)
    s = real(z)/(nx*ny)
  end function fft_backward
```

Finally, this is the cleanup call after the FFT's are complete. NOTE: In previous versions of this library, the FFT routines leaked memory. I don't know if that has been fixed in this version.

```
  subroutine fft_cleanup
    implicit none
    integer :: status

    status = DftiFreeDescriptor(handle)
  end subroutine fft_cleanup
```

5. Plot Routines

I used the plotting routines we use in the Helmholtz program to plot the FFT results. They were based on piping calls to the gplot program. While the gplot program is adequate for 1d plots, it wasn't very good for surface plots. I replaced that package with driver's based onthe plplot library which has higher quality graphics routines. The simple_plot module declares a plot class which handles a lot of the details of the library. To use the library, first add the public module symbols with a use statement.

```
  use simple_plot
```

Next declare a variable of type plot.

```
  type(plot) plt
```

In the executable code, create a plot object of this type. There are two possible constructors. The default constructor draws the plots in a graphics window.

```
  plt = plot()
```

To plot the graphics in a PNG file, specify the output file name in the constructor.

```
  plt = plot("osurf.png")
```

To plot a surface, call the splot method in the class with three arguments, the array of x coordinates, the array of y coordinates, and the surface height.

```
  use iso_fortran_env, only: real64
  integer, parameter :: NPTS=128
  real(real64)::y(NPTS,NPTS), xp(NPTS), yp(NPTS)
  ...
  call plt%splot(xp,yp,y)
```

You can call splot multiple times before calling the done method. If you are plotting to a window, the window will display the plots in sequence when you right-click on the window. Right-clicking on the window while displaying the final plot will exit the window and return control to the calling program. If you are plotting to a file, the program will create a sequence of graphics files with each plot with trailing numbers to distinguish them. For example, if you specified an output filename of "osurf.png", as above, the output files would be "osurf.png.1", "osurf.png.2", and so forth. When you are finished plotting, you need to call the done routine to clean up the storage, finish the plotting, and close open files.

```
call plt%done
```

6. Interpolation

We use the pspline library to create bicubic splines for interpolation. The EZSpline Fortran90 interface is the easiest to use.

```
use EZspline_obj
use EZspline
```

To create a double precision bicubic spline, create a tyep to store it as follows.

```
type(EZspline2_r8) :: spl ! 2-D real*8 interpolation object
```

Since we have a periodic, FFT generated, surface. Periodic boundary conditions are a good choice. In the example program, these are stored in the two arrays, bcs1 and bcs2. The first array has the boundary conditions for the x axis and the second for the y axis.

```
integer:: bcs1(2), bcs2(2), ier, i, j
...
! Set periodic boundary conditions
bcs1 = -1
bcs2 = -1
```

Next, call the initialization routine to allocate storage and initialize variables.

```
call EZspline_init(spl, nx, ny, bcs1, bcs2, ier)
call EZspline_error(ier)
```

The ier variable is returned as 0 if there is no error. Otherwise it has an error code. If there is an error code, EZspline_error prints an error message. After filling the f array with the data to be interpolated, a call to EZspline_setup prepares the spline for future interpolations.

```
call EZspline_setup(spl, f, ier)
call EZspline_error(ier)
```

There are three kinds of interpolation possible:

- using individual points (point interpolation)

- using an array of randomly spaced points (cloud interpolation)
- using an array of equally spaced points on a grid (array interpolation)

I think the cloud interpolation is the most helpful for us.

```
real(real64)::zx(2),zy(2),fz(2)
...
! cloud interpolation
zx(1)=1.1d0
zx(2)=1.3d0
zy(1)=0.8d0
zy(2)=0.9d0
call EZspline_interp(spl, 2, zx, zy, fz, ier)
call EZspline_error(ier)
```

There are also routines for computing the derivative. The interface for cloud interpolation is as follows.

