RLD C++ Code Tutorial

Contents

C++ Library

C++ Files

There are eight ".cpp" files that are included when building the library. A brief description of each is provided:

File Name	Purpose
RLD.cpp	Main logic loop of Richardson-lucy reconstruction with either single or double precision
SMART.cpp	Main logic to perform SMART reconstruction with either single or double precision. Normalization is slightly modified from original form
SART.cpp	Structure to perform SART reconstruction. Normalization factors not implemented so currently unstable (not working)
Project.cpp	Projection functions to perform forward and backward projections
PlanCreate.cpp	Low level FFT plan creations to perform forward and backward projections repeatedly
IterOps.cpp	Low level array operations (e.g. summing array over axis) used in the
IterOpsSIMD.cpp	Same as above, but SIMD instructions were used for speedup (saw little to no effect so this is currently unused)
PSF.cpp	Convenience functions to transform PSF to frequency domain

FFTW

The basis of the reconstruction requires the computation of a large number of fast Fourier transforms (FFTs). To perform these I used the FFTW library: https://www.fftw.org/

To compile and build the RLD library requires that this library be installed.

Compiling Library

Compiling the library requires a C++ compiler. I used g++. This part is magic to me and I always forget how to do it. I program in vscode and it uses a tasks.json file to hold all the commands that need to be passed to the compiler. I included this as the Build.txt file. However, it will require some modification to use, here's an example of the command that builds the library:

g++.exe -shared -fPIC -fdiagnostics-color=always -g RLD.cpp SMART.cpp SART.cpp Project.cpp PlanCreate.cpp IterOps.cpp IterOpsSIMD.cpp PSF.cpp -o OUTPUTPATH/Recon.dll -I(LIBRARYPATH)/fftw-3.3.5-dll64 -L(LIBRARYPATH)/fftw-3.3.5-dll64 -lfftw3-3 -lfftw3f-3 -O3 -mavx2 -fopenmp

Here are the individual parts of that explained

g++.exe	Compiler
-shared	Tells compiler to create shared library not executable
-fPIC	Command required for library to work
-fdiagnostics-color=always	Error message formatting option
-g	Controls the amount of debugging information
RLD.cpp SMART.cpp SART.cpp Project.cpp PlanCreate.cpp IterOps.cpp PSF.cpp	Included files
-o {OUTPUTPATH}/Recon.dll	Tells the compiler the path and name of compiled library I believe the file extension should be '.so' for linux or MAC
-I{LIBRARYPATH}/fftw-3.3.5-dll64	Includes the path of the installed FFTW library
-L{LIBRARYPATH}/fftw-3.3.5-dll64	Includes the FFTW library
-lfftw3-3 -lfftw3f-3	Loads the FFTW3 library (double and single precision)
-O3	Tells the compiler how aggressively to optimize the code
-mavx2 -fopenmp	Includes libraries for parallelizing CPU operations (These should be included with g++)

The final library should have the name Recon.dll or Recon.so. The python code expects these files to be in the same directory as where the python code is run. These paths could be manually specified instead. The following code snippet where the libraries are loaded are found at the top of the Bindings_{Alg}.py files:

```
# Load the shared library
if os.name == "nt":  # Windows
    dll_name = "Recon.dll"
    dllabspath = os.path.dirname(os.path.abspath(__file__)) + os.path.sep + dll_name
    SMART_lib = ctypes.CDLL(dllabspath, winmode=0)
elif os.name == "posix":  # Linux / macOS
    SMART_lib = ctypes.CDLL("./Recon.so")  # Use .dylib for macOS
#
```

Reconstructions from Python Jupyter Notebook

Reconstruction Settings

Precision

The following cell allows the choice of algorithm and changes whether every floating point number will be 32 or 64 bit. Simply change the precision variable to be a string that is “single” or “double”.

```
Precision = "single" # either 'single' or 'double'

DataType = dtype_choice[Precision]
RLD = rld_choice[Precision]
```

Image and Range

The following options provide reconstruction options

```
# If the central PSF is not centered use these offsets
# They are the offset between the center of the image and the center of the central view

VerticalOffset = -50 # [pixels], Positive is up ^
HorizontalOffset = 50 # [pixels], Positive is right ->

# Assuming Circular Elemental View
# This is the radius of the elemental image in pixels
EI_radius = 220 # [pixels]

# Depth Range
z_min = -2000 # [um]
z_max = 500 # [um]

# Step Size
z_step = 20 # [um]

# Number of Iters
ITERS = 50
```

```
FLAG = "LABEL"
```

Offset (Vertical and Horizontal)

These act as a correction if the centers of your PSF and the centers of your elemental images have an offset.

EI_radius

The radius in pixels of the elemental images

z_min, z_max, z_step

These are my parameters to control which depth planes get reconstructed. Used to generate an array of **zPlanes**

ITERS

Controls how many iterations of RLD/SMART/SART to do

FLAG

This parameter is just a string that will be appended to any outputs, you can use it to label reconstructions with different parameters

There's an additional function defined called zToIndex(z) which I used in some instances where I need to get the index (integer value) of a z depth.

Load PSF Stack

This section loads the experimental or synthetic PSF images. It starts by defining a list of file paths to individual PSF images for each depth plane to reconstruct **PSF_filepaths**. Currently it is setup so that all PSF images are assumed to be contained in one folder {PATH_TO_PSF_FILES}, and follow a file format where the name of each is '{z_depth}.tif'. This can be easily modified to any file naming convention.

```
# Generate PSF file paths
PSF_filepaths = [f"PATH_TO_PSF_FILES/{z:.0f}.tif" for z in zPlanes]
```

```
# ReadPSF Stack, all PSFs put in a single 3D array
# Optional Argument 'clip' sets any intensity below (clip*max_value) to zero
# Found to be important for experimental images with noise
PSFstack, (L, M, N) = ReadPSFstack(PSF_filepaths, dtype=DataType, shift=True, clip=0.05)
# if the PSFs are centered (They should be) the shift Parameter performs an FFT shift on them
# they become ucentered so that the reconstruction is

# L = NUMBER OF Z PLANES
# M = NUMBER OF ROWS
# N = NUMBER OF COLUMNS

print(f"Stack Shape: ({L} x {M} x {N})")
```

The list **PSF_filepaths** is passed to **ReadPSFstack** which loads all the PSF images and puts them in a 3D array. This function simultaneously performs an image normalization so that each depth plane has a summed intensity of one. For experimental images, intensity from noise should not be included in the normalization. Therefore, the function has an optional argument **clip**, which sets all pixels with an intensity below (**clip** * maximum image intensity) to zero intensity.

Load Image

Here the image to reconstruct a 3d volume from is loaded by specifying its filepath, it includes the optional argument **Invert** to invert the image.

```
RawImageFileName = "PATH_TO_IMAGE_TO_RECONSTRUCT"
IMG = ReadRawImage(RawImageFileName, DataType, invert=True)
```

ge

Blank Volume

The following code initializes a 3d array to store the reconstructed volume. It has an important subtlety that was *missing* in some Matlab implementations. The lateral dimensions of the array are determined by the image size, but the lateral extent of the volume is determined by the size of a single elemental image. The intensity of voxels outside of this lateral extent should *always* be zero.

```python
def FillVol(L, M, N, EmptyVol, Radius, OffsetH = 0, OffsetV=0):

    # EmptyVol = np.zeros((self.nz, self.ny, self.nx), dtype=dtype)
    EmptyVol[:, :, :] = 1.0

    xpx = np.arange(0, N) - N//2 - OffsetH
    ypx = np.arange(0, M) - M//2 + OffsetV

    Xpx, Ypx = np.meshgrid(xpx, ypx)

    # Masks the space to a region around the center

    # ValidSpace = (Xpx**2 + Ypx**2) < (Radius**2) # Circular region
    ValidSpace = (np.abs(Xpx) < Radius) * (np.abs(Ypx) < Radius) # Square region
    # ValidSpace = (np.abs(Xpx) < 3/2*Radius)*(np.abs(Ypx) < Radius) # Rectangular region

    for i in range(L):
        EmptyVol[i, :, :] *= (ValidSpace) #.astype(dtype))
```