```
call EZspline_derivative(spl, 1, 0, 2, zx, zy, dpx, ier)
call EZspline_error(ier)
call EZspline_derivative(spl, 0, 1, 2, zx, zy, dpy, ier)
call EZspline_error(ier)
call EZspline_gradient(spl, 2, zx, zy, gp, ier)
call EZspline_error(ier)
```

The first call to EZspline_derivate computes the partial derivative with respect to x (the first coordinate). The second call to EZspline_derivate computes $\partial/\partial y$. The final call computes the gradient. The first index of the gradient gp is the point number in the cloud. The second index is 1 for $\partial/\partial x$ and 2 for $\partial/\partial y$.

7. CMake File

In order to link to the MKL library, you'll need something like the following in your CMakeLists.txt file. First you need to specify the directories where the libraries are found.

```
# Intel libraries on nystrom
set(MKL_ROOT /opt/intel/mkl)
set(LAPACK_LIB ${MKL_ROOT}/lib/intel64_lin)
link_directories(${LAPACK_LIB} /usr/local/lib/intel64)
```

Next you need to specify where to find the include files.

```
set(MKL_INCLUDE /opt/intel/compilers_and_libraries_2017.1.132/linux/mkl/includ
include_directories(${MKL_INCLUDE})
```

Finally, you need to specify the appropriate libraries to link the program. If you are using slatec in your program, be sure to list the MKL libraries before the slatec libraries.

Otherwise it will use the slower slatec routines that have the same names as the MKL routines.

```
add_executable(surface surface.f95 gplotc.c gplot.f90)
target_link_libraries(surface mkl_gf_lp64 mkl_sequential
  mkl_core pthread m dl)
```

The libraries pthread, m, and dl are actually system libraries that are needed by the MKL library.

A. CMakeLists.txt

```
# Specify the minimum version for CMake
# It might work with earlier ones, but specify my current
# version for now.
cmake_minimum_required(VERSION 2.8.12)

# Project's name
project(fft)

set(PSPLINE "/usr/local")
set(PSPLINE_LIB ${PSPLINE}/lib)
set(PSPLINE_MOD ${PSPLINE}/mod)

set(CMAKE_BINARY_DIR ${CMAKE_SOURCE_DIR}/bin)
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR})
set(LIBRARY_OUTPUT_PATH ${CMAKE_BINARY_DIR})
set(PROJECT_BINARY_DIR ${PROJECT_SOURCE_DIR}/bin)
set(MODULE_DIRECTORY ${CMAKE_BINARY_DIR}/mod)
set(CMAKE_Fortran_MODULE_DIRECTORY ${MODULE_DIRECTORY})
include_directories(${MODULE_DIRECTORY} ${PSPLINE_MOD})
link_directories(${PSPLINE_LIB})
set(SPLINE_LIBS pspline)

set(PLPLOT_MOD "/usr/lib/x86_64-linux-gnu/fortran/modules/plplot")
if(EXISTS ${PLPLOT_MOD})
  include_directories(${PLPLOT_MOD})
endif(EXISTS ${PLPLOT_MOD})

set(MKL_ROOT /opt/intel/mkl)
set(LAPACK_LIB ${MKL_ROOT}/lib/intel64_lin)
link_directories(${LAPACK_LIB} /usr/local/lib/intel64)
set(MKL_INCLUDE /opt/intel/compilers_and_libraries_2017.1.132/linux/mkl/includ
include_directories(${MKL_INCLUDE})
```

```
set(MKL_MOD ${MKL_INCLUDE}/intel64/lp64)
include_directories(${MKL_MOD})

enable_language(Fortran)
set(CMAKE_Fortran_FLAGS "-g -Wall" ${CMAKE_Fortran_FLAGS})

add_subdirectory(doc)

set(LIB_SOURCES plot_template.f95 gplotc.c gplot.f90 fft_surface.f95 surface.f
  normal.f95 simple_plot.f95)
add_library(fftlib ${LIB_SOURCES})