The ValidSpace array defines a volume mask using the predefined **EI_radius** parameter. It can define either a cylindrical region to reconstruct using this parameter (the commented line), or a square (uncommented) or rectangular (commented) region to reconstruct.

Image Mask

It's important to mask the regions outside of the elemental images that can be reconstructed for numerical stability. The following code generates this mask using the masked volume. This operation is slow because it's performing volume projections in pure python. If a mask has previously been saved it tries to load that one:

```python
Mask = None

if 'Mask' in locals():
    Mask = CreateMask(VOL, PSFstack)
    Mask = Mask > (2**-15) #0.01*Mask.max()
    imwrite(f"{'.'.join(PSF_filepaths[0].split('.')[:-1])}_MASK_{FLAG}.png", Mask.astype(np.uint8))
elif Mask.split('.')[-1] == ".npy":
    Mask = np.load(Mask)
else:
    Mask = ReadRawImage(Mask, DataType)
```

Reconstruction

The following code performs (and times) the reconstruction:

```
t0 = perf_counter()
RLD(VOL, PSFstack, IMG, ITERS)
tf = perf_counter()
print(f"RLD w/ C++ Time: {tf-t0} sec")
```

The following code checks the reconstruction to see if it was successful:

```
# If any one of these conditions are met the RLD output is invalid
# Try reurunning all the cells from PSFstack creation down
# When a new volume size is run for the first time it usually fails
# I sort of know why this is, but in a more real way I don't

if VOL.min() < -1e-16:
    raise ValueError(f"RLD output is negative, minimum = {VOL.min()} at {VOL.argmin()}")
if VOL.max() == 0:
    if PSFstack.max() == 0:
        raise ValueError("RLD output and PSF are all zero")
    raise ValueError("RLD output all zeros")
if np.isnan(VOL).any():
    raise ValueError("RLD output has NaNs")
```

Every time I do a reconstruction with a new volume size the reconstruction fails, subsequent runs work though. Internally the FFTW code is tracks what it does for different volume sizes and stores this in a “fftw_wisdom.dat” file.

Output:

```
bit_depth = 16

# Normalize Intensity to make use of the bit depth
VOL_crop /= VOL_crop.max()
VOL_crop *= (2**(16) - 1)

RawImgPathFolder = f"{'.'.join(RawImageFileName.split('.')[:-1])}"

try: mkdir(RawImgPathFolder)
except FileExistsError: pass

try: mkdir(f"{RawImgPathFolder}/zStack_{RunTime}_{FLAG}")
except FileExistsError: pass

for k in range(L):
    Slice = VOL_crop[k, :, :].astype(np.uint16)

    zOffset = zPlanes[0]

    imwrite(f"{RawImgPathFolder}/zStack_{RunTime}_{FLAG}/z{k:03d}.tif", Slice)

np.savez(f"{RawImgPathFolder}/log.npz", Z=zPlanes, VerticalOffset=VerticalOffset, HorizontalOffset=HorizontalOffset,
         z_min = z_min, z_max = z_max, z_step=z_step, ITERS = ITERS, EI_radius=EI_radius)
```

The following outputs the volume as a series of images. The images are saved in a folder created in the directory where the original loaded image was. The reconstruction is labelled by the time that the code finished running.

Batch Reconstructions

I used the file RLD_Batch.py to run a series of reconstructions sequentially. It follows the exact same setup as the jupyter notebook, just configured to run on a list of images instead of one.