set(MKL_ROOT $ENV{MKLROOT})
if(EXISTS ${MKL_ROOT}/interfaces)
  # This is marylou
  message("This is Marylou")
  set(LAPACK_LIB ${MKL_ROOT}/lib/intel64)
  set(LAPACK_INC ${MKL_ROOT}/include/intel64/lp64)
  link_directories(${LAPACK_LIB})
  include_directories(${LAPACK_INC})
  set(FFT_LIBS fftlib mkl_intel_lp64 mkl_core mkl_intel_thread pthread m iomp5
else(EXISTS ${MKL_ROOT}/interfaces)
  # Not marylou, assume ubuntu
  message("Not Marylou, assuming ubuntu")
  set(FFT_LIBS fftlib mkl_gf_lp64 mkl_sequential
    mkl_core pthread dl m)
endif(EXISTS ${MKL_ROOT}/interfaces)
set(PLOT_LIBS plplotf95d plplotf95cd)

add_executable(fft_example fft_example.f95)
target_link_libraries(fft_example ${FFT_LIBS} ${PLOT_LIBS})

add_executable(twod_example twod_example.f95)
target_link_libraries(twod_example ${FFT_LIBS})

add_executable(surfplot surfplot.f95)
target_link_libraries(surfplot fftlib)

add_executable(deriv deriv.f95)
target_link_libraries(deriv ${FFT_LIBS})

add_executable(surface_test surface_test.f95)
target_link_libraries(surface_test ${FFT_LIBS})
```

```
add_executable(pspline_test pspline_test.f95)
target_link_libraries(pspline_test ${SPLINE_LIBS})

add_executable(reshapex reshapex.f95)

add_executable(tplt tplt.f95)
target_link_libraries(tplt ${PLOT_LIBS})

add_executable(tplt3d tplt3d.f95)
target_link_libraries(tplt3d ${PLOT_LIBS})
```

B. simple_plot.f90

```
module simple_plot
  use plplot
  use iso_fortran_env, only: real64
  implicit none
  private
  type plot
     private
   contains
     procedure :: splot
     procedure :: done
  end type plot
  public splot, done, plot
  ! Constructors
  interface plot
     module procedure :: init
     module procedure :: init_file
  end interface plot

contains

  function init()
    type(plot):: init
    ! Select qtwidget output, otherwise it will prompt
    call plsdev("qtwidget") ! window on workstation; could be hardcopy file
    call plinit()
    call cmap1_init(0) ! color instead of gray
  end function init

  function init_file(filename)
    type(plot) :: init_file
    character(*) :: filename
```

```
  call plsfam(1,1,16000000)
  call plsdev("pngqt") ! use png graphics
  call plsfnam(trim(filename))
  call plinit()
  call cmap1_init(0) ! color instead of gray
end function init_file

subroutine splot(this,x,y,z)
  class(plot) :: this
  real(real64), intent(in)::x(:),y(:),z(:,:)
  real(real64), parameter :: basex=1.d0, basey=1.d0, height=1.d0
  real(real64) :: xmin, xmax, ymin, ymax,zmin, zmax
  real(real64), parameter :: altitude=30.d0, azimuth=30.d0
  integer, parameter :: nlevel = 10
  real(real64) :: cmin, cmax, cstep, clevel(nlevel)
  integer :: i

! bounds check
  if(size(x) /= size(z,1)) call error("X dimension mismatch in splot")
  if(size(y) /= size(z,2)) call error("Y dimension mismatch in splot")

  ! contour levels for surface plot
  cmin = minval(z)
  cmax = maxval(z)
  cstep = (zmax-zmin)/(nlevel+1)
  clevel = cmin + cstep * (/ (i, i=1, nlevel) /)

  call pladv(0) ! advance to next subframe (window in plot)
  call plcol0(1) ! color selection for axes and labels
  call plvpor(0.0d0, 1.0d0, 0.0d0, 0.9d0) ! viewport
  call plwind(-1.0d0, 1.0d0, -1.0d0, 1.5d0) ! window
  ! bounds
  xmin = minval(x)
  xmax = maxval(x)
  ymin = minval(y)
  ymax = maxval(y)
  zmin = cmin
  zmax = cmax
  call plw3d(basex, basey, height, xmin, xmax, ymin, ymax, zmin, zmax,&
       altitude, azimuth)
  call plbox3('bnstu', 'x axis', 0.0d0, 0, &
       'bnstu', 'y axis', 0.0d0, 0, &
       'bcdmnstuv', 'z axis', 0.0d0, 0) ! axis labels
  call plcol0(2) ! color for surface
```

```
  call plbox3('bnstu', 'x axis', 0.0d0, 0, &
       'bnstu', 'y axis', 0.0d0, 0, &
       'bcdmnstuv', 'z axis', 0.0d0, 0) ! axis labels
  call plcol0(2) ! color for surface
  ! possible options: MAG_COLOR (contour colors), FACETED,
  ! BASE_CONT (contour at base),, SURF_CONT (contour on surface),
  ! DRAW_SIDES. You can add these instead of doing an ior
  call plsurf3d(x,y,z,ior(MAG_COLOR, FACETED),clevel) ! clevel are contour l
end subroutine splot

subroutine done(this)
  class(plot) :: this
  call plend
end subroutine done

subroutine error(msg)
 character(*), intent(in) :: msg
  write(*,*) msg
  stop
end subroutine error

subroutine cmap1_init(gray)
  !    For gray.eq.1, basic grayscale variation from half-dark
  !    to light.  Otherwise, hue variations around the front of the
  !    colour wheel from blue to green to red with constant lightness
  !    and saturation.

  integer, intent(in) ::  gray
  real(real64) i(0:1), h(0:1), l(0:1), s(0:1)

  !   left boundary
  i(0) = 0._plflt
  !   right boundary
  i(1) = 1._plflt
  if (gray == 1) then
     !       hue -- low: red (arbitrary if s=0)
     h(0) = 0.0_plflt
     !       hue -- high: red (arbitrary if s=0)
     h(1) = 0.0_plflt
     !       lightness -- low: half-dark
     l(0) = 0.5_plflt
     !       lightness -- high: light
     l(1) = 1.0_plflt
     !       minimum saturation
```

```
        s(0) = 0.0_plflt
        !       minimum saturation
        s(1) = 0.0_plflt
    else
        !       This combination of hues ranges from blue to cyan to green to y
        !       to red (front of colour wheel) with constant lightness = 0.6
        !       and saturation = 0.8.

        !       hue -- low: blue
        h(0) = 240._plflt
        !       hue -- high: red
        h(1) = 0.0_plflt
        !       lightness -- low:
        l(0) = 0.6_plflt
        !       lightness -- high:
        l(1) = 0.6_plflt
        !       saturation
        s(0) = 0.8_plflt
        !       minimum saturation
        s(1) = 0.8_plflt
    endif
    call plscmap1n(256)
    call plscmap1l(.false., i, h, l, s)
  end subroutine cmap1_init

end module simple_plot
```

C. fft_surface.f95

```
module fft_surface
  use iso_fortran_env
  use MKL_DFTI
  implicit none
  type(DFTI_DESCRIPTOR), POINTER :: handle
  integer :: stat
  integer :: nx,ny

  private
  public create_surface

contains

  subroutine create_surface(sigma, rms, s)
    use iso_fortran_env, only: real64
```

```
  implicit none
  real(real64), intent(out) :: s(:,:)
  real(real64), intent(in) :: sigma, rms
  complex(real64), allocatable :: z(:)
  real(real64), allocatable :: s1(:)
  integer :: shape1(1)

  nx = size(s,1)
  ny = size(s,2)
  allocate(z(nx*ny),s1(nx*ny))
  call fft_init(stat)
  call random_surface(s)
  shape1(1)=nx*ny
  z = RESHAPE(s,shape1)-0.5d0
  ! write(*,*)"original surface z(1:4)=",real(z(1:4))
  call fft_forward(z, stat)
  ! write(*,*)"FFT of surface z(1:4)=",z(1:4)
  call filter(z, sigma)
  ! write(*,*)"filtered surface z(1:4)=",z(1:4)
  s1 = fft_backward(z, stat)
  ! write(*,*)"reversed transform s1(1:4)=",s1(1:4)
  s = RESHAPE(s1,shape(s))
  ! write(*,*)"reversed transform s(1:4,1)=",s(1:4,1)
  deallocate(z,s1)
  call setrms(s, rms)
  call fft_cleanup
end subroutine create_surface

subroutine setrms(s, sigma)
  use iso_fortran_env
  implicit none
  real(real64), intent(inout) :: s(:,:)
  real(real64), intent(in) :: sigma
  real(real64) :: sm, smsq, rms

  sm = sum(s)/(nx*ny)
  smsq = sum(s**2)/(nx*ny)
  rms = sqrt(smsq-sm**2)
  s = s*sigma/rms
end subroutine setrms

subroutine filter(z,sigma)
  use iso_fortran_env
  implicit none
```

```
  complex(real64), intent(inout) :: z(nx,ny)
  real(real64), intent(in) :: sigma
  integer :: i,j
  real(real64)::kx,ky,ksq,zf

  ! Assume even number of points
  do i=1,nx
     if(i <= nx/2+1)then
        kx=i-1
     else
        kx=i-nx-1
     end if
     ! if j = 1, do nothing, we are multiplying by 1
     do j=1,ny
        if(j <= ny/2+1) then
           ky = j-1
        else
           ky = j-ny-1
        end if
        ksq=kx**2+ky**2
        zf = exp(-ksq/(2*sigma**2))
        z(i,j)=z(i,j)*zf
     end do
  end do
end subroutine filter

subroutine fft_init(status)
  use MKL_DFTI
  use iso_fortran_env
  integer, intent(out) :: status
  integer :: l(2)

  ! Initialize data descriptors
  l(1)=nx
  l(2)=ny
  status = DftiCreateDescriptor(handle, DFTI_DOUBLE,&
       DFTI_COMPLEX, 2, l)
  status = DftiCommitDescriptor(handle)
end subroutine fft_init

subroutine fft_cleanup
  implicit none
  integer :: status
```

```
    status = DftiFreeDescriptor(handle)
  end subroutine fft_cleanup

  function fft_backward(z, status) result(s)
    use iso_fortran_env
    implicit none
    integer, intent(out) :: status
    complex(real64), intent(inout) :: z(:)
    real(real64), dimension(size(z)) ::  s

    status = DftiComputeBackward(handle,z)
    s = real(z)/(nx*ny)
  end function fft_backward

  subroutine fft_forward(z, status)
    use iso_fortran_env
    implicit none
    integer, intent(out) :: status
    complex(real64), intent(inout) :: z(:)

    status = DftiComputeForward(handle,z)
  end subroutine fft_forward

  subroutine random_surface(s)
    use normal
    real(real64), intent(out) :: s(:,:)
    integer :: nx, ny, i, j

    nx=size(s,1)
    ny=size(s,2)
    do i=1,nx
       do j=1,ny
          s(i,j)=nrand()
       end do
    end do
  end subroutine random_surface
end module fft_surface
```

D. fft_example.f95

```
program fft_example
  use fft_surface
  use iso_fortran_env, only: real64
  use simple_plot
```

```
  implicit none
  integer, parameter :: NPTS=128
  real(real64)::y(NPTS,NPTS), xp(NPTS), yp(NPTS)
  real(real64), parameter:: sigma=NPTS/6d0, rms=0.1
  real(real64), parameter:: dx=4.0/NPTS, dy=4.0/NPTS
  integer :: i
  type(plot) :: plt

  write(*,*)'2d FFT Example Program'
  call create_surface(sigma*1000, rms, y)
  ! plt = plot('osurf.png')
  plt = plot()
  xp = (/ (dx*i, i=0,NPTS-1) /)
  yp = (/ (dy*i, i=0,NPTS-1) /)
  call plt%splot(xp,yp,y)
  call create_surface(sigma/2, rms, y)
  call plt%splot(xp,yp,y)
  call plt%done()
end program fft_example
